Analysing blockchains and smart contracts: tools and techniques

Thesis · March 2018

DOI: 10.13140/RG.2.2.19734.04161

1 author:

Livio Pompianu
Università degli studi di Cagliari
17 PUBLICATIONS **136** CITATIONS

SEE PROFILE

Università degli Studi di Cagliari

Dipartimento di Matematica e Informatica
Dottorato di Ricerca in Matematica e Informatica
Ciclo XXX

Ph.D. Thesis

Analysing blockchains and smart contracts: tools and techniques

S.S.D. INF/01

Candidate
Livio Pompianu

Supervisors
Prof. Maurizio Atzori,
Prof. Massimo Bartoletti

PhD Coordinator
Prof. Giuseppe Rodriguez

March 2018
Final examination academic year 2016/2017

Abstract

Modern cryptocurrencies exploit decentralised blockchains to record a public and unalterable history of transactions. Besides transactions, further *metadata* is stored for different, and often undisclosed, purposes. Metadata is mostly generated by *protocols* and *smart contracts*, i.e. programs whose correct execution is automatically enforced without relying on a trusted authority. This work investigates tools and techniques for analysing blockchains, their metadata, and smart contracts, focussing on Bitcoin and Ethereum. The main contributions are:

- a survey of the techniques for embedding metadata in the Bitcoin blockchain, comparing them, pointing out pros and cons;
- a study of the protocols embedding metadata in Bitcoin, classifying them by their application domain;
- an analysis of the metadata stored in the Bitcoin blockchain over the years, measuring its evolution in time, the space consumption, and the distribution of metadata by type, by embedding technique, and by application domain;
- an analysis of the smart contracts deployed in Ethereum, quantifying their usage in relation to their application domain, and identifying the most common programming patterns;
- a comparison of several platforms for executing smart contracts;
- a development and evaluation of a general-purpose framework, seamlessly supporting data analytics on both Bitcoin and Ethereum, allowing users to integrate relevant blockchain data with information from other sources.

Acknowledgments

This work is partially supported by Aut. Reg. of Sardinia P.I.A. 2013 “NOMAD”, and EU COST Action IC1406 cHiPSET.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Motivations

The last few years have witnessed an increasing interest in *Bitcoin* [109], and *blockchains*. Bitcoin is a *cryptocurrency* leaning on the blockchain technology. Although the original and most widespread application of blockchain technologies are cryptocurrencies, recently further use cases have been proposed [72]. In this thesis we explore modern blockchain applications with a focus on *metadata* and *smart contracts*.

A blockchain is an *immutable* and continuously growing list of records, called *blocks*, which are linked and secured using cryptography. Modern cryptocurrencies store the full list of money transfers in a blockchain. In such a system, when a user wants to make a payment, she produces a *transaction* with the required data (e.g. the amount of currency to send and the receiver) and publish it. Subsequently, the transaction is included in a new block that will be appended to the blockchain. While a single entity is in charge of collecting transactions and updating the blockchain of centralised currencies, *decentralised* cryptocurrencies are typically managed by a peer-to-peer network collectively adhering to a protocol for validating new blocks. In this way the system eliminates the risks that come with centrally hold data. Bitcoin is the first and most widespread decentralised cryptocurrency. The *Ethereum* [75] platform instead is attracting attention for its native support to the execution of smart contracts, i.e. programs executed in a decentralised fashion.

The rising of blockchain platforms. Besides Bitcoin and Ethereum, a remarkable number of alternative platforms have flourished over the last few years, either implementing cryptocurrencies or some forms of smart contracts [77, 98, 40, 12, 31].

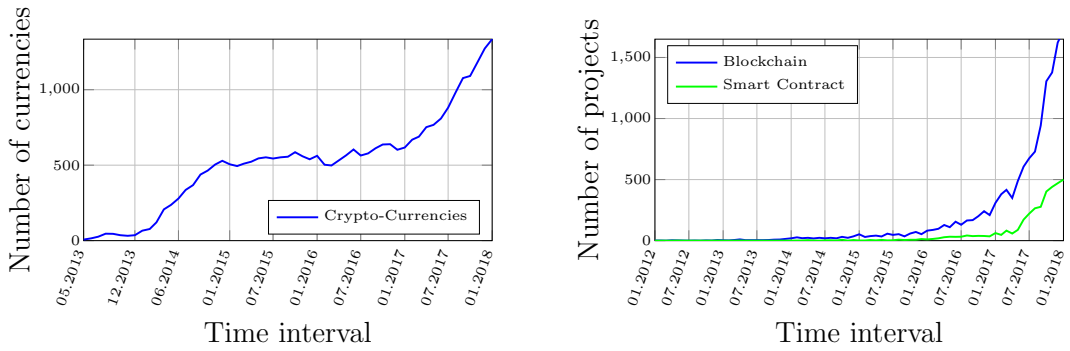


Figure 1.1: Projects related to crypto-Currencies and smart contracts.

For instance, we measure that the number of crypto-currencies hosted on coinmarketcap.com has increased from 0 to more than 1300 since 2012 (see Figure 1.1, left chart); the number of github projects related to blockchains and smart contracts has reached, respectively, 2,715 and 445 units (see Figure 1.1, right chart). Deloitte measures [25] over 86,000 blockchain initiatives. Their report also inspects the geographical areas in which blockchain projects have been developed. The major area is San Francisco (1,279 users and 101 organizations), followed by London, and New York. In the meanwhile, ICT companies, banks (e.g. European Central Bank [82]), and several national governments (e.g. UK [125], Japan [110], EU [83]) have started dealing with these topics, also with significant investments. Finally, several scientific communities (e.g. ITU-T, ISITC Europe, W3C Blockchain Community Group), started studying these topics.

Blockchains beyond currency: metadata and smart contracts. Although the main goal of Bitcoin is to transfer digital currency, the immutability and openness of its blockchain have inspired the development of new *protocols*, which "piggyback" metadata on transactions in order to implement a variety of applications beyond cryptocurrency. For instance, some protocols allow to certify the existence of a document (e.g., [43, 26, 37]), while some others allow to track the ownership of a digital or a physical asset (e.g., [34, 17, 33]). Furthermore, the public and append-only ledger of transaction (the blockchain) and the decentralized consensus protocol that Bitcoin nodes use to extend it, have revived Nick Szabo's idea of smart contracts. The archetypal implementation of smart contracts is Ethereum [75], a platform where they are rendered in a Turing-complete language. The consensus

protocol of Ethereum ensures that all and only the valid updates to the contract states are recorded on the blockchain, so ensuring their correct execution.

Issues of metadata. Despite the growing hype on blockchains and smart contracts, the understanding of the actual benefits of these technologies, and of their trustworthiness and security, has still to be assessed. In the Bitcoin community, a debate about scalability has been taking place over the last few years [1, 38, 39]. In particular, users argue over whether the blockchain should allow for storing spurious data, not inherent to currency transfers. Although many recent works analyse the Bitcoin blockchain [104, 60, 117, 114, 97], as well as some services for embedding metadata [4, 41, 35, 28], many relevant questions are still open. Since Bitcoin was not designed for embedding metadata, what is the impact of these data in the Bitcoin system? What is the size of metadata respect to the total size of the blockchain? Which kinds of blockchain-based applications are exploiting metadata, and how?

Issues of smart contracts. In the Ethereum community, the developers often release the source code of their smart contracts, in order to let users verify the behaviour of the proposed contracts. However, this also exposes code vulnerabilities to malicious users that repeatedly exploit them since the published sources are immutable. The consequences can be fatal, as witnessed by the unfortunate epilogue of the DAO contract [47], a crowdfunding service plundered of $\sim 50M$ USD because of a programming error. Since then, many other vulnerabilities in smart contract have been reported [56, 98, 46, 48, 128]. What are the most common types of smart contracts developed? What are the design patterns used developing smart contracts?

Understanding blockchains by applying analytics. Developing analytics on metadata and smart contracts allows us to obtain several insights, answering the above questions. Specifically, understanding what applications are embedding metadata in the Bitcoin blockchain, could help designers of blockchains to create systems that explicitly support these applications. Understanding how smart contracts are used and how they are implemented provides us valuable information for creating new domain-specific languages (not necessarily Turing complete [74, 76, 84, 113]), which *by-design* avoid vulnerabilities as the ones discussed above. Further, this

knowledge could help to improve analysis techniques for smart contracts (like e.g. the ones in [70, 98]), by targeting contracts with specific programming patterns.

Issues of blockchain analytics. Although these studies often have several common traits, researchers so far tended to implement new ad-hoc tools, rather than reusing standard libraries. Further, most of the few available tools have limitations. The consequence is that the same functionalities have been implemented again and again as new analytics have been developed. Therefore, in this context, we believe that the introduction of an efficient, modular and general-purpose abstraction layer to manage internal and external information is key for blockchain data analytics, along the lines of the software engineering best practices of *reuse*.

1.1 Contributions

In this work we study tools and techniques for analysing blockchains and smart contracts. Our main contributions can be summarised as follows:

- We survey the existing techniques for embedding metadata in the Bitcoin blockchain, identifying 11 distinct ones. We compare these techniques, pointing out their side effects, and we compare their evolution over time, quantifying the amount of metadata embedded through them.

- We search the blockchain for metadata, and we parse them to infer the intended usage. To this purpose, we consider both metadata as single units of information, and as aggregates of pieces scattered through the blockchain (e.g., images). Overall, we recognise 12 different types of metadata, which we group into 5 categories according to their actual content. We quantify the amount and size of metadata by type and category.

- We identify 45 distinct protocols which are used by applications to embed metadata in the blockchain. We classify them according to their application domain, and we measure the amount of metadata they produced. We analyse the correlation between embedding techniques, metadata types and protocols.

- We compare the size of the extracted metadata with the overall size of the blockchain, and we investigate peaks of metadata that occurred over the years.

We distribute the results of our analysis, together with the source code of our tools for extracting and parsing metadata, under an open source license[1].

- We examine the Web for news about smart contracts in the period from June 2013 to September 2016, collecting data about 12 platforms. We choose from them a sample of 6 platforms which are amenable to analytical investigation. We analyse and compare several aspects of the platforms in this sample, mainly concerning their usage, and their support for programming smart contracts. We propose a taxonomy of smart contracts, sorting them into categories which reflect their application domain.

- We collect from the blockchains of Bitcoin and Ethereum a sample of 834 smart contracts, which we classify according to our taxonomy. We then study the usage of smart contracts, measuring the distribution of their transactions by category. This allows us to compare the different usage of Bitcoin and Ethereum as platforms for smart contracts.

- We identify 9 common design patterns, and we quantify their usage in contracts, also in relation to the associated category. Together with the previous point, ours constitutes the first quantitative investigation on the usage and programming of smart contract in Ethereum. We distribute the results of our survey under an open source license[2].

- We develop a framework to create general-purpose analytics on the blockchains of Bitcoin and Ethereum. The design of our tool is based on an exhaustive survey of the literature on the analysis of blockchains. Our tool supports the most commonly used external data, e.g. exchange rates, address tags, protocol identifiers, and can be easily extended by linking the relevant data sources.

- We evaluate the effectiveness of our framework by means of a set of paradigmatic use cases, which we distribute, together with the source code of our library, under an open source license[3]. We exploit our use cases to evaluate the performance of SQL *vs.* NoSQL databases for storing and querying blockchain views.

- We discuss some implementation details of our framework, and we will evaluate its effectiveness, and the choice between SQL or NoSQL. We compare the existing general-purpose blockchain parsers with ours.

We make available online the raw data we have extracted from the blockchains, as well as the tools we have developed for our analyses[1] [2] [3].

1.2 Structure of the work

- In Chapter 2 we give some background on Bitcoin and Ethereum.
- In Chapter 3 we present a methodical survey on Bitcoin metadata, based on the analysis of the first 480,000 blocks (i.e. all the blocks published up to August 2017).

  Part of this material borrows from [65].
- In Chapter 4 we provide a survey on smart contracts, with a focus on Bitcoin and Ethereum — the two most widespread platforms currently supporting them.

  Part of this material borrows from [66].
- In Chapter 5 we show a framework to create general-purpose analytics on the blockchains of Bitcoin and Ethereum.

  Part of this material borrows from [62].
- Chapter 6 contains a summarised view of our work, and proposes some directions for further work.

[1] http://blockchain.unica.it/projects/blockchain-analytics/metadata.html
[2] http://blockchain.unica.it/projects/blockchain-analytics/contracts.html
[3] http://blockchain.unica.it/projects/blockchain-analytics/analytics.html

Chapter 2

Background on Bitcoin and Ethereum

In Section 2.1 we present the main concepts related to Bitcoin. In Section 2.2 we discuss Ethereum.

2.1 Bitcoin

Bitcoin [109] is a decentralized infrastructure to exchange virtual currency — the *bitcoins*. Users interact with Bitcoin through *addresses*, by publishing *transactions* to transfer bitcoins from one address to another. The log of all transactions is recorded on the blockchain, a public and immutable data structure maintained by the nodes of the Bitcoin network. A subset of these nodes, called *miners*, gather the transactions broadcast by users, aggregate them in blocks, and try to append these blocks to the blockchain. A consensus protocol based on moderately-hard "proof-of-work" puzzles is used to resolve conflicts that may happen when different miners concurrently try to extend the blockchain, or when some miner attempts to append a block with invalid transactions. Ideally, the blockchain is globally agreed upon, and free from invalid transactions, unless the adversary controls the majority of the computational power of the network [58, 85, 93]. The security of the consensus protocol relies on the assumption that honest miners are rational, i.e. that it is more convenient for a miner to follow the protocol than to try to attack it. To make this assumption hold, miners receive some economic incentives for performing the (time-consuming) computations required by the protocol. Part of these incentives

is given by the *fees* paid by users upon each transaction.

To illustrate how transfers of bitcoins work, we consider two transactions T_0 and T_1 of the following form:

T_0	T_1
previous transaction: $\cdots$ in-script: $\cdots$	previous transaction: T_0 in-script: $sig_k(\bullet)$
value: v_0 out-script(T, σ): $ver_k(T, \sigma)$	value: v_1 out-script: $\cdots$

The transaction T_0 contains a value of v_0 bitcoins. One can redeem this amount by publishing a transaction (e.g., T_1), whose previous transaction field contains the identifier of T_0 (a hash of the transaction, displayed just as T_0 in the figure) and whose in-script contains values making the out-script[1] of T_0 evaluate to true. When this happens, the value of T_0 is transferred to the new transaction T_1, and T_0 becomes unredeemable. A subsequent transaction can then redeem T_1 likewise. In the transaction T_0 above, the out-script just checks the digital signature σ on the redeeming transaction w.r.t. a given key k. We denote with $ver_k(T, \sigma)$ the signature verification, and with $sig_k(\bullet)$ the signature of the enclosing transaction (T_1 in our example), including all the parts of the transaction but its in-script (obviously, because it contains the signature itself).

Now, assume that T_0 is redeemable on the blockchain when someone tries to append T_1. This is possible if $v_1 \leq v_0$, and the out-script of T_0, applied to T_1 and to the signature $sig_k(\bullet)$, evaluates true. The *Unspent Transaction Output* (in short, UTXO) is the set of redeemable outputs of all transactions included in the blockchain. To be valid, a transaction must only use elements of the UTXO as inputs.

The previous example shows the simple case of transaction with only one input and one output. In general, Bitcoin transactions have the form displayed in Figure 2.1. First, there can be multiple inputs and outputs (denoted with array notation in the figure): in-counter specifies the number of inputs, and out-counter that of outputs. Each input (resp. output) has its own in-script (resp. out-script). The script sizes are given (in-script length and out-script length) in order to simplify parsing operations. Since each output value can be redeemed independently, previous transaction fields must specify which one they are redeeming (previous out-index in the

[1] in-script/out-script are called scriptPubKey/scriptSig, respectively, in the Bitcoin wiki.

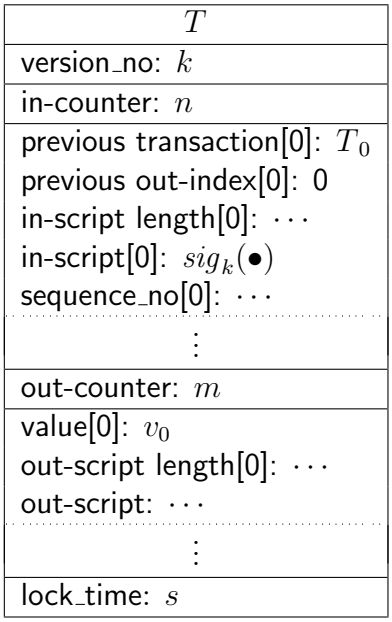


Figure 2.1: A Bitcoin transaction.

figure). A transaction with multiple inputs redeems *all* the (outputs of) transactions in its previous transaction fields, providing a suitable in-script for each of them. The sum of the values of all the inputs must be greater or equal to the sum of the values of all outputs, otherwise, the transaction can not be appended to the blockchain. Furthermore, none of the inputs must have been redeemed yet.

The lock_time field specifies the earliest moment in time when the transaction can appear on the blockchain. The version_no field is currently set to 1. Transaction inputs contain also a 4-bytes field called `sequence_no`. Normally its value is `0xFFFFFFFF`, and it is ignored unless the transaction lock_time is greater than 0 [14].[2]

2.1.1 Scripts

In its general form, a script is a program in a non Turing-complete, stack-based scripting language, which features a limited set of logic, arithmetic, and cryptographic operators. Values are pushed onto the stack by using the PUSH_DATA instructions. The intended purpose of these instructions is to allow the transaction creator to specify arguments usually, addresses and address hashes for use by other

[2]This mechanism was disabled in 2010, and more recently the code has been removed completely, due to concerns over people using it to perform DoS attacks[12].

opcodes (such as cryptographic or conditional operators).

Below we illustrate the main Bitcoin scripts exploited for embedding metadata. For each one, we show the `in-script` and the `out-script`.

`pay-to-PubkeyHash` [14]

```
input  = <sig> <pubKey>
output = OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
```

This method implements the signature verification ver_k seen above (actually, the script does not contain the public key k, but its hash $H(k)$). To make the script evaluate to true, the redeeming transaction T has to provide the signature σ and a public key k such that $H(k) = h$ and $ver_k(T, \sigma)$.

`pay-to-Pubkey` [13]

```
input  = <pubKey> OP_CHECKSIG
output = <sig>
```

The `pay-to-Pubkey` is quite similar to the previous script. The public key is required in place of its hash.

`pay-to-ScriptHash` [14]

```
input  = DATA
output = OP_HASH160 <scriptHash> OP_EQUAL
```

In a `pay-to-ScriptHash`, bitcoins are sent to a script hash instead of a public key hash. Addresses starting with 1 are `pay-to-PubkeyHash`, and addresses starting with 3 are `pay-to-ScriptHash`. In order to spend `pay-to-ScriptHash` bitcoins, the recipient must provide a script matching the script hash and data which makes the script evaluate to true.

`op_return` [13]

```
output = OP_RETURN {zero or more ops}
```

An `out-script` containing the OP_RETURN instruction always evaluates to false, hence the output is provably unspendable, and its transaction can be safely removed from the UTXO. Therefore, there is no correspondent `in-script`, and we only show the `out-script`.

`multi-signature` [14]

```
input  = 0 <sig1> ... <script>
output = OP_m <pubKey1> ... OP_n OP_CHECKMULTISIG
```

M of N Multisig is a pubkey script that provides N number of pubkeys and requires the corresponding signature script to provide M minimum number signatures corresponding to the provided pubkeys.

2.2 Ethereum

Similarly to Bitcoin, Ethereum [75] is a blockchain-based platform providing a cryptocurrency called *ether*. Furthermore, Ethereum is a decentralized computing platform, which executes programs, called *smart contracts*. Contracts are composed of functions, defined by sequences of bytecode instructions, and can transfer *ether* to/from users and to other contracts. Ethereum executes contracts through the Turing-complete *EVM* — the Ethereum Virtual Machine [129]. The blockchain stores *transactions*, allowing user to: (i) create new contracts; (ii) invoke functions of a contract; (iii) transfer ether to contracts or to other users. The sequence of transactions on the blockchain determines the state of each contract, and the balance of each user. Both contracts and users are uniquely identified by *addresses* which are sequences of 160 bits. As in Bitcoin, a network of *miners*, executes all the transactions. Users pay *execution fees* to miners for performing the computations required by the protocol. Moreover, fees prevent from *denial-of-service* attacks where users try to overwhelm the network with time-consuming computations.

