# Reasoning about the Future in Blockchain Databases

Sara Cohen
The Rachel and Selim Benin School of Computer Science and Engineering
The Hebrew University of Jerusalem
Jerusalem, Israel
sara@cs.huji.ac.il

Adam Rosenthal
The Rachel and Selim Benin School of Computer Science and Engineering
The Hebrew University of Jerusalem
Jerusalem, Israel
adamrosenthal2@gmail.com

Aviv Zohar
The Rachel and Selim Benin School of Computer Science and Engineering
The Hebrew University of Jerusalem
Jerusalem, Israel
avivz@cs.huji.ac.il

## ABSTRACT

Modern blockchain systems are a fresh look at the paradigm of distributed computing, applied under new assumptions of large-scale open public networks. They can be used to store and share information without a trusted central party. There has been much theoretical and practical effort to develop blockchain systems for a myriad of uses, ranging from cryptocurrencies to identity control, supply chain management, clearing and settlement, and more. However, none of this work has directly studied the many fundamental database-related issues that arise when using blockchains as the underlying infrastructure to store and manage data.

A key difference between using blockchains to store data and centrally controlled databases is that transactions are accepted to a blockchain via a consensus mechanism, and not by a controlling central party. Hence, once a user has issued a transaction, she cannot be certain if it will be accepted. Moreover, a yet unaccepted transaction cannot be retracted by the user, and may be appended to the blockchain at any point in the future. This causes difficulties as the user may wish to reissue a transaction, if it was not accepted. Yet this data may then be appended twice to the blockchain.

We introduce a formal abstraction for blockchains as a data storage layer that underlies a database. The main issue that we tackle is the need to reason about possible worlds, due to the inherent uncertainty in transaction appending. In particular, we consider the theoretical complexity of determining whether it is possible for a denial constraint to be contradicted, given the current state of the blockchain, pending transactions, and integrity constraints on blockchain data. We then present practical algorithms for this problem, and show experimentally over the blockchain that our algorithms work well in practice.

**PVLDB Reference Format:**
xxx. xxx. *PVLDB*, 12(xxx): xxxx-yyyy, 2019.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx


*Proceedings of the VLDB Endowment,* Vol. 12, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

## 1. INTRODUCTION

Blockchains are becoming increasingly popular as a foundation for systems that share information in a trustworthy way, but without a trusted central party. Among other uses, they form the basis for permissionless open systems in which participation is unregulated.

Bitcoin,[1] created by Satoshi Nakamoto [32], is perhaps the prominent example of a permissionless blockchain system. It allows anyone to hold and transfer funds. To do so, rather than having a central server on which balances are kept, it decentralizes the management of funds between the many nodes in the Bitcoin network. Each node holds a copy of all transactions made within the blockchain. Thus, the Bitcoin network, as well as other cryptocurrency networks, serve, de-facto, as *fully decentralized financial database systems.*

Bitcoin's appearance inspired the use of similar techniques to synchronize other forms of information in other systems, but also brought a resurgence of interest in decentralizing existing systems, in the enterprise setting. Such proposed decentralized database systems have been touted as solutions to problems such as digital ownership, tracking legal flow of goods (such as diamonds or automobiles), making credit histories open and accessible, and smart utility monitoring without centralized control.

While it is not yet clear how many of these use cases will be based on blockchains in the long run, both the systems-oriented and theory-oriented database communities have begun to recognize the need to study the use of blockchains as an underlying data structure, as demonstrated by tutorials in [28,31,33]. Indeed one of the determining factors in adoption of blockchains as a data storage mechanism is likely to be the ability to overcome some of the inherent database issues arising in such systems.

While there is a myriad of previous work on blockchains and consensus mechanisms (discussed briefly in Section 3), no work has specifically studied key differences in blockchain storage, from other forms of data storage, on the conceptual level. In this paper we take the first step in this direction. Our aim is to explore some of the foundations of blockchains as a storage level in a manner that generalizes across different protocols and consensus mechanisms. Conceptually, blockchain systems consist of a consensus layer that coordinates data synchronization between nodes (often in adver-

[1]We use the standard uppercase Bitcoin when referring to the system, and the lowercase bitcoin when referring to units of the currency.

sarial settings like Byzantine Fault Tolerance), and the data consistency layer that determines what are the legal states of the data and how it may evolve. In this paper we focus on the data consistency layer, and on the ramifications of the consensus layer on the possible states of the data.

*Differences from classical database systems.* Typically, a blockchain is an append-only data structure, which is essentially a serialized record of accepted entries. Each individual block is an ordered batch of such updates that were committed together. Updates are added through a consensus mechanism in which all participants continually construct new blocks and accept them into the chain.

This interaction is quite different from traditional database systems. While traditional systems may employ sharding or distribute the load of processing the data, they are still essentially keeping information under the control of a *single*, possibly distributed, software system, which can be considered to be a centralized entity. In blockchains, the main assumption is that participants are separate entities that are not controlled by the same administrator, and that records are accepted by consensus mechanisms.

Centralized entities are more efficient than blockchain systems, but in exchange are highly susceptible to the failure or corruption of that entity. When only a single entity is in charge of record keeping, users may have their records erased, or altered without their permission. In currency systems, such interventions amount to freezing assets, or the seizure of funds without permission from their owner. Mainly for this reason, cryptocurrencies are implemented in a decentralized manner.

While updates to the blockchain become irreversible once committed (at least with very high probability), there is no guarantee that a certain update will indeed be accepted.[2] Rules on the consistency of the database and the validity of transactions, that are applied by every individual node, effectively govern what enters the blockchain, in addition to external considerations that may affect nodes (such as fees paid to include certain transactions).

Users that wish to add a record to the blockchain must create (and possibly sign) an appropriate message. The message is sent to all the nodes that participate in the consensus mechanism for consideration. In this way, all participants in the system have knowledge of pending transactions.

Miners choose transactions to include in a new block, typically while trying to maximize the transaction fees. However, it is intractable to determine an optimal set of transactions to be included, as this is a constrained version of the knapsack problem: blocks have a maximum length; transactions have varying lengths and fees. In addition, the inclusion of some transactions may be dependent on whether certain other transactions are, or are not, in the new block (due to integrity constraints), further complicating the problem. For some miners, the timing of a transaction may also affect its likelihood to be included.

Taken together, it is not possible to determine, a priori, when and whether a transaction will be included in the blockchain, in *any* blockchain system available currently. Even an old record that was created and propagated to other nodes by its original creator may be suddenly added to the blockchain much later on. Unlike a typical database system, there is no method to retract a transaction, once it has been issued.[3] Users thus have a high level of uncertainty regarding transactions that have yet to be accepted.

*A motivating example.* As a simple motivating example, we will consider a Bitcoin exchange that regularly accepts and issues payments in bitcoins in return for dollars or euros. Each payment that is issued by the exchange requires a miner's fee that is paid out to the node that approves the transaction and includes it inside a Bitcoin block. Fees form a part of the transaction message, and as such must be specified when the transaction is prepared.

Once the exchange broadcasts the transaction to the network, it must hope that it will be included in the blockchain. Unfortunately, since fees in the network can often fluctuate (depending on competition for limited space in Bitcoin blocks) it is quite possible for transactions not to be accepted by the network, or to get delayed for extended periods. In this case, the exchange would still be interested in issuing the transaction, perhaps with an increased fee the second time around. Alas, once two transaction messages are broadcast to the network, it is again quite possible that both the new and the old message will be included in the blockchain. Once signed, a transaction can be rebroadcast to the network by anyone, and is in fact often stored by nodes, with the hope of later inclusion in the blockchain. The unfortunate consequence of two such transactions making it to the blockchain is that the exchange would then pay its customer twice the amount it had intended.

The design of Bitcoin transactions does include a remedy for this situation: two transactions can be made to conflict in such a way as to rule out their co-existence in the blockchain. Careful practices by the exchange would require that reissued transactions be set up in such a manner. In Section 2 we go into more details and explain the structure of Bitcoin transactions and the validity rules that introduce such relations between them.

It is interesting to note that the problem we describe above is not merely hypothetical, but was actually used to attack exchanges in the past. Bitcoin transactions used to be somewhat malleable: one could change the contents of transactions in ways that would preserve their effects and maintain the validity of all cryptographic signatures. Attackers who withdrew funds from exchanges then rebroadcasted a changed version of their withdrawal transaction which was later accepted into the blockchain. Unfortunately, the exchanges did not notice, and reissued the payments. This resulted in withdrawals being issued more than once [15].

This example is specific to Bitcoin, but points to a deeper issue that arises when using a blockchain to store data. Uncertainty as to which transactions will be committed may lead to different possible worlds. It is critical to ascertain that there are no possibilities for undesirable outcomes.

*Contributions.* Our main contributions are as follows.

- **Modeling:** We present an abstract model of databases using blockchains as a storage layer, called *blockchain*

[2]Some blockchain systems have guaranteed irreversibility, while others, like Bitcoin, do so probabilistically with probability approaching one as time proceeds.

[3]In practice, users can attempt to retract a transaction by issuing a more attractive contradicting transaction, e.g., one with higher fee. The notion of contradicting transactions for an arbitrary blockchain is precisely the topic of this paper.

*databases*. Our model captures the main properties of such databases, while being independent of the specific protocol and implementation of the blockchain. Since blockchains are already the underlying storage layer for cryptocurrencies, and are being extensively explored for other data storage scenarios, such a modelling is a first step towards studying the ramifications of using a blockchain as storage in a database.

