The Interdisciplinary Center, Herzlia
Efi Arazi School of Computer Science
M.Sc. program - Research Track

Bitcoin+: Cheap Support For Complex Spending Conditions in a UTXO Ledger

by
Yaron Kaner

M.Sc. dissertation, submitted in partial fulfillment of the requirements for the M.Sc. degree, research track, School of Computer Science
The Interdisciplinary Center, Herzliya

September 2020
This work was carried out under the supervision of Dr. Tal Moran from the Efi Arazi School of Computer Science, The Interdisciplinary Center, Herzliya.
Acknowledgements

I would like to express my deepest appreciation and gratitude to my advisor, Dr. Tal Moran from the Interdisciplinary Center (IDC), for the great part he had in this work, for his uncompromising mission towards quality and accuracy, and for the countless hours he devoted to mentoring me. His patience and never-ending willingness to teach are what made this journey both possible and enjoyable.
Abstract

Cryptocurrencies can be used merely to transfer value between identities, but many of the more interesting uses of cryptocurrencies require contracts, e.g., “a transfer of X coins from party S to party R should occur only if conditions A and B hold”. Bitcoin (and related cryptocurrencies) place strict limits on the language in which these conditions can be phrased. In particular, they have limited length and don’t allow loops.

In this work, we show how to augment the Bitcoin scripting language with a single “innocuous” operation to that allows us to create “meta conditions” with much more expressive power (e.g., as defined by arbitrarily-sized circuits).

We construct a protocol to compile such meta-conditions into a set of (augmented) Bitcoin transactions. We then show how to use this compiler to realize a full “meta-ledger” functionality, which we show is secure in the universal composability framework.
1 Introduction

Introduced by Satoshi Nakamoto in 2008 [8], Bitcoin is the first decentralized cryptocurrency, and still the most popular in terms of market cap and transaction volume (as measured in USD).

Bitcoin’s design is based on a transaction ledger. Loosely, a transaction ledger accepts transactions as input from honest parties and outputs an ordered list of transactions (which all honest parties agree on).

Bitcoin Transactions and the UTXO Ledger

The transaction is the primary abstraction in Bitcoin; there are no separate notions of “accounts” or “balances”. Instead, each Bitcoin transaction has inputs and outputs. Every transaction output (TXO) specifies an amount (in Bitcoins) and a spending condition, and every input references a single TXO and has some additional “proof data”.

A transaction output can be thought of as a “coin” whose value is the amount specified by the output. “Spending” the coin is done by publishing a new transaction whose input references the corresponding TXO. The spending condition specifies who is allowed to spend the coin.

In order to verify the validity of a transaction, nodes need to store only the set of unspent transaction outputs—the UTXO database, for which this type of ledger is named.

Scripts and Smart Contracts

The most common type of spending condition is verification of a signature with respect to a specific public key: an input can spend such a TXO if its proof data contains a valid signature (under the corresponding public key) of the spending transaction’s ID (a collision-resistant hash of the spending transaction).

While signature verification is sufficient for simple transfer of funds, one of the innovations introduced by Bitcoin is a scripting language for specifying output conditions. This allows a novel use of cryptocurrencies: the creation of smart contracts—cryptographic “boxes” that contain value and “automatically” unlock it if certain conditions are met.

Bitcoin scripting conditions can define more complex “ownership structures” (e.g., funds can be spent if any k out of n public keys sign the spending transaction), can postpone spending until some point in the future (“time-locked transactions”) or even require some cryptographic puzzle to be solved before an output can be spent.

Limited Expressiveness

The Bitcoin scripting language is kept purposefully simple. A major motivation for this decision is that the cost of evaluating the spending conditions is borne by every Bitcoin node—every node must verify all transactions in the ledger to ensure that they are in consensus on the state of the UTXO database. This cost is offset by a transaction fee. However, because the fee is deducted from the spent TXOs, there is no way to charge fees for invalid transactions (which are not “allowed” to spend the TXO at all). If an expensive computation is required to determine transaction validity, this would open the door to cheap denial-of-service (DoS) attacks.

Thus, Bitcoin script is designed to ensure that the execution cost is always known and bounded. In particular, Bitcoin scripts are not Turing complete; in fact, the language does
not allow loops of any kind, and scripts have bounded length. Moreover, the set of basic arithmetic operations is also quite limited.

While the existing functionality already allows for interesting use-cases, there are many for which Bitcoin’s scripting language is insufficiently expressive. (One interesting example, and a motivation for this work, is the notion of secure computation “with penalties”, in which parties run a secure multiparty computation protocol without an honest majority, but with the guarantee that if the adversary aborts the computation the honest parties will be compensated financially.)

1.1 Our Contributions

This work aims to increase the expressiveness of the Bitcoin scripting language, without sacrificing the simplicity and efficiency of the UTXO ledger.

Cheap Support for Arbitrary Circuit Conditions

Our main concrete contribution is a protocol that allows a TXO to be bound to an arbitrary boolean circuit—i.e., the output can be spent only by publishing a satisfying assignment for the circuit.

The protocol relies on adding a single new opcode to the Bitcoin scripting language, one that allows a spending condition to test whether a given TXO is in the UTXO database. We refer to this opcode as \texttt{is-txo-unspent}, and to the Bitcoin scripting language augmented by the opcode as $\texttt{Bitcoin+}$.

Because executing this opcode only requires access to the existing UTXO database, in exactly the same way as testing for a double-spend, adding it to existing Bitcoin code is very simple, and it has minimal execution overhead (we verified this by modifying the Bitcoin core code to include the new opcode; this required changes to less than 40 lines of code, including additional comment lines).

Universally Composable Overlay Ledgers

Existing protocols that require complex spending conditions are often written assuming an ideal ledger functionality. To allow modular use of our compiler in existing such protocols we formally define a generalized ideal UTXO Ledger functionality, $\mathcal{F}_{\text{ledger}}$, which is parameterized by the language in which spending conditions may be specified.

The ideal functionality preserves “in spirit” the guarantees of the simple UTXO ledger (i.e., the guarantees relied on by most high-level ledger protocols), but at the same time allows a single transaction in the meta-ledger to be represented by multiple transactions in the base-ledger. Defining such a functionality turns out to be quite tricky, since some of the properties of a simple UTXO cannot be guaranteed in this setting. (In chapter 3, we provide an overview of the challenges we faced in defining the generalized ledger functionality and the intuitions behind the relaxations we made to the simple UTXO ledger.)

Our model is in the same spirit as that of Badertscher, Maurer, Tschudi and Zikas [1]. However, we focus specifically on a UTXO ledger, and the required relaxations to support meta (overlay) ledgers as instances of the same functionality.

To simplify the protocol descriptions and proofs, we model the ledger as a sequence of transactions, completely ignoring their grouping into blocks—this grouping is irrelevant for most higher-level protocols but adds complexity to the model.

We construct a protocol in the $\mathcal{F}_{\text{ledger}}$-hybrid model (i.e., with access to an ideal UTXO ledger whose spending conditions may be specified in the language $\mathcal{L}$) that realizes the $\mathcal{F}_{\text{ledger}}\cup\mathcal{L}_{\text{Circuit}}$ ledger functionality, where $\mathcal{L}_{\text{Circuit}}$ is the language of boolean circuits. Our protocol works for
every “base” language $\mathcal{L}$ that is sufficiently expressive (the Bitcoin scripting language, augmented with is-txo-unspent, is an example).

Our proof of security is in the Externalized UC model [2], where we model the clock as an external shared functionality. This allows any protocol that is secure when using the ideal ledger functionality $F_{\text{ledger}}^{\mathcal{L}/\mathcal{L}_{\text{Circuit}}}$ to be transparently “compiled” into a protocol that is secure using only the base-ledger, and composed with other protocols sharing the same global clock.

In addition to being useful in itself, and as a template for constructing other “meta-ledgers”, our protocol demonstrates the usefulness of our generalized ledger definition.

1.2 Related Work

Account-based Ledgers

The limitations of the Bitcoin scripting language have been clear from the start. One solution, exemplified by the Ethereum protocol, is to replace the UTXO model with an account-based system. At a very high level, the first-class object in such a system is an account. Each account has an associated value, corresponding to the funds held by the account, and a contract. The contract is essentially a program that accepts inputs and funds, and can send funds and inputs to other accounts as part of its execution. A transaction in this model is a set of inputs to a specific account’s contract [4].

Ethereum’s account-based model is extremely expressive—it allows contracts to be written in a Turing-complete language. However, the cost is paid in higher complexity. Since all miners in the system must execute every transaction to maintain consensus, they must all run more complex code (increasing the probability of bugs and security vulnerabilities) and expend more CPU/memory resources (leading to higher costs and more complex incentive schemes).

Both UTXO and account-based schemes are deployed today, and it does not appear that either one will supplant the other. Thus, it makes sense to study how to increase the expressiveness of UTXO-based ledgers without making the leap to a fully account-based scheme, with its corresponding tradeoffs.

Bitcoin Covenants

Möser, Eyal and Gün Sirer proposed extending Bitcoin script to include “covenants” [7]. This extension allows an output condition to specify constraints on the spending transaction—for example, to enforce that the spending transaction’s outputs have a specific format. They show some specific use-cases allowed by covenants (such as mitigating the results of private-key theft using “vaults”). Several following works explored ways to implement covenants in different ways, and the use of covenants for additional tasks [9] [11] [12].

In contrast, we focus on allowing general computation rather than ad-hoc use-cases. Moreover, we formally prove the security of our protocols (the existing works on covenants do not include formal security proofs at all), and introduce a new, generic framework for formally analyzing overlay ledgers. (Analyzing a covenant-based overlay in this framework is an interesting open question.)

Extending UTXO

Chakravarty, Chapman, MacKenzie, Melkonian, Peyton and Wadler formalize an “Extended UTXO Model” [3], whose purpose appears very similar to that of this work: to support more expressive computation while retaining the advantages of the UTXO model. However, while the high-level goals are the same, the strategies are very different.

1This is a gross oversimplification of how Ethereum’s account model is constructed, of course, but suffices for the purposes of comparing to the UTXO model.
Our protocol focuses on building a more expressive ledger using a simpler ledger as a substrate—and in particular making a minimal change to the current Bitcoin scripting language (both in code and in runtime complexity) that would allow complex conditions to be supported. In contrast, Chakravarty et al. define a more powerful model (e.g., transaction outputs can have extra state, and output conditions can take into account the contents of the transaction that spends them), but at the cost of losing compatibility with existing UTXO-based ledgers—effectively, implementing their modifications would require a completely new ledger.

Ledger Formalizations

There are several works trying to formalize UTXO ledgers and to reconcile UTXO and account-based ledgers.

Garay, Kiayias and Leonardos gave the first formal security definitions for a blockchain ledger [4], in the context of analyzing the security of the Bitcoin protocol. Pass, Seeman and Shelat extended the analysis to a more realistic setting (with communication delays) and proposed slightly different security definitions [10]. Both of these works defined security as a set of properties.

Kiayias, Zhou and Zikas defined the ledger functionality for the first time as an ideal functionality in the UC framework [6], with the goal of making it usable in other higher-level protocols. However, their functionality was not realizable using existing blockchain protocols (e.g., it did not allow for a “rushing” adversary that can prevent an honest transaction from entering the state by inserting a conflicting transaction before it). Badertscher, Maurer, Tschudi and Zikas defined a slightly weaker ideal functionality [1] that addresses these problems and is still strong enough to be useful in high-level protocols.

Our ideal functionality has the same flavor as [1], but tailored to UTXO ledger overlays. The novelty in our definition is encapsulated in the equivalent of the “validation rules” from [1]: we carefully construct these rules to achieve a ledger that is both powerful enough to be useful, but “weak” enough to allow it to be realized as an overlay. (In particular, our ledger functionalities can be “stacked”, to construct meta-meta-ledgers, etc.)

Combining UTXO and account-based ledgers

Zahnentferner defines “Chimeric Ledgers” [14]. He formalizes the two types of ledgers and translations between them—e.g., for a given transaction in a UTXO-based ledger, what is the set of transactions in an account-based ledger that have an equivalent effect. However, this paper deals only with “accounting”, explicitly ignoring “cryptographic authorization” (i.e., scripts). A follow-up paper by the same author provides definitions for UTXO ledgers with scripts [13].

Both of these papers deal with formalization rather than protocols (there are algorithms for “transforming” from one type of ledger to another, but these are external to the ledgers themselves). Their definitions are in the formal-logic style, which is useful for interfacing with automated theorem-proving tools. In contrast, our formal definitions are ideal functionalities, which are easier to use in protocol constructions, and allow us to prove the security of our protocols in the UC model.
2 Compiling Arbitrary Circuits

We refer to output conditions that are not supported by the underlying ledger as a “meta-conditions”. We refer to conditions supported by the underlying ledger as “base-conditions”. In this section we construct a compiler from an arbitrary boolean circuit meta-condition to a sequence of Bitcoin+ transactions (the underlying ledger) with base output conditions. The compilation guarantees that if the output of a special keystone transaction is spent, it is always possible to extract a satisfying assignment to the circuit.

More concretely, let $C$ be an arbitrary circuit, and let $txo_C$ be a transaction with a meta-output whose condition is $C$ (a meta-condition). In section 2.1 we construct a compiler that takes as input a transaction such as $txo_C$ and outputs a tuple $(\text{fragments}_C, tx_C-\text{keystone})$, where $\text{fragments}_C$ is a set of fragment transactions and $tx_C-\text{keystone}$ is the special final transaction whose output will be spendable only by publishing a satisfying assignment for $C$. All of the $\text{fragments}_C$ transactions and $tx_C-\text{keystone}$ are valid ledger transactions.

Section 2.3 proves the main soundness guarantee achieved by our circuit compiler — if $\text{fragments}_C$ and $tx_C-\text{keystone}$ are compiled and published using the methods described in section 2.1, and the output of $tx_C-\text{keystone}$ is successfully spent in the base-ledger, it is always possible to extract a satisfying assignment for $C$.

We refer to input data that are not supported by the underlying ledger as a “meta-data”. We refer to data supported by the underlying ledger as “base-data”. In section 2.2, we show the completeness of our compiler. That is, given an assignment $A$ of boolean values that satisfies the circuit $C$, and a transaction $txo_A$ with a meta-input $txi_A$ that “meta-spends” $txo_C$, we can compile $txo_A$ into a sequence of base transactions $(\text{fragments}_A, tx_A-\text{keystone})$. If published in order, $tx_A-\text{keystone}$ will successfully spend the output of $tx_C-\text{keystone}$.

We refer to $txo_C$ and $txo_A$ as meta-transactions. For simplicity, we restrict the discussion to meta-transactions with either a single meta-input and a single base-output, or a single base-input and single meta-output (our techniques extend to more complex transactions in a straightforward way). We also assume, w.l.o.g., that the circuits are composed solely of NAND gates.

We note that the compiler described in this chapter does not, by itself, realize a composable ideal ledger functionality. In chapters 4 and 5 we show how it can be used to construct such a ledger (whose ideal functionality is explained in chapter 3), as a component in a generic ledger protocol.

2.1 Compiling a Circuit Spending Condition

Consider a transaction whose single meta output’s condition is a circuit. $txo_{\text{circuit}}^m$ shown at the top of fig. 2.1.1 is one such transaction. It has a single output $txo_{\text{circuit}}^m$, whose condition is $\neg(a \land b) = \text{true}$.

Compilation of a circuit $C$ yields four types of base transactions — “input-bit transactions” and “gate transactions” corresponding to the circuit’s input-bits and the circuit’s logical gates, a single “keystone transaction” whose output represents the circuit and “splitter transactions” that are used to fund the former three types.

To reduce verbiage, we will say $x$ is a component if $x$ is an input-bit, a gate or the keystone.
2 Compiling Arbitrary Circuits

2.1.1 Component Transactions

Input-bit Transactions

Each of \( C \)'s input-bits has a corresponding input-bit transaction. In fig. 2.1.1 input-bits \( a \) and \( b \)'s corresponding transactions are \( tx_a \) and \( tx_b \). An input-bit transaction, like any other transaction, should have at least one transaction input, and contain a transaction fee, made up of the difference between incoming and outgoing coins. Each input-bit transaction has two outputs, one for each boolean value (\( txo_{a+} \), \( txo_{a-} \), \( txo_{b+} \) and \( txo_{b-} \) in fig. 2.1.1).

Gate Transactions

For each internal boolean gate a gate transaction is created. In fig. 2.1.1 gate \(-(a \land b)\)'s corresponding transaction is \( tx_{-(a \land b)} \). Generally, like the input-bit transaction, each gate transaction has at least one input and two outputs, one for each boolean value.

Circuit-keystone Transaction

An exception to the “two-outputs-per-transaction” rule is the transaction created for the “circuit-keystone”, the output gate of the circuit. This transaction only has one output and it holds the reward from \( txo_{m_{circuit}} \). In fig. 2.1.1 \( tx_{-(a \land b)} \) is also the circuit-keystone transaction.

2.1.2 Wiring the Gates

Labels

As described above, \( tx_x \), the transaction corresponding to component \( x \), has a positive output \( txo_{x+} \) and a negative output \( txo_{x-} \). We label each transaction with a value in \{true, false, unset\}. The label of \( tx_x \) is denoted \( \text{label}(tx_x) \), or simply as \( \text{label}(x) \). \( \text{label}(tx_x) = \text{true} \) if only \( txo_{x+} \) is spent, \( \text{label}(tx_x) = \text{false} \) if only \( txo_{x-} \) is spent, and otherwise \( \text{label}(tx_x) = \text{unset} \). Note that the label of a component is a function of the transaction ledger state (it can change as new transactions are appended to the ledger).

Gate Functions

Each gate is dependent upon its (two) inputs, and its value is a Boolean (binary) function of their labels. Let \( tx_g \) be gate \( g \)'s transaction and \( f_g(\cdot, \cdot) \) its Boolean function. Also, let \( a, b \) be \( g \)'s two inputs, and \( tx_a \) and \( tx_b \) their corresponding transactions. The conditions at the outputs of \( tx_g \) should enforce the following:

- \( txo_{g+} \) can be spent iff \( f_x(\text{label}(a), \text{label}(b)) = \text{true} \),
- \( txo_{g-} \) can be spent iff \( f_x(\text{label}(a), \text{label}(b)) = \text{false} \).

For example, for the \(-(a \land b)\) gate, the positive output’s condition will be \( -(\text{label}(a) = \text{true} \land \text{label}(b) = \text{true}) \).

2.1.3 Computing Labels in Bitcoin+

In order to implement the gate conditions as described in section 2.1.2, the function \text{label} must be expressible in the base-ledger’s spending condition language. Bitcoin’s current scripting language is not sufficient for this purpose. We propose to address this by augmenting the Bitcoin scripting language with a single new opcode, \text{OP\_IS\_TXO\_UNSPENT} (creating the extended language Bitcoin+). \text{OP\_IS\_TXO\_UNSPENT} accepts the ID of an output and returns \text{true} iff the TXO is present in the UTXO database, i.e unspent. More concretely, it pops the ID of the transaction and then the index of the output from the stack and pushes
the result back. We will use a shorter syntax in this work, denoting \texttt{is-txo-unspent}(txo) the result of the computation above with respect to TXO \texttt{txo}.

Querying whether a transaction output is \textit{spent} may seem more natural. We chose \texttt{OP-\textit{IS-TXO-UNSPENT}} because it is straightforward to implement given the UTXO database that is already maintained by Bitcoin nodes. Note that in itself, \texttt{OP-\textit{IS-TXO-UNSPENT}} doesn’t answer whether an output is \textit{spent}. The absence of an output from UTXO does not necessarily imply that it was spent; it could also mean that the transaction was never published.

To bridge this gap we use the fact that transaction outputs enter the Bitcoin ledger atomically (i.e., either all transaction outputs of a single transaction are inserted into UTXO or none are). Consider a component transaction \(tx_x\) that has two outputs \(txo_{x+}\) and \(txo_{x-}\) (the same argument applies for input-bits as well). We compute \texttt{LABEL}(x) in the following manner:

1. If both \(txo_{x+}\) and \(txo_{x-}\) are in UTXO, \(tx_x\) wasn’t labeled yet, and so \texttt{LABEL}(x) is \texttt{unset}.

2. If neither \(txo_{x+}\) nor \(txo_{x-}\) is in UTXO, we can’t know if \(tx_x\) was accepted by the ledger, and so \texttt{LABEL}(x) is \texttt{unset}.

3. But, if \(txo_{x+}\) is in UTXO, and \(txo_{x-}\) isn’t, then \(txo_{x-}\) must have been spent, and therefore \texttt{LABEL}(x) = \texttt{false}.

4. Similarly, if \(txo_{x-}\) is in UTXO, and \(txo_{x+}\) isn’t, then \(txo_{x+}\) must have been spent, and therefore \texttt{LABEL}(x) = \texttt{true}.

Thus, the condition of \(txo_{-(a\land b)+}\) in fig. 2.1.1 that is supposed to implement \(\neg(\texttt{LABEL}(a) \land \texttt{LABEL}(b))\) can be implemented in Bitcoin+ by applying the following replacements:

\[
\text{LABEL}(a) = \texttt{true} \iff \neg \texttt{is-txo-unspent}(txo_{a+}) \land \texttt{is-txo-unspent}(txo_{a-})
\]

\[
\text{LABEL}(b) = \texttt{true} \iff \neg \texttt{is-txo-unspent}(txo_{b+}) \land \texttt{is-txo-unspent}(txo_{b-})
\]
2 Compiling Arbitrary Circuits

2.1.4 Splitter Transactions

We also create a transaction to claim the output referenced by the \( tx_{\text{circuit}} \) and distribute the coins between the component transactions (\( tx_{\text{splitter}} \) in fig. 2.1.1). We refer to it as the splitter transaction. An output should be created for each funded transaction (e.g., \( tx_a \), \( tx_b \), \( tx_{\neg(a\land b)} \) in fig. 2.1.1), and funding should cover transaction fees and the reward.

To simplify the description below, we assume a single splitter transaction suffices; however, the compiler can easily be extended to the case where the number of components is larger than the maximum number of outputs the ledger allows per transaction. If this occurs, the compiler uses a tree of transactions to get to the required output count. In this case, the root of the tree is the splitter transaction. It has one output reserved for the circuit keystone, and the rest for the transactions that comprise the internal nodes of the tree. All internal nodes have one input (spending one of their parent’s outputs) and the maximal number of outputs allowed by the underlying ledger. The leaf nodes’ outputs are spent by the other component transactions.

2.1.5 Loss of Atomicity and Publishing Considerations

Since we use several transactions to implement \( tx_{\text{circuit}} \) and their insertion into the ledger is not atomic, an attacker may try to “hijack” the insertion while it is happening. For example, the attacker might claim an output of a splitter transaction that is intended to fund one of the gate transactions, rendering the sender unable to complete the insertion. To prevent this attack, the spending condition for internal TXOs verifies a signature with respect to a freshly generated signature key.

Component transactions spend splitter transaction outputs, and so the splitter transaction must be published first in order for subsequent transactions to be included in the ledger.

2.1.6 Efficiency and Optimizations

Since the compilation creates a transaction for each component, as the circuit grows larger, the total fee required to publish the transactions grows. Our compiler emphasizes simplicity and clarity over efficiency—it is not optimized to use the smallest number of transactions. However, we briefly mention some strategies to lower the transaction count.

For example, it is possible to group the outputs of the input-bit transactions to fewer transactions. Grouping outputs of gate transactions is also possible, but more tricky—a condition of one such output may need to query about the label of an output that is part of the same transaction, and it is not trivial to expect the ledger to provide this ability (this is impossible in Bitcoin, for example). It is possible, however, to merge transactions for gates of the same circuit depth, as they do not reference one another.

The described construction creates one transaction per gate. Another possible optimization is to build transactions for sub-circuits, or allow gates of fan-in larger than two (making each transaction reference more than two inputs).

2.2 Compiler Completeness: Compiling a Circuit Assignment

Consider a transaction whose single meta input specifies as the targeted output an output of a circuit transaction and provides an assignment of boolean values to the input-bits of the circuit. \( tx_{\text{assignment}} \), appearing at the top of fig. 2.2.1, is one such transaction. Its meta input \( tx_{\text{assignment}} \) specifies \( tx_{\text{circuit}} \) (shown in fig. 2.1.1) as the targeted output and provides the assignment “\( a \leftarrow \text{true}, b \leftarrow \text{true} \)” to the input-bits of \( tx_{\text{circuit}} \)’s circuit.
2.2.1 Component-Labeling Transactions

Our compiler produces three types of base transactions — “input-bit-labeling transactions” and “gate-labeling transactions” corresponding to the circuit’s input-bits and the circuit’s logical gates, and a special assignment-keystone transaction, that will spend the circuit output.

Input-bit-labeling Transactions

Input-bit-labeling transactions implement the assignment $(tx_{a\leftarrow\text{true}}, tx_{b\leftarrow\text{false}})$ in fig. 2.2.1. One transaction is created per input-bit, and therefore per input-bit transaction. Input-bit transactions $(tx_{a}, tx_{b}$ in fig. 2.1.1), have two outputs each, one positive and one negative. An input-bit-labeling transaction spends one of the outputs, according to the assignment $(tx_{a\leftarrow\text{true}}$’s input spends $txo_{a+}$ and $tx_{b\leftarrow\text{false}}$’s input spends $txo_{b+}$).

Gate-labeling Transactions

Gate-labeling transactions “evaluate” the circuit given the assignment $(tx_{a\leftarrow\text{true}}, tx_{b\leftarrow\text{false}}$ in fig. 2.2.1. One is created per gate, and therefore per gate transaction. Gate transactions $(tx_{\neg(a\land b)}$ in fig. 2.1.1), compiled from a circuit output, have two outputs each, one positive and one negative (an exception is the circuit-keystone transaction, that only has one output). A non-keystone gate-labeling transaction spends one of the outputs of the corresponding gate transaction, according to the gate values that are computed from the assignment $(tx_{a\leftarrow\text{true}}, tx_{b\leftarrow\text{false}}$’s input spends $txo_{\neg(a\land b)+}$).

Assignment-keystone Transaction

One transaction is created that spends the circuit-keystone transaction’s output; this transaction is called the assignment-keystone transaction. Its output will be identical to $tx_{\text{assignment}}$’s output. Transaction fees of labeling transactions should come from the outputs they claim.

Figure 2.2.1: $tx_{\text{assignment}}$ Compilation
2.3 Compiler Soundness: Extracting a Satisfying Assignment

The soundness requirement from our circuit compiler is that the output of the circuit keystone cannot be spent unless a satisfying assignment is published to the ledger. Formally, we require the existence of an extraction algorithm that, given a ledger in which the $tx_C$-keystone is spent, can output a satisfying assignment for $C$. This extraction algorithm is described in algorithm [2.11] and theorem [2.3.1] guarantees that it works (for the formal statement of the theorem, we need to define some additional notation, which we do in section [2.3.1]).

Essentially, the extractor finds the input-bit transactions by traversing the DAG rooted at the circuit-keystone transaction, and by observing state of the ledger can recover the labels of the input-bits. The proof of theorem [2.3.1] is by induction on the depth of the circuit.

2.3.1 Soundness Proof Details

Notation

Let $x$ be an arbitrary component. We define the following terms:

- $\text{LABEL}^t(x)$ — the value of $\text{LABEL}(x)$ at time $t$.
- $t^+_x$ — the time $\text{LABEL}(x)$ first transitions from $\text{unset}$, or $\bot$ if it never does
- $\text{LABEL}^*(x)$ — $\text{LABEL}^t(x)$ if $t^+_x \neq \bot$, or $\text{unset}$ otherwise.
- $\text{INP}(x)$ — the set of input bits corresponding to $x$ (i.e., if $x$ is an input-bit then $\text{INP}(x) = x$, otherwise if $x$ is the gate $x = \neg(y \land z)$ then $\text{INP}(x) = \text{INP}(y) \cup \text{INP}(z)$.
- $\text{ASSIGN}^*(x)$ — $\{(z, \text{LABEL}^*(z)) | z \in \text{INP}(x)\}$ (the assignment to $x$’s inputs.)

We can now state our soundness theorem:

Theorem 2.3.1. Let $C$ be an arbitrary circuit, and let $tx^m_C$ be a single-base-input, single-meta-circuit-output transaction, where $C$ is the condition in the output. Assume $tx^m_C$ was compiled and the resulting transactions were published successfully.

For every component $x$, if $\text{LABEL}^*(x) \neq \text{unset}$ then $\text{LABEL}^*(x) = C_x(\text{ASSIGN}^*(x))$ where $C_x$ is the sub-circuit rooted at $x$.

