Proof-of-Stake Sidechains

Proof-of-Stake Sidechains

Peter Gaži[1], Aggelos Kiayias[1,2], and Dionysis Zindros[1,3]

[1] IOHK
[2] University of Edinburgh
[3] National and Kapodistrian University of Athens

December 18, 2018

Abstract. Sidechains have long been heralded as the key enabler of blockchain scalability and interoperability. However, no modeling of the concept or a provably secure construction has so far been attempted.

We provide the first formal definition of what a sidechain system is and how assets can be moved between sidechains securely. We put forth a security definition that augments the known transaction ledger properties of persistence and liveness to hold across multiple ledgers and enhance them with a new "firewall" security property which safeguards each blockchain from its sidechains, limiting the impact of an otherwise catastrophic sidechain failure.

We then provide a sidechain construction that is suitable for proof-of-stake (PoS) sidechain systems. As an exemplary concrete instantiation we present our construction for an epoch-based PoS system consistent with Ouroboros (Crypto 2017), the PoS blockchain protocol used in Cardano which is one of the largest pure PoS systems by market capitalisation, and we also comment how the construction can be adapted for other protocols such as Ouroboros Praos (Eurocrypt 2018), Ouroboros Genesis (CCS 2018), Snow White and Algorand. An important feature of our construction is *merged-staking* that prevents "goldfinger" attacks against a sidechain that is only carrying a small amount of stake. An important technique for pegging chains that we use in our construction is cross-chain certification which is facilitated by a novel cryptographic primitive we introduce called ad-hoc threshold multisignatures (ATMS) which may be of independent interest. We show how ATMS can be securely instantiated by regular and aggregate digital signatures as well as succinct arguments of knowledge such as STARKs and bulletproofs with varying degrees of storage efficiency.

1 Introduction

Blockchain protocols and their most prominent application so far, cryptocurrencies like Bitcoin [27], have been gaining increasing popularity and acceptance by a wider community. While enjoying wide adoption, there are several fundamental open questions remaining to be resolved that include (i) Interoperability: How can different blockchains interoperate and exchange assets or other data? (ii) Scalability: How can blockchain protocols scale, especially proportionally to the number of participating nodes? (iii) Upgradability: How can a deployed blockchain protocol codebase evolve to support a new functionality, or correct an implementation problem?

The main function of a blockchain protocol is to organise *application data* into *blocks* so that a set of nodes that evolves over time can arrive eventually to consensus about the sequence of events that took place. The consensus component can be achieved in a number of ways, the most popular is using proof-of-work [16] (cf. [27,17]), while a promising alternative is to use proof-of-stake (cf. [26,20,5,13]). Application data typically consists of *transactions* indicating some transfer of value as in the case of Bitcoin [27]. The transfer of value can be conditioned on arbitrary predicates called *smart contracts* such as, for example, in Ethereum [11,31].

The conditions used to validate transactions depend on local blockchain events according to the view of each node and they typically cannot be dependent on other blockchain sessions. Being able to perform operations across blockchains, for instance from a main blockchain such as Bitcoin to a "sidechain" that has some enhanced functionality, has been frequently considered a fundamental technology enabler in the blockchain space.[4]

[4] See e.g., `https://blockstream.com/technology/` and [1].

Sidechains, introduced in [1], are a way for multiple blockchains to communicate with each other and have one react to events in the other. Sidechains can exist in two forms. In the first case, they are simply a mechanism for two existing *stand-alone blockchains* to communicate, in which case any of the two blockchains can be the sidechain of the other and they are treated as equals. In the second case, the sidechain can be a "child" of an existing blockchain, the *mainchain*, in that its genesis block, the first block of the blockchain, is somehow seeded from the parent blockchain and the child blockchain is meant to depend on the parent blockchain, at least during an initial bootstrapping stage.

A sidechain system can choose to enable certain types of interactions between the participating blockchains. The most basic interaction is the transfer of assets from one blockchain to another. In this application, the nature of the asset transferred is retained in that it is not transformed into a different class of asset (this is in contrast to a related but different concept of *atomic swaps*). As such, it maintains its value and may also be transferred back. The ability to move assets back and forth between two chains is sometimes referred to as a *2-way peg*. Provided the two chains are both secure as individual blockchains, a secure sidechain protocol construction allows this security to be carried on to cross-chain transfers.

