The Transaction Graph for Modeling Blockchain Semantics

Christian Cachin[1], Angelo De Caro[1], Pedro Moreno-Sanchez[2]*, Björn Tackmann[1], and Marko Vukolić[1]

[1] IBM Research – Zurich, Switzerland, (cca|adc|bta|mvu)@zurich.ibm.com
[2] Purdue University, USA, pmorenos@purdue.edu

Abstract. The advent of Bitcoin paved the way for a plethora of blockchain systems supporting diverse applications beyond cryptocurrencies. Although in-depth studies of the protocols, security, and privacy of blockchains are available, there is no formal model of the transaction semantics that a blockchain is supposed to guarantee.

In this work, we fill this gap, motivated by the observation that the semantics of transactions in blockchain systems can be captured by a directed acyclic graph. Such a transaction graph, or TDAG, generally consists of the states and the transactions as transitions between the states, together with conditions for the consistency and validity of transactions. We instantiate the TDAG model for three prominent blockchain systems: Bitcoin, Ethereum, and Hyperledger Fabric. We specify the states and transactions as well as the validity conditions of the TDAG for each one. This demonstrates the applicability of the model and formalizes the transaction-level semantics that these systems aim for.

1 Introduction

The success of Bitcoin [16] has sparked the development of many other blockchains. Whereas the first blockchains after Bitcoin were so-called *alt-coins* that resembled the cryptocurrency functionality offered by Bitcoin and mostly differed in the choice of certain parameters, Ethereum [9] was the pioneer of so-called *smart contract* systems that support arbitrary (deterministic) computation on the blockchain. Platforms for running smart contracts are seen to be of wide-spread interest for replacing trusted parties, whether in public blockchains where participation is open to anyone or in private blockchains inside a consortium.

Many recent blockchain platforms run generic computations, model specific asset classes, or add cryptographic privacy guarantees; prominent systems today include Hyperledger Fabric [6], R3 Corda [13], Tendermint/Cosmos [14], and Chain Core [8].

Blockchain systems have attracted attention not only from industry but also from academia. Many works have analyzed blockchains from different perspectives, for example, focusing on the underlying consensus protocols [7, 10, 11],

* Work done at IBM Research – Zurich.

their privacy guarantees [4, 15, 18], and many more aspects. This collection is necessarily partial; excellent surveys exist in the literature [3, 5, 17, 19].

What is, surprisingly, missing to date is a formal model of the semantics of a blockchain, addressing the transaction-level consistency guarantees that it aims to achieve. These guarantees are intuitive and easy to grasp in the context of Bitcoin: given a proper modeling of the mining of new coins, the overall amount of bitcoins must remain invariant. For the newer, generic, and more complex blockchains, such as Ethereum or Hyperledger Fabric, a proper model of the guarantees they provide appear necessary. For instance, such a model should allow for reasoning whether the intuitively expected guarantees are indeed achieved. It should also capture the operation of a blockchain at an appropriate level, such that the properties of a system can be captured concisely and differences across platforms become visible. In particular, since the validation of transactions through consensus plays an important role for blockchains, the validity checks on transactions that manipulate state should be expressed in the model.

This state of affairs motivates our work. We introduce a formal model, called the *transaction graph* or *TDAG* for short, a directed acyclic graph that models the *transactions* occurring on a blockchain and how they interact through *states*. In a nutshell, a TDAG is a graph consisting of transactions that link states to each other. Each transaction may consume, observe, or produce states, and occurs only with respect to an external input that triggers the transaction. The model abstracts the transaction validation into a predicate that can be evaluated locally in the graph, in the sense that validation only considers the relevant states; this corresponds to how many blockchains work, during the process of transaction validation and consensus, which must be efficient and based on local state. The TDAG is a generic model to encode properties expected from every blockchain system, such as notions of *validity* and *consistency*, and for characterizing the *invariants* that must be enforced in a blockchain.

We instantiate the TDAG model for three different prominent blockchains: Bitcoin, Ethereum, and Hyperledger Fabric. For each system, we formally define the states and transactions of the TDAG, specify the notion of consistency, and describe the validity of transactions. This shows the broad applicability of our model, and results in an abstract description of these real-world systems.

2 Transaction graphs

This section introduces the *transaction directed acyclic graph*, abbreviated *transaction graph* or *TDAG* for representing the semantics of a blockchain. It models the *context* held by the blockchain and its evolution through *transactions* that obey *validation* rules.

We start by introducing some notation. Let $\mathcal{E} \subseteq \mathcal{X} \times \mathcal{Y}$ be a relation between sets $\mathcal{X}$ and $\mathcal{Y}$. For the predicate $(x, y) \in \mathcal{E}$, we also write $x\mathcal{E}y$. Furthermore, we denote the set $\{y : x\mathcal{E}y\}$ by $x\mathcal{E}\star$ and its size by $|x\mathcal{E}\star|$.

2.1 Definition

A *transaction graph* or TDAG is a directed acyclic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. The vertices $\mathcal{V}$ can be partitioned into *states* $\mathcal{S}$ and *witnesses* $\mathcal{W}$, that is, $\mathcal{V} = \mathcal{S} \dot\cup \mathcal{W}$. At a high level the edges $\mathcal{E}$ represent transitions between states. More precisely, an edge $e \in \mathcal{E}$ represents the relation between a state and a witness in the context of a transaction, and an edge may connect a state to a witness or vice versa. The edges can be partitioned into *consuming*, *observing*, and *producing* edges, denoted $\mathcal{E}_C$, $\mathcal{E}_O$, and $\mathcal{E}_P$, respectively, such that $\mathcal{E} = \mathcal{E}_C \dot\cup \mathcal{E}_O \dot\cup \mathcal{E}_P$. We now introduce the elements of $\mathcal{G}$ informally.

States $\bigcirc$: The first type of vertex, $s \in \mathcal{S}$, denotes an atomic *state* represented by the blockchain and is depicted by a circle $\bigcirc$. It models an individual asset, a digital coin, some coins controlled by a particular cryptographic key, a variable of a smart contract at a moment in time, and so on. The complete context of the blockchain consists of all states that exist at a particular time. A state results from a transaction on the blockchain and can transition to other states through a transaction.

There is a special *genesis state* $s_g \in \mathcal{S}$, which represents the initial state of the blockchain. There is a single genesis state by intention because the blockchain system can be initialized exactly once.

Witnesses $\square$: The second kind of vertex, $w \in \mathcal{W}$, denotes a *witness* in the context of a transaction and is depicted by a rectangle $\square$. It represents any data included in a transaction that is required for the transaction to be valid according to the validation rules of the blockchain system. Every transaction of the blockchain system contains exactly one witness.

Consuming edges $\bigcirc \longrightarrow \square$: A *consuming edge* $e \in \mathcal{E}_C$ connects a state to a witness and models that the state $\bigcirc$ is *consumed* by the transaction that involves witness $\square$, i.e., the unique transaction that corresonds to $\square$. A state can be consumed exactly once, i.e., it is not available for being consumed by another transaction once it has been consumed. Consuming a state means that the state is "updated" or "overwritten" by the transaction.

Observing edges $\bigcirc\ {-}\,{-}{\rightarrow} \square$: An *observing edge* $e \in \mathcal{E}_O$ also connects a state to a witness; it models that the state enters into the transaction represented by the witness, but that it remains available for consumption by another transaction. A state can be observed by many transactions, independently of whether it is also consumed or not. Intuitively a transaction that observes a state "reads" it.

Producing edges $\square \longrightarrow \bigcirc$: A *producing edge* $e \in \mathcal{E}_P$ connects a witness to a state, and denotes that the state is *created* or *produced* by the transaction corresponding to the witness. Every state apart from the genesis state is produced exactly once.

With these notions, a transaction represents a transition from one state, or from some set of states, in a TDAG to another set of states according to the blockchain system. The transaction is linked to a unique witness, which makes it "valid" as described later. We say that a transaction has *input states* that are consumed or observed by the transaction and *output states* that are produced by the transaction. More formally, a transaction is also a weakly connected DAG, i.e., a DAG that is connected as a graph.

Definition 1 (Transaction). *A weakly connected DAG $T = (\mathcal{V}, \mathcal{E})$ with a set of* input states $\mathcal{S}_I$*, a set of* output states $\mathcal{S}_O$*, and a witness w is called a* transaction *whenever*

- *Every input state in $\mathcal{S}_I$ is a source (has indegree zero);*
- *Every output state in $\mathcal{S}_O$ is a sink (has outdegree zero);*
- $\mathcal{V} = \mathcal{S}_I \mathbin{\dot\cup} \mathcal{S}_O \mathbin{\dot\cup} \{w\}$*;*
- *Every edge in $\mathcal{E}$ is either a consuming edge or an observing edge and links some input state $s_i \in \mathcal{S}_I$ to w, or it is a producing edge and links w to some output state $s_o \in \mathcal{S}_O$.*

As the name implies, a transaction graph consists of many transactions.

Definition 2 (TDAG). *A* transaction graph (TDAG) *is a directed unweighted graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \mathcal{S} \mathbin{\dot\cup} \mathcal{W}$ are the vertices and $\mathcal{E} = \mathcal{E}_C \mathbin{\dot\cup} \mathcal{E}_O \mathbin{\dot\cup} \mathcal{E}_P$ are the edges. The set $\mathcal{S}$ denotes the states and contains a special state s_g called* genesis*. The set $\mathcal{W}$ denotes the witnesses. Edges are partitioned into three subsets, where $\mathcal{E}_C \subseteq \mathcal{S} \times \mathcal{W}$ denotes* consuming edges*, $\mathcal{E}_O \subseteq \mathcal{S} \times \mathcal{W}$ denotes* observing edges*, and $\mathcal{E}_P \subseteq \mathcal{W} \times \mathcal{S}$ denotes the* producing edges*.*

It satisfies the following conditions:

1. *s_g does not have any producing or observing edges and it has a single consuming edge, i.e., $|\star\mathcal{E}_P s_g| = 0 \wedge |s_g\mathcal{E}_O\star| = 0 \wedge \exists! w \in \mathcal{W} : s_g\mathcal{E}_C w$.*
2. *Every state except for the genesis state has exactly one producing edge, i.e., $\forall s \in \mathcal{S} \setminus \{s_g\}\ \exists! w \in \mathcal{W} : w\mathcal{E}_P s$.*
3. *Every state except for the genesis state may have multiple successors, but at most one among them is connected with a consuming edge, i.e., $\forall s \in \mathcal{S} : |s\mathcal{E}_C\star| \leq 1$.*
4. *$\mathcal{G}$ is weakly connected.*
5. *$\mathcal{G}$ has no cycles.*

The consuming and observing edges incident to a state are also called the *outgoing edges* of that state. Similarly, the consuming and observing edges incident to a witness are called *incoming edges* of that witness. The producing edges of a witness are *outgoing edges* of the witness. There is no order among the edges incident to a vertex in a TDAG. The set of all *unconsumed* states in a TDAG are the states without an incident consuming edge.