In order to prove the theorem, we will require some helpful claims:

Claim 2.3.2. For every component $x$ and $\alpha \in \{\text{true}, \text{false}\}$: if $\text{LABEL}^*(x) = \alpha$, then for every $t \neq t^+_x$ it holds that $\text{LABEL}^t(x) \in \{\alpha, \text{unset}\}$.

Proof. For $t < t^+_x$, $\text{LABEL}^t(x) = \text{unset}$ by definition. For $t > t^+_x$, suppose, in contradiction, that $\text{LABEL}^t(x) = \neg \alpha$. By definition, this means that at time $t$, $txo_{x^\alpha}$ is unspent. However, $\text{LABEL}^*(tx_x) = \alpha$ means that at time $t^+_x$, $txo_{x^\alpha}$ is spent. This contradicts the ledger’s guarantee that a transaction output cannot re-enter the UTXO.

Corollary 2.3.3. For every gate $g$ with inputs $a$ and $b$ and $\alpha \in \{\text{true}, \text{false}\}$, if $\text{LABEL}^*(g) = \alpha$ then $\neg(\text{LABEL}^*(a) \land \text{LABEL}^*(b)) = \alpha$.

Proof. Since $\text{LABEL}^*(g) = \alpha$, $txo_{\alpha^g}$ was spent at $t^+_g$, and therefore at that time its condition $\neg(\text{LABEL}(a) \land \text{LABEL}(b)) = \alpha$ was satisfied. Since $\alpha \neq \text{unset}$, it must be that both $\text{LABEL}^*(a) \neq \text{unset}$ and $\text{LABEL}^*(b) \neq \text{unset}$. Thus, by claim [2.3.2] it holds that: $\neg(\text{LABEL}^*(a) \land \text{LABEL}^*(b)) = \neg(\text{LABEL}^*(a) \land \text{LABEL}^*(b)) = \alpha$.

The proof of theorem [2.3.1] follows:
Algorithm 2.1: EXTRACT-ASSIGNMENT

1: procedure EXTRACT-ASSIGNMENT(\textit{tx}_{\text{keystone}}) \\
2: \hspace{1em} return EXTRACT-ASSIGNMENT-INTERNAL(\textit{tx}_{\text{keystone}}) \\
3: end procedure \\

4: procedure EXTRACT-ASSIGNMENT-INTERNAL(\textit{tx}) \\
5: \hspace{1em} if \textit{tx} corresponds to input-bit \textit{i} then \\
6: \hspace{2em} \text{LABEL}^*(*i) \leftarrow \text{COMPUTE-LABEL}^*(\textit{tx}) \\
7: \hspace{2em} return \{(*i, \text{LABEL}^*(*i))\} \\
8: \hspace{1em} else // \textit{tx} corresponds to gate \textit{g} \\
9: \hspace{2em} \textit{tx}_a, \textit{tx}_b \leftarrow \text{the transactions referenced by}\ g's output conditions using IS-TXO-UNSPENT \\
10: \hspace{2em} \text{ASSIGN}^*(\textit{a}) \leftarrow \text{EXTRACT-ASSIGNMENT-INTERNAL(\textit{tx}_a)} \\
11: \hspace{2em} \text{ASSIGN}^*(\textit{b}) \leftarrow \text{EXTRACT-ASSIGNMENT-INTERNAL(\textit{tx}_b)} \\
12: \hspace{2em} return \text{ASSIGN}^*(\textit{a}) \cup \text{ASSIGN}^*(\textit{b}) \\
13: end if \\
14: end procedure \\

15: procedure COMPUTE-LABEL^*(\textit{tx}_i) \\
16: \hspace{1em} \textit{txo}_i+, \textit{txo}_i- \leftarrow \textit{tx} outputs \\
17: \hspace{1em} if neither \textit{txo}_i+ nor \textit{txo}_i- is spent then \\
18: \hspace{2em} return unset \\
19: \hspace{1em} else if both \textit{txo}_i+ and \textit{txo}_i- are spent then \\
20: \hspace{2em} \textit{txo}_i^\alpha \leftarrow \text{Scan the ledger to learn which was spent first} \\
21: \hspace{2em} return \alpha \\
22: \hspace{1em} else // only \textit{txo}_i^\alpha is spent \\
23: \hspace{2em} return \alpha \\
24: end if \\
25: end procedure \\

Proof. The proof is by induction on \textit{d} the depth of component \textit{x}. When \textit{x} is a gate, denote its inputs as \textit{a} and \textit{b}.

1. For \textit{d} = 0, \textit{x} is an input-bit. Then by definition \text{LABEL}^*(\textit{x}) = \text{C}_g(\text{ASSIGN}^*(\textit{x})).

2. Assume the induction hypothesis holds for all \textit{d} \leq \textit{n}.

3. For \textit{d} = \textit{n} + 1, sub-circuit \textit{C}_x is made up of gate \textit{x} of depth \textit{n} + 1 whose two inputs \textit{a} and \textit{b} are the roots of two sub-circuits \textit{C}_a, \textit{C}_b of depth at most \textit{n}.

Since \text{LABEL}^*(\textit{x}) \neq \text{unset}, by corollary 2.3.3 \text{LABEL}^*(\textit{a}) \wedge \text{LABEL}^*(\textit{b}) \neq \text{unset}. Thus, both \text{LABEL}^*(\textit{a}) \neq \text{unset} and \text{LABEL}^*(\textit{b}) \neq \text{unset}, so by the induction hypothesis, \text{LABEL}^*(\textit{a}) = \text{C}_a(\text{ASSIGN}^*(\textit{a})) \text{LABEL}^*(\textit{b}) = \text{C}_b(\text{ASSIGN}^*(\textit{b})) and thus

\text{LABEL}^*(\textit{x}) = \neg(\text{LABEL}^*(\textit{a}) \wedge \text{LABEL}^*(\textit{b})) \\
= \neg(\text{C}_a(\text{ASSIGN}^*(\textit{a})) \wedge \text{C}_b(\text{ASSIGN}^*(\textit{b}))) \\
\text{ASSIGN}^*(\textit{x}) = \text{ASSIGN}^*(\textit{a}) \cup \text{ASSIGN}^*(\textit{b}) \\
= \neg(\text{C}_a(\text{ASSIGN}^*(\textit{x})) \wedge \text{C}_b(\text{ASSIGN}^*(\textit{x}))) \\
= \text{C}_g(\text{ASSIGN}^*(\textit{x}))
2 Compiling Arbitrary Circuits
3 An Ideal Ledger Functionality

One of the contributions of this paper is a new, formal definition for an ideal ledger functionality. Our motivation for the new definition is twofold. On the one hand, it should be useful, conforming to the intuitive notions of security we expect from a UTXO ledger, and simple enough to use in higher-level protocols. At the same time, it should be flexible enough to capture the functionality and security of a “meta-ledger” that is realized as a protocol overlaying a “base” ledger.

A note about UC (in)formalism

To make our pseudocode shorter and more readable, we define all ideal functionalities using functional interfaces rather than message passing. In the fully formal UC specification, an ideal functionality is an ITM that receives messages from external parties, and responds by sending messages. We replace sending a message of the form \((\text{Msg-Name}, x_1, x_2, \ldots)\) to functionality \(F\) with calling the function \(F.\text{MsgNAME}(x_1, x_2, \ldots)\). We also omit the session ID parameters (those are implied in all calls to a functionality).

Finally, we model the clock as an external functionality (as defined in \([2]\)). There are multiple subtleties in modeling global clocks in the UC setting (e.g., see \([5]\)). However, we make use of the clock only for defining liveness (rather than round synchronization). Thus, while our construction is compatible with the fully formal definition, in this paper we simplify by having the ideal functionality itself provide a \text{TICK} function, which can only be called by the environment to announce that the clock has advanced.

3.1 Basic Ledger Security

Most definitions for ledger security include the following properties, which we consider base requirements:

1. **Consensus**: the ledger outputs the same list of transactions, in the same order, to all honest parties.

2. **Validity**: all published transactions satisfy the ledger’s validity conditions.

3. **Liveness**: if an honest party sends a transaction to the ledger, it will be published by the ledger within some bounded time (which we call the \text{liveness delay}), unless it becomes invalid within that time. The liveness delay is a function of the transaction contents and the state of the ledger, and liveness might not be guaranteed at all for some valid transactions (e.g., if the transaction fee is too low).

We phrase our consensus guarantee as being about the ledger’s \text{output} to honest parties. This is a stronger guarantee than is generally given for blockchain formalizations; for example, it doesn’t allow disagreement on a suffix of the ledger or probabilistic agreement. We justify this by noting that most high-level protocols implicitly assume that the ledger actually does behave in this way, and when implemented over a real blockchain will wait until a transaction becomes “finalized” in order to use it (e.g., in Bitcoin a transaction could be considered “finalized” if it’s at least six blocks in the past).
For similar reasons, we do not consider properties such as chain quality that are important for analysis of the ledger protocol itself, but, as far as most higher-level protocols are concerned, are captured in one of the base guarantees (e.g., liveness in the case of chain quality).

**UTXO Validity**

In the case of a UTXO ledger, the validity conditions for every transaction include the following:

- **Valid sources**: For every transaction input, the transaction output it points to exists on the ledger and was not spent by a previous transaction.

- **Output condition is satisfied**: For every transaction input, the spending condition of the transaction output pointed to by that input is satisfied by the proof data.

- **Non-inflationary**: The sum of the output amounts is no more than the sum of the input amounts.

**Adversarial Capabilities**

As is usual in cryptography, we prefer our model to be conservative with regards to the adversary. In our modeling, we consider an adversary that has the following capabilities:

- The adversary can arbitrarily reorder transactions, as long as the liveness guarantee isn’t violated. This is justified by the fact that the adversary may have partial control over the network, and because an adversarial miner can determine the selection and order of transactions in a published block.

- The adversary can arbitrarily use the transaction fees from published transactions. This is justified by the fact that in most existing blockchains, the transaction fees are awarded to the block miner, who may be adversarial. Although we do strengthen the adversary by allowing it to use the fee immediately, we don’t believe this reduces the usefulness of the ledger when its output is fully immutable (as it is in our modeling).

Note that while the liveness delay is an upper bound on the time between the submission of a transaction and the time at which the transaction is published by the ledger, the adversary can choose to publish transactions earlier. This includes publishing transactions for which the liveness delay is infinite (i.e., they are not guaranteed to appear at all), such as transactions that have zero transaction fees.

### 3.2 Formalizing Bitcoin’s UTXO Ledger

Before describing our full functionality, we describe a simpler functionality that models Bitcoin’s UTXO ledger more closely. This serves as base for ours, and explaining its limitations will motivate the modifications we made to arrive at our generalized functionality.

The full pseudocode for the Bitcoin-UTXO functionality appears in functionality 3.1 (the “external” APIs appear in functionality 3.1 (part 1) while the internal functions appear in functionality 3.1 (part 2)).

For the honest parties, the functionality consists of a single API call: `send-tx(tx)`. Calling this function does not result in immediate publication; it merely inserts the transaction into a mempool.

---

1To simplify the discussion, we’re ignoring “coinbase” transactions, which are special transactions that do not have any inputs.

2The mempool corresponds to the “buffer” in [1], but we use the mempool nomenclature as it is more familiar to non-theorists in the blockchain space.
The adversary may call `process-tx(tx)` at any time. This will result in the `tx` being published if it’s valid, or being removed from the mempool if it’s not. This interface corresponds to the ability of the adversary to select which transactions will be included in a block.

Finally, the clock tick is used to guarantee liveness: when the `tick` method is called, the ledger will automatically process any transactions that have been in the mempool for longer than their liveness delay. This ensures liveness because the call to `tick` is not under adversarial control—it is called directly by the environment. (This is motivated by the fact that, w.h.p., after sufficient time any valid transaction that hasn’t previously been included in a block will be included by an honest miner.)

Internally, the Bitcoin ledger functionality uses the `validate-and-apply-tx` function to verify transaction validity. In this case, the validity functions exactly correspond to Bitcoin’s validity. The important point to note is that we abstract the scripting language to a generic “condition” that is specified for every transaction output, which must be satisfied by input data. The condition function receives not only the input data, but also the ID of the spending transaction and the UTXO database as inputs. The ID is required because Bitcoin’s signature verification opcode checks for a signature on the spending transaction’s ID. We added the UTXO as input to allow use of our new `is-txo-unspent` opcode as well.

### 3.3 Important Properties of the Ledger

The Bitcoin ledger follows Bitcoin’s actual implementation. Thus, while Bitcoin actually realizes this (or something similar in spirit), not all of the properties of the ideal ledger are actually important for building higher-level protocols. In this section, we note some of the properties that are important to preserve when defining a generalized ledger. In sections 3.4 and 3.5, on the other hand, we discuss the properties we are willing to (or must) sacrifice.

#### Simple Signatures

As defined, the Bitcoin ledger functionality itself is not aware of signatures, except as part of the spending condition script. This means there are no restrictions on the properties of the signature scheme (or even a requirement that signatures be verified in spending conditions). An additional important property is that creation of signatures is performed externally to the functionality, rather than using the functionality itself.

#### Client-Agnostic

The ledger functionality is *client agnostic*—the behavior of the ledger in response to honest parties depends only on the contents of the messages it receives, not their source.

This is also an important property in terms of implementing such a ledger: the gossip networks underlying most existing ledgers don’t record the source of a message.

Together with the fact that signatures are external to the functionality, this allows use-cases such as receiving an unsigned transaction, signing it, and sending the signed transaction through other, “off-chain” communication channels—once the transaction is signed, it doesn’t matter which party forwards it to the ideal ledger.

### 3.4 Challenges for a Meta-Ledger

Our goal is to realize a meta-ledger functionality with a richer condition language by compiling meta-transactions to multiple transactions on a base-ledger.

Ideally, we would like to simply realize the Bitcoin functionality described in functionality 3.1. Unfortunately, this is not always possible.
3.4.1 Spending Is Not Atomic

The main reason is that in the Bitcoin ledger functionality, once a transaction output enters the UTXO database, it can only be in one of two states: either it is (1) unspent (in the UTXO), in which case a transaction whose input points to that output and satisfies its condition is valid, or (2) it is spent—in which case the ledger contains a subsequent transaction that spends the output (and hence contains data that satisfies its spending condition).

However, if we compile a meta-transaction to multiple base-transactions using the is-txo-unspent opcode, in order to satisfy the spending condition of the “meta-output” a user might have to publish multiple “fragment” transactions. If it’s possible for this process to be interrupted by the adversary (e.g., by publishing a transaction that spends one of the fragment outputs), then the publishing of the meta-transaction might fail (e.g., the subsequent fragments are no longer valid).

Thus, a meta-output could end up in an “intermediate” state in which spending is not possible (even if the meta-output condition could be satisfied), but no satisfying assignment can be extracted from the ledger.

In our compiler (from chapter 2), this can occur, for example, if an adversary spends the outputs of the input-bit-labeling transactions—in this case an honest user might not be able to compile an assignment meta-transaction, even if she knows a satisfying assignment.

3.4.2 Outputs and Inputs Are Not Atomic

In Bitcoin’s ledger, the inputs and outputs of a transaction affect the ledger atomically—if a transaction spends a set $I$ of TXOs and generates a new set of TXOS $O$, either all TXOs in $I$ were spent and all TXOs in $O$ are in the UTXO, or neither happens.

However, requiring input/output atomicity would greatly restrict the class of realizable meta-ledgers—it would require every meta-transaction to correspond to a specific base-transaction that has exactly the same inputs and outputs.

If this is not ensured, a transaction that spends TXOs $I$ and creates TXOs $O$ may correspond to a set of base transactions in which one transaction, $tx_I$, spends the TXOs in $I$ while another, $tx_O$, creates the outputs $O$. Since the adversary is allowed to delay and reorder transactions, there may be a point in time at which $tx_I$ was published in the base-ledger while $tx_O$ was not.

This situation does, in fact, occur in our compiler, when compiling a meta-transaction that has an output with a circuit condition as described in chapter 2. The meta-transactions inputs are spent by the splitter transaction (c.f. section 2.1.4) while the outputs are generated by the circuit-keystone transaction (c.f. section 2.1.1).

Thus, we must relax the input/output atomicity requirement if we wish to support meta-ledger constructions such as ours.

3.4.3 Signatures Are Hard To Forge

Bitcoin’s spending-condition language supports signature verification—an output condition can require the input data to contain a valid signature of the spending transaction’s ID.

When we compile a meta-transaction into base transactions, we also have to translate the input data (including signatures) into input data that is valid in the base-ledger. Since the meta-transaction is (necessarily) different from any base transaction, the security of the signature scheme would prevent any attempt to naively transfer a signature from a meta-transaction to a signature on a base transaction.

3.4.4 Fragments Must Appear in Meta-Ledger Too

The compiler takes a single meta-transaction and creates multiple base transactions. However, these base transactions cannot be completely hidden from the meta-ledger. First, the spending
condition language can query the contents of the UTXO using is-txo-unspent (otherwise the ledger functionality would not be powerful enough to support our own compiler). This means that, at the very least, unspent outputs in the base-ledger must be reflected in the meta-ledger.

Second, fragment outputs may have monetary value. If they are hidden from the meta-ledger, an adversarial transaction that spends them in the base-ledger may also have to be hidden (since this input would not exist in the meta-ledger, such a transaction could violate the non-inflation rule); over time, this could cause a significant fraction of the base-ledger to be hidden.

The problem this creates is that the fragments in the meta-ledger must have a source of funding, but since they are the result of compilation (and do not appear in the input), their funding must somehow be “siphoned off” from the input of meta-transaction. Conceptually, this is fine—we treat the creation of fragments as part of the fee for the meta-transaction. However, the ledger guarantees that the input transactions appear in the output unmodified—and we cannot require the input transaction to have a specific format, such as an extra “fragment-funding” output (since the input to the ledger is set by the adversarial environment).

### 3.4.5 Fragments Must Be Recognizable

Finally, a technical point: if the meta-ledger can contain base transactions as well as meta transactions, honest parties must be able to differentiate between a base transaction that is a the result of a compiled meta-transaction and a regular base transaction. Otherwise, an honest party could receive as input the output of a valid compilation, in which case the ledger output would differ from its input (it would contain a meta transaction instead).

### 3.5 Our Generalized Ledger

The full pseudocode for our ledger functionality appears in functionality 3.2. In this section, we describe the differences between our generalized ledger and the simpler Bitcoin ledger, with intuitive justifications for the modifications.

#### 3.5.1 Addition of a locked Output State and a Locking Condition

In order to handle non-atomic spends (cf. section 3.4.1), we introduce a locked state for outputs. A locked output is still considered unspent (e.g., by the is-txo-unspent opcode), but any transaction that spends this output is not guaranteed liveness (this is equivalent to saying that honest users cannot spend locked outputs, but the adversary can).

Of course, allowing the adversary to lock outputs at will means that no transaction is guaranteed liveness. To restrict this, we add the concept of a locking condition (the locking condition language becomes an additional parameter of the ledger functionality).

Every output has both a locking condition (specified in the locking-condition language) and a spending condition (specified in the spending-condition language). In order to spend an output, both the locking condition and the spending condition must be satisfied by the input. In order to lock an output, it’s sufficient to satisfy the locking condition (note that locking can only be done by the adversary, and does not allow the adversary to transfer value from the locked output).

The addition of a locking condition is a generalization (rather than weakening) of the Bitcoin ledger. To see this, consider the case of identical locking and spending condition languages. Then honest users can simply specify the same condition for both locking and spending, in which case any adversary who could lock an output could also spend it (so locking does not confer extra capabilities).

In our protocol, we will use different spending and locking languages, allowing us to capture the “intermediate” state that occurs when fragments are spent in an adversarial way: the spending language will consist of circuits, and the locking language will be the locking language
of the base-ledger. Thus, an honest user can use the base-ledger language to prevent the adversary from locking outputs (e.g., by requiring a specific signature to satisfy the locking condition).

### 3.5.2 Addition of pending and pending-locked Output States

Our relaxation of input/output atomicity (c.f., section 3.4.2) is to allow the ledger to express an intermediate state where the inputs of a transaction have been applied to the state, but the outputs are not yet available for spending. Thus, we separate the publication of a transaction from the “release of its outputs”.

In the intermediate period, the outputs are in a new state: **pending**. The ledger does not allow spending of outputs in state **pending**. To preserve liveness guarantees, **pending** outputs automatically transition into state **unspent** after the maximum liveness delay has elapsed. (The adversary may choose to release the outputs of a transaction before that time.)

If, before a meta output is released, one of its corresponding fragment outputs was spent or locked, we require the meta output to transition into state **locked**, rather than state **unspent**. To mark it as such we introduce an additional state: **pending-locked**.

### 3.5.3 Allowing the Adversary to Choose Transaction IDs

We sidestep the problem of signature translation (cf. section 3.4.3) by allowing IDs to be “translated” instead—i.e., letting IDs from the base-ledger be “copied” to transactions in the meta-ledger (and vice versa), so that the signatures can be used unmodified.

We implement this in our ledger functionality by making two changes:

1. Whenever the ledger functionality encounters a new transaction, it asks the adversary for its ID (rather than using a fixed hash function). The ledger verifies that the IDs returned by the adversary are unique (otherwise it picks a unique ID itself).

2. We separate the ID of the transaction from the IDs of its outputs (instead of treating the TXO ID as a simple index relative to the transaction ID).

While these changes are a weakening of the ledger functionality (we give the adversary additional capabilities), they do preserve **collision-resistance** of IDs, which is the critical property used by most higher-level protocols.

#### Transactions may have multiple IDs

The changes above are not quite sufficient for our purposes, however. The reason lies in the way we use this capability in our security proof. In the proof, the simulator (ideal-world adversary) uses the ID-generation capability to copy an ID from a fragment output by the compiler to the meta-transaction. Once the meta-transaction is signed (this is done by the environment in our proof), the signature can then be used, unmodified as a signature of the fragment in the base-ledger.

However, the compiler is randomized—for example, our compiler generates a fresh signature key for every compilation, which is used in creating the fragment spending conditions. Thus, each call of the compiler on the same meta-transaction can produce different fragment transactions, with different IDs. To reflect this, our ledger allows a transaction to have multiple IDs—every request for a transaction ID will be answered by a fresh ID, even if it is for the same transaction.
IDs with Auxiliary Data

Allowing multiple IDs per transaction leads to a new problem, however: in order to allow the ledger to be client-agnostic (i.e., it doesn’t matter who sent a transaction, only what the transaction contains), the simulator needs additional information that would allow it to “invert” the transaction ID into the appropriate fragments.

We handle this by having the ledger return not just IDs but also some opaque “auxiliary data”, which must be sent together with the transaction. This auxiliary data is used internally by the ledger, but does not appear in the ledger output. (In our case, the auxiliary data is simply the entire result of the compilation)

To make things a little clearer, the following describes the interactions of an honest party, Alice, with the ledger in order to send a transaction:

1. Alice creates the transaction “core”, \( \hat{tx} \). This contains the inputs and output conditions, but without IDs or any signature data (Alice needs the IDs in order to generate signatures).
2. Alice sends \( \hat{tx} \) to the ledger and requests its IDs.
3. The ledger forwards \( \hat{tx} \) to the adversary, Eve, to request IDs, and waits for her response. Eve responds with the transaction ID and the IDs of the transaction outputs. She also sends the auxiliary data (The ledger ensures that all the IDs sent by Eve are fresh).
4. The ledger sends the IDs to Alice, together with the auxiliary data.
5. Alice computes the full transaction \( tx \), which includes both the IDs and signatures.
6. Alice sends \( tx \) together with the auxiliary data to the ledger to be published. (The ledger verifies that the IDs match the transaction and auxiliary data.)
7. The ledger output will contain \( tx \) (including the IDs and signatures), but not the auxiliary data.

The ledger doesn’t verify that Alice is the one who requested the transaction IDs; Alice could send \( tx \) and the auxiliary data to Bob, who could later send it to the ledger. This is an important property, since there are ledger protocols that rely on off-chain transmission of transactions.

Note that our protocol uses an ideal base-ledger, and the IDs output in the meta-ledger are a fixed function of the base-ledger IDs (which, if the base-ledger is Bitcoin, are set according to the Bitcoin rules). However, the protocol is still secure if based on the weaker ledger functionality we define.

### 3.5.4 Allowing (Adversarial) Partial Spending of a TXO

We address the problem of funding fragments in the meta-ledger (cf. section 3.4.4) by allowing the adversary to spend transaction fees arbitrarily. This is implemented in our generalized ledger by:

1. Letting the adversary decide, for every published transaction, an input amount for each of its inputs. The input amount specifies how much of the corresponding output’s value is used by that input. The sum of the input amounts must still be greater or equal to the sum of the output amounts.
2. Recording in the UTXO how much of an output amount remains. If an output is spent (i.e., its state in the UTXO is spent), but the remaining amount is greater than zero, we consider that output partially-spent.
3. A partially-spent output counts as spent for the purpose of is-txo-unspent, and transactions that attempt to spend it are not guaranteed liveness (i.e., honest parties cannot spend it). However,

4. Any input that spends a partially-spent output is valid, as long as the input value is not more than the remaining output amount. Thus, the adversary can use any remaining value without having to generate new signatures, for example.

Essentially, partially-spent outputs represent transaction fees. In our protocol, we use the meta-transaction fees to fund the fragment transactions in the meta-ledger by having both the meta-transaction and a fragment partially-spend the same source TXO.

3.5.5 “Fragment” and “Keystone” Marks on TXs

We require base transactions to be marked in some identifiable way that isn’t likely to occur in “natural” transactions. For simplicity, we assume that there are two types of marks: “Fragment” marks and “Keystone” marks.

The ledger functionality recognizes fragment and keystone transactions, and does not guarantee liveness for these transactions (this solves the problem from section 3.4.5).

We assume these marks are ledger-specific (i.e., a transaction that the meta-ledger recognizes as a fragment would be considered a standard transaction by the base-ledger)

In practice, these marks can be implemented using a special “unlikely” spending condition (e.g., fixing a publicly-known value $x$, and checking that $x$ appears in the input data).

3.5.6 Notation Notes

Syntactic Sugar

To make our proofs simpler, we added a REMOVE-TX($tx$) call to the functionality, which can be called by the adversary to remove the transaction $tx$ from the mempool—as long as $tx$ is invalid or liveness is not guaranteed when REMOVE-TX($tx$) is called. This is merely syntactic sugar, as the adversary would get the same result by calling PROCESS-TX($tx$) if $tx$ is invalid, or by letting $tx$ remain in the mempool and never processing it if its liveness is not guaranteed (since, in this case, the ledger functionality will never call PROCESS-TX($tx$) itself).

Ledger States and Ledger Output

In our ledger, a state contains an extension of a UTXO—for each output it includes its state, and also its output conditions and remaining amount.