A secure sidechain system could be of a great value vis-à-vis all three of the pressing open questions in blockchain systems mentioned above. Specifically:

Interoperability. There are currently hundreds of cryptocurrencies deployed in production. Transferring assets between different chains requires transacting with intermediaries (such as exchanges). Furthermore, there is no way to securely interface with another blockchain to react to events occurring within it. Enabling sidechains allows blockchains of different nature to communicate, including interfacing with the legacy banking system which can be made available through the use of a private ledger.

Scalability. While sidechains were not originally proposed for scalability purposes, they can be used to off-load the load of a blockchain in terms of transactions processed. As long as 2-way pegs are enabled, a particular sidechain can offer specialization by, e.g., industry, in order to avoid requiring the mainchain to handle all the transactions occurring within a particular economic sector. This provides a straightforward way to "shard" blockchains, cf. [25,21,33].

Upgradability. A child sidechain can be created from a parent mainchain as a means of exploring a new feature, e.g., in the scripting language, or the consensus mechanism without requiring a soft, hard, or velvet fork [19,34]. The sidechain does not need to maintain its own separate currency, as value can be moved between the sidechain and the mainchain at will. If the feature of the sidechain proves to be popular, the mainchain can eventually be abandoned by moving all assets to the sidechain, which can become the new mainchain.

Given the benefits listed above for distributed ledgers, there is a pressing need to address the question of sidechain security and feasibility, which so far, perhaps surprisingly, has not received any proper formal treatment.

Our contributions. First, we formalize the notion of sidechains by proposing a rigorous cryptographic definition, the first one to the best of our knowledge. The definition is abstract enough to be able to capture the security for blockchains based on proof-of-work, proof-of-stake, and other consensus mechanisms.

A critical security feature of a sidechain system that we formalise is the *firewall property* in which a catastrophic failure in one of the chains, such as a violation of its security assumptions, does not make the other chains vulnerable providing a sense of limited liability.[5] The firewall property formalises and generalises the concept of a blockchain *firewall* which was described in high level in [1]. Informally the blockchain firewall

[5] To follow the analogy with the term of limited liability in corporate law, a catastrophic sidechain failure is akin to a corporation going bankrupt and unable to pay its debtors. In a similar fashion, a sidechain in which the security assumptions are violated may not be able to cover all of its debtors. We give no assurances regarding assets residing on a sidechain if its security assumptions are broken. However, in the same way that stakeholders of a corporation are personally protected in case of corporate bankruptcy, the mainchain is also protected in case of sidechain security failures. Our security will guarantee that each incoming transaction from a sidechain will always have a valid explanation in the sidechain ledger independently of whether the underlying security assumptions are violated or not. A simple embodiment of this rule is that a sidechain can return to the mainchain at most as many coins as they have been sent to the sidechain over all time.

suggests that no more money can ever return from the sidechain than the amount that was moved into it. Our general firewall property allows relying on an arbitrary definition of exactly how assets can correctly be moved back and forth between the two chains, we capture this by a so-called *validity language*. In case of failure, the firewall ensures that transfers from the sidechain into the mainchain are rejected unless there exists a (not necessarily unique) plausible history of events on the sidechain that could, in case the sidechain was still secure, cause the particular transfers to take place.

Second, we outline a concrete exemplary construction for sidechains for proof-of-stake blockchains. For conciseness our construction is described with respect to a generic PoS blockchain consistent with the Ouroboros protocol [20] that underlies the Cardano blockchain, which is currently one of the largest pure PoS blockchains by market capitalisation,[6] nevertheless we also discuss how to modify our construction to operate for Ouroboros Praos [13], Ouroboros Genesis [2], Snow White [6] and Algorand [26].

We prove our construction secure using standard cryptographic assumptions. We show that our construction (i) supports safe cross-chain value transfers when the security assumptions of both chains are satisfied, namely that a majority of honest stake exists in both chains, and (ii) in case of a one-sided failure, maintains the firewall property, thus containing the damage to the chains whose security conditions have been violated.