- **Complexity:** We study the problem of determining whether a blockchain database can reach an undesirable state and provide complete complexity results. (A formal notion of desirability of outcomes is part of our model.)
- **Algorithms:** We present a novel algorithms to test, in a real system, whether a blockchain database can reach an undesirable state. Several optimizations are also presented, that significantly reduce runtime.
- **Experimentation:** Finally, we experimentally validate our algorithms over the Bitcoin blockchain with pending transactions and show that our algorithms are efficient and scalable.

Our paper is organized as follows. In Section 2, we review Bitcoin's blockchain and structure of transactions, to ground our work in an important real-world implementation. Next, in Section 3, we review related work pertaining to blockchains and to databases. We present the formal framework in Section 4, study the complexity of the denial constraint satisfaction problem in Section 5 and present algorithms for this problem in Section 6. Extensive experimentation is presented in Section 7 and Section 8 concludes.

## 2. BITCOIN'S BLOCKCHAIN AND THE STRUCTURE OF TRANSACTIONS

While the paper presents results that generalize to any blockchain database, due to its popularity, we use Bitcoin as our running example throughout the paper. In this section, we provide a brief overview of the Bitcoin protocol and its specific use of the blockchain. The reader is referred to other, more-detailed explanations [45] for more information, and to [1] for a discussion of blockchain consensus protocols from the viewpoint of foundations of distributed computing. The goal of this overview is to ground the discussion in concrete terms and relate the results to a real-world system.

The Bitcoin network is comprised of a set of nodes that together form a P2P network. The *mempool* of a node contains the set of yet unauthorized transactions. Nodes in the network authorize Bitcoin transactions and include them in batches called blocks. Each block is a collection of transactions that transfer the bitcoin currency between different addresses. Addresses are effectively public keys that are associated with the money being transfered to denote ownership.

Each block additionally contains a cryptographic hash of a predecessor block. Blocks are thus arranged in a chain. The first block that was created at the inception of the system is known as the Genesis Block. The act of creation of blocks requires a difficult computation (proof-of-work). It rewards the creator of the block with bitcoins collected as fees from each transaction, as well as bitcoins that are minted at a predetermined rate. The act of block creation is known as *mining*, and nodes that invest the computational effort to try and create blocks are referred to as *miners*.

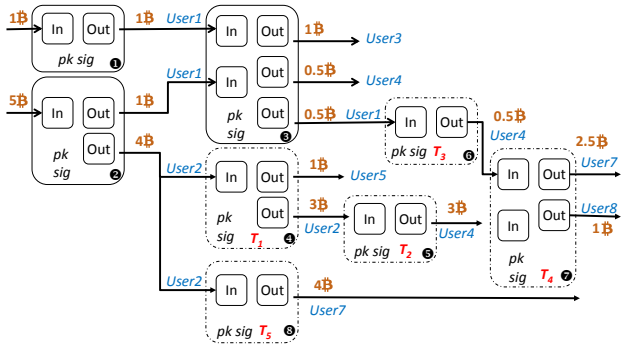



Figure 1: Structure of some Bitcoin transactions.

As blocks can be created concurrently, slightly different versions of the chain may be held by different nodes. Miners reach consensus on the blockchain by selecting to adopt the chain with the most accumulated proof-of-work. They quickly distribute newly found blocks to one another as well as transaction messages that are meant for inclusion in blocks via a gossip protocol. Conflicting transactions, or blocks that are off the main chain, are not propagated and are immediately discarded.

A transaction in Bitcoin is constructed as a transfer of funds from inputs to outputs (many-to-many). Outputs are essentially an association between an amount of bitcoins and a script that specifies how this money is to be claimed. The typical script in Bitcoin requires the spender to present a valid cryptographic signature in order to spend funds, but other scripts are also possible, e.g., requiring a preimage to a cryptographic hash to free funds, or several signatures matching different public keys. The inputs of each transaction essentially point to previously accepted outputs, and provide the necessary response to the challenge posed by the output scripts that are needed to spend the money. Funds are taken from the inputs and redistributed to the output of a transaction.

A Bitcoin transaction is considered to fully spend all inputs that it uses. Two transactions that share even a single input are thus considered conflicting transactions and cannot be accepted into the blockchain together. Transactions that spend the outputs of previous ones are dependent upon these parent transactions and cannot be included without first including the parent transactions. Figure 1 shows an example of Bitcoin transactions, along with their inputs and outputs. For now, the dotted lines around some of the transactions can be ignored.

## 3. RELATED WORK

This paper touches on two areas, namely *(1)* blockchains (and bitcoin, in particular) and *(2)* uncertain and inconsistent databases. We discuss related work in each of these areas next.

*Blockchains and Bitcoin.* There has been significant research on blockchains in general, and on Bitcoin in particular. For example, recent work has studied susceptibility of Bitcoin to selfish mining attacks (in which users selectively delay publishing data) [17,37], protocols to incentivize

data propagation [6], to ensure correct incentives in mining pools [27, 36], and modifications to the consensus protocol that will enable better scalability [16, 20, 40, 41].

The structure of transactions in Bitcoin allows for deanonymization and analysis [34, 35], which lead to several works that improved the privacy of cryptocurrencies [14, 26, 30, 38]. The security of the protocol to double-spending attacks was widely researched [19, 23, 42]. Other works have dealt with the susceptibility of Bitcoin to networking attacks [3, 22]. However, no work to date has taken a higher level view of a blockchain as a storage layer, in which the implementation details of the mechanisms are abstracted away, and the fundamental querying problems are given full focus.

*Uncertain and Inconsistent Databases.* Conceptually, a a database using blockchains as a storage layer, consists of a current state (i.e., a set of relations, stored in the blockchain), integrity constraints that must hold in every state, and a set of append-only transactions that have been issued by users, but have not been incorporated (yet) into the blockchain. Thus, a blockchain database is a succinct representation of a large number of possible worlds. Each possible world is a consistent set of relations, including the current state, and perhaps the results of additional transactions from among those issued. Possible worlds are not necessarily *maximal*, i.e., they may not contain some of the issued transactions, even though these transactions can be included while preserving consistency w.r.t. the integrity constraints.

Previous work considered probabilistic databases [10, 13, 39] and inconsistent databases [5, 7, 8, 11, 18, 25, 29, 43], both of which represent many possible worlds in a succinct manner. Querying is a challenge in both settings, as in the former setting, answers must be returned with their associated probabilities, and in the latter, only certain answers should be returned.

While a blockchain database is uncertain (as one cannot be sure which transactions will be accepted), it differs from a probabilistic database. Notably, it is not clear how to realistically associate probabilities with transactions, as miners can decide which transactions to try to include in a block, based on many considerations. The results in this paper are somewhat in the spirit of previous work on inconsistent databases. However, the setting is significantly different, as are the problems studied, and thus, previous results cannot be used. First and foremost, a blockchain database is always consistent. Moreover, while repairs (for inconsistent databases) are typically maximally close to the given data, in some sense, possible worlds for blockchains need not be maximal in any sense.

# 4. FORMAL FRAMEWORK

A *relation*[4] $R(A_1, \ldots, A_n)$ is associated with a set of ground tuples of arity $n$. Given a set of relations $\mathcal{R}$, an *insert transaction* (or simply, *transaction*, for short) for $\mathcal{R}$ is a set $T$ of ground tuples for (some of) the relations in $\mathcal{R}$. Intuitively, $T$ is a set of tuples that we would like to insert into the relations of $\mathcal{R}$. We consider only insert transactions, as blockchain databases are append-only databases.

We consider three types of integrity constraints, namely, *key constraints*, *functional dependencies* and *inclusion dependencies*. Let $R(\bar{A})$ be a relation, where $\bar{A}$ is a sequence of attributes. A *functional dependency* over $R$ has the form $X \to Y$, where $X, Y \subseteq \bar{A}$. We say that this dependency is satisfied by $R$, if, for every two tuples $t$ and $s$ in $R$, whenever $t[X] = s[X]$, also $t[Y] = s[Y]$. A functional dependency is a *key constraint* if $Y = \bar{A}$. Thus, key constraints are a special case of functional dependencies. Finally, let $R'(\bar{B})$ be a relation, $X \subseteq \bar{A}$, $Y \subseteq \bar{B}$. An *inclusion dependency* has the form $R[X] \subseteq R'[Y]$ and is satisfied by $R$ and $R'$ if, for every tuple $t \in R$, there exists a tuple $t' \in R'$ such that $t[X] = t'[Y]$. Satisfaction of a set of integrity constraints $\mathcal{I}$ by a set of relations $\mathcal{R}$ is defined in the standard fashion, and is denoted $\mathcal{R} \models \mathcal{I}$.