Figure 2.2 illustrates an example of smart contract implemented in *Solidity*, a Javascript-like programming language which compiles into EVM bytecode. Notably, Solidity is the only high-level language currently supported by the Ethereum community. The contract implements a personal wallet that can receive ether from users. The owner of the wallet can send ether to other users via the function `pay`. The wallet has also an hashtable `outflow`. This structure tracks all the addresses that received money from the wallet, and associates to each of them the total transferred amount. The total amount of ether available in the contract is recorded in the `balance`, a variable which is automatically updated and cannot be modified by the programmer.

Contracts are composed by fields (which updates are stored permanently in the blockchain), and functions. Users invoke a function by sending a transaction to the network. Although each transaction *must* include the execution fee for the miners, it does not necessarily transfer ether (e.g. it simply updated an internal field of a

```
1  contract Wallet{
2      address owner;
3      mapping (address => uint) public outflow;
4
5      function Wallet(){ owner = msg.sender; }
6
7      function pay(uint amount, address recipient) returns (bool){
8          if (msg.sender != owner ||  msg.value != 0) throw;
9          if (amount > this.balance) return false;
10     outflow[recipient] += amount;
11         if (!recipient.send(amount)) throw;
12         return true;
13     }
14 }
```

Figure 2.2: A simple wallet contract.

contract). The Solidity language also provides exceptions. When an exception is thrown, it cannot be caught: the execution stops, the fee is lost, and all the side effects — including transfers of ether — are reverted.

The function `Wallet` at line 5 is a constructor, run only once when the contract is created. The function `pay` sends `amount` *wei* ($1\,wei = 10^{-18}\,ether$) from the contract to `recipient`. The function invocation carries some extra information: `msg.value` is the amount of ether transferred to the contract, and `msg.sender` is the caller address. At line 8 the contract throws an exception if the caller (`msg.sender`) is not the owner, or if some ether (`msg.value`) is attached to the invocation and transferred to the contract. Since exceptions revert side effects, this ether is returned to the caller (who however loses the fee). At line 9, the call terminates if the required amount of ether is unavailable; in this case, there is no need to revert the state with an exception. At line 10, the contract updates the `outflow` registry, before transferring the ether to the recipient (by using `send` at line 11).

Chapter 3

A journey into Bitcoin metadata

Bitcoin transactions do not contain a specific, dedicated field in which metadata can be directly stored. Nevertheless, over the years Bitcoin users have devised a variety of ways to embed metadata in the blockchain. In some cases, they have disguised their metadata as legit data within transactions fields; in others, they have exploited specific features of the scripting language to inject metadata within scripts. Bitcoin nodes process transactions without the need (and, in many cases, without the possibility) of detecting the metadata embedded. On the other hand, users relying on these metadata usually know how to retrieve them from transaction fields, and how to interpret them. Although the functional behaviour of the Bitcoin network is not affected by metadata, the techniques used to embed them in transactions may have several side effects on the non-functional behaviour of the network.

In Section 3.1 we discuss several techniques for embedding metadata. We also compare them, quantifying the metadata embedded by using each method. In Section 3.2 we show how to extract the bytes related to metadata, and we reconstruct the elements originally embedded, quantifying and categorizing them. In Section 3.3 we classify the protocols producing metadata, associating each protocol to a category that describes its intended application domain. In Section 3.4 we discuss overall statistics about metadata, e.g. transaction peaks and space consumption. Finally, in Section 3.5 we discuss other analyses and tools related to this work.

3.1 Embedding metadata in the blockchain

In Sections 3.1.1 to 3.1.7 we present and discuss various techniques to embed metadata in the blockchain. As far as we know, our collection includes all the techniques that have been actually devised so far. In Section 3.1.8 we show how to split large pieces of metadata into smaller pieces that can be distributed in sets of transactions. Finally, in Section 3.1.9 we draw some conclusions on embedding techniques. In particular, we (i) discuss how to commit to specific values without writing them in the blockchain, (ii) classify the different techniques, (iii) describe the historical evolution of them, (iv) compare the techniques, and present several statistics on their usage.

3.1.1 Value field

Transaction outputs specify the amount of Satoshis to send through a field of 8 bytes size. A first way to encode a message m in the blockchain is to build a transaction with an output whose value is the number that represents m. Although users can easily retrieve the moved funds (e.g. by specifying an own address as receiver) this methodology requires to own at least the amount of Satoshis needed to represent m. Moreover, it provides a very few space respect to that provided by other techniques. The BitcoinTimestamp protocol (see Table 3.3) exploits this method for saving a hash. The hash is first split into 16 pieces. Each one is then translated into a Bitcoin amount. Finally, the protocol builds a transaction containing an output for each amount[1].

3.1.2 Input sequence

Users could exploit the `sequence_no` field for appending their own metadata. Despite the fact that it offers only 4 bytes space, being the smallest one, if compared to the other techniques we discovered, this methodology does not have particular side effects on the Bitcoin system.

[1]For instance see transactions f6f89...3ec46 and 49a13...dd64f

3.1.3 Pay-to public key and Pay-to-public key hash

In a `pay-to-Pubkey` output (see Section 2.1.1) users specify the public key allowed to spend the output (65 bytes, 33 when the key is compressed). In a similar fashion, the `pay-to-PubkeyHash` output accepts a 20 byte string `pubKeyHash` representing the hash of the public key. Technically, in both methods users can store an arbitrary message m in place of the expected value. However, we found only protocols using the `pay-to-PubkeyHash` script (see Table 3.3).

There is a cost to the network associated with this technique. Since computing a value k such that $H(k) = m$ (i.e., a preimage of m) and a signature σ such that $ver_k(T, \sigma)$ is a computationally hard operation, transaction outputs crafted in this way are unspendable in practice. However, these outputs are not easily distinguishable from the spendable ones. After all, a Bitcoin node has no way of knowing whether or not a user exists who possesses such a preimage (nor does it know that the data was never intended to represent an address in the first place). As a result, the nodes of the Bitcoin network must keep these transactions in their UTXO set [2] indefinitely. Since the UTXO set is usually stored in RAM for efficiency concerns [36], the bloating of the set negatively affects the memory consumption of nodes [60].

3.1.4 Pay-to-script hash

In a `pay-to-ScriptHash` metadata can be embedded in several ways, e.g.:

- in the output script, saving data in place of the hash.
- in the input script, using (possibly several) `ignored PUSH_DATA` i.e. pushing data into the stack with the PUSH_DATA instruction and subsequently removing it with a OP_DROP.

The former method is quite similar to the idea discussed in Section 3.1.3. The latter pushes data onto the stack. As long as the transaction script completes execution successfully and there is nonzero data remaining on the stack after completion, the transaction is considered valid and can be mined into a block[13]. Both inputs and outputs may contain PUSH_DATA opcodes. There is no rule specifying that the data accumulated on the stack during the script execution must be cleared. In fact, it is necessary for a minimum of one nonzero item to remain in order for execution

to be considered successful. Stack items that are below the topmost item at the end of execution are simply ignored. Consider e.g. a set of scripts such as the following:

```
input  = OP_1
output = OP_15 OP_ADD OP_16 OP_EQUAL
```

The input and output are concatenated as follows:

```
OP_1 OP_15 OP_ADD OP_16 OP_EQUAL
```

The execution of this script will result in a stack containing a single value, `true` (or `0x1`), which is the result of the final `OP_EQUAL` opcode. If, on the other hand, we prepend one or more PUSH_DATA opcodes to the input script, we will obtain an equally valid result. Consider the following scripts:

```
input  = OP_PUSHDATA1 8 0xaabbccddeeff0011 OP_1
output = OP_15 OP_ADD OP_16 OP_EQUAL
```

These are concatenated as follows:

```
OP_PUSHDATA1 8 0xaabbccddeeff0011 OP_1 OP_15 OP_ADD OP_16 OP_EQUAL
```

At the end of the execution of this script, the top item on the stack is the `1` (or `true`) resulting from the `OP_EQUAL` comparison. Underneath this value is `0xaabbccddeeff0011`, which is ignored. The transaction is valid.

Note that transactions that make use of ignored PUSH_DATA opcodes for embedding data do not necessarily bloat the UTXO set: their outputs may be spent by valid addresses because they do not need to overwrite the address fields in the transaction scripts.

We found proposals for adopting `pay-to-ScriptHash` embedding techniques[2]. Further details on `pay-to-ScriptHash` methods are given by Sward *et al.* [123].

3.1.5 OP_RETURN

The OP_RETURN instruction allows to save up to 80 bytes of metadata[3]. However, unlike `pay-to-PubkeyHash`, an `out-script` containing OP_RETURN always evaluates to false, hence the output is provably unspendable, and its transaction can be safely removed from the UTXO. In this way, OP_RETURN overcomes the UTXO consumption issue highlighted in Section 3.1.3.

[2] e.g. https://counterpartytalk.org/t/cip-proposal-p2sh-data-encoding/2169
[3] e.g. transaction d84f8...3a68a

3.1.6 Vanity address

This technique allows to encode metadata in Bitcoin addresses. A personalized address[4] can be generated by brute forcing through keys. Given a pattern, vanity addresses generators like Vanitygen generate a list of addresses and the related private keys. This methodology is resource intensive: despite the maximum size of a message corresponds to that of an address (20 bytes), this technique is impractical for messages bigger than few bytes. The Counterparty protocol uses a vanity address[5] for developing a consensus mechanism called Proof-of-burn. Since it is impractical that someone owns the private key of this address, users voluntary "burn" their Bitcoins by sending them to it. Metadata can be also distributed in several addresses. One such example is the transaction[6] which inputs spell a sequence of short, plain English words: "We're fine, 8chan post fake".

3.1.7 Coinbase transaction

Miners specify how to redeem the reward for the mined block (and the fees of the appended transactions) through the first transaction of the block. This transaction does not have an input script and it contains a field called `coinbase`, that miners usually fill in with metadata. The coinbase data size is between 2 and 100 bytes. Nevertheless, after block 227,835 the available space is reduced, since the Bitcoin Improvement Proposal 34 (BIP0034) [51] requires the first bytes of the coinbase field storing the block height index. Usually coinbase field is used by miners for storing messages identifying the mining pool and voting BIPs (for instance they voted to sopport either the BIP0016 or BIP0017[55]). However, the most famous message appended by using this technique, is included in the genesis block: *"The Times 03/Jan/ 2009 Chancellor on brink of second bailout for banks"*.

3.1.8 Distributing metadata

The techniques discussed in Sections 3.1.1 to 3.1.7 involve storing metadata within a single field. However, sometimes more space is required. Below we describe three techniques used in order to split metadata on multiple fields. Particularly, the first

[4] For instance these Ponzi schemes addresses 1**ponzi**Uju...64q, and 12**PoNZi**Eta...5nq
[5] `1CounterpartyXXXXXXXXXXXXXXXUWLpVr`
[6] Hash: 72162...16807

method uses multiple `pay-to-PubkeyHash` while the second one combines multiple transaction inputs and outputs. Finally, the *transaction chains* technique distributes data on different transactions.

Multi-signature scripts Another approach for saving metadata requires a 1 of N Multisig where $N-1$ pubkeys host the metadata (as `pay-to-PubkeyHash`) and the other signature is adopted for redeeming the transaction. This methodology bloats the UTXO only until the transaction is redeemed.

Multiple transaction inputs or outputs There is a hard limit of 10,000 bytes on the size of a single script [11]. However, there is no upper bound on the number of inputs or outputs a transaction may contain. In the event that a user wishes to encode more metadata than the 10,000 byte limit would permit, he or she may break the data into chunks and distribute them among many inputs or outputs within a single transaction. To ensure that the data can be reconstructed from these fragments, it is necessary for the encoder to decide upon an ordering criteria. The most straightforward scheme relies upon a simple property of Bitcoins transaction format. Inputs and outputs are recorded in two distinct arrays in the transaction data structure. Therefore, they possess an inherent ordering. If the encoder breaks the data into N chunks and stores those chunks sequentially in $output_0$, $output_1$, ... $output_N$, reconstructing the embedded data is as simple as concatenating each chunk in the same order found in the transaction. There are embedding formats that use other criteria for ordering. For example, in the case of the BIT-COMM protocol, which chunks data into a series of transaction outputs, the amount of Bitcoin transferred by each output is used to order the data[7]. Transactions making use of this technique may or may not bloat the UTXO set that is determined by the structure of the individual transaction outputs.

Transaction chains The previous techniques involve storing metadata within a single transaction. However, there are cases in which this may not be ideal, or even possible. For example:

- If the amount of data to be stored exceeds the maximum block size the transaction containing the data will be rejected by the network.[8]

[7] e.g. see transaction 5970a...8ee41
[8] github.com/bitcoin/bitcoin/blob/e5f1f...b0a4c/src/main.cpp#L829

- Large amounts of data will likely require the inclusion of a large transaction fee. In theory, it is possible to send a transaction without any fee at all. In practice, a transaction with no fee (or a too low fee) is unlikely to be mined. Depending on current fee market dynamics, it may be more cost-effective to split the data across multiple transactions.
- A single, large transaction might attract too much attention if the sender wishes to conceal its significance.
- Transactions greater than a certain size are considered "non-standard" and many nodes refuse to relay them. This limit has varied over time, and is now replaced by the concept of "transaction weight" which is similar, but accounts for Segregated Witness data in a different manner.[9] [10] [11]
- At the time of this writing, only one OP_RETURN output is permitted per transaction.[12]

Due to these considerations, it may be preferable to split data into a series of transactions. Given a sufficient amount of time for transaction mining to occur, and a sufficient quantity of Bitcoin to pay for transaction fees, a user implementing this embedding scheme could theoretically encode an infinite amount of data. While it is not necessary to do so, it can be practical to connect transactions containing related data in a chain structure to ensure that the full data set can be more easily recovered. Each transaction in such a chain flows clearly into another transaction, providing a simple, knowable ordering. This allows the user decoding the data to possess nothing more than a single transaction hash or in some cases, nothing at all (see Section 3.2) to detect and recover it. When building a transaction chain, one of the techniques from the previous sections is selected for encoding data into each individual transaction (address, script stack data, etc.). Then, a valid, spendable transaction output is added to each transaction. This output is used as the input to the following transaction in the chain. For this reason, the output must be spendable by an address over which the user embedding the data has control (i.e., the private

[9] github.com/bitcoin/bitcoin/blob/e5f1f...b0a4c/src/main.h#L56
[10] github.com/bitcoin/bitcoin/blob/e5f1f...b0a4c/src/main.cpp#L644-L648
[11] github.com/bitcoin/bitcoin/blob/9022a...8590a/src/policy/policy.cpp#L111
[12] github.com/bitcoin/bitcoin/blob/9022a...8590a/src/policy/policy.cpp#L155

key). When decoding data in these types of chains, it is necessary to ignore any data in the spent output, as it is only used to provide the links the chain.

3.1.9 Comparison and statistics

Embedding vs committing metadata. One of the main blockchain use cases is the notarization of some data, often represented by its hash (see Section 3.3). We observe that several blockchain protocols (at least those that merely require a timestamped commitment) could simply commit to a hash without actually embedding it into the blockchain. Specifically, the `pay-to-contract` (described by Gerhardt et al.[86]) and the `sign-to-contract` technologies[13] allow one to commit to data in arbitrary transactions without requiring extra space beyond that the original transaction would have taken.

We identified some implementations. Btproof uses the SHA-256 hash of the metadata as input for RIPE160 and directly turns it into an address. In a similar fashion Originstamp daily aggregates the hashes received into a seed which will then be hashed into a Bitcoin private key, then public key and address are derived. Both protocols pay a Satoshi to the generated address in order to publish it in the blockchain. The ContractHashTool exploits basic EC math for building an inscript which public keys are only spendable by the holder of the original private key and that cryptographically commits to the contract hash specified. The UTXO is bloated temporary since the transaction can be redeemed with the private key built. Furthermore, this methodology saves space and is better for privacy (metadata is not explicitly written in a field). However, this prevents us from recognizing and quantifying metadata unless the protocol tracks its transactions[14].

Classification of embedding methods. We classify the methods associating each one to a category that describes how it manages chunks of metadata. The *Single* category groups all the methods embedding the whole element into a single chunk. The main difference between the methods belonging to this category is the field in which they embed the chunk. The techniques of the *Multi* category divide an element into multiple chunks and store all the pieces into a single transaction. Each chunk is embedded by exploiting a *Single* technique. For instance, the `multi-signature`

[13] https://bitcointalk.org/index.php?topic=915828.msg10056796
[14] see for instance Originstamp https://app.originstamp.org/status

technique splits an element in several chunks and saves each chunk into a different `out-script` of a transaction by using the `pay-to-PubkeyHash` method. Finally, the *Chains* category spreads pieces across multiple transactions. Therefore, transaction chains exploit the *Single* techniques and, eventually, the *Multi* ones.

Historical perspective. The very first metadata was embedded in the genesis block by Satoshi Nakamoto. Although he exploited the `coinbase` technique, we observe that miners started using frequently this method only around October 2011. The `input-sequence` technique it is not widely adopted, we suppose because it provides few space respect to other techniques. The most straightforward method for embedding arbitrary data in the first 3 years involved using `pay-to-PubkeyHash`. In order to mitigate the UTXO bloating phenomenathe `op_return` method was granted from March 2014 [10]. Notably, the release notes of Bitcoin Core version 0.9.0 state that: *"This change is not an endorsement of storing data in the blockchain. The OP_RETURN change creates a provably-prunable output, to avoid data storage schemes [...] that were storing arbitrary data such as images as forever-unspendable TX outputs, bloating bitcoins UTXO database."*. Technically the OP_RETURN instruction has been part of the scripting language since the first releases of Bitcoin but originally it was considered *non-standard* by nodes, so transactions containing it were difficult to reliably get mined. In Bitcoin Core 0.9.0 the instruction became standard, meaning that all nodes started to relay unconfirmed OP_RETURN transactions. The limit for storing data with `op_return` technique was originally planned to be 80 bytes, but the first official client supporting the instruction, i.e. the release 0.9.0 [10], allowed only 40 bytes. This animated a long debate [20, 5, 6, 18]. From the release 0.10.0 [7] nodes could choose whether to accept or not OP_RETURN transactions, and set a maximum for their size. The release 0.11.0 [8] extended the data limit to 80 bytes, and the release 0.12.0 [9] to a maximum of 83 bytes. In Section 3.3 we discuss the main blockchain use cases (behind currency transfer) and we identify several protocols built on top of Bitcoin. Since the space provided by the `op_return` method is enough to support them, from March 2014 the `op_return` clearly became the most adopted technique. The `pay-to-PubkeyHash` is now used for embedding large files (e.g. images) with also the support of the `multiple in/out`, `multi-signature`, and `transaction-chains` techniques. Protocols like Counterparty migrated from `pay-to-PubkeyHash` to `op_return` and we

rarely found protocols still using `pay-to-PubkeyHash`.

Comparison of embedding methods. We extract and measure the amount of metadata embedded with each technique. Table 3.1 shows the results. In the first two columns we list categories and techniques. The third one illustrates the size of the field storing metadata. The next column shows where is located the field exploited. The fifth column lists the techniques bloating the UTXO. In the last four columns we illustrates the date in which first chunk of metadata appears (sixth column), the number of times a method has been adopted (seventh column)[15], and total and average size of metadata embedded (last two columns).

Note that metadata of the *Chains* category are a subset of metadata shown in the first two categories and *Multi* metadata are a subset of the *Single* metadata. In order to correctly calculate the overall statistics (i.e. avoid to count the same metadata multiple times), the row *Total* must sum only the values of the *Single* techniques. We extract 4,582,661 chunks with a total size of $\sim 97MB$, which is the total amount of metadata embedded in the Bitcoin blockchain up to block number 480,000. Furthermore, this number of chunks is also a good indicator of the total number of transactions containing metadata. Indeed, the number of transactions containing more than one chunk (i.e. the elements produced by *Multi* techniques) is negligible. We observe that $\frac{3}{4}$ of the total metadata are stored using OP_RETURN. The average size of transactions chains is higher than other methods since this technique is exploited for embedding large elements (e.g. images).

Type	Method	Field Size (B)	Hosted in	UTXO Bloating	First Element	Total Elements[15]	Tot. Size (B)	Avg. Size (B)
Single	Value	8	Tx output	No	N/A	N/A	N/A	N/A
	Input sequence	4	Tx input	No	2011/02/25	1,305,372	5,221,488	4
	P2PK-P2PKH	20, 33, 65	Script	Yes	2013/03/16	66,762	1,335,240	20
	P2SH	520	Script	Yes	2013/04/10	1,578	31,560	20
	OP_RETURN	80	Script	No	2014/03/12	2,903,186	76,700,965	26.4
	Vanity Address	20	Script	No	N/A	N/A	N/A	N/A
	Coinbase	2 – 100	Tx input	No	2009/01/03	305,763	18,442,641	60.3
Total	—	—	—	—	**2009/01/03**	**4,582,661**	**101,731,894**	**22.2**
Multi	Multi-signature	Variable	Script	Transient	2013/04/06	15,067	2,926,590	194.23
	Multiple in/out	Variable	Variable	Variable	2013/03/16	529	4,437,616	8,388.68
Chains	Transaction chains	Variable	Variable	Variable	2013/04/06	60	3,470,870	57,847.83

Table 3.1: Statistics about embedding methods.