The ledger state contains only the transaction outputs, not the transactions themselves. However, it is a deterministic function of the ledger output—which consists of the ordered list of published transactions and transaction output release events. (For ease of notation, when we use a ledger output out in place of a ledger state, this is simply shorthand for “the unique state $s$ computed from out”.)
Functionality 3.1 (part 1): Bitcoin UTXO Ledger: APIs

The functionality maintains:

- **mempool**: a set of transactions pending processing
- **UTXO**: set of unspent TXOs

---

**Honest API**

1. `procedure SEND-TX(tx)`
2. Add `tx` to `mempool`
3. Send `(Tx-Received, tx)` to `A`
4. `end procedure`

**Adversarial API**

5. `procedure PROCESS-TX(tx)`
6. if `tx` is in `mempool` then
7. Remove `tx` from `mempool`
8. `end if`
9. `UTXO' ← UTXO`
10. `is_success ← validate-and-apply-tx(tx)`
11. if `is_success` then
12. Send `(Tx-Published, tx)` to all parties
13. else
14. `UTXO ← UTXO'` // Restore “backup”
15. `end if`
16. `end procedure`

**External (Environment) API**

18. `procedure TICK()` // Can only be called by `E`
19. Output `(Tick)` only to `A`
20. for all `tx ∈ mempool` do
21. if
22. `tx` was added to `mempool` more than $\delta_{bitcoin}$ ticks ago and
23. IS-LIVE$^{bitcoin}(tx)$
24. then
25. `PROCESS-TX(tx)`
26. `end if`
27. `end for`
28. `end procedure`

...Continued in functionality 3.1 (part 2)
procedure VALIDATE-AND-APPLY-TX(tx) 

(\text{inputs}_{tx}, \text{outputs}_{tx}) \leftarrow \text{parse} tx 

\text{incoming}_{coins} \leftarrow \text{the sum of the amounts associated with inputs}_{tx} 

\text{outgoing}_{coins} \leftarrow \text{the sum of the amounts associated with outputs}_{tx} 

\text{if outgoing}_{coins} > \text{incoming}_{coins} \text{ and tx is not a coin-minting transaction then}

\text{return} \text{false} 

\text{end if} 

\text{for all } tx_i \in \text{inputs}_{tx} \text{ do:}

(\text{id}_{txo}, \text{data}_{txi}^{\text{spend}}) \leftarrow \text{parse} tx_i 

\text{if } \text{id}_{txo} \notin \text{UTXO} \text{ then} \quad \quad // \text{id}_{txo} \text{ is unrecognized}

\text{return} \text{false} 

\text{end if} 

(\text{amount}_{txo}, \varphi_{txo}^{\text{spend}}) \leftarrow txo 

\text{if } \varphi_{txo}^{\text{spend}}(\text{data}_{txi}^{\text{spend}}, \text{id}_{tx}, \text{UTXO}) = \text{false} \text{ then}

\text{return} \text{false} \quad // \varphi_{txo}^{\text{spend}} \text{ isn’t satisfied}

\text{end if} 

\text{Remove } txo \text{ from UTXO} 

\text{end for} 

\text{Add outputs}_{tx} \text{ to UTXO} 

\text{return} \text{true} 

\text{end procedure} 

function IS-LIVE_{\text{bitcoin}}(tx) 

\text{if } tx\text{’s fee is larger than or equal to min.fee then}

\text{return} \text{true} 

\text{else}

\text{return} \text{false} 

\text{end if} 

\text{end function}
Functionality 3.2 (part 1): $F_{\text{ledger}}^{L_{\text{spend}},L_{\text{lock}},\delta,\text{IS-LIVE}}$

(Honest API)

The functionality maintains:
- **mempool**: a set of transactions pending processing
- **UTXO**: maps output IDs to a tuple $(\text{state}, \text{amount}, \varphi^{\text{lock}}, \varphi^{\text{spend}})$
- **TxIds**: maps transactions to their IDs

1: **procedure** get-tx-ids($\hat{tx}$) // Called upon receiving (Get-Tx-IDs-Request, $\hat{tx}$)
2: $\hat{tx}_{\text{stripped}} \leftarrow$ strip $\hat{tx}$ of locking inputs and input amounts
3: Send (Create-Tx-IDs-Request, $\hat{tx}_{\text{stripped}}$) to $\mathcal{A}$
4: Await (Create-Tx-IDs-Response, $tx$, aux) response // $tx$ is of the form ($IDs$, $\hat{tx}_{\text{stripped}}$)
5:
6: if $tx$’s IDs are missing or already used then
7: Replace them with fresh random IDs
8: end if
9: Add ($tx$, aux) to the set $TxIds[\hat{tx}_{\text{stripped}}]$
10: return ($tx$, aux)
11: **end procedure**

12: **procedure** send-tx($tx$, aux) // Called upon receiving (Send-Tx, $tx$, aux)
13: $tx_{\text{stripped}} \leftarrow$ strip $tx$ of locking inputs and input amounts
14: $\hat{tx}_{\text{stripped}} \leftarrow$ strip $tx_{\text{stripped}}$ of transaction and output IDs
15: if $(tx_{\text{stripped}}, aux) \in TxIds[tx_{\text{stripped}}]$ then
16: Add $tx_{\text{stripped}}$ to mempool
17: Send (Tx-Received, $tx_{\text{stripped}}$, aux) to $\mathcal{A}$
18: end if
19: **end procedure**

...Continued in functionality 3.2 (part 2)...
3 An Ideal Ledger Functionality

Functionality 3.2 (part 2): $F^\mathcal{L}_\text{spend}, \mathcal{L}_\text{lock}, \delta, \text{IS-LIVE}_\text{ledger}$
(Adversarial API)

...Continued from functionality 3.2 (part 1)...

20: procedure PROCESS-TX($tx$)
21: $tx_{\text{stripped}} \leftarrow$ strip $tx$ of locking inputs and input amounts
22: $\hat{tx}_{\text{stripped}} \leftarrow$ strip $tx_{\text{stripped}}$ of transaction and output IDs
23: if $\nexists$ aux such that $(tx_{\text{stripped}}, aux) \in TxIds[\hat{tx}_{\text{stripped}}]$ then
24: return
25: end if
26:
27: $UTXO' \leftarrow$\textunderscore fresh\textunderscore copy $UTXO$
28: $\text{is\_success} \leftarrow$ VALIDATE-AND-APPLY-TX($tx$)
29: if $\text{is\_success}$ then
30: Remove $tx_{\text{stripped}}$ from mempool
31: Send (Tx-Published, $tx$) to all parties
32: else
33: $UTXO \leftarrow UTXO'$ // Restore “backup”
34: end if
35: end procedure

36: procedure RELEASE-TXOS($tx$)
37: if $tx$’s outputs are in state pending then
38: Set the states of $tx$’s output to unspent in $UTXO$
39: Send (Txos-Released, $tx$) to all parties
40: else if $tx$’s outputs are in state pending-locked then
41: Set the states of $tx$’s output to locked in $UTXO$
42: Send (Txos-Released, $tx$) to all parties
43: end if
44: end procedure

45: procedure REMOVE-TX($tx$) // Syntactic Sugar
46: $\hat{tx}_{\text{stripped}} \leftarrow$ strip $tx$ of locking inputs and input amounts
47: if $\hat{tx}_{\text{stripped}} \in$ mempool and $\neg$IS-STRONGLY-VALID($\hat{tx}_{\text{stripped}}$) then
48: Remove $\hat{tx}_{\text{stripped}}$ from mempool
49: end if
50: end procedure

...Continued in functionality 3.2 (part 3)...

24
Functionality 3.2 (part 3): \( F_{\text{ledger}}^{\text{spend}, \text{lock}, \delta, \text{IS-LIVE}} \)
(Adversarial API)

...Continued from functionality 3.2 (part 2)...

51: procedure tick() // Can only be called by \( \mathcal{E} \)
52: for all \( tx \in \text{mempool} \) do
53: if
54: \( tx \) was added to mempool \( \delta \) ticks ago and
55: IS-LIVE(\( tx \))
56: then
57: PROCESS-TX(\( tx \))
58: end if
59: end for
60: for all \( tx \in \{ \text{transactions published} \} \) do
61: if
62: \( tx \) was added to mempool \( \delta \) ticks ago and
63: \( tx \)'s outputs are pending or pending-locked in UTXO
64: then
65: RELEASE-TXOS(\( tx \))
66: end if
67: end for
68: Send (Tick) to all parties
69: end procedure

...Continued in functionality 3.2 (part 4)...

25
3. An Ideal Ledger Functionality

Functionality 3.2 (part 4): $\mathcal{F}_{\text{ledger}}^{\text{lock}, \delta, \text{IS-LIVE}}$

(Internals)

...Continued from functionality 3.2 (part 3)...

70: \textbf{procedure} VALIDATE-AND-APPLY-TX($tx$)
71: \quad $((id_{tx}, txo\_ids_{tx}), (inputs_{tx}, outputs_{tx})) \xrightarrow{\text{parse}} tx$
72: \quad // incoming coins \geq outgoing coins:
73: \quad \text{if } \sum_{tzo \in outputs_{tx}} \text{amount}_{tzo} > \sum_{txi \in inputs_{tx}} \text{amount}_{txi} \text{ then}
74: \quad \quad \text{return false}
75: \quad \end{if}
76: \quad // valid inputs:
77: \quad \text{if } \neg \text{VALIDATE-AND-APPLY-INPUTS}(\text{inputs}_{tx}, id_{tx}) \text{ then}
78: \quad \quad \text{return false}
79: \quad \end{if}
80: \quad \text{APPLY-OUTPUTS}(\text{outputs}_{tx}, tzo\_ids_{tx})
81: \quad \text{return true}
82: \quad \end{procedure}

83: \textbf{procedure} VALIDATE-AND-APPLY-INPUTS($\text{inputs}_{tx}, id_{tx}$)
84: \quad \text{for all } k \in [1 \ldots \text{size}(\text{inputs}_{tx})] \text{ do:}
85: \quad \quad txi \leftarrow \text{inputs}_{tx}[k]
86: \quad \quad (id, amount, data_{lock}, data_{spend}) \xrightarrow{\text{parse}} txi
87: \quad \quad \text{if } \psi_{\text{spend}} = \bot \text{ then}
88: \quad \quad \quad \text{if } \neg \text{VALIDATE-AND-APPLY-LOCKING-INPUT}(txi, id_{tx}) \text{ then}
89: \quad \quad \quad \quad \text{return false}
90: \quad \quad \quad \end{if}
91: \quad \quad \text{else}
92: \quad \quad \quad \quad \text{if } \neg \text{VALIDATE-AND-APPLY-SPENDING-INPUT}(txi, id_{tx}) \text{ then}
93: \quad \quad \quad \quad \quad \text{return false}
94: \quad \quad \quad \end{if}
95: \quad \quad \end{if}
96: \quad \end{for}
97: \quad \text{return true}
98: \quad \end{procedure}

...Continued in functionality 3.2 (part 5)...

26
Functionality 3.2 (part 5): \( \mathcal{F}_{\text{ledger}}^{\mathcal{L}_{\text{spend}}, \mathcal{L}_{\text{lock}}, \text{IS-LIVE}} \)
(Internals (cont.))

...Continued from functionality 3.2 (part 4)...

99: procedure VALIDATE-AND-APPLY-SPENDING-INPUT(txi, idtx)
100:   \((id_{txo}, amount_{txo}, \varphi_{lock}^{txo}, \varphi_{spend}^{txo}) \leftarrow \text{parse } txi\)
101:   // referenced output is in a spendable state:
102:   if \(\text{STATE}(UTXO, id_{txo}) \in \{\bot, \text{pending, pending-locked}\}\) then
103:     return false
104:   end if
105:   \((state_{txo}, amount_{txo}, \varphi_{lock}^{txo}, \varphi_{spend}^{txo}) \leftarrow UTXO[id_{txo}]\)
106:   // referenced output has sufficient remaining coins:
107:   if \(amount_{txi} > amount_{txo}\) then
108:     return false
109:   end if
110:   // referenced output is already spent:
111:   if \(state_{txo} = \text{spent}\) then
112:     SPEND-OUTPUT\( (id_{txo}, amount_{txo} - amount_{txi}) \)
113:     return true
114:   end if
115:   // locking condition is satisfied:
116:   if \(\varphi_{lock}^{txo}(data_{lock}^{txi}, id_{tx}, UTXO) = \text{false}\) then
117:     return false
118:   end if
119:   // spending condition is satisfied:
120:   if \(\varphi_{spend}^{txo}(data_{spend}^{txi}, id_{tx}, UTXO) = \text{false}\) then
121:     return false
122:   end if
123:   SPEND-OUTPUT\( (id_{txo}, amount_{txo} - amount_{txi}) \)
124:   return true
125: end procedure

126: procedure SPEND-OUTPUT\( (id_{txo}, amount_r) \)
127:    \((state_{txo}, amount_{txo}, \varphi_{lock}^{txo}, \varphi_{spend}^{txo}) \leftarrow UTXO[id_{txo}]\)
128:    \(UTXO[id_{txo}] \leftarrow (\text{spent}, amount_r, \varphi_{lock}^{txo}, \varphi_{spend}^{txo})\)
129: end procedure

...Continued in functionality 3.2 (part 6)...

27
...Continued from functionality 3.2 (part 5)...

130: procedure VALIDATE-AND-APPLY-LOCKING-INPUT(txi, id_{txz})
131:   (id_{txzo}, amount_{txz}, data_{lock}^{txz}, data_{spend}^{txz}) ← parse txi // data_{spend} = ⊥
132:   if state(UTXO, id_{txo}) = ⊥ then
133:     return false
134:   end if
135:   if amount_{txz} ≠ 0 then
136:     return false
137:   end if
138:   if state_{txo} ∈ {pending-locked, locked} then
139:     return true
140:   end if
141:   if locking condition is satisfied:
142:     if φ_{txo}^{lock}(data_{lock}^{txz}, id_{txz}, UTXO) = false then
143:       return false
144:     end if
145:     if txo’s state in UTXO is pending then
146:       Set txo’s state to pending-locked in UTXO
147:     else if txo’s state in UTXO is unspent then
148:       Set txo’s state to locked in UTXO
149:     end if
150:   end if
151: end procedure

...Continued in functionality 3.2 (part 7)...

28

3 An Ideal Ledger Functionality

Functionality 3.2 (part 6): $F_{\text{ledger}}^{\text{spend}, \text{lock}, \text{is-live}}$
(Internals (cont.))
Functionality 3.2 (part 7): $F_{\text{ledger}}^{\text{spend,lock,\text{is-live}}}$
(Internals (cont.))

...Continued from functionality 3.2 (part 6)...

156: **procedure** APPLY-OUTPUTS($outputs$, $txo\_ids$)
157:     **for all** $k \in [1 \ldots \text{SIZE}(outputs)]$ **do**
158:         $txo \leftarrow outputs[k]$
159:         $id \leftarrow txo\_ids[k]$
160:     **end for**
161:     **end procedure**

163: **procedure** IS-STRONGLY-VALID($tx$)
164:     $UTXO' \overset{\text{fresh copy}}{\leftarrow} UTXO$
165:     applied_successfully $\leftarrow$ VALIDATE-AND-APPLY-TX($tx$)
166:     $UTXO \leftarrow UTXO'$ // discard side-effects
167:     **if**
168:         applied_successfully **and**
169:         is-live($tx$) **and**
170:         $tx$ doesn’t spend any outputs in state pending-locked/locked/spent
171:     **then**
172:         return true
173:     **end if**
174:     return false
175: **end procedure**
4 Generically Realizing a Meta-Ledger

4.1 Overview of the Protocol

Our protocol (formally appearing as protocol 4.1) realizes the meta-ledger, so a party running it should first and foremost be able to send meta-transactions. The protocol realizes the meta-ledger over a base-ledger that does not recognize meta-transactions. Hence its logic can be divided into two main complementary categories:

- Compilation of meta-transactions to base-transactions and sending thereof to the base-ledger and
- Identifying these base-transactions as they are published by the base-ledger and the decompilation back to meta-transactions.

The output of an honest party running the protocol will differ from the output of the base-ledger, as some transactions published by the base-ledger will be modified before being output — the most-notable modification is the replacing of base-transactions with meta-transactions.

Compilation and sending are discussed in section 4.1.2, and monitoring the base-ledger and decompilation are discussed in section 4.1.3. However, we first address in section 4.1.1 a few decompilation complexities that affected the compilation as well.

Abstracting the Compilation/Decompilation of Meta-Transactions

The protocol is parameterized by a compiler/decompiler pair, \((\text{COMP}, \text{DECOMP})\). Given a meta-transaction \(tx^m\), the current meta-ledger output and the base-ledger output, \(\text{COMP}\) will compile \(tx^m\) into a sequence of base-transaction sets (the fragments), whose order indicates the order in which they should be sent (a formal description appears in definition 4.2.1). Given a base transaction, the meta-ledger output and the base-ledger output, \(\text{DECOMP}\) will attempt to reconstruct a meta-transaction (a formal description appears in definition 4.2.2).

The protocol works for any \((\text{COMP}, \text{DECOMP})\) pair satisfying some basic requirements (see section 4.2.4). When instantiated with the compiler/decompiler from chapter 5, \(\text{COMP}_{\text{Circuit}}\) (protocol 5.1) and \(\text{DECOMP}_{\text{Circuit}}\) (protocol 5.2), it realizes the meta-ledger with circuit-spending conditions.

In fig. 4.1.1 we give an example of compiling and decompiling two complementary meta-transactions that can be given as input to an honest party running protocol instantiated with the \(\text{COMP}_{\text{Circuit}}\) and \(\text{DECOMP}_{\text{Circuit}}\) from chapter 5. The figure is structured as follows:

- Column “Circuit” is dedicated to \(tx^m_{\text{circuit}}\), whose output \(txo^m_{\text{circuit}}\) contains circuit condition \(\neg(a \land b) = \text{true}\).
- Column “Source” is dedicated to \(tx^m_{\text{src}}\) whose output \(txo^m_{\text{src}}\) is spent by \(tx^m_{\text{circuit}}\).
- Column “Assignment” is dedicated to \(tx^m_{\text{assign}}\), whose input \(txi^m_{\text{assign}}\) contains assignment data “\(a \rightarrow \text{true}, b \rightarrow \text{false}\)”.
- Row “Honest Input” shows the transaction as it is given by \(E\) to the honest party to send.
- Row “COMP output to Base-Ledger” shows the fragments that are the output of \(\text{COMP}\) and that are sent in the real-world.
Row “DECOMP output to Meta-Ledger” shows the output of the honest party and DECOMP.

Special Fragments

We identify two special fragments from the output COMP creates for a meta-transaction $tx^m$. We require of COMP/DECOMP that exactly one of the fragments will share $tx^m$’s output IDs. We refer to that fragment as the keystone (c.f. definition 4.2.9). We also require that exactly one of the fragments will reference the same outputs as $tx^m$. We refer to that fragment as the splitter (c.f. definition 4.2.10).

We say a keystone is simple if it is also the splitter, that is, if it also spends the same outputs as $tx^m$. Such is $tx_{k, assign}$ (fig. 4.1.1, Base-Ledger, Assignment) with respect to $tx^m_{assign}$ (Input, Assignment).

We say a keystone is non-simple if the keystone and the splitter are two different fragments. $tx_{k,circuit}$ (fig. 4.1.1, Base-Ledger, Circuit) is a non-simple keystone with respect to $tx^m_{circuit}$ (Input, Circuit).

Note that both the keystone and the splitter of a decompiled base-transaction are the transaction itself.

High-level Protocol Description

First, the decompiler “queues” all transactions it receives from the base-ledger for a period of time before output. This allows it to modify transactions “retroactively” (up to the liveness delay of the meta-ledger).

For each new base-transaction that is received, the protocol determines which of the following three cases it falls into (this can be done efficiently in the case of circuit meta-transactions by using the keystone and fragment marks, together with pattern-matching on the output conditions):
Case 1: The transaction, together with a subset of the preceding transactions, could have been an output of COMP. In this case, it can also extract the input to COMP, and use that to compute a meta-transaction. (For fig. 4.1.1, this is always the case.)

Case 2: The transaction could not have been part of a COMP output, but an output it spends corresponds to a meta-output in the current meta-ledger.

Case 3: The transaction satisfies neither of the two conditions above.

In the third case, the decompiler can simply output the transaction unmodified. In the first two cases, it will output a meta-transaction. There are two main ideas that allow the decompiler to ensure the meta transaction that it outputs is valid in the meta-ledger:

1. If the keystone is simple (as is the case in the assignment transaction of fig. 4.1.1), it replaces the keystone with the meta-transaction, keeping the transaction ID and the output IDs unchanged. Thus, it can transfer signatures on the keystone unmodified to use as signatures on corresponding meta-transaction. In this case, the meta-transaction appears in the meta-ledger at exactly the same place at which the keystone appeared, so a valid keystone “automatically” implies that any base-outputs that the meta-transaction spends are also valid (for meta-outputs, validity is guaranteed from the compiler soundness).

2. If the keystone is not simple, the handling is a little more complicated. In this case, it can’t simply replace the keystone with the meta-transaction—because the splitter is the transaction that actually spends the source inputs (e.g., consider tx₁ in the Circuit column of fig. 4.1.1). Thus, it needs to transfer signatures from the splitter to the meta-transaction for it to be valid. The solution is to copy the splitter’s ID to the meta-transaction. However, we can’t just replace the splitter with the meta-transaction—the splitter’s outputs are spent by fragment transactions, which should still appear in the meta-ledger. Instead, the protocol gives the splitter a new ID, but inserts the meta-transaction before the splitter in the ledger output (this is one of the retroactive modifications it performs on the queue). The meta-transaction is modified to spend only part of the source inputs, so that the splitter can spend the rest. Because the ledger functionality allows partially spent transactions to be spent by the adversary without satisfying the spending conditions again, the modified splitter is valid in the meta-ledger.

In order to maintain the invariant that the UTXO state of the meta-ledger remains equivalent to the UTXO state of the base-ledger, the outputs of the meta-transaction aren’t released immediately (they enter in a “pending state”), but wait until the corresponding keystone outputs are released in the base-ledger (this corresponds to the dashed line in fig. 4.1.1).

Finally, there is one more “trick” used by the decompiler: whenever a transaction spends a fragment of a previously-decompiled transaction that has a meta-output, the decompiler adds a locking input to that transaction that locks the corresponding meta-output. This possible because the compiler is required to ensure that all fragment outputs have the same locking condition as the meta-output. The reason the decompiler must do this is because, otherwise an adversary could “wedge” a meta-output by spending some of its fragments (e.g., think of a circuit for which the adversary sets some of the input bits), in which case an honest party that compiles a valid meta-transaction isn’t guaranteed to be able to insert the corresponding fragments into the ledger (e.g., if the circuit-input used by the honest party is different from the wedged bits). By locking the meta-output, the meta-ledger prevents this situation from happening.
4.1.1 Decompiling Meta-Transactions

Valid Decompiled Transaction Inputs

For a transaction to be valid, its inputs need to be valid, which among other things, requires that they provide data satisfying the conditions of the outputs they reference.

In fig. 4.1.1, the input data of decompiled $tx^m_{circuit}$ (Meta-Ledger, Circuit) should satisfy $tx_{circ}$’s (Meta-Ledger, Source) and the input data of decompiled $tx^m_{assign}$ (Meta-Ledger, Assignment) should satisfy $tx^m_{circuit}$’s (Meta-Ledger, Assignment) conditions.

Output conditions often check whether the signature provided in the input data is valid. Recall that signatures are validated with respect to the ID of the spending transaction.

It is necessary to support meta-transactions that are required to provide signatures. A decompiled, honestly-sent transaction should be funded by the same outputs it originally spent (those that are spent in the base-ledger by the splitter). Additionally, due to the state-transition order preservation requirement, a decompiled transaction should be output by the honest party when its splitter is published in the base-ledger.

The way the protocol settles this is by “giving” the IDs of the splitters to the meta-transactions. In fig. 4.1.1, $tx^m_{circuit}$’s ID will be that of $tx_s$ (Meta-Ledger, Circuit), and $tx^m_{assign}$’s ID will be that of $tx_{k,assign}$’s input (Meta-Ledger, Assignment).

Valid Decompiled Inputs of Transactions With Non-Simple Keystones

Ideally, every decompiled meta-transaction would have a simple keystone. If that was the case, the honest party would have been able to simply replace the keystone (which is also the splitter) with the decompiled meta-transaction. However, it is sometimes useful to separate the inputs and the outputs of a meta-transaction to two different fragments, as is done by COMP Circuit (see $tx_s$ and $tx_{k,circuit}$ in fig. 4.1.1, Base-Ledger, Circuit, the splitter and the keystone of $tx^m_{circuit}$ respectively). Hence, not all keystones are simple.

The decompilation of such a meta-transaction is possible only after the keystone transaction is published by the base-ledger. Hence the protocol delays publishing of all transactions if any of those it needs to publish is a potential fragment. Delaying output means that the honest party can output the decompiled meta-transaction before it published the splitter. Delay is implemented using MsgQueue—a queue of messages the honest party maintains.

However, if the decompiled meta-transaction simply copies the inputs from the splitter, then between the two, they spend from the referenced outputs twice the amount the original meta-transaction had. Suffice to say for now that the honest party adjusts the input amounts of the two transactions so that added-up they spend the same amount the original meta-transaction had. This is also made possible thanks to the delay the honest party introduces.

In the scenario in fig. 4.1.1, the honest party would not output the splitter $tx_s$ immediately as it is published by the base-ledger. It would wait, in case the keystone is published. If and when the keystone $tx_{k,circuit}$ is published, DECOMP Circuit can be used to decompile $tx^m_{circuit}$. The honest party would output decompiled meta-transaction $tx^m_{circuit}$ before $tx_s$ after having adjusted the input amounts of the two transactions.

The Use of Partial Spending

The transfer of IDs causes a different issue though, which is only relevant for meta-transactions with non-simple keystones ($tx_s$ in fig. 4.1.1, Meta-Ledger, Circuit) — on the one hand, we still want the splitter to appear in the meta-ledger and therefore in the output of the honest party. On the other hand, its ID cannot be the same as it was in the base-ledger since the ID is already taken by the meta-transaction, and IDs are unique.

This is where partial spending comes into play (see section 3.5.4) — the honest party first outputs the decompiled meta-transaction. Since the splitter’s inputs are valid in the base-ledger
with respect to the splitter’s ID, the inputs of the decompiled meta-transaction will be valid. Only then does it output the splitter — since its inputs spend already-spent outputs, they are valid.

### 4.1.2 Compiling and Sending of Transactions

The honest party’s API appears in protocol [4.1 (part 1)] and includes GET-TX-IDS (line 11) and SEND-TX (line 10).

Both the ledger’s SEND-TX and the honest party’s SEND-TX accept an already-signed transaction. $E$ requires a transaction’s ID in order to provide a signed transaction to the SEND-TX procedures, since signing a transaction essentially means signing its unique ID. Hence the GET-TX-IDS procedures — both the ledger and the protocol implement it.

When asked for the IDs of transaction $tx$, the protocol’s GET-TX-IDS invokes COMP with $tx$. COMP internally uses the ledger’s GET-TX-IDS procedure, and all the fragments, apart from the splitter, are returned signed. $tx$’s IDs a returned alongside $aux$ auxiliary data required for sending — the entire output of COMP.

The IDs returned by the protocol’s GET-TX-IDS are selected in a very specific way, to allow the output of the honest party to be indistinguishable from a meta-ledger as per the considerations discussed in section [4.1.1]:

- The transaction ID is always chosen to be that of the splitter. Recall that “later”, during decompilation, the ID of the splitter is “moved” to the decompiled meta-transaction.
- The output IDs are chosen to be those of the keystone (in the case of a simple keystone, and therefore also in the case of a base transaction, these are also the output IDs of the splitter).

When asked to send transaction $tx$ with auxiliary data $aux$ (that is, in fact, the fragments), the honest party sends the fragments instead of $tx$. It “signs” the splitter by planting $tx$’s signature in the splitter’s inputs. The honest party also verifies $aux$ is a valid and admissible COMP output for $tx$.

The fragments are sent one step at a time — the $i^{th}$ step is sent when all the outputs of step $(i-1)^{th}$ have been released. This logic is implemented in TXOS-RELEASED-BY-$F_{ledger}$ (line 52), since the function is invoked whenever the ledger releases outputs. SEND-NEXT-STEP (line 77) contains the logic for sending the next step, and it invoked from two places — from SEND-TX for sending the first step, and from TXOS-RELEASED-BY-$F_{ledger}$ to send all other steps.

This mechanism helps guarantee that if the first step, that contains the splitter, is published by the honest party, then the adversary cannot interrupt the publication of all other steps and fragments (see condition [3] of definition [4.2.18]). Consider $tx_m^{circuit}$ (fig. 4.1.1, Input, Circuit) — since $tx_a$ and $tx_b$ spend $tx_s$’s outputs (fig. 4.1.1, Input, Base), they must be sent after $tx_s$’s outputs have been released. Otherwise they are invalid until such time as $tx_s$’s outputs are released, and if the ledger attempts to process them, they will be discarded.

The honest party also verifies when asked to send a transaction that it is live — this is required for proving that the output of the honest party is indistinguishable from the meta-ledger.

### 4.1.3 Processing Base-ledger Output and Decompilation

An honest party is subscribed to messages output by the base-ledger — when $(Tx-Published, tx)$ is output, TX-PUBLISHED-BY-$F_{ledger}$ ($tx$) is run (line 23 of protocol [4.1 (part 2)]).

As the honest party sees transactions being published by the ledger it considers each to see whether they should be replaced with meta-transactions. This is in line 25 done by calling DECOMP with the published transaction $tx$, and based on the base and meta states. The call returns $tx^m$. If the decompilation succeeds, then $tx^m$ is a meta-transaction and it is returned
alongside its fragments; if \( tx \) is not a keystone transaction, or if the decompilation fails, then \( tx^m \) is identical to \( tx \), and \( tx \) is also the only fragment.

The transaction that the honest party outputs is not necessarily identical to \( tx^m \) as it was output by DECOMP. It is also not output immediately — instead, it is added somewhere in the Deciding what to output is based (mostly) on whether the keystone is simple.

### A Simple Keystone Was Published

SIMPLE-KEYSTONE-PUBLISHED (line 115) is called if the decompiled transaction has a simple keystone, with the keystone \( tx_k \) and the meta-transaction \( tx^m \). The honest party pushes a Tx-Published message at the end of \( MsgQueue \) with a transaction made up from \( tx^m \)'s inputs using \( tx^m \)'s ID and from \( tx_k \)'s outputs using \( tx_k \)'s output IDs, discarding \( tx^m \)'s outputs.

