Mechanized Formal Model of Bitcoin's Blockchain Validation Procedures

Kristijan Rupić
Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia
kristijan.rupic@fer.hr

Lovro Rožić
Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia
lorozic33@gmail.com

Ante Derek
Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia
ante.derek@fer.hr

Abstract

We present the first mechanized formal model of Bitcoin's transaction and blockchain data structures including the formalization of the blockchain validation procedures. Our formal model, though still a simplified representation of an actual Bitcoin blockchain, includes regular and coinbase transactions, segregated witnesses, relative and absolute locktime, the Bitcoin Script language expressions together with a denotational semantics, transaction fees and block rewards. We formally specify the details of validity checks performed when adding new blocks to the blockchain. We assume perfect cryptography and use the symbolic approach for modeling hash functions and digital signatures.

To demonstrate the utility of the model, we formally state and prove several essential properties of a valid blockchain – transactions are unique, each coin can be spent at most once and the new value is only created through block rewards. The model and the proofs are largely independent of Bitcoin specific details and easily generalize to any cryptocurrency blockchain based on the Unspent Transaction Output (UTXO) paradigm.

We mechanize all the results using the Coq proof assistant.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases blockchain, Bitcoin, program verification, Coq

Digital Object Identifier 10.4230/OASIcs.FMBC.2020.7

Supplementary Material The Coq artifacts are available as as an open-source project at `https://github.com/krupic2402/bitcoin-blockchain-validation-formal-model`.

Funding *Ante Derek*: This research has been partially supported by the European Regional Development Fund under the grant KK.01.1.1.01.0009 (DATACROSS).

1 Introduction

In the past decade, due to the popularity of Bitcoin [18] and other cryptocurrencies, as well as new applications such as smart contracts [10], blockchain systems have attracted significant attention from the scientific community. The blockchain systems implement *distributed ledgers* where the data and transaction integrity is enforced using cryptography and consensus mechanisms.

Despite the openness of the Bitcoin system, serious design and implementation flaws have been discovered over the years. For example, a simple design flaw made it possible to include two different *coinbase* transactions with the same transaction identifier (TXID) into the blockchain [2]. The flaw was subsequently fixed in two *Bitcoin Improvement Proposals*: BIP 30 [2] made the older of the two transactions unspendable and included explicit checks for uniqueness of TXID's, BIP 34 [3] mandated that coinbase transactions must include block height information, thereby fixing the design flaw. More recently (and more seriously), an implementation error in the transaction and block verification logic of the official Bitcoin client [5] made it possible for malicious miners to launch double-spending attacks.

In this paper, we build a formal model of Bitcoin's blockchain validation logic and we fully mechanize it using the Coq proof assistant [24]. We use the model to verify essential properties of a valid blockchain including the absence of both flaws described above.

We use the formal model of Bitcoin transactions by Atzei et al. [9] as the reference point for our formalization and mechanization efforts. We extend the simple "blockchain" model (i.e. a simple list of transactions) of [9] by adding an explicit *blockchain* data structure containing blocks of transactions linked by hash pointers. Our model also includes the complete treatment of coinbase transactions, the block height information as mandated by BIP34, transaction fees and block rewards. Finally, we model the blockchain validation procedures by formally specifying the sanity and validity checks performed by Bitcoin clients when adding new blocks; we define the blockchain to be *valid* if it passes the said validation procedures.

Contributions

Contributions of this paper are as follows:

1. We propose a fully mechanized model for Bitcoin transaction and the blockchain data structures. While simplified, the model includes many important details such as multi-signatures, segregated witnesses, absolute and relative locktimes, coinbase transactions, transaction fees and block rewards.
2. We define a denotational semantics for symbolic typed variant of Bitcoin Script language.
3. We define the sanity and validity checks performed by clients when adding new blocks to the blockchain.
4. We demonstrate the utility of the model by giving machine-verified proofs for three essential properties of a valid blockchain – same coin cannot be spent twice, transactions are unique, the total value of unspent coins is equal to the total value of block rewards.
5. We mechanize all the above results using the Coq proof assistant.

We make a number of simplifying assumptions in our work. First, we use the Dolev-Yao [15] model of cryptography where hash functions and digital signatures are abstract operations with perfect security properties. We simplify the Blockchain data structure by ignoring the Merkle trees that are normally used to include transactions and witnesses in block headers. Instead of a stack-based Script language and the corresponding execution model, we formalize the output scripts using an expression language with typed denotational semantics. Finally, many important aspects of the Bitcoin system such as the proof-of-work consensus mechanism, peer-to-peer network protocol, transaction and block discovery methods, etc. are out of scope of this work. Note that, there are efforts underway to mechanize those aspects of the Bitcoin system [22] – we view them as complementary to results presented in this paper.

We assume the reader is familiar with the Bitcoin system in general as well as the details of transaction and blockchain data structures including the notions of *inputs*, *outputs*, *witness* scripts and *coinbase transactions*. Due to space constraints, we defer details for the several aspects of the formal model (e.g., the semantics of the script language expressions) as well as proofs to the Coq artifacts.

$$\texttt{Satoshi}, \texttt{Index}, \texttt{Time} \triangleq \mathbb{N} \tag{1}$$

$$\texttt{PK}, \texttt{SK} \triangleq \mathbb{N} \tag{2}$$

$$\mathit{is\_key\_pair} : \texttt{PK} \rightarrow \texttt{SK} \rightarrow \texttt{bool} \tag{3}$$

$$\texttt{Modifier} \triangleq \{aa, an, as, sa, sn, ss\} \tag{4}$$

Figure 1 Basic definitions, key pairs and hash flags.