[15]Specifically, elements of type *Single* are chunks; *Multi* elements are transactions; *Chains* elements are chains of transactions.

3.2 Collection and analysis of Bitcoin metadata

In this section we first show how we parse metadata and reconstruct the original elements (e.g. texts, images, *etc.*) that users intended to represent (Section 3.2.1). Then, in Section 3.2.2, we categorize the metadata extracted discussing the different types of elements found. Finally, we quantify and compare the elements extracted in Section 3.2.3.

3.2.1 Collecting metadata

Users exploit the embedding techniques of Section 3.1 for storing metadata as strings, each one representing either a whole element or just a chunk. In this section we parse the strings extracted and merge pieces in order to reconstruct the elements that users originally stored. One of the most effective techniques for recognising portions of elements involves searching strings for suspicious byte patterns. For example, long strings of contiguous ASCII characters are extremely unlikely to occur in regular transaction data. Similarly, the probability of finding specific bytestrings, like the Gzip header `0x1f9d9070`, is extremely low. Finding such a bytestring is a good indication that further investigation is warranted. We employed several types of these searches which we will discuss below.

Frequency analysis The GNU *strings* utility[16] takes a data source as input and yields all of the ASCII plaintext characters found in that source as output. It provides a flag for filtering out strings of contiguous ASCII characters under a given length. It is possible to run *strings* directly on Bitcoin Core's .dat files, but care must be taken when tuning the filter. Obviously, too low a threshold will yield a huge number of false positives. On the other hand, due to the way inputs and outputs are encoded in transaction data, too high a threshold eliminates plaintext that has been split across multiple transactions or transaction scripts.

While this approach is quite simple, some of the data that we discovered particularly, the conversations and code uploaded to the blockchain by Peter Todd mention that they are specifically intended to be discovered and extracted via this method (see the "Peter Todd plaintext uploader" entry in Section 3.2.2).

[16] man7.org/linux/man-pages/man1/strings.1.html

Compared to the other extraction methods we employed, it offers the lowest barrier-to-entry. Therefore, users encoding relatively large quantities of plaintext data that are intended to be easily retrievable should make note of encoding methods that lend well to retrieval via *strings*.

The `ignored PUSH_DATA` technique (see Section 3.1.4), particularly when applied to input scripts, is quite effective to this end. It allows large amounts of arbitrary data to be stored with minimal interruption by non-ASCII bytes, while ensuring that the transaction is still considered valid by the network. For example, if a large ASCII string is stored in a series of input scripts in a single transaction, the only necessary interruptions will be the minimal set of opcodes required to ensure that the transaction is valid (because input scripts are stored contiguously in transaction data).

File signature Many file formats require the inclusion of specific bytestrings that are common to all files of a given format. For example, many JPEG images begin with the bytestring `0xffd8ffe000104a464946000101`. Similarly, ASCII-armored PGP messages begin with `-----BEGIN PGP MESSAGE-----`. These bytestrings often occur in the header or footer of the file, although there are formats that place them elsewhere. The probability of finding such bytestrings in Bitcoin blocks is exceedingly low, and as such, they provide a useful indicator of embedded data.

We used several tools to detect file signatures present in Bitcoin transactions:

- *binwalk* [80] is a highly-extensible tool for discovering valid files embedded into other data. It provides a powerful language for defining file signatures, as well as a large database of pre-defined signatures for common file formats. It also has the ability to carve detected files out of the surrounding binary data. One can produce a number of valid results simply by running *binwalk* on the Bitcoin Core .dat files. However, because the tool is unaware of the Bitcoin block format, it is only suitable for recovering files embedded into a single transaction script.
- *binary-grep* [67] searches a collection of input files for a single bytestring specified by the user. It outputs the byte offsets of any matches, and possesses a simple carving function.

- *local-blockchain-parser* [68] provides a "binary-grep" subcommand. Unlike *binwalk* and the standalone *binary-grep* utility, this tool is aware of the Bitcoin block format, and searches transactions directly, skipping parts of the format that cannot contain embedded data. When matches are found, it outputs the block hash, transaction hash, script type (input or output), and byte offset of each match.

One of the more successful workflows we discovered for recovering binary files based on file signatures was the following:

(i) We ran *binwalk* and/or *binary-grep* on a .dat file, making note of any results that appeared to be true positives. (ii) If there were promising results, we then ran the "binary-grep" subcommand of *local-blockchain-parser* on that .dat file, which yielded the transaction IDs where those results were found. (iii) For each resulting transaction ID, we manually inspected the transaction graph around that transaction. If it appeared to be an isolated transaction, we ran the tx-info subcommand of *local-blockchain-parser*. If it appeared to be a part of a chain, we ran the "tx-chain" subcommand instead. (iv) We inspected the binary output from the previous step, performing manual carving where necessary, and attempted to ascertain the validity of the results by opening them with applications appropriate to their file type.

Protocol Identifier Numerous protocols mark their metadata by writing a specific string in the first few bytes of each chunk, but the exact number of bytes may vary from protocol to protocol. In Section 3.3 we take advantage of this for associating metadata to protocols. Furthermore, since protocols give a detailed description of the format of the elements produced, in Section 3.2.2 we distinguish different types of metadata and classify them.

Hence, in order to associate metadata to protocols we:

(i) search the web for known associations between identifiers and protocols; (ii) accordingly classify strings beginning with one of the identifiers obtained. In more details, in the first step we query Google to obtain public identifier/protocol bindings. For instance, since we discover that many protocols use the OP_RETURN technique, we execute the query *"Bitcoin OP_RETURN"*, that returns ∼26,500 results, and we manually inspect the first few pages of them. Note that a protocol can be associated with more than one identifier

(e.g., Stampery, Blockstore [49]), or even do not have any identifier. In this way we obtain 45 protocols associated to 39 identifiers; further, we find several protocols that do not use any identifier (e.g., Diploma [22], Chainpoint [15]). We also distinguish the main types of metadata produced by protocols (e.g. Text, Hash and Record). The second step is performed by our tool: it associates chunks of metadata to a protocol. The full list of protocols discovered is shown in Table 3.3, identifiers are listed in Table A.1.

Transaction chains Given that all spent transaction outputs in the Bitcoin ledger naturally form a chain structure, and given the variety of ways in which data can be embedded into the transactions forming a chain, identifying chains containing embedded data is not entirely straightforward.

A transaction may have certain "giveaway" characteristics that suggest the presence of a chain containing data, such as:

1. One or more provably unspendable outputs (that is, OP_RETURN outputs), plus a single spent output. The unspendable output(s) would contain data, while the spent output would be used to continue the chain.
2. One or more unspent outputs (possibly indicative of a `pay-to-PubkeyHash` embedding), plus a single spent output. The unspent output(s) would contain data, while the spent output would be used to continue the chain.
3. The unspent outputs, if any, contain very little Bitcoin value (such outputs are also known as *dust* outputs). Except in the case of the Bit-Comm protocol, which uses output values to order the data in the output scripts, the funds included into outputs that can never be spent are effectively "burned," and add no information to the embedded data. This disincentivizes the embedder from including any more value than is strictly necessary to create a valid transaction.
4. The spent output contains a relatively large amount of Bitcoin, used to fund further dust outputs in subsequent links in the chain.
5. Preceding or subsequent transactions share a similar structure with the transaction in question. Many of the transaction chains we found appeared to have been constructed with the help of software (e.g. the Python

source we extracted). The software we found tends to create strings of transactions sharing a highly similar format. While it is altogether possible to embed data into chains of dissimilar transactions, they would be difficult to find and complex to decode.

These are helpful clues, but not definitive criteria. In fact, there are many other types of transactions which possess the characteristics described above. For example, payouts from mining pools and Bitcoin casinos often send small amounts of Bitcoin to many users at once. These payout transactions are often constructed algorithmically (according to some set of "threshold" rules intended to minimize the impact of the fee on the payout), meaning that preceding and following transactions share a similar structure.

Therefore, it is generally necessary to have some understanding of the embedded data in order to determine whether a given chain is of interest. If a transaction contains a file signature for a file type that is unlikely to fit into the data provided by that transaction, it warrants further investigation.

Extraction of data from interesting transaction chains is relatively easy when using the *local-blockchain-parser* utility. This utility has a "tx-chain" subcommand that takes a single transaction hash and crawls backwards and forwards through the transaction graph, collecting data from the transaction scripts. This data is filtered and permuted to account for the various ways in which transaction chains are constructed. Finally, the data from each transaction are concatenated in the order that they appear in the chain. This process yields a collection of binary files corresponding to the different ways in which data can be embedded into a chain.

3.2.2 Classifying metadata

Below we classify the elements we successfully reconstructed according to the type of data they represent. We distinguish 12 types of metadata and 5 categories.

Plaintext

The first metadata embedded in the blockchain was a message by Satoshi Nakamoto. Since then, users published a significant amount of uncensurable plaintexts. A not exhaustive list of plaintexts includes:

- **Text messages** (e.g. birthday wishes, love statements, greetings). A message belonging to this category is often an end in itself. Several protocols and tools allow to store text messages and retrieve them.

- **Academic articles** We identify the *Bitcoin whitepaper*, and *SHA-1 is dead!* PDFs demonstrating SHA-1 collisions. These PDFs are referenced in the original paper [122].

- **Other texts** We collect several other texts that require extensive manual verification since their encoding is much less standardized than the binary results. For instance, we identify developer conversations and magnet links (Bittorrent links and metadata related to popular video games and movies). Moreover, this category includes coinbase messages, produced by miners. They usually publish text messages for Bitcoin-related purposes, as identify their blocks, vote on proposals, announce what features they support. For instance, miners publish the "P2SH" string for indicating that they accept `pay-to-ScriptHash` scripts. However, we also found miners messages not related to mining (e.g. prayers).

Hash

Document notarization is a widespread use case in the Bitcoin blockchain. Since storing entire documents is expensive and not trivial (see Section 3.1), usually users embed just a hash of the original document. Sometimes documents are aggregated in order to timestamp multiple objects within a single hash. Therefore, this category includes metadata generated by protocols embedding: (i) Hashes of documents, (ii) Hashes of hashes, and (iii) Merkle trees.

Record

We discover protocols publishing strings composed of fields whose meaning can be derived by applying a set of rules established by the originating protocol. We distinguish two main types of record saved in the Bitcoin blockchain:

- **Financial** These records certify the ownership and exchange of digital and physical goods. Goods are represented as digital tokens while users are identified by means of their Bitcoin addresses. Let consider a transaction T from

Alice to *Bob* whose attached metadata is a record exchanging N units of a token G. T proves a change of ownership of N units of the good represented by G.

- **Copyright** These records certify the ownership of digital arts such as photos and videos. Copyright records are also produced by protocols acting as market places in which artists publish and sell their files to other users. Digital arts are not embedded in the blockchain.

Script

Numerous scripts have been embedded in the Bitcoin blockchain.

- **Python scripts** *Satoshi uploader*, *Satoshi downloader*, Cryptograffiti uploader, and the Peter Todd text uploader (occurs twice).
- **Bash scripts** Password script, and OpenSSL encoder.
- **Video games** *LinPyro* (gzipped), and *Bong ball* (HTML).
- **Miscellaneous** *Lucifer* (a burn-in stress testing utility, gzipped).

File

Finally, we reconstruct files by merging metadata spread across multiple transactions. We distinguish the following subcategories:

- **Images** The most embedded files are PNG, JPEG, and GIF.
- **Archives** e.g. the WikiLeaks *Cablegate* archive (gzipped).
- **Encrypted files** Files usually encrypted with *OpenSSL*.

3.2.3 Statistics

Table 3.2 shows statistics about the elements we reconstructed. We aggregate results based on the data type of each element (second column). Types are grouped by their categories (first column), as discussed in Section 3.2.2. Third column indicates the day in which the first element belonging to the correspondent type appears in the blockchain. Next, we show the total number of elements found, followed by their

total size in bytes (fifth column), and their average size (last column). In the *Hash* category we aggregate all its subcategories presented in Section 3.2.2.

The total size of metadata recognised is lower than the $\sim 97MB$ extracted in Table 3.1, since we have *unclassified* metadata, i.e. bytes that we can not associate to a specific element. For instance, consider an OP_RETURN chunk containing the hash of a document and no protocol identifier. Since the data following the OP_RETURN operator is not related to the Bitcoin protocol, the chunk must be considered as metadata in Table 3.1. However, we are not able to categorize it in Table 3.2. Indeed, although its size is compatible with the size of hashes returned by common hashing algorithms, we are not able to prove that it is a hash and not a different element (e.g. an encrypted message, part of a file, a number, *etc.*). Scripts and Files have an average size higher than the maximum size of a script, highlighting the correlation between these categories, and both *Multi* and *Chains* techniques. Although the first Record appears in May 2014, Records are $\frac{3}{4}$ of the elements reconstructed, and they require the majority of the space.

Category	Type	First Element	Tot. Elements	Tot. Size	Avg. Size
Plaintext	Text messages	2015/06/24	4,129	177,916	43.1
	Academic articles	2013/04/06	2	190,772	95,386
	Other texts	2009/01/03	305,763	18,442,641	60.3
	Total	**2009/01/03**	**309,894**	**18,811,329**	**60.7**
Hash	**Total**	**2013/12/18**	**200,832**	**7,617,392**	**37.9**
Record	Financial	2014/05/03	1,427,313	37,699,809	26.4
	Copyrights	2014/12/19	116,406	3,503,170	30.1
	Total	**2014/05/03**	**1,543,719**	**41,202,979**	**26.7**
Script	Python	2013/04/06	5	21,573	4,314.6
	Bash	2013/07/12	2	597	298.5
	Games	2013/04/10	2	84,379	42,189.5
	Others	2013/04/10	1	31,600	31,600
	Total	**2013/04/06**	**10**	**138,149**	**13,814.9**
File	Images	2013/03/17	108	1,523,529	14,106.75
	Archives	2013/04/06	12	2,838,760	236,563.33
	Encrypted files	N/A	N/A	N/A	N/A
	Total	**2013/03/17**	**120**	**4,362,289**	**36,352.4**
TOTAL	—	**2009/01/03**	**2,054,575**	**72,132,138**	**35.1**

Table 3.2: Statistics about metadata.

3.3 Analysis of Bitcoin-based protocols

In this section we study the protocols (introduced in Section 3.2.1) which embed metadata in the Bitcoin blockchain. First, we classify these protocols according to their application domain (Section 3.3.1). Then, in Section 3.3.2, we quantify and compare them.

3.3.1 Classifying protocols

We now classify the protocols, associating each one to a category that describes its intended application domain. To this purpose, we manually inspect the web pages of each protocol.

Financial includes protocols that manage, gather, or distribute money as preeminent feature. Some protocols certify the ownership of a real-world asset, endorse its value, and keep track of trades (e.g., Colu currently tracks over 50,000 assets). Metadata in these transactions are used to specify e.g. the value of the asset, the amount of the asset transferred, the new owner. Moreover, Helperbit implements a donation service, gathering money from users in order to fund humanitarian projects.

Notary includes protocols for certifying the ownership and timestamp of documents. They allow users to publish the hash of a document in a transaction, thus proving its existence and integrity. Notably, since the transaction is signed with his private key, users can also certify the document ownership.

Digital Rights Management includes protocols for declaring access rights and copy rights on digital art files, like e.g. pictures or music.

Message groups protocols for storing short text messages on the blockchain.

Subchain gathers protocols which store sequences of transaction metadata forming *subchains*[64] of the blockchain, used to embed tamper-proof execution traces of third-party contracts.

Empty category is strictly related to the OP_RETURN method. It includes transactions that do not attach any data to OP_RETURN.

We report our classification of protocols in the first two columns of Table 3.3. Due to the OP_RETURN space limit, long pieces of metadata require to be split in many transactions, and higher fees. Hence, `Financial` protocols usually feature complex rules, have space-efficient representations of data, and often propose off-chain solutions [16]. We distinguish `Notary` protocols from `DRM` protocols for the following reason. First, most `Notary` applications do not require users to send their documents (since the hash can be computed locally). Furthermore, their main purpose (certifying ownership) can be fulfilled also when the application is no longer alive. Conversely, `DRM` protocols usually need to gather user documents, and require interactions with users (e.g. they often play the role of broker between producers and consumers). While `Notary`, `DRM` and `Message` elements are unrelated to those published before, `Financial` and `Subchain` elements must be consistent with respect to some system state. For instance, nodes of `Financial` protocols keep internal counters to track the total amount of assets owned by each user, and reach consistency by updating their counters whenever a new record appears in the blockchain. If an attacker produces a record that claims to sell goods that she does not currently own, nodes easily recognise the attack and discard the metadata. In a similar fashion, `Subchain` protocols discard invalid updates. However, we distinguish `Subchain` protocols because they apply a generalization of the above approach. Indeed, they could perform more complex computations (e.g. executing Turing-complete programs) in order to decide if an update is correct.

3.3.2 Statistics

Table 3.3 shows some statistics about protocols. The first and the second columns indicate, respectively, the protocol category (introduced in Section 3.3) and the name. The next column shows the type of metadata embedded (presented in Section 3.2.2). The fourth column lists the embedding techniques used by each protocol (described in Section 3.1). Next column shows the date in which the protocol generated the first element. Since transactions do not have a "date" field, we infer dates from the timestamp of the block containing the transaction. The next two columns count the total number of elements produced by each protocol, and the total size (in bytes) of the stored data. To compute the size we only consider the metadata, i.e. we do not count neither script instructions nor other fields of the transaction. The last column shows the average size of the metadata. The full list of protocols identifiers

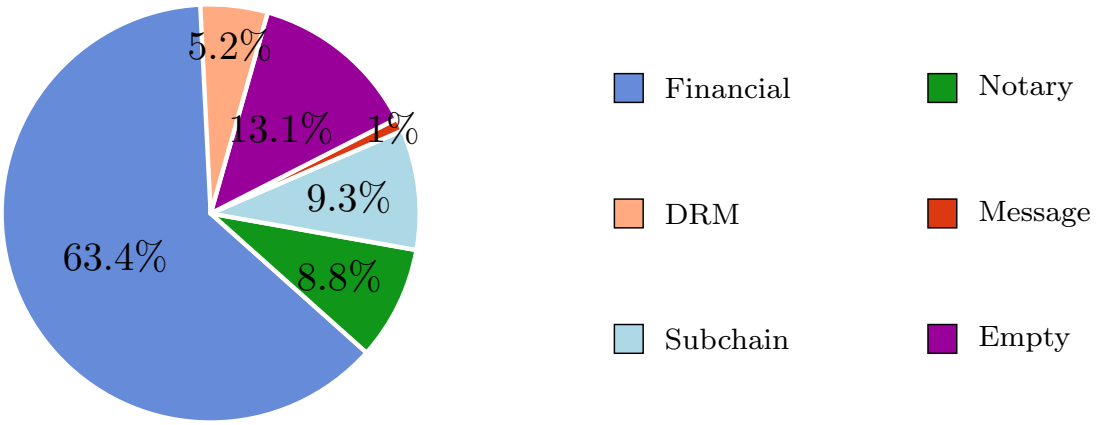


Figure 3.1: Distribution of elements by category.