EXAMPLE 1. *Consider a database containing a simplified version of bitcoin data, with two relations describing transaction outputs and inputs:*

$$\mathsf{TxOut}(\underline{\mathsf{txId}, \mathsf{ser}}, \mathsf{pk}, \mathsf{amount})$$

$$\mathsf{TxIn}(\underline{\mathsf{prevTxId}, \mathsf{prevSer}}, \mathsf{pk}, \mathsf{amount}, \mathsf{newTxId}, \mathsf{sig}).$$

*In* TxOut*, attribute* txId *is a transaction identifier,* ser *is the serial number of the particular transaction output (of which there may be several),* pk *is the public key of the entity receiving this particular output and* amount *is the number of bitcoins in the output. In* TxIn*, the values* prevTxId, prevSer, pk, amount *indicate the transaction number, serial number, public key and amount of the input being consumed. Attribute* newTxId *is the new transaction identifier, and* sig *is a cryptographic signature corresponding to the public key of the entity whose output is being consumed.*

*The keys of both relations are underlined. We consider the inclusion dependencies*

$$\mathsf{TxIn}[\mathsf{prevTxId}, \mathsf{prevSer}, \mathsf{pk}, \mathsf{amount}] \subseteq \mathsf{TxOut}[\mathsf{txId}, \mathsf{ser}, \mathsf{pk}, \mathsf{amount}],$$
$$\mathsf{TxIn}[\mathsf{newTxId}] \subseteq \mathsf{TxOut}[\mathsf{txId}]$$

*i.e., every input consumed was created as the output of some transaction, and every new transaction has outputs.* □

We now formally define the notion of a *blockchain database*. A *blockchain database* $\mathcal{D}$ is a triple $(\mathcal{R}, \mathcal{I}, \mathcal{T})$, where

- $\mathcal{R}$ is a set of relations, called the *current state,*
- $\mathcal{I}$ is a set of integrity constraints, such that $\mathcal{R} \models \mathcal{I}$,
- $\mathcal{T} = \{T_1, \ldots, T_k\}$ is a finite set of transactions for $\mathcal{R}$, each of which is a set of tuples.

EXAMPLE 2. *Consider again the diagram from Figure 1 (depicting financial transactions). The transactions that have been issued and not yet been accepted into the blockchain are denoted as blocks with a dotted outline. Figure 2 contains an instance of a blockchain database, with the schema from Example 1, reflecting the contents of the diagram. In each table, the first column indicates whether the tuples belong to the current state $\mathcal{R}$, or to a transaction $T_i$.* □

The transactions in $\mathcal{T}$ may be appended to the database. However, it is also possible that they will not be appended, e.g., because they are not mutually consistent with $\mathcal{I}$, because of network problems, or simply due to lack of financial incentive.[5]

[4]By abuse of notation we use $R$ to denote both the schema of a relation, and its contents.

[5]In the real world, transactions are also associated with a fee that is accrued when it is added to the current state.

TxIn

| | prevTxId | prevSer | pk | amount | newTxId | sig |
|---|---|---|---|---|---|---|
| $\mathcal{R}$ | 1 | 1 | U1Pk | 1 | 3 | U1Sig |
| | 2 | 1 | U1Pk | 1 | 3 | U1Sig |
| $T_1$ | 2 | 2 | U2Pk | 4 | 4 | U2Sig |
| $T_2$ | 4 | 2 | U2Pk | 3 | 5 | U2Sig |
| $T_3$ | 3 | 3 | U1Pk | 0.5 | 6 | U1Sig |
| $T_4$ | 6 | 1 | U4Pk | 0.5 | 7 | U4Sig |
| | 5 | 1 | U4Pk | 3 | 7 | U4Sig |
| $T_5$ | 2 | 2 | U2Pk | 4 | 8 | U2Sig |

TxOut

| | txId | ser | pk | amount |
|---|---|---|---|---|
| $\mathcal{R}$ | 1 | 1 | U1Pk | 1 |
| | 2 | 1 | U1Pk | 1 |
| | 2 | 2 | U2Pk | 4 |
| | 3 | 1 | U3Pk | 1 |
| | 3 | 2 | U4Pk | 0.5 |
| | 3 | 3 | U1Pk | 0.5 |
| $T_1$ | 4 | 1 | U5Pk | 1 |
| | 4 | 2 | U2Pk | 3 |
| $T_2$ | 5 | 1 | U4Pk | 3 |
| $T_3$ | 6 | 1 | U4Pk | 0.5 |
| $T_4$ | 7 | 1 | U7Pk | 2.5 |
| | 7 | 2 | U8Pk | 1 |
| $T_5$ | 8 | 1 | U7Pk | 4 |

Figure 2: Running example of a simplified Bitcoin blockchain database.

We define the *can-append relationship* $\mathcal{R} \rightarrow_{\mathcal{T},\mathcal{I}} \mathcal{R}'$ for $\mathcal{D}$, as follows. We write $\mathcal{R} \rightarrow_{\mathcal{T},\mathcal{I}} \mathcal{R}'$ if $\mathcal{R}' = \mathcal{R}$ or there is some $T \in \mathcal{T}$ such that $\mathcal{R}' = \mathcal{R} \cup T$ and $\mathcal{R}' \models \mathcal{I}$. We denote the transitive closure of this relationship as $\mathcal{R} \Rightarrow_{\mathcal{T},\mathcal{I}} \mathcal{R}'$. Intuitively, this relationship defines a new instance of relations that can be derived by incrementally adding transactions from $\mathcal{T}$, while preserving all integrity constraints. We say that $\mathcal{R}'$ is a *possible world* for $\mathcal{D}$ if $\mathcal{R} \Rightarrow_{\mathcal{T},\mathcal{I}} \mathcal{R}'$, and denote the set of all possible worlds of $\mathcal{D}$ by $Poss(\mathcal{D})$.

Example 3. *Consider the blockchain database*

$$\mathcal{D} = (\mathcal{R}, \mathcal{I}, \{T_1, T_2, T_3, T_4, T_5\})$$

*depicted in Figure 2. Observe that transactions $T_1$ and $T_5$ are not mutually consistent, as including both would contradict the key constraint on* TxIn *(intuitively, corresponding to a double-spending of the same money). It is interesting to note that one of the outputs of $T_2$ is given to User2, who is the same user creating $T_2$. This accurately reflects the manner in which transactions are performed in Bitcoin, as users return to their own wallet the remainder of the input not being sent to another user. Also note that the transaction $T_4$ is dependent on $T_2$ and $T_3$ (due to the inclusion dependency), and $T_2$, in turn, is dependent on $T_1$.*

*Thus, $Poss(\mathcal{D})$ contains $\mathcal{R}$, $\mathcal{R} \cup T_1$, $\mathcal{R} \cup T_3$, $\mathcal{R} \cup T_1 \cup T_3$, $\mathcal{R} \cup T_1 \cup T_2$, $\mathcal{R} \cup T_1 \cup T_2 \cup T_3$, $\mathcal{R} \cup T_1 \cup T_2 \cup T_3 \cup T_4$, $\mathcal{R} \cup T_5$ and $\mathcal{R} \cup T_3 \cup T_5$.* □

Possible worlds can be efficiently recognized.[6] Proofs are omitted due to lack of space, but are available at [12].

Proposition 1. *Let $\mathcal{D}$ be a blockchain database and let $\mathcal{R}'$ be a set of relations. It is possible to determine if $\mathcal{R}' \in Poss(\mathcal{D})$ in PTIME.*

Remark 1. *In our model of a blockchain database we purposely retain only the essentials that are required to study the pivotal database issues at hand. Thus, for example, we ignore the incentives, consensus mechanisms and confirmation policies used while constructing blockchains. These are all orthogonal to our study, and also differ between various blockchain systems. We do not make explicit within $\mathcal{R}$ information about how tuples are distributed among the physical blockchain, as this is not of importance after data is accepted into the chain. (Transaction ordering is important, due to integrity constraints, while adding transactions, and this is specified in our model using the can-append relationship.)*

*We also do not consider forks in the chain, which may create temporary inconsistent versions of the database, for two main reasons. First, whether or not such forks can exist, and how they are resolved is system dependent. (For example, Algorand [21], Solida [2] and ByzCoin [24] do not allow forking.) Second, typically forks are resolved within a few seconds and, from a theoretical standpoint, such uncertain data is not considered as included in the database (yet).*

[6]In some systems it is possible that $\mathcal{T}$ is not fully known, as transactions are issued by multiple users concurrently. However, often transactions that have been issued are propagated among all nodes and appear in their mempools.

# 5. DENIAL CONSTRAINT SATISFACTION

Over time a blockchain database evolves by the addition of transactions to the current state. However, at any given point in time, it is not possible to know for certain which pending transactions in $\mathcal{T}$ will be permanently added to $\mathcal{R}$. One classical question that may be asked in this setting is as follows: *Given a query $q$ and a blockchain database $\mathcal{D}$, what are all certain answers to $q$ over $\mathcal{D}$?* Here certain answers are defined in the natural fashion (in the spirit of [4]), as tuples that will appear in the result over all possible worlds. In the setting of blockchain databases, it is not clear that this is of interest, e.g., for conjunctive queries, the set of certain answers is precisely the result of evaluating $q$ over $\mathcal{R}$.

Looking at this from the opposite perspective, recall that users often issue many transactions, and during this process they do not know for certain which previous transactions will or will not be appended to the current state. Hence, such a user may wish to ensure that in all possible worlds, specific undesirable things will not happen, by answering the following question: *Given a Boolean query $q$, does $q$ evaluate to false over all possible worlds of $\mathcal{D}$?* We call such a Boolean query, which we desire to remain unsatisfied, a *denial constraint*. Formally, a denial constraint $q$ is satisfied

by a blockchain database $\mathcal{D}$, denoted $\mathcal{D} \models \neg q$, if, for all $\mathcal{R}' \in Poss(\mathcal{D})$, it holds that $q(\mathcal{R}') = \text{false}$.