Outline

Section 2 presents our model of Bitcoin transactions formalized using the Coq proof assistant. In Section 3 we give the formal model of the blockchain data structure. In Section 4 we use the model to provide machine-verified proofs for the essential properties of a valid blockchain. In Section 5 we discuss the limitations of our model. We address related work in Section 6 and conclude in Section 7.

2 Formal Model of Bitcoin Transactions and Blockchain

We present a Coq model of the Bitcoin blockchain and the Bitcoin Script language. For now, we are primarily interested in transaction and blockchain validity.

Notation

For some type τ we use τ^* to denote the type of lists of elements of type τ. We denote the empty list as $[]$ and the singleton list containing some element x by $[x]$. We use '+' to denote list concatenation, $|\cdot|$ to denote list length, and $\in$ to denote list membership. Dot notation is used to denote access to individual members of structures. For example, we write $T.wit(i)$ to access the i-th index of the witness field of some transaction T. We will abbreviate $T.stub.inputs$ with $T.inputs$ (and similarly with other fields of transaction stubs). These notations might differ slightly from our Coq code but correspond to it in a one-to-one fashion.

2.1 The Transaction Model

We start out with a model of transactions and transaction histories, i.e., lists of transactions ordered by logical time. We model the Bitcoin Script language in order to provide an end-to-end model of transaction verification, although the proofs of various properties of our model could be made parametric with respect to a choice of the script language with relative ease, since their details tend to not affect higher-level properties.

As mentioned in the introduction, we use the symbolic approach when modeling cryptographic primitives. This allows us to simplify hashes of objects to only the objects themselves equipped with a decidable equality predicate, making the hash function essentially be the identity function which is injective and therefore also collision-resistant in a trivial way.

We begin by listing the basic definitions (Figure 1) which we will use throughout the rest of the formalization. Amounts of money (Satoshis, the name of the smallest Bitcoin denomination) and logical time in the system are both modeled as natural numbers for simplicity (1). Next, we define key pairs (2) for public-key digital signatures as trivial inductive types wrapping a value with decidable equality (in particular, a natural number) and we define a public and secret key to belong to the same pair if and only if they wrap equal values (3). We also define modifiers (4) corresponding to SIGHASH flags used in transaction signing [9].

Next, we need to define transactions (22). A regular transaction definition should consist of at least the following: a list of transaction inputs (16); a list of transaction outputs (17); a list of witness data associated with the inputs (24). Since we model SegWit [4], in our model we will distinguish between transactions and transactions paired with their respective witnesses depending on the context. The model of transactions also includes the absolute lock time (18) (nLockTime), which is a constraint on the earliest time the transaction can appear in a valid blockchain. While Bitcoin allows this to be either a block height or a UNIX timestamp depending on the range of the value [1], we only model some abstract, logical time. Extending the model to allow for block height or UNIX time should be fairly trivial. Unlike Atzei et al. [9], we also model coinbase transactions explicitly. They contain outputs but no inputs. These outputs represent the reward for mining of blocks and should be the sole supply of money in the system. They also contain their *block height*, i.e., the number of the block they are contained in in order to make them distinct as in BIP 34 [3].

Inputs (16) are references to outputs of other transactions, i.e., pairs of the referenced transaction and an index into its output list, along with a relative lock time which is another temporal constraint used in transaction verification. Unlike a Bitcoin implementation, this reference contains referenced transactions themselves instead of their hashes. Therefore, we require a decidable equality predicate on transactions, as well as an induction principle for its proof of correctness; we write an induction principle for transactions and their mutually inductive types manually. This is due to the fact that our inductive datatypes contain lists of the datatypes themselves – this creates an implicit mutual induction with lists which needs an induction principle more involved than ones Coq can automatically generate. This could have been avoided had we inlined lists (i.e. made our own datatypes using constructors analogous to *cons* and *nil*), however we would lose access to various existing theorems about lists contained in the standard library.

A transaction output (17) consists of its value in Satoshis and a script (5) for the verification of attempts to redeem the output. The Bitcoin Script language is a stack-based language that is used to write output scripts that verify that the conditions for redeeming the output are met. A script takes a fixed number of inputs which depends on the commands used; these inputs are called the witness and a redeeming transaction must provide them. Following Atzei et al. [9], we model the script language as an expression-based language instead as that allows us to easily specify denotational semantics for the scripts.

In a Bitcoin implementation all script values are just byte vectors at most 520 bytes long and their interpretation is made by the stack commands as either numbers, truth values, signatures, hashes etc. As we model hashes and signatures symbolically, we need our script input value type `StackValue` (9) to represent those possibilities as well, so we choose to impose a rudimentary type system on the values and their denotations that allows for integers (10), booleans (11), transaction signatures (13), and hashes of any type of value (12). As a transaction signature (26) is simply a wrapper for a secret key and a transaction "hash", a value will possibly contain transactions as well, making `StackValue` mutually inductive with transactions in our model (Figure 2).