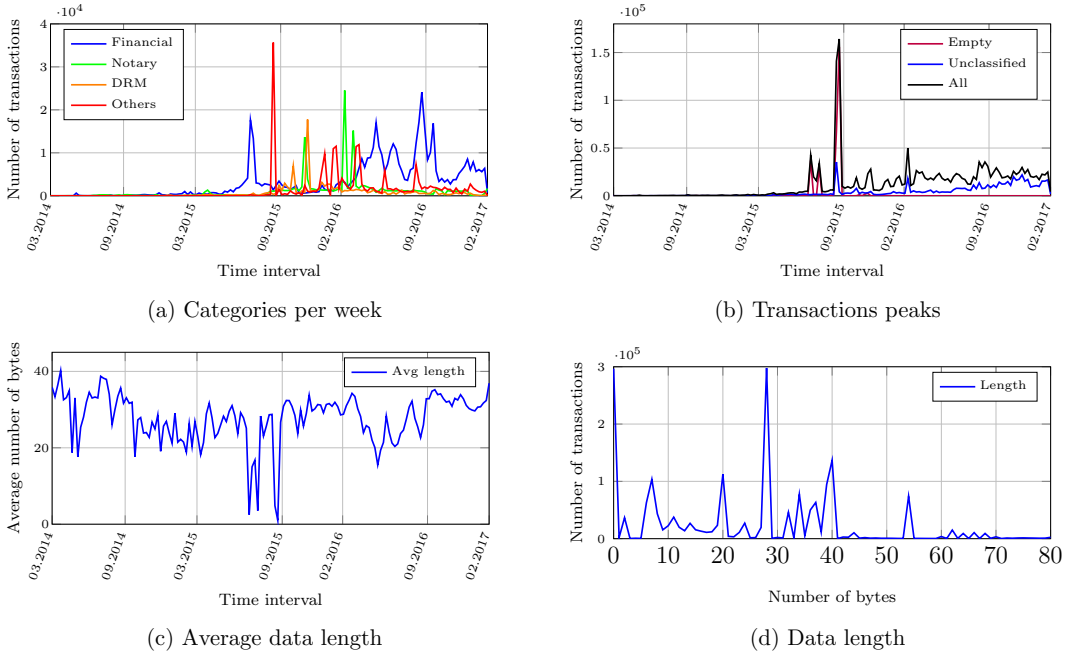
is shown in Table A.1.

The following charts focus on OP_RETURN transactions, since almost all metadata associated to protocols are published by using the OP_RETURN method. Figure 3.1 displays how the metadata elements are distributed into the categories. Figure 3.2a shows the number of elements published per week by each category (we render `Message` and `Subchain` categories in a unique `Other` line for graphical reasons). Figure 3.2b represents peaks of OP_RETURN transactions. For each week, it shows the number of (i) `Empty` transactions, (ii) OP_RETURN transactions not related to any protocol, and (iii) and total OP_RETURN transactions. Figure 3.2c shows the average length of the OP_RETURN metadata of each week. Figure 3.2d represents the number of transactions with a given metadata length.

Unclassified metadata. We associate $\sim 52.5MB$ of metadata to protocols. This value is lower than the total metadata extracted ($\sim 97MB$) for the following reasons. (i) Users often embed metadata not related to any protocol. Particularly, we believe that several images and text messages fall in this case. Furthermore, we do not consider miners metadata (`coinbase` metadata) as part of a protocol. (ii) Several protocols do not use any identifier. Although we discover and classify several of these protocols, we can not recognise their metadata. (iii) There may be other protocols we are unaware of. If they publish OP_RETURN transactions, we track all their metadata in Table 3.1 but we are not able to recognise and categorise them.

Category	Protocol	Metadata	Embedding Method	First Element	Tot. Elements	Tot. Size	Avg. Size
Financial	Colu	Financial Record	OP_RETURN	2015/07/09	244,411	4,425,702	18.1
	CoinSpark	Financial Record	OP_RETURN	2014/07/02	28,120	960,664	34.2
	OpenAssets	Financial Record	OP_RETURN	2014/05/03	207,132	3,255,499	15.7
	Omni	Financial Record	OP_RETURN	2015/08/10	311,605	6,249,883	20
	Openchain	Hash	OP_RETURN	2015/10/21	2,758	115,283	41.8
	Helperbit	Financial Record	OP_RETURN	2015/09/18	33	1,251	37.9
			OP_RETURN	2014/06/16	636,012	22,806,810	35.9
	Counterparty	Financial Record	P2PKH	N/A	N/A	N/A	N/A
			MULTISIG	N/A	N/A	N/A	N/A
	Total	—	—	**2014/06/16**	**1,430,071**	**37,815,092**	**26.4**
Notary	Factom	Merkle root	OP_RETURN	2014/04/11	105,188	4,207,262	40
	Stampery[42]	Merkle root, Hash	OP_RETURN	2015/03/09	74,887	2,648,102	35.4
	Proof of Existence	Hash	OP_RETURN	2014/04/21	5,464	218,513	40
	Blocksign	Hash	OP_RETURN	2014/08/04	1,477	55,676	37.7
	CryptoCopyright	Hash	OP_RETURN	2014/08/02	46	1,840	40
	Stampd	Hash	OP_RETURN	2015/01/03	562	22,427	39.9
	BitProof	Hash	OP_RETURN	2015/02/25	770	30,800	40
	ProveBit	Hash	OP_RETURN	2015/04/05	57	2,280	40
	Remembr	Hash	OP_RETURN	2015/08/25	28	1,128	40.3
	OriginalMy	Hash	OP_RETURN	2015/07/12	126	4,788	38
	LaPreuve	Hash	OP_RETURN	2014/12/07	68	2,663	39.1
	Nicosia	Hash of hashes	OP_RETURN	2014/09/12	24	840	35
	SmartBit	Merkle root	OP_RETURN	2015/11/24	8,472	304,992	36
	Notary	Hash	OP_RETURN	2017/04/11	21	798	38
	Originstamp	Hash of hashes	COMMIT	2013/12/18	905	0	0
	Btproof	Hash	COMMIT	N/A	N/A	N/A	N/A
	BitcoinTimestamp	Hash	VALUE, MULTIPLE	N/A	N/A	N/A	N/A
	Blocknotary	Merkle root	OP_RETURN	N/A	N/A	N/A	N/A
	Tangible	Hash	OP_RETURN	N/A	N/A	N/A	N/A
	Chainpoint	Merkle root	OP_RETURN	N/A	N/A	N/A	N/A
	Diploma	Hash	OP_RETURN	N/A	N/A	N/A	N/A
	Apertus	Hash	P2PKH	N/A	N/A	N/A	N/A
	Chronobit	Hash	N/A	N/A	N/A	N/A	N/A
	Seclytics	Hash	OP_RETURN	N/A	N/A	N/A	N/A
	Total	—	—	**2013/12/18**	**198,095**	**7,502,109**	**37.9**
DRM	Monegraph	Copyright Record	OP_RETURN	2015/06/28	67,286	2,464,282	36.6
	Blockai	Copyright Record	OP_RETURN	2015/01/09	670	38,327	57.2
	Ascribe	Copyright Record	OP_RETURN	2014/12/19	48,450	1,000,561	20.7
	Verisart	Merkle root	N/A	N/A	N/A	N/A	N/A
	Total	—	—	**2014/12/19**	**116,406**	**3,503,170**	**30.1**
Message	Eternity Wall	Text	OP_RETURN	2015/06/24	4,129	177,916	43.1
	Cryptograffiti	Text	P2PKH, MULTIPLE	N/A	N/A	N/A	N/A
	BIT-COMM	Text	P2PKH, MULTIPLE	N/A	N/A	N/A	N/A
	Stone	Text, File	P2PKH, MULTIPLE	N/A	N/A	N/A	N/A
	Key.run	Magnet	OP_RETURN	N/A	N/A	N/A	N/A
	BitAlias	Secret number, Hash	OP_RETURN	—	0	0	0
	Total	—	—	**2015/06/24**	**4,129**	**177,916**	**43.1**
Subchain	Keybase	Merkle root	OP_RETURN	N/A	N/A	N/A	N/A
	Uniquebits	PGP signed hash	P2PKH, P2SH	N/A	N/A	N/A	N/A
	Blockstore	Key-Value	OP_RETURN	2014/12/10	209,422	6,068,584	29
	Catena[124]	Text	OP_RETURN, CHAIN	N/A	N/A	N/A	N/A
	Total	—	—	**2014/12/10**	**209,422**	**6,068,584**	**29**
Empty	**Total**	—	OP_RETURN	**2014/03/20**	**296,396**	**0**	**0**
TOTAL	—	—	—	**2009/01/03**	**2,254,519**	**55,066,871**	**24.4**

Table 3.3: Statistics about protocols.



(a) Categories per week

(b) Transactions peaks

(c) Average data length

(d) Data length

Figure 3.2: Usage and size of OP_RETURN transactions.

Distribution of protocols by category. Although `Financial` protocols produce the highest number of transactions, the most numerous category is `Notary`. Figure 3.2a and the fifth column of Table 3.3 suggest that, originally, the protocols embedding metadata were in the categories `Financial` and `Notary`, while the other use cases were introduced subsequently (indeed, the others categories were not inhabited before the end of 2014). We note a relevant component of `Empty` transactions (296,396 transactions, ∼10% of the total OP_RETURN transactions). Empty transactions use OP_RETURN without any data attached, so they are not associated to any protocol. We evaluate that ∼ 96% of these transactions are related to the peaks discussed in Section 3.4.1. Since those peaks happened in the same period of the stress tests and spam campaign discussed in [60], we conjecture that `Empty` transactions are related to those events[17]. We identify 19 protocols that write data without using any identifier. We also identify one protocol [29] that besides using an identifier for saving document hashes, allows to save text messages without any identifier.

[17]To verify this conjecture we would need to compare the transaction identifiers of our *empty* transactions with the identifiers of [60], which are not available online.

3.4 Overall statistics

We detect 4,582,661 chunks of metadata, by scanning the blockchain until block number 480,000. We observe that the majority of the metadata extracted was embedded by using OP_RETURN. Overall, we found 2,903,186 OP_RETURN transactions, that constitute $\sim 1,18\%$ of the total transactions in the blockchain, and $\sim 0.057\%$ of the size of the blockchain. Furthermore, they constitute the $\sim 1,37\%$ of the transactions and $\sim 0.065\%$ of the size of the portion of the blockchain from 2014/03/12 (when the first OP_RETURN transaction appeared). Although the former measurement considers 8 years of transactions while the latter only considers the last 3 years, we note that the values are very close. We explain this fact by observing that the daily number of transactions rapidly increased since July 2014.

3.4.1 Transaction peaks

Figures 3.2a and 3.2b display the number of OP_RETURN transactions per week, from 2014/03 (date of the first OP_RETURN transaction) to 2017/02. In the graph we note several peaks, that we explain as follows:

1. ∼100,000 transactions from 2015/07/08 to 2015/08/05. This peak is mainly composed of two different peaks of `Empty` transactions: the july peak (∼36,900 transactions from 2015/07/08 to 2015/07/10) and the august peak (∼29,200 transactions from 2015-08-01 to 2015-08-03). Both peaks seem to be caused by a spam campaign that resulted in a DoS attack on Bitcoin which happened in the same period, as reported in [60].

2. ∼300,000 transactions from 2015/09/09 to 2015/09/23. This second peak is the highest and longest-lasting one. As before, it is mainly caused by `Empty` transactions (∼223,000), although here we also observe a component of `Unclassified` and Blockstore transactions (∼35,000 each). The work [60] detects a spike also in this period, precisely around 2015/09/13, where an anonymous group performed a stress-test on the network with a *money drop*. This involves a public release of private keys, with the aim to cause a big race which would cause a large number of *double-spend* transactions.

3. ∼50,000 transactions from 2016/03/02 to 2016/03/09. The last peak is due to the sum of two different peaks: `Unclassified` (about 18,000) and Stampery

(about 23,000) transactions. We conjecture that this peak is caused by the testing and bootstrap of protocols.

We observe that the Bitcoin blockchain has also other peaks, not related to OP_RETURN transactions. For instance, starting from the 2015/05/22 and for a duration of 100 blocks, the Bitcoin network was targeted by a stress test [3], during which the network was flooded with a huge number of transactions. Actually, the usage of OP_RETURN transactions in the period of this peak does not seem to diverge from their normal usage.

3.4.2 Space consumption

A debated topic in the Bitcoin community is whether it is acceptable or not to save arbitrary data in the blockchain. The seventh column in Table 3.3 shows, for each protocol, the total size of metadata (i.e., not considering the bytes of script instructions and other fields). The last row of Table 3.3 shows that the total size of metadata is $\sim 97MB$ (in the same date, the size of the whole blockchain was $\sim 127,974MB$).

For the most widespread embedding method, the OP_RETURN, Figure 3.2c shows the average length of the metadata of each week. Generally, the average length of metadata is less than 40 bytes, despite the extension to 80 bytes introduced on 2015/07/12. Peaks down on the same period are related to the `Empty` transactions discussed in Section 3.4.1. Figure 3.2d represents the number of OP_RETURN transactions with a given data length: also this chart confirms a small number of transactions that use more than the half of the available space. Note that the discussed peak appears also in this chart, in correspondence of the 0 value. From the last column of Table 3.3 we see that despite Blockai has the higher average size, its value is much less than the 80 bytes available. Several `Notary` protocols take 40 bytes on average: this depends from their identifiers, composed of 16 bytes, and from the size of the hash they save. Generally, `Notary` protocols carry longer data than the other protocols.

We now evaluate the minimum space consumption of the OP_RETURN transactions on the whole blockchain. First, we observe that an `Empty` transaction with one input and one output has a total size of 156 bytes. From Table 3.3 we see that OP_RETURN transaction carry ∼24.4 bytes of metadata, on average. Hence, we approximate the average size of OP_RETURN transaction as ∼180.4 bytes, and so

an approximation of the space consumption of all the OP_RETURN transactions is ∼499 MB.

Finally, we estimate the ratio between the total size of metadata and the size of all the transactions on the blockchain. The block header has size 97 bytes at most. Hence, removing the size of the headers of our 480,000 extracted blocks (∼ 44 MB) from the total size of the blockchain at 2017/08/10, we obtain ∼ 125 GB of transactions. From this we conclude that metadata consume ∼ 0.076% of the total space on the blockchain.

3.5 Related work

There is a growing literature on the analysis of the Bitcoin blockchain [104, 60, 117, 114, 97], and also some online services which perform statistics on Bitcoin metadata [4, 41, 35, 28]. Below, we group the related works into three categories.

The first one includes public tools and services showing metadata embedded in the Bitcoin blockchain. For instance, blockchainarchaeology.com collects files hidden in the blockchain. These files are usually split into several parts, stored e.g. on different output scripts in a transaction. Various techniques are used to detect how the files were embedded (e.g. by binary grep on the PNG pattern), and to reconstruct them. The Bitcoin wiki [4] lists a few protocols using OP_RETURN, together with their identifiers. Excluding the protocol identifiers that are no longer used at time of writing, the collection in [4] is strictly included in ours. The website opreturn.org shows statistics about OP_RETURN transactions, organised by protocol, and statistics about their usage in a certain time frame. The website smartbit.com recognises some OP_RETURN identifiers and shows related statistics. Finally, the website kaiko.com sells data about OP_RETURN transactions.

The second category of related works contains analyses of the data insertion techniques presented in Section 3.1. At the best of our knowledge, besides our work, this category includes only [123, 116], which have been developed concurrently and independently from ours. Despite the common goals, [123, 116] present several differences from our work. In particular: (i) the methods using `pay-to-ScriptHash` are detailed only in [123]; (ii) the `transaction-chains` methods and the techniques for committing metadata are described only in our work; (iii) only our work and [116] extract and quantify the embedded metadata. Further differences between our work

and [116] are discussed below.

The third category of related works includes the analyses of the types of metadata, as that in Section 3.2. Also in this case, the work [116] is the closest to ours: the main difference between the two works is that, while [116] is focussed on discussing the benefits and risks related to metadata (e.g. privacy violations, illegal and condemned contents), we develop a protocol-wise analysis, measuring how much (and when) metadata is embedded by each protocol, and studying which use cases they support. Further, we recognize a few types of metadata (hash, financial records, and copyright records) which are not dealt with by [116].

Chapter 4

An empirical analysis of smart contracts

Ethereum is currently the most prominent platform for executing smart contracts. Contracts are also supported by Bitcoin and several new emerging platforms. In this Chapter 4 we study the usage of smart contracts from various perspectives.

In Section 4.1 we examine and compare a sample of 6 platforms for smart contracts. We study a sample of 834 contracts executed in the Bitcoin and Ethereum platforms, categorizing each of them by its application domain, and measuring the relevance of each of these categories (Section 4.2). In Section 4.3 we analyse the most common design patterns adopted when writing smart contracts. Finally, in Section 4.4 we discuss some related works.

4.1 Platforms for smart contracts

In this section we analyse various platforms for smart contracts. We start by presenting the methodology we have followed to choose the candidate platforms (Section 4.1.1). Then we describe the key features of each platform, pinpointing differences and similarities, and drawing some general statistics (Section 4.1.2).

4.1.1 Methodology

To choose the platforms subject of our study, we have drawn up a candidate list by examining all the articles of coindesk.com in the "smart contracts" category[1]. Starting from June 2013, when the first article appeared, up to the 15th of September 2016, 175 articles were published, describing projects, events, companies and technologies related to smart contracts and blockchains. By manually inspecting all these articles, we have found references to 12 platforms: Bitcoin, Codius, Counterparty, DAML, Dogeparty, Ethereum, Lisk, Monax, Rootstock, Symbiont, Stellar, and Tezos.

We have then excluded from our sample the platforms which, at the time of writing, do not satisfy one of the following criteria: (i) have already been launched, (ii) are running and supported from a community of developers, and (iii) are publicly accessible. For the last point we mean that, e.g., it must be possible to write a contract and test it, or to explore the blockchain through some tools, or to run a node. We have inspected each of the candidate platforms, examining the related resources available online (e.g., official websites, white-papers, forum discussions, *etc.*) After this phase, we have removed 6 platforms from our list: Tezos and Rootstock, as they do not satisfy condition (i); Codius and Dogeparty, which violate condition (ii), DAML and Symbiont, which violate (iii). Summing up, we have a sample of 6 platforms: Bitcoin, Ethereum, Counterparty, Stellar, Monax and Lisk, which we discuss in the following.

4.1.2 Analysis of platforms

We now describe the general features of the collected platforms, focussing on: (i) whether the platform has its own blockchain, or if it just piggy-backs on an already existing one; (ii) for platforms with a public blockchain, their consensus protocol, and whether the blockchain is public or private to a specific set of nodes; (iii) the languages used to write smart contracts.

Bitcoin [109] is a platform for transferring digital currency, the bitcoins (BTC). It has been the first decentralized cryptocurrency to be created, and now is the one with the largest market capitalization. The platform relies on a public blockchain

[1] http://www.coindesk.com/category/technology/smart-contracts-news

to record the complete history of currency transactions. The nodes of the Bitcoin network use a consensus algorithm based moderately hard *"proof-of-work"* puzzles to establish how to append a new block of transactions to the blockchain. Nodes work in competition to generate the next block of the chain. The first node that solves the puzzle earns a reward in BTC.

Although the main goal of Bitcoin is to transfer currency, the immutability and openness of its blockchain have inspired the development of protocols that implement (limited forms of) smart contracts. Bitcoin features a non-Turing complete scripting language, which allows to specify under which conditions a transaction can be redeemed. The scripting language is quite limited, as it only features some basic arithmetic, logical, and crypto operations (e.g., hashing and verification of digital signatures). A further limitation to its expressiveness is the fact that only a small fraction of the nodes of the Bitcoin network processes transactions whose script is more complex than verifying a signature[2].

Ethereum [75] is the second platform for market capitalization, after Bitcoin. Similarly to Bitcoin, it relies on a public blockchain, with a consensus algorithm similar to that of Bitcoin[3]. Ethereum has its own currency, called *ether* (ETH). Smart contracts are written in a stack-based bytecode language [129], which is Turing-complete, unlike Bitcoin's. There also exist a few high level languages (the most prominent being *Solidity*[4]), which compile into the bytecode language. Users create contracts and invoke their functions by sending transactions to the blockchain, whose effects are validated by the network. Both users and contracts can store money and send/receive ETH to other contracts or users.

Counterparty [19] is a platform without its own blockchain; rather, it embeds its data into Bitcoin transactions. While the nodes of the Bitcoin network ignore the data embedded in these transactions, the nodes of Counterparty recognise and interpret them. Smart contracts can be written in the same language used by Ethereum. However, unlike Ethereum, no consensus protocol is used to validate the results of computations[27]. Counterparty has its own currency, which can be

[2]As far as we know, currently only the *Eligius* mining pool accepts more general transactions (called *non-standard* in the Bitcoin community). However, this pool only mines $\sim 1\%$ of the total mined blocks [59].

[3]The consensus mechanism of Ethereum is a variant of the GHOST protocol in [120].

[4]Solidity: `http://solidity.readthedocs.io/en/develop/index.html`

transferred between users, and be spent for executing contracts. Unlike Ethereum, nodes do not obtain fees for executing contracts; rather, the fees paid by clients are destroyed. This mechanism is called *proof-of-burn*.