In the context of a TDAG every witness w corresponds to a unique transaction $t(w)$. The next definition follows naturally and is easily seen to be equivalent to Definition 1.

Definition 3 (Transaction in a TDAG). *Given a TDAG $\mathcal{G} = (\mathcal{S} \mathbin{\dot\cup} \mathcal{W}, \mathcal{E})$ and a witness $w \in \mathcal{W}$, the* transaction *with witness w is the unique subgraph $t = (\mathcal{S}' \mathbin{\dot\cup} \{w\}, \mathcal{E}') \subseteq \mathcal{G}$, where*

- *$w \in \mathcal{W}$ is the* witness *of the transaction;*
- *$\mathcal{S}'$ is the set of states connected to w, i.e., $\mathcal{S}' = \{s \in \mathcal{S} : s\mathcal{E}_C w \vee s\mathcal{E}_O w \vee w\mathcal{E}_P s\}$; and*
- *$\mathcal{E}'$ are the edges with both endpoints in $\mathcal{S}' \dot{\cup} \{w\}$.*

The *input states* of $t(w)$ are the states being observed or consumed by $t(w)$, and the *output states* of $t(w)$ are the states being produced by $t(w)$. With this terminology a transaction $t \subseteq \mathcal{G}$ can have one of the following five types, which depends mostly on the number of input and output states:

INIT: A unique *initialization transaction* exists in every non-empty TDAG, consisting of a consuming edge that links the genesis state to a witness w and a set of producing edges that link w to a set of states.

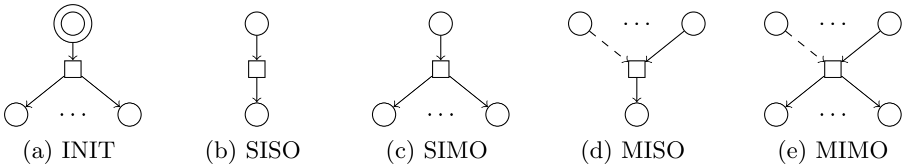
SISO: A *single-input, single-output transaction* consists of one consuming edge that links one input state to a witness w and one producing edge that links w to an output state.

SIMO: A *single-input, multi-output transaction* consists of one consuming edge that links an input state s to a witness w, and a set of producing edges that link w to a set of output states.

MISO: A *multi-input, single-output transaction* contains a set of multiple consuming and observing edges that link distinct input states to a witness w and one producing edge that links w to an output state.

MIMO: A *multi-input, multi-output transaction* contains a set of multiple consuming and observing edges that link distinct input states to a witness w, and a set of producing edges that link w to a set of output states.

Figure 1 shows the possible transaction types in a TDAG. The initialization transaction plays a special role; it represents the creation of the blockchain, which typically creates all assets represented by the states. Modeling initialization through a specific transaction is a deliberate design choice that will become clear later, in the context of transaction validation. The other types represent “ordinary” transactions that consume (and possibly observe) one or more states and produce one or more states. We note that observing edges are not consider for SISO and SIMO transactions as they have a single input state. This models



(a) INIT (b) SISO (c) SIMO (d) MISO (e) MIMO

Fig. 1: Graphical representation of transactions. States are represented by circles and witnesses are represented by boxes. Two concentric circles represent the genesis state. Observing edges are represented with a dashed arrow whereas producing edges and consuming edges are represented with solid arrows.

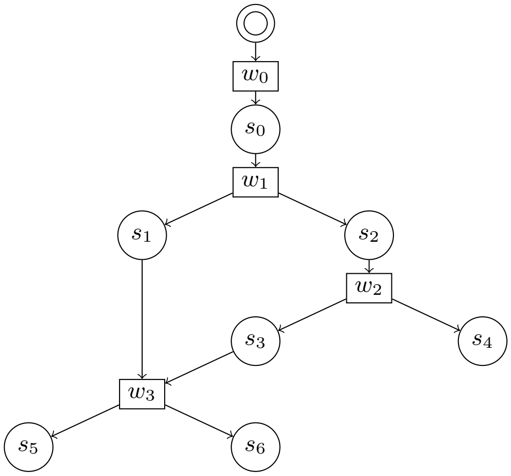


Fig. 2: Illustrative example of a TDAG. Here, we use the same notation as in Figure 1. Graphically, each transaction $t_i(w_i)$ is the subgraph where vertices are the set composed of w_i along with the set of states sharing an edge with w_i; and edges are the set of incoming edges and outgoing edges for w_i.

that a transaction must update or overwrite at least one state to make it into the blockchain as simple read queries can be handled by inspecting the blockchain.

For the moment, it suffices to say that the initialization transaction typically creates all "assets" modeled by the blockchain or the "states" that it holds, setting them to a predefined value. This allows a subsequent transaction to be linked only with the state to which it refers and that it consumes. Otherwise, all transactions that modify any state would be linked from the genesis state (with a consuming edge), contradicting that every state has at most one consuming edge. We consider this an important property of the TDAG model. A further argument for modeling only one initialization transaction goes as follows. If there were multiple INIT transactions, then it would not be easily possible to assess whether one INIT transaction is "valid" without looking also at the other ones. For instance, an INIT transaction that creates a new asset is only valid if no other INIT transaction has created the same asset beforehand.

Therefore, we purposely restrict the model so that it has a single initialization transaction for simplicity, but without loss of generality as this unique initialization transaction can create as many states as required in the blockchain.

Figure 2 shows an illustrative example of a TDAG modeling a Bitcoin execution with four transactions. First, $t_0(w_0)$ represents the creation of the Bitcoin blockchain by minting the all available bitcoins into a Bitcoin address containing unmined bitcoins (s_0). Here, w_0 represents the Bitcoin creation rules. Second, $t_1(w_1)$ represents a transaction that transfers some unmined bitcoins (s_0) to the Bitcoin address of a user u that successfully mined the first Bitcoin block (s_2); $t_1(w_1)$ saves the remaining unmined bitcoins (s_1) for subsequent block creations.

Here, w_1 represents proof-of-work in the block mined by u. Third, $t_2(w_2)$ represents a transaction where u transfers some of her bitcoins (s_2) to another Bitcoin address (s_4). The associated transaction fee is modeled as another address (s_3). Here, w_2 represents the authorization of the transaction in the form of a digital signature by u. Finally, $t_3(w_3)$ represents a transaction that rewards a user for creating a Bitcoin block containing $t_2(w_2)$. In that sense, $t_3(w_3)$ is similar to $t_1(w_1)$, with the difference that $t_3(w_3)$ also captures the fact that the user also receives the fees associated to $t_2(w_2)$.

We note that this example does not contain any observing edge. This results from the fact that read-only operations are not supported in Bitcoin.

2.2 Conflicts and validity

A central goal of blockchain systems is to prevent conflicts among transactions and to ensure validity for all transactions, as a result of a consensus process executed among the participating entities. The TDAG model permits to have a closer look at the semantics of conflicts and validity; modeling consensus is outside the scope of this work.

Intuitively, a *conflict* in a blockchain underlying a cryptocurrency such as Bitcoin occurs in an attempt to "double-spend" money. According to the example describing Bitcoin from before (and expanded in Section 3), assume that a state s in a TDAG corresponds to bitcoins held by a particular Bitcoin address. Two transactions that double-spend such bitcoins map to two transactions that both consume s. But every state in a TDAG can be consumed at most once, hence, the TDAG model already prevents this form of conflict.

In blockchains for arbitrary smart contracts, a conflict corresponds to a situation where generic validation rules for transactions are violated. Such rules may refer to coins (such as an amount of ether in Ethereum) or to other assets modeled in the blockchain. The TDAG model for these blockchains also imposes that every state can be consumed at most once.

When one considers an arbitrary set of transactions (not arising from the same transaction graph), such as transactions that have merely been proposed and are not executed on the blockchain yet, then conflicts among them could exist. This is the case in a cryptocurrency like Bitcoin when a miner searches for the next block, for example, and two transactions might be floating around in the network that both attempt to consume the same state s. Similarly, conflicting transactions exist in smart-contract platforms during the process of reaching consensus on a valid blockchain execution.

We now consider a set of transactions (in the form of a graph) and define what it means that they are free of conflicts among each other.

Definition 4 (Conflict-freedom). *Consider a DAG $\mathcal{T} = (\mathcal{S}_T \,\dot{\cup}\, \mathcal{W}_T, \mathcal{E}_T)$ with states $\mathcal{S}_T$, witnesses $\mathcal{W}_T$, producing edges $\mathcal{E}_P \subseteq \mathcal{E}_T$ and consuming edges $\mathcal{E}_C \subseteq \mathcal{E}_T$ that contains a transaction for every witness $w \in \mathcal{W}_T$. We say that $\mathcal{T}$* has no conflicts *if every state has at most one producing edge and one consuming edge, i.e., $\forall s \in \mathcal{S}_T : |\star\mathcal{E}_P s| \leq 1 \wedge |s\mathcal{E}_C\star| \leq 1$.*

A conflict-free set of transactions can be added to a TDAG. To ensure that its addition does not cause any conflicts with the TDAG only simple and local conditions have to be verified.