The output script expression language is relatively simple. Most notable expression types are variables (6), constants of any `StackValue` (7), a multi-signature verification primitive (8) and several other arithmetic and comparison operations. We model it with an inductive type `Exp` (5) mutually inductive with `StackValue` and `TxStub` due to the fact that arbitrary

$$\begin{aligned}
\texttt{Exp} : \texttt{Set} &::= && (5)\\
\textit{e\_var} : \texttt{string} &\to \texttt{Exp} && (6)\\
\textit{e\_const} : \texttt{StackValue} &\to \texttt{Exp} && (7)\\
\textit{e\_plus} : \texttt{Exp} &\to \texttt{Exp} \to \texttt{Exp}\\
\textit{e\_minus} : \texttt{Exp} &\to \texttt{Exp} \to \texttt{Exp}\\
\textit{e\_equal} : \texttt{Exp} &\to \texttt{Exp} \to \texttt{Exp}\\
\textit{e\_less} : \texttt{Exp} &\to \texttt{Exp} \to \texttt{Exp}\\
\textit{e\_if} : \texttt{Exp} &\to \texttt{Exp} \to \texttt{Exp} \to \texttt{Exp}\\
\textit{e\_length} : \texttt{Exp} &\to \texttt{Exp}\\
\textit{e\_hash} : \texttt{Exp} &\to \texttt{Exp}\\
\textit{e\_versig} : \texttt{PK}^* &\to \texttt{Exp}^* \to \texttt{Exp} && (8)\\
\textit{e\_abs\_after} : \texttt{Time} &\to \texttt{Exp} \to \texttt{Exp}\\
\textit{e\_rel\_after} : \texttt{Time} &\to \texttt{Exp} \to \texttt{Exp}
\end{aligned}$$

$$\begin{aligned}
\texttt{StackValue} : \texttt{Set} &::= && (9)\\
\textit{sv\_int} : \mathbb{Z} &\to \texttt{StackValue} && (10)\\
\textit{sv\_bool} : \texttt{bool} &\to \texttt{StackValue} && (11)\\
\textit{sv\_hash} : \texttt{StackValue} &\to \texttt{StackValue} && (12)\\
\textit{sv\_sig} : \texttt{TxStub} &\to \texttt{SK} \to \texttt{Modifier}\\
&\to \texttt{Index} \to \texttt{StackValue} && (13)\\
\texttt{TxStub} : \texttt{Set} &::= && (14)\\
\textit{tx\_stub}\ &\{ && (15)\\
\textit{inputs} : &(\texttt{TxStub} \times \texttt{Index} \times \texttt{Time})^*; && (16)\\
\textit{outputs} : &(\texttt{Exp} \times \texttt{Satoshi})^*; && (17)\\
\textit{absLock} : &\texttt{Time}\ \} && (18)\\
\textit{coinbase}\ &\{ && (19)\\
\textit{block\_height} : &\mathbb{N}\ ; && (20)\\
\textit{outputs} : &(\texttt{Exp} \times \texttt{Satoshi})^*\ \} && (21)
\end{aligned}$$

$$\begin{aligned}
\texttt{Tx} : \texttt{Set} &::= \textit{tx}\ \{ && (22)\\
\textit{stub} : &\ \texttt{TxStub}; && (23)\\
\textit{witnesses} : &\ (\texttt{StackValue}^*)^*\ \} && (24)
\end{aligned}$$

Figure 2 Mutually inductive transaction, witness value and script definition.

StackValues can be contained as constants in the expressions, which is made necessary by our imposed type system in order to meaningfully define arithmetic and comparison operations. The final result are three mutually inductive types (Figure 2) together with their mutual induction principle.

The witnesses (24) are data associated with each input. When verifying a redeeming attempt, they are used as the initial stack value in the output scripts of their associated inputs. Note that it is impossible to sign the witnesses along with the rest of transaction due to the fact that usually the witness data needs to contain the transaction signature itself. Not signing the witnesses implies that they can be changed before being included in a block, changing the hash of the transaction with witnesses included, a problem known as transaction malleability. This was resolved by the implementation of a protocol upgrade called SegWit (Segregated Witness) introduced by BIP141 [4]. We account for these subtleties in our model by separating the witnesses from input data in our model as well. In implementations of SegWit the witnesses are moved outside transaction data structures into their own Merkle tree stored in the containing block's coinbase transaction. To be able to talk about transaction history validity, we will sometimes have to associate transactions with their corresponding witnesses regardless of SegWit; to achieve this, we separate the transaction model into two layers of inductive types: the type TxStub (14) containing the transaction data save for the witnesses, and full transaction Tx (22) containing its stub and a list of witnesses (24). The transaction hash for input referencing purposes (TXID) is modeled by the the TxStub type.

$$\texttt{TxStubHash} ::= \mathit{tx\_hash} : \texttt{TxStub} \to \texttt{Modifier} \to \texttt{Index} \to \texttt{TxStubHash} \tag{25}$$
$$\texttt{Sig} ::= \mathit{sig} : \texttt{SK} \to \texttt{Modifier} \to \texttt{Index} \to \texttt{TxStub} \to \texttt{Sig} \tag{26}$$
$$\mathit{ver} : \texttt{PK} \to \texttt{Sig} \times \texttt{Modifier} \to \texttt{TxStub} \to \texttt{Index} \to \texttt{bool} \tag{27}$$
$$\mathit{multi\_ver} : \texttt{PK}^* \to (\texttt{Sig} \times \texttt{Modifier})^* \to \texttt{TxStub} \to \texttt{Index} \to \texttt{bool} \tag{28}$$

Figure 3 Signatures and routines for their verification.