Stellar [44] features a public blockchain with its own cryptocurrency, governed by a consensus algorithm inspired to federated Byzantine agreement [45]. Basically, a node agrees on a transaction if the nodes in its neighbourhood (that are considered more trusted than the others) agree as well. When the transaction has been accepted by enough nodes of the network, it becomes infeasible for an attacker to roll it back, and it is considered as confirmed. Compared to *proof-of-work*, this protocol consumes far less computing power, since it does not involve solve cryptographic puzzles. Unlike Ethereum, there is no specific language for smart contracts; still, it is possible to gather together some transactions (possibly ordered in a chain) and write them atomically in the blockchain. Since transactions in a chain can involve different addresses, this feature can be used to implement basic smart contracts. For instance, assume that a participant A wants to pay B only if B promises to pay C after receiving the payment from A. This behaviour can be enforced by putting these transactions in the same chain. While this specific example can be implemented on Bitcoin as well, Stellar also allows to batch operations different from payments[5], e.g. creating new accounts. Similarly to Bitcoin, Stellar features special accounts, called *multisignature*, which can be handled by several owners. To perform operations on these accounts, a threshold of consensus must be reached among the owners. Transaction chaining and multisignature accounts can be combined to create more complex contracts.

Monax [32] supports the execution of Ethereum contracts, without having its own currency. Monax allows users to create private blockchains, and to define authorisation policies for accessing them. Its consensus protocol[6] is organised in rounds, where a participant proposes a new block of transactions, and the others vote for it. When a block fails to be approved, the protocol moves to the next round, where another participant will be in charge of proposing blocks. A block is confirmed when it is approved by at least 2/3 of the total voting power.

[5] https://www.stellar.org/developers/guides/concepts/operations.html
[6] https://tendermint.com/

Platform	Blockchain			Contract Language	Total Tx	Volume (K USD)	Marketcap (M USD)
	Type	Size	Block int.				
Bitcoin	Public	96 GB	10 min.	Bitcoin scripts + signatures	184,045,240	83,178	15,482
Counterparty				EVM bytecode	12,170,386	33	4
Ethereum	Public	17-60 GB	12 sec.	EVM bytecode	14,754,984	10,354	723
Stellar	Public	?	3 sec.	Transaction chains + signatures	?	35	17
Monax	Private	?	Custom	EVM bytecode + permissions	?	n/a	n/a
Lisk	Private	?	Custom	JavaScript	?	45	15

Table 4.1: General statistics of platforms for smart contracts.

Lisk [30] has its own currency, and a public blockchain with a *delegated proof-of-stake* consensus mechanism[7]. More specifically, 101 active delegates, each one elected by the stakeholders, have the authority to generate blocks. Stakeholders can take part to the electoral process, by placing votes for delegates in their favour, or by becoming candidates themselves. Lisk supports the execution of Turing-complete smart contracts, written either in JavaScript or in Node.js. Unlike Ethereum, determinism of executions is not ensured by the language: rather, programmers must take care of it, e.g. by not using functions like *Math.random*. Although Lisk has a main blockchain, each smart contract is executed on a separated one. Users can deposit or withdraw currency from a contract to the main chain, while avoiding double spending. Contract owners can customise their blockchain before deploying their contracts, e.g. choosing which nodes can participate to the consensus mechanism.

Table 4.1 summarizes the main features of the analysed platforms. The question mark in some of the cells indicates that we were unable to retrieve the information (e.g., we have not been able to determine the size of Monax blockchains, since they are private). The first three columns next to the platform name describe features of the blockchain: whether it is public; its size; the average time between two consecutive blocks. Note that Bitcoin and Counterparty share the same cell, since the second platform uses the Bitcoin blockchain. Measuring the size of the Ethereum blockchain depends on which client and which pruning mode is used. For instance, using the Geth client, we obtain a measure of 17GB in "fast sync" mode, and of 60GB in "archive" mode[8]. In platforms with private blockchains, their block interval is custom. The fifth column describes the support for writing contracts. The sixth column shows the total number of transactions[9]. The last two

[7] https://lisk.io/documentation?i=lisk-handbooks/DelegateHandbook

[8] https://redd.it/5om2lw

[9] Sources: https://blockchain.info/charts/n-transactions-total (for Bitcoin), https:

columns show the daily volume of currency transfers, and the market capitalisation of the currency (both in USD, rounded, respectively, to thousands and millions). Market capitalisations from coinmarketcap.com. All values reported on Table 4.1 are updated to January 1st, 2017.

4.2 Analysing the usage of smart contracts

In this section we analyse the usage of smart contracts, proposing a classification which reflects their application domain. Then, focussing on Bitcoin and Ethereum, we quantify the usage of smart contracts in relation to their application domain. We start by presenting the methodology we have followed to sample and classify Bitcoin and Ethereum smart contracts (Section 4.2.1). Then, we introduce our classification and our statistical analysis (Sections 4.2.2 and 4.2.3).

4.2.1 Methodology

We sample contracts from Bitcoin and Ethereum as follows:

- for Ethereum, we collect on January 1st, 2017 all the contracts marked as "verified" on the blockchain explorer `etherscan.io`. This means that the contract bytecode stored on the blockchain matches the source code (generally written in a high level language, such as Solidity) submitted to the explorer. In this way, we obtain a sample of 811 contracts.

- for Bitcoin, we start by observing that many smart contracts save their metadata on the blockchain through the OP_RETURN instruction of the Bitcoin scripting language [12, 31, 4, 65]. We then scan the Bitcoin blockchain on January 1st 2017, searching for transactions that embed in an OP_RETURN some metadata attributable to a Bitcoin smart contract. To this purpose we use an explorer[10] which recognises 23 smart contracts, and extracts all the transactions related to them.

`//blockscan.com` (Counterparty), and `https://etherscan.io` (Ethereum).

[10] `https://github.com/BitcoinOpReturn/OpReturnTool`

4.2.2 A taxonomy of smart contracts

We propose a taxonomy of smart contracts into five categories, which describe their intended application domain. We then classify the contracts in our sample according to the taxonomy. To this purpose, for Ethereum contracts we manually inspect the Solidity source code, while for Bitcoin contracts we search their web pages and related discussion forums. After this manual investigation, we distribute all the contracts into the five categories, that we present below.

Financial. Contracts in this category manage, gather, or distribute money as pre-eminent feature. Some contracts certify the ownership of a real-world asset, endorse its value, and keep track of trades (e.g., Colu currently tracks over 50,000 assets on Bitcoin). Other contracts implement crowdfunding services, gathering money from investors in order to fund projects (the Ethereum DAO project was the most representative one, until its collapse due to an attack in June 2016). High-yield investment programs are a type of Ponzi schemes [63] that collect money from users under the promise that they will receive back their funds with interest if new investors join the scheme (e.g., Government, KingOfTheEtherThrone). Some contracts provide an insurance on setbacks which are digitally provable (e.g., Etherisc sells insurance policies for flights; if a flight is delayed or cancelled, one obtains a refund). Other contracts publish advertisement messages (e.g., PixelMap is inspired to the Million Dollar Homepage).

Notary. Contracts in this category exploit the immutability of the blockchain to store some data persistently, and in some cases to certify their ownership and provenance. Some contracts allow users to write the hash of a document on the blockchain, so that they can prove document existence and integrity (e.g., Proof of Existence). Others allow to declare copyrights on digital arts files, like photos or music (e.g., Monegraph). Some contracts (e.g., Eternity Wall) just allow users to write down on the blockchain messages that everyone can read. Other contracts associate users to addresses (often represented as public keys), in order to certify their identity (e.g., Physical Address).

Game. This category gathers contracts which implement *games of chance* (e.g., LooneyLottery, Dice, Roulette, RockPaperScissors) and *games of skill* (e.g.,

Category	Platform	Contracts	Transactions
Financial	Bitcoin	6	470,391
	Ethereum	373	624,046
Notary	Bitcoin	17	443,269
	Ethereum	79	35,253
Game	Bitcoin	0	0
	Ethereum	158	58,257
Wallet	Bitcoin	0	0
	Ethereum	17	1,342
Library	Bitcoin	0	0
	Ethereum	29	37,034
Unclassified	Bitcoin	0	0
	Ethereum	155	3,679
Total	Bitcoin	23	913,660
	Ethereum	811	759,611
	Overall	834	1,673,271

Table 4.2: Transactions by category.

Etherization), as well as some games which mix chance and skill (e.g., PRNG challenge pays for the solution of a puzzle).

Wallet. The contracts in this category handle keys, send transactions, manage money, deploy and watch contracts, in order to simplify the interaction with the blockchain. Wallets can be managed by one or many owners, in the latter case requiring multiple authorizations (like, e.g. in Multi-owned).

Library. These contracts implement general-purpose operations (like e.g., math and string transformations), to be used by other contracts.

4.2.3 Quantifying the usage of smart contracts by category

We analyse all the transactions related to the 834 smart contracts in our sample. Table 4.2 displays how the transactions are distributed in the categories of Section 4.2.2. For both Bitcoin and Ethereum, we show the number of detected contracts (third column), and the total number of transactions (fourth column).

Overall, we have 1,673,271 transactions. Notably, although Bitcoin contracts are fewer than those running on Ethereum, they have a larger amount of transactions each. A clear example of this is witnessed by the financial category, where 6 Bitcoin contracts[11] totalize two thirds of the transactions published by the 373 Ethereum contracts in the same category. While both Bitcoin and Ethereum are

[11]Bitcoin financial contracts: Colu, CoinSpark, OpenAssets, Omni, SmartBit, BitPos.

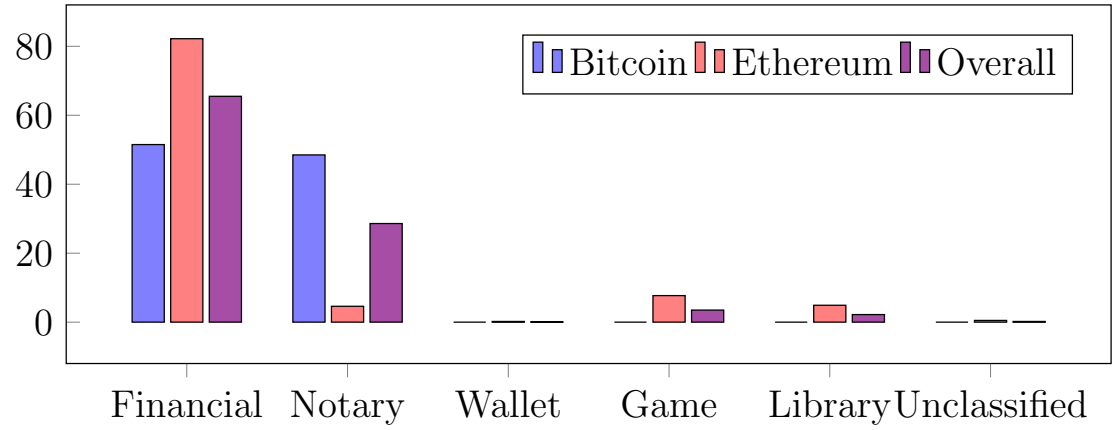


Figure 4.1: Distribution of transactions by category.

mainly focussed on financial contracts, we observe major differences about the other categories. For instance, the Bitcoin contracts in the Notary category[12] have an amount of transactions similar to that of the Financial category, unlike in Ethereum. The second most used category in Ethereum is Game. Although some games (e.g., lotteries [53, 54, 57, 69] and poker [94]) which run on Bitcoin have been proposed in the last few years, the interest on them is still mainly academic, and we have no experimental evidence that these contracts are used in practice. Instead, the greater flexibility of the Ethereum programming language simplifies the development of this kind of contracts (although with some quirks [79] and limitations[13]). Note that in some cases there are not enough elements to categorise a contract. This happens e.g., when the contract does not link to the project webpage, and there are neither comments in online forums nor in the contract sources.

4.3 Design patterns for Ethereum smart contracts

In this section we study design patterns for Ethereum smart contracts. To this purpose, we consider the sample of 811 contracts collected through the methodology

[12]Bitcoin notary contracts: Factom, Stampery, Proof of Existence, Blocksign, CryptoCopyright, Stampd, BitProof, ProveBit, Remembr, OriginalMy, LaPreuve, Nicosia, Chainpoint, Diploma, Monegraph, Blockai, Ascribe, Eternity Wall, Blockstore.

[13]Although the Ethereum virtual machine is designed to be Turing-complete, in practice the limitations on the amount of gas which can be used to invoke contracts also limit the set of computable functions (e.g., verifying checkmate exceeds the current gas limits of a transaction [89]).

described in Section 4.2. By manually inspecting the Solidity source code of each of these contracts, we identify some common design patterns. We start in Section 4.3.1 by describing these patterns. Then, in Section 4.3.2 we measure the usage of the patterns in the various categories of contracts identified in Section 4.2.

4.3.1 Design patterns

Token. This pattern is used to distribute some fungible goods (represented by tokens) to users. Tokens can represent a wide variety of goods, like e.g. coins, shares, outcomes or tickets, or everything else which is transferable and countable. The implications of owning a token depend on the protocol and the use case for which the token has been issued. Tokens can be used to track the ownership of physical properties (e.g., gold [21]), or digital ones (e.g., cryptocurrency). Some crowdfunding systems issue tokens in exchange for donations (e.g., the Congress contract). Tokens are also used to regulate user authorizations and identities. For instance, the DVIP contract specifies rights and term of services for owners of its tokens. To vote on the poll ETCSurvey, users must possess a suitable token. Given the popularity of this pattern, its standardisation has been proposed [24]. Notably, the majority of analysed Ethereum contracts which issue tokens already adhere to it.

Authorization. This pattern is used to restrict the execution of code according to the caller address. Majority of the analysed contracts check if the caller address is that of the contract owner, before performing critical operations (e.g., sending ether, invoking suicide or selfdestruct). For instance, the owner of Doubler is authorized to move all funds to a new address *at any time* (this may raise some concerns about the trustworthiness of the contract, as a dishonest owner can easily steal money). Corporation checks addresses to ensure that every user can vote only once per poll. CharlyLifeLog uses a white-list of addresses to decide who can withdraw funds.

Oracle. Some contracts may need to acquire data from outside the blockchain, e.g. from a website, to determine the winner of a bet. The Ethereum language does not allow contracts to query external sites: otherwise, the determinism of computations would be broken, as different nodes could receive different results for the same query. Oracles are the interface between contracts and

the outside. Technically, they are just contracts, and as such their state can be updated by sending them transactions. In practice, instead of querying an external service, a contract queries an oracle; and when the external service needs to update its data, it sends a suitable transaction to the oracle. Since the oracle is a contract, it can be queried from other contracts without consistency issues. One of the most common oracles is Oraclize[14]: in our sample, it is used by almost all the contracts which resort to oracles.

Randomness. Dealing with randomness is not a trivial task in Ethereum. Since contract execution must be deterministic, all the nodes must obtain the same value when asking for a random number: this struggles with the randomness requirements wished. To address this issue, several contracts (e.g., Slot) query oracles that generate these values off-chain. Others (e.g., Lottery) try to generate the numbers locally, by using values not predictable *a priori*, as the hash of a block not yet created. However, these techniques are not generally considered secure [56].

Poll. Polls allows users to vote on some question. Often this is a side feature in a more complex scenario. For instance, in the Dice game, when a certain state is reached, the owner issues a poll to decide whether an emergency withdrawal is needed. To determine who can vote and to keep track of the votes, polls can use tokens, or they can check the voters' addresses.

Time constraint. Many contracts implement time constraints, e.g. to specify when an action is permitted. For instance, BirthdayGift allows users to collect funds, which will be redeemable only after their birthday. In notary contracts, time constraints are used to prove that a document is owned from a certain date. In game contracts, e.g. Lottery, time constraints mark the stages of the game.

Termination. Since the blockchain is immutable, a contract cannot be deleted when its use has come to an end. Hence, developers must forethink a way to disable it, so that it is still present but unresponsive. This can be done manually, by inserting ad-hoc code in the contract, or automatically, calling `selfdestruct` or `suicide`. Usually, only the contract owner is authorized to terminate a contract (e.g., as in SimpleCoinFlipGame).

[14] `http://www.oraclize.it/`

	Token	Auth.	Oracle	Random.	Poll	Time	Termin.	Fork	Math	None
Financial	24-51	51-39	2-15	1-2	5-29	23-31	14-30	8-69	4-47	29-66
Notary	13-6	52-9	1-2	0-0	8-9	20-6	29-13	0-0	1-3	30-15
Game	3-3	84-27	25-74	72-93	25-57	73-43	21-19	1-3	2-9	1-1
Wallet	18-2	100-3	0-0	0-0	0-0	94-6	100-10	0-0	12-6	0-0
Library	0-0	31-2	0-0	14-3	0-0	24-3	24-4	34-24	21-19	17-3
Unclassified	43-39	66-21	3-9	1-1	3-6	18-10	28-25	28-25	1-5	15-15
Total	*21-100*	*61-100*	*7-100*	*15-100*	*9-100*	*33-100*	*22-100*	*5-100*	*4-100*	*20-100*

Table 4.3: Relations between design patterns and contract categories.

Math. Contracts using this pattern encode the logic which guards the execution of some critical operations. For instance, Badge implements a method named `subtractSafely` to avoid subtracting a value from a balance when there are not enough funds in an account.

Fork check. The Ethereum blockchain has been forked four times, starting from July 20th, 2016, when a fork was performed to contrast the effect of the DAO attack [23]. To know whether or not the fork took place, some contracts inspect the final balance of the DAO. Other contracts use this check to detect whether they are running on the main chain or on the fork, performing different actions in the two cases. AmIOnTheFork is a library contract that can be used to distinguish the main chain from the forked one.

4.3.2 Quantifying the usage of design patterns by category

We now study how the design patterns identified in Section 4.3.1 are used in smart contracts. Out of the 811 analysed contracts, 648 use at least one of the 9 patterns presented, for a grand total of 1427 occurrences of usage.

Table 4.3 shows the correlation between the usage of design patterns and contract categories, as defined in Section 4.2. A cell at row i and column j shows a pair of values: the first value is the percentage of contracts of category i that use the pattern of column j; the second one is the percentage of contracts with pattern j which belongs to category i. So, for instance, 24% of the contracts in the financial category use the token pattern, and 51% of all the contracts with the token pattern are financial ones.

We observe that *token*, *authorization*, *time constraint*, and *termination* are generally the most used patterns. Some patterns are spread across several categories

(e.g., *termination* and *time constraint*), while others are mainly adopted only in one. For instance, *oracle* and *randomness* patterns are peculiar of game contracts, while the *token* pattern is mostly used in financial contracts. Although *math* is the less used, it appears in each category. Some contracts do not use any pattern (29% of financial and 30% of notary); almost all the contracts in game and wallet categories uses at least one. Further, only 15% of all the unclassified contracts do no use any pattern at all.

The most frequent patterns in financial contracts are *token* (24%), *authorization* (51%), and *time constraint* (23%). Due to the presence of contracts which implement assets and crowdfunding services, we have that half of contracts using *token* and *math* patterns belong to the financial category. For instance, these services use *token* for representing goods or developing polls. Moreover, a great 69% of contracts that use the *fork check* pattern is financial. This is caused by the necessity of knowing the branch of the fork before deciding to move funds. Finally, several financial applications (29%) perform simple operations (e.g. sending a payment) without using any of our described patterns.

The *authorization* pattern is used in many notary contracts to ensure that only the owner of a document can add or modify its data, in order to avoid tampering. Most gambling games involve players who pay fees to join the game, and rewards that can be collected by the winner. *Authorization* pattern is used to let the owner to be the only one able to redeem participants' fees or to perform administrative operations, and to let the winner withdraw his reward. The *time constraint* pattern is used to distinguish the different phases of the game. For instance, within a specific time interval players can join the game and/or bet; then, bets are over, and the game determines a winner. To choose the winner, some gambling games resort to random numbers, which are often generated through an oracle. Indeed, 25% of games use the *oracle* pattern, and the pattern itself is used 74% of cases by a game contract. Since *all* game contracts invoking an *oracle* (25%) ask for random values, and since 72% of contracts use the *random* pattern, we can deduce that 47% of them generate random numbers without resorting to oracles.

Notably, 100% of wallet contracts adopt both *authorization* and *termination* design patterns. A high 94% also uses *time constraint*. On the contrary, *oracle*, *poll*, and *randomness* patterns are of little use when developing a wallet, while *math* is sometimes used for securing operations on the balance.

4.4 Related work

Due to the mixed flavour of our analysis, which compares different platforms and studies how smart contracts are interpreted on each them, our work relates to various topics. The work [100] proposes design patterns for altering and undoing of smart contracts; so far, our analysis in Section 4.3.2 has not still found instances of these patterns in Ethereum. Among the works which study blockchain technologies, [50] compares four blockchains, with a special focus on the Ethereum one; [119] examines a larger set of blockchains, including also some which does not fit the criteria we have used in our methodology (e.g., RootStock and Tezos). Many works on Bitcoin perform empirical analyses of its blockchain. For instance, [114, 117] study users deanonymization, [104] measures transactions fees, and [60] analyses Denial-of-Service attacks on Bitcoin. The work [88] investigates whether Bitcoin users are interested more on digital currencies as asset or as currency, with the aim of detecting the most popular use cases of Bitcoin contracts, similarly to what we have done in Section 4.2.3. Our classification of Bitcoin protocols based on OP_RETURN transactions is inspired from [65], which also measures the space consumption and temporal trend of OP_RETURN transactions.