Definition 5 (Adding transactions to a TDAG). *Consider a TDAG* $\mathcal{G} = (\mathcal{S} \dot{\cup} \mathcal{W}, \mathcal{E})$ *and a DAG* $\mathcal{T} = (\mathcal{S}_T \dot{\cup} \mathcal{W}_T, \mathcal{E}_T)$ *containing a conflict-free set of transactions such that*

1. *No witness of* $\mathcal{T}$ *is in* $\mathcal{G}$*, i.e.,* $\mathcal{W} \cap \mathcal{W}_T = \emptyset$*;*
2. *Every input state of* $\mathcal{T}$ *is an unconsumed state of* $\mathcal{G}$*, i.e.,* $\{s \in \mathcal{S}_T : |\star\mathcal{E}_T s| = 0\} \subseteq \{s \in \mathcal{S} : |s\mathcal{E}_C \star| = 0\}$*;*
3. *The output states of* $\mathcal{T}$ *do not exist in* $\mathcal{G}$*, i.e.,* $\{s \in \mathcal{S}_T : |s\mathcal{E}_C \star| = 0\} \cap \mathcal{S} = \emptyset$*.*

Then the result of adding $\mathcal{T}$ to $\mathcal{G}$ *is the DAG* $\bar{\mathcal{G}} = (\bar{\mathcal{S}} \dot{\cup} \bar{\mathcal{W}}, \bar{\mathcal{E}})$*, with* $\bar{\mathcal{S}} = \mathcal{S} \cup \mathcal{S}_T$*,* $\bar{\mathcal{W}} = \mathcal{W} \dot{\cup} \mathcal{W}_T$*, and* $\bar{\mathcal{E}} = \mathcal{E} \dot{\cup} \mathcal{E}_T$*.*

Theorem 1. *When a conflict-free set of transactions* $\mathcal{T} = (\mathcal{S}_T \dot{\cup} \mathcal{W}_T, \mathcal{E}_T)$ *is added to a TDAG* $\mathcal{G} = (\mathcal{S} \dot{\cup} \mathcal{W}, \mathcal{E})$*, then the resulting graph* $\bar{\mathcal{G}} = (\bar{\mathcal{S}} \dot{\cup} \bar{\mathcal{W}}, \bar{\mathcal{E}})$ *is also a TDAG.*

Proof. Here we show that $\bar{\mathcal{G}}$ satisfies the conditions to be a TDAG.

1. The genesis state must not have producing or observing edges and it must have a single consuming edge. This condition is fulfilled since $\mathcal{G}$ is a TDAG and $\mathcal{T}$ does not contain the genesis state if it is already consumed in $\mathcal{G}$.
2. Every state, other than genesis, must have a single producing edge. This condition is fulfilled in $\mathcal{G}$ and in $\mathcal{T}$ by definition. Now, the addition of t to $\mathcal{G}$ does not create new edges. Therefore, this condition holds also in $\bar{\mathcal{G}}$.
3. Every state, other than the genesis, can have multiple successors, but at most one among them is connected with a consuming edge. It is easy to see that $\bar{\mathcal{G}}$ fulfills this condition following an argument similar as before.
4. The graph must be weakly connected. Note that by the definition of TDAG, each vertex $v \in \mathcal{S} \dot{\cup} \mathcal{W}$ is weakly connected to every unconsumed state in $\mathcal{G}$. Moreover, every vertex v' in $\mathcal{S}_T \dot{\cup} \mathcal{W}_T$ is weakly connected to at least one input state of $\mathcal{T}$. Now, as the set of input states in $\mathcal{T}$ is a subset of the unconsumed states in $\mathcal{G}$, it follows that $\bar{\mathcal{G}}$ is weakly connected.
5. The graph must not have cycles. According to the assumptions on $\mathcal{T}$ and because $\mathcal{G}$ is a DAG, and through the way in which $\bar{\mathcal{G}}$ is constructed, it is easy to see that $\bar{\mathcal{G}}$ has no cycles.

□

We now introduce the notion of *validity* for transactions in a TDAG, which models the fact that on a blockchain only "valid" transactions are executed. As an important design choice of the model, the validity of a transaction in a TDAG must be decidable locally, that is, from the transaction alone, considering only its input states, the witness, and the output states. To capture this, we assume that the blockchain context defines a boolean *validation predicate* $\mathbb{P}(\cdot)$ on the space of all transactions.

Definition 6 (Validity). *Let t be a transaction in a TDAG $\mathcal{G}$. Then t is* valid *whenever $\mathbb{P}(t) = \textsc{True}$. Furthermore, $\mathcal{G}$ is a* valid transaction graph *if all transactions in $\mathcal{G}$ are valid.*

Combined with the locally checkable conditions for adding transactions to a TDAG, the fact that the validity of a transaction is locally decidable defines, in an influential way, how many blockchain systems work during consensus, validation, and execution of new transactions. The only steps needed for validation are to ensure the validity predicate of a candidate transaction plus the checks according to Definition 5 involving the states to which the transaction refers.

Transaction validation also relies on the property that all states in the TDAG are distinct. In a typical blockchain, the validation function relies on a cryptographic hash of the states to which it refers; this directly ensures uniqueness. For example, consider an execution of a smart contract that holds state on the blockchain in the form of a local variable *var*. The contract may update *var* multiple times, and it may write the same value to *var* more than once. To make the resulting states in the TDAG different, the model will usually include a version number in the state that makes each assignment unique.

At this point, let us review our design choice of a single INIT transaction. Using a single transaction to create all assets represented by the states enables to locally check the validity of the initialization of the blockchain as well as preserve the locally checkable conditions for further transactions consuming those states.

2.3 Composition of transaction graphs

In the following, we describe the composition of transaction graphs, which states the conditions under which two TDAGs can be merged into a single one. One may then reason about their consistency and validity in a unified manner.

Definition 7 (TDAG composition). *Consider two TDAGs $\mathcal{G} := (\mathcal{S} \dot\cup \mathcal{W}, \mathcal{E})$ and $\mathcal{G}' := (\mathcal{S}' \dot\cup \mathcal{W}', \mathcal{E}')$. Assume that $t(w)$ denotes the INIT transaction in $\mathcal{G}$ and $t'(w')$ denotes the INIT transaction in $\mathcal{G}'$. Further assume that $\widehat{t}(\widehat{w})$ denotes a INIT transaction where $\widehat{w} = (w, w')$ and the output states are the union of output states from $t(w)$ and $t'(w')$. Then, the* composition of $\mathcal{G}$ *and* $\mathcal{G}'$ *is the TDAG $\widehat{\mathcal{G}} = \mathcal{T}_{\mathcal{G}} \setminus \{t(w)\} \cup \mathcal{T}_{\mathcal{G}'} \setminus \{t'(w')\} \cup \widehat{t}(\widehat{w})$.*

Theorem 2 (Composition of two TDAGs into one TDAG). *The composition of two TDAGs $\mathcal{G}$ and $\mathcal{G}'$ results in a graph $\widehat{\mathcal{G}}$, which is also a TDAG.*

Proof. Here we show that $\widehat{\mathcal{G}}$ satisfies the conditions to be a TDAG.

1. The genesis state must not have producing or observing edges and it must have a single consuming edge. This condition is fulfilled by our definition of the INIT transaction $\widehat{t}(\widehat{w})$.
2. Every state, other than genesis, must have a single producing edge. As $\mathcal{G}$ and $\mathcal{G}'$ are two TDAGs, it is easy to see that each state in $\mathcal{T}_{\mathcal{G}} \setminus \{t(w)\}$ and $\mathcal{T}_{\mathcal{G}'} \setminus \{t'(w')\}$ has a single producing edge. Moreover, by definition of INIT transaction, each output state in $\widehat{t}(\widehat{w})$ has a single producing edge.

3. Every state, other than the genesis, can have multiple successors, but at most one among them is connected with a consuming edge. It is easy to see that $\widehat{\mathcal{G}}$ fulfills this condition along the lines of previous argument.
4. The graph must be weakly connected. $\mathcal{T}_{\mathcal{G}} \setminus \{t(w)\}$ and $\mathcal{T}_{\mathcal{G}'} \setminus \{t'(w')\}$ are connected by definition, as $\mathcal{G}$ and $\mathcal{G}'$ are two TDAGs. Moreover, the definition of the INIT transaction $\widehat{t}(\widehat{w})$ ensures that any vertex in $\mathcal{T}_{\mathcal{G}} \setminus \{t(w)\}$ is connected to any vertex in $\mathcal{T}_{\mathcal{G}'} \setminus \{t'(w')\}$ through $\widehat{w}$.
5. The graph must not have cycles. $\mathcal{T}_{\mathcal{G}} \setminus \{t(w)\}$ and $\mathcal{T}_{\mathcal{G}'} \setminus \{t'(w')\}$ are acyclic by definition, as $\mathcal{G}$ and $\mathcal{G}'$ are two TDAGs. Moreover, the addition of $\widehat{t}(\widehat{w})$ clearly does not introduce any cycle.

□

3 Applications

In this section, we describe how executions of different blockchain systems are modeled by transaction graphs. We cover three prominent blockchains: Bitcoin, Ethereum, and Hyperledger Fabric (HLF). They differ in how they store assets in their state. Bitcoin, for example, does not have state "variables" but maintains an asset only in the context of the transaction that created it. Ethereum, on the other hand, uses variables and accounts for its state. The data model in HLF is a key-value store (KVS), which can be mapped to local database on each node. Due to lack of space, this section only gives a short overview and more details appear in the appendix. In particular, we describe the transaction graph instance for Bitcoin in Appendix A and the transaction graph instance for HLF in Appendix B.

Throughout this section, we denote by $y \leftarrow \mathsf{H}(x)$ a cryptographic, collision-free hash function that takes as input a bit-string $x \in \{0,1\}^*$ of arbitrary length and returns a fixed-length string $y \in \{0,1\}^l$.

3.1 Bitcoin

Since Bitcoin (`bitcoin.org`) is the prototype of all blockchain systems, there are many publicly available descriptions [2, 16] and we keep the background short. Likewise, the discussion here applies to all *altcoins* patterned after Bitcoin.

Bitcoin combines transaction validation, coin mining, and agreement on the ledger with the "Nakamoto protocol" that uses proof-of-work and ensures consensus. A *block* in Bitcoin can hold two types of *transactions*:

- A *coinbase transaction* that transfers yet unmined bitcoins to a Bitcoin address as chosen by the miner of the corresponding block, as a reward for creating the block. This transaction is valid if (i) it transfers a number of bitcoins according to the height of the block to a Bitcoin address, and (ii) is accompanied by the solution to the proof-of-work puzzle for successful mining of the block.

- A *regular transaction* transfers bitcoins from a set of Bitcoin (input) addresses to another set of Bitcoin (output) addresses. It also incurs a fee, defined as the difference between the bitcoin amounts in the input and output, which is assigned to the miner of the block in which the transaction appears. A regular transaction is valid if it includes a confirmation for each input for the amount and output and if it does not create new bitcoins.

Bitcoin value exists in the blockchain in the form of *unspent transaction output*, often abbreviated *UTXO*, which has been assigned to an address, representing a digital-signature public key. This value is controlled by the holder of the corresponding private key. It can be spent and transferred to another address by signing a transaction with the private key.

In the TDAG modeling Bitcoin, we let every state be a tuple of the form

$$(\mathit{addr}, \mathit{val}, \mathit{hash}, \mathit{height}) \ ,$$

where *addr* denotes an address, *val* denotes the amount of bitcoins held in this state, *hash* is the cryptographic hash of other states (whose UTXO is transferred by the transaction), and *height* denotes the index of the block in which the state was produced.