2.2 Signature Verification and Output Redeeming

We now define our model of transaction signatures and their verification (Figure 3). A transaction signature is the SK-signed hash of a transaction with some fields disregarded in a way controlled by SIGHASH flags; in particular, some of the inputs are disregarded depending on the exact flags. We model hashes computed in this manner with the inductive type `TxStubHash` (25) wrapping the hashed transaction and the hashing flags, along with a comparison predicate which is based on transaction stub equality modulo hash flags. Signatures are represented by the inductive type `Sig` (26) wrapping everything a `TxStubHash` wraps, as well as the secret key. A signature needs to be paired with the hash flags used to compute it as they affect the result and are required for checking; this is implemented in Bitcoin by appending a byte denoting the hash flags to the signature. We model this explicitly by using `Sig` $\times$ `Modifier` even though we could introspect our inductive wrappers for their value.

We proceed to define single (27) and multiple (28) signature verification routines. We model successful signature verification with a public key using a simple check for pairedness of the given public key with the wrapped secret key with the function *is_key_pair* (3), and a check for hash equality by comparing both `TxStubHash` and the hash flags for equality; the verification succeeds if all comparisons do. Multiple signature verification tries to verify a list of signatures, in order, using an ordered list of public keys. The procedure repeatedly calls the single signature verification routine for each signature with successive public keys from the list until success, or until all public keys have been exhausted and no matching keys have been found; the whole routine succeeds if all signatures have been successfully verified and fails otherwise.

We define a straightforward denotational semantics for the script language based on Atzei et al. [9]. We impose a type system onto the values appearing in the script language (which are untyped in Bitcoin), consisting of the same types as `StackValue`, as well as a bottom type denoting failed computations or invalid types. We define the context of a witness $\mathit{make\_context}\ e\ T.\mathit{wit}(i)$ to be the mapping from variables $(\mathit{free\_vars}\ e)$ to the values in the witness. The order of the variables is determined by a preorder traversal of the expression's syntax tree. The denotation of a script expression depends on the redeeming transaction, the index of the redeeming input and the context constructed from the corresponding witness. In the definition below, *den_bool* is a constructor for denotational values which wraps a boolean value. We refer the reader to the Coq development for details due to space constraints.

▶ **Definition 1** (Script verification). *We say a transaction T's i-th input verifies a script e if:*

$$\mathit{verifies}(T, i, e) \triangleq |\mathit{free\_vars}\ e| = |T.\mathit{wit}(i)| \wedge [\![e]\!]_{T,i,\mathit{make\_context}\ e\ T.\mathit{wit}(i)} = \mathit{den\_bool}\ \texttt{true}.$$

$$\begin{aligned}
&\texttt{TxHistory} \triangleq (\texttt{Tx} \times \texttt{Time})^* && (29) \\
&UTXO : \texttt{TxHistory} \to (\texttt{TxStub} \times \texttt{Index})^* && (30) \\
&STXO : \texttt{TxHistory} \to (\texttt{TxStub} \times \texttt{Index})^* && (31)
\end{aligned}$$

$$\begin{aligned}
&sum\_inputs : \texttt{TxStub} \to \texttt{Satoshi} && (32) \\
&sum\_outputs : \texttt{TxStub} \to \texttt{Satoshi} && (33) \\
&UTXO\_value : \texttt{TxHistory} \to \texttt{Satoshi} && (34) \\
&coinbase\_value : \texttt{TxHistory} \to \texttt{Satoshi} && (35) \\
&coinbase\_height : \texttt{TxHistory} \to \mathbb{N} && (36)
\end{aligned}$$

Figure 4 Transaction history model.

▶ **Definition 2** (Output redeeming). *We say the j-th input of transaction T_2 at logical time t_2* redeems *the i-th output of transaction T_1 at logical time t_1 for a value of v Satoshis if:*

$$\begin{aligned}
&redeems(T_1, i, t_1, v, T_2, j, t_2) \triangleq \\
(i)\quad &\exists\, relLock\ e, T_2.inputs(j) = (T_1, i, relLock)\ \wedge\ T_1.outputs(i) = (e, v)\ \wedge \\
(ii)\quad &T_2.absLock \leq t_2\ \wedge\ t_1 + relLock \leq\ t_2\ \wedge \\
(iii)\quad &verifies(T_2, j, e)
\end{aligned}$$

3 Blockchain Model and Validity

We begin our model of the Bitcoin blockchain by first considering transaction histories and their validity. We then define our model of the blockchain and its validity by requiring that the transaction history encoded by the blockchain be valid, among other things.