EXAMPLE 4. *Suppose Alice wishes to pay Bob one bitcoin (e.g., for a service Bob provided). Alice adds a transaction containing two tuples:*

1. *a tuple for* TxIn*, with information about her bitcoin that will be consumed,*[7] *and*
2. *a tuple for* TxOut *with the public key information of Bob.*

*After some time passes, Bob complains that he never received payment, i.e., that this transaction was not added to $\mathcal{R}$. Therefore, Alice sends a new transaction, again containing two tuples, to transfer money to Bob. Now, Alice may want to be certain that in all possible worlds, she has sent at most one bitcoin to Bob.*

*Assuming AlicePK is the public key of Alice, and BobPK is the public key of Bob, she may use the denial constraint*

$$\begin{aligned} q_1() \leftarrow & \mathsf{TxIn}(pt_1, ps_1, \text{'AlicePK'}, 1, ntx_1, \text{'AliceSig'}), \\ & \mathsf{TxOut}(ntx_1, ns_1, \text{'BobPK'}, 1), \\ & \mathsf{TxIn}(pt_2, ps_2, \text{'AlicePK'}, 1, ntx_2, \text{'AliceSig'}), \\ & \mathsf{TxOut}(ntx_2, ns_2, \text{'BobPK'}, 1), ntx_1 \neq ntx_2 \end{aligned}$$

*specifying that there are two different transactions in which Alice transferred money to Bob.*

*In practice, Alice may use this denial constraint over the hypothetical situation in which she has issued the second transaction (i.e., as a dry run,* before *actually issuing), in order to determine the safety of performing the second transaction.* □

Being able to determine whether a denial constraint is satisfied by a blockchain database is a fundamental issue, that allows users to interact with the database with some degree of clarity as to the possible downsides of their transactions. Hence, we formally define and study this problem. Let $\mathcal{D}$ be a blockchain database and $q$ be a denial constraint. The *denial constraint satisfaction problem* is to determine whether $\mathcal{D} \models \neg q$. We use the measure of data complexity (i.e., we assume the denial constraint is of constant size, while the size of the database is unbounded), as otherwise, query evaluation over a standard database is already intractable.

The complexity of the denial constraint satisfaction problem varies greatly, depending on the language of the denial constraints, as well as the types of integrity constraints allowed. In the following we use $\Delta$ to denote a subset of $\{key, fd, ind\}$ (standing for key, functional dependency, inclusion dependency). A blockchain database is *allowed by* $\Delta$, if it only contains integrity constraints of types appearing in $\Delta$.

Given a class of Boolean queries $\mathcal{Q}$, and a set $\Delta$ of integrity constraints, we use $\text{DCSAT}(\mathcal{Q}, \Delta)$ to denote the denial constraint satisfaction problem for the given class of queries and types of integrity constraints. Formally, $\text{DCSAT}(\mathcal{Q}, \Delta)$ is in complexity class $\mathcal{C}$ if, for all $q \in \mathcal{Q}$ and for all blockchain databases $\mathcal{D}$ allowed by $\Delta$, it is possible to determine whether $\mathcal{D} \models \neg q$ in time $\mathcal{C}$.

[7]This, in turn, must be an output from some previous transaction in which Alice received this coin.

We note in passing that all hardness results for functional dependencies already hold when there are only key constraints, and that all hardness results for inclusion dependencies already hold even when there is a single inclusion dependency.

***Classes of Queries.*** We consider several classes of queries, defined next. A *conjunctive query* has the form $q() \leftarrow P, N, C$ where $P$ is a conjunction of positive relational atoms, $N$ is a conjunction of negated relational atoms and $C$ is a conjunction of comparisons of variables to one another or to constants (using $=, <, >$, or $\neq$). We assume that all queries are *safe*, i.e., every variable appears in a positive relational atom. A conjunctive query is *positive* if it does not contain any negated relational atoms.

Given a set of relations $\mathcal{R}$, the query $q$ returns true if there is a *satisfying assignment* $h$ of the variables in the body of $q$ to constants in $\mathcal{R}$ such that *(1)* for all positive relational atoms $a$, $h(a) \in \mathcal{R}$, *(2)* for all negated relational atoms $a$, $h(a) \notin \mathcal{R}$ and *(3)* $h(C)$ is satisfied. Otherwise, $q$ returns false. We use $q(\mathcal{R})$ to denote the result of $q$ on $\mathcal{R}$. We use $\mathcal{Q}_c$ to denote the class of conjunctive queries and $\mathcal{Q}_c^+$ to denote the subclass containing only positive queries.

An *aggregate function* $\alpha(\bar{x})$, where $\bar{x}$ is an $n$-ary tuple of variables, is applied to a bag of $n$-ary tuples of values, and returns a single value.[8] An *aggregate query* has the form $[q(\alpha(\bar{x})) \leftarrow P, N, C]\, \theta\, c$ where $\theta \in \{=, <, >\}$ is a comparison and $c$ is a constant.

Given a set of relations $\mathcal{R}$, let $H$ be the set of all satisfying assignments of the body of $q$ into $\mathcal{R}$. Let $B$ be the bag $\{\!\{h(\bar{x}) \mid h \in H\}\!\}$. Then, the query $q$ returns true if and only if $\alpha(B)\ \theta\ c$ is true. For the special case in which $B$ is empty, we consider $\alpha(B)\ \theta\ c$ to be false.[9] The syntax and semantics of the aggregate functions considered in this paper are defined in the standard fashion, extending those of conjunctive queries. We use $\mathcal{Q}_\alpha$ to denote the class of all aggregate queries using $\alpha$, and $\mathcal{Q}_\alpha^+$ to denote its positive subclass. Finally, we use $\mathcal{Q}_{\alpha,\theta}$ and $\mathcal{Q}_{\alpha,\theta}^+$ to denote the subclasses of $\mathcal{Q}_\alpha$ and $\mathcal{Q}_\alpha^+$, respectively, in which the comparison operator is $\theta$.

EXAMPLE 5. *The denial constraint $q_1$ in Example 4 is a positive conjunctive query. We demonstrate additional denial constraints. Suppose the relation* Trusted(pk) *contains a list of private keys of trustworthy individuals. Then $q_2$, ensuring that all bitcoins from Alice are given to trusted individuals, is a conjunctive query, but is not positive.*

$$\begin{aligned} q_2() \leftarrow & \mathsf{TxIn}(pt, ps, \text{'AlcPK'}, a, ntx, \text{'AlcSig'}), \\ & \mathsf{TxOut}(ntx, s, pk, a'), \neg\mathsf{Trusted}(pk) \end{aligned}$$

*Denial constraint $q_3$ requires that Alice spent at most five bitcoins in total (as otherwise, it would return the value true)*

$$[q_3(\mathsf{sum}(a)) \leftarrow \mathsf{TxIn}(t, s, \text{'AlcPK'}, a, nt, \text{'AlcSig'})] > 5\,,$$

[8]We allow $n = 0$ to accommodate the aggregate function count, when applied to bags of empty tuples.

[9]This choice of semantics seems to be closest to the spirit of SQL. However, one can also choose to consider $\alpha(B)\ \theta\ c$ to be true in this case, or to determine the truth value based on $\theta$. Such changes in semantics will change some of the complexity results.

*while $q_4$ ensures that Alice participated in at most ten transactions in which bitcoins were given to Bob*

$$[q_4(\mathsf{cntd}(ntx)) \leftarrow \mathsf{TxIn}(pt, ps, \text{'AlcPK'}, a, ntx, \text{'AlcSig'}), \mathsf{TxOut}(ntx, s, \text{'BobPK'}, a')] > 10\,.$$

***Complexity.*** Before studying the complexity of the problem $\text{DCSat}(\mathcal{Q}, \Delta)$ for specific cases of $\mathcal{Q}$ and $\Delta$, we observe the following upper bound on the complexity of the problems at hand.

COROLLARY 1. $\text{DCSat}(\mathcal{Q}, \Delta)$ *is in CoNP.*

This result is a corollary of Proposition 1. To see this, suppose $q$ is a denial constraint and $\mathcal{D}$ is a blockchain database $(\mathcal{R}, \mathcal{I}, \mathcal{T})$. Given any subset $\mathcal{T}' \subseteq \mathcal{T}$ and an instance $\mathcal{R}' = \mathcal{R} \cup \mathcal{T}'$ it is possible to determine in PTIME whether $\mathcal{R}'$ is a possible world of $\mathcal{D}$ (Proposition 1). Since we assume that $q$ is of constant size, it is possible to determine in PTIME whether $q$ evaluates to true over $\mathcal{R}'$. This provides us with the upper bound.

The following theorem states the complexity of denial constraint satisfaction for different combinations of query classes and types of integrity constraints. We note that this theorem provides a full characterization for non-aggregate denial constraints, as the cases discussed clearly imply the complexity of the remaining cases. For example, it is easy to see that $\text{DCSat}(\mathcal{Q}_c, \{key, ind\})$ is CoNP-complete, since $\text{DCSat}(\mathcal{Q}_c^+, \{key, ind\})$ is CoNP-complete (by Theorem 1), and all problems are in CoNP (by Corollary 1).

THEOREM 1. *The denial constraint satisfaction problem has the following complexity:*

1. $\text{DCSat}(\mathcal{Q}_c, \{key, fd\})$ *and* $\text{DCSat}(\mathcal{Q}_c, \{ind\})$ *are in PTIME.*
2. $\text{DCSat}(\mathcal{Q}_c^+, \{key, ind\})$ *is CoNP-complete.*

We now consider query classes with aggregation. The complexity of the denial constraint satisfaction problem depends on the precise aggregate function considered. In the following theorem we provide a full characterization for the aggregate functions count, cntd (count distinct), sum and max. We note that the results for max can easily be used to determine the complexity for min.