This applies to fig. 4.1.1's \( tx^m_{assign} \) (Meta-Ledger, Assignment) — it is made up of the inputs of the decompiled meta-transaction and the outputs of \( tx_{k,assign} \), using the ID of the decompiled meta-transaction and the output IDs of the keystone.

SIMPLE-KEYSTONE-PUBLISHED can also be called if the decompiled transaction has a non-simple keystone, but its splitter has already been output so it is impossible to reduce its amounts. In which case, the honest party treats the keystone as a standard base-transaction, pushing a Tx-Published message at the end of \( MsgQueue \) with the keystone.

### A Non-Simple Keystone Was Published

NON-SIMPLE-KEYSTONE-PUBLISHED (line 120) is called if the decompiled transaction has a non-simple keystone, with the keystone \( tx_k \), the splitter \( tx_s \) and the meta-transaction \( tx^m \). The honest party creates three transactions:

1. \( tx^m_s \), from \( ts_s \)'s inputs using \( ts_s \)'s ID and \( tx^m \)'s outputs using \( tx^m \)'s output IDs (\( tx^m \)'s inputs are discarded),

2. \( tx_s \), a fresh copy of the splitter, but using a new unique ID,

3. \( tx_{dummy} \), from \( tx_k \)'s input using \( tx_k \)'s ID (it is ensured by condition 7 of definition 4.2.13 that there only exists one) and with no outputs.

Denoting \( amount_k \) the sum of the output amount of \( tx_k \), the honest party reduces \( amount_k \) from \( tx_{dummy} \)'s input and from the \( ts_s \)'s output it spends. It then fixes the input amounts of \( ts_s \) and of \( tx^m \) by greedily reducing a total of \( amount_k \) from \( ts_s \)'s inputs and setting \( tx^m_s \)'s amounts to whatever was reduced from \( ts_s \) inputs. Note that now \( tx^m_s \)'s inputs specify the same total amount as its outputs (\( amount_k \)).

Finally three modifications are made to \( MsgQueue \):

1. a (Tx-Published, \( tx^m_s \)) is added before (Tx-Published, \( ts_s \)),

2. (Tx-Published, \( ts_s \)) is replaced with (Tx-Published, \( ts_s \)) and

3. (Tx-Published, \( tx_{dummy} \)) is pushed to the end.

This applies to fig. 4.1.1's \( tx^m_{circuits} \) (Meta-Ledger, Circuit) — it is made up of the inputs of \( ts_s \) and the outputs of the decompiled meta-transaction, using the ID of \( ts_s \) and the output IDs of the decompiled meta-transaction.
4 Generically Realizing a Meta-Ledger

Adding Locking Inputs

Every time \textsc{tx-published-by-$F_{\text{ledger}}$} is called, \textsc{add-locking-inputs} (line 100) is called internally. \textsc{add-locking-inputs} goes over all \textsc{Tx-Published} messages in \textit{MsgQueue} and adds locking inputs when required. Adding a locking input to \textit{tx} is required if \textit{tx} spends a locking fragment output that was published by the base-ledger before \textit{tx}. A locking input locks the meta-output corresponding to the fragment output.

4.2 Definitions for Transaction Compiler/Decompiler

4.2.1 Input and Output Behavior

We first give formal definitions for \textsc{comp} and \textsc{decomp},

\textbf{Definition 4.2.1} (\textsc{comp}). On input,
- a base-ledger output \textit{out}_b,
- a meta-ledger output \textit{out}_m and
- a meta-transaction \textit{tx}_m,
\textsc{comp} outputs \([T_{tx_1}^m, \ldots, T_{tx_n}^m]\), where \(T_{tx_i}^m\) for \(i \in [n]\) is a set of tuples of the form: (\textit{tx}, aux), where \textit{tx} is transaction with IDs and aux is auxiliary data required for sending \textit{tx}.

To keep notation simple, we use \((\text{fragments, tx}_b) \leftarrow \text{comp}(\textit{out}_b, \textit{out}_m, \textit{tx}_m)\) when we only care about distinguishing between the keystone and the rest of the fragments, and treat \textit{fragments} as a set of transactions when we don’t care about the internal order.

\textbf{Definition 4.2.2} (\textsc{decomp}). On input,
- a base-ledger output \textit{out}_b,
- a meta-ledger output \textit{out}_m and
- a base-transaction \textit{tx}_b,
\textsc{decomp} outputs a tuple \((\textit{tx}_m, \text{fragments})\) where \textit{tx}_m is a meta-ledger transaction and \textit{fragments} is a set of base-transactions.

4.2.2 Relations between Base and Meta-Ledger States

In order to use a \textsc{comp}/\textsc{decomp} pair, they must satisfy completeness and validity requirements. We first define some terms that will allow us to specify the requirements,

\textbf{Definition 4.2.3} (State Order). We define the following complete ordering on transaction output states:

\[
\perp < \text{pending} < \text{pending-locked} < \text{unspent} < \text{locked} < \text{spent}.
\]

This ordering satisfies the property that an output’s state in the ledger is weakly monotone—it cannot change from a higher to a lower order state.

\textbf{Definition 4.2.4} (Ledger State Extension). We say a ledger state \(s^{ext}\) is an \textit{extension} of a ledger state \(s\) if it satisfies for every \textit{tzo}:
- \text{STATE}(\textit{tzo}, s) \leq \text{STATE}(\textit{tzo}, s^{ext})
4 Generically Realizing a Meta-Ledger

- If \( \text{STATE}(\text{txo}, s) = \text{pending-locked} \) then \( \text{STATE}(\text{txo}, s^{\text{ext}}) \neq \text{unspent} \).

Note that a ledger state \( s^{\text{ext}} \) is an extension of a state \( s \) iff there exist ledger outputs \( \text{out}^{\text{ext}} \) and \( \text{out} \) such that \( s^{\text{ext}} \) is the state computed from \( \text{out}^{\text{ext}} \), \( s \) is the state computed from \( \text{out} \) and \( \text{out}^{\text{ext}} \) is an extension of \( \text{out} \).

**Definition 4.2.5** (\( \leadsto \) Weak Equivalence). Let \( s_a \) and \( s_b \) be two ledger states. We say that \( s_a \) and \( s_b \) are weakly-equivalent and denote \( s_a \stackrel{\text{state}}{\leadsto} s_b \) if for every transaction output ID \( id \):

- \( \text{STATE}(s_a, id) \in \{\bot, \text{pending}, \text{pending-locked}\} \iff \text{STATE}(s_b, id) \in \{\bot, \text{pending}, \text{pending-locked}\} \).
- \( \text{STATE}(s_a, id) \in \{\text{unspent}, \text{locked}\} \iff \text{STATE}(s_b, id) \in \{\text{unspent}, \text{locked}\} \).
- \( \text{STATE}(s_a, id) = \text{spent} \iff \text{STATE}(s_b, id) = \text{spent} \).

The weak equivalence definition captures equivalence with respect to spending and locking conditions—we require that weakly-equivalent states must be “indistinguishable” to conditions that take state into account.

**Definition 4.2.6** (\( \leadsto \) Consistency). Let \( s_b \) and \( s_m \) be two ledger states. We say that \( s_m \) is consistent with \( s_b \) and denote \( s_m \stackrel{\text{state}}{\leadsto} s_b \) if \( s_m \stackrel{\text{state}}{\leadsto} s_b \) and for every transaction output \( \text{txo}^m \in s_m \):

1. \( \text{STATE}(s_b, id_{\text{txo}^m}) \leq \text{STATE}(s_m, id_{\text{txo}^m}) \)
2. \( \text{REMAINING}(s_m, id_{\text{txo}^m}) = \text{REMAINING}(s_b, id_{\text{txo}^m}) \)
3. \( \varphi_{s_m, id_{\text{txo}^m}}^{\text{lock}} = \varphi_{s_b, id_{\text{txo}^m}}^{\text{lock}} \)

This definition captures the property that the adversary has at least as much “control” over outputs in the meta-ledger as it does in the base-ledger (i.e., if a transaction is live in the meta-ledger—that is, if the ledger could “force” the adversary to process, it must also be live in the base-ledger).

**Definition 4.2.7** (Explainable TXO). Let \( \text{out}_b \) and \( \text{out}_m \) be ledger outputs and \( \text{txo}^m \) a transaction output of \( \text{tx}^m \in \text{out}_m \). We say \( \text{txo}^m \) is explainable with respect to \( \text{out}_b \) and \( \text{out}_m \) if there exists a transaction \( \text{tx}^b \in \text{out}_b \) with output \( \text{txo}^b \) such that \( id_{\text{txo}^b} = id_{\text{txo}^m} \), and

\[
\begin{cases}
\text{txo}^m = \text{txo}^b & \text{if } \varphi_{\text{txo}^m}^{\text{spend}} \text{ is in } L_{\text{base}} \\
\exists \text{fragments} : (\text{tx}^m, \text{fragments}) = \text{DECOMP}(\text{out}_b^{\text{pre}}, \text{out}_m^{\text{pre}}, \text{tx}^b) & \text{otherwise}
\end{cases}
\]

(where \( \text{out}_b^{\text{pre}} \) is the prefix of \( \text{out}_b \) up to, excluding \( \text{tx}^b \) and \( \text{out}_m^{\text{pre}} \) is the prefix of \( \text{out}_m \) up to, excluding \( \text{tx}^m \)).

When \( \text{txo}^m \) is explainable w.r.t \( \text{out}_b \) and \( \text{out}_m \), we call the set of fragments returned by \( \text{DECOMP} \) the explanatory fragments for \( \text{txo}^m \) (this set is empty if \( \text{txo}^m \) is a base output).

**Definition 4.2.8** (\( \leadsto \) Consistency). Let \( \text{out}_b \) and \( \text{out}_m \) be ledger outputs, and \( s_b \) and \( s_m \) their corresponding ledger states. We say that \( \text{out}_m \) is consistent with \( \text{out}_b \) and denote \( \text{out}_m \leadsto \text{out}_b \) if \( \text{out}_m = \text{out}_b \) or if \( s_m \stackrel{\text{state}}{\leadsto} s_b \) and for every transaction output \( \text{txo}^m \in s_m \), \( \text{txo}^m \) is explainable with respect to \( \text{out}_b \) and \( \text{out}_m \).
This definition captures the property that all outputs of the meta-ledger state “can be explained” as either a valid decompilation of the base-ledger, or an exact copy of a base-ledger output.

We note that \(\text{out}_m \xrightarrow{tx} \text{out}_b\) implies that a transaction output ID appears in \(s_m\) iff it appears in \(s_b\) (if it exists in \(s_b\), then item \(\square\) implies it exists in \(s_m\), while explainability implies the other direction).

### 4.2.3 Special Transactions

In order to concisely define the required behavior of \(\text{COMP}\) and \(\text{DECOMP}\), we give names to base transactions with specific properties with relation to meta-transactions.

Let \((tx^m, \text{fragments}) \leftarrow \text{DECOMP}(\text{out}_b, \text{out}_m, tx)\) be the output of \(\text{DECOMP}\).

**Definition 4.2.9 (Keystone).** If there exists a single base transaction \(tx_k \in \text{fragments}\) such that \(tx_k\) and \(tx^m\) have the same output IDs, we call \(tx_k\) the keystone transaction for \((tx^m, \text{fragments})\).

**Definition 4.2.10 (Splitter).** If there exists a single base transaction \(tx_s \in \text{fragments}\) such that \(tx_s\) and \(tx^m\) reference the same output IDs, we call \(tx_s\) the splitter transaction for \((tx^m, \text{fragments})\).

**Definition 4.2.11 (Simple Keystone).** We say a keystone transaction \(tx_k\) is a simple keystone if it is also the splitter transaction.

A keystone that is not a simple keystone is a non-simple keystone.

For \(\text{COMP}_{\text{Circuit}}\) and \(\text{DECOMP}_{\text{Circuit}}\) (protocols 5.1 and 5.2) that are discussed in sections 5.1 and 5.2, and use the construction explained in chapter 2, the compilation of a meta-circuit-condition output yields a non-simple keystone \((tx^m, \text{fragments}^\ast) \leftarrow \text{DECOMP}(\text{out}^\ast_b, \text{out}^\ast_m, tx)\), where \(tx^m\) is identical to \(tx^m\) up to input amount changes.

In addition to completeness, we require both \(\text{COMP}\) and \(\text{DECOMP}\) to satisfy a set of validity requirements separately:

**Definition 4.2.12 (COMP/DECOMP Completeness).** We say the pair \((\text{COMP}, \text{DECOMP})\) is complete if they satisfy the following condition:

For every base-ledger output \(\text{out}_b\), meta-ledger output \(\text{out}_m\) such that \(\text{out}_m \xrightarrow{tx} \text{out}_b\), every meta transaction \(tx^m\) that is valid in \(\text{out}_m\) and every execution \((\text{fragments}, tx_k) \leftarrow \text{COMP}(\text{out}_b, \text{out}_m, tx^m)\), if \(\text{out}^\ast_b\) is an extension of \(\text{out}_b\) that contains all the transactions in \(\text{fragments}\), then \((tx^m, \text{fragments}) \leftarrow \text{DECOMP}(\text{out}^\ast_b, \text{out}^\ast_m, tx_k)\), where \(tx^m\) is identical to \(tx^m\) up to input amount changes.

In addition to completeness, we require both \(\text{COMP}\) and \(\text{DECOMP}\) to satisfy a set of validity requirements separately:

**Definition 4.2.13 (DECOMP Validity).** We say the decompiler \(\text{DECOMP}\) is valid if, for every base-ledger output \(\text{out}_b\), every meta-ledger output \(\text{out}_m\) such that \(\text{out}_m \xrightarrow{tx} \text{out}_b\) and every transaction \(tx\) such that \(tx\) is valid in \(\text{out}_b\), \((tx^m, \text{fragments}) \leftarrow \text{DECOMP}(\text{out}_b, \text{out}_m, tx)\) satisfies all of the following conditions:
4 Generically Realizing a Meta-Ledger

Condition 1: fragments $\subseteq \text{out}_b$

Condition 2: $tx$ is a keystone transaction and there exists a splitter transaction $tx_s \in \text{out}_b||tx$ (it is possible that $tx_s = tx$).

Condition 3: For every transaction output $txo$ of $tx^m$, the output amount of $txo$ (in the meta-ledger) is equal to the amount of the corresponding output of the keystone $tx$ (in the base-ledger).

Condition 4: If $tx^m$ has meta outputs, then there exists a condition $\phi^{lock} \in \mathcal{L}_{lock}$ such that $\phi^{lock}$ is the locking condition for the meta outputs and for all fragment outputs that are not spent by other fragments.

Condition 5: For every base transaction output $txo^b$ of $tx^m$, the spending and locking conditions of $txo^b$ (in the meta-ledger) are identical to those of the corresponding output of the keystone $tx$ (in the base-ledger).

Condition 6: If $tx$ spends a meta-output in $\text{out}_m$, then $tx^m$ has a simple keystone.

Condition 7: If $tx^m$ has a simple keystone, then $tx^m$ is valid in $\text{out}_m$. (Note that if $tx^m$ has a non-simple keystone, there are no requirements about its validity.)

Condition 8: If $tx$ is not simple, it only spends a single output of the splitter $tx_s$.

The validity conditions for COMP ensure that compilation generates valid base transactions, and that we can always compute meta-transaction IDs deterministically from base-transaction IDs. The conditions also ensure that a computationally-bounded adversary cannot “block” the transactions generated by COMP from being inserted into the ledger. To this end, we will require some additional definitions.

**Definition 4.2.14** (Live Transaction Sequence). Let $\text{out}_b$ be a ledger output and $T = (tx_1, \ldots, tx_n)$ a sequence of transactions. We say $T$ is live in $\text{out}_b$ if for all $i \in \{1, \ldots, n\}$ it holds that $tx_i$ is live in $\text{out}_b||tx_1||\cdots||tx_{i-1}$.

That is, a sequence is live if an honest party is guaranteed to be able to publish the transactions in the sequence, in order, as long as no other transactions are published in between.

**Definition 4.2.15** (Valid Transaction Sequence). Let $\text{out}_b$ be a ledger output and $T = (tx_1, \ldots, tx_n)$ a sequence of transactions. We say $T$ is valid in $\text{out}_b$ if for all $i \in \{1, \ldots, n\}$ it holds that $tx_i$ is valid in $\text{out}_b||tx_1||\cdots||tx_{i-1}$.

A valid sequence can be appended to the ledger by the adversary, but isn’t guaranteed to be live. However, every live sequence is necessarily valid.

**Definition 4.2.16** (Blocking Transaction Sequence). Let $\text{out}_b$ be a ledger output and $T = (tx_1, \ldots, tx_n)$ a sequence of transactions. We say a transaction sequence $X$ blocks $T$ in $\text{out}_b$ if the sequence $T^\prime = T \setminus X$ (consisting of all transactions in $T$ that are not in $X$) is not live in $\text{out}_b||X$. (We consider the empty sequence to be live.)

We can now define what it means for a sequence to be “unblockable”. Intuitively, we want to capture the property that the transaction fragments output by COMP can either be blocked “early” (in our case this will be before some prefix is published) or not at all. Formally,

**Definition 4.2.17** (Unblockable Sequence Distribution). Let $\text{out}_b$ be a ledger output and $\mathcal{T}_1, \mathcal{T}_2$ distributions over transaction sequences. We say $\mathcal{T}_2$ is an unblockable sequence distribution following $\mathcal{T}_1$ in $\text{out}_b$ if for every computationally-bounded adversary $\mathcal{A}$,

$$\Pr\left[(T_1, T_2) \leftarrow (\mathcal{T}_1, \mathcal{T}_2), X \leftarrow \mathcal{A}(1^\lambda, \text{out}_b, T_1, T_2), T_1 \subseteq X, X \text{ is valid and blocks } T_2\right] < \varepsilon(\lambda)$$
where $\varepsilon$ is a negligible function of the security parameter $\lambda$, and the probability is over the distribution $T$ and the coins of the adversary.

To understand the definition, think of $T_1, T_2$ as a single long sequence output by COMP; the first part might be blockable, but if it’s not ($T_1 \subseteq X$), the second part cannot be blocked.

**Definition 4.2.18 (COMP Validity).** COMP is valid iff it satisfies the following conditions for every base-ledger output $out_b$, every meta-ledger output $out_m$ such that $out_m \xrightarrow{tx} out_b$, every meta transaction $tx^m$ that is live in $out_m$ and $(\text{fragments}, tx_k) \leftarrow \text{COMP}(out_b, out_m, tx^m)$:

**Condition 1:** *(Splitter exists)* $(\text{fragments}, tx_k)$ contains a splitter for $tx^m$. (The splitter can be $tx_k$ itself.)

**Condition 2:** *(Fragments prior to splitter are blockable only by locking relevant outputs)* Let $S$ be the set of txos spent by $tx^m$. We define the set of relevant outputs for $S$ to be:

- Transaction-outputs in $S$.
- Transaction outputs of explanatory fragments for txos in $S$ (c.f. definition 4.2.7).

For every extension $out^b_{ext}$ of $out_b$ that does not contain the splitter transaction, if $(\text{fragments}, tx_k) \setminus out^b_{ext}$ is not live in $out^b_{ext}$, then either there exists a transaction in $out^b_{ext}$ that locks a relevant output, or for every extension $out^m_{ext}$ of $out_m$ such that $out^m_{ext} \xrightarrow{tx} out^b_{ext}$, it holds that $tx^m$ is not live in $out^m_{ext}$.

**Condition 3:** *(Fragments following splitter are unblockable)* Loosely speaking, this condition ensures that if the splitter has been published, no computationally-bounded adversary can prevent the remaining fragments from being published.

Define the distribution $T(out^b, out_m, tx^m) = \text{fragments}$ to consist of the fragments output by COMP$(out_b, out_m, tx^m)$, $T_1$ to be the fragments in $T(out^b, out_m, tx^m)$ up to (including) the the splitter fragment $tx_k$ (ensured by condition 1) and $T_2$ the remaining fragments.

Then $T_2$ is an unblockable sequence distribution following $T_1$ in $out_b$ (c.f., definition 4.2.17).

**Definition 4.2.19 (Admissibility).** A pair of algorithms (COMP, DECOMP) is admissible if DECOMP satisfies definition 4.2.13, COMP satisfies definition 4.2.18 and the pair satisfies COMP/DECOMP completeness.

The formal description of our generic ledger protocol appears in protocol 4.1 (external APIs are in protocol 4.1 (part 1), handling of ledger events (i.e. methods triggered by output of the base-ledger) is in protocols 4.1 (part 2) and 4.1 (part 3) and internal methods are in protocols 4.1 (part 5) and 4.1 (part 6).

### 4.3 Security Analysis of the Generic Meta-Ledger Protocol

We prove the security of the protocol in the UC model. More formally, our main security claim appear in theorem 4.3.1.

**Theorem 4.3.1 (Protocol 4.1 realizes $F^{\text{meta}}_{\text{ledger}}$).** For all $L_{\text{base}}, L_{\text{lock}}$ such that $L_{\text{base}}$ satisfies .... and every $l$-valid COMP$L_{\text{meta}} \rightarrow L_{\text{base}} / \text{DECOMP}L_{\text{base}} \rightarrow L_{\text{meta}}$ pair and all $\delta_{\text{base}} > 0$, protocol 4.1 realizes $F_{\text{ledger}}^{L_{\text{meta}} \cup L_{\text{base}}, L_{\text{lock}}, \delta_{\text{meta}}, \text{IS-LIVE}^{\text{meta}}}$ in the $F_{\text{ledger}}^{L_{\text{base}}, L_{\text{base}}, \delta_{\text{base}}, \text{IS-LIVE}^{\text{base}}}$-hybrid model,
for $\delta_{meta} = f(\delta_{base})$ and

\[
\text{IS-LIVE}^{meta}(tx) = \text{tx is base} \land \text{IS-LIVE}^{base}(tx) \land \text{tx is not marked as fragment or keystone} \\
\lor \text{tx is meta} \land I(tx)
\]

## 4.3.1 Proof Overview

Our proof uses a standard hybrid argument. We build four hybrids, $H_1$-$H_4$, where $H_1$ is the real-world and $H_4$ is the ideal-world with the full simulator (algorithm 4.1).

$H_2$ and its indistinguishability from $H_1$ are discussed in section 4.3.2. The main difference between $H_2$ and $H_1$ is that in $H_2$ the ideal-world $S$ simulates a single honest party (denote it as $S_{P_h}$) which handles all honest inputs and generates outputs for all honest users, whereas in $H_1$ there are multiple independent honest parties. (In addition, there is a ‘syntactic’ difference in that in $H_2$ the real base-ledger $\mathcal{F}_{ledger}$'s output replaces a simulated ledger (denote it as $S_{F_{base}}$); the ideal functionality in $H_2$ forwards to $S$ any message it receives and is told by $S$ what to output (the $S_{P_h}$'s output).

$S$ is able to simulate all honest parties by using a single instance of an honest party since $\mathcal{F}_{ledger}$ is agnostic of the identity of the party that sent it a transaction, and the honest party does not maintain its identity or any other identity-bound property (e.g., signing key or randomness).

Since the clock is shared between all parties, and $\Pi_{ledger}$ is deterministic (COMP can be random, but it does not accept the identity of the caller) the outputs of all honest parties “monitoring” the base-ledger are identical.

$H_3$ and its indistinguishability from $H_2$ are discussed in section 4.3.3. $H_3$ adds honest party input extraction — honest party inputs are now sent to an instance of $\mathcal{F}_{meta}$ and $S$ learns what input was given to the honest parties from messages it receives from the $\mathcal{F}_{meta}$'s instance. Proving indistinguishability is relatively easy, since $\mathcal{F}_{ledger}$ notifies $A$ when it had received messages, specifying the arguments.

$H_4$ and its indistinguishability from $H_3$ are discussed in section 4.3.4. $S_{P_h}$'s output is no longer output by the ideal-functionality. Instead the output of the $\mathcal{F}_{meta}$ instance is used as honest party output, and $S$ translates $S_{P_h}$ output to calls to the adversarial API of $\mathcal{F}_{meta}$.

Due to the $H_3$ leap and since $S$ forwards messages $E$ sends $\mathcal{F}_{base}$ to $S_{F_{base}}$, the inputs to $S_{F_{base}}$ are indistinguishable from those of $\mathcal{F}_{base}$ in real-world ($H_1$). Hence the messages $S_{P_h}$ sees from $S_{F_{base}}$ are indistinguishable from those the honest parties see from $\mathcal{F}_{meta}$ in the real-world.

Since $S_{P_h}$ is an instance of $\Pi_{ledger}$, it acts like the real-world honest parties. When $S_{P_h}$ outputs (Tx-Published, tx), $S$ calls $\mathcal{F}_{ledger}$'s PROCESS-TX with tx and when $S_{P_h}$ outputs (Txos-Released, tx), $S$ calls $\mathcal{F}_{meta}$'s RELEASE-TXOS with tx. In corollary 4.3.26 we show these calls always succeed, and that otherwise the state of $\mathcal{F}_{meta}$, doesn’t change.

To prove corollary 4.3.26 we show that $s_{m+q}^{state} = s_b$, i.e., that the application of $S_{P_h}$'s $\text{MsgQueue}$ to $\mathcal{F}_{meta_{ledger}}$'s state $(s_q)$:

- Consistency is defined in definition 4.2.6 and captures the difference between the states of the meta-ledger and the underlying base-ledger (e.g., meta-outputs replace their corresponding base keystone outputs, meta-inputs replace their corresponding base splitter inputs).
- Recall that the honest party maintains a queue $\text{MsgQueue}$ that causes a delay and a “gap” between $S_{P_h}$ and $\mathcal{F}_{meta_{ledger}}$ (messages received from the ledger are not output immediately).

Applying a $\text{MsgQueue}$ to a ledger state is defined in definition 4.3.17 and essentially means calling different methods with respect to the ledger state, thereby changing it —
PROCESS-TX for Tx-Published messages, RELEASE-TXOS for Txos-Released messages and REMOVE-TX for Tx-Removed messages.

Hence we prove in lemma 4.3.25 that the combined state of $F^\text{meta} _\text{ledger}$ and $S_{F^\text{base} _\text{ledger}}$ is “always” consistent with that of $S_{F^\text{base} _\text{ledger}}$ — to do this we map out all possible modifications to the states of $F^\text{meta} _\text{ledger}$, $S_{F^\text{base} _\text{ledger}}$, and $S_{P_h}$, and group them into eight event types (event types 1 to 8). Each event is associated with a list of code segments that change one or more states ($F^\text{meta} _\text{ledger}$/$S_{F^\text{base} _\text{ledger}}$/$S_{P_h}$).

Event type 1's code segments, for example, cover the successful processing of a transaction within 2·δ $\delta_{base}$. Event type 2's code segments, for example, cover the successful processing of a transaction in $S_{F^\text{base} _\text{ledger}}$ and the application of the transaction to $S_{P_h}$'s MsgQueue. Event type 3's code segments, cover the popping of a Tx-Published message from $S_{P_h}$'s MsgQueue and the processing of the transaction in $F^\text{meta} _\text{ledger}$.

In claim 4.3.23 we show that all changes to the three states occur in one of the events, in claim 4.3.22 we show that the code segments of each event run consecutively during simulation and are disjoint from code segments of other events, and in lemma 4.3.24 we show that (most of) the events preserve consistency — that if the three states were consistent, and the event occurred (i.e., all code segments were run), then the resulting states were still consistent. Hence we can prove lemma 4.3.25 ($s_{m+q} \xleftarrow{s_{m+q}} S_{P_h}$ is always preserved) by induction on events.

To help us prove the each event maintains consistency separately, we show in lemmas 4.3.20 and 4.3.21 that the consistency is preserved “even though” the base-ledger keystone is decomplied into a meta-transaction, and in lemma 4.3.19 we prove the addition of locking inputs to transactions in the meta-ledger (that do not exist in the base-ledger) also preserves consistency.

We prove lemmas 4.3.19 to 4.3.21 by using a number of supporting claims that mainly show consistency is maintained in different settings. In lemma 4.3.16 we show that the application of an identical transaction to two consistent states will yield consistent states. Claim 4.3.15 shows the same holds for an identical output and claim 4.3.14 for an identical list of inputs.

To show that the application of an identical list of inputs preserves consistency (claim 4.3.14), we prove similar arguments for a single identical locking-input (lemma 4.3.13), spending-input (Lemma 4.3.12), and for the spending of an amount from the same output claim 4.3.11.