For Bitcoin to function as a currency, it is crucial to control the way in which money is created. Only coinbase transactions should increase the total sum of money in the system. However, if a transaction output was to be spent more than once, it would essentially act as duplicated money. Therefore, it is necessary to ensure that transaction outputs can be spent at most once. Transactions attempting to spend an already spent output, or spend an unspent output multiple times at once must be disallowed in a valid transaction history. We provide a formal definition of the transaction history validity predicate that enforces this and certain other conditions necessary for a history to be considered valid. We later prove that this property indeed implies that no double spending of transaction outputs is happening within a valid history, as well as that the total sum of unspent transaction outputs never exceeds supply, i.e., the sum of coinbase outputs.

We define a transaction history (29) as a list of transactions with witnesses and the logical time at which they occur. We also define the notions of spent and unspent transaction outputs; an output at index i of a transaction T_1 in the blockchain is *unspent* in a history TH if there is no transaction anywhere in TH that has an input (T_1, i), whereas an output is *spent* in TH if such a transaction and input exist. We define functions $STXO$ and $UTXO$ (31, 30) on histories that compute respectively the list of spent and unspent outputs, with outputs represented as pair of the containing transaction and the output's index. We also formally prove the obvious fact that every output of every transaction in a blockchain is either spent or unspent. We define the sum of values of inputs (32) and outputs (33) of a transaction, as well as the sum of values of all UTXO-s (34) and all coinbase outputs (35) in a transaction history which should represent the total supply of money in a transaction history following some validity rules which we will define. We define *coinbase_height* to be the number of coinbase transactions in a transaction history; note that this is going to be equal to the block height, but is formalized independently.

Using the work of Atzei et al. [9] as a reference point, we define *transaction history validity* (4) inductively by requiring that each valid transaction history is formed by a sequence of *valid updates* (3) each extending the history by a single transaction in a way that enforces the necessary invariants.

▶ **Definition 3** (Valid update for transaction histories).

$$
\begin{aligned}
&is\_valid\_update(TH, T, t) \triangleq \\
&(i) \quad \exists\ block\_height\ outputs,\ T.stub = coinbase\ block\_height\ outputs\ \wedge \\
&(ii) \quad \forall\ TH'\ T'\ t', TH = TH' + [(T', t')] \implies t' \leq t\ \wedge \\
&(iii) \quad T.block\_height = coinbase\_height\ TH \\
&\vee \\
&(iv) \quad sum\_inputs(T) \geq sum\_outputs(T)\ \wedge \\
&(v) \quad \forall\ TH'\ T'\ t', TH = TH' + [(T', t')] \implies t' \leq t\ \wedge \\
&(vi) \quad T.inputs \neq [] \wedge \forall\ i\ j\ T_i'\ o_i\ r_i\ T_j'\ o_j\ r_j, i \neq j\ \wedge \\
&\quad T.inputs(i) = (T_i', o_i, r_i)\ \wedge\ T.inputs(j) = (T_j', o_j, r_j) \implies (T_i', o_i) \neq (T_j', o_j)\ \wedge \\
&(vii) \quad \forall\ j\ T'\ o\ r\ t'\ s\ v, (T', t') \in TH\ \wedge T.inputs(j) = (T', i, r)\ \wedge\ T'.outputs(i) = (s, v) \\
&\quad \implies (T', i) \in UTXO(TH)\ \wedge\ redeems(T', i, t', v, T, j, t)
\end{aligned}
$$

▶ **Definition 4** (Transaction history validity).

$$
\begin{aligned}
&tx\_history\_valid(TH) ::= \\
&\quad bc\_empty : TH = [] \to tx\_history\_valid(TH) \\
&\quad\quad bc\_cons : \forall\ TH'\ T\ t,\ TH = TH' + [(T, t)] \to tx\_history\_valid(TH') \\
&\quad\quad\quad\quad \to valid\_update(TH', T, t') \to tx\_history\_valid(TH)
\end{aligned}
$$

Now we define a *blockchain* (Figure 5, 37) as an inductive type. Hash pointers to blocks are, as before, represented by the blocks themselves. As we do not deal with proof-of-work or consensus, the only contents of a block are the pointer to the previous block (40), the transactions (41) and witnesses (42) of the block, and the block's timestamp (43). Transactions and witnesses are both represented as lists instead of Merkle trees, but are separated according to SegWit. We also define *block_height* (44) to be the number of blocks in the blockchain, and *bc_to_tx_history* (45) to be a function that flattens a blockchain into the transaction history it represents by concatenating lists of transactions paired with their respective witnesses. We define the *block reward* (47), a function from block height of the block to be minted to the base value to include in the block's coinbase transaction; and $transaction\_fees$ (46) to be the sum of the differences between input and output value for each transaction in a list.

The definitions of valid updates of blockchains by blocks and valid blockchains are analogous to the definitions for transaction histories.