THEOREM 2. *Let $\alpha$ be an aggregation function. Then,*

1. $\text{DCSat}(\mathcal{Q}_{\mathsf{max}}, \{key, fd\})$ *is in PTIME.*
2. $\text{DCSat}(\mathcal{Q}_{\alpha,<}, \{key, fd\})$, *with* $\alpha \in \{\mathsf{count}, \mathsf{cntd}, \mathsf{sum}\}$, *is in PTIME.*
3. $\text{DCSat}(\mathcal{Q}^+_{\alpha,\theta}, \{key\})$, *with* $\alpha \in \{\mathsf{count}, \mathsf{cntd}, \mathsf{sum}\}$, $\theta \in \{>, =\}$, *is CoNP-complete.*
4. $\text{DCSat}(\mathcal{Q}^+_{\alpha,>}, \{ind\})$, *with* $\alpha \in \{\mathsf{count}, \mathsf{cntd}, \mathsf{sum}, \mathsf{max}\}$, *is in PTIME.*
5. $\text{DCSat}(\mathcal{Q}^+_{\alpha,\theta}, \{ind\})$, *with* $\alpha \in \{\mathsf{count}, \mathsf{cntd}, \mathsf{sum}, \mathsf{max}\}$, $\theta \in \{<, =\}$, *is CoNP-complete.*
6. $\text{DCSat}(\mathcal{Q}_{\alpha,>}, \{ind\})$, *with* $\alpha \in \{\mathsf{count}, \mathsf{cntd}, \mathsf{sum}\}$, *is CoNP-complete.*
7. $\text{DCSat}(\mathcal{Q}_{\mathsf{max},>}, \{ind\})$ *is in PTIME.*
8. $\text{DCSat}(\mathcal{Q}^+_{\mathsf{max}}\{key, ind\})$ *is CoNP-complete.*

# 6. ALGORITHMS

From a practical standpoint, the complexity results of the previous section are of questionable usefulness. First, only limited cases are tractable. Indeed, denial constraint satisfaction is intractable for all query classes if databases include both key constraints and inclusion dependencies. This result is not satisfying, as *(1)* it is common to have both key constraints and inclusion dependencies and *(2)* notwithstanding the theoretical results, it is possible that in practice such worst case scenarios will not arise. Second, for the tractable cases, our proofs provide a polynomial time algorithm. However, careful observation shows that this algorithm is of time that is order of the database with an exponent in the size of the query. Since the query size is constant, this gives a theoretically polynomial time algorithm. However, in practice, such an algorithm is quite expensive.

In this section we present algorithms for the denial constraint satisfaction problem in the presence of functional dependencies (including key constraints) and inclusion dependencies, when the denial constraint is *monotonic*. We also show how to improve this algorithm for denial constraints that are *connected*. In Section 7 we experimentally show the efficiency of our algorithms for important cases.

## 6.1 Monotonic Denial Constraints

A Boolean query $q$ is *monotonic* if, for all sets of relations $\mathcal{R}$ and $\mathcal{R}'$, such that $\mathcal{R} \subseteq \mathcal{R}'$, if $q(\mathcal{R})$ holds then also $q(\mathcal{R}')$ holds. Intuitively, for monotonic denial constraints it is sufficient to consider possible worlds that are maximal.

In the following, let $\mathcal{T}$ be a set of transactions and $\mathcal{I}_{fd}$ be the functional dependencies in the blockchain database. The *fd-transaction graph* $G^{fd}_{\mathcal{T}}$ has a node for each $T \in \mathcal{T}$ and contains an edge $(T, T')$ if $T \cup T' \models \mathcal{I}_{fd}$. For example, Figure 3 contains the graph $G^{fd}_{\mathcal{T}}$ for the running example of Figure 2.

Note that every possible world must correspond to a clique in $G^{fd}_{\mathcal{T}}$, as all pairs of transactions must be mutually consistent with respect to $\mathcal{I}_{fd}$ (but the opposite may not hold, due to inclusion dependencies). Moreover, since we only consider denial constraints that are monotonic, it is sufficient to consider sets of transactions that form maximal cliques in $G^{fd}_{\mathcal{T}}$. (A set of nodes $\mathcal{T}' \subseteq \mathcal{T}$ is a *maximal clique* if it is a clique, and there is no strictly containing $\mathcal{T}' \subset \mathcal{T}'' \subseteq \mathcal{T}$ for which $\mathcal{T}''$ is a clique.)

The algorithm NAIVEDCSAT in Figure 4 can be used to determine satisfaction of a monotonic denial constraint. Observe that once we restrict ourselves to sets of transactions $\mathcal{T}' \subseteq \mathcal{T}$ that are cliques in $G^{fd}_{\mathcal{T}}$, for each such clique there is a single maximal possible world over $(\mathcal{R}, \mathcal{I}, \mathcal{T}')$. This maximal possible world can be generated using the algorithm GETMAXIMAL. Thus, the algorithm NAIVEDCSAT simply iterates over all maximal possible worlds, and checks whether the denial constraint $q$ returns true over at least one such world. This algorithm is polynomial with respect to the number of maximal possible worlds, but this does not contradict our previous complexity results, as there can be exponentially many maximal possible worlds.

EXAMPLE 6. *Consider running algorithm* NAIVEDCSAT *over the blockchain database of Figure 2 with the denial constraint* $q_s() \leftarrow \mathsf{TxOut}(t, s, \text{'U8Pk'}, a)$. *There are two maximal cliques in the graph* $G^{fd}_{\mathcal{T}}$, *namely* $\{T_2, T_3, T_4, T_5\}$ *and* $\{T_1, T_2, T_3, T_4\}$. *For the former, the maximal world returned*

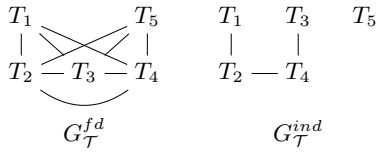



Figure 3: Precomputed data structures.

*by* GETMAXIMAL *is* $\mathcal{R} \cup \{T_3, T_5\}$ *and* $q_s$ *is false over this world. For the latter,* GETMAXIMAL *returns* $\mathcal{R} \cup \{T_1, T_2, T_3, T_4\}$ *over which* $q_s$ *evaluates to true. Hence, the denial constraint* $q_s$ *is not satisfied and* NAIVEDCSAT *returns false.*

## 6.2 Connected Monotonic Denial Constraints

For queries that are *connected* we can improve upon the algorithm NAIVEDCSAT. Formally, a query is *connected* if it is conjunctive (i.e., not an aggregate query) and the *Gaifman graph* of the query is connected. Recall that the nodes of the Gaifman graph for a query $q$ are the terms appearing in the relational atoms of $q$. There is an edge between $x$ and $y$ if $x$ and $y$ appear in the same relational atom. Thus, for example, $q() \leftarrow R(x, y), S(w, v), T(x, v)$ is connected, but $q() \leftarrow R(x, y), S(w, v), y < v$ is not.

The key idea is to divide up the set of pending transactions $\mathcal{T}$ into disjoint subsets that can be considered independently. The *ind-q-transaction graph* is used to divide transactions into subsets, such that, intuitively, transactions that may be used together to satisfy a query will be in the same subset. The idea behind this division is to leverage the fact that the query is connected. We utilize both inclusion dependencies and the connections between atoms in the query to create this graph, as explained next.

Let $R[\bar{A}]$ and $S[\bar{B}]$ be relations. An *equality constraint* $\theta$ is an expression of the form $R[\bar{X}] = S[\bar{Y}]$ where $\bar{X} \subseteq \bar{A}$ and $\bar{Y} \subseteq \bar{B}$. We say that $\theta$ is *satisfied* by a pair of tuples $t \in R, s \in S$ if $t[\bar{X}] = s[\bar{Y}]$, and $\theta$ is *satisfied* by a pair of transactions $T, T'$ if there are $t \in T, t' \in T'$ such that $\theta$ is satisfied by $t, t'$.

In the following, we use $\mathcal{I}_{ind}$ to denote the inclusion dependencies in $\mathcal{I}$. We derive equality constraints from two sources. First, every inclusion dependency $R[\bar{X}] \subseteq S[\bar{Y}] \in \mathcal{I}_{ind}$ gives rise to the equality constraint $R[\bar{X}] = S[\bar{Y}]$. We use $\Theta_{\mathcal{I}}$ to denote the set of all equality constraints implied by $\mathcal{I}_{ind}$.

Second, consider a query $q$ with positive atoms $R(\bar{x})$ and $S(\bar{y})$. Let $i_1, \ldots, i_k$ and $j_1, \ldots, j_k$ be the maximal sequence of distinct indices such that, (1) $i_1 < \cdots < i_k$ and (2) for all $1 \leq h \leq k$, the $i_h$-th variable in $\bar{x}$ is identical to the $j_h$-th variable in $\bar{y}$ (or they are implied to be equal by the comparisons). Then, we say that $R(\bar{x})$ and $S(\bar{y})$ *imply* the equality constraint $R[A_{i_1}, \ldots, A_{i_k}] = S[B_{j_1}, \ldots, B_{j_k}]$ where $A_{i_l}$ and $B_{j_l}$ are the $i_l$-th and $j_l$-th attributes in $R$ and $S$ respectively. We use $\Theta_q$ to denote the set of all equality constraints implied by pairs of atoms in the body of $q$.