To prove lemma 4.3.25 it is also required to show that all changes in $F^\text{meta} _\text{ledger}$ correspond are changes in $S_{P_h}$ and $S_{F^\text{base} _\text{ledger}}$ — that the $F^\text{meta} _\text{ledger}$ state is not changed due to internal $F^\text{meta} _\text{ledger}$ processes. Particularly, that no event of event types 7 and 8 ever occurs. To prove this, we use claim 4.3.10 that if a (Send-Tx, $tx$) is given to any honest party in the ideal-world, then within 2·$s$·$\delta_{base}$, $S$ has called either $F^\text{meta} _\text{ledger}$'s PROCESS-TX and RELEASE-TXOS with $tx'$ (which is almost identical to $tx$) or $F^\text{meta} _\text{ledger}$'s REMOVE-TX with $tx$.

In order to prove claim 4.3.10 we showed there exist upper bounds on the time it takes an honest party to send a transaction ($s$·$\delta_{base}$, see claim 4.3.9) and on the time an honest can delay a message after it had been published by the ledger (a further $s$·$\delta_{base}$, see claim 4.3.8).

### 4.3.2 $H_2$ — Single Honest Party

Following are the differences between $H_2$ and $H_1$:

1. In $H_2$ the real base-ledger $F^\text{base} _\text{ledger}$ is replaced with a simulated ledger (denote it as $S_{F^\text{base} _\text{ledger}}$) per definition 4.3.2.
2. In $H_2$ the ideal-world $S$ simulates all honest parties by using a single instance (denote it as $S_{P_h}$) per definition 4.3.3 whereas in $H_1$ there are multiple independent honest parties.
3. In $H_2$ the ideal functionality forwards to $S$ any message it receives and is told by $S$ what to output (the $S_{P_h}$’s output) — $S$ is notified when $E$ sends a message to an honest party; messages output from $S_{P_h}$ are passed to $E$ (simulating the real-world honest parties).
4 Generically Realizing a Meta-Ledger

4. When the ideal functionality receives a (Tick)message, $S$ invokes $S_{F_{base}}$’s TICK.

Definition 4.3.2 (S simulation of $F_{base}^*ledger$). $S$ simulates $F_{ledger}^*$ — the simulated instance is referred to as $S_{F_{base}}^*ledger$.

- All messages sent from $E$ to $F_{ledger}^*$ are sent to $S_{F_{base}}^*ledger$.
- All messages output from $S_{F_{base}}^*ledger$ are passed to $E$.

Definition 4.3.3 (S simulation of honest parties). $S$ simulates an honest party running $\Pi_{ledger}$ — the simulated instance is referred to as $S_{P_h}$.

- When $SP_h$ sends a message to $F_{base}^*ledger$, $S$ pauses the $SP_h$ simulation, storing its execution state, runs the $S_{F_{base}}^*ledger$ code that handles the message, and finally restores the $SP_h$ execution state and continues simulating it.
- When $SP_h$ outputs a message, $S$ pauses the $SP_h$ simulation, storing its execution state, runs the $S_{F_{base}}^*ledger$ code that handles the message, and finally restores the $SP_h$ execution state and continues simulating it.
- When $S_{F_{base}}^*ledger$ outputs a message addressed to honest parties, $S$ pauses the $S_{F_{base}}^*ledger$ simulation, storing its execution state, runs the $SP_h$ code that handles the message, and finally restores the $S_{F_{base}}^*ledger$ execution state and continues simulating it.

Lemma 4.3.4. $H_1 \sim H_2$.

Proof. $S$ is able to simulates all honest parties by using a single instance of an honest party since $F_{ledger}^*$ is agnostic of the identity of the party that sent it a transaction, and the honest party does not maintain it’s identity or any other identity-bound property (e.g., signing key or randomness).

The send-tx receives an already-signed transaction (as the transaction or aux). Hence it is $E$’s responsibility to sign transactions.

Since the clock is shared between all parties, and $\Pi_{ledger}$ is deterministic (comp can be random, but it does not accept the identity of the caller) the outputs of all honest parties “monitoring” the base-ledger are identical.

Finally, since for every $F_{metaledger}^*$ tick, $S$ calls $S_{F_{base}}^*ledger$’s tick the outputs of $H_1$ and $H_2$ are indistinguishable. 

4.3.3 $H_3$ — Honest Party Input Extraction

In $H_3$, $S$ extracts $E$ honest party inputs from $F_{metaledger}^*$ messages it receives, whereas in $H_2$, $S$ receives honest party inputs from the ideal functionality.

The extraction is done as follows (see PROCESS-$F_{metaledger}^*$MESSAGE in line 1 of algorithm 4.1):

- A $F_{ledger}^*$ (Tx-Received, tx) message was triggered by a (Send-Tx, tx) input and
- A $F_{ledger}^*$ (Create-Tx-IDs-Request, tx) message was triggered by a (Get-Tx-IDs-Request, tx) input.

Lemma 4.3.5. $H_2 \sim H_3$.

Proof. By observation of $F_{ledger}^*$:

- $F_{ledger}^*$ outputs (Create-Tx-IDs-Request, $\hat{tx}$) on each invocation of GET-TX-IDS($\hat{tx}$) (line 3 of functionality 3.2 (part 1)), and hence on each (Get-Tx-IDs-Request, $\hat{tx}$) honest party input, and
• $F_{ledger}$ outputs $(Tx-Received, tx)$ on each invocation of $send-tx(tx)$ (line 17 of functionality 3.2 (part 1)), and hence on each $(Send-Tx, tx)$ honest party input.

The outputs are identical since:
• for each $(Create-Tx-IDs-Request, \hat{tx})$ from $F_{meta}$ ledger, the $S$ simulates $E$ sending $(Get-Tx-IDs-Request, \hat{tx})$ input to $S_{P_h}$ (line 4) and returns a $Create-Tx-IDs-Response$ with $S_{P_h}$’s output, and
• for each $(Tx-Received, tx)$ from $F_{meta}$ ledger, the $S$ simulates $E$ sending $(Send-Tx, tx)$ input to $S_{P_h}$ (line 8).

4.3.4 $H_4$ — Complete Ideal-World Simulation

In $H_4$ the output of the honest parties is that of the ideal honest parties, i.e., the output of $F_{meta}$ ledger, whereas in $H_3$ the output of the honest parties is that of $S_{P_h}$.

$S$ “connects” $S_{P_h}$ and $F_{meta}$ ledger in $process-S_{P_h}$-message (line 14 of algorithm 4.1):

• when $S_{P_h}$ outputs a $(Tx-Published, tx)$ message, $S$ calls $F_{ledger}$’s process-tx with $tx$, and
• when $S_{P_h}$ outputs a $(Txos-Released, tx)$ message, $S$ calls $F_{meta}$ ledger’s release-txos with $tx$.

Honest sending of a meta transactions in the idela-world is atomic, however it is not in the real-world where it can be interrupted. To guarantee that a transaction that was sent to $F_{meta}$ ledger and was interrupted is not published, $S$ removes the transaction from $F_{meta}$ ledger when it notices that was interrupted.

The identification part is done in check-$S_{FBase}$-Fragment-Validity (line 26). The procedure is called after each change to the state of $S_{FBase}$ ledger (a transaction was published or the outputs of a transaction were released), and after $S_{P_h}$ has finished applying the change to its internal state. Upon identification, $S$ pushes a $Tx-Removed$ message to $S_{P_h}$’s $MsgQueue$.

The actual removal is done when $S_{P_h}$ pops the message from $MsgQueue$. $S$ “sees” the popping and calls $F_{ledger}$’s $remove-tx$ with the transaction.

Lemma 4.3.6. $H_3 \sim H_4$.

Proof. Note that only $process-tx$, $release-txos$ calls translate to $F_{meta}$ ledger output visible to honest parties. By corollary 4.3.26, all $S$ calls to $F_{meta}$ ledger’s $process-tx$ and $release-txos$ succeed and these procedures are only called by $S$.

By observation of $S$’s $process-S_{P_h}$-message (line 14), $S$ calls these procedures whenever they happen in $S_{P_h}$ with the exact same arguments. □

Claim 4.3.7 ($F_{ledger}$ — Transaction processing delay). For every transaction $tx$, if $F_{ledger}$ (functionality 3.2) is given input $(Send-Tx, tx)$ and $tx$ is live $(\text{is-live}(tx) = true)$, then within $\delta$ ticks $process-tx$ is called with $tx$ and then $release-txos$ is called with $tx$.

Proof. By observation of $F_{ledger}$’s $tick$ (line 51, functionality 3.2). □

Claim 4.3.8 ($\Pi_{ledger}$ — $MsgQueue$ publishing delay). For every message $m$, if $m$ is added to $MsgQueue$ in time $t$ (ticks), then $m$ is popped from $MsgQueue$ by time $t + s \cdot \delta$.

Proof. By observation of $\Pi_{ledger}$’s $flush-msg-queue$ (line 84), popping from $MsgQueue$ is halted only if the message at the head of the queue is a $(Tx-Published, tx)$ message and the following conditions hold:
Algorithm 4.1 (part 1): $L_{Meta,Comp,Decomp}^\mathfrak{c}$

- $S$ simulates $F_{Base}^{ledger}$ per definition 4.3.2 — the simulated instance is referred to as $S_{F_{Base}^{ledger}}$.
- $S$ simulates all honest parties by using a single instance per definition 4.3.3 — denote it as $S_{P_h}$. In fact, it uses a slightly-modified version of $\Pi_{ledger}$ — $MsgQueue$ may now also contain $Tx-Removed$ messages.
- When $S_{P_h}$ pops a $Tx-Removed$ message from $MsgQueue$ in line 94 of $flush-msg-queue$, and $S$’s $S_{P_h}.tx-removed-popped$ is called, the message is regarded as an output of $S_{P_h}$ in the sense that it pauses the $S_{P_h}$ simulation.

1: procedure $process-F_{meta}^{ledger}-message(m)$
2:  // Called when $F_{meta}^{ledger}$ sends a message $m$
3:  if $m = (Create-Tx-IDs-Request, \hat{tx})$ then
4:    Simulate $E$ sending $(Get-Tx-IDs-Request, \hat{tx})$ input to $S_{P_h}$
5:    Await $(Get-Tx-IDs-Response, tx, aux)$ response
6:    Send $(Create-Tx-IDs-Response, tx, aux)$ to $F_{meta}^{ledger}$
7:  else if $m = (Tx-Received, tx, aux)$ then
8:    Simulate $E$ sending $(Send-Tx, tx, aux)$ input to $S_{P_h}$
9:  else if $m = (Tick)$ then
10:     $S_{F_{Base}^{ledger}}.tick()$
11: end if
12:  // $Tx-Published$ and $Txos-Released$ messages are discarded
13: end procedure

...Continued in algorithm 4.1 (part 2)...

1. $tx$ is marked as fragment and
2. $tx$’s outputs are not in $FragmentTxs$ and
3. $tx$ has been published by $F_{ledger}$ less than $s \cdot \delta$ ticks ago.

Suppose that messages in $MsgQueue$ are sorted by the time they were inserted. In which case the claim holds. Suppose in contradiction the popping is halted due to message $m_{halt}$, and that $MsgQueue$ contains a message $m_{delayed}$ that was added to the queue more than $s \cdot \delta$ ticks ago. Since $MsgQueue$ is sorted $m_{halt}$ too was added to the queue more than $s \cdot \delta$ ticks ago and thus should have been popped.

In fact, messages in $MsgQueue$ are “almost sorted” — we show that any message that is not added at $MsgQueue$’s end or that is modified in-place, won’t cause the popping to halt. We do this by an analysis of all the places where $MsgQueue$ is modified:

- In lines 55, 118 and 149 a $Tx-Published$ message is pushed at the end of $MsgQueue$.
- In line 148 a message is modified. The modification changes input/output amounts of the transaction in the $Tx-Published$ message, and thus does not affect the popping from $MsgQueue$.
- In line 147 a $(Tx-Published, tx^m)$ is inserted in the middle of $MsgQueue$. $tx^m$ is the result of a successful $Decomp$ call and therefore cannot be a fragment and will be output when first considered and will not halt the popping from $MsgQueue$.
Claim 4.3.9 (Simulation — Send-Tx to Tx-Published/Tx-Removed delay). For every transaction $tx$, if input $(Send-Tx, tx)$ is given to $S_{\mathsf p_h}$, then within $s \cdot \delta_{\mathsf base}$ ticks either:

- $(Tx-Published, tx')$ and then $(Txos-Released, tx')$ are added to $\mathit{MsgQueue}$, where $tx'$ is identical to $tx$ up to input amount changes and addition of locking inputs or
- $(Tx-Removed, tx)$ is added to $\mathit{MsgQueue}$.

Proof. By observation of $\Pi_{\mathsf l}^\mathsf M$:

- $tx$’s first step is sent immediately in line 20 of Send-Tx (line 10).
- $tx$’s $(l + 1)^{th}$ step is sent when all of the outputs in steps $\leq l$ were released in lines 59 to 61 of Txos-Released-by-$F_{\mathsf l}$ (line 52).
- $tx$ is removed from $\mathit{TxStepQueues}$ if the last step sent was sent more than $\delta_{\mathsf base}$ ticks ago in line 73 of Tick-by-$F_{\mathsf l}$ (line 68).

Hence, if the outputs of each step’s transactions are released within $\delta_{\mathsf base}$ (see claim 4.3.7), then by definition 4.2.12, Decomp returns a transaction $tx^m$ that is identical to $tx$ up to input amount changes, and within $s \cdot \delta_{\mathsf base}$ ticks $(Tx-Published, tx^m)$ and $(Txos-Released, tx^m)$ are added to $\mathit{MsgQueue}$.

If within $s \cdot \delta_{\mathsf base}$ such a message was not added to $\mathit{MsgQueue}$ then at least one step’s outputs were not published, which implies that it was removed from mempool, which in turn implies that $tx$ had not been strongly-valid.

By observation of $\mathcal S$’s (algorithm 4.1) Check-Check-$\mathcal S_{\mathsf base}$-Fragment-Validity (line 26), if a fragment of $tx$ became not strongly-valid, $\mathcal S$ would push a $(Tx-Removed, tx)$ to $S_{\mathsf p_h}$’s $\mathit{MsgQueue}$. □

Claim 4.3.10 (Simulation — Send-Tx to $F_{\mathsf l}$ calls delay). For every transaction $tx$, if input $(Send-Tx, tx)$ is given to any honest party in the idea-world at time $t$ (ticks), then by time $t + 2 \cdot s \cdot \delta_{\mathsf base}$ $\mathcal S$ has called either:

- $F_{\mathsf l}$’s Process-Tx and then Release-Txos with $tx'$, where $tx'$ is identical to $tx$ up to input amount changes and addition of locking inputs or
- $F_{\mathsf l}$’s Remove-Tx with $tx$.

Proof. When $F_{\mathsf l}$’s Send-Tx is called with transaction $tx$, a $(Tx-Received, tx)$ message is sent to $\mathcal A$ (see functionality 3.2 (part 1) line 17). When $\mathcal S$ sees the message, it simulates sending a $(Send-Tx, tx)$ input to $S_{\mathsf p_h}$ (see line 8 of algorithm 4.1).

Suppose in contradiction that by $t + s \cdot \delta_{\mathsf base}$ neither happened, that is, neither Process-Tx nor Remove-Tx were called.

By claim 4.3.8 by time $t + s \cdot \delta_{\mathsf base}$, neither $(Txos-Released, tx')$ for any such $tx'$ nor $(Tx-Removed, tx)$ were present in $S_{\mathsf p_h}$’s $\mathit{MsgQueue}$.

By claim 4.3.9 by time $t$, $S_{\mathsf p_h}$ was not given input $(Send-Tx, tx)$, contradicting the assumption that it did. □

Claim 4.3.11 ($s_m \overset{\mathit{state}}{\sim}_{txo} s_b$ is preserved by $F_{\mathsf l}$’s Spend-Output). Let $s_m$ and $s_b$ be two $F_{\mathsf l}$ states, and suppose $s_m \overset{\mathit{state}}{\sim}_{txo} s_b$, and let $id_{txo}$ be an output ID and $\mathit{amount}_r$ be an amount. Define:

- $s_b'$ as the state resulting from running Spend-Output($id_{txo}, \mathit{amount}_r$) (line 129 functionality 3.3) with respect to $s_b$ and
\begin{itemize}
\item $s'_m$ as the state resulting from running \textit{Spend-Output}$(\text{id}_{txo}, \text{amount}_r)$ with respect to $s_m$.
\end{itemize}

If $\text{STATE}(s_m, \text{id}_{txo}) \neq \bot$, then $s'_m \xrightarrow{\text{txo}} s_b$.

\begin{proof}
Since $s_m \xrightarrow{\text{txo}} s_b$, $\text{STATE}(s_b, \text{id}_{txo}) \neq \bot$. Both base and meta states are changed to \textit{spent} and both amounts to $\text{amount}_r$. All other $\xrightarrow{\text{txo}}$ conditions follow from $s_m \xrightarrow{\text{txo}} s_b$.
\end{proof}

\begin{lemma}
\label{lemma:4.3.12}
$s_m \xrightarrow{\text{txo}} s_b$ is preserved by $\mathcal{F}_{\text{ledger}}$'s \textit{Validate-and-Apply-Spending-Input}.
\end{lemma}

Let $s_m$ and $s_b$ be two $\mathcal{F}_{\text{ledger}}$ states, and suppose $s_m \xrightarrow{\text{txo}} s_b$, and let $tx_b$ and $tx_m$ be two identical transaction inputs and $id_x$ be a transaction ID and define:

\begin{itemize}
\item $s'_b$ as the state resulting from running \textit{Validate-and-Apply-Spending-Input}(tx_b, id_x) (line 99, functionality 3.2) with respect to $s_b$ and
\item $s'_m$ as the state resulting from running \textit{Validate-and-Apply-Spending-Input}(tx_m, id_x) with respect to $s_m$.
\end{itemize}

If $\varphi^{\text{spend}}_{s_m, \text{id}_{txo}} \neq \bot$ and $\varphi^{\text{spend}}_{s_m, \text{id}_{txo}} \in \mathcal{L}_{\text{base}}$ where $\text{id}_{txo}$ is the referenced transaction output ID, then either:

\begin{itemize}
\item both calls return \textit{true} and $s'_m \xrightarrow{\text{txo}} s'_b$ or
\item both calls return \textit{false}.
\end{itemize}

\begin{proof}
Denote the amount spent as $\text{amount}_{txi}$, and the spend and lock data as $\text{data}_{\text{spend}}$ and $\text{data}_{\text{lock}}$ respectively.

The proof is by exhaustive case analysis of all lines in which \textit{Validate-and-Apply-Spending-Input} can return. For line $l$ we show that either both calls return or both don’t and that when they do, they return the same boolean value and consistency is preserved.

1. Line 103 (false is returned if the referenced output is not in a spendable state): Due to weak equivalence, $\text{STATE}(s_n, id) = \{\bot, \text{pending}, \text{pending-locked}\} \iff \text{STATE}(s_b, id) \in \{\bot, \text{pending}, \text{pending-locked}\}$.

Hence, either both calls will return \textit{false} in line 103 or neither will and we can assume in the following lines that $\text{STATE}(s_m, \text{id}_{txo}) \notin \{\bot, \text{pending}, \text{pending-locked}\}$ and $\text{STATE}(s_b, \text{id}_{txo}) \notin \{\bot, \text{pending}, \text{pending-locked}\}$.

2. Line 108 (false is returned if the referenced output has insufficient remaining coins): Since $s_m \xrightarrow{\text{txo}} s_b$, $\text{remaining}(s_m, \text{id}_{txo}) = \text{remaining}(s_b, \text{id}_{txo})$.

Hence, if $\text{amount}_{txi} > \text{remaining}(s_b, \text{id}_{txo}) = \text{remaining}(s_m, \text{id}_{txo})$, both calls will return \textit{false} in line 108.

Otherwise, both calls will not return in line 108 and we can assume in the following lines that $\text{amount}_{txi} \leq \text{remaining}(s_b, \text{id}_{txo}) = \text{remaining}(s_m, \text{id}_{txo})$.

3. Line 113 (true is returned if the referenced output is spent): Since $s_b$ and $s_m$ are weakly-equivalent, $\text{STATE}(s_b, \text{id}_{txo}) = \text{spent} \iff \text{STATE}(s_m, \text{id}_{txo}) = \text{spent}$.

Hence, if $\text{STATE}(s_b, \text{id}_{txo}) = \text{STATE}(s_m, \text{id}_{txo}) = \text{spent}$, both calls will return \textit{true} in line 113. Also, claim 1.3.11 applies to the \textit{Spend-Output} call in line 112 so consistency is preserved.

Otherwise, both calls will not return in line 113 and we can assume in the following lines that $\text{STATE}(s_m, \text{id}_{txo}) \neq \text{spent}$ and $\text{STATE}(s_b, \text{id}_{txo}) \neq \text{spent}$.

\end{proof}
4. Line 117 (false is returned if the locking condition is not satisfied): Since $s_m \rightarrow_{txo} s_b$, $\varphi_{lock}^{s_m, id_{txo}} = \varphi_{lock}^{s_b, id_{txo}}$. Since $s_m$ and $s_b$ are weakly equivalent, and because $\varphi_{lock}$ is agnostic to \( \perp / \text{pending/pending-locked} \) and \( \text{unspent/locked} \) deviations, $\varphi_{lock}^{s_m, id_{txo}}(data_{lock}^{m}, id_{tx}, UTXO_{s_m}) = \varphi_{lock}^{s_b, id_{txo}}(data_{lock}^{m}, id_{tx}, UTXO_{s_b})$. Hence, if $\varphi_{lock}^{s_m, id_{txo}}(data_{lock}^{m}, id_{tx}, UTXO_{s_m}) = \varphi_{lock}^{s_b, id_{txo}}(data_{lock}^{m}, id_{tx}, UTXO_{s_b}) = \perp$, both calls will return false in line 117.

Otherwise, both calls will not return in line 117, and we can assume in the following lines that $\varphi_{lock}^{s_m, id_{txo}}(data_{lock}^{m}, id_{tx}, UTXO_{s_m}) = \varphi_{lock}^{s_b, id_{txo}}(data_{lock}^{m}, id_{tx}, UTXO_{s_m}) = \perp$. Both calls will return false in line 117 and we can assume in the following lines that $\varphi_{lock}^{s_m, id_{txo}}(data_{lock}^{m}, id_{tx}, UTXO_{s_m}) = \varphi_{lock}^{s_b, id_{txo}}(data_{lock}^{m}, id_{tx}, UTXO_{s_m}) = \perp$, both calls will return false in line 117.

Otherwise, both calls will not return in line 117, and we can assume in the following lines that $\varphi_{lock}^{s_m, id_{txo}}(data_{lock}^{m}, id_{tx}, UTXO_{s_m}) = \varphi_{lock}^{s_b, id_{txo}}(data_{lock}^{m}, id_{tx}, UTXO_{s_m}) = \perp$, both calls will return false in line 117.

5. Line 121 (false is returned if the spending condition is not satisfied): Since $s_m \rightarrow_{txo} s_b$ and $\varphi_{spend}^{s_m, id_{txo}} \in \mathcal{L}_{base}$, $\varphi_{spend}^{s_m, id_{txo}} = \varphi_{spend}^{s_b, id_{txo}}$. Since $s_m$ and $s_b$ are weakly equivalent, and because $\varphi_{spend}$ is agnostic to \( \perp / \text{pending/pending-locked} \) and \( \text{unspent/locked} \) deviations, $\varphi_{spend}^{s_m, id_{txo}}(data_{spend}^{m}, id_{tx}, UTXO_{s_m}) = \varphi_{spend}^{s_b, id_{txo}}(data_{spend}^{m}, id_{tx}, UTXO_{s_m})$. Hence, if $\varphi_{spend}^{s_m, id_{txo}}(data_{spend}^{m}, id_{tx}, UTXO_{s_m}) = \varphi_{spend}^{s_b, id_{txo}}(data_{spend}^{m}, id_{tx}, UTXO_{s_m}) = \perp$, both calls will return false in line 121.

Otherwise, both calls will not return in line 121, and we can assume in the following lines that $\varphi_{spend}^{s_m, id_{txo}}(data_{spend}^{m}, id_{tx}, UTXO_{s_m}) = \varphi_{spend}^{s_b, id_{txo}}(data_{spend}^{m}, id_{tx}, UTXO_{s_m}) = \perp$, both calls will return false in line 121.

6. Line 124: both calls will return true in line 124. Also, claim 4.3.11 applies to the SPEND-OUTPUT call in line 124 so consistency is preserved.

\[\square\]

**Lemma 4.3.13** ($s_m \rightarrow_{txo} s_b$ is preserved by $\mathcal{F}_{ledger}$’s VALIDATE-AND-APPLY-LOCKING-INPUT).

Let $s_m$ and $s_b$ be two $\mathcal{F}_{ledger}$ states, and suppose $s_m \rightarrow_{txo} s_b$, and let $tx_{b}$ and $tx_{m}$ be two identical transaction inputs and $id_{tx}$ be a transaction ID and define:

- $s'_{b}$ the state resulting from running VALIDATE-AND-APPLY-LOCKING-INPUT($tx_{b}, id_{tx}$) (line 130) functionality 3.2) with respect to $s_b$ and
- $s_m$ as the state resulting from running VALIDATE-AND-APPLY-LOCKING-INPUT($tx_{m}, id_{tx}$) with respect to $s_m$.

If $\varphi_{spend}^{s_m, id_{txo}} = \perp$ and $\varphi_{spend}^{s_{b}, id_{txo}} \in \mathcal{L}_{base}$ where $id_{txo}$ is the referenced transaction output ID, then either:

- both calls return true and $s'_{m} \rightarrow_{txo} s'_{b}$ or
- both calls return false.

**Proof.** Denote the amount spent as $amount_{id_{tx}}$ and the lock data as $data_{lock}^{m}$.

The proof is by exhaustive case analysis of all lines in which VALIDATE-AND-APPLY-LOCKING-INPUT can return. For line $l$ we show that either both calls return or both don’t and that when they do, they return the same boolean value and consistency is preserved.

1. Line 134 (false is returned if the referenced output is not in a lockable state): Since $s_m \rightarrow_{txo} s_b$, STATE($s_m, id$) $\neq \perp \iff$ STATE($s_b, id$) $\neq \perp$.

Hence, either both calls will return false in line 134 or neither will and we can assume in the following lines that STATE($s_m, id_{txo}$) $\neq \perp$ and STATE($s_b, id_{txo}$) $\neq \perp$. 

48
2. Line 139 (false is returned if locking is done with amount 0): \( tx_i \) and \( tx_m \) are identical and both spend \( amount_{txi} \).

Hence, if \( amount_{txi} \neq 0 \), both calls will return false in line 139. Otherwise, both calls will not return in line 139 and we can assume in the following lines that \( amount_{txi} = 0 \).

3. Line 143 (true is returned if the referenced output is pending-locked/locked): Since \( s_b \) and \( s_m \) are weakly-equivalent and \( \text{STATE}(s_m, id_{txo}) \neq \bot \) and \( \text{STATE}(s_b, id_{txo}) \neq \bot \), \( \text{STATE}(s_m, id) \in \{ \text{pending, pending-locked} \} \iff \text{STATE}(s_b, id) \in \{ \text{pending, pending-locked} \} \).

Hence, if \( \text{STATE}(s_m, id) \in \{ \text{pending, pending-locked} \} \) or \( \text{STATE}(s_b, id) \in \{ \text{pending, pending-locked} \} \), both calls will return true in line 143. Also, the states remain unchanged so consistency is preserved.

Otherwise, both calls will not return in line 143 and we can assume in the following lines that \( \text{STATE}(s_m, id) \notin \{ \text{pending, pending-locked} \} \) and \( \text{STATE}(s_b, id) \notin \{ \text{pending, pending-locked} \} \).

4. Line 147 (false is returned if the locking condition is not satisfied): Since \( s_m \xrightarrow{state}_{txo} s_b \), \( \varphi_{s_m, id_{txo}}^\text{lock} = \varphi_{s_b, id_{txo}}^\text{lock} \). Since \( s_m \) and \( s_b \) are weakly equivalent, and because \( \varphi_{s_b, id_{txo}}^\text{lock} \) is agnostic to \( \bot/\text{pending/pending-locked} \) and unspent/locked deviations, \( \varphi_{s_m, id_{txo}}^\text{lock}(data_{lock}, id_{tx}, UTXO_{s_b}) = \varphi_{s_m, id_{txo}}^\text{lock}(data_{lock}, id_{tx}, UTXO_{s_m}) \).

Hence, if \( \varphi_{s_m, id_{txo}}^\text{lock}(data_{lock}, id_{tx}, UTXO_{s_b}) = \varphi_{s_m, id_{txo}}^\text{lock}(data_{lock}, id_{tx}, UTXO_{s_m}) = \text{false} \), both calls will return false in line 147.