▶ **Definition 5** (Valid update for blockchains). *A blockchain B is validly updated with a new block containing $(transactions, witnesses, timestamp)$ when*

1. *transactions list contains exactly one coinbase transaction CB as the first transaction*
2. $CB.block\_height = block\_height\ B$
3. $sum\_outputs\ CB = block\_reward\ (block\_height\ B) + transaction\_fees\ transactions$
4. $tx\_history\_valid\ (bc\_to\_tx\_history\ (Block\ B\ transactions\ witnesses\ timestamp))$

$$
\begin{aligned}
\texttt{Blockchain} &::= && (37)\\
&\quad \mathit{Empty} && (38)\\
&\quad \mathit{Block}\ \{ && (39)\\
\mathit{prevBlock} &: \texttt{Blockchain}; && (40)\\
\mathit{transactions} &: \texttt{TxStub}^*; && (41)\\
\mathit{witnesses} &: ((\texttt{StackValue}^*)^*)^*; && (42)\\
\mathit{timestamp} &: \texttt{Time}\ \} && (43)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{block\_height} &: \texttt{Blockchain} \to \mathbb{N} && (44)\\
\mathit{bc\_to\_tx\_history} &: \texttt{Blockchain} \to \texttt{TxHistory} && (45)\\
\mathit{transaction\_fees} &: \texttt{TxStub}^* \to \texttt{Satoshi} && (46)\\
\mathit{block\_reward} &: \mathbb{N} \to \texttt{Satoshi} && (47)
\end{aligned}
$$

Figure 5 Blockchain model.

▶ **Definition 6** (Blockchain validity). *We define the validity of a blockchain inductively.*

- *An Empty blockchain is valid.*
- *A blockchain B with a block appended is valid whenever B was valid, the length of the block's transactions and witnesses lists is equal, and the appended block validly updates B.*

4 Formally Verified Blockchain Properties

With all the definitions in place, we move on to state several important properties of valid transaction histories and blockchains, which we have proven in our Coq development. Here we list only a part of the development due to space constraints; it can be seen in full along with the proofs in the accompanying materials.

First, we prove that blockchain validity implies the validity of the transaction history it stores. This follows directly from the definition of valid updates with blocks applied to the last block in the chain, if any. This result allows us to reason about valid transaction histories instead of blockchains, which can be more convenient e.g., when proving the impossibility of double spending in a valid blockchain (and transaction history).

▶ **Lemma 7** (Blockchain validity implies transaction history validity). *Let B be a valid blockchain. Then $bc\_to\_tx\_history\ B$ is a valid transaction history.*

Note that the definition of a transaction history does not order the transactions according to output spending. In a valid transaction history, however, every transaction input refers to an output of a transaction earlier in the history, which we proved as a lemma.

The first key property of the blockchain we consider is the impossibility of spending the same output multiple times.

▶ **Theorem 8** (No double spending). *Let B be a valid blockchain, and TH be its (valid) transaction history. Let T_i and T_j be two transactions in TH at indices i, j respectively. Then*

$$
\begin{aligned}
\forall k_i\ k_j,\ (T_i.inputs(k_i) = (T_i', l_i, t_{rl,i}) \land T_j.inputs(k_j) = (T_j', l_j, t_{rl,j}) \land (i, k_i) \neq (j, k_j))\\
\implies (T_i', l_i) \neq (T_j', l_j).
\end{aligned}
$$

Another key property of valid transaction histories is that transactions identifiers are unique. While in reality we have to allow for the noninjectivity of hashes, in our model transactions are wholly unique within valid histories.

▶ **Theorem 9** (Transaction uniqueness)**.** *Let B be a valid blockchain, and TH be its (valid) transaction history. Let txs be the list of* `TxStubs` *in the history (i.e., TH with timestamps and witnesses removed). Let T_i and T_j be transactions at indices i, j in txs, respectively. If $T_i = T_j$, then $i = j$.*

In the remainder of this section we consider properties of the total supply of money in the system. This should be equal to the sum of all coinbase output values, however it is also allowed to be smaller than that due to the presence of transaction fees.

▶ **Theorem 10** (Coinbase value bounds UTXO value above)**.** *Let TH be a valid transaction history. Then*

$$UTXO\_value\ TH \leq coinbase\_value\ TH.$$

The following theorem illustrates the fact that only UTXO-s may be used as transaction inputs quantitatively. The proof follows from the definition of valid updates.

▶ **Theorem 11** (UTXO value bounds input value sum)**.** *Let $TH + [(T, t)]$ be a valid transaction history. Then*

$$sum\_inputs\ (stub\ T) \leq UTXO\_value\ TH.$$

The final theorem is a strengthening of (10) shows that supply is exactly controlled by block rewards. It boils down to proving that transaction fees are properly collected in the coinbase transaction outputs of each block.

▶ **Theorem 12** (Total block reward equals UTXO value)**.** *Let B be a valid blockchain, and $TH = bc\_to\_tx\_history\ B$. Then:*

$$UTXO\_value\ TH = \sum_{b=0}^{block\_height\ B-1} block\_reward\ b.$$

5 Limitations

Here we briefly discuss the limitations of our model and compare it to the Bitcoin client.

Since we use the symbolic model for digital signatures and hash functions, we are unable to prove the desired properties in the computational model of cryptography. Of course, we are also unable to extract the code for a verified client. We can overcome the latter by reusing Coq models of the cryptographic primitives (e.g. [7] for the SHA256 hash function) As for the former, since we are not concerned with the proof-of-work verification, we only rely on hash functions for data integrity and only need their collision resistance property. In the computational model, we could verify the properties under the assumption no collision occurred anywhere in that blockchain. For modeling properties of digital signatures in Coq we could attempt to use the toolset of the Foundational Cryptography Framework [21]. Alternatively, we could try to model the system within the universally composable (UC) security framework and replace the digital signature implementation with an ideal functionality similarly to the approach taken in [12] to develop mechanized analysis of a key exchange protocol.