A critical consideration in a sidechain construction is safeguarding a new sidechain in its initial "bootstrapping" stage against a "goldfinger" type of attack [22]. Our construction features a mechanism we call *merged-staking* that allows mainchain stakeholders who have signalled sidechain awareness to create sidechain blocks even without moving stake to the sidechain. In this way, sidechain security can be maintained assuming honest stake majority among the entities that have signaled sidechain awareness that, especially in the bootstrapping stage, are expected to be a large superset of the set of stakeholders that maintain assets in the sidechain.

Our techniques can be used to facilitate various forms of 2-way peggings between two chains. As an illustrative example we focus on a parent-child mainchain-sidechain configuration where sidechain nodes follow also the mainchain (what we call direct observation) while mainchain nodes need to be able to receive cryptographically certified signals from the sidechain maintainers, taking advantage of the proof-of-stake nature of the underlying protocol. This is achieved by having mainchain nodes maintain sufficient information about the sidechain that allows them to authenticate a small subset of sidechain stakeholders that is sufficient to reliably represent the view of a stakeholder majority on the sidechain. This piece of information is updated in regular intervals to account for stake shifting on the sidechain. Exploiting this, each withdrawal transaction from the sidechain to the mainchain is signed by this small subset of sidechain stakeholders. To minimise overheads we batch this authentication information and all the withdrawal transactions from the sidechain in a single message that will be prepared once per "epoch." We will refer to this signaling as *cross-chain certification*.

In greater detail, adopting some terminology from [20], the sidechain certificate is constructed by obtaining signatures from the set of so-called *slot leaders* of the last $\Theta(k)$ slots of the previous epoch, where k is the security parameter. Subsequently, these signatures will be combined together with all necessary information to convince the mainchain nodes (that do not have access to the sidechain) that the sidechain certificate is valid.

We abstract the notion of this trust transition into a new cryptographic primitive called *ad-hoc threshold multisignatures (ATMS)* that we implement in three distinct ways. The first one simply concatenates signatures of elected slot leaders. While secure, the disadvantage of this implementation is that the size of the sidechain certificate is $\Theta(k)$ signatures. An improvement can be achieved by employing multisignatures and Merkle-tree hashing for verification key aggregation; using this we can drop the sidechain-certificate size to $\Theta(r)$ signatures where r slot leaders do not participate in its generation; in the optimistic case $r \ll k$ and thus this scheme can be a significant improvement in practice. Finally, we show that STARKs and bulletproofs [4,10] can be used to bring down the size of the certificate to be optimally succinct in the random oracle model. We observe that in the case of an active sidechain (e.g., one that returns assets at least once per epoch) our construction with succinct sidechain certificates has optimal storage requirements in the mainchain.

[6] See `https://coinmarketcap.com`.

Related work. Sidechains were first proposed as a high level concept in [1]. Notable proposed implementations of the concept are given in [29,23]. In these works, no formal proof of security is provided and their performance is sometimes akin to maintaining the whole blockchain within the sidechain, limiting any potential scalability gains. There have been several attempts to create various cross-chain transfer mechanisms including Polkadot [32], Cosmos [9], Blockstream's Liquid [14] and Interledger [30]. These constructions differ in various aspects from our work including in that they focus on proof-of-work or private (Byzantine) blockchains, require federations, are not decentralized and — in all cases — lack a formal security model and analysis. Threshold multi-signatures were considered before, e.g., [24], without the ad-hoc characteristic we consider here. A related primitive that has been considered as potentially useful for enabling proof-of-work (PoW) sidechains (rather than PoS ones) is a (non-interactive) proof of proof-of-work [18,19]; nevertheless, these works do not give a formal security definition for sidechains, nor provide a complete sidechain construction. We reiterate that while we focus on PoS, our definitions and model are fully relevant for the PoW setting as well.

2 Preliminaries

2.1 Our Model

We employ the model from [13], which is in turn based on [20] and [17]. The formalization we use below captures both synchronous and semi-synchronous communication; as well as both semi-adaptive and fully adaptive corruptions.