In contrast to the Bitcoin code, we model transaction fees and unmined bitcoins as held by or associated to an (imaginary) address. This allows a coherent model for the TDAG. Thus, the state resulting from the special INIT transaction contains $(\textsc{addr}_0, 21\mathrm{M}, \dots)$, representing all 21M bitcoins that ever exist.

The form of a witness depends on the transaction type: The witness for a coinbase transaction is the solution for the proof-of-work to assign the bitcoins to the address designated by the miner. For a regular transaction, the witness consists of a set of confirmations for the transfer of bitcoin, in the form of a digital signature for each UTXO, over the input and output addresses of the transfer. Finally, the INIT transaction does not require any witness.

The TDAG for Bitcoin contains producing and consuming edges but no observing edges. For a coinbase transaction, the input states are the unconsumed state of unmined bitcoins and the fee states for the transactions included in the mined block. One producing edge leads to a state for collecting the fees and the mining reward, another one to a state containing the remaining unmined bitcoins. Its witness is the mining proof. For a regular transaction, the input states are the unconsumed states representing the transaction inputs and the produced output states correspond to the transactions output addresses. The witness holds a set of confirmations (digital signatures), confirming for each input state the transfer of some bitcoins to the corresponding output addresses.

The transaction predicate incorporates the validation rules of Bitcoin, as expressed in the states, witnesses, and transactions of the TDAG.

With these definitions, one can then show the intuitive result that except with negligible probability, every (legal) execution of Bitcoin, considering only bitcoin transactions that are "deep enough" in the blockchain (e.g., six blocks deep) [11] gives rise to a TDAG constructed like this. The formal analysis of this result exploits that the DAG formed by the hash-function applications among states has no cycles, and therefore satisfies the properties of a TDAG.

3.2 Ethereum

Ethereum [9] is the most prominent public blockchain and cryptocurrency supporting generic smart contracts today (`ethereum.org`). In Ethereum there exist two types of accounts, called *externally owned accounts* and *contract accounts*. Externally owned accounts largely resemble the accounts of other cryptocurrencies such as Bitcoin, in which users maintain their currency balance in Ether, owned by them. But the main innovation of Ethereum lies in contract accounts, which represent a smart contract (an arbitrary piece of code in the platform-specific language) and that executes a set of instructions upon receiving suitable input. A contract account also holds and controls its own Ether balance and specifies a *gas price*, which determines the cost of executing its code for anyone that invokes the contract.

Ethereum supports several types of transactions. First, a transaction in Ethereum can be used to transfer Ether between two externally owned accounts. This type of transaction is like the exchange of coins in other cryptocurrencies. Second, a transaction can be used to create a contract with the code of the contract and an externally owned account as inputs. It outputs a contract account with the information required to initialize the implemented code (e.g., the inputs for the init function). Finally, a transaction can be used to invoke an existing contract on the blockchain.

An Ethereum transaction includes as input the sender's address (an externally owned account), a recipient address (another account), a transaction value to be transferred from the sender's address to the recipient, some arguments with parameters for the contract, and a *gas limit*, specifying a maximum price for the execution. A contract may also call functions of other contracts; however, this will not give rise to new transactions, as these calls take place in the context of the original transaction.

To model an Ethereum execution as a TDAG, we let each state consist of a tuple

$$(\mathit{addr}, \mathit{account\text{-}type}, \mathit{code}, \mathit{local\text{-}state}, \mathit{gas\text{-}price}, \mathit{val})\ .$$

Here, *addr* denotes the account address that produced the state, *account-type* determines whether this is a state of an external account or a contract account, *code* is a hash of the smart contract's code, *local-state* denotes collectively all variables held by the contract, *gas-price* is the price for executing transactions with this contract, and *val* is the Ether balance held by the account after the execution that produced the state. If *account-type* specifies an externally owned account, then the smart contract is the fixed logic to validate payments from such accounts.

There is also a genesis state that models the creation of an Ethereum blockchain. In contrast to Bitcoin, there is currently no bound on the amount of Ether that will exist in the public Ethereum blockchain; the creation of new Ether is therefore subsumed into the mining operation and its validation.

A transaction in the TDAG is determined by the witness. It corresponds to an invocation of a smart contract and contains a gas limit and regular input arguments that validate the transaction. For instance, these arguments must contain a digital signature valid under the public key associated to the invoking external account that runs the transaction.

The transaction contains the state of the invoking account and the state of the contract as input states, with consuming edges to the witness. It also produces two states, an updated state of the invoking account and an updated state of the contract, as resulting from running the contract with the given gas limit and input arguments. If the contract calls functions of other contracts and they modify their state, then the states representing these contracts are also part of the transaction in the TDAG (as input states and output states). The validation predicate simply executes the code.

For mining new Ether, running transactions, and collecting the corresponding fees, similar states and validation logic as in the TDAG model of Bitcoin are added. Given these notions one can show that every (legal) execution of Ethereum, considering as in Bitcoin only those transactions that are deep enough in the blockchain, produces a valid TDAG.

3.3 Hyperledger Fabric

Hyperledger Fabric (`www.hyperledger.org/projects/fabric`), or *HLF* for short, is a permissioned blockchain framework, designed to support modular implementations of different components, including its consensus protocol, membership provider, and cryptography library [6]. The nodes executing the HLF blockchain are called *peers*.

An instance of HLF may contain multiple channels that may run on different sets of peers, where each channel operates like a blockchain system independent of the others, apart from using some of the same code infrastructure, ordering protocol, and other components. We therefore consider only one channel here, modeling one blockchain.

On a channel, a *configuration* transaction (*configtx*) sets the initial values used for transaction processing, such as the credentials of the peers or organizations controlling the channel, the implementation of its ordering service, and so on. Once a channel has been prepared like this, it is ready to execute operations on its peers. Transactions in HLF are executed by smart contracts called *chaincode*. We remark that several chaincodes might coexist in a channel and their transactions are differentiated by the corresponding chaincode's identifier. Nevertheless, for ease of exposition we assume in the following that there exists a single chaincode per channel.

Chaincode is first *installed* on the peer and may later be *upgraded*; it must be *instantiated* for a specific channel before it can process transactions. Once instantiated on the channel, a chaincode supports two types of transactions: *init* and *invoke*. An *init* transaction is executed once after the chaincode has been installed or upgraded; it specifies an *endorsement policy* that determines how any subsequent transaction of this chaincode should be authorized. A chaincode determines through the endorsement policy on which peers it executes: whether all peers in the channel execute it, or only some, and which peers or which set of peers are sufficient to authorize the execution of the transaction.

An *invoke* transaction is used to execute a computation that may read and modify the state of the chaincode, which is a set of key-value pairs. The operations to access the state are $\textsc{GetState}(k) \rightarrow v$ (given a key k, return the last value v written to it) and $\textsc{PutState}(k, v)$ (write the value v to storage under the key k).

The processing of a transaction on HLF proceeds like this [1]:

1. A client creates and signs a transaction proposal for a particular chaincode and sends it to the respective endorsing peers.
2. The endorsing peers simulate the transaction on their current current copy of the key-value store (KVS), verifying that the client is authorized to execute it. If successful, each endorsing peer returns the result of the executed simulation to the client. This is also called an *endorsement.* It comes in the form of a proposal response containing a signed readset and writeset (with the key-value pairs accessed during simulation, including a *version* for every value in the readset, determined by the logical time when this value was written). The endorsement serves as a static representation of the chaincode execution simulation.
3. When the client has assembled enough endorsements that produce the same KVS changes and that satisfy the endorsement policy, it combines them to a transaction. Then the client broadcasts this transaction proposal to the ordering service, which simply orders transactions without considering their semantics. Currently an ordering service based on Apache Kafka (`kafka.apache.org`) running in a cluster is supported and an ordering service using BFT consensus is under development [7, 20].
4. The ordering service disseminates an ordered stream of transactions (grouped into blocks) to the peers on the channel. Each peer on its own then validates each transaction, by verifying that the endorsement policy is satisfied and that there were no changes to the key-value pairs contained in the readset (since transaction simulation).
5. If successful, the peer appends the block to the blockchain (of the channel) and performs the updates from the writeset to its local copy of the KVS. This assigns a version to the modified key-value pairs. Since the validation is deterministic, the states and versions are the same for all correct peers.

In the TDAG for HLF, the states correspond to the entries in the KVS. Every state is a tuple containing at least

$$(\mathit{key}, \mathit{version}) \; .$$

It is assumed that an *init* transaction implicitly initializes every *key* used by the chaincode later with a default value $(-)$. The *init* transaction is always valid.

Furthermore, every *invoke* transaction that reads or writes a set of keys $\mathcal{K}$, contains an observing edge for every $k \in \mathcal{K}$ accessed by an operation $\textsc{GetState}(k)$ but not by an operation $\textsc{PutState}(k, \star)$, and a consuming edge for every k that is written using an operation $\textsc{PutState}(k, \star)$. In other words, every key is implicitly read before it is written and, thus, a transaction in the TDAG modeling an HLF execution has the same number of consuming edges as the number of producing edges.

A witness in the TDAG corresponds to a valid endorsement, in the form of signatures from the endorsers issued on the same readset/writeset pair from the transaction proposal. The validation predicate $\mathbb{P}(\cdot)$ contains the steps that each peer takes to validate a transaction coming from the ordering service, with respect to its local KVS. Notice that this validation only accesses the versions in the readset, but no other state entry in the KVS. Since these states are also contained in the transaction in the TDAG, the evaluation of $\mathbb{P}(\cdot)$ in the graph is local.

Given that the ordering service of HLF outputs the same stream of blocks with transactions to every connected peer, it is easy to verify that the graph resulting from any execution of HLF is a TDAG.

4 Conclusion

Blockchains and distributed ledger platforms are of great interest for the financial industry today, due to their role as trustless intermediaries gained from their resilience to attacks and subversion. For gaining confidence in a new technology, it is paramount to study its security with formal models.

This work has proposed transaction graphs or TDAGs as a discrete model for the semantics of the interactions in a blockchain system. In contrast to existing event-based models for generic distributed and concurrent systems, it explicitly takes into account the validation of transactions, which is an important aspect of blockchains. For instance, the TDAG model allows to model assets and their transfer among different entities. It also facilitates comparisons among different technologies available today.