EXAMPLE 7. *Consider the query*

$$q() \leftarrow R(w, x, u), S(x, w, z), T(y, x)$$

*over relations* $R(A_1, A_2, A_3)$, $S(B_1, B_2, B_3)$ *and* $T(C_1, C_2)$. *Then* $\Theta_q$ *contains the following equality constraints:*

$$R[A_1, A_2] = S[B_2, B_1] \quad R[A_2] = T[C_2] \quad S[B_1] = T[C_2]$$

Given $\mathcal{D} = (\mathcal{R}, \mathcal{I}, \mathcal{T})$ and a denial constraint $q$, let $\Theta = \Theta_{\mathcal{I}} \cup \Theta_q$. Now, the *ind-q-transaction graph* $G_{\mathcal{T}}^{q,ind}$ is over the set of nodes $\mathcal{T}$ and contains an edge $(T, T')$ if there is a $\theta \in \Theta$ that is satisfied by $T, T'$. It is not difficult to show the following result.

PROPOSITION 2. *Let* $q$ *be a connected query and* $\mathcal{D} = (\mathcal{R}, \mathcal{I}, \mathcal{T})$ *be a database. Let* $T, T' \in \mathcal{T}$ *be transactions in different connected components of* $G_{\mathcal{T}}^{q,ind}$. *Then, there is no satisfying assignment of* $q$ *over* $\mathcal{R} \cup \mathcal{T}$ *that maps atoms to tuples in both* $T$ *and* $T'$.

This proposition immediately implies that it is sufficient to check for satisfaction of denial constraints for each connected component of $G_{\mathcal{T}}^{q,ind}$, independently.

We also further improve our algorithm by considering constants appearing in the query. Such constants can be used to filter out the set of possible worlds that must be considered. Let $q$ be a query and $R(\bar{x})$ be a positive relational atom in $q$. Let $i_1, \ldots, i_k$ be a maximal sequence of distinct indices such that for all $1 \leq h \leq k$, the $i_h$-th argument in $\bar{x}$ is a constant. Then, we say that a relation $R(\bar{A})$ *covers the constants* from $R(\bar{x})$ if there is a tuple $t \in R$ such that $t[A_{i_1}, \ldots, A_{i_k}] = (x_{i_1}, \ldots, x_{i_k})$. Given a set of relations $\mathcal{R}$ and a set of transactions $\mathcal{T}$, we say that the pair $(\mathcal{R}, \mathcal{T})$ *covers the constants* in the atom $R(\bar{x})$ of $q$ if the set of ground tuples for $R$ in $\mathcal{R}$ and $\mathcal{T}$ together cover the constants from $R(\bar{x})$. The pair $(\mathcal{R}, \mathcal{T})$ covers all constants in $q$, denoted COVERS$(\mathcal{R}, \mathcal{T}, q)$ if it covers the constants of all atoms in the body of $q$. For example, $(\mathcal{R}, \{T_4\})$ covers all constants in $q_s() \leftarrow \mathsf{TxOut}(t, s, \text{'U8Pk'}, a)$.

The algorithm OPTDCSAT for determining satisfaction of connected, monotonic denial constraints appears in Figure 5. Intuitively, this algorithm iterates over all connected components in $G_{\mathcal{T}}^{q,ind}$ (Line 1), and considers only those components $\mathcal{T}'$ whose transactions can be used to cover the constants of $q$ (Line 2). Next, for each maximal clique $\mathcal{T}''$

**Algorithm** NAIVEDCSAT$(\mathcal{R}, \mathcal{I}, \mathcal{T}, q)$
1. **for** each maximal clique $\mathcal{T}'$ in $G_{\mathcal{T}}^{fd}$
2. **do** $\mathcal{R}' \leftarrow$ GETMAXIMAL$(\mathcal{R}, \mathcal{I}, \mathcal{T}')$
3. **if** $q(\mathcal{R}') =$ true
4. **then return** false
5. **return** true

**Algorithm** GETMAXIMAL$(\mathcal{R}, \mathcal{I}, \mathcal{T})$
1. $\mathcal{R}' \leftarrow \mathcal{R}$
2. *size* $\leftarrow 0$
3. **while** $\mathcal{T} \neq \emptyset$ and *size* $\neq |\mathcal{T}|$
4. **do** *size* $\leftarrow |\mathcal{T}|$
5. **for** each $T \in \mathcal{T}$
6. **do if** $\mathcal{R}' \cup T \models \mathcal{I}$
7. **then** $\mathcal{T} \leftarrow \mathcal{T} \setminus T$
8. $\mathcal{R}' \leftarrow \mathcal{R}' \cup T$
9. **return** $\mathcal{R}'$

Figure 4: Algorithm for denial constraint satisfaction, for denial constraints that are monotonic.

**Algorithm** OPTDCSAT($\mathcal{R}, \mathcal{I}, \mathcal{T}, q$)
1. **for** each connected component $\mathcal{T}'$ in $G^{q,ind}_{\mathcal{T}}$
2. **do if** COVERS($\mathcal{R}, \mathcal{T}', q$)
3. **then for** each maximal clique $\mathcal{T}''$ in $G^{fd}_{\mathcal{T}'}$
4. **do** $\mathcal{R}' \leftarrow$ GETMAXIMAL($\mathcal{R}, \mathcal{I}, \mathcal{T}''$)
5. **if** $q(\mathcal{R}')$ = true
6. **then return** false
7. **return** true

Figure 5: Algorithm for denial constraint satisfaction, for denial constraints that are monotonic and connected.

| $\mathcal{R}$ | Blocks | Transactions | Input | Output |
|---|---|---|---|---|
| D100 | 100,000 | 216,571 | 292,362 | 264,249 |
| D200 | 200,000 | 7,316,306 | 14,510,970 | 16,629,437 |
| D300 | 300,000 | 38,463,550 | 88,691,799 | 99,245,291 |
| $\mathcal{T}$ | Blocks | Transactions | Input | Output |
| D100 | 780 | 2,741 | 3,937 | 4,198 |
| D200 | 30 | 3,733 | 9,902 | 8,649 |
| D300 | 18 | 2,766 | 8,872 | 8,929 |

Table 1: Datasets

of $\mathcal{T}'$ (Line 3), we create the maximal possible world (using GETMAXIMAL), and check for satisfaction of the denial constraint $q$ (Lines 4–5). If $q$ evaluates to true over some possible world, we return false (Line 6), as the denial constraint is not satisfied in all possible worlds. Otherwise, we return true (Line 7).

EXAMPLE 8. *Consider again the denial constraint from Example 6. No equality constraints are implied by the denial constraint $q_s$, hence $G^{q,ind}_{\mathcal{T}}$ is determined entirely by the inclusion dependencies, and appears on the right in Figure 3.*

*When running* OPTDCSAT *with the denial constraint $q_s$ consdiered earlier, there are two connected components, only one of which covers the constants in $q_s$. Hence, we will only consider maximal possible worlds in the component $\{T_1, T_2, T_3, T_4\}$. Once again,* OPTDCSAT *will return false.*

We note that in our toy example, the difference between NAIVEDCSAT and OPTDCSAT may not be obvious. In practice, there are likely to be a very large number of connected components. Focusing on connected components individually significantly reduces the size of the possible worlds that are considered, and typically also reduces the number of maximal worlds that must be considered.

## 6.3 Implementation

The algorithms we described can be used for a blockchain database over any type of data. In our experimentation, we test our algorithms over Bitcoin data. Hence, in this section we briefly describe how OPTDCSAT is implemented within a Bitcoin system.[10] Implementation specific details (such as storing transaction identifiers) can be tailored to a different system, if desired. The algorithm is implemented at a Bitcoin node, and thus, has access both to the accepted transactions $\mathcal{R}$, and to the pending transactions $\mathcal{T}$.

The current contents of the blockchain $\mathcal{R}$ is stored in a relational database (Postgres). The transactions $\mathcal{T}$ are stored both within the relational database and in the main memory. In order to differentiate between tuples of $\mathcal{R}$ and tuples of $\mathcal{T}$ within the database, we augment each table in the database with a Boolean column current which indicates whether the tuple is currently part of the possible world being considered. In the steady state, tuples from $\mathcal{R}$ have the value "true" for this column, while tuples from $\mathcal{T}$ have the value "false".

Some information and data structures are stored and updated, in the steady state, as transactions are included into the blockchain (i.e., added to $\mathcal{R}$), and as new transactions are issued (i.e., added to $\mathcal{T}$). These structures allow denial constraint satisfaction to be checked more quickly.

In particular, we store the following information:

- For each $T \in \mathcal{T}$ we determine whether $T$ can be included in $\mathcal{R}$, i.e., if $\mathcal{R} \cup \{T\} \models \mathcal{I}$. Intuitively, this means that any inclusion dependencies related to $T$ can already be satisfied within $\mathcal{R}$. We store this status information. In our running example, these transactions are $T_1, T_3, T_5$.
- We precompute and store the graph $G^{fd}_{\mathcal{T}}$ containing edges for transactions that are mutually consistent with respect to the functional dependencies.
- We precompute and store the graph $G^{ind}_{\mathcal{T}}$ which contains the edges of $G^{q,ind}_{\mathcal{T}}$ that are derived from the inclusion dependencies. When the user would like to determine denial constraint satisfaction for a specific denial constraint $q$, we augment $G^{ind}_{\mathcal{T}}$ with the additional edges that are derived from $q$.