Otherwise, both calls will not return in line 147 and we can assume in the following lines that \( \varphi_{s_m, id_{txo}}^\text{lock}(data_{lock}, id_{tx}, UTXO_{s_b}) = \varphi_{s_m, id_{txo}}^\text{lock}(data_{lock}, id_{tx}, UTXO_{s_m}) = \text{true} \).

5. Line 154 both calls will return true in line 154.

By observation of lines 149 to 153 only the state of the referenced output is modified, and only pending \( \rightarrow \) pending-locked and unspent \( \rightarrow \) locked transitions are possible.

Since \( s_m \xrightarrow{state}_{txo} s_b \) and because \( \text{STATE}(s_m, id_{txo}) \neq \bot \) and \( \text{STATE}(s_b, id_{txo}) \neq \bot \), the only base-state/meta-state combinations possible are:

- such that the two states are the same \( \text{STATE}(s_m, id_{txo}) = \text{STATE}(s_b, id_{txo}) \),
- \( \text{STATE}(s_b, id_{txo}) = \text{pending} \) and \( \text{STATE}(s_m, id_{txo}) = \text{pending-locked} \),
- \( \text{STATE}(s_b, id_{txo}) = \text{unspent} \) and \( \text{STATE}(s_m, id_{txo}) = \text{locked} \).

In the first case, both states will be changed to the same state, in the second case \( \text{STATE}(s_b, id_{txo}) \) will be changed to pending-locked, and in the third case \( \text{STATE}(s_b, id_{txo}) \) will be changed to locked. Hence, \( s'_m \xrightarrow{state}_{txo} s'_b \).

\[ \square \]

**Claim 4.3.14** (\( s_m \xrightarrow{state}_{txo} s_b \) is preserved by \( F_{ledger} \)'s \text{VALIDATE-AND-APPLY-INPUTS}). Let \( s_m \) and \( s_b \) be two \( F_{ledger} \) states, and suppose \( s_m \xrightarrow{state}_{txo} s_b \), and let inputs\( s_b \) and inputs\( s_m \) be two identical lists of respectively identical transaction inputs and id\( s \) be a transaction ID and define:

- \( s'_b \) as the state resulting from running \text{VALIDATE-AND-APPLY-INPUTS}(\text{inputs}_s, id_{tx}) \) (line 83 functionality 3.2) with respect to \( s_b \) and
- \( s'_m \) as the state resulting from running \text{VALIDATE-AND-APPLY-INPUTS}(\text{inputs}_m, id_{tx}) \) with respect to \( s_m \).
If for every \( txi \in \text{inputs}_m \), \( \varphi^{\text{spend}}_{s_m, \text{id}_{txo}} \in L_{\text{base}} \) where \( \text{id}_{txo} \) is the transaction output ID referenced by \( txi \), then either:

- both calls return \( \text{true} \) and \( s'_m \stackrel{\text{state}}{\rightarrow} \text{txo} s'_b \) or
- both calls return \( \text{false} \).

**Proof.** The proof is by induction on the index of the input. Suppose that the induction hypothesis holds for (up to and including) the \( i \)th input.

If \( \varphi^{\text{spend}} = \bot \), then by lemma 4.3.13 either both \text{VALIDATE-AND-APPLY-LOCKING-INPUT} calls will return \( \text{true} \) and the consistency is preserved, or both return \( \text{false} \), in which case both \text{VALIDATE-AND-APPLY-INPUTS} calls will return \( \text{false} \) in line 89.

Otherwise, then by lemma 4.3.12 either both \text{VALIDATE-AND-APPLY-SPENDING-INPUT} calls will return \( \text{true} \) and the consistency is preserved, or both return \( \text{false} \), in which case both \text{VALIDATE-AND-APPLY-INPUTS} calls will return \( \text{false} \) in line 93.

Claim 4.3.15 (\( s_m \stackrel{\text{state}}{\rightarrow} \text{txo} s_b \) is preserved by \( F_{\text{ledger}} \)’s \text{APPLY-OUTPUTS}). Let \( s_m \) and \( s_b \) be two \( F_{\text{ledger}} \) states, and suppose \( s_m \stackrel{\text{state}}{\rightarrow} \text{txo} s_b \), and let \( \text{outputs}_b \) and \( \text{outputs}_m \) be two identical lists of respectively identical transaction outputs and \( \text{txo} \text{id}\text{s} \) be a list of IDs of the same length. Define:

- \( s'_b \) as the state resulting from running \( \text{APPLY-OUTPUTS}(\text{id}_{txo}, \text{txo} \text{id}\text{s}) \) (line 126, functionality 3.2) with respect to \( s_b \) and
- \( s'_m \) as the state resulting from running \( \text{APPLY-OUTPUTS}(\text{id}_{txo}, \text{txo} \text{id}\text{s}) \) with respect to \( s_m \).

Then it holds that \( s'_m \stackrel{\text{state}}{\rightarrow} \text{txo} s'_b \).

**Proof.** The same IDs are mapped to respectively identical outputs in state \text{pending}. \( \square \)

**Lemma 4.3.16** (\( s_m \stackrel{\text{state}}{\rightarrow} \text{txo} s_b \) is preserved by \( F_{\text{ledger}} \)’s \text{VALIDATE-AND-APPLY-TX}). Let \( s_m \) and \( s_b \) be two \( F_{\text{ledger}} \) states, and suppose \( s_m \stackrel{\text{state}}{\rightarrow} \text{txo} s_b \), and let \( \text{tx}_b \) and \( \text{tx}_m \) be two identical transactions and define:

- \( s'_b \) as the state resulting from running \( \text{VALIDATE-AND-APPLY-TX}(\text{tx}_b) \) (line 76, functionality 3.2) with respect to \( s_b \) and
- \( s'_m \) as the state resulting from running \( \text{VALIDATE-AND-APPLY-TX}(\text{tx}_m) \) with respect to \( s_m \).

If \( \varphi^{\text{spend}}_{s_m, \text{id}_{txo}} \in L_{\text{base}} \) for each output \( \text{txo} \) spent by \( \text{tx}_m \), then either:

- both calls return \( \text{true} \) and \( s'_m \stackrel{\text{state}}{\rightarrow} \text{txo} s'_b \) or
- both calls return \( \text{false} \).

**Proof.** The proof is by exhaustive case analysis of all lines in which \text{VALIDATE-AND-APPLY-TX} can return. For line \( l \) we show that either both calls return or both don’t and that when they do, they return the same boolean value and it holds that the resulting states are consistent.

1. Line 74 (\( \text{false} \) is returned if incoming coins < outgoing coins): Since \( \text{tx}_b \) and \( \text{tx}_m \) are identical:

   - \( \text{out}_b = \sum_{\text{txo} \in \text{outputs}_b} \text{amount}_{\text{txo}} = \sum_{\text{txo} \in \text{outputs}_m} \text{amount}_{\text{txo}} = \text{out}_m \)
   - \( \text{in}_b = \sum_{\text{txi} \in \text{inputs}_b} \text{amount}_{\text{txi}} = \sum_{\text{txi} \in \text{inputs}_m} \text{amount}_{\text{txi}} = \text{in}_m \)
Thus, $out_b > in_b \iff out_m > in_m$. Hence, if $out_b > in_b = out_m > in_m = true$, both calls will return false in line 74. Otherwise, both calls will not return in line 74 and we can assume in the following lines that $out_b \leq in_b = out_m \leq in_m = true$.

2. Line 78 (false is returned if any of the inputs is invalid): $tx_b$ and $tx_m$ are identical, so their inputs $inputs_b$ and $inputs_m$ are also respectively identical. Hence claim 4.3.14 applies, and therefore either the applications of both $inputs_b$ and $inputs_m$ will fail, or both will succeed and the consistency will be preserved.

3. Line 81: both calls will return true in line 81. By claim 4.3.15 $s_m' \xrightarrow{state \ txo} s_b'$ so consistency is preserved.

**Definition 4.3.17** (Applying $MsgQueue$ to $F_{ledger}$ state). Given an honest party (protocol 4.1) $MsgQueue q$, and a state of $F_{ledger}$ (functionality 3.2) $s$, $apply-queue(q,s)$ is defined in algorithm 4.2.

Denote $q_{[m]}$ the prefix of $q$ that ends with message $m_x$ and $q'_{[m_y]}$ the prefix of $q'$ that ends with message $m_y$. Let $s_{m_y} = apply-queue(q_{[m]},s_m)$.

**Claim 4.3.18** ($s_m = s_b$ is preserved by $apply-queue$). Let $s_m$ and $s_b$ be two identical $F_{ledger}$ states and $q$ a $MsgQueue$. Define:

- $s_b'$ as the state resulting from running $apply-queue(q,s_b)$ (definition 4.3.17) and
- $s_m'$ as the state resulting from running $apply-queue(q,s_m)$.

Then $s_m' = s_b'$.

**Proof.** Holds since $apply-queue$ is deterministic.

**Lemma 4.3.19** ($s_{m+q} \xrightarrow{state \ txo} s_b$ is preserved by $\Pi_{ledger}$’s ADD-LOCKING-INPUTS). Let $s_m$ and $s_b$ be two $F_{ledger}$ states, and $q$ be a state of $S_{P_b}$’s $MsgQueue$. Define:

- as $q'$ as $q$ after running ADD-LOCKING-INPUTS (protocol 4.1 line 100) with respect to $q$,
- $s_{m+q} = apply-queue(q,s_m)$,
- $s_{m+q'} = apply-queue(q',s_m)$.

If $s_{m+q} \xrightarrow{state \ txo} s_b$ and $s_{m+q'} \xrightarrow{state \ txo} s_b'$, then $s_{m+q'} \xrightarrow{state \ txo} s_b'$.

**Proof.** The proof is by induction on the index of the added locking input. Suppose that the induction hypothesis holds for (up to and including) the $i$th input.

Let $tx_i$ be the $(i+1)$th input, and denote its transaction $tx$ and the $Tx-Published$ message $m_{tx_i}$. Also denote as $txof$ the fragment output referenced by $tx_i$, and as $tx_om$ the meta-output that $txof$ is mapped to in $LockingFragmentTxos$. Suppose that $tx$ doesn’t already have an input that locks $txof$.

If $q$ is of the following form: $[m_1,\ldots,m_k,m_{tx_i},m_1,\ldots,m_n]$, then $q'$ is of the following form: $[m_1,\ldots,m_k,m_{txlock},m_1,\ldots,m_n]$, where $m_{txlock} = (Tx-Published,txlock)$ and $txlock$ is identical to $tx$ with the addition of the locking input $tx_{lock}$.

We show that each $m \in q'$ is applied successfully in turn when evaluating $apply-queue(q',s_m)$:
1. \([m_1, \ldots, m_k]\) — by claim 4.3.18, \(s_{m_k} = s'_{m_k}\), and since \(s_{m_k} \neq \bot\), then \(s'_{m_k} \neq \bot\).

2. \(m_{tx_{lock}}\) — since \(s_{m_k} = s'_{m_k}\), the two states are consistent. Recall that \(tx_{lock}\) differs from \(tx\) only in the addition of \(tx_{lock}\). We show that the application of \(tx_{lock}\) is successful. First note that by construction, \(tx_{lock}\) spends amount 0, hence the “incoming coins \(\geq\) outgoing coins” condition (line 14) still holds for \(tx_{lock}\).

Since \(tx\)’s existing inputs were not modified, the states after the application of these inputs \(s_{m_k+}\) and \(s'_{m_k+}\) are still identical.

We prove that the call \(VALIDATE-AND-APPLY-LOCKING-INPUT(tx_{lock})\) with respect to \(s'_{m_k+}\) will return true in line 154 or in line 143 and that consistency is preserved by showing that it cannot return false in any other line:

a) Line 134 (false is returned if the referenced output is not in a lockable state): Since \(LockingFragmentTxos\) maps \(tx_o\) to \(txo^m\), \(txo^m\) can’t be in state \(\bot\). So the call will not return in line 134.

b) Line 139 (false is returned if locking is done with amount 0): By construction, \(tx_{lock}\) spends amount 0, so the call will not return in line 139.

c) Line 147 (false is returned if the locking condition is not satisfied): Since:

- by condition 4 of definition 4.2.13, \(txo^m\) and \(txo^m\) share the locking condition, by construction, \(tx_{lock}\)’s locking data is identical to that of \(tx\),
- \(s_{m_k+}\) and \(s'_{m_k+}\) are identical,
- \(tx\) and \(tx_{lock}\) are inputs of the same transaction \(tx\).

Then: \(\varphi_{tx_{lock}}(data_{tx_{lock}}^f, id_{tx}, s_{m_k+}) = \varphi_{txo^m}(data_{txo^m}^f, id_{tx}, s_{m_k+}) = true\). So the call will not return in line 147.

Since \(tx\)’s outputs were not modified, \(s'_{m_{lock}}\) and \(s_{m_{lock}}\) differ only in \(txo^m\)’s state:

- \(\text{STATE}(s_{m_{lock}}, txo^m) = pending \Rightarrow \text{STATE}(s'_{m_{lock}}, txo^m) = pending-locked\)
- \(\text{STATE}(s_{m_{lock}}, txo^m) = unspent \Rightarrow \text{STATE}(s'_{m_{lock}}, txo^m) = locked\)
- \(\text{STATE}(s_{m_{lock}}, txo^m) \notin \{pending, unspent\} \Rightarrow \text{STATE}(s'_{m_{lock}}, txo^m) = \text{STATE}(s_{m_{lock}}, txo^m)\)

Hence all \(\text{STATE}_{txo}^f\) conditions still hold.

3. \([m_1, \ldots, m_n]\) — Suppose \(s'_{m_{j-1}}\) and \(s_{m_{j-1}}\) differ only in \(txo^m\)’s state as specified above — we prove the same applies for \(s'_{m_j}\) and \(s_{m_j}\), after applying \(m_j\).

a) If \(m_j = (Tx-Published, tx_j)\) and \(tx_j\)’s input \(tx_i\) references \(txo^m\), then it is implied that \(txo^m\) has been released in both states, and either:

- the two states are equal, in which case the application of \(tx_j\) will have the same effect, or
- \(\text{STATE}(s_{m_{j-1}}, txo^m) = unspent\) and \(\text{STATE}(s'_{m_{j-1}}, txo^m) = locked\). Locking would not affect the states, and spending is agnostic to whether the output that is being spent in state unspent or in state locked — the state would transition to spent either way.

b) Otherwise if \(m_j = (Txos-Released, tx_j)\) and \(tx_j = tx^m\) then \(\text{STATE}(s_{m_{j-1}}, txo^m) \in \{pending, pending-locked\}\) and \(\text{STATE}(s'_{m_{j-1}}, txo^m) = pending-locked:\n
- If \(\text{STATE}(s_{m_{j-1}}, txo^m) = pending\), then \(\text{STATE}(s_{m_j}, txo^m) = unspent, \text{STATE}(s'_{m_j}, txo^m) = locked\).
4 Generically Realizing a Meta-Ledger

- Otherwise \( \text{STATE}(s_{m_j-1},txo^m) = \text{pending-locked} \), then \( \text{STATE}(s_{m_j},txo^m) = \text{locked} \), \( \text{STATE}(s'_{m_j},txo^m) = \text{locked} \).

c) Otherwise \( m_j = (\text{Tx-Removed},tx_j) \), then \( tx_j \) wasn’t strongly-valid with respect to \( s_{m_j-1} \) (see line 165 of functionality 7.2) and one of the following must hold:

- \( \text{VALIDATE-AND-APPLY-TX}(tx_j) \) returned \text{false} with respect to \( s_{m_j-1} \). Since the conditions hold at \( (s'_{m_j-1},s_{m_j-1}) \), the only relevant difference between the two states is the state of \( txo^m \), which was locked. Hence it is impossible that \( \text{VALIDATE-AND-APPLY-TX}(tx_j) = \text{true} \) with respect to \( s_{m_j-1} \).

- \( \text{IS-LIVE}(tx_j) \) returned \text{false}. Since \( \text{IS-LIVE} \) is a function of the transaction (not of the ledger state), \( \text{IS-LIVE} \) will still return \text{false} in \( s_{m_j} \).

- \( tx_j \) spent an output in state \( \text{locked/spent} \) with respect to \( s_{m_j-1} \).
  - \( \text{STATE}(s_{m_j-1},txo^m) = \text{locked} \Rightarrow \text{STATE}(s'_{m_j-1},txo^m) = \text{locked} \)
  - \( \text{STATE}(s_{m_j-1},txo^m) = \text{spent} \Rightarrow \text{STATE}(s'_{m_j-1},txo^m) = \text{spent} \)

By the induction hypothesis, \( \text{STATE}(s_{m_j},txo) = \text{STATE}(s'_{m_j},txo) \), so the corresponding output is in the same state in \( \text{STATE}(s'_{m_j},txo) \).

Hence \( s'_m \) and \( s_m \) are identical up to locking of \( txo^m \) (\text{pending} \rightarrow \text{pending-locked} / \text{unspent} \rightarrow \text{locked} \), so \( s_{m+q} \leftarrow_{txo} s_b \).

\[ \]

**Lemma 4.3.20** (\( s_{m+q} \leftarrow_{txo} s_b \) is preserved by \( \Pi_{\text{ledger}} \)'s \text{SIMPLE-KEYSTONE-PUBLISHED}). Denote the states of \( S_{\text{ledger}}^{\text{base}} \) and \( F_{\text{ledger}}^{\text{meta}} \) as \( s_b \) and \( s_m \) respectively and \( S_{\text{ph}} \)'s \text{MsgQueue} as \( q \). Define \( s_{m+q} = \text{APPLY-QUEUE}(q,s_m) \) and suppose \( s_{m+q} \neq \perp \) and \( s_{m+q} \leftarrow_{txo} s_b \).

Let \( tx \) be a transaction and suppose \( tx \) is successfully-applied to \( s_b \), and denote the resulting state as \( s'_{b} \). If as a result, \( S_{\text{ph}} \) calls \text{SIMPLE-KEYSTONE-PUBLISHED} (\text{protocol \[4.1, line 115\] with respect to \( q \), then \( s_{m+q} \neq \perp \) and \( s_{m+q} \leftarrow_{txo} s'_{b} \), where \( q' \) is \( q \) after the \text{SIMPLE-KEYSTONE-PUBLISHED} call and \( s_{m+q} = \text{APPLY-QUEUE}(q',s_m) \).

**Proof.** If \( q \) is of the following form: \( [m_1,\ldots,m_n] \), then \( q' \) is of the following form: \( [m_1,\ldots,m_n,m_{tx}] \), where \( m_{tx} = (\text{Tx-Published},tx^m) \) and \( tx^m \) is the transaction output by \text{DECOMP}. By claim 4.3.18 \( s'_{m} = s_{m} = s_{m+q} \), and since \( s_{m+q} \neq \perp \), then \( s'_m \neq \perp \).

By observation of \text{TX-PUBLISHED-BY-\Pi_{\text{ledger}}} (\text{line 53} of protocol \[4.1\]) and of \text{SIMPLE-KEYSTONE-PUBLISHED}:

- \( tx^m \)'s ID and outputs are taken from the keystone \( tx \).

- \( tx^m \)'s inputs are taken from:
  - the keystone \( tx \) if it is non-simple and the splitter is not present in \text{MsgQueue} or
  - the transaction output by the \text{DECOMP} call in line 25 otherwise.

Due to the lemma condition, \( s_{m+q} \leftarrow_{txo} s_b \). If \( tx^m \)'s inputs are taken from the keystone \( tx \), \( tx^m \) is identical to \( tx \), and we can apply lemma 4.3.16 to conclude that \( tx^m \) can be successfully-applied to \( s_{m+q} \), and that the consistency is preserved.

Otherwise, \( tx^m \)'s inputs are taken from the the transaction output by the \text{DECOMP} call, and we can apply condition 7 in definition 4.2.13 to conclude that \( tx^m \) can be successfully-applied to \( s_{m+q} \) and that the consistency is preserved. \( \square \)
Lemma 4.3.21 \((s_{m+q} \xrightarrow{\text{state}_{\xrightarrow{\text{txo}}}} s_b)\) is preserved by \(\Pi_{\text{ledger}}\)’s NON-SIMPLE-KEYSTONE-PUBLISHED.

Denote the states of \(S_{\text{phase}_{\text{ledger}}}^m\) and \(F_{\text{meta}_{\text{ledger}}}^m\) as \(s_b\) and \(s_m\) respectively and \(S_{\text{h}}\)’s MsgQueue as \(q\).

Define \(s_{m+q} = \text{APPLY-QUEUE}(q, s_m)\) and suppose \(s_{m+q} \neq \bot\) and \(s_{m+q} \xrightarrow{\text{state}_{\xrightarrow{\text{txo}}}} s_b\).

Let \(tx\) be a transaction and suppose \(tx\) is successfully-applied to \(S_{\text{h}}\) and denote the resulting state as \(s_b\). If as a result, \(S_{\text{h}}\) calls NON-SIMPLE-KEYSTONE-PUBLISHED (protocol 4.4, line 120) with respect to \(q\), then \(s_{m+q} \neq \bot\) and \(s_{m+q} \xrightarrow{\text{state}_{\xrightarrow{\text{txo}}}} s_b\), where \(q'\) is \(q\) after the NON-SIMPLE-KEYSTONE-PUBLISHED call and \(s_{m+q} = \text{APPLY-QUEUE}(q', s_m)\).

**Proof.** If \(q\) is of the following form: \([m_1, \ldots, m_k, m_{txa}, m_1, \ldots, m_n]\), then \(q'\) is of the following form: \([m_1, \ldots, m_k, m_{tx^m}, m_{tx^*, m_1, \ldots, m_n, m_{txd}}]\), where:

- \(m_{txa} = (\text{Tx-Published}, tx_a)\) and \(tx_a\) is the MsgQueue-version of splitter,
- \(m_{tx^m} = (\text{Tx-Published}, tx^m)\) and \(tx^m\) is initialized in line 126 with \(tx_a\)’s inputs and the outputs of the meta-transaction returned from the call to DECOMP in line 25,
- \(m_{tx^*} = (\text{Tx-Published}, tx^*)\) and \(tx^*\) initialized to \(tx_a\) in line 128,
- \(m_{txd} = (\text{Tx-Published}, tx_d)\) and \(tx_d\) initialized in line 129 with \(tx_a\)’s single input and no outputs.

Note that \(s_{m_a} = s_{m+q}\) and \(s_{m_d}' = \text{APPLY-QUEUE}(q'[m_d], s_m)\).

We show that each \(m \in q\) is applied successfully in turn when evaluating \(\text{APPLY-QUEUE}(q', s_m)\):

1. \([m_1, \ldots, m_k]\) — by claim 4.3.18 \(s_{m_k} = s_{m+q}\), and since \(s_{m+q} \neq \bot\), then \(s_{m_k}' \neq \bot\).
2. \(m_{tx^m}, m_{tx^*}\) — Following is an exhaustive list of lines in which validate-and-apply-tx can return \emph{false} — for each we show simultaneously for \(tx^m\) and for \(tx^*\) that the calls don’t return there.
   a) Line 74 (\emph{false} is returned if incoming coins < outgoing coins):
      By condition 3 in definition 4.2.13 \(tx^m\)’s output amounts are identical to those of \(tx_a\); the same total amount \((\text{amount}_k)\) is set to \(tx^m\) inputs in line 142 of protocol 4.4.
      The same amount reduced from \(tx_a\)’s inputs \((\text{amount}_k)\) in line 135 is reduced from its outputs in line 141. Since \(tx_a\) is valid in this regard, then so is \(tx^*\).
      Hence the two calls will not return (\emph{false}) in line 74.
   b) Line 78 (\emph{false} is returned if any of the inputs is invalid):
      \(tx^m\)’s inputs are initialized as a fresh copy of \(tx_a\) inputs, \(tx^*\) is initialized as a fresh copy of \(tx_a\). \(tx^m\) also receives \(tx_a\)’s ID.
      Following is an exhaustive list of lines in which validate-and-apply-spending-input can return \emph{false} — for each we show simultaneously for corresponding spending inputs of \(tx^m\) and \(tx^*\) that the calls don’t return there.
      - Line 103 (\emph{false} is returned if the referenced output is not in a spendable state):
         Since the referenced outputs are shared by \(tx^m\) and \(tx^*\), then because \(s_{m_k} = s_{m_k}\) and \(tx_a\)’s inputs are valid in this regard, then so are \(tx^m\)’s inputs.
         Since the referenced outputs are shared by \(tx^*\) and \(tx^m\), then because the outputs have already been spent by \(tx^m\), \(tx^*\)’s inputs are valid in this regard.
         Hence the two calls will not return (\emph{false}) in line 103.
• Line 108 (false is returned if the referenced output has insufficient remaining coins): The (vector) addition $tx^m$’s input amounts and $tx_s$’s input amounts sums up to $tx_s$’s input amounts (see lines 141 and 142). Then because $s_{m_k} = s_{m_k}$ and $tx_s$’s inputs are valid in this regard, then so are $tx^m$’s and $tx_s$’s inputs.

Hence the two calls will not return (false) in line 108

• Line 113 (true is returned if the referenced output is spent): Since the referenced outputs are shared by $tx^m$ and $tx_s$, then because $s'_{m_k} = s_{m_k}$ a VALIDATE-AND-APPLY-SPENDING-INPUT call for an $tx^m$ input will return true in line 113 if the call for the corresponding $tx_s$ input will.

Hence the $tx^m$ call will return true in line 113 if the call for the corresponding $tx_s$ input did.

Since the referenced outputs are shared by $tx_s$- and $tx^m$, then because the outputs have already been spent by $tx^m$, all VALIDATE-AND-APPLY-SPENDING-INPUT calls for $tx_s$’s inputs will return true in line 113

• Line 117 (false is returned if the locking condition is not satisfied): Since $tx^m$’s inputs are copied from $tx_s$ and only the amounts are modified, then because $s_{m_k}$ and $tx_s$’s inputs are valid in this regard, then so are $tx^m$’s inputs.

Hence the call will not return (false) in line 117

• Line 121 (false is returned if the spending condition is not satisfied): Since $tx^m$’s inputs are copied from $tx_s$ and only the amounts are modified, then because $s_{m_k} = s_{m_k}$ a VALIDATE-AND-APPLY-SPENDING-INPUT call for an $tx^m$ input will return in line 121 if the call for the corresponding $tx_s$ input will.

Hence the call will not return (false) in line 121

Following is an exhaustive list of lines in which VALIDATE-AND-APPLY-LOCKING-INPUT can return false — for each we show simultaneously for corresponding locking inputs of $tx^m$ and $tx_s$- that the calls don’t return there.

i. Line 134 (false is returned if the referenced output is not in a lockable state): Since the referenced outputs are shared by $tx^m$, $tx_s$- and $tx_s$, then because $s'_{m_k} = s_{m_k}$ and $tx_s$’s inputs are valid in this regard, then so are $tx^m$’s inputs.

Hence the two calls will not return (false) in line 134

ii. Line 139 (false is returned if locking is done with amount 0): Since the $tx^m$ and $tx_s$- are instantiated with $tx_s$’s inputs, then because $tx_s$’s inputs are valid in this regard, then so are $tx^m$’s and $tx_s$’s inputs.

Hence the two calls will not return (false) in line 139

iii. Line 143 (true is returned if the referenced output is pending-locked/locked): Since the referenced outputs are shared by $tx^m$ and $tx_s$, then because $s_{m_k} = s_{m_k}$ a VALIDATE-AND-APPLY-LOCKING-INPUT call for an $tx^m$ input will return true in line 143 if the call for the corresponding $tx_s$ input will.

Hence the $tx^m$ call will return true in line 143 if the call for the corresponding $tx_s$ input did.

Since the referenced outputs are shared by $tx_s$- and $tx^m$, then because the outputs have already been locked by $tx^m$, all VALIDATE-AND-APPLY-LOCKING-INPUT calls for $tx_s$’s inputs will return true in line 143

iv. Line 147 (false is returned if the locking condition is not satisfied): Since $tx^m$’s inputs are copied from $tx_s$ and only the amounts are modified, then because $s_{m_k} = s_{m_k}$ and $tx_s$’s inputs are valid in this regard, then so are $tx^m$’s inputs.
Hence the call will not return (false) in line 147.