We envision that richer semantics can be expressed by refining the TDAG model. For instance, one may argue about further invariants of the blockchain system as properties of the TDAG, similar to modeling Bitcoin's fixed coin supply. One might also use a TDAG to formally model the provenance for generic assets that are handled by smart contracts, building on the paths through which the asset was transferred in the TDAG.

Acknowledgments

We thank Elli Androulaki and Alessandro Sorniotti for collaboration and discussions on this topic.

This work has been supported in part by the European Commission through the Horizon 2020 Framework Programme (H2020-ICT-2014-1) under grant agreement number 644579 ESCUDO-CLOUD and in part by the Swiss State Secretariat for Education, Research and Innovation (SERI) under contracts number 15.0087.

References

1. Androulaki, E., Cachin, C., Christidis, K., Murthy, C., Nguyen, B., Vukolić, M.: Next consensus architecture proposal. Hyperledger Wiki, Fabric Design Documents, available at https://github.com/hyperledger/fabric/blob/master/proposals/r1/Next-Consensus-Architecture-Proposal.md (2016)
2. Antonopoulos, A.: Mastering Bitcoin. O'Reilly (2014)
3. Armknecht, F., Karame, G.O., Mandal, A., Youssef, F., Zenner, E.: Ripple: Overview and outlook. In: Proc. Trust and Trustworthy Computing (TRUST). pp. 163–180 (2015)
4. Ben-Sasson, E., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., Virza, M.: Zerocash: Decentralized anonymous payments from bitcoin. In: IEEE Symposium on Security and Privacy. pp. 459–474 (2014)
5. Bonneau, J., Miller, A., Clark, J., Narayanan, A., Kroll, J.A., Felten, E.W.: Sok: Research perspectives and challenges for bitcoin and cryptocurrencies. In: IEEE Symposium on Security and Privacy. pp. 104–121 (May 2015)
6. Cachin, C.: Architecture of the hyperledger blockchain fabric, https://www.zurich.ibm.com/dccl/papers/cachin_dccl.pdf. Accessed August 2017
7. Cachin, C., Vukolić, M.: Blockchain consensus protocols in the wild. arXiv:1707.01873v2, https://arxiv.org/abs/1707.01873v2
8. Chain protocol whitepaper. White Paper (2017), https://chain.com/docs/1.2/protocol/papers/whitepaper
9. Ethereum, a next-generation smart contract and decentralized application platform. White Paper (2017), https://github.com/ethereum/wiki/wiki/White-Paper
10. Eyal, I., Gencer, A.E., Sirer, E.G., Renesse, R.V.: Bitcoin-ng: A scalable blockchain protocol. In: USENIX Symposium on Networked Systems Design and Implementation (NSDI). pp. 45–59 (2016)
11. Garay, J.A., Kiayias, A., Leonardos, N.: The bitcoin backbone protocol: Analysis and applications. In: Advances in Cryptology: Eurocrypt 2015. Lecture Notes in Computer Science, vol. 9057, pp. 281–310. Springer (2015)
12. Goldwasser, S., Bellare, M.: Lecture notes on cryptography, http://cseweb.ucsd.edu/~mihir/papers/gb.pdf. Accessed August 2017
13. Hearn, M.: Corda: A distributed ledger. Technical White Paper, https://docs.corda.net/_static/corda-technical-whitepaper.pdf
14. Kwon, J., Buchman, E.: Cosmos: A network of distributed ledgers. White Paper (2017), https://cosmos.network/whitepaper
15. Meiklejohn, S., Pomarole, M., Jordan, G., Levchenko, K., McCoy, D., Voelker, G.M., Savage, S.: A fistful of bitcoins: Characterizing payments among men with no names. In: ACM Internet Measurement Conference (IMC). pp. 127–140 (2013)
16. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. http://www.bitcoin.org/bitcoin.pdf (2008)

17. Narayanan, A., Bonneau, J., Felten, E.W., Miller, A., Goldfeder, S.: Bitcoin and Cryptocurrency Technologies – A Comprehensive Introduction. Princeton University Press (2016)
18. Ruffing, T., Moreno-Sanchez, P.: Mixing confidential transactions: Comprehensive transaction privacy for bitcoin. In: Bitcoin and Blockchain Research (BITCOIN) (2017)
19. Tschorsch, F., Scheuermann, B.: Bitcoin and beyond: A technical survey on decentralized digital currencies. IEEE Communications Surveys & Tutorials 18(3), 2084–2123 (2016)
20. Vukolić, M.: Rethinking permissioned blockchains. In: Proc. ACM Workshop on Blockchain, Cryptocurrencies and Contracts (BCC '17) (2017)

Appendix

A Transaction graph for Bitcoin

A.1 Detailed description

We start with the description of an execution of the Bitcoin system as represented by the corresponding blockchain. A Bitcoin blockchain is composed of *blocks*, where each block is created as a result of successfully executing the Bitcoin mining process [16]. The miner of such block (i.e., user showing a valid proof of successful mining) chooses a set of *regular* transactions to be added in the block along with a single *coinbase* transaction. There exists a special block, denoted as *genesis* block, that represents the initialization of the blockchain.

A coinbase transaction transfers unmined bitcoins to a (set of) Bitcoin address, chosen by the corresponding miner, as a *reward* for creating the block. A coinbase transaction is valid if it transfers only the number of bitcoins set as reward according to the height of the mined blocked. A regular transaction transfers bitcoins from a set of Bitcoin *addresses* (i.e., input addresses) to another set of Bitcoin addresses (i.e., output addresses). A regular transaction is valid if: (i) it includes a *confirmation* for each input address; (ii) it does not create new bitcoins. Finally, a regular transaction has an associated fee (i.e., between the bitcoins held at input and output addresses).

Definition 8 (Bitcoin execution). *A Bitcoin execution $\mathcal{L}_{\mathsf{BTC}}$ is a set of blocks $\mathcal{B} := \{\textsc{b}_g, \textsc{b}_1, \ldots, \textsc{b}_n\}$, where $\textsc{b}_g$ denotes the genesis block and contains a single initialization transaction. Each other block $\textsc{b}_i := (\textsc{mp}, \{\textsc{cbtx}, \textsc{rtx}_1, \ldots, \textsc{rtx}_n\})$ is a tuple composed of a proof of successful mining* $\textsc{mp}$*, and a set of transactions containing a coinbase transaction* $\textsc{cbtx}$ *and regular transactions* $\textsc{rtx}_i$*. A* $\textsc{cbtx}$ *contains a Bitcoin address* $\textsc{addr}$*. A* $\textsc{rtx}$ *is a tuple* ($\textsc{addr}_{in}$, $\mathcal{F}$, $\textsc{addr}_{out}$)*, where* $\textsc{addr}_{in}$ *and* $\textsc{addr}_{out}$ *are two sets of Bitcoin addresses and* $\mathcal{F}$ *is a set of confirmations* $\textsc{cf}$*.*

We now describe our modeling of a given execution of Bitcoin as a TDAG. A state represents a Bitcoin address that holds a group of bitcoins, a transaction fee or the yet unmined bitcoins. We note that fees and unmined Bitcoins are not associated to an address in the real Bitcoin, but we model them as held by an address to have a coherent transaction graph model. The genesis state represents a Bitcoin address holding the 21M bitcoins ever existing in the Bitcoin system. Each witness represents either a proof of successful mining for a block or the (set of) confirmations required in a regular transaction. Finally, we consider two types of edges: producing and consuming edges. A producing edge links unconsumed addresses for unmined bitcoins and transaction fees to the mining proof for the corresponding coinbase transaction; or an input address to the corresponding confirmation in a regular transaction. A consuming edge links a mining proof to the Bitcoin addresses getting the reward, or a set of confirmations to the corresponding output addresses receiving (part of) the transferred bitcoins.

Definition 9 (Transaction graph). *We model an execution of Bitcoin system as a graph* $\mathcal{G}_{\mathsf{BTC}} := (\mathcal{S}_{\mathsf{BTC}} \dot\cup \mathcal{W}_{\mathsf{BTC}}, \mathcal{E}_{\mathsf{BTC}})$ *defined as follows:*

State: *Each state* $s \in \mathcal{S}_{\mathsf{BTC}}$ *is defined as a tuple (addr, val, hash, height), where addr denotes a Bitcoin address, val denotes the amount of Bitcoins held at addr, hash denotes the result of applying* H *to a set of vertices* $\mathcal{S}'_{\mathsf{BTC}} \dot\cup \{w\}$ *with* $\mathcal{S}'_{\mathsf{BTC}} \subset \mathcal{S}_{\mathsf{BTC}}$*, and height denotes a block index. The genesis state* s_g *is defined as the fixed tuple* $(\text{ADDR}_0, 21\text{M}, \mathsf{H}(\emptyset), 0)$.

Witness: *Each witness* $w \in \mathcal{W}_{\mathsf{BTC}}$ *is defined by a tuple (txtype,* $\mathcal{F}$*), where txtype denotes the type of the transaction and determines the content of* $\mathcal{F}$*. In particular, (* TINITX*,* $\emptyset$*) is the witness for the initialization transaction; (* TCBTX*,* MP*) denotes a witness for a coinbase transaction and (* TRTX*,* $\{\text{CF}_i\}$*) denotes the witness for a regular transaction.*

Edge: *Each edge* $e \in \mathcal{E}$ *is defined either as consuming edge or producing edge.*

The transaction graph presented here determines the modeling of the possible transactions in a Bitcoin execution. The next definition maps transaction in a Bitcoin execution to transaction types supported in a TDAG. For simplicity and consistency the state resulting from the INIT transaction again contains all 21M unmined bitcoins.

Definition 10 (Transaction types). *A coinbase transaction is modeled as a SIMO transaction. A regular transaction is modeled as a SISO, SIMO, MISO or MIMO transaction depending on* $|\text{ADDR}_{in}|$ *and* $|\text{ADDR}_{out}|$*. For instance, SISO models a regular transaction where* $|\text{ADDR}_{in}| = 1 \wedge |\text{ADDR}_{out}| = 1$*. The rest are derived accordingly. Finally, we define the initialization transaction included in the genesis block as an INIT transaction of the form* $t := (\{s_g, s, w\}, \{(s_g, w), (w, s)\})$*, where* $(s_g, w) \in \mathcal{E}_C$*,* $(w, s) \in \mathcal{E}_P$ *and* $s := (\text{ADDR}_m, 21\text{M}, \mathsf{H}(\{s_g, w\}), 0)$*, where* ADDR_m *denotes a Bitcoin address that contains unmined bitcoins.* s_g *and* w *are as defined in Definition 9.*

Finally, we complete our description of the Bitcoin context with the corresponding transaction predicate $\mathbb{P}$. For that, we use $\mathsf{VerifyContract}$ (ADDR, CF) as a function that on input a Bitcoin address ADDR and a confirmation CF, returns TRUE if CF encodes a valid confirmation to spend the bitcoins held at ADDR. Otherwise, it returns FALSE. Additionally, we use $\mathsf{VerifyWork}$ (MP) as a function that on input a mining proof MP, returns TRUE if MP is a valid proof-of-work for the corresponding block, or FALSE otherwise. We thereby abstract away the implementation details for validation of Bitcoin scripts and mining proofs.