Chapter 5

A general framework for blockchain analytics

The last few years have witnessed a growing interest in the analytic studies of blockchains, from their theoretical foundations — both cryptographic [72, 85] and economic [99, 118] — to their security and privacy [52, 73, 87, 92, 103]. Among the research topics emerging from blockchain technologies, we focus on the analysis of the data stored in blockchains. Many works on data analytics have been recently published, addressing anonymity issues, e.g. by de-anonymising users [102, 103, 111, 115], clustering transactions [90, 121], or evaluating anonymising services [106]. Other analyses have addressed criminal activities, e.g. by studying denial-of-service attacks [61, 127], ransomware [96], and various financial frauds [107, 108, 126]. Many statistics on Bitcoin and Ethereum exist, measuring e.g. economic indicators [97, 117], transaction fees [105], the usage of metadata [65], *etc.* A common trait of these works is that they create *views* of the blockchain which contain all the data needed for the goals of the analysis. In many cases, this requires to combine data *within* the blockchain with data from the *outside*. These data are retrieved from a variety of sources, e.g. blockchain explorers, wikis, discussion forums, and dedicated sites (see Table 5.1 for a brief survey).

Although the analytics studies shown in Table 5.1 share several common operations, e.g., scanning all the blocks and the transactions in the blockchain, converting the value of a transaction from bitcoins to *USD*, *etc.*, researchers so far tended to implement ad-hoc tools for their analyses, rather than reusing standard libraries. Further, most of the few available tools have limitations, e.g. they feature a fixed set

Analysis goal	Gathered data	Sources
Anonymity	Transactions graph	bitcoind [102, 103, 106, 115, 121], forum.bitcoin.org [115]
	OP_RETURN metadata	bitcoind [106]
	IP addresses	bitcoin faucet [115], blockchain.info [106]
	address tags	blockchain.info [102, 103, 121], bitcointalk.org [102, 103, 121]
	address tags	bitcoin-otc.com [121], bitfunder.org [121]
Market analytics	Transactions graph	bitcoind [97], blockexplorer.com [117]
	IP addresses	blockchain.info, ipinfo.io [97]
	address tags	blockchain.info [97]
	trade data	bitcoincharts.com [97]
Cyber-crime	Transactions graph	bitcoind [61, 126, 127], blockchain.info [96, 107], Bitcore [71]
	mempool	bitcoind [61]
	unconfirmed transactions	bitcoind [61]
	no longer online services	archive.org [126, 127]
	list of DDoS attacks	bitcointalk.org [127]
	mining pools	blockchain.info, bitcoin wiki [127]
	trades on assets/services	bitcoin wiki [127]
	list of fraudulent services	bitcointalk.org [96, 126], badbitcoin.org [126], cryptohyips.com [126]
	address tags	blockchain.info [126]
	exchange rate	bitcoincharts.com [96, 126], quandl.com [96]
Metadata	OP_RETURN transactions	bitcoind [65]
	OP_RETURN identifiers	kaiko.com, opreturn.org, bitcoin wiki [65]
Transaction fees	Transactions graph	bitcoind [105]
	exchange rate	coindesk.com [105]
	mining pools	blockchain.info [105]

Table 5.1: Data gathered by various blockchain analyses.

of analytics, or they do not allow to combine blockchain data with external data, or they are not amenable to be updated. The consequence is that the same functionalities have been implemented again and again as new analytics have been developed, as witnessed by Table 5.1.

We propose a framework for developing general-purpose analytics on the blockchains of Bitcoin and Ethereum. Its main component is a Scala library which can be used to construct views of the blockchain, possibly integrating blockchain data with data retrieved from external sources. Blockchain views can be stored as SQL or NoSQL databases, and can be analysed by using their query languages.

In Section 5.1 we present our tool through a series of case studies. Then, in Section 5.2 we evaluate the performance of our tool. Finally, in Section 5.3 we compare other general-purpose blockchain analysis tools with ours.

5.1 Creating blockchain analytics

We illustrate our framework through some case studies, which, for uniformity, have been developed for the Bitcoin case. We refer to our project page[3] for some Ethereum examples. Our library APIs provide the following Scala classes to represent the

primitive entities of the blockchain:

- `BlockchainLib`: main library class. It provides the `getBlockchain` method, to iterate over `Block` objects.
- `Block`: contains a list of transactions, and some block-related attributes (e.g., block hash and creation time).
- `Transaction`: contains various related attributes (e.g., transaction hash and size).

The library constructs the above-mentioned Scala objects by scanning a local copy of the blockchain. It uses the client, either Bitcoin Core or Parity, to have a direct access to the blocks, exploiting the provided indices. For Bitcoin, it uses the BitcoinJ library as a basis to represent the various kinds of objects, while for Ethereum it uses suitable Scala representations. The APIs allow constructed objects to be exported as MongoDB documents or MySQL records. In MongoDB (a widespread non-relational DBMS) a database is a set of *collections*, each of them containing *documents*. Documents are lists of pairs `(k,v)`, where `k` is a string (called *field name*), and `v` is either a value or a MongoDB document. Conversely, MySQL implements the relational model, and represents an objects as a record in a table. In Sections 5.1.1 to 5.1.5 we develop a series of analytics on Bitcoin. Full Scala code which builds the needed blockchain views, queries, and analysis results can be found in the project page[3].

5.1.1 A basic view of the Bitcoin blockchain

Since all the analyses shown in Table 5.1 explore the transaction graph (e.g. they investigate output values, timestamps, metadata, *etc.*), our first case study focusses on a basic view of the Bitcoin blockchain containing no external data. The documents in the resulting collection represent transactions, and they include: (i) the transaction hash; (ii) the hash of the enclosing block; (iii) the date in which the block was appended to the blockchain; (iv) the list of transaction inputs and outputs.

We show in Figure 5.1 how to use our APIs to construct this collection. Lines 1-2 are standard Scala instructions to define the `main` function. The object `blockchain` constructed at line 4 is a handle to the Bitcoin blockchain. At line 5 we setup the connection to Bitcoin Core, by providing the needed parameters (user, password,

```
1  object MyBlockchain {
2      def main(args: Array[String]): Unit = {
3
4      val blockchain = BlockchainLib.getBitcoinBlockchain(
5          new BitcoinSettings("user", "password", "8332", MainNet))
6      val mongo = new DatabaseSettings("myDatabase", MongoDB, "user", "password")
7      val myBlockchain = new Collection("myBlockchain", mongo)
8
9      blockchain.end(473100).foreach(block => {
10         block.bitcoinTxs.foreach(tx => {
11             myBlockchain.append(List(
12                 ("txHash", tx.hash),
13                 ("blockHash", block.hash),
14                 ("date", block.date),
15                 ("inputs", tx.inputs),
16                 ("outputs", tx.outputs)
17             ))
18         })
19     })
20     }
21 }
```

Figure 5.1: A basic view of the blockchain.

and port), and by indicating that we want to use the main network (alternatively, the parameter `TestNet` allows to use the test network). At line 6 we setup the connection to MongoDB (alternatively, the parameter `MySQL` allows to use MySQL). Since lines 1-6 are similar for all our case studies, for the sake of brevity we will omit them in the subsequent listings. We declare the target collection `myBlockchain` at line 7. At this point, we start navigating the blockchain (from the origin block up to block number 473100) to populate the collection. To do that we iterate over the blocks (line 9) (note that `b => {...}` is an anonymous function, where `b` is a parameter, and {...} is its body), and for each block we iterate over its transactions (at line 10). For each transaction we append a new document to `myBlockchain` (lines 11-16). This document is a set of fields of the form `(k,v)`, where `k` is the field name, and `v` is the associated value. For instance, at line 12 we stipulate that the field `txHash` will contain the hash of the transaction, represented by `tx.hash`. This value is obtained by the API `BitcoinTransaction`.

Running this piece of code results in a view, which we can process to obtain several standard statistics, like e.g. the number of daily transactions, their average value, the largest recent transactions, *etc.*[1] Hereafter we consider another kind of analysis, i.e. the evolution over the years of the number of transaction inputs and outputs. To this purpose, we run the MongoDB query shown in Figure 5.2. The

[1]Note that one could also perform these queries during the construction of the view. However, this would not be convenient in general, since — as we will see also in the following case studies — many relevant queries can be performed on the same view.

```
db.myBlockchain.aggregate([
    { $group : {
        _id: {
            year : { $year : "$date" },
            month : { $month : "$date" },
            day : { $dayOfMonth : "$date" },
        },
        avgIn: { $avg: {$size : "$inputs"} },
        avgOut: { $avg: {$size : "$outputs"} }
    }},
    { $sort : { _id : 1}}
]);
```

Figure 5.2: A query to estimate the average number of inputs and outputs by date.

query first groups the documents with the same date. Then, for each group, it computes the average number of inputs and outputs. Finally, the results are sorted in ascending order. The results of the query are graphically rendered in Figure 5.3, which shows the average number of inputs/outputs by date. We see that, after an initial phase, the average number of inputs and outputs has stabilised between 2 and 3. This is mainly due to the fact that most transactions are published through standard wallets, which try to minimise the number of inputs; a typical transaction has two outputs, one to perform the payment and the other for the change. We also observe a few peaks in the number of inputs and outputs, which are probably related to experimentation of new services, like e.g. CoinJoin.

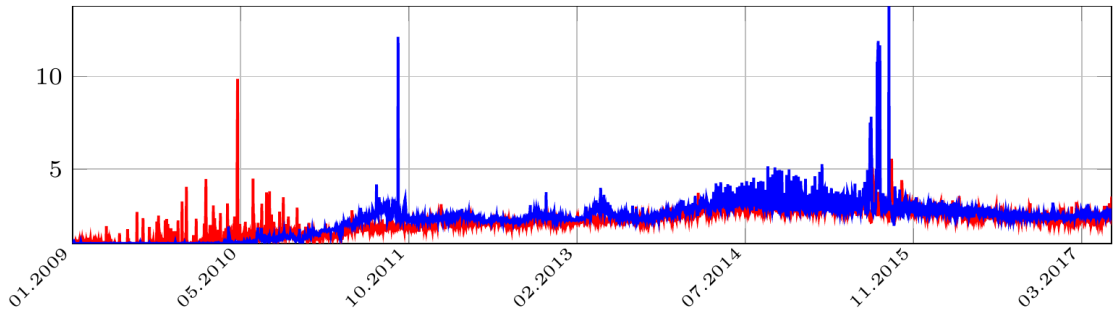


Figure 5.3: Average number of inputs (red line) and outputs (blue line) by date.

5.1.2 Analysing OP_RETURN metadata

Besides being used as a cryptocurrency, Bitcoin allows for appending a few bytes of metadata to transaction outputs. This is done preeminently through the OP_RETURN operator. Several protocols exploit this feature to implement blockchain-based applications, like e.g. digital assets and notarization services [65].

```
1  val opReturnOutputs = new Collection("opReturn", mongo)
2
3  blockchain.start(290000).end(473100).foreach(block => {
4      block.bitcoinTxs.foreach(tx => {
5          tx.outputs.foreach(out => {
6          if(out.isOpreturn()) {
7            opReturnOutputs.append(List(
8            ("txHash", tx.hash),
9            ("date", block.date),
10           ("protocol", OpReturn.getApplication(out.outScript.toString)),
11           ("metadata", out.getMetadata())
12           ))
13         }
14         })
15     })
16 })
```

Figure 5.4: Exposing OP_RETURN metadata.

We now construct a view of the blockchain which exposes the protocol metadata. More specifically, the entries of the view represent transaction outputs, and are composed of: (i) the hash of the transaction containing the output; (ii) the date in which the transaction has been appended to the blockchain; (iii) the name of the protocol that produced the transaction; (iv) the metadata contained in the OP_RETURN script. Figure 5.4 shows the Scala code to construct this collection (we omit the declaration of the `main` method, already shown in Figure 5.1). At line 3 we scan the blockchain, starting from block 290,000 since OP_RETURN transactions were only relayed as standard transactions after the release 0.9.0 of Bitcoin Core. We then iterate through transactions at line 4, and through their outputs at line 5. We append a new document to our collection (lines 7-11) whenever the output of the corresponding transaction is an OP_RETURN (line 6). The method `OpReturn.getApplication` of our APIs takes as input a piece of metadata, and returns the name of the associated protocol. This is inferred by the results of the analysis in [65].

The obtained view can be used to perform various analyses. For instance, we show in Figure 5.5 the number of transactions associated with the most used protocols (only those with at least 1000 transactions). The protocol with the highest number of transactions is Colu, which is used to certify and transfer the ownership of physical assets. The second most used protocol is Omni, followed by Blockstore, a key-value store upon which other protocols are based.

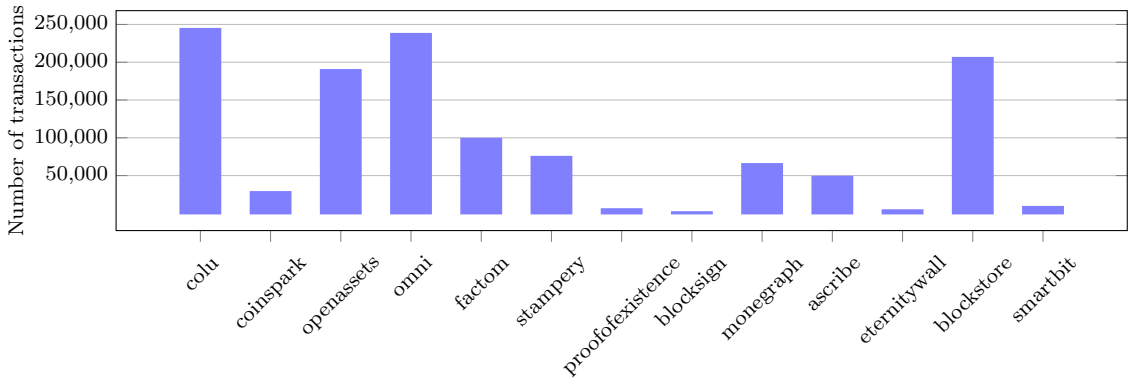


Figure 5.5: Number of transactions per protocol.

```
1  val txWithRates = new Collection("txWithRates", mongo)
2
3  blockchain.end(473100).foreach(block => {
4      block.bitcoinTxs.foreach(tx => {
5          txWithRates.append(List(
6              ("txHash", tx.hash),
7              ("date", block.date),
8              ("outputSum", tx.getOutputsSum()),
9              ("rate", Exchange.getRate(block.date))
10         ))
11     })
12 })
```

Figure 5.6: Exposing exchange rates.

5.1.3 Exchange rates

Several analyses in Table 5.1 use exchange rates for quantifying the economic impact of various phenomena (e.g. cyber-crime attacks, transaction fees, business activities). In this section we analyse how the value transferred in transactions is affected by the exchange rate between *USD* and ₿over the years. Since exchange rates are not stored in the Bitcoin blockchain, we need to obtain these data from an external source, e.g. the Coindesk APIs. Using these data, we construct a blockchain view where each transaction is associated with the exchange rate at the time it has been appended to the blockchain. More specifically, we construct a MongoDB collection whose documents represent transactions containing: (i) the transaction hash; (ii) the date in which the transaction has been appended to the blockchain; (iii) the sum of its output values (in ₿); (iv) the exchange rate between ₿and *USD* in such date.

Figure 5.6 shows the Scala code which builds this collection, using our APIs. At line 1 we declare the collection that we are going to build, `txWithRates`. At lines 3-4 we iterate over all the transactions in the Bitcoin blockchain. For each one, at lines 5-9 we add a new document to `txWithRates`. The total amount of ₿sent by

the current transaction is stored in the field `outputSum` (line 8). The exchange rate is obtained by invoking the method `Exchange` of our APIs (line 9). This method takes a date and retrieves from Coindesk the exchange rate Ƀ/*USD* in that date.

We can analyse the obtained collection in many ways, in order to study how exchange rates are related to the movements of currency in Bitcoin. For instance, one can obtain statistics about the daily transaction volume in *USD*, the market capitalization, the list of richest addresses, *etc.* Hereafter, we measure the average value of outputs (in Ƀ) of the transactions in intervals of exchange rates. The diagram in Figure 5.7 shows the results of this analysis, where we have split exchange rates in 7 intervals of equal size. In the first five intervals we observe the expected behaviour, i.e. the value of outputs decreases as the exchange rate increases. Perhaps surprisingly, the last two intervals show an increase in the value of outputs when the value Ƀhas surpassed 1500 *USD*. This may be explained by speculative operations on Bitcoin.

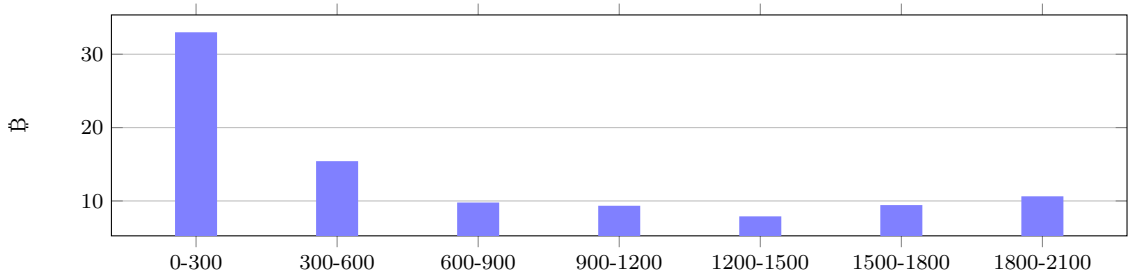


Figure 5.7: Average value of outputs (in Ƀ) by exchange rate.

5.1.4 Transaction fees

In this section we study *transaction fees*, which are earned by miners when they append a new block to the blockchain. Each transaction in the block pays a fee, which in Bitcoin is defined as the difference between its input and output values. While the values of outputs are stored explicitly in the transaction, those of inputs are not: to obtain them, one must retrieve from a past block the transaction that is redeemed by the input. This can be obtained through a "deep" scan of the blockchain, which is featured by our library. We show in Figure 5.8 how to construct a collection which contains, for each transaction: (i) the hash of the enclosing block; (ii) the transaction hash; (iii) the fee; (iv) the date in which the transaction was appended to the blockchain; (v) the exchange rate between Ƀand *USD* in such date.

```
1  val blockchain = BlockchainLib.getBitcoinBlockchain(new BitcoinSettings("user","password","
       8332",MainNet,true))
2  val mongo = new DatabaseSettings("myDatabase", MongoDB, "user", "password")
3  val txWithFees = new Collection("txWithFees", mongo)
4
5  blockchain.end(473100).foreach(block => {
6      block.bitcoinTxs.foreach(tx => {
7          txWithFees.append(List(
8              ("blockHash", block.hash),
9              ("txHash", tx.hash),
10             ("fee", tx.getInputsSum() - tx.getOutputsSum()),
11             ("date", block.date),
12             ("rate", Exchange.getRate(block.date))
13         ))
14     })
15 })
```

Figure 5.8: Exposing transaction fees.

The extra parameter `true` in the `BitcoinSettings` constructor (missing in the previous example), triggers the "deep" scan. When scanning the blockchain in this way, the library maintains a map which associates transaction outputs to their values, and inspects this map to obtain the value of inputs[2]. The methods `getInputsSum` (resp., `getOutputsSum`) at line 10 returns the sum of the values of the inputs (resp., the outputs) of a transaction.

The obtained collection can be used to perform several standard statistics, e.g. the daily total transaction fees, the average fee, the percentage earned by miners from transaction fees, *etc.* Here we analyse the so-called *whale transactions* [95], which pay a unusually high fee to miners. To obtain the whale transactions, we first compute the average $\bar{x}$ and standard deviation σ of the fees in all transactions: in *USD*, we have $\bar{x} = 0.41$, $\sigma = 12.09$. Then, we define whale transactions as those which pay a fee greater than $\bar{x} + 2\sigma = 24.58$ *USD*. Overall we collect 242,839 whale transactions; those with biggest fee are displayed in Figure 5.9.