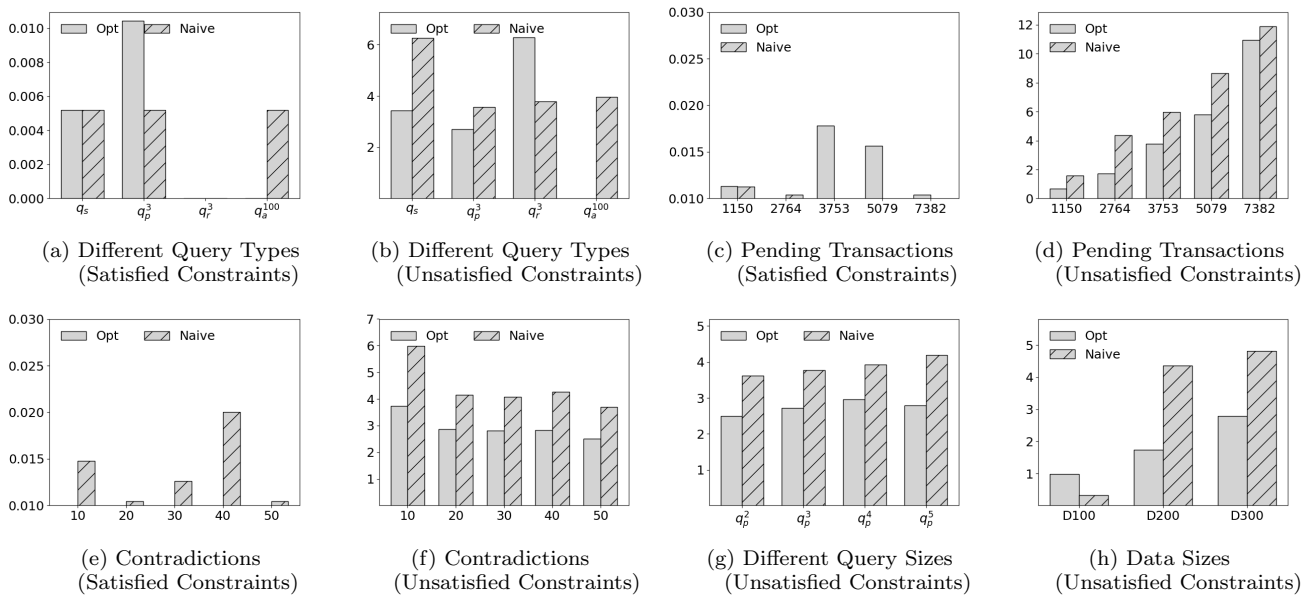
In our implementation of OPTDCSAT, we use the Bron-Kerbosch algorithm for maximal clique enumeration [9], with the pivoting optimization from [44]. One of the key operations in OPTDCSAT is to check, for a possible world $\mathcal{R}'$ whether $q(\mathcal{R}')$ returns true. In order to check this, we update the current column of each transaction in $\mathcal{R}'$ to be "true". We then check for satisfaction of $q$ in the database, with respect to tuples that have the value "true" in column current. Updating the set of current transactions, and query evaluation, is often a large portion of the runtime, if many possible worlds must be considered.

Finally, we incorporated an additional optimization. Since we only consider monotonic denial constraints, before running NAIVEDCSAT or OPTDCSAT, we first check if the denial constraint $q$ evaluates to true over $\mathcal{R} \cup \mathcal{T}$. If $q$ evaluates to false, we can be sure that it will be false on all possible worlds (as they are contained in $\mathcal{R} \cup \mathcal{T}$), i.e., the denial constraint is satisfied. If $q$ evaluates to true, we must fully run NAIVEDCSAT or OPTDCSAT, as $\mathcal{R} \cup \mathcal{T}$ is not actually (usually) a legitimate possible world.

# 7. EXPERIMENTATION

We implemented DCSAT and CONNDCSAT in Python 3. All experimentation was run on a standard Windows 10 laptop. We used Postgres for the underlying database. Due to the large size of the data, the database was run on a 1TB SSD external drive. In the following we discuss the data and experimental results.

[10] We do not discuss NAIVEDCSAT directly, as it can be seen as a special case of OPTDCSAT.




(a) Different Query Types (Satisfied Constraints)
(b) Different Query Types (Unsatisfied Constraints)
(c) Pending Transactions (Satisfied Constraints)
(d) Pending Transactions (Unsatisfied Constraints)
(e) Contradictions (Satisfied Constraints)
(f) Contradictions (Unsatisfied Constraints)
(g) Different Query Sizes (Unsatisfied Constraints)
(h) Data Sizes (Unsatisfied Constraints)

Figure 6: Execution time for determining satisfaction of denial constraints.

*Dataset and Denial Constraints.* We ran a Bitcoin node to get real Bitcoin data. The transaction data was parsed and inserted into tables in Postgres. Due to the large size of the blockchain and our limited resources, we used between 100,00 and 300,000 of the first Bitcoin blocks as the current state of the data, and then used subsequent blocks as our pending transactions. (Currently, Bitcoin has approximately 550,000 blocks.)

Precise details about our datasets appear in Table 1. For each dataset, we specify the number of blocks used, the number of transactions appearing in these blocks and the number of input and output rows produced for these transactions. In addition, we specify the size of the pending transactions used (again, with the same parameters).

We consider four types of denial constraints of the forms:

$$q_s() \leftarrow \mathsf{TxOut}(ntx, s, X, a)$$

$$q_p^3() \leftarrow \mathsf{TxOut}(ntx_1, s_1, X, a_1), \mathsf{TxIn}(ntx_1, s_1, pk_2, a_2, ntx_2, sig_2), \mathsf{TxOut}(ntx_2, s_2, pk_3, a_3), \mathsf{TxIn}(ntx_2, s_2, Y, a_3, ntx_4, sig_3)$$

$$q_r^3() \leftarrow \mathsf{TxIn}(pntx_1, s_1, X, a_1, ntx_1, sig_1), \mathsf{TxOut}(ntx_1, s_1, pk_1, a_2), \mathsf{TxIn}(pntx_2, s_2, X, a_2, ntx_2, sig_2), \mathsf{TxOut}(ntx_2, s_2, pk_2, a_4), \mathsf{TxIn}(pntx_3, s_3, X, a_3, ntx_3, sig_3), \mathsf{TxOut}(ntx_3, s_3, pk_3, a_6)$$
$$ntx_1 \neq ntx_2, ntx_1 \neq ntx_3, ntx_2 \neq ntx_3$$

$$[q_a^n(\mathsf{sum}(a)) \leftarrow \mathsf{TxOut}(ntx, s, X, a)] \geq n$$

where

- $q_s$ is a *simple denial constraint*, indicating that a particular address did receive bitcoins in a transaction;
- $q_p^i$ is a *path denial constraint* stating that there is no series of length $i$ of transactions transferring bitcoins;
- $q_r^i$ is a *star denial constraint* stating that a specific address did not transfer bitcoins to $i$ different addresses;
- $q_a^n$ is an *aggregation constraint* requiring that a specific address did not receive more than $n$ bitcoins.

In all of the queries we will instantiate $X$ or $Y$ with constants. These constants will sometimes be chosen so as to satisfy the underlying queries of the denial constraints (i.e., the denial constraint will not hold), or conversely, in a way that does not satisfy the underlying queries.

A key issue in the runtime is whether the denial constraints considered are satisfied. If they are, then we will often discover this already in the first step as they will return false over $\mathcal{R} \cup \mathcal{T}$. On the other hand, if the denial constraints are not satisfied (i.e., there is a world over which the underlying query returns true), we must iterate over all possible worlds until such a world is identified. Hence, these denial constraints will typically have longer runtime.

As will be apparent from the experimentation, our algorithms are very fast for satisfied denial constraints, running at sub-second speeds. For unsatisfied denial constraints, the runtime is significantly longer. Typically, our algorithms will be run when a user wishes to issue a transaction, and wants to determine that no bad outcome can occur. The user hypothetically adds her transaction and runs the algorithm. When the transaction can be safely issued (i.e., the denial constraint is satisfied), the user will usually be notified of this quickly. For denial constraints that are unsatisfied, the user will often need to wait longer. However, this should be acceptable as in such cases the user indeed should avoid issuing her transaction.

In all graphs the runtime is in seconds, and it is the average runtime of three executions. We note in passing that while our runtime is usually very low, it is possible that some denial constraints will take significantly longer—this is unavoidable as we are solving a problem that has been shown to be CoNP-complete.

In our experimentation, we studied the scalability of the system with respect to the data size, the query type and size, the number of pending transactions and the number of contradictions in the pending transactions. For each experiment we keep all parameters except one at their default

values, and vary the remaining parameter. As a default, we use the D200 dataset, with path queries of size three, 3,733 pending transactions and 20 functional dependency contradictions. (In practice few contradictions are seen within Bitcoin transactions.)

*Query Type.* We ran all four query types over D200, with both NaiveDCSat and OptDCSat for $q_s, q_p^3, q_r^3$ and only NaiveDCSat for $q_a^{100}$ (as this query is not connected). In Figure 6a and 6b we chose constants such that the denial constraints are satisfied and unsatisfied, respectively.