Definition 11 (Transaction predicate). *Consider a transaction* $t := (\mathcal{S} \dot\cup \{w\}, \mathcal{E})$*. Then,* $\mathbb{P}(t)$ *returns* TRUE *if the following conditions hold and* FALSE *otherwise.*

1. *If t is a regular transaction (w.txtype =* TRTX*), the witness holds a valid confirmation for each input state i.e.,* $\forall s \in \star\mathcal{E}w, \exists \text{CF} \in w.\mathcal{F} : \mathsf{VerifyContract}(s.addr, \text{CF})$.

2. *If t is a coinbase transaction ($w.txtype =$ TCBTX), the witness contains a valid mining proof, i.e., $w.\mathcal{F} := \{\textsc{mp}\} \wedge \mathsf{VerifyWork}(\textsc{mp})$.*
3. *Each output state represents a positive number of bitcoins, i.e., $\forall s \in w\mathcal{E}\star : s.val > 0$.*
4. *The sum of bitcoins held at the input states must be equal to the sum of bitcoins held at the output states, i.e., $\sum_{s \in \star\mathcal{E}w} s.val = \sum_{s' \in w\mathcal{E}\star} s'.val$*
5. *Each output state contains the evaluation of the hash function over input states and the witness, i.e., $\forall s \in w\mathcal{E}\star : s.hash = \mathsf{H}(\star\mathcal{E}w \,\dot\cup\, \{w\})$.*

A.2 Model analysis

We star this section by analyzing the definition of transaction graph presented in the previous section. We start by showing that it is a TDAG. Here, we consider *legal*, a Bitcoin execution that contains only transactions that are "deep enough" in the blockchain (e.g., six blocks deep). We thereby enable the study of any Bitcoin execution in terms of the properties of a TDAG such as conflict-freedom or validity.

Theorem 3. *Assume H is a collision-resistant hash function [12] and assume that $\mathcal{L}_{\mathsf{BTC}}$ is a legal Bitcoin execution. Then, the graph $\mathcal{G}_{\mathsf{BTC}}$ resulting from modeling $\mathcal{L}_{\mathsf{BTC}}$ is a TDAG.*

Proof. Here, we show that $\mathcal{G}_{\mathsf{BTC}} = (\mathcal{S}_{\mathsf{BTC}} \,\dot\cup\, \mathcal{W}_{\mathsf{BTC}}, \mathcal{E}_{\mathsf{BTC}})$ fulfills the conditions to be a TDAG.

1. The genesis state must not have producing or observing edges and it must have a single producing edge. Our designed INIT transaction ensures this.
2. Every state, other than the genesis, must have a single producing edge. Assume by contradiction that it is not fulfilled. Then, there is a state $s \in \mathcal{S}_{\mathsf{BTC}}$ with at least two producing edges and that implies that there exists two different sets $\mathcal{V} := \mathcal{S} \,\dot\cup\, \{w\}$ and $\mathcal{V}' := \mathcal{S}' \,\dot\cup\, \{w'\}$ such that $\mathsf{H}(\mathcal{V}) = \mathsf{H}(\mathcal{V}')$. However, $\mathcal{V}$ and $\mathcal{V}'$ contradict the assumption that H is collision resistant.
3. Every state other than the genesis can have multiple successors, but at most one among them is connected with a consuming edge. Each Bitcoin address is consumed only once in a legal Bitcoin execution. Therefore, this condition is fulfilled.
4. The graph must be weakly connected. Each new transaction consumes a previously unconsumed state in the graph , i.e., either a unspent Bitcoin address or mines yet unmined bitcoins and consumes unclaimed fees. Therefore, the overall graph is weakly connected.
5. The graph must not have cycles. Assume by contradiction that there is a cycle in $\mathcal{G}_{\mathsf{BTC}}$. This, however, implies that there are two different transactions t and t' that produce the same state. However, as we have seen before, this contradicts the fact that H is collision resistant.

Remember from Definition 6 that a TDAG is *valid* if each transaction individually is valid according to a transaction predicate $\mathbb{P}$. Next, we show that validating Bitcoin transactions individually in our model, suffices to safely consider that unconsumed states represent all bitcoins in the system.

Definition 12 (Unspent bitcoins). *Consider $\mathcal{G}_{\mathsf{BTC}}$ a TDAG modeling a Bitcoin execution. Then, the* unspent bitcoins *in $\mathcal{G}_{\mathsf{BTC}}$ are the sum of bitcoins held at unconsumed states of $\mathcal{G}_{\mathsf{BTC}}$.*

Theorem 4 (Unspent bitcoins are all bitcoins in the system). *Consider $\mathcal{G}_{\mathsf{BTC}}$ a valid TDAG that models a Bitcoin execution. Then, the amount of unspent bitcoins in $\mathcal{G}_{\mathsf{BTC}}$ is equal to all bitcoins ever existing in the system. More formally, let $\mathcal{S}'$ be the set of unconsumed states in $\mathcal{G}_{\mathsf{BTC}}$, then $\sum_{s \in \mathcal{S}} s.val = s_g.val$.*

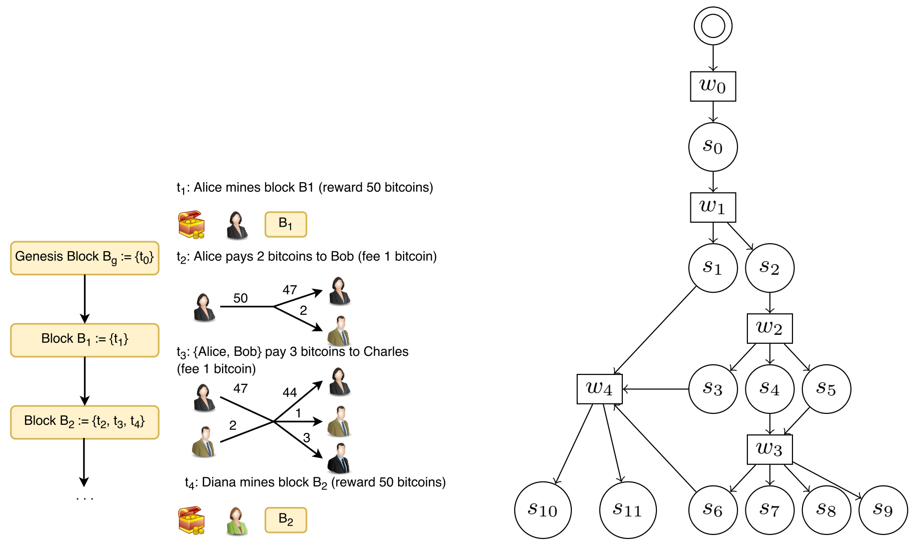
Proof. Assume by contradiction that Theorem 4 does not hold. Then, there must exist a transaction $t := (\mathcal{S} \dot\cup \{w\}, \mathcal{E})$ in $\mathcal{T}_{\mathcal{G}_{\mathsf{BTC}}}$ such that $\sum_{s \in \star\mathcal{E}w} s.val \neq \sum_{s' \in w\mathcal{E}\star} s'.val$. This, however, clearly implies that $\mathbb{P}(t)$ returns FALSE, which contradicts the assumption that $\mathcal{G}_{\mathsf{BTC}}$ is a valid TDAG.

A.3 Modeling an example of bitcoin execution

Here, we describe our modeling for an illustrative example of Bitcoin execution. We assume for simplicity that the block reward is fixed to a value of 50 bitcoins as it was the first reward set in the Bitcoin system. Additionally, we assume that the transaction fee is fixed to 1 bitcoin. We stress, however, that the TDAG model is expressive enough to relax these assumptions.

We focus in the illustrative example depicted in Figure 3. In particular, Figure 3a shows a possible Bitcoin execution $\mathcal{L}_{\mathsf{BTC}} := \{\textsc{b}_g, \textsc{b}_1, \textsc{b}_2\}$, where $\textsc{b}_g := (\emptyset, \{t_0\}), \textsc{b}_1 := (\textsc{mp}, \{t_1\})$ and $\textsc{b}_2 := (\textsc{mp}', \{t_2, t_3, t_4\})$. We note that this example is similar to that in Figure 2 and due to lack of space we do not describe it here again. However, we remark that it is expanded here with an extra MIMO transaction (i.e., $t_3(w_3)$) to show how we model transactions that involve multiple payers and multiple payees. Instead, we focus on the description of $\mathcal{G}_{\mathsf{BTC}} := (\mathcal{S} \dot\cup \mathcal{W}, \mathcal{E}_P \dot\cup \mathcal{E}_C)$, a transaction graph modeling the aforementioned Bitcoin execution as depicted in Figure 3b.