Fee (USD)	Date	Transaction hash
136243.37	2016-04-26 14:15:22	cc455ae816e6cdafdb58d54e35d4f46d860047458eacf1c7405dc634631c570d
56493.50	2017-01-04 20:01:28	d38bd67153d774a7dab80a055cb52571aa85f6cac8f35f936c4349ca308e6380
39502.15	2017-05-31 14:28:51	cb95ab3aef378c14bc59d0db682d96202b981c1f8fad7d66e23e0be06f2a00c4
25095.71	2017-05-31 14:28:51	8e12a1aba87e4657f5fabec1121ed57f706805ad6d4ffe88c6fce78596bd9b75
23518.00	2013-08-28 10:45:17	4ed20e0768124bc67dc684d57941be1482ccdaa45dadb64be12afba8c8554537

Figure 5.9: The five biggest whale transactions.

[2]Since inputs can only redeemed transactions on past blocks, the map always contains the required output. Although coinbase inputs do not have a value in the map, we calculate their value using the total fees of the current block and the block height (reward is halved each 210,000 blocks).

```
1  val mySQL = new DatabaseSettings("outwithtags", MySQL, "user", "password")
2  val tags=new Tag("src/main/scala/tcs/custom/input.txt")
3  val outTable = new Table(sql"""
4      create table if not exists tagsoutputs(
5        id serial not null primary key,
6        transactionHash varchar(256) not null,
7        txdate TIMESTAMP not null,
8        outvalue bigint unsigned,
9        address varchar(256),
10       tag varchar(256)
11    )""", mySQL)
12
13 blockchain.end(473100).foreach(block => {
14   block.bitcoinTxs.foreach(tx => {
15     tx.outputs.foreach(out => {
16       out.getAddress(MainNet) match {
17         case Some(add) =>
18           tags.getValue(add) match {
19             case Some(tag) => {
20               outTable.insert(sql"insert into tagsoutputs (transactionHash, txdate,
                      outvalue, address, tag) values (${tx.hash.toString}, ${block.
                     date}, ${out.value}, ${add.toString}, ${tags.getValue(add)})")}
21             case None => }
22         case None =>
23       }
24     })
25   })
26 })
```

Figure 5.10: Associating transaction outputs with tags (SQL version).

5.1.5 Address tags

The webpage blockchain.info/tags hosts a list of associations between Bitcoin addresses and *tags* which briefly describe their usage[3]. Table 5.1 shows that address tags are widely adopted, e.g. analytics for cyber-crime usually retrieve addresses tagged as scam or ransomware on forums; market analyses exploit tags for recognising addresses of business services; anonymity studies tag the addresses that seem to belong to the same entity. In this section we construct a blockchain view where outputs are associated with the tags of the address which can redeem them (we discard the outputs with untagged addresses). More specifically, we construct an SQL table whose columns represent transaction outputs containing: (i) hash of the enclosing transaction; (ii) the date in which the transaction has been appended to the blockchain; (iii) the output value (in Ƀ); (iv) the address receiving the payment; (v) the tag associated to the address.

Figure 5.10 shows the Scala script which builds this table. At line 1, we connect to the MySQL database. We retrieve tags from an external source, the

[3]For instance, address 1PQCrkzWweCw4huVLcDXttAZbSrrLbJ92L is associated to tag *Linux Mint Donations* http://www.linuxmint.com/donors.php

`blockchain.info` website. While in the previous case studies we have retrieved external data by querying the source (e.g. the Coindesk APIs), in this case we query a local file in which we have stored the data fetched from `blockchain.info`. At line 2, given the file containing tags, the `Tag` class builds a `Map` which associate each address to the correspondent tag. At lines 4–11 we create a new table. At lines 13–15 we iterate over all the transaction outputs. At line 16 we try to extract the address which can redeem the current output. If we find it (line 17), then we search the map for the associated tag (line 18); if a tag is found (line 19) we insert a new row into the `tagsoutputs` table (line 20).

Using the obtained view, one can aggregate transactions on different business levels [97] to obtain statistics about the total number of transactions, the amount of Ƀexchanged, the geographical distributions of tagged service, *etc.* In particular, we aggregate all addresses whose tag starts with *SatoshiDICE*, and then we measure the number of daily transactions which send Ƀto one of these addresses. The diagram in Figure 5.11 shows the results of this analysis. The fall in the number of transactions at the start of 2015 may be due to the fact that SatoshiDICE is using untagged addresses.

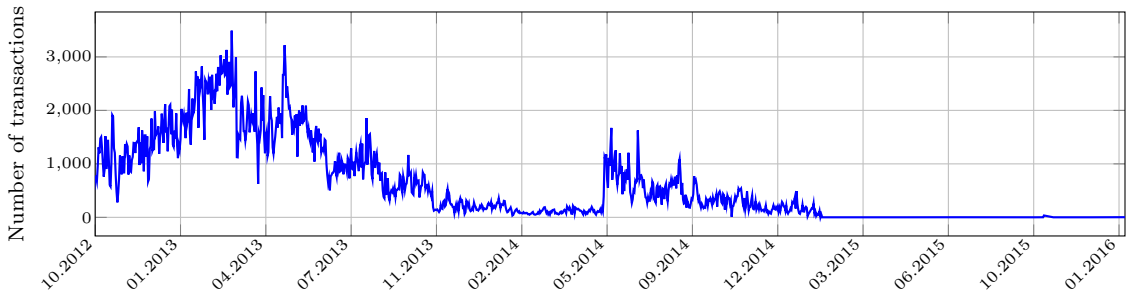


Figure 5.11: Number of daily transactions to addresses tagged with *SatoshiDICE**.

5.2 Implementation and validation

We implement the Ethereum-side of our library by exploiting Parity, queried by means of the web3j library. Bitcoin data is provided by both BitcoinJ and the RPC interface of Bitcoin Core. While BitcoinJ APIs only allow the programmer to retrieve a block by its hash, Bitcoin Core's interface exposes calls to do so by its height on the chain. Furthermore, BitcoinJ's "block objects" do not carry information about block height and the hash of the next block (they only have backward pointers, as

Case study	MongoDB			MySQL		
	Create	Query	Size	Create	Query	Size
Basic view	9 h	2860 s	300 GB	9 h	3.5 h	266 GB
OP_RETURN metadata	2 h	0.5 s	0.5 GB	1.4 h	2.5 s	0.5 GB
Exchange rates	5 h	477 s	34 GB	4.5 h	243 s	27 GB
Transaction fees	9 h	448 s	51 GB	8.5 h	614 s	43.5 GB
Address tags	4 h	1.8 s	0.8 GB	2.3 h	2.7 s	0.6 GB

Table 5.2: Performance evaluation of our framework.

defined in the blockchain), which can be fetched by using Bitcoin Core. Our APIs allow to navigate blockchains. Particularly, in the Bitcoin case, we do this by iterating over these steps: (i) get the hash h of the block of height i, by using Bitcoin Core; (ii) get the block with hash h, by using BitcoinJ; (iii) increment i. By default, the loop starts from 0 and stops at the last block. The methods `blockchain.start(i)`, and `blockchain.end(j)` allow to scan an interval of blockchains, as shown in Section 5.1.2. We write the SQL queries exploiting ScalikeJDBC, a SQL-based DB access library for Scala. ScalikeJDBC provides also a DSL for writing SQL queries.

We carry out our experiments using consumer hardware, i.e. a PC with a quad-core Intel Core i5-4440 CPU @ 3.10GHz, equipped with 32GB of RAM and 2TB of hard disk storage. All the experiments scan the Bitcoin blockchain from the origin block up to block number 473100 (added on 2017/06/27). Table 5.2 displays a comparison of the size of each view, and the time required to create and query it.

Note that the size of the blockchain view constructed in `Basic` (Section 5.1.1) is more than twice than the current Bitcoin blockchain. This is because, while Bitcoin stores scripts in binary format, our library writes them as strings, so to allow for constructing indices and performing queries on scripts. Moreover, the SQL query in `Basic` is particularly slow because of the join operations it performs. Note instead that the query times in SQL and MongoDB are quite similar in all the other cases, where no join operation is required.

5.3 Comparison with related tools

We now compare other general-purpose blockchain analysis tools with ours. Table 5.3 summarises the comparison, focussing on the target blockchain, the DBMS used, the support for creating a custom schema, and for embedding external data.

Tool	Blockchain	Database	Schema	Ext. data	Updated
blockparser	BTC	RAM-only	Custom	Custom	2015-12
rusty-blockparser	BTC	SQL, CSV	Fixed	Custom	2017-09
blockchainsql.io	BTC	SQL	Fixed	None	N/A
BlockSci	BTC	RAM-only	Custom	Custom	2017-09
python-parser	BTC	None	None	Custom	2017-05
Our framework	**BTC, ETH**	**MySQL, MongoDB**	**Custom**	**Custom**	**2017-09**

Table 5.3: General-purpose blockchain analytics frameworks.

The rightmost column indicates the date of the most recent commit in the repository. Note that all the tools which support Bitcoin also work on Bitcoin-based *altcoins*.

The projects `blockparser` and `rusty-blockparser` allow one to perform full scans of the blockchain, and to define custom listeners which are called each time a new block or transaction is read. Unlike our library, these tools offer limited built-in support for combining blockchain and external data. The website `blockchainsql.io` has a GUI through which one can write and execute SQL queries on the Bitcoin blockchain. This is the only tool, among those mentioned in Table 5.3, that does not need to store a local copy of the blockchain. A drawback is that the database schema is fixed, hence it is not possible to use it for analytics which require external data. While the other tools store results on secondary memory, `blockparser` and `BlockSci` keep all the data in RAM. Although this speeds up the execution, it demands for "big memory servers", since the size of the blockchains of both Bitcoin and Ethereum has largely surpassed the amount of RAM available on consumer hardware. Note instead that the disk-based tools also work on consumer hardware. Some low-level optimizations, combined with an in-memory DBMS, help [91] to overwhelm the performance of the disk-based tools. Unlike the other tools, [91] provides also data about transactions broadcast on the peer-to-peer network.

Remarkably, as far as we know none of the analyses mentioned in Table 5.1 uses the general-purpose tools in Table 5.3. Instead, several of them acquire blockchain raw data by using Bitcoin Core[4] (the reference Bitcoin client), and encapsulate them into Java objects with the BitcoinJ APIs before processing. However, neither Bitcoin Core nor BitcoinJ are natural tools to analyse the blockchain: the intended

[4] `https://bitcoin.org/en/bitcoin-core`. Another popular tool for accessing the blockchain was Bitcointools (`https://github.com/gavinandresen/bitcointools`), but it seems no longer available.

use of BitcoinJ is to support the development of wallets, and so it only gives direct access to blocks and transactions from their *hash*, but it does not allow to perform forward scans of the blockchain. On the other hand, Bitcoin Core would provide the means to scan the blockchain, but this requires expertise on its low-level RPC interface, and even doing so would result in raw pieces of JSON data, without any abstraction layer.

A precise comparison of the performance of these tools against ours is beyond the goals of this thesis. The performance analysis in Table 5.2 is a first step towards the definition of a suite of benchmarks for evaluating blockchain parsers.

Chapter 6

Conclusions

In this thesis we have shown tools and techniques for analysing blockchains and smart contracts. We have analysed and compared six different blockchains. Focusing on Bitcoin and Ethereum, we have studied the embedded metadata and the operating smart contracts. Furthermore, we have proposed a general framework for analysing blockchains. The results of our studies have revealed several insights, which are reported below in Section 6.1. In Section 6.2 we propose some directions for further work.

6.1 Summary of the main results

Comparison of the techniques for embedding metadata. Although Bitcoin does not explicitly provide a way to embed metadata into transactions, over the years users have devised various techniques that reach such purpose. We have illustrated and compared these techniques, and we have extracted all the metadata embedded up to 2017/08/10 (first 480,000 blocks), measuring the amount of bytes stored by each technique. We have shown that the most used technique during the first years was the `pay-to-PubkeyHash`. However, since March 2014, when OP_RETURN transactions became standard, this embedding method rapidly has overcome the others. Indeed, $\sim 75\%$ of metadata in our collection have been embedded by using OP_RETURN. While in the first year of existence of OP_RETURN transactions only a few hundreds were appended per week, their usage has been steadily increasing since March 2015. In the last weeks of our experiments we counted $\sim 40,000$ new OP_RETURN transactions per week. We have shown that

`pay-to-Pubkey`, `pay-to-PubkeyHash`, and `pay-to-ScriptHash` produce outputs unspendable in practice, contributing to the UTXO bloating phenomena that negatively affects miners performance. We have measured that at least 68,340 outputs belonging to the UTXO set are unspendable in practice because they actually are chunks of metadata. Conversely, OP_RETURN outputs do not bloat the UTXO, however they still affect the total size of the blockchain.

Analysis of the embedded elements. We have parsed the bytes extracted from metadata in order to reconstruct the original contents. We have recognised 2,054,575 items, with a total size of $\sim 69MB$ (out of $\sim 97MB$ totally embedded). We have classified the items found, distinguishing 12 types of metadata which we grouped into 5 categories. The majority of the items are record produced by financial protocols (1,427,313). We have shown that financial records have a small size (26.4 bytes on average), and they do not bloat the UTXO since they are usually embedded with OP_RETURN. Conversely, files are often saved using `pay-to-PubkeyHash`. We have reconstructed 120 files (108 of them are images), for a total size of $\sim 4MB$.

Study of the protocols built on top of Bitcoin. We have discovered 45 distinct protocols embedding metadata in the Bitcoin blockchain, and we have associated each one to a category (5 categories overall) based on its application domain. Furthermore, we have identified which types of metadata each protocol produces, and which embedding techniques it uses. We have measured that 2,254,519 transactions belong to the protocols discovered, with a total size of $\sim 52.5MB$. We have shown that although the majority of the protocols belong to the Notary category, Financial ones produce $\sim 75\%$ of transactions. Protocols usually produce one type of metadata (that depends on the protocol category) and embed information using one technique (almost always OP_RETURN).

Comparison of blockchain data and embedded metadata. The total amount of metadata embedded is $\sim 97MB$, distributed in 4,582,661 transactions. Overall, we have estimated that metadata transactions constitute $\sim 1,9\%$ of the transactions in the blockchain, and use $\sim 0.076\%$ of its space. Although the official Bitcoin documentation discourages the use of the blockchain to store arbitrary data, the trend seems to be a growth in the number of blockchain-based applications that embed their metadata in OP_RETURN transactions. We suppose that the perceived

sense of security and persistence of the Bitcoin blockchain is the main motivation to avoid using cheaper and more efficient storage. If this trend will be confirmed, the specific needs of these applications could affect the future evolution of the Bitcoin protocol.

Survey of smart contracts platforms. We have analysed the usage of smart contracts from various perspectives. In Section 4.1 we have examined a sample of 6 platforms for smart contracts, pinpointing some crucial technical differences between them.

Comparison of Bitcoin and Ethereum smart contracts. For the two most prominent platforms — Bitcoin and Ethereum — we have studied a sample of 834 contracts, we have categorized each one according to its application domain and finally, we have measured the relevance of each category (Section 4.2).

Analysis of design patterns of Ethereum smart contracts. The availability of source code for Ethereum contracts has allowed us to analyse the most common design patterns adopted in writing smart contracts (Section 4.3). We believe that this survey may provide valuable information to developers about new, domain-specific languages for smart contracts. In particular, studying the most common use cases allows to understand which domains deserve more investments. Furthermore, our study of the correlation between design patterns and application domains can be exploited to drive the correct choice of programming primitives of domain-specific languages for smart contracts.

Development of a general framework for blockchain analytics. We have presented a framework for developing general-purpose analytics on the Bitcoin and Ethereum blockchains. Its main component is a Scala library which can be used to construct views of the blockchain, possibly integrating blockchain data with data retrieved from external sources. Blockchain views can be stored as SQL or NoSQL databases, and can be analysed by using their query languages.

Framework validation. Our experiments confirmed the effectiveness and generality of our approach, which uniformly comprises several use cases addressed by

various ad-hoc approaches in literature in a single framework. Indeed, the expressiveness of our framework overcomes that of the closer proposals in the built-in support for external data, and the support of different kinds of databases and blockchains. Importantly, coming in the form of an open source library for a mainstream language, our framework is amenable of being validated and extended by a community effort, following reuse best practices.

Comparison of SQL *vs* NoSQL. On the comparison of SQL *vs* NoSQL, our experiments did not highlight significant differences in the complexity of writing and executing queries in the two languages. Instead, we observed that the schema-less nature of NoSQL databases simplifies the Scala scripts. From Table 5.2 we see that both creation and query time are comparable as order of magnitude. As already discussed in Section 5.2, the difference in the execution time of queries is due to join operations in SQL. A more accurate analysis, carried over a larger benchmark, is scope for future work. Anyway, it is worth recalling that the goal of our proposal is provide to the final user the flexibility to choose the preferred database, rather than ascertain an idea of best-fit-for-all in the choice.

6.2 Future work

We discuss some possible developments of our framework and the analysis presented.

Scaling Ethereum smart contract analysis. We have manually inspected all the contract sources published up to January 1st, 2017. Specifically, we have analysed 811 smart contracts, in order to determine their application domain and inspect the design patterns adopted. However, the amount of published sources has increased in the last months. The total number of sources available on Etherescan in November 2017 is ~ 7000. In this context, it is particularly important to automate the sources analysis, in order to easily extend our studies to new codes. Although it is difficult to understand the application domain of a contract without manually inspecting comments and related forums, we believe that the design patterns analysis could be automated.

Investigating vulnerabilities and attacks on smart contracts. Recently, some authors have started to analyse the security of Ethereum smart contracts:

among these, [56] surveys vulnerabilities and attacks, while [98] and [70] propose analysis techniques to detect them. Our study on design patterns for Ethereum smart contracts could help to improve these techniques, by targeting contracts with specific programming patterns.

Extending our framework for blockchain analytics. Although our framework is general enough to cover most of the analyses in Table 5.1, it has some limitations that can be overcome with future extensions. In particular, some analyses addressing e.g. information propagation, forks and attacks [78, 81, 101, 112] require to gather data from the underlying peer-to-peer network. To support this kind of analyses one has to run a customized node (either of Bitcoin or Ethereum). Such an extension would also be helpful to obtain on-the-fly updates of the analyses.

Bibliography

[1] Bicoin scalability. https://en.bitcoin.it/wiki/Scalability_FAQ. Last accessed 2018/01/01.

[2] Bitcoin core dev update 5 transaction fees embedded data. http://www.coindesk.com/bitcoin-core-dev-update-5-transaction-fees-embedded-data/. Last accessed 2018/01/01.

[3] Bitcoin network survives surprise stress test. http://www.coindesk.com/bitcoin-network-survives-stress-test/. Last accessed 2018/01/01.

[4] Bitcoin OP_RETURN wiki page. https://en.bitcoin.it/wiki/OP_RETURN. Last accessed 2018/01/01.

[5] Bitcoin pull request 5075. https://github.com/bitcoin/bitcoin/pull/5075. Last accessed 2018/01/01.

[6] Bitcoin pull request 5286. https://github.com/bitcoin/bitcoin/pull/5286. Last accessed 2018/01/01.

[7] Bitcoin release 0.10.0. https://bitcoin.org/en/release/v0.10.0. Last accessed 2018/01/01.

[8] Bitcoin release 0.11.0. https://bitcoin.org/en/release/v0.11.0. Last accessed 2018/01/01.

[9] Bitcoin release 0.12.0. https://bitcoin.org/en/release/v0.12.0. Last accessed 2018/01/01.

[10] Bitcoin release 0.9.0. https://bitcoin.org/en/release/v0.9.0. Last accessed 2018/01/01.

[11] Bitcoin script interpreter. https://github.com/bitcoin/bitcoin/blob/fcf646c9b08e7f846d6c99314f937ace50809d7a/src/script/interpreter.cpp#L256. Last accessed 2018/01/01.

[12] Bitcoin wiki contract. https://en.bitcoin.it/wiki/Contract. Last accessed 2018/01/01.

[13] Bitcoin wiki script. https://en.bitcoin.it/wiki/Script. Last accessed 2018/01/01.

[14] Bitcoin wiki transaction. https://en.bitcoin.it/wiki/Transaction. Last accessed 2018/01/01.

[15] Chainpoint website. http://www.chainpoint.org/. Last accessed 2018/01/01.

[16] Colu protocol, torrents. https://github.com/Colored-Coins/Colored-Coins-Protocol-Specification/wiki/Metadata#torrents. Last accessed 2018/01/01.

[17] Colu website. https://www.colu.com/. Last accessed 2018/01/01.

[18] Counterparty open letter and plea to the Bitcoin core development team. http://counterparty.io/news/an-open-letter-and-plea-to-the-bitcoin-core-development-team/. Last accessed 2018/01/01.

[19] Counterparty: Protocol specification. http://counterparty.io/docs/protocol_specification/. Last accessed 2018/01/01.