For the satisfied denial constraint, all runs complete in just a few milliseconds. In fact, some of the runtimes were so short that they are not visible in the graph. For unsatisfied denial constraints, runtime takes up to six seconds. For unsatisfied denial constraints usually the runtime of OptDCSat is significantly lower than that of NaiveDCSat, as the former considers much smaller possible worlds. Sometimes, however, the trend in runtime reverses, as algorithm NaiveDCSat may consider fewer possible worlds before it finds a satisfying assignment for underlying query (and thus, concluding that the denial constraint is not satisfied) when the worlds are larger. This is precisely what happens for $q_r^3$. The same phenomenon appears in some of the other graphs.

*Number of Pending Transactions.* We ran $q_p^3$ over D200 while varying the number of pending transactions. This is done by varying the number of additional Bitcoin blocks seen as pending. We consider 10, 20, 30, 40 and 50 blocks with 1150, 2764, 3753, 5079 and 7382 transactions, respectively. Figure 6c shows the runtime for satisfied denial constraints, while Figure 6d shows the runtime for unsatisfied denial constraints. For satisfied denial constraints, runtime remains sub-second, even when the number of transactions increases. The runtime of OptDCSat is consistently less than that of NaiveDCSat for unsatisfied denial constraints.

*Number of Contradictions.* We vary the number of contradictions to the functional dependencies to values between 10 and 50. Our results for satisfied and unsatisfied denial constraints appear in Figures 6e and 6f, respectively. Interestingly, runtime is actually greatest when there are only ten contradictions. This is because when there are fewer contradictions possible worlds are larger, and thus, updating the current column for included transactions takes longer.

*Query Size.* We vary the lengths of the path queries between two and five over D200, running both NaiveDCSat and OptDCSat, and choosing constants such that the denial constraints are not satisfied. (Due to space limitations, we omit satisfied denial constraints in this experiment and the next. However, they have sub-second runtimes.) Runtime only increases slightly as the query grows, since the query evaluation is only a very small proportion of the total runtime.

*Data Size.* Finally, we ran $q_p^3$ over datasets D100, D200 and D300 with both NaiveDCSat and OptDCSat. Additionaly, each dataset contains approximately 3000 pending transactions. The runtime results appear in Figure 6h. Runtime increases only moderately as the data size grows, with OptDCSat remaining signficantly faster than NaiveDCSat.

# 8. CONCLUSION

This paper formally defined a blockchain database, and introduced an important problem, namely denial constraint satisfaction. The ability to determine if a denial constraint will always hold is critical in deciding if a new transaction can be safely issued. We study the complexity of this problem, and propose algorithms that work well in practice, as shown by the experimentation.

As future work, we will consider the problem of scaling our system to the entire blockchain, perhaps using a distributed environment. We also intend to study problems such as how to automatically derive a new transaction that contradicts previous transactions, as well as denial constraint satisfaction when weighting possible worlds by learning an estimation of their actual likelihood.

# 9. REFERENCES


[1] I. Abraham and D. Malkhi. The blockchain consensus layer and BFT. *Bulletin of the EATCS*, 123, 2017.

[2] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and A. Spiegelman. Solidus: An incentive-compatible cryptocurrency based on permissionless byzantine consensus. *CoRR*, abs/1612.02916, 2016.

[3] M. Apostolaki, A. Zohar, and L. Vanbever. Hijacking bitcoin: Routing attacks on cryptocurrencies. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 375–392, 2017.

[4] M. Arenas, L. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *Proceedings of the Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '99, pages 68–79, New York, NY, USA, 1999. ACM.

[5] M. Arenas, L. E. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *SIGMOD*, pages 68–79, 1999.

[6] M. Babaioff, S. Dobzinski, S. Oren, and A. Zohar. On bitcoin and red balloons. In *Proceedings of the 13th ACM Conference on Electronic Commerce*, EC '12, pages 56–73, New York, NY, USA, 2012. ACM.

[7] L. E. Bertossi. Consistent query answering in databases. *SIGMOD Record*, 35(2):68–76, 2006.

[8] L. E. Bertossi and J. Chomicki. Query answering in inconsistent databases. In *Logics for Emerging Applications of Databases [outcome of a Dagstuhl seminar]*, pages 43–83, 2003.

[9] C. Bron and J. Kerbosch. Algorithm 457: Finding all cliques of an undirected graph. *Commun. ACM*, 16(9):575–577, Sept. 1973.

[10] R. Cavallo and M. Pittarelli. The theory of probabilistic databases. In *VLDB*, volume 87, pages 1–4, 1987.

[11] J. Chomicki, J. Marcinkowski, and S. Staworko. Cikm. CIKM '04, pages 417–426, New York, NY, USA, 2004. ACM.

[12] S. Cohen and A. Zohar. Database perspectives on blockchains. *CoRR*, abs/1803.06015, 2018.

[13] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *The VLDB Journal*, 16(4):523–544, 2007.

[14] G. Danezis, C. Fournet, M. Kohlweiss, and B. Parno. Pinocchio coin: building zerocoin from a succinct pairing-based proof system. In *Proceedings of the First ACM workshop on Language support for privacy-enhancing technologies*, pages 27–30. ACM, 2013.

[15] C. Decker and R. Wattenhofer. Bitcoin transaction malleability and mtgox. In *European Symposium on Research in Computer Security*, pages 313–326. Springer, 2014.

[16] I. Eyal, A. E. Gencer, E. G. Sirer, and R. van Renesse. Bitcoin-ng: A scalable blockchain protocol. In *USENIX*, pages 45–59, 2016.

[17] I. Eyal and E. G. Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *Financial Cryptography and Data Security - 18th International Conference, FC 2014, Christ Church, Barbados, March 3-7, 2014, Revised Selected Papers*, pages 436–454, 2014.

[18] A. Fuxman and R. J. Miller. First-order query rewriting for inconsistent databases. *J. Comput. Syst. Sci.*, 73(4):610–635, 2007.

[19] J. A. Garay, A. Kiayias, and N. Leonardos. The bitcoin backbone protocol: Analysis and applications. In *EUROCRYPT (2)*, pages 281–310, 2015.

[20] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. *IACR Cryptology ePrint Archive*, 2017:454, 2017.

[21] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68. ACM, 2017.

[22] E. Heilman, A. Kendler, A. Zohar, and S. Goldberg. Eclipse attacks on bitcoin's peer-to-peer network. In *USENIX Security Symposium*, pages 129–144, 2015.

[23] G. O. Karame, E. Androulaki, and S. Capkun. Double-spending fast payments in bitcoin. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 906–917. ACM, 2012.

[24] E. K. Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 279–296, 2016.

[25] P. G. Kolaitis, E. Pema, and W. Tan. Efficient querying of inconsistent databases with binary integer programming. *PVLDB*, 6(6):397–408, 2013.

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

[27] Y. Lewenberg, Y. Bachrach, Y. Sompolinsky, A. Zohar, and J. S. Rosenschein. Bitcoin mining pools: A cooperative game theoretic analysis. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, pages 919–927. International Foundation for Autonomous Agents and Multiagent Systems, 2015.

[28] S. Maiyya, V. Zakhary, D. Agrawal, and A. E. Abbadi. Database and distributed computing fundamentals for scalable, fault-tolerant, and consistent maintenance of blockchains. *Proc. VLDB Endow.*, 11(12):2098–2101, Aug. 2018.

[29] M. C. Marileo and L. E. Bertossi. The consistency extractor system: Answer set programs for consistent query answering in databases. *Data Knowl. Eng.*, 69(6):545–572, 2010.

[30] I. Miers, C. Garman, M. Green, and A. D. Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 397–411. IEEE, 2013.

[31] C. Mohan. Blockchains and databases. *PVLDB*, 10(12):2000–2001, 2017.

[32] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system.

[33] A. Narayanan. Blockchains: Past, present, and future. In *SIGMOD*, page 193, 2018.

[34] 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.

[35] D. Ron and A. Shamir. Quantitative analysis of the full bitcoin transaction graph. In *International Conference on Financial Cryptography and Data Security*, pages 6–24. Springer, 2013.

[36] M. Rosenfeld. Analysis of bitcoin pooled mining reward systems. *arXiv preprint arXiv:1112.4980*, 2011.

[37] A. Sapirshtein, Y. Sompolinsky, and A. Zohar. *Optimal Selfish Mining Strategies in Bitcoin*, pages 515–532. Springer Berlin Heidelberg, Berlin, Heidelberg, 2017.

[38] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 459–474. IEEE, 2014.

[39] P. Sen and A. Deshpande. Representing and querying correlated tuples in probabilistic databases. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 596–605. IEEE, 2007.

[40] Y. Sompolinsky, Y. Lewenberg, and A. Zohar. Spectre: A fast and scalable cryptocurrency protocol. *IACR Cryptology ePrint Archive*, 2016:1159, 2016.

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

[42] Y. Sompolinsky and A. Zohar. Bitcoin's security model revisited. *arXiv preprint arXiv:1605.09193*, 2016.

[43] S. Staworko, J. Chomicki, and J. Marcinkowski. Prioritized repairing and consistent query answering in relational databases. *Ann. Math. Artif. Intell.*, 64(2-3):209–246, 2012.

[44] E. Tomita, A. Tanaka, and H. Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theor. Comput. Sci.*, 363(1):28–42, Oct. 2006.

[45] A. Zohar. Bitcoin: under the hood. *Communications of the ACM*, 58(9):104–113, 2015.