The blockchain verification procedures are currently modeled mostly as first order inductive predicates rather than decidable routines and, hence, cannot be used to extract verified code. We plan to address this by writing the missing decision routines that generate proofs or disproofs of our propositions as well as routines that parse our model from serialized data, which would give us verified extractable blockchain validation code.

Comparison with the official Bitcoin client

First, we do not attempt to model several important aspects of the validation logic, since we do not consider them to be relevant to the correctness properties we wished to tackle first. Most notably, we omit proof-of-work verification and the corresponding data fields from the model. Transactions and blocks in our model do not have version numbers accounting for protocol updates. We do not enforce block and transaction size limits, coinbase maturity and we do not reject transactions with absurdly high fees.

We make a number of technical choices that result in a simpler formal model and diverge from the Bitcoin client. For example, we have explicit coinbase transactions while in the Bitcoin client coinbase transactions are stored in the same data structure and are distinguished by a single input field with the zero hash pointer. We feel that addressing these differences is a technical matter, albeit tedious and time consuming.

In our model, only transactions with segregated witnesses are supported, while the Bitcoin client additionally supports legacy transactions where the witness is a part of the transaction's input field. There are several other examples of extensions where both current and legacy features are supported. Moreover, these are almost always implemented in a backward-compatible manner. From a consensus perspective it is desirable that the blockchain verification procedures are updated by a *soft-fork* – old nodes *must* recognize the new blocks as valid. Hence, new features often need to be *hacked* into the existing protocol in order to satisfy the old validation procedures (e.g., see the "segregated witness" implementation [4]).

We feel that the multitude of supported options along with backwards-compatible implementations present the most significant challenge for building a complete mechanized formal model that is faithful to the wire-level protocol. Hence, more research is needed to produce methods of building and using such models without the exploding complexity.

6 Related Work

Bitcoin and similar systems have received a lot of attention in the scientific community in recent years with many attempts to formally specify and verify various aspects of blockchain systems.

Formal treatment of the Bitcoin system

First, we give an overview of formal models aimed at specification and verification of various aspects of the Bitcoin system.

Atzei et al.[9] give a formal model of Bitcoin transactions that we use a starting point for our formalization and mechanization efforts. The model includes transaction and blockchain data structures, as well as the semantics for the Bitcoin Script language. The model is used to formally prove "well-formedness" properties of the Bitcoin blockchain including the impossibility of double-spending. In contrast to a simplified "linked list" model of [9], we fully model blocks and the blockchain including coinbase transactions in each block, block height information, block rewards and transaction fees. For transactions themselves, our models are different in several details where we try to be closer to the behavior of the Bitcoin client. Most notably, in [9] segregated witnesses are a part of the transaction structure, while we store them in blocks, independently of transactions. Mechanization using the Coq proof assistant forces us to carefully specify all the details of the model. For example, mutually inductive definitions have to be explicitly taken into account. Similarly, we need to explicitly state and prove many assumptions that are implicit in [9], such as the temporal properties of spent outputs and explicit encoding of witnesses and hash functions. We replicate the no-double-spent result given in [9] but in a more general setting (with a blockchain data structure) and with a proof that is machine-verified. More importantly, we prove two additional properties of a valid blockchain.

In [13], the authors present the Extended UTXO model (EUTXO), which aims to extend Bitcoin's UTXO model in order to allow more expressive validation scripts. The work determines the expressive power of the model by showing its equivalence with Constraint Emitting Machines, a variant of state machines which is unimplementable in Bitcoin's script. As part of the work, the authors mechanize a transaction model very similar to the one in [9] using the Agda proof assistant. While close to our work, the authors do not attempt to follow Bitcoin specifically and work with a more general UTXO model (i.e. they do not model SegWit), and the overlapping part of the work does not extend beyond [9]. Thus, the previously stated differences between [9] (other than the mechanization) and our work apply here as well.

In [11] an alternative model is given for the semantics of blockchain transactions by using directed acyclic graphs to abstract the interactions of an incoming transaction with the blockchain. They provide a general blockchain model which they instantiate to to Bitcoin, Ethereum and Hyperledger Fabric systems.

Formal models of the Bitcoin Script language have also been an area of active research. In [8], the model of [9] is applied to development of a high-level domain specific language which then compiles into Bitcoin Script language, with the goal of systematically analyzing actual smart contracts proposed by researchers and Bitcoin developers. In [17] authors formalize the Bitcoin Script language with the goal of automatically finding inputs that satisfy a given script.

Finally, formal pen-and-paper treatments of Bitcoin's consensus mechanism include [16] where the focus is on quantifying the *quality* of the blockchain system by determining how many adversarial blocks are expected on the blockchain; and [14] where the authors work out the probability of a successful double-spending attack (assuming some nodes are malicious) and use the UPPAAL model checker to verify the results.