[20] Developers battle over bitcoin block chain. http://www.coindesk.com/developers-battle-bitcoin-block-chain/. Last accessed 2018/01/01.

[21] Dgx website. https://www.dgx.io/. Last accessed 2018/01/01.

[22] Diploma website. http://diploma.report/. Last accessed 2018/01/01.

[23] Ethereum hard fork 20 july 2016. https://blog.ethereum.org/2016/07/20/hard-fork-completed/. Last accessed 2018/01/01.

[24] Ethereum request for comment 20. https://github.com/ethereum/wiki/wiki/Standardized_Contract_APIs. Last accessed 2018/01/01.

[25] Evolution of blockchain technology: Insights from the github platform. https://dupress.deloitte.com/dup-us-en/industry/financial-services/evolution-of-blockchain-github-platform.html. Last accessed 2018/01/01.

[26] Factom website. https://www.factom.com/. Last accessed 2018/01/01.

[27] Faq: How do smart contracts form a consensus on counterparty? http://counterparty.io/docs/faq-smartcontracts/#how-do-smart-contracts-form-a-consensus-on-counterparty. Last accessed 2018/01/01.

[28] Kaiko data store. https://www.kaiko.com/. Last accessed 2018/01/01.

[29] La preuve website. http://lapreuve.eu/explication.html. Last accessed 2016/12/15.

[30] Lisk. https://lisk.io/. Last accessed 2018/01/01.

[31] Making sense of blockchain smart contracts. http://www.coindesk.com/making-sense-smart-contracts/. Last accessed 2018/01/01.

[32] Monax. https://monax.io/. Last accessed 2018/01/01.

[33] Omni website. http://www.omnilayer.org/. Last accessed 2018/01/01.

[34] Open assets website. https://github.com/OpenAssets/. Last accessed 2018/01/01.

[35] opreturn.org. http://opreturn.org/. Last accessed 2018/01/01.

[36] Peter Todd delayed txo commitments. https://petertodd.org/2016/delayed-txo-commitments. Last accessed 2018/01/01.

[37] Proof of existence website. https://proofofexistence.com/. Last accessed 2018/01/01.

[38] Scalability debate ever end. https://www.cryptocoinsnews.com/will-bitcoin-scalability-debate-ever-end/. Last accessed 2018/01/01.

[39] Scaling debate in Reddit. http://www.coindesk.com/viabtc-ceo-sparks-bitcoin-scaling-debate-reddit-ama/. Last accessed 2018/01/01.

[40] Smart contracts: The good, the bad and the lazy. http://www.multichain.com/blog/2015/11/smart-contracts-good-bad-lazy/. Last accessed 2018/01/01.

[41] Smartbit OP_RETURN statistics. https://www.smartbit.com.au/op-returns. Last accessed 2018/01/01.

[42] Stampery blockchain timestamping architecture. https://s3.amazonaws.com/stampery-cdn/docs/Stampery-BTA-v6-whitepaper.pdf. Last accessed 2018/01/01.

[43] Stampery website. https://stampery.com/. Last accessed 2018/01/01.

[44] Stellar. https://www.stellar.org/. Last accessed 2018/01/01.

[45] The Stellar consensus protocol. https://www.stellar.org/papers/stellar-consensus-protocol.pdf. Last accessed 2018/01/01.

[46] Thinking about smart contract security. https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security/. Last accessed 2018/01/01.

[47] Understanding the DAO attack. http://www.coindesk.com/understanding-dao-hack-journalists/. Last accessed 2018/01/01.

[48] Another bug in the ens, you can win with an unlimited high bid without paying for it, 2017. https://www.reddit.com/r/ethereum/comments/5zctus/another_bug_in_the_ens_you_can_win_with_an/. Last accessed 2018/01/01.

[49] M. Ali, J. Nelson, R. Shea, and M. J. Freedman. Blockstack: A global naming and storage system secured by blockchains. In *USENIX Annual Technical Conference*, 2016.

[50] L. Anderson, R. Holz, A. Ponomarev, P. Rimba, and I. Weber. New kids on the block: an analysis of modern blockchains. *CoRR*, abs/1606.06530, 2016.

[51] G. Andresen. Block v2, height in coinbase. BIP 034, https://github.com/bitcoin/bips/blob/master/bip-0034.mediawiki. Last accessed 2018/01/01.

[52] E. Androulaki, G. Karame, M. Roeschlin, T. Scherer, and S. Capkun. Evaluating user privacy in Bitcoin. In *Financial Cryptography and Data Security*, volume 7859 of *LNCS*, pages 34–51. Springer, 2013.

[53] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. Secure multiparty computations on Bitcoin. In *IEEE S & P*, pages 443–458, 2014.

[54] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. Secure multiparty computations on Bitcoin. *Commun. ACM*, 59(4):76–84, 2016.

[55] A. M. Antonopoulos. *Mastering Bitcoin: unlocking digital cryptocurrencies.* " O'Reilly Media, Inc.", 2014.

[56] N. Atzei, M. Bartoletti, and T. Cimoli. A survey of attacks on Ethereum smart contracts (SoK). In *Principles of Security and Trust (POST)*, volume 10204 of *LNCS*, pages 164–186. Springer, 2017.

[57] A. Back and I. Bentov. Note on fair coin toss via Bitcoin. http://www.cs.technion.ac.il/~idddo/cointossBitcoin.pdf. Last accessed 2018/01/01, 2013.

[58] C. Badertscher, U. Maurer, D. Tschudi, and V. Zikas. Bitcoin as a transaction ledger: A composable treatment. In *CRYPTO*, pages 324–356, 2017.

[59] W. Banasik, S. Dziembowski, and D. Malinowski. Efficient zero-knowledge contingent payments in cryptocurrencies without scripts. In *ESORICS*, pages 261–280, 2016.

[60] K. Baqer, D. Y. Huang, D. McCoy, and N. Weaver. Stressing out: Bitcoin "stress testing". In *Bitcoin Workshop*, pages 3–18, 2016.

[61] K. Baqer, D. Y. Huang, D. McCoy, and N. Weaver. Stressing out: Bitcoin "stress testing". In *Financial Cryptography Workshops*, volume 9604 of *LNCS*, pages 3–18. Springer, 2016.

[62] M. Bartoletti, A. Bracciali, S. Lande, and L. Pompianu. A general framework for bitcoin analytics. *CoRR*, abs/1707.01021, 2017.

[63] M. Bartoletti, S. Carta, T. Cimoli, and R. Saia. Dissecting Ponzi schemes on Ethereum: identification, analysis, and impact. *CoRR*, abs/1703.03779, 2017.

[64] M. Bartoletti, S. Lande, and A. S. Podda. A proof-of-stake protocol for consensus on bitcoin subchains. In *Workshop on Trusted Smart Contracts*, 2017.

[65] M. Bartoletti and L. Pompianu. An analysis of Bitcoin OP_RETURN metadata. In *Financial Cryptography Workshops*, volume 10323 of *LNCS*. Springer, 2017.

[66] M. Bartoletti and L. Pompianu. An empirical analysis of smart contracts: platforms, applications, and design patterns. In *Workshop on Trusted Smart Contracts*, 2017. Also available as arXiv preprint 1703.06322.

[67] B. Bellomy. binary-grep. https://github.com/spooktheducks/binary-grep. Last accessed 2018/01/01.

[68] B. Bellomy. local-blockchain-parser. https://github.com/spooktheducks/local-blockchain-parser. Last accessed 2018/01/01.

[69] I. Bentov and R. Kumaresan. How to use Bitcoin to design fair protocols. In *CRYPTO*, pages 421–439, 2014.

[70] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Beguelin. Formal verification of smart contracts. In *PLAS*, 2016.

[71] S. Bistarelli and F. Santini. Go with the -Bitcoin- flow, with visual analytics. In *ARES*, pages 38:1–38:6, 2017.

[72] J. Bonneau, A. Miller, J. Clark, A. Narayanan, J. A. Kroll, and E. W. Felten. SoK: Research perspectives and challenges for Bitcoin and cryptocurrencies. In *IEEE S & P*, pages 104–121, 2015.

[73] J. Bonneau, A. Narayanan, A. Miller, J. Clark, J. A. Kroll, and E. W. Felten. Mixcoin: Anonymity for Bitcoin with accountable mixes. In *Financial Cryptography and Data Security*, volume 8437 of *LNCS*, pages 486–504. Springer, 2014.

[74] R. G. Brown, J. Carlyle, I. Grigg, and M. Hearn. Corda: An introduction. http://r3cev.com/s/corda-introductory-whitepaper-final.pdf. Last accessed 2018/01/01, 2016.

[75] V. Buterin. Ethereum: a next generation smart contract and decentralized application platform, 2013. https://github.com/ethereum/wiki/wiki/White-Paper. Last accessed 2018/01/01.

[76] A. Churyumov. Byteball: a decentralized system for transfer of value. https://byteball.org/Byteball.pdf. Last accessed 2018/01/01, 2016.

[77] C. D. Clack, V. A. Bakshi, and L. Braine. Smart contract templates: foundations, design landscape and research directions. *CoRR*, abs/1608.00771, 2016.

[78] C. Decker and R. Wattenhofer. Information propagation in the Bitcoin network. In *P2P*, pages 1–10. IEEE, 2013.

[79] K. Delmolino, M. Arnett, A. Miller, A. Kosba, and E. Shi. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *Bitcoin Workshop*, 2016.

[80] devttys0. binwalk. https://github.com/devttys0/binwalk. Last accessed 2018/01/01.

[81] J. A. D. Donet, C. Pérez-Solà, and J. Herrera-Joancomartí. The Bitcoin P2P network. In *Financial Cryptography Workshops*, volume 8438 of *LNCS*, pages 87–102. Springer, 2014.

[82] European Central Bank. Terms of reference task force on distributed ledger technologies. https://www.ecb.europa.eu/paym/initiatives/shared/docs/dlt_task_force_mandate.pdf. Last accessed 2018/01/01.

[83] European Parliamentary Research Service. How blockchain technology could change our lives. http://www.europarl.europa.eu/RegData/etudes/IDAN/2017/581948/EPRS_IDA(2017)581948_EN.pdf. Last accessed 2018/01/01.

[84] C. K. Frantz and M. Nowostawski. From institutions to code: towards automated generation of smart contracts. In *Workshop on Engineering Collective Adaptive Systems (eCAS)*, 2016.

[85] J. A. Garay, A. Kiayias, and N. Leonardos. The Bitcoin backbone protocol: Analysis and applications. In *EUROCRYPT*, volume 9057 of *LNCS*, pages 281–310. Springer, 2015.

[86] I. Gerhardt and T. Hanke. Homomorphic payment addresses and the pay-to-contract protocol. *arXiv preprint arXiv:1212.3257*, 2012.

[87] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun. On the security and performance of proof of work blockchains. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 3–16. ACM, 2016.

[88] F. Glaser, K. Zimmermann, M. Haferkorn, and M. C. Weber. Bitcoin - asset or currency? revealing users' hidden intentions. In *European Conference on Information Systems (ECIS)*, 2014.

[89] P. Grau. Lessons learned from making a chess game for Ethereum, 2016. https://medium.com/@graycoding/lessons-learned-from-making-a-chess-game-for-ethereum-6917c01178b6#.fwtdwly6e. Last accessed 2018/01/01.

[90] M. Harrigan and C. Fretter. The unreasonable effectiveness of address clustering. In *UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld*, pages 368–373. IEEE, 2016.

[91] H. Kalodner, S. Goldfeder, A. Chator, M. Möser, and A. Narayanan. Blocksci: Design and applications of a blockchain analysis platform. *arXiv preprint arXiv:1709.02489*, 2017.

[92] G. O. Karame, E. Androulaki, M. Roeschlin, A. Gervais, and S. Capkun. Misbehavior in Bitcoin: A study of double-spending and accountability. *ACM Trans. Inf. Syst. Secur.*, 18(1):2, 2015.

[93] A. E. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *IEEE Symp. on Security and Privacy*, pages 839–858, 2016.

[94] R. Kumaresan, T. Moran, and I. Bentov. How to use Bitcoin to play decentralized poker. In *ACM CCS*, pages 195–206, 2015.

[95] K. Liao and J. Katz. Incentivizing blockchain forks via whale transactions. In *Financial Cryptography Workshops*, volume 10323 of *LNCS*. Springer, 2017.

[96] K. Liao, Z. Zhao, A. Doupé, and G. Ahn. Behind closed doors: measurement and analysis of CryptoLocker ransoms in Bitcoin. In *APWG Symp. on Electronic Crime Research (eCrime)*, pages 1–13. IEEE, 2016.

[97] M. Lischke and B. Fabian. Analyzing the Bitcoin network: The first four years. *Future Internet*, 8(1):7, 2016.

[98] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *ACM CCS*, 2016. http://eprint.iacr.org/2016/633. Last accessed 2018/01/01.

[99] L. Luu, R. Saha, I. Parameshwaran, P. Saxena, and A. Hobor. On power splitting games in distributed computation: The case of Bitcoin pooled mining. In *IEEE Computer Security Foundations Symposium*, pages 397–411. IEEE, 2015.

[100] B. Marino and A. Juels. Setting standards for altering and undoing smart contracts. In *RuleML*, pages 151–166, 2016.

[101] P. McCorry, S. F. Shahandashti, and F. Hao. Refund attacks on Bitcoin's payment protocol. In *Financial Cryptography and Data Security*, volume 9603 of *LNCS*, pages 581–599. Springer, 2016.

[102] S. Meiklejohn, M. Pomarole, G. Jordan, K. Levchenko, D. McCoy, G. M. Voelker, and S. Savage. A fistful of bitcoins: characterizing payments among

men with no names. In *Internet Measurement Conference*, pages 127–140. ACM, 2013.

[103] S. Meiklejohn, M. Pomarole, G. Jordan, K. Levchenko, D. McCoy, G. M. Voelker, and S. Savage. A fistful of Bitcoins: characterizing payments among men with no names. *Commun. ACM*, 59(4):86–93, 2016.

[104] M. Möser and R. Böhme. Trends, tips, tolls: A longitudinal study of bitcoin transaction fees. In *International Conference on Financial Cryptography and Data Security*, pages 19–33. Springer, 2015.

[105] M. Möser and R. Böhme. Trends, tips, tolls: A longitudinal study of Bitcoin transaction fees. In *Financial Cryptography Workshops*, volume 8976 of *LNCS*, pages 19–33. Springer, 2015.

[106] M. Möser and R. Böhme. Anonymous alone? measuring bitcoin's second-generation anonymization techniques. In *EuroS&P Workshops*, pages 32–41, 2017.

[107] M. Möser, R. Böhme, and D. Breuker. An inquiry into money laundering tools in the Bitcoin ecosystem. In *APWG Symp. on Electronic Crime Research (eCrime)*, pages 1–14. IEEE, 2013.

[108] M. Möser, R. Böhme, and D. Breuker. Towards risk scoring of Bitcoin transactions. In *Financial Cryptography Workshops*, volume 8438 of *LNCS*, pages 16–32. Springer, 2014.

[109] S. Nakamoto. Bitcoin: a peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf. Last accessed 2018/01/01, 2008.

[110] Nomura Research Institute. Survey on blockchain technologies and related services. http://www.meti.go.jp/english/press/2016/pdf/0531_01f.pdf. Last accessed 2018/01/01.

[111] M. Ober, S. Katzenbeisser, and K. Hamacher. Structure and anonymity of the Bitcoin transaction graph. *Future Internet*, 5(2):237–250, 2013.

[112] G. Pappalardo, T. di Matteo, G. Caldarelli, and T. Aste. Blockchain inefficiency in the Bitcoin peers network. *CoRR*, abs/1704.01414, 2017.

[113] S. Popejoy. The Pact smart contract language, 2016. http://kadena.io/pact. Last accessed 2018/01/01.

[114] F. Reid and M. Harrigan. An analysis of anonymity in the Bitcoin system. In *Security and privacy in social networks*, pages 197–223. Springer, 2013.

[115] F. Reid and M. Harrigan. An analysis of anonymity in the Bitcoin system. In *Security and privacy in social networks*, pages 197–223. Springer, 2013.

[116] M. H. J. H. Z. D. M. O. H. Roman Matzutt, Jens Hiller and K. Wehrle. A quantitative analysis of the impact of arbitrary blockchain content on bitcoin. 2018.

[117] D. Ron and A. Shamir. Quantitative analysis of the full Bitcoin transaction graph. In *Financial Cryptography and Data Security*, volume 7859 of *LNCS*, pages 6–24. Springer, 2013.

[118] O. Schrijvers, J. Bonneau, D. Boneh, and T. Roughgarden. Incentive compatibility of Bitcoin mining pool reward functions. In *Financial Cryptography and Data Security*, volume 9603 of *LNCS*, pages 477–498. Springer, 2016.

[119] P. L. Seijas, S. Thompson, and D. McAdams. Scripting smart contracts for distributed ledger technology. Cryptology ePrint Archive, Report 2016/1156, 2016. http://eprint.iacr.org/2016/1156. Last accessed 2018/01/01.

[120] Y. Sompolinsky and A. Zohar. Secure high-rate transaction processing in bitcoin. In *Financial Cryptography and Data Security*, pages 507–527, 2015.

[121] M. Spagnuolo, F. Maggi, and S. Zanero. Bitiodine: Extracting intelligence from the Bitcoin network. In *Financial Cryptography and Data Security*, volume 8437 of *LNCS*, pages 457–468. Springer, 2014.

[122] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov. The first collision for full sha-1. https://shattered.io/static/shattered.pdf. Last accessed 2018/01/01.

[123] A. Sward, V. OP_0, and F. Stonedahl. Data insertion in bitcoin's blockchain. 2017.

[124] A. Tomescu and S. Devadas. Catena: Efficient non-equivocation via bitcoin. In *IEEE Symp. on Security and Privacy*, 2017.

[125] UK Government Chief Scientific Adviser. Distributed ledger technology: beyond block chain. https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/492972/gs-16-1-distributed-ledger-technology.pdf. Last accessed 2018/01/01.

[126] M. Vasek and T. Moore. There's no free lunch, even using Bitcoin: Tracking the popularity and profits of virtual currency scams. In *Financial Cryptography and Data Security*, volume 8975 of *LNCS*, pages 44–61. Springer, 2015.

[127] M. Vasek, M. Thornton, and T. Moore. Empirical analysis of Denial-of-Service attacks in the Bitcoin ecosystem. In *Financial Cryptography Workshops*, volume 8438 of *LNCS*, pages 57–71. Springer, 2014.

[128] weekinethereum. Week in ethereum news - november 8, 2017. http://www.weekinethereum.com/post/167279242888/november-8-2017. Last accessed 2018/01/01.

[129] G. Wood. Ethereum: a secure decentralised generalised transaction ledger, 2014. http://gavwood.com/paper.pdf. Last accessed 2018/01/01.

Appendix A

Identifiers

Table A.1 shows the list of identifiers that our tool exploits for associating metadata to protocols. We obtained this list by manually inspecting each protocol website. Note that Counterparty metadata must be first deobfuscated with ARC4 encryption using the transaction identifier (TXID) of the first unspent transaction output (UTXO) as the encryption key https://counterparty.io/docs/protocol_specification/

Category	Protocol	Identifiers
Financial	Colu	CC
	CoinSpark	SPK
	OpenAssets	OA
	Omni	omni
	Openchain	OC
	Helperbit	HB
	Counterparty	CNTRPRTY
Notary	Factom	Factom!!, FACTOM00, Fa, FA
	Stampery[42]	S1, S2, S3, S4, S5, S6
	Proof of Existence	DOCPROOF
	Blocksign	BS
	CryptoCopyright	CryptoTests-, CryptoProof-
	Stampd	STAMPD##
	BitProof	BITPROOF
	ProveBit	ProveBit
	Remembr	RMBd, RMBe
	OriginalMy	ORIGMY
	LaPreuve	LaPreuve
	Nicosia	UNicDC
	SmartBit	SB.D
	Notary	Notary
	BitcoinTimestamp	N/A
	Blocknotary	N/A
	Tangible	N/A
	Chainpoint	N/A
	Diploma	N/A
	Apertus	N/A
	Chronobit	N/A
	Seclytics	N/A
Arts	Monegraph	MG
	Blockai	0x1f00
	Ascribe	ASCRIBE
	Verisart	N/A
Messages	Eternity Wall	EW
	Cryptograffiti	N/A
	BIT-COMM	N/A
	Stone	N/A
	Key.run	N/A
	BitAlias	BALI
Subchains	Keybase	N/A
	Uniquebits	N/A
	Blockstore	id, 0x5888, 0x5808
	Catena[124]	N/A

Table A.1: Protocols identifiers.