Protocol Execution. We divide time into discrete units called *slots*. Players are equipped with (roughly) synchronized clocks that indicate the current slot: we assume that any clock drift is subsumed in the slot length. Each slot sl_r is indexed by an integer $r \in \{1, 2, \ldots\}$. We consider a UC-style [12] execution of a protocol Π, involving an environment $\mathcal{Z}$, a number of parties P_i, functionalities that these parties can access while running the protocol (such as the $\mathsf{DDiffuse}$ used for communication, described below), and an adversary $\mathcal{A}$. All these entities are interactive algorithms. The environment controls the execution by activating parties via inputs it provides to them. The parties, unless corrupted, respond to such activations by following the protocol Π and invoking the available functionalities as needed.

(Semi-)Adaptive Corruptions. The adversary influences the protocol execution by interacting with the available functionalities, and by corrupting parties. The adversary can only corrupt a party P_i if it is given permission by the environment $\mathcal{Z}$ running the protocol execution (captured as a special message from $\mathcal{Z}$ to $\mathcal{A}$). Upon receiving permission from the environment, the adversary corrupts P_i after a certain delay of Λ slots, where Λ is a parameter of our model. In particular, if $\Lambda = 0$ we talk about *fully adaptive corruptions* and the corruption is immediate. The model with $\Lambda > 0$ is referred to as allowing *Λ-semi-adaptive corruptions* (as opposed to the *static corruptions model*, where parties can only be corrupted before the start of the execution). A corrupted party P_i will relinquish its entire state to $\mathcal{A}$; from this point on, the adversary will be activated in place of the party P_i.

(Semi-)Synchronous Communication. We employ the "Delayed Diffuse" functionality $\mathsf{DDiffuse}_{\Delta}$ given in [13] to model (semi-)synchronous communication among the parties. It allows each party to diffuse a message once per round, with the guarantee that it will be delivered to all other parties in at most Δ slots (the delay within this interval is under adversarial control). The adversary can also read and reorder all messages that are in transit, as well as inject new messages. We provide a detailed description of the functionality $\mathsf{DDiffuse}_{\Delta}$ in Appendix A for completeness.

We refer to the setting where honest parties communicate via $\mathsf{DDiffuse}_{\Delta}$ as the *Δ-semi-synchronous setting* and sometimes omit Δ if it is clear from the context. The special case of $\Delta = 0$ is referred to as the *synchronous setting*.

Clearly, the above model is by itself too strong to allow us to prove any meaningful security guarantees for the executed protocol without further restrictions (as it, for example, does not prevent the adversary from corrupting all the participating parties). Therefore, in what follows, we will consider such additional assumptions, and will only provide security guarantees as long as such assumptions are satisfied. These assumptions will be specific to the protocol in consideration, and will be an explicit part of our statements.[7]

2.2 Blockchains and Ledgers

A *blockchain* (or a *chain*) (denoted e.g. C) is a sequence of blocks where each one is connected to the previous one by containing its hash.

Blockchains (and in general, any sequences) are indexed using bracket notation. $\mathsf{C}[i]$ indicates the i^{th} block, starting from $\mathsf{C}[0]$, the genesis block. $\mathsf{C}[-i]$ indicates the i^{th} block from the end, with $\mathsf{C}[-1]$ being the tip of the blockchain. $\mathsf{C}[i:j]$ indicates a subsequence, or *subchain* of the blockchain starting from block i (inclusive) and ending at block j (exclusive). Any of these two indices can be negative. Omitting one of the two indexes in the range addressing takes the subsequence to the beginning or the end of the blockchain, respectively. Given blocks A and Z in C, we let $\mathsf{C}\{A:Z\}$ denotes the subchain obtained by only keeping the blocks from A (inclusive) to Z (exclusive). Again any of these two blocks can be omitted to indicate a subchain from the beginning or to the end of the blockchain, respectively. In blockchain protocols, each honest party P maintains a currently adopted chain. We denote $\mathsf{C}^P[t]$ the chain adopted by party P at slot t.