Note that the outputs spent by \( tx_s \) were spent in the same manner by \( tx^m \) and \( tx_{s_r} \). Also note that \( tx_{s_r} \) shares \( tx_s \)’s outputs and output IDs with one difference — the output spent by the \( tx_k \) and by \( tx_{\text{dummy}} \) — in \( s_{m_{txs}}’s \) amount is \( \text{amount}_k \) coins short. Hence the following conditions hold for \( s’ = s_{m_{txs}}’ \) and \( s = s_{m_{txs}} \):

a) \( s’ \neq \bot \)

b) For all output ID \( id \in s’ \):

\[
\begin{align*}
\text{STATE}(s’, id) &= \text{STATE}(s, id) \quad \text{and} \\
\varphi^{\text{lock}}_{s’, id} &= \varphi^{\text{lock}}_{s, id} \quad \text{and} \\
\varphi^{\text{spend}}_{s’, id} &= \varphi^{\text{spend}}_{s, id} \quad \text{and} \\
\text{If } id \text{ is the ID of the output of the splitter } tx_s \text{ that is spent by the keystone } tx_k, \\
\text{REMAINING}(s’, id) &= \text{REMAINING}(s, id) - \text{amount}_k
\end{align*}
\]

where \( \text{amount}_k \) is the total output amount of the keystone. Otherwise,

\[
\text{REMAINING}(s’, id) = \text{REMAINING}(s, id)
\]

c) For all output ID \( id \in s’ \setminus s, s’[id] \) is the state of an output of \( tx^m \).

3. \([m_j, \ldots, m_n]\). We prove by induction that the above conditions hold for all \((s’, s) \in ([s’_{m_{txs}}, s_{m_{txs}}), (s’_{m_1}, s_{m_1}), \ldots, (s’_{m_n}, s_{m_n})]\). For the base of the induction, we’ve already shown the conditions hold for \((s, s) = (s’_{m_{txs}}, s_{m_{txs}})\). For the induction step, note that \( s \neq \bot \) for all \( s \in [s_{m_{txs}}, s_{m_1}; \ldots, s_{m_n}] \).

Since the keystone \( tx_k \) has not yet been applied in \( s_{m_n} \) (nor has its output been released), the transactions published in \([m_1, \ldots, m_n]\) do not reference its outputs. This implies that \( tx^m \)’s outputs are not referenced by the transactions published in \([m_1, \ldots, m_n]\). Hence they remain in \textit{pending} for all \( s’ \in [s’_{m_1}, \ldots, s’_{m_n}] \). \textit{Pending} outputs are effectively treated as missing by conditions, hence the existence of \( tx^m \)’s output in \( s’_{m_1}, \ldots, s_{m_n} \) does not affect the evaluation of any condition.

Suppose the conditions hold at \((s’_{m_j-1}, s_{m_j-1})\) — we prove they hold for \((s’_{m_j}, s_{m_j})\), after applying \( m_j \).

If \( m_j = (Tx_{-\text{Published}}, tx_j) \), the only way for \textit{validate-and-apply-tx} to fail with respect to \( s’_{m_j-1} \) is if an input of \( tx_j \) attempts to spend from the reduced-remaining-amount output more than it has left. However, in this case the remaining amount in \( s_{m_j} \) (after \( tx_j \) was applied) must be less than the input amount of the keystone \( tx_k \), hence \( tx_k \) could not have been applied successfully (contradicting our assumption that the application of \( tx_k \) to \( s_k \) is successful).

Otherwise suppose \( m_j = (Tx_{-\text{Released}}, tx_j) \). \( tx_j \neq tx^m \) since \( tx_k \) has not yet been published in \( s_{m_j-1} \), hence the \( tx_j \)’s outputs are in the same states in \( s_{m_j-1} \) and in \( s_{m_j-1} \).

Otherwise if \( m_j = (Tx_{-\text{Remove}}, tx_j) \), then \( tx_j \) wasn’t strongly-valid with respect to \( s_{m_j-1} \) (see line 163 of functionality 3.2) and one of the following must hold:

- \textit{validate-and-apply-tx}(\( tx_j \)) returned \textit{false} with respect to \( s_{m_j-1} \). Since the conditions hold at \((s’_{m_j-1}, s_{m_j-1})\), the only relevant difference between the two states is the lower amount of the splitter output. Hence it is impossible that \textit{validate-and-apply-tx}(\( tx_j \)) = \textit{true} with respect to \( s’_{m_j-1} \).
4 Generically Realizing a Meta-Ledger

- **IS-LIVE**\((tx_j)\) returned **false**. Since **IS-LIVE** is a function of the transaction (not of the ledger state), **IS-LIVE** will still return **false** in \(s'_{m_j}\).
- \(tx_j\) spent an output in state **locked/spent** with respect to \(s_{m_j-1}\). By the induction hypothesis, \(\text{STATE}(s_{m_j}, txo) = \text{STATE}(s'_{m_j}, txo)\), so the corresponding output is in the same state in \(\text{STATE}(s'_{m_j}, txo)\).

Hence the conditions holds for \((s'_{m_n}, s_{m_n})\), so \(s'_{m_n}\) identical \(s_{m_n}\) but for the **amount**-reduced amount of the output spent by \(tx_k\), and the “extra” IDs added by \(tx^m\).

4. \(m_{tx\text{dummy}} - tx\text{dummy}\) is constructed in line 129. To show that **VALIDATE-AND-APPLY-TX**\((tx_s)\) (line 70) with respect to \(s'_{m_n}\) returns **true** in line 81, we show that it doesn’t return (**false**) in lines 74 and 78.

a) Line 74 (**false**) is returned if incoming coins < outgoing coins:

\(tx\text{dummy}\) has no outputs, so the “incoming coins < outgoing coins” condition doesn’t hold, hence \(tx\text{dummy}\) is valid in this regard.

b) Line 78 (**false**) is returned if any of the inputs is invalid:

Recall that \(s_{m_n} = s_{m_n+q}\) and therefore \(s_{m_n} \xrightarrow{ txo } s_\text{b} \). Hence \(s'_{m_n} \xrightarrow{ txo } s_\text{b} \) but for the **amount**-reduced amount of \(txo\), the output spent by \(tx_k\), and the “extra” IDs added by \(tx^m\). Also recall that \(tx_k\) is valid with respect to \(s_\text{b}\).

\(tx\text{dummy}\) is initialized with \(tx_k\)'s only input and no outputs.

The amount of \(txo\), the output spent by \(tx\text{dummy}\), is later reduced and is lower in \(s'_{m_n}\) than that of \(txo\), the output spent by \(tx_k\), in \(s_{m_n}\). However, the amount of \(tx\text{dummy}'\)s input is also reduced by **amountk**.

\(txo\) and \(txo\) share the same conditions since they’re base, so they are satisfied with \(tx\text{dummy}\) and \(tx_k\) data respectively.

Hence, \(s_{m_n+q} \neq \perp\). Since \(tx^m\) and \(tx_k\) share output IDs and amounts \(s'_{m_n} \xrightarrow{ txo } s_\text{b}'\).

Finally, since \(tx^m\) was output by **DECOMP**, \(s_{m_n+q} \xrightarrow{ txo } s_\text{b}'\).

\(\Box\)

Define the following simulation events. Each event is specified by a line number in \(S/S^\text{base}_{metaledger}/S^\text{F}_h/F^\text{metaledger}\). The event triggers after the line was executed. For each event we specify an ordered list of code segments that we later prove is run sequentially prior to the event:

1. Line 14 of \(S^\text{P}_h\)'s **TX-PUBLISHED-BY-**\(F\)\_ledger**.

   **Code segments:**
   - \(S^\text{base}_{ledger}\)'s **PROCESS-TX** (line 20): up to line 31 — \(tx\) was applied to the state of \(S^\text{base}_{ledger}\).
   - \(S^\text{P}_h\)'s **TX-PUBLISHED-BY-**\(F\)\_ledger** (line 23): up to line 44 — \(tx\) was applied to \(S^\text{P}_h\)'s **MsgQueue**.

2. Line 55 of \(S^\text{P}_h\)'s **TXOS-RELEASED-BY-**\(F\)\_ledger**.

   **Code segments:**
   - \(S^\text{base}_{ledger}\)'s **RELEASE-TXOS** (line 30): up to line 39 or line 42 — \(tx\)'s outputs were released in \(S^\text{base}_{ledger}\).
   - \(S^\text{P}_h\)'s **TXOS-RELEASED-BY-**\(F\)\_ledger** (line 52): up to line 55 — a **Txos-Released** for \(tx\) message was added to \(S^\text{P}_h\)'s **MsgQueue**.
3. Line 34 of $S$’s CHECK-$S^\text{F-base}_{\text{ledger}}$-FRAGMENT-VALIDITY.

Code segments:
- $S$’s CHECK-$S^\text{F-base}_{\text{ledger}}$-FRAGMENT-VALIDITY (line 26): line 34 — $S$ has added a Tx-Removed message for $tx$ to $S^\text{P}_h$’s MsgQueue.

4. Line 17 of $S$’s PROCESS-$S^\text{P}_h$-MESSAGE.

Code segments:
- $S^\text{P}_h$’s FLUSH-MSG-QUEUE (line 54): line 96 — (Tx-Published, $tx$) was popped from $S^\text{P}_h$’s MsgQueue,
- $S$’s PROCESS-$S^\text{P}_h$-MESSAGE (line 14): up to line 17 — $S$ called $F^\text{meta}_{\text{ledger}}$’s PROCESS-TX with $tx$,
- $F^\text{meta}_{\text{ledger}}$’s PROCESS-TX (line 20).

5. Line 19 of $S$’s PROCESS-$S^\text{P}_h$-MESSAGE.

Code segments:
- $S^\text{P}_h$’s FLUSH-MSG-QUEUE (line 54): line 96 — (Txos-Released, $tx$) was popped from $S^\text{P}_h$’s MsgQueue,
- $S$’s PROCESS-$S^\text{P}_h$-MESSAGE (line 14): up to line 19 — $S$ called $F^\text{meta}_{\text{ledger}}$’s RELEASE-TXOS with $tx$,
- $F^\text{meta}_{\text{ledger}}$’s RELEASE-TXOS (line 36).

6. Line 24 of $S$’s $S^\text{P}_h$-TX-REMOVED-POPPED.

Code segments:
- $S^\text{P}_h$’s FLUSH-MSG-QUEUE (line 54): line 96 — (Tx-Removed, $tx$) was popped from $S^\text{P}_h$’s MsgQueue,
- $S$’s $S^\text{P}_h$-TX-REMOVED-POPPED (line 22): up to line 24 — $S$ called $F^\text{meta}_{\text{ledger}}$’s REMOVE-TX with $tx$,
- $F^\text{meta}_{\text{ledger}}$’s REMOVE-TX (line 45).

7. Line 44 of $F^\text{meta}_{\text{ledger}}$’s TX-PUBLISHED-BY-$F_{\text{ledger}}$.

Code segments:
- $F^\text{meta}_{\text{ledger}}$’s TICK (line 57) — PROCESS-TX was called with $tx$,
- $F^\text{meta}_{\text{ledger}}$’s PROCESS-TX (line 20): up to line 31 — $tx$ was applied to the state of $F^\text{meta}_{\text{ledger}}$.

8. Line 55 of $F^\text{meta}_{\text{ledger}}$’s TXOS-RELEASED-BY-$F_{\text{ledger}}$.

Code segments:
- $F^\text{meta}_{\text{ledger}}$’s TICK (line 65) — RELEASE-TXOS was called with $tx$,
- $F^\text{meta}_{\text{ledger}}$’s RELEASE-TXOS (line 36): up to line 39 or line 42 — $tx$’s outputs were released in $F^\text{meta}_{\text{ledger}}$.

Claim 4.3.22 (Consecutive Execution of Event Code Segments). For each event of event types 1 to 7, it holds for the code segments of the event that:

- they are run consecutively prior to the event, and
- they are disjoint from code segments of other events.
4 Generically Realizing a Meta-Ledger

Proof. By observation of the code of functionality 3.2, protocol 4.1 and algorithm 4.1, the code segment are disjoint, but this is true because each event handles a different process during simulation:

- Event types 1 and 2 handle state changes in $S_{F_{\text{base}}}$ ledger and the $S_{P_h}$ changes they cause, processing transactions and releasing of outputs, respectively.
- Event type 3 deals with the addition of Tx-Removed messages to $S_{P_h}$’s $MsgQueue$ when a fragment has become not-strongly-valid.
- Event types 4 to 6 handle with popping of different messages from $S_{P_h}$’s $MsgQueue$ (Tx-Published, Txos-Released, Tx-Removed), and their effect on the state of $F_{\text{meta}}$ ledger.
- Event types 7 and 8 deal with processes internally-triggered in $F_{\text{meta}}$ ledger, processing of transactions and releasing of outputs respectively.

We prove by analysis of each event type that the code segments happen consecutively:

- Event types 1 and 2
  When $S_{F_{\text{base}}}$ ledger outputs a message addressed to honest parties, $S$ pauses the $S_{F_{\text{base}}}$ simulation and immediately runs the $S_{P_h}$ code that handles the message:
    - When a Tx-Published message is output from $S_{F_{\text{base}}}$ ledger (line 31), $S_{P_h}$’s tx-published-by-$F_{\text{ledger}}$ (line 23) is called (where $S_{P_h}$ applies $tx$ to $MsgQueue$),
    - When a Txos-Released message is output from $S_{F_{\text{base}}}$ ledger (line 39 or line 42), $S_{P_h}$’s txos-released-by-$F_{\text{ledger}}$ (line 52) is called (where $S_{P_h}$ adds a Txos-Released message for $tx$ to $MsgQueue$).

Hence the respective second code segments of event types 1 and 2 immediately follow the respective first code segments during simulation.

By observation of protocol 4.1:
  - Tx-Published messages are only added to $MsgQueue$ in TX-PUBLISHED-BY-$F_{\text{ledger}}$.
    Specifically, in simple-keystone-published (line 115) and non-simple-keystone-published (line 120).
  - Txos-Released messages are only added to $MsgQueue$ in TXOS-RELEASED-BY-$F_{\text{ledger}}$.

Hence the respective second code segments of event types 1 and 2 only follow the respective first code segments during simulation.

- Event type 3
  CHECK-$S_{F_{\text{base}}}$-FRAGMENT-VALIDITY is called after $S_{P_h}$ runs either:
    - line 44 of TX-PUBLISHED-BY-$F_{\text{ledger}}$ or
    - line 55 of TXOS-RELEASED-BY-$F_{\text{ledger}}$,
  that is, immediately after event types 1 and 2.

- Event types 4 to 6
  When $S_{P_h}$ outputs a message or pops a (Tx-Removed, $tx$) message from $MsgQueue$ in line 94 of FLUSH-MSG-QUEUE, $S$ pauses the $S_{P_h}$ simulation and immediately runs the $S$ code that handles the message:
    - when $S_{P_h}$ outputs a Tx-Published message (line 96), $F_{\text{meta}}$’s PROCESS-TX is called (line 17),
    - when $S_{P_h}$ outputs a Txos-Released message (line 96), $F_{\text{meta}}$’s RELEASE-TXOS is called (line 19),
when \( S_p \) outputs a \text{Tx-Removed} message (line 94), \( F_{\text{ledger}}^{\text{meta}} \)'s \text{REMOVE-TX} is called (line 24).

Hence the respective second code segments of event types 1 to 6 immediately follow the respective first code segments during simulation.

By observation of algorithm 4.1:

- Line 17 is the only place \( S \) calls \( F_{\text{ledger}}^{\text{meta}} \)'s \text{process-tx}.
- Line 19 is the only place \( S \) calls \( F_{\text{ledger}}^{\text{meta}} \)'s \text{release-txos}.
- Line 24 is the only place \( S \) calls \( F_{\text{ledger}}^{\text{meta}} \)'s \text{remove-tx}.

Hence the respective last two code segments of event types 1 to 8 only follow the respective first code segments during simulation.

- Event types 7 and 8: The code segments are internal to \( F_{\text{meta}} \), hence they happen consecutively.

We say a state change is attributed to an event \( e \) if it happens due to the execution of \( e \)'s code segments.

\textbf{Claim 4.3.23 (State Change Attribution Coverage).} All changes to the states of \( F_{\text{base}} \), \( S_p \) and \( F_{\text{meta}} \) during simulation can be attributed to an event of event types 1 to 6.

\textit{Proof.} By observation of functionality 3.2, the state of \( F_{\text{ledger}} \) can only be modified when processing a transaction in \text{process-tx} (line 20) and when releasing the outputs of a transaction in \text{release-txos} (line 36). The two procedures can be called adversarially, or otherwise internally from \text{TICK} (line 51). The four combinations are covered in event types 1, 2, 7 and 8.

By observation of protocol 4.1, \( \Pi_{\text{ledger}} \)'s \text{MsgQueue} contains \text{Tx-Published} and \text{Txos-Released} messages — their additions to \text{MsgQueue} are covered in event types 1 to 8 respectively and the poppings in event types 4 and 5 respectively.

By observation of algorithm 4.1, \( S \) also adds \text{Tx-Removed} messages to \( S_p \)'s \text{MsgQueue}. Their additions are covered in event type 3 and the poppings in event type 6.

\textbf{Lemma 4.3.24 (s\textsubscript{m+q} \xrightarrow{\text{state} \ x_{to}} \ s_b \text{ is preserved by event types 1 to 6).} Denote the states of \( F_{\text{base}} \) and \( F_{\text{meta}} \) as \( s_b \) and \( s_m \) respectively and \( S_p \)'s \text{MsgQueue} as \( q \). Define \( s_{m+q} = \text{apply-queue}(q,s_m) \) and suppose \( s_{m+q} \neq \perp \) and \( s_{m+q} \xrightarrow{\text{state} \ x_{to}} s_b \).

Let \( e \) be an event of event types 1 to 6 and denote \( s_b, s_m \) and \( q \) after \( e \) as \( s'_b, s'_m \) and \( q' \), and define \( s_{m'+q'} = \text{apply-queue}(q',s_m) \). Then \( s_{m'+q'} \neq \perp \) and \( s_{m'+q'} \xrightarrow{\text{state} \ x_{to}} s'_b \).

\textit{Proof.} We do a case analysis by the type of the event.

- **Event type 1** (\text{Tx-Published}, \( tx \)) was published by \( S_{\text{base}} \), hence \( tx \) was applied to \( s_b \). \( S_{\text{base}} \)'s \text{TX-PUBLISHED-BY-Fledger} (line 23) of protocol 4.1 is called with \( tx \).
  
  If \text{SIMPLE-KEYSTONE-PUBLISHED} is called in line 39 then lemma 4.3.20 applies. Otherwise, \text{NON-SIMPLE-KEYSTONE-PUBLISHED} is called in line 42 then lemma 4.3.21 applies. Regardless, \( s_{m'+q'} \neq \perp \) and \( s_{m'+q'} \xrightarrow{\text{state} \ x_{to}} s'_b \).

- **Event type 2** First note that \( S_p \) pushes (\text{Txos-Released}, \( tx_m \)) to \text{MsgQueue}, where \( tx_m \) is the meta version of \( tx \) — the one that shares it’s outputs.
Since $S_{\text{base}}$ releases $tx$’s outputs they must be pending or pending-locked in $s_b$. Due to consistency, if the outputs are pending-locked in $s_b$, they must be pending-locked in $s_m$ too, hence releasing will succeed $s_m$ and will have the same effect as in $s_b$. If the outputs are pending in $s_b$, they can be either pending or pending-locked in $s_m$, so releasing will succeed in $s_m$. The outputs will be unspent in $s_b'$ and either unspent or locked in $s_m'$ so consistency is maintained.

- **Event type 3** $S$ pushes (Tx-Removed, $tx$) when $S_{p_h}$ reaches line 73 of TICK-BY-$F_{\text{ledger}}$ (line 68).

This implies a transaction $tx_f$ in COMP($tx$) was sent to $S_{\text{base}}$ but was not published within $\delta_{\text{base}}$ ticks. By condition 3 of definition 4.2.18 $tx_f$ cannot appear after the splitter (since this would mean the environment found a blocking sequence for COMP($tx$) that includes the splitter). Thus, by condition 2 $tx$ is removable in $s_m$.

- **Event types 4 to 6** ($F_{\text{meta}}$ ledger calls): By the lemma hypothesis, $s_{m+q} \neq \bot$, hence:

$$
\bot \neq s_{m+q} = \text{APPLY-QUEUE}(q, s_m) \quad // \text{msg} \leftarrow q.\text{POP}()
$$

$$
= \text{APPLY-QUEUE}(q', \text{APPLY-MESSAGE}(msg, s_m))
$$

$$
= \text{APPLY-QUEUE}(q', s_m')
$$

The APPLY-MESSAGE call succeeds because $s_{m+q} \neq \bot$. So $s_{m'+q'} \neq \bot$. Also, since this is a Pop event, nothing was changed in the base-ledger, hence:

$$
s_{m'+q'} = s_{m+q} \xrightarrow{\text{state}}_{txo} s_b = s'_b
$$

\[\square\]

**Lemma 4.3.25** ($\text{state}^\text{to}_{txo}$ is always preserved). Denote $s_b$ and $s_m$ the states of $S_{\text{base}}$ and $F_{\text{ledger}}$ respectively and $q$ $S_{p_h}$’s $\text{MsgQueue}$. Then it holds that:

- no event of event types 7 and 8 ever occurs, and

- after every event of event types 1 to 6 it holds that $s_{m+q} \neq \bot$ and $s_{m+q} \xrightarrow{\text{state}}_{txo} s_b$, where $s_{m+q} = \text{APPLY-QUEUE}(q, s_m)$.

**Proof.** The proof is by induction on the index of the event (event types 1 to 6). Suppose the induction hypothesis holds for (up to and including) the $i$th event.

By claim 4.3.22 the code segments leading to an event are executed consecutively and are not interrupted.

By claim 4.3.23 the code segments cover all changes to the state, hence no state changes can be made after one event and before the first segment of the following event.

To show that no event of event types 7 and 8 ever occurs, suppose in contradiction that the event is of one of these event types. Hence the transaction $tx$ that was processed or whose outputs were released, was live (IS-LIVE$^{\text{meta}}(tx) = \text{true}$) and has been in $F_{\text{ledger}}$’s mempool more than $\delta_{\text{meta}}$ ticks. Thus it was added to $S_{\text{base}}$’s mempool more than $\delta_{\text{meta}}$ ticks ago. By claim 4.3.10 within $(2 \cdot i \cdot \delta_{\text{base}}) \leq \delta_{\text{meta}}$, $S$ has called either:

- $F_{\text{ledger}}$’s PROCESS-TX and then RELEASE-TXOS with $tx'$, where $tx'$ is identical to $tx$ up to input amount changes or

- $F_{\text{ledger}}$’s REMOVE-TX with $tx$. 

61
4 Generically Realizing a Meta-Ledger

If the first case holds, due to the induction hypothesis, \( tx' \) was successfully applied to \( F_{\text{meta}} \)'s state and its outputs were released. Since \( F_{\text{ledger}} \)'s \textsc{process-tx} strips input amounts and locking inputs, \( tx \) is successfully removed from its mempool.

If the second case holds, \textsc{remove-tx}(\( tx \)) was called and by the induction hypothesis was successful, thereby removing \( tx \) from mempool.

If the event is of event type 8 then the transaction \( tx \) whose outputs were released has been in

If the event is of event types 1 to 6 then by lemma 4.3.24 \( s_{m+q} \neq \bot \) and \( s_{m+q} \xrightarrow{\text{txo}} s_b \) after the \((i + 1)\)th event.

\[ \square \]

**Corollary 4.3.26** (\( S \) calls to \( F_{\text{meta}} \)'s adversarial APIs always succeed). Every \( S \) call to \( F_{\text{meta}} \)'s \textsc{process-tx} (line 20 of functionality 3.2 (part 2)) and \textsc{release-txos} (line 36) succeeds and the two procedures are never called internally from \textsc{tick} (line 51).

\[ \square \]

**Proof.** Every \textsc{process-tx} call is in the code segments of event type 4 and every \textsc{release-txos} call is in the code segments of event type 5. By lemma 4.3.25 these events always succeed and no event of event types 7 and 8 ever occurs.
Protocol 4.1 (part 1): $\Pi_{\text{ledger}}^{L_{\text{meta}}, L_{\text{base}}, \text{COMP}, \text{DECOMP}, \text{IS-LIVE}}$

The party maintains:

- $\text{TxStepQueues}$— maps transactions to their steps
- $\text{LockingFragmentTxos}$— maps locking fragment outputs to explicitly-meta outputs
- $\text{FragmentTxos}$— all fragment outputs
- $\text{MsgQueue}$— a queue of messages to be output

1: procedure GET-TX-IDS($\hat{t}_x$)

2: $[T_{x_1}^{m}, ..., T_{x_n}^{m}] \leftarrow \text{COMP}(\text{out}_b, \text{out}_m, \hat{t}_x)$

3: $tx_k \leftarrow$ the keystone $\quad // (tx_k, aux_k) \in \bigcup_{i=1}^{n} T_i^{tx_m}$, see definition 4.2.9

4: $tx_s \leftarrow$ the splitter $\quad // (tx_s, aux_s) \in \bigcup_{i=1}^{n} T_i^{tx_m}$, see definition 4.2.10

5: $((\text{id}_k, \text{txo_ids}_k), \hat{t}_x_k) \leftarrow \text{parse} tx_k$

6: $((\text{id}_s, \text{txo_ids}_s), \hat{t}_x_s) \leftarrow \text{parse} tx_s$

7: $aux \leftarrow$ entire output of $\text{COMP} \quad // [T_{x_1}^{tx_m}, ..., T_{x_n}^{tx_m}]$

8: return $((\text{id}_s, \text{txo_ids}_s), \hat{t}_x, aux)$

9: end procedure

10: procedure SEND-TX($tx, aux$)

11: if $\text{IS-LIVE}^{\text{meta}}(tx)$ and $aux$ is a valid and admissible COMP output for $tx$

12: then

13: $[T_{x_1}^{tx_m}, ..., T_{x_n}^{tx_m}] \leftarrow \text{parse} aux \quad // aux$ is output of $\text{COMP}$

14: Copy signatures from $tx$ to the splitter’s inputs

15: $\text{pending_steps}_{tx} \leftarrow [T_{x_1}^{tx_m}, ..., T_{x_n}^{tx_m}]$

16: $\text{sent_steps}_{tx} \leftarrow$ empty list

17: $\text{TxStepQueues}[tx] \leftarrow (\text{sent_steps}_{tx}, \text{pending_steps}_{tx})$

18: $\text{SEND-NEXT-STEP}(tx)$

19: end if

20: end procedure

...Continued in protocol 4.1 (part 2)...

63
4 Generically Realizing a Meta-Ledger

Protocol 4.1 (part 2): $\Pi^L_{\text{meta}, \text{base}, \text{COMP}, \text{DECOMP}, \text{IS-LIVE}}$

...Continued from protocol 4.1 (part 1)...

23: procedure $\text{TX-PUBLISHED-BY-} \mathcal{F}_\text{ledger} (tx)$
24: \hspace{3mm} // Called when $\mathcal{F}_\text{ledger}$ outputs (Tx-Published, $tx$)
25: \hspace{3mm} $(tx^\text{out}_{\text{DECOMP}}, \text{fragments}) \leftarrow \text{DECOMP}(out_b, out_m, tx)$
26: \hspace{3mm} $tx_k \leftarrow$ the keystone \hspace{3mm} // $tx_k = tx$, see definition 4.2.9
27: \hspace{3mm} $tx_s \leftarrow$ the splitter \hspace{3mm} // $tx_s \in \text{fragments}$, see definition 4.2.10
28: \hspace{3mm} // Existence of $tx_k$ and $tx_s$ is ensured by condition 2 in definition 4.2.13
29: \hspace{3mm} if
30: \hspace{5mm} $tx_k$ is a non-simple keystone with respect to $tx^\text{out}_{\text{DECOMP}}$ and
31: \hspace{5mm} \hspace{3mm} // i.e., $tx_k \neq tx_s$, see definition 4.2.11
32: \hspace{5mm} $tx_s \notin \text{MsgQueue}$
33: \hspace{5mm} then
34: \hspace{7mm} $tx^m \leftarrow tx_k$
35: \hspace{5mm} else
36: \hspace{7mm} $tx^m \leftarrow tx^\text{out}_{\text{DECOMP}}$
37: \hspace{5mm} end if
38: \hspace{3mm} if $tx_k$ is a simple keystone with respect to $tx^m$ then
39: \hspace{5mm} SIMPLE-KEYSTONE-PUBLISHED($tx_k, tx^m$)
40: \hspace{5mm} else
41: \hspace{7mm} $tx'_s \leftarrow tx_s$’s version in $\text{MsgQueue}$
42: \hspace{7mm} NON-SIMPLE-KEYSTONE-PUBLISHED($tx_k, tx'_s, tx^m$)
43: \hspace{5mm} end if
44: ADD-LOCKING-INPUTS() \hspace{3mm} // End of $\text{MsgQueue}$ modifications
45: for all meta output $txo^m \in tx^m$ do
46: \hspace{3mm} Add all fragment outputs in $\text{fragments}$ to $\text{FragmentTxos}$
47: \hspace{3mm} Map all locking fragment outputs in $\text{fragments}$ to $txo^m$ in $\text{LockingFragmentTxos}$
48: end for
49: FLUSH-MSG-QUEUE()
50: end procedure

...Continued in protocol 4.1 (part 3)..
4 Generically Realizing a Meta-Ledger

Protocol 4.1 (part 3): $\Pi_{\mathcal{L}_{\mathit{meta}},\mathcal{L}_{\mathit{base}},\mathit{COMP},\mathit{DECOMP},\mathit{IS-LIVE}}$

(Ledger Event)

...Continued from protocol [4.1 (part 2)]...