Consensus mechanization

In [22], the authors focus on mechanizing protocols and data structures necessary for establishing distributed consensus in blockchain systems. They formally prove a form of eventual consistency in a network,while precisely characterizing all assumptions on implementations of underlying security primitives. In [23], authors build and mechanize a probabilistic model of blockchain consensus with the eventual goal of stating and proving probabilistic security properties in a Byzantine setting. Other efforts towards automated verification of blockchain consensus mechanisms include [19, 20] that focus on the proposed proof-of-stake mechanism for the Ethereum system. All above efforts use the Coq proof assistant.

7 Conclusions and Future Work

In this paper, we have presented a Coq formalization for the Bitcoin's blockchain validation procedures including the models of basic data structures of the Bitcoin blockchain system and the denotational semantics for the typed variant of the Bitcoin Script language. We have used the model to provide machine-verified proofs for three essential properties of a valid blockchain: impossibility of double-spending, uniqueness of transactions and that cryptocurrency value is created only through block rewards.

In the future, we are going to discharge a number of simplifying assumptions and attempt to further bridge the gap between the abstract model and the reference client. In particular, we plan to model Merkle trees and use them to store transactions and witnesses in blocks. We also plan to make segregated witnesses optional and investigate the interaction between different types of transactions. More generally, we wish to investigate the scenarios where validity checks are updated. This will enable us to formally model the notion of soft-forks and evaluate proposed changes to the Bitcoin protocol such as spending rules based on Taproot, Schnorr signatures, and Merkle branches [6].

References

1 Bitcoin documentation. `https://en.bitcoin.it/wiki/Protocol_documentation`, 2010.

2 Bitcoin improvement proposal 30: Duplicate transactions. bip-0030.mediawiki, 2012.

3 Bitcoin improvement proposal 34: Block v2, height in coinbase. bip-0034.mediawiki, 2012.

4 Bitcoin improvement proposal 141: Segregated witness (consensus layer). bip-0141.mediawiki, 2015.

5 Double spending in bitcoin clients. CVE-2018-17144, 2018.

6 Bitcoin improvement proposal 341: Segwit version 1 spending rules. bip-0341.mediawiki, 2020.

7 Andrew W. Appel. Verification of a cryptographic primitive: Sha-256. *ACM Trans. Program. Lang. Syst.*, 37(2), April 2015. `doi:10.1145/2701415`.

8 Nicola Atzei, Massimo Bartoletti, Tiziana Cimoli, Stefano Lande, and Roberto Zunino. Sok: Unraveling bitcoin smart contracts. In *Principles of Security and Trust*, pages 217–242, Cham, 2018. Springer International Publishing.

9 Nicola Atzei, Massimo Bartoletti, Stefano Lande, and Roberto Zunino. A formal model of bitcoin transactions. *IACR Cryptology ePrint Archive*, 2017:1124, 2017.

10 Vitalik Buterin. A next-generation smart contract and decentralized application platform. `https://github.com/ethereum/wiki/wiki/White-Paper`, 2014. White-paper.

11 Christian Cachin, Angelo De Caro, Pedro Moreno-Sanchez, Björn Tackmann, and Marko Vukolic. The transaction graph for modeling blockchain semantics. *IACR Cryptology ePrint Archive*, 2017:1070, 2017.

12 R. Canetti, A. Stoughton, and M. Varia. Easyuc: Using easycrypt to mechanize proofs of universally composable security. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, pages 167–16716, June 2019.

13 Manuel Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Michael Peyton Jones, and Philip Wadler. The extended utxo model. In *Workshop on Trusted Smart Contracts*, 2020.

14 Kaylash Chaudhary, Ansgar Fehnker, Jan Cornelis van de Pol, and Mariëlle Ida Antoinette Stoelinga. Modeling and verification of the bitcoin protocol. In *Proceedings of the Workshop on Models for Formal Analysis of Real Systems (MARS 2015)*, Electronic Proceedings in Theoretical Computer Science, pages 46–60, Australia, November 2015. Open Publishing Association.

15 D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, March 1983.

16 Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Advances in Cryptology - EUROCRYPT 2015*, pages 281–310, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

17 Rick Klomp and Andrea Bracciali. On symbolic verification of bitcoin's SCRIPT language. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 38–56, Cham, 2018. Springer International Publishing.

18 Satoshi Nakamoto. Bitcoin: a peer-to-peer electronic cash system. `https://bitcoin.org/bitcoin.pdf`, 2008.

19 Karl Palmskog, Milos Gligoric, Lucas Pena, Brandon Moore, and Grigore Rosu. Verification of casper in the coq proof assistant. `https://github.com/runtimeverification/casper-proofs`, 2018. Technical report.

20 Karl Palmskog, Milos Gligoric, Lucas Pena, Brandon Moore, and Grigore Rosu. Verifying finality for blockchain systems. In *CoqPL'19*, 2019.

21 Adam Petcher and Greg Morrisett. The foundational cryptography framework. In *Principles of Security and Trust*, pages 53–72, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

22 George Pirlea and Ilya Sergey. Mechanising blockchain concensus. In *7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM New York, 2018.

23 Ilya Sergey and Kiran Gopinathan. Towards mechanising probabilistic properties of a blockchain. In *CoqPL'19*, 2019.

24 The Coq Development Team. The coq proof assistant, version 8.10.0, October 2019. `doi:10.5281/zenodo.3476303`.