- $t_0 := (\{s_g, s_0, w\}, \{(s_g, w), (w, s_0)\})$, where $(s_g, w) \in \mathcal{E}_C$ and $(w, s_0) \in \mathcal{E}_P$. This represents the initialization transaction where $s_g := (\textsc{addr}_0, 21\text{M}, \mathsf{H}(\emptyset), 0)$, $w_0 := (\textsc{tinitx}, \emptyset)$ and $s_0 := (\textsc{addr}_m, 21\text{M}, \mathsf{H}(\{s_g, w_0\}), 0)$.
- $t_1 := (\{s_0, s_1, s_2, w_1\}, \{(s_0, w_1), (w_1, s_1), (w_1, s_2)\})$, where $(s_0, w_1) \in \mathcal{E}_C$ and $\{(w_1, s_1), (w_1, s_2)\} \subseteq \mathcal{E}_P$. A SIMO transaction that issues bitcoins to Alice after she has successfully mined a block. In a bit more detail, $w_1 := (\textsc{tcbtx}, \textsc{mp})$, $s_1 := (\textsc{addr}_m, (21\text{M}{-}50), \mathsf{H}(\{s_0, w_1\}), 1)$ and $s_2 := (\textsc{addr}_{Alice}, 50, \mathsf{H}(\{s_0, w_1\}), 1)$, where $\textsc{addr}_{Alice}$ denotes a Bitcoin address owned by Alice. We follow this notion in the rest of the example for the addresses owned by the example users. s_0 is defined as in t_0.
- $t_2 := (\{s_2, s_3, s_4, s_5, w_2\}, \{(s_2, w_2), (w_2, s_3), (w_2, s_4), (w_2, s_5)\})$, where $(s_2, w_2) \in \mathcal{E}_C$ and $\{(w_2, s_3), (w_2, s_4), (w_2, s_5)\} \subseteq \mathcal{E}_P$. A SIMO transaction that pays 2 bitcoins to Bob and the remaining bitcoins are sent back to Alice. In a bit more detail, $w_2 := (\textsc{trtx}, \{\textsc{cf}_{Alice}\})$, $s_3 := (\textsc{addr}'_m, 1, \mathsf{H}(\{s_2, w_2\}), 2)$, $s_4 := (\textsc{addr}_{Bob}, 2, \mathsf{H}(\{s_2, w_2\}), 2)$ and $s_5 := (\textsc{addr}'_{Alice}, 47, \mathsf{H}(\{s_2, w_2\}), 2)$. The rest of states are defined as in previous transactions.


t1: Alice mines block B1 (reward 50 bitcoins)
B1
Genesis Block Bg := {t0}
t2: Alice pays 2 bitcoins to Bob (fee 1 bitcoin)
50
47
2
Block B1 := {t1}
t3: {Alice, Bob} pay 3 bitcoins to Charles (fee 1 bitcoin)
47
44
2
1
3
Block B2 := {t2, t3, t4}
t4: Diana mines block B2 (reward 50 bitcoins)
...
B2
w0
s0
w1
s1
s2
w2
w4
s3
s4
s5
w3
s10
s11
s6
s7
s8
s9

(a) Example of Bitcoin execution $\mathcal{L}_{\mathsf{BTC}} := \{\mathrm{B}_g, \mathrm{B}_1, \mathrm{B}_2\}$. Only involved blocks and graphical description of transactions $t_1 - t_4$ have been shown. Here, t_0 represents the initialization transaction.

(b) Example of a $\mathcal{G}_{\mathsf{BTC}}$ instance.

Fig. 3: Illustrative example of our modeling of an execution of the Bitcoin system.

- $t_3 := (\{s_4, s_5, s_6, s_7, s_8, s_9, w_3\}, \{(s_4, w_3), (s_5, w_3), (w_3, s_6), (w_3, s_7), (w_3, s_8), (w_3, s_9)\})$, where $\{(s_4, w_3), (s_5, w_3)\} \subseteq \mathcal{E}_C$ and $\{(w_3, s_6), (w_3, s_7), (w_3, s_8), (w_3, s_9)\} \subseteq \mathcal{E}_P$. A MIMO transaction that pays 3 bitcoins to Charles, jointly by Alice and Bob. In a bit more detail, $w_3 := (\textsc{trtx}, \{\textsc{cf}'_{Alice}, \textsc{cf}_{Bob}\})$, $s_6 := (\textsc{addr}''_m, 1, \mathsf{H}(\{s_4, s_5, w_3\}), 2)$, $s_7 := (\textsc{addr}'_{Bob}, 1, \mathsf{H}(\{s_4, s_5, w_3\}), 2)$, $s_8 := (\textsc{addr}''_{Alice}, 44, \mathsf{H}(\{s_4, s_5, w_3\}), 2)$, and $s_9 := (\textsc{addr}_{Charles}, 3, \mathsf{H}(\{s_4, s_5, w_3\}), 2)$. The rest of states are defined as previous transactions.

- $t_4 := (\{s_1, s_3, s_6, s_{10}, s_{11}, w_4\}, \{(s_1, w_4), (s_3, w_4), (s_6, w_4), (w_4, s_{10}), (w_4, s_{11}\})$, where $\{(s_1, w_4), (s_3, w_4), (s_6, w_4)\} \in \mathcal{E}_C$ and $\{(w_4, s_{10}), (w_4, s_{11}\} \subseteq \mathcal{E}_P$. A MIMO transaction that issues bitcoins to Diana after she has successfully mined a block. Additionally, Diana claims the transaction fees for transactions t_2 and t_3. In a bit more detail, $w_4 := (\textsc{tcbtx}, \textsc{mp})$, $s_{10} := (\textsc{addr}'''_m, (21\mathrm{M} - 100), \mathsf{H}(\{s_0, w_4\}), 2)$, and $s_{11} := (\textsc{addr}_{Diana}, 52, \mathsf{H}(\{s_0, w_4\}), 2)$.

B Transaction graph for Hyperledger Fabric

B.1 Detailed description

In this section, we study the Hyperledger Fabric (HLF) [6] blockchain-based system. We start by the description of a execution of HLF. An execution of HLF is represented as a set of blockchains, one per channel. However, as each single blockchain evolves independently from each other, we restrict our description here to a single blockchain. This description, however, can be easily extended to model a HLF execution with multiple channels.

A blockchain is composed of blocks. We denote the first block as *genesis* block and each subsequent block is created by the *ordering service*. Such ordering service chooses the sorted set of transactions to be included in each block. HLF supports two types of transactions: *Init* and *Invoke*. An *init* transaction is included in the genesis block and it is used to initialize every key used in the blockchain to a default value − and includes an *endorsement policy*, that determines how any subsequent transaction should be authorized. We consider that an initialization transaction is always valid.

An *invoke* transaction is used to carry out updates in a set of key-value pairs for the local current key-value store (KVS) through two operations: (i) $\textsc{GetState}(k) \to v$, that given a key k provides the most current value v associated to it; and (ii) $\textsc{PutState}(k, v)$, that updates value associated to a given key k to the newly provided value v. An *invoke* transaction is valid if it contains enough *endorsements* from the set of *endorsers* specified in the *endorsement policy*.

Definition 13 (HLF execution). *A HLF execution $\mathcal{L}_{\mathsf{HLF}}$ is a set of blocks $\mathcal{B} := \{\textsc{b}_g, \textsc{b}_1, \ldots, \textsc{b}_n\}$, where $\textsc{b}_g$ denotes the genesis block that contains a single init transaction, denoted by $\textsc{initx}$. Each block $\textsc{b}_i := \{\textsc{invtx}_1, \ldots, \textsc{invtx}_n\}$ is a set of invoke transactions $\textsc{invtx}_i$. An $\textsc{initx}$ contains a single endorsement policy $\textsc{ep}$. Each transaction $\textsc{invtx}_i$ is defined as a tuple ($\mathcal{F}$, $\mathcal{U}$), where $\mathcal{F}$ denotes the set of endorsements ($\{\textsc{end}_1, \ldots, \textsc{end}_n\}$), and $\mathcal{U}$ denotes the set of $\{\textsc{GetState}(\star), \textsc{PutState}(\star, \star)\}$ operations to update key-value pairs.*

We continue by describing the modeling of a HLF execution. Informally, each state in our model represents a key-value pair. Each witness represents the set of endorsements required for a transaction to be valid. Finally, here we consider three type of edges: observing, consuming and producing edges. An observing edge links a key k to the endorsement specified in a transaction that reads k but does not modify it (e.g., an invoke transaction that contains only a $\textsc{GetState}(k)$ operation). If the key k is modified (e.g., an invoke transaction that contains $\textsc{PutState}(k, \star)$ operation), a consuming edge links then the key k with the endorsements for such transaction. Finally, a producing edge links the endorsements to a key k a transaction has modified it (e.g., by means of a $\textsc{PutState}(k, \star)$ operation).

Definition 14 (Transaction graph). *We model a HLF execution $\mathcal{L}_{\mathsf{HLF}}$ as a graph $\mathcal{G}_{\mathsf{HLF}} := (\mathcal{S}_{\mathsf{HLF}} \mathbin{\dot\cup} \mathcal{W}_{\mathsf{HLF}}, \mathcal{E}_{\mathsf{HLF}})$ defined as follows:*

States: *Each state $s \in \mathcal{S}_{\mathsf{HLF}}$ is defined as a tuple (key, version), where key denotes the key part of a key-value pair and version denotes the current version number of the key-value pair. The genesis state is defined as $s_g := (params, 0)$ and denotes a special key-value pair that holds the configuration parameters for a channel as indicated in channel initialization.*

Witness: *Each witness $w \in \mathcal{W}_{\mathsf{HLF}}$ is defined as a tuple (txtype, $\mathcal{F}$), where txtype set to* $\textsc{tinitx}$ *indicates an init transaction and set to* $\textsc{tinvtx}$ *indicates an invoke transaction. $\mathcal{F}$ denotes an endorsement policy* $\textsc{ep}$ *if txtype $=$* $\textsc{tinitx}$ *or a set of endorsements $\{\textsc{end}_i\}$ if txtype $=$* $\textsc{tinvtx}$*. For simplicity, we assume that an endorsement* $\textsc{end}$ *also contains the corresponding set of operations* $\textsc{GetState}(\star)$ *and* $\textsc{PutState}(\star, \star)$*.*

Edges: *Each edge $e \in \mathcal{E}_{\mathsf{HLF}}$ is defined as either observing, consuming or producing edge.*

Definition 15 (Transaction types). *An invoke transaction is modeled as a SISO, MISO or MIMO transaction depending on the set of operations* $\textsc{GetState}(\star)$ *and* $\textsc{PutState}(\star, \star)$ *that it uses. For instance, a SISO transaction models a transaction that uses a single* $\textsc{PutState}(k, \star)$ *operation for a key k. A MISO transaction models a transaction that updates a single key k and reads at least one additional key k' (e.g.,* $\{\textsc{GetState}(k), \textsc{PutState}(k', v)\}$*). Finally, a MIMO transaction models a transaction that updates several keys and possibly reads other additional keys (e.g.,* $\{\textsc{GetState}(k), \textsc{PutState}(k', v), \textsc{PutState}(k'', v')\}$*). An init transaction is of type INIT and is defined as $t := (\{s_g, w\} \dot\cup \{s_i\}, \{(s_g, w)\} \dot\cup \{(w, s_1), \ldots, (w, s_n)\})$, where $(s_g, w) \in \mathcal{E}_C$, $\{(w, s_1), \ldots, (w, s_n)\} \subseteq \mathcal{E}_P$, $w := (\textsc{tinitx}, \textsc{ep})$, and each $s_i := (k_i, -)$. The genesis state s_g is defined in Definition 14.*

We make two observations in the definition of the transaction types. First, MISO and MIMO types are restricted in the sense that they must have the same number of consuming and producing edges. This is due to the fact that we model each $\textsc{PutState}(\star, \star)$ operation as a consuming edge from the state of the key being updated and a producing edge to the state corresponding to the updated key-value pair. We note, however, that this is a characteristic inherent to all systems based on key-value stores and not a particular limitation of HLF.