52: **procedure** TXOS-RELEASED-BY-$F_{\mathit{ledger}}$($tx$)
53:     // Called when $F_{\mathit{ledger}}$ outputs (Txos-Released, $tx$)
54:     $tx^m \leftarrow$ the transaction that was already published and shares $tx$’s outputs
55:     $\mathit{MsgQueue}.\mathit{push}((\mathit{Txos-Released}, tx^m))$ // End of $\mathit{MsgQueue}$ modifications
56:     
57:     for all $tx \in \mathit{TxStepQueues}$ do
58:         ($\mathit{sent\_steps}_{tx}, \mathit{next\_steps}_{tx}) \xleftarrow{\text{parse}} \mathit{TxStepQueues}[tx]
59:         if all of $\mathit{sent\_steps}_{tx}$’s outputs were released by $F_{\mathit{ledger}}$ then
60:             if $\mathit{next\_steps}_{tx}$ contains more steps then
61:                 SEND-NEXT-STEP($tx$)
62:             else
63:                 Remove $tx$ from $\mathit{TxStepQueues}$
64:         end if
65:     end if
66: end for
67: **end procedure**

68: **procedure** TICK-BY-$F_{\mathit{ledger}}()$
69:     // Called when $F_{\mathit{ledger}}$ outputs (Tick)
70:     for all $tx \in \mathit{TxStepQueues}$ do
71:         $T_{last\_sent} \leftarrow$ last sent step in $\mathit{TxStepQueues}[tx]$
72:         if $T_{last\_sent}$ was sent to $F_{\mathit{ledger}}$ more than $\delta_{\mathit{base}}$ ticks ago then
73:             Remove $tx$ from $\mathit{TxStepQueues}$
74:         end if
75:     end for
76: **end procedure**

...Continued in protocol [4.1 (part 4)]...

65
Protocol 4.1 (part 4): \( \Pi_{\text{ledger}}^{L_{\text{meta}}, L_{\text{base}}, \text{COMP}, \text{DECOMP}, \text{IS-LIVE}} \)
(Internals)

...Continued from protocol 4.1 (part 3)...

77: procedure SEND-NEXT-STEP\( (tx) \)
78: \( ([T_1^{tx}, ..., T_{m-1}^{tx}], [T_m^{tx}, ..., T_n^{tx}]) \xlongequal{\text{parse}} \text{TxStepQueues}\{tx\} \)
79: for all \( (tx_f, aux_f) \in T_m^{tx} \) do
80: \( F_{\text{ledger}}.\text{SEND-TX}(tx_f, aux_f) \)
81: end for
82: \( \text{TxStepQueues}\{tx\} \leftarrow ([T_1^{tx}, ..., T_m^{tx}], [T_{m+1}^{tx}, ..., T_n^{tx}]) \)
83: end procedure

84: procedure FLUSH-MSG-QUEUE\()
85: while \( \text{MsgQueue} \) is not empty do
86: if
87: \( \text{MsgQueue}.\text{PEEK}() \) is \( \text{(Tx-Published, tx)} \) and
88: \( tx \) is marked as fragment and
89: \( tx \)'s outputs are not in \( \text{FragmentTxos} \) and
90: \( tx \) has been published by \( F_{\text{ledger}} \) less than \( s \cdot \delta_{\text{base}} \) ticks ago
91: then
92: break
93: end if
94: \( msg \leftarrow \text{MsgQueue}.\text{POP}() \)
95: if \( msg \) is \( \text{Tx-Published} \) or \( \text{Txos-Released} \) then
96: Output \( msg \)
97: end if
98: end while
99: end procedure

...Continued in protocol 4.1 (part 5)...

66
Protocol 4.1 (part 5): $\Pi_{\text{ledger}}^{L_{\text{meta}}, L_{\text{base}}, \text{COMP}, \text{DECOMP}, IS-LIVE}$

(Decompile)

...Continued from protocol 4.1 (part 4)...

100: procedure ADD-LOCKING-INPUTS()
101:     for all $m \in \text{MsgQueue}$ if $m = (\text{Tx-Published, } tx)$ do
102:         $\text{inputs}_{\text{temp}} \leftarrow \text{fresh copy of } tx$’s inputs
103:             for all $txi \in \text{inputs}_{\text{temp}}$ do
104:                 if $txi$ references a fragment output $txo_f$ in $\text{LockingFragmentTxos}$ then
105:                     $(id_{txo_f}, \text{amount, } data^{lock}, data^{spend}) \xrightarrow{\text{parse}} txi$
106:                     $txo^m \leftarrow \text{LockingFragmentTxos}[txo_f]$
107:                     $txi_{\text{lock}} \leftarrow (id_{txo^m}, 0, data^{lock}, \perp)$
108:                     if $tx$ doesn’t already have an input identical to $txi_{\text{lock}}$ then
109:                         Append $txi_{\text{lock}}$ to $tx$’s inputs in $m$ in $\text{MsgQueue}$
110:                 end if
111:             end for
112: end for
113: end procedure

115: procedure SIMPLE-KEYSTONE-PUBLISHED($tx_k, tx^m$)
116:     $((id_k, txo_{\text{ids}_k}), (\text{inputs}_k, \text{outputs}_k)) \xrightarrow{\text{parse}} tx_k$
117:     $((id_m, txo_{\text{ids}_m}), (\text{inputs}_m, \text{outputs}_m)) \xrightarrow{\text{parse}} tx^m$
118:     $\text{MsgQueue}.\text{PUSH}(((\text{Tx-Published, } (id_m, txo_{\text{ids}_k}), (\text{inputs}_m, \text{outputs}_k))))$
119: end procedure

...Continued in protocol 4.1 (part 6)...

Protocol 4.1 (part 6): $\Pi_{\text{ledger}}^{L_{\text{meta}}, L_{\text{base}}, \text{COMP}, \text{DECOMP}, \text{IS-LIVE}}$

(Decomilation)

...Continued from protocol 4.1 (part 5)...

120: procedure NON-SIMPLE-KEYSTONE-PUBLISHED($tx_k, tx_s, tx^m$)
121:   (($id_k, txo_ids_k), ([txi_k], outputs_k))$\xleftarrow{\text{parse}}$ $tx_k$
122:   (($id_s, txo_ids_s), (inputs_s, outputs_s))$\xleftarrow{\text{parse}}$ $tx_s$
123:   (($id_m, txo_ids_m), (inputs_m, outputs_m))$\xleftarrow{\text{parse}}$ $tx^m$  // $txo_ids_m = txo_ids_k$
124: // Initializations, fresh copies:
125: $tx^m_\star \leftarrow ((id_s, txo_ids_m), (inputs_s, outputs_m))$  // inputs_m are discarded
126: $id_s \leftarrow$ new ID in the “meta namespace”
127: $txs_\star \leftarrow ((id_s, txo_ids_s), (inputs_s, outputs_s))$
128: $tx_dummy \leftarrow ((id_k, [\,]), ([txi_k], [\,]))$  // no outputs
129: amount_k $\leftarrow$ the sum of outputs_k’s amounts
130: $txo_s \leftarrow tx_s$’s output spent by $txi_k$
131: $txo_s \prime \leftarrow txo_s$’s output that corresponds to $txo_s$
132: Subtract amount_k from $txo_s$’s output amount
133: Subtract amount_k from the amount of $tx_dummy$’s input
134: $i \leftarrow 1$
135: while $i \leq txs_\star$’s input count do
136:   $txs_{\star, i} \leftarrow txs_\star$’s $i$th input
137:   amount_{k, i} $\leftarrow$ min(amount_{txs_{\star, i}}, amount_k)
138:   Subtract amount_{k, i} from $txs_{\star, i}$’s input amount
139:   Set the input amount of $tx^m_\star$’s $i$th input to amount_{k, i}
140:   amount_k $\leftarrow$ amount_k $-$ amount_{k, i}
141:   $i \leftarrow i + 1$
142: end while
143:
144: Insert (Tx-Published, $tx^m$) in $\text{MsgQueue}$ before (Tx-Published, $tx_s$)
145: Replace (Tx-Published, $tx_s$) with (Tx-Published, $tx_s$) in $\text{MsgQueue}$
146: $\text{MsgQueue}.\text{push}((\text{Tx-Published}, tx_{dummy}))$
147: end procedure
Algorithm 4.1 (part 2): $S_{\text{ledger}}^{\text{meta,comp,decomp}}$

...Continued from algorithm 4.1 (part 1)...

14: **procedure** $\text{PROCESS-}S_{\text{ph}}\text{-MESSAGE}(m)$
15: // Called when $S_{\text{ph}}$ outputs a message $m$
16: if $m = (\text{Tx-Published}, tx)$ then
17: \hspace{0.5cm} $F_{\text{meta}}\cdot\text{PROCESS-TX}(tx)$
18: else if $m = (\text{Txos-Released}, tx)$ then
19: \hspace{0.5cm} $F_{\text{meta}}\cdot\text{RELEASE-TXOS}(tx)$
20: end if
21: end procedure

22: **procedure** $S_{\text{ph}}\text{-TX-REMOVED-POPPED}(tx)$
23: // Called when $S_{\text{ph}}$ pops a $(\text{Tx-Removed}, tx)$ message from $\text{MsgQueue}$ in line 94 of FLUSH-MSG-QUEUE.
24: \hspace{0.5cm} $F_{\text{meta}}\cdot\text{REMOVE-TX}(tx)$
25: end procedure

26: **procedure** $\text{CHECK-}S_{\text{ph}}\text{-FRAGMENT-VALIDITY}()$
27: // Called after $S_{\text{ph}}$ runs either:
28: * line 44 of TX-PUBLISHED-BY-$F_{\text{ledger}}$ or
29: * line 55 of TXOS-RELEASED-BY-$F_{\text{ledger}}$,
30: Pause the $S_{\text{ph}}$ simulation, store its execution state
31: for all $tx \in S_{\text{ph}}$’s mempool do
32: if $tx$ was sent by $S_{\text{ph}}$ and $\neg\text{IS-STRONGLY-VALID}(tx)$ then
33: Let $tx_{\text{origin}}$ be the transaction that
34: triggered the sending of $tx$.
35: $S_{\text{ph}}\cdot\text{MsgQueue}.\text{PUSH}((\text{Tx-Removed}, tx_{\text{origin}}))$
36: end if
37: end for
38: Restore the $S_{\text{ph}}$ execution state, continue its simulation
39: end procedure
Algorithm 4.2: APPLY-QUEUE

1: procedure APPLY-QUEUE(q, s)
2:     $s' \leftarrow \text{fresh copy of } s$; $q' \leftarrow \text{fresh copy of } q$
3:     while $q'$ is not empty do
4:         $msg \leftarrow q'.\text{POP}()$
5:         if $s' \leftarrow \text{APPLY-MESSAGE}(msg, s') = \bot$ then
6:             return $\bot$
7:         end if
8:     end while
9:     return $s'$
10: end procedure

11: procedure APPLY-MESSAGE(msg, s)
12:     $s' \leftarrow \text{fresh copy of } s$
13:     if msg is (Tx-Published, tx) then
14:         Call $F_{\text{ledger}}.\text{PROCESS-TX}(tx)$ with respect to $s'$ (see functionality 3.2, line 20)
15:         if the internal call to VALIDATE-AND-APPLY-TX returned false then
16:             return $\bot$
17:         end if
18:     end if
19:     else if msg is (Txos-Released, tx) then
20:         Call $F_{\text{ledger}}.\text{RELEASE-TXOS}(tx)$ with respect to $s'$ (see functionality 3.2, line 36)
21:         if tx’s outputs were not released then
22:             return $\bot$
23:         end if
24:     end if
25:     else if msg is (Tx-Removed, tx) then
26:         Call $F_{\text{ledger}}.\text{REMOVE-TX}(tx)$ with respect to $s'$ (see functionality 3.2, line 45)
27:         if tx was not removed from $s'$’s mempool in line 48 then
28:             return $\bot$
29:         end if
30:     end if
31:     return $s'$
32: end procedure
5 A Meta-Ledger With Circuit Support

In this chapter we describe the protocols for COMP and DECOMP—the remaining pieces that, together with our generic ledger protocol complete the construction of a meta-ledger with support for boolean circuit spending conditions.

5.1 Circuit Compiler

The compiler for circuit transactions appears in protocol 5.1. This compiler uses the construction explained in chapter 2. For brevity, we don’t repeat the details of the compilation process in pseudocode. However, it is helpful to make the following points explicit:

- Our compiler does not use the current base-ledger and meta-ledger states.
- When we say “Construct a transaction” in our pseudocode, this means generating a transaction tuple that can be sent as input to the base-ledger’s send-tx method. In particular, it includes querying the base-ledger for the transaction’s ID and auxiliary data (using the ledger’s get-tx-ids method).
- All the fragments generated by the compiler are also signed by the compiler—except for the splitter transaction, which is returned unsigned. (The splitter cannot be signed by the compiler, since these signatures are verified by the outputs it spends, with respect to the verification keys specified in these outputs—unlike internal fragments, whose outputs are generated by the compiler, including their signature key pairs). To compute the splitter signatures, the generic protocol converts the honest party’s signature on the meta-transaction to a signature on the splitter base transaction.
- Fragment transactions are specially marked in a way that is allowed in the base-ledger but not in the meta-ledger (c.f. section 3.5.5).

5.1.1 Validating Compilation

The output of the compiler is easily identifiable, using the fragment marks and transaction graph. Given a potential keystone transaction $tx_k$ and current base-ledger state, the following algorithm will verify if it could indeed have been generated by the compiler (and if so, reconstruct the input to the compiler).

Case 1: If $tx_k$ satisfies all of the following:

- $tx_k$ is marked as a keystone
- $tx_k$ has a single input that spends a fragment-marked transaction $tx_s$
- The other outputs of $tx_s$ are all spent by fragment-marked transactions that match the format of input-bit transactions or circuit-gate transactions (cf. section 2.1.1) and all have the same locking condition.

Input-bit and gate transactions can be recognized and told apart by the content of their output conditions (see section 2.1). Both input-bit and gate transactions have two outputs each; while an input-bit output condition doesn’t reference other
Protocol 5.1: Circuit Compiler

1: procedure \( \text{comp}^\text{Circuit}(\text{out}_b, \text{out}_m, \text{tx}^m) \)
2: \begin{align*}
3: & \text{if } \text{tx}^m \text{ has a circuit-output } C \text{ then} \\
4: & \quad [T_1^{\text{splitter}}, \ldots, T_i^{\text{splitter}}] \leftarrow \text{Construct splitter transactions as described in section 2.1.4} \\
5: & \quad T_1^{\text{input-bit}} \leftarrow \text{Construct input-bit transactions as described in section 2.1.1} \\
6: & \quad [T_1^{\text{gate}}, \{\text{tx}^b_{C-\text{keystone}}\}] \leftarrow \text{Construct gate transactions as described in section 2.1.1} \\
7: & \quad \text{return } [T_1^{\text{splitter}}, \ldots, T_i^{\text{splitter}}, T_1^{\text{input-bit}} \cup T_1^{\text{gate}}, \{\text{tx}^b_{C-\text{keystone}}\}] \\
8: & \text{else if } \text{tx}^m \text{ has a circuit-input assignment then} \\
9: & \quad T_1^{\text{input-bit-label}} \leftarrow \text{Construct input-bit-labeling transactions as described in section 2.2.1} \\
10: & \quad [T_1^{\text{gate-label}}, \ldots, T_i^{\text{gate-label}}] \leftarrow \text{Construct gate-labeling transactions as described in section 2.2.1} \\
11: & \quad \text{return } [T_1^{\text{input-bit-label}}, T_1^{\text{gate-label}}, \ldots, T_i^{\text{gate-label}}, \{\text{tx}^b_{A-\text{keystone}}\}] \\
12: & \text{else } // \text{tx}^m \text{ is a base transaction} \\
13: & \quad \text{return } \{\text{tx}^m\} \\
14: & \end{align*}
15: \text{end procedure}

transaction outputs, each gate transaction output condition will reference two “previous” input-bit and gate transactions using \text{OP\_IS\_TXO\_UNSPENT}. Because the output conditions have a fixed format, with the only difference between them being the id of the referenced transaction outputs, simple pattern matching can be used to identify them.

Then \( \text{tx}_s \) is the keystone of a circuit-output transaction. The circuit can be recovered by parsing the \text{OP\_IS\_TXO\_UNSPENT} references between the gate and input bit transactions. The fragments consist of the splitter transaction \( \text{tx}_s \) and all the input-bit and gate transactions that spend outputs of \( \text{tx}_s \).

Case 2: If \( \text{tx}_k \) satisfies all of the following:
- \( \text{tx}_k \) is marked as a keystone
- \( \text{tx}_k \) has a single input that spends a keystone-marked transaction \( \text{tx}_{k'} \)
- \( \text{tx}_{k'} \) is recognized as the keystone of a circuit-output transaction \( \text{tx}^m \)
- All transactions that spend fragments of \( \text{tx}^m \) are themselves fragment-marked.

Then \( \text{tx}_k \) is the keystone of a circuit-assignment transaction. The assignment itself can be recovered using the \text{EXTRACT-ASSIGNMENT} procedure (cf. section 2.3). (By observation, the result of this procedure on an honestly-compiled circuit-assignment will return the same assignment). The corresponding fragments are the transactions that spend fragments of \( \text{tx}^m \).

Case 3: Otherwise, \( \text{tx}_k \) is not a keystone that could have been output by \( \text{comp}^\text{Circuit} \).

5.2 Circuit Decompiler

The circuit decompiler, given the outputs of the base and meta-ledgers, and keystone base transaction (i.e., one that is, potentially, the keystone of a meta-transaction), can distinguish the following cases:
5 A Meta-Ledger With Circuit Support

Case 1: The keystone is validated as an output of \( \text{COMP}^{\text{Circuit}} \) on transaction \( tx^m \), and \( tx^m \) is valid in \( out_m \). In this case, the decompiler returns \( tx^m \).

Case 2: The keystone is not validated as an output of \( \text{COMP}^{\text{Circuit}} \), but spends an output that is a meta-output in the meta-ledger. In this case, the decompiler uses the extraction algorithm guaranteed by section 2.3 to create a valid meta-transaction with a meta-input, whose outputs are identical to the keystone outputs (i.e., they are base outputs in this case).

Case 3: Otherwise, the decompilation “fails”. In this case, the decompiler returns the keystone transaction as-is (i.e., treated as a base transaction).

Protocol 5.2: Circuit Decompiler

1: procedure \( \text{DECOMP}^{\text{Circuit}}(out_b, out_m, tx_k) \)
2: if (fragments, \( tx_k \)) is a valid output of \( \text{COMP}^{\text{Circuit}}(\cdot, \cdot, tx^m) \) then // Case 1
3: return (fragments, \( tx^m \))
4: else if \( tx_k \) spends an output that is a meta-output in \( out_m \) then // Case 2
5: for every meta-output \( txo^m \) spent by \( tx_k \) do
6: Compute the assignment \( A_{txo^m} \) by calling \( \text{EXTRACT-ASSIGNMENT} \) with \( out_b \) and the corresponding input.
7: create the meta-input \( txi^m \) using this assignment
8: end for
9: return a new \( tx^m \) that is identical to \( tx_k \), except that all inputs that spend meta-outputs are replaced by the corresponding meta-inputs.
10: else // Case 3
11: return \( tx_k \)
12: end if
13: end procedure

5.3 Security Analysis: Circuit Compiler and Decompiler are Admissible

We note that the decompilation cannot fail if a meta-transaction was compiled honestly (since in that situation case 1 will hold; thus, by construction):

**Lemma 5.3.1** (\( \text{COMP}^{\text{Circuit}} / \text{DECOMP}^{\text{Circuit}} \) are complete). \( \text{COMP}^{\text{Circuit}} \) and \( \text{DECOMP}^{\text{Circuit}} \) satisfy definition 4.2.12.

On the other hand, if the decompiler does fail, case 2 cannot hold, so all of the keystone’s inputs are base outputs in \( out_m \). This implies that the unmodified keystone transaction is valid in \( out_m \), because it is valid in \( out_b \) and \( out_m \xrightarrow{\text{state}} out_b \).

In order to plug our compiler/decompiler pair into the generic ledger protocol, we must prove they are admissible (c.f. definition 4.2.19). We do this in lemmas 5.3.2 to 5.3.1:

**Lemma 5.3.2** (\( \text{COMP}^{\text{Circuit}} \) is valid). \( \text{COMP}^{\text{Circuit}} \) (cf. protocol 5.1) satisfies definition 4.2.18.

**Proof Sketch.** We will consider separately each validity condition:
Condition 1: The output of $\text{COMP}_{\text{Circuit}}$ always contains a splitter—i.e., a base transaction whose inputs are identical to the meta transaction. This can be seen by observation: if $tx^m$ has a meta-input, the splitter is the keystone, otherwise it is explicitly constructed (c.f. section 2.1.4).

Condition 2: If $tx^m$ has a meta-output, the splitter is the first fragment output by $\text{COMP}$. Let $out_b^{\text{ext}}$ be an extension of $out_b$ that does not contain the splitter. Then if the splitter is not live in $out_b^{\text{ext}}$, one of its inputs must not be valid. Since, by definition, the inputs of the splitter are identical to the inputs of $tx^m$, for every ledger output $out_m^{\text{ext}} \rightsquigarrow_{tx} out_b^{\text{ext}}$ it must hold that $tx^m$ is not live in $out_m^{\text{ext}}$. If $tx^m$ has a meta-input, the splitter transaction is the keystone. All the inputs of all other fragments spend explanatory fragments for the meta-output spent by $tx^m$. These are fragments that could have been output by $\text{COMP}_{\text{Circuit}}$, so their locking conditions must all be identical to each other and to the corresponding meta-output’s locking condition, while their spending conditions only consider the state of the other explanatory fragments for the same meta-output.

Thus, in order to block a fragment (or the splitter itself), one of the explanatory fragments’ outputs must have been locked.

Condition 3: This condition is trivial for the simple keystone case (if $tx^m$ has a meta-input, or $tx^m$ is a base transaction), since in this case the splitter is always the last fragment in $\mathcal{T}(out_b, out_m, tx^m)$. When $tx^m$ has a meta-output, the splitter is always the first fragment of $\mathcal{T}(out_b, out_m, tx^m)$. The keystone and all fragments after the splitter spend only the splitter’s outputs. The spending (and locking) conditions for these outputs only require a signature under a key that is freshly generated by the compiler. Hence, in order to block the fragment transactions (which are correctly signed), the adversary must be able to lock the splitter outputs before the fragments are processed. However, given an adversary that can do this, we can use it as a black box to forge signatures. Thus, no computationally-bounded adversary can do so (by the security of the signature scheme).

Lemma 5.3.3 ($\text{DECOMP}_{\text{Circuit}}$ is valid). $\text{DECOMP}_{\text{Circuit}}$ (cf. protocol 5.2) satisfies definition 4.2.13.

Proof Sketch. We will consider separately each validity condition:

Condition 1: By observation, the fragments returned by $\text{DECOMP}_{\text{Circuit}}$ always exist in the base-ledger output.

Condition 2: By observation, $\text{DECOMP}_{\text{Circuit}}$ always returns the input base transaction as the keystone, and if it is not the splitter itself, the first fragment returned is the splitter (since this means we are in case 1 and $\text{COMP}_{\text{Circuit}}$ outputs a splitter as the first fragment).

Conditions 3 to 5: If $tx^m$ has a meta-output, we are in case 1; therefore the keystone was output by $\text{COMP}_{\text{Circuit}}$, which always preserves the output amount and sets the locking condition of all unspent-output fragments (i.e., the input bits and gate outputs) to be identical to the locking condition of $tx^m$. Otherwise (cases 2 and 3), the meta-outputs generated by $\text{DECOMP}_{\text{Circuit}}$ are identical to the keystone outputs (including in amount).

Condition 6: If $tx^m$ is output due to case 1, this condition holds because $\text{COMP}_{\text{Circuit}}$ will only accept a meta-transaction with a meta-input if it does not also have a meta-output—in which case it outputs a simple keystone. If we’re in case 3, $\text{DECOMP}_{\text{Circuit}}$ returns the keystone
itself as $tx^m$, hence it is simple. Otherwise (we’re in case [2]), DECOMP\text{Circuit} outputs a meta-transaction that spends the meta-output corresponding to the base output spend by $tx_k$, and has the same outputs as $tx_k$—hence, by definition, $tx_k$ is a simple keystone for $tx^m$.

Condition 7 If we are in case [1], $tx^m$ is by definition valid. If we are in case [3], $tx^m$ is simply the keystone, which does not spend a meta input (otherwise we would be in case [2]). Since the keystone is valid in $out_b$, and the outputs it spends are identical in $out_m$, due to $out_m \xrightarrow{\text{state}} out_b$, it will also be valid in $out_m$. If we are in case [2] the keystone does spend a meta-output, which by consistency must be explainable as a valid output COMP\text{Circuit}. Thus, we can use compiler-soundness (theorem 2.3.1) to conclude that the extracted assignment must be valid in $out_m$.

Condition 8 The only case that could return a non-simple keystone is case [1]. However, in case [1] the decompiler returns $tx^m$ only if the fragments could have been generated by an honest execution of COMP\text{Circuit}. Since COMP\text{Circuit} generates a simple keystone when $tx^m$ has a meta-input, the condition is satisfied.

Taken together, lemmas 5.3.1 to 5.3.3 immediately imply the following:

**Corollary 5.3.4** (Admissibility), COMP\text{Circuit} and DECOMP\text{Circuit} are admissible (satisfy definition 4.2.19).
6 Discussion and Open Questions

We have shown how to construct a meta-ledger that supports arbitrary circuit conditions using a slightly enhanced version of Bitcoin. Our ledger functionality is composable and “stackable”—given a suitable compiler/decompiler pair, our protocol can construct a meta-meta-ledger.

Many interesting open questions remain.

- Can we construct an admissible compiler/decompiler pair for arbitrary script conditions? As a first approximation, it may be useful to consider the current Bitcoin scripting language, but remove the length limitation (i.e., allow longer scripts by using multiple transactions)

- Our requirements from comp/decomp may appear unduly restrictive. While some requirements appear to be inherent, it is an interesting open question to design meta-ledger protocols that support a wider variety of spending conditions. In particular, our protocols do not support spending conditions that can depend on the contents of a spending transaction (rather than just its ID), such as the extensions proposed by Bitcoin covenants.
Bibliography


bitcoin-plus
השימוש הבסיסי במטבע מבוזר (cryptocurrency) הוא העברת ערך כספי בין זהויות, אך רבים מהשימושים המעניינים יותר דורשים שימוש בחוזים, למשל: "העברה של \( X \) מטבעות מישות \( S \) לישות \( R \) תתבצע רק אם תנאים \( A \) ו\( B \) יתקימו". ביטקוין (ומטבעות מבוזרים דומים) מציבים גבולות נוקשים על השפה בה מנוסחים אותם תנאים. בפרט, התנאים מורכבים ולא ניתן לבלול.

בעבודה זו אנו מראים כיצד ניתן להרחיב את שפת התסריט (scripting language) של ביטקוין עם פקודת יחידה (opcode) יחידה, שהוספתה לא מזיקה, היא מאפשרת ליצור "מטא-תנאים" בעלי כוח הבעתי רב יותר (בצורת מעגליים בוליאניים בגודל שרירותי).

אנחנו בונים פרוטוקול של הסדרת עסקאות ביטקוין (transactions) בסיסיות שכוללות את הפקודה החדשה. אנחנו מראים איך להשתמש במהדר זה כדי לממש פונקציונאליות meta-ledger, ואתו מוכיחים שהוא בטוח במודל של Universal Composability. Composability
עבודת זו∧ בוצעה בהדרכתו של דר’ טל מורן מבי”ס אפי ארזי למדעי המחשב, המרכז ההיבנתחומי, הרצליה.
המרכז הבינתחומי בהרצליה
בית הספר אפי ארזי למדעי המחשב
התכנית לתואר שני (M.Sc) – מסלול מחקרי

-work of Yaron Kuhr

研究报告: Bitcoin+
UTXO Ledger

စာရင်း: セキュリティのためのネットワーキング