Second, as any system based in a key-value store, each key must exist only once. For that, we model our initialization transaction such that all the keys used in the given HLF's execution are created and initialized to a fixed initial value $(-)$.

Now, we finalize the description of our model by defining the transaction predicate for HLF. Here, we denote by $\mathsf{VerifyEndorsement}(\{\textsc{end}_i\})$ a boolean function that takes a set of endorsements $\{\textsc{end}_i\}$ and returns $\textsc{True}$ if $\{\textsc{end}_i\}$ represents a valid set of endorsements according to the endorsement policy $\textsc{ep}$, and $\textsc{False}$ otherwise. Here, we assume that $\textsc{ep}$ is obtained from the initialization transaction included in the corresponding HLF execution.

Definition 16 (Transaction predicate). *Consider a transaction $t := (\mathcal{S} \dot\cup \{w\}, \mathcal{E}_O \dot\cup \mathcal{E}_P \dot\cup \mathcal{E}_C)$. Then, $\mathbb{P}(t)$ returns* $\textsc{True}$ *if the following conditions hold and* $\textsc{False}$ *otherwise.*

1. *If t is an invoke transaction, the witness must contain a set of valid endorsements, i.e., $w.txtype = \textsc{tinvtx} \Rightarrow \mathsf{VerifyEndorsement}(w.\mathcal{F})$.*
2. *If t is an invoke transaction, each output state must represent an update of a key included in a input state. Moreover, the version number for the output state must be bigger than the version number for the input state representing the same key, i.e., $w.txtype = \textsc{tinvtx} \Rightarrow \forall s' \in w\mathcal{E}_P\star, \exists s \in \star\mathcal{E}_C w : s'.key = s.key \wedge s'.version > s.version$.*

B.2 Model Analysis

In this section we analyze our model for the execution of the HLF system. We start by showing that any legal HLF execution modeled as aforementioned results in a TDAG. Here, we consider as legal a HLF execution that contains only blocks included in the blockchain that have been produced by the ordering service.

Theorem 5. *Assume that $\mathcal{L}_{\mathsf{HLF}}$ is a legal HLF execution. Then, the $\mathcal{G}_{\mathsf{HLF}}$ instance modeling $\mathcal{L}_{\mathsf{HLF}}$ is a TDAG.*

Proof. Here, we show that $\mathcal{G}_{\mathsf{HLF}}$ fulfills all the conditions required in Definition 2.

1. The genesis state must not have any producing or observing edges and it must have a single producing edge. This condition is ensured by our definition of initialization transaction.
2. Every state, other than the genesis, must have a single producing edge. Assume by contradiction that $\exists s \in \mathcal{S}_{\mathsf{HLF}} \setminus \{s_g\} : |\star\mathcal{E}_P s| > 1$.[3] This implies that there are at least two transactions t and t' in $\mathcal{G}_{\mathsf{HLF}}$ that update the same key-value pair simultaneously. This, however, contradicts the assumption that a valid execution contains only transactions *sorted* by an ordering service.
3. Every state other than the genesis can have multiple successors, but at most one among them is connected with a consuming edge. The proof for this condition holds along the same lines as for the previous condition.
4. The graph must be weakly connected. Each new transaction reads our updates a key represented by an unconsumed state in the graph. Therefore, the overall graph is weakly connected.
5. The graph must not have cycles. Assume by contradiction that there is a cycle in $\mathcal{G}_{\mathsf{HLF}}$. This necessarily implies that there are two transactions that produce the same state. However, as we argued before, this contradicts the fact that the ordering service establishes a total order among the transactions.

As we did with the Bitcoin model, here we show that validating HLF transactions individually suffices to reason about properties of the complete HLF execution. In particular, we show that if $\mathcal{G}_{\mathsf{HLF}}$ is a valid TDAG, then the highest version number (i.e., most recent) for any given key is represented in a unconsumed state of $\mathcal{G}_{\mathsf{HLF}}$.

[3] We rule out the case $|\star\mathcal{E}_P s| = 0$ because a state only exists in $\mathcal{G}_{\mathsf{HLF}}$ if it has been produced by a transaction.

Definition 17 (Most recent key-value pairs). *Consider that $\mathcal{G}_{\mathsf{HLF}}$ is a TDAG modeling a HLF execution. Then, we define* the states representing the most recent key-value pairs *as the set of states with the highest version number for each key, i.e.,* $\{s \in \mathcal{S}_{\mathsf{HLF}} : s' \in \mathcal{S}_{\mathsf{HLF}} \land s.key = s'.key \Rightarrow s.version > s'.version\}$*.*

Theorem 6 (Unconsumed states represent most recent key-value pairs). *Assume that $\mathcal{G}_{\mathsf{HLF}}$ is a valid TDAG and models a legal HLF execution $\mathcal{L}_{\mathsf{HLF}}$. Then, the unconsumed states of $\mathcal{G}_{\mathsf{HLF}}$ represent the most recent key-value pairs.*

Proof. Assume by contradiction that Theorem 6 does not hold. Then, there must exist at least a transaction where the *version* field in the output state for a key is smaller than the *version* field in the input state for the same key, i.e., $\exists t := (\mathcal{S} \dot\cup \{w\}, \mathcal{E}) \in \mathcal{T}_{\mathcal{G}_{\mathsf{HLF}}}, \exists s, s' \in \mathcal{S} : (s, w) \in \mathcal{E} \land (w, s') \in \mathcal{E} \land s.key = s'.key \land s.version < s'.version$. However, $\mathbb{P}(t)$ would return FALSE, which contradicts the fact that $\mathcal{G}_{\mathsf{HLF}}$ is a valid TDAG.

B.3 Modeling an Example of Execution for HLF

Here we describe how we model an illustrative example of HLF execution. We assume for simplicity that the endorsement policy requires a single endorsement for each transaction.

Throughout our description, we focus in the illustrative example depicted in Figure 4. In particular, Figure 4b shows the HLF execution $\mathcal{L}_{\mathsf{HLF}} := \{\textsc{b}_g, \textsc{b}_1, \textsc{b}_2\}$, where $\textsc{b}_g := \{t_0 := \textsc{ep}\}$, $\textsc{b}_1 := \{t_1 := (\textsc{end}, f_1)\}$ and $\textsc{b}_2 := \{t_2 := (\textsc{end}', f_2)\}$. In a bit more detail, t_0 represents the initialization transaction that initializes the key-value pairs used later in the execution and it is included in the genesis block. Moreover, t_1 represents an invoke transaction that calls the function f_1 and t_2 represents another invoke transaction that calls f_2 in this case.

Now, we describe how we model such HLF execution as an instance of $\mathcal{G}_{\mathsf{HLF}}$ as shown in Figure 4c:

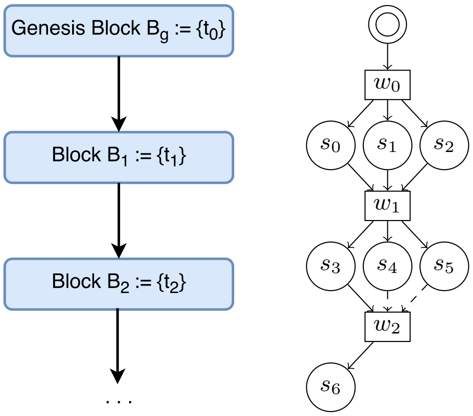
- $t_0 := (\{s_g, s_0, s_1, s_2, w_0\}, \{(s_g, w_0), (w_0, s_0), (w_0, s_1), (w_0, s_2)\})$, where $(s_g, w_0) \in \mathcal{E}_C$ and $\{(w_0, s_0), (w_0, s_1), (w_0, s_2)\} \subseteq \mathcal{E}_P$. It represents the initialization transaction as described above. In more detail, $s_g := (params, 0)$, $w_0 := (\textsc{tinitx}, \textsc{ep})$, $s_0 := (a, 1)$, $s_1 := (b, 1)$ and $s_2 := (c, 1)$.
- $t_1 := (\{s_0, s_1, s_2, s_3, s_4, s_5 w_1\}, \{(s_0, w_1), (s_1, w_1), (s_2, w_1), (w_1, s_3), (w_1, s_4), (w_1, s_5)\})$, where $\{(s_0, w_1), (s_1, w_1), (s_2, w_1)\} \subseteq \mathcal{E}_C$ and $\{(w_1, s_3), (w_1, s_4), (w_1, s_5)\} \subseteq \mathcal{E}_P$. A MIMO transaction that updates the values associates to keys a, b, c. In a bit more detail, $w_1 := (\textsc{tinvtx}, \textsc{end})$, $s_3 := (a, 2)$, $s_4 := (b, 2)$ and $s_5 := (c, 2)$. The rest of states are defined as described for t_0.
- $t_2 := (\{s_3, s_4, s_5, s_6, w_2\}, \{(s_3, w_2), (s_4, w_2), (s_5, w_2), (w_2, s_6)\})$, where $(s_3, w_2) \in \mathcal{E}_C$, $\{(s_4, w_2), (s_5, w_2)\} \subseteq \mathcal{E}_O$ and $(w_2, s_6) \in \mathcal{E}_P$. A MISO transaction that reads the values associates to keys a, b, c, and updates the value associated to key a. In a bit more detail, $w_2 := (\textsc{tinvtx}, \textsc{end}')$ and $s_6 := (a, 3)$. The rest of states are defines as described for t_1.

Algorithm 1: Function f_1

```
PutState ("a", 0);
PutState ("b", 5);
PutState ("c", 3);
```

Algorithm 2: Function f_2

```
$a \leftarrow$ GetState ("a");
$b \leftarrow$ GetState ("b");
$c \leftarrow$ GetState ("c");
PutState ("a", $a + b + c$);
```



(a) Set of GetState and PutState operations for each function defined in this example.

(b) Example HLF execution $\mathcal{L}_{\mathsf{HLF}} := \{\mathrm{B}_g, \mathrm{B}_1, \mathrm{B}_2\}$. Here, t_1 invokes f_1 and t_2 invokes f_2.

(c) Example of $\mathcal{G}_{\mathsf{HLF}}$ instance.

Fig. 4: Illustrative example of the modeling of an execution of HLF. We model an execution that contains the setup transaction (t_0), followed by an invocation to f_1 (modeled in t_1) and finally an invocation to f_2 (modeled in t_2).