A *ledger* (denoted in bold-face, e.g. $\mathbf{L}$) is a mechanism for maintaining a sequence of transactions, often stored in the form of a blockchain. In this paper, we slightly abuse the language by letting $\mathbf{L}$ (without further qualifiers) interchangeably refer to the algorithms used to maintain the sequence, and all the views of the participants of the state of these algorithms when being executed. For example, the (existing) ledger Bitcoin consists of the set of all transactions that ever took place in the Bitcoin network, the current UTXO set, as well as the local views of all the participants.

In contrast, we call a *ledger state* a concrete sequence of transactions $\mathsf{tx}_1, \mathsf{tx}_2, \ldots$ stored in the *stable* part of a ledger $\mathbf{L}$, typically as viewed by a particular party. Hence, in every blockchain-based ledger $\mathbf{L}$, every fixed chain C defines a concrete ledger state by applying the interpretation rules given as a part of the description of $\mathbf{L}$ (for example, the ledger state is obtained from the blockchain by dropping the last k blocks and serializing the transactions in the remaining blocks). We maintain the typographic convention that a ledger state (e.g. L) always belongs to the bold-face ledger of the same name (e.g. $\mathbf{L}$). We denote by $\mathbf{L}^P[t]$ the ledger state of a ledger $\mathbf{L}$ as viewed by a party P at the beginning of a time slot t, and by $\check{\mathbf{L}}^P[t]$ the complete state of the ledger (at time t) including all pending transactions that are not stable yet. For two ledger states (or, more generally, any sequences), we denote by $\preceq$ the prefix relation.

Recall the definitions of persistence and liveness of a robust public transaction ledger given in the most recent version of [17]:

Persistence. For any two honest parties P_1, P_2 and two time slots $t_1 \leq t_2$, it holds $\mathbf{L}^{P_1}[t_1] \preceq \check{\mathbf{L}}^{P_2}[t_2]$.
Liveness. If all honest parties in the system attempt to include a transaction then, at any slot t after u slots (called the liveness parameter), any honest party P, if queried, will report $\mathsf{tx} \in \mathbf{L}^P[t]$.

For a ledger $\mathbf{L}$ that satisfies persistence at time t, we denote by $\mathbf{L}^{\cup}[t]$ (resp. $\mathbf{L}^{\cap}[t]$) the sequence of transactions that are seen as included in the ledger by *at least one* (resp., *all*) of the honest parties. Finally, $\mathsf{length}(\mathbf{L})$ denotes the length of the ledger $\mathbf{L}$, i.e., the number of transactions it contains.

2.3 Underlying Proof-of-Stake Protocols

For conciseness we present our construction on a generic PoS protoocol based on Ouroboros PoS [20]. As we outline in Appendix B, our construction can be easily adapted to other provably secure proof-of-stake protocols: Ouroboros Praos [13], Ouroboros Genesis [2], Snow White [6], and Algorand [26]. While a full

[7] As an example, we will be assuming that a majority of a certain pool of stake is controlled by uncorrupted parties.

understanding of all details of these protocols is not required to follow our work (and cannot be provided in this limited space), an overview of Ouroboros is helpful to follow the main body of the paper. We provide this high-level overview here, and point an interested reader to Appendix B (or the original papers) for details on the other protocols.

Ouroboros. The protocol operates (and was analyzed) in the synchronous model with semi-adaptive corruptions. In each slot, each of the parties can determine whether she qualifies as a so-called *slot leader* for this slot. The event of a particular party becoming a slot leader occurs with a probability proportional to the stake controlled by that party and is independent for two different slots. It is determined by a public, deterministic computation from the stake distribution and so-called *epoch randomness* (we will discuss shortly where this randomness comes from) in such a way that for each slot, exactly one leader is elected.

If a party is elected to act as a slot leader for the current slot, she is allowed to create, sign, and broadcast a block (containing transactions that move stake among stakeholders). Parties participating in the protocol are collecting such valid blocks and always update their current state to reflect the longest chain they have seen so far that did not fork from their previous state by too many blocks into the past.

Multiple slots are collected into *epochs*, each of which contains $R \in \mathbb{N}$ slots. The security arguments in [20] require $R \geq 10k$ for a security parameter k; we will consider $R = 12k$ as additional $2k$ slots in each epoch will be useful for our construction. Each epoch is indexed by an index $j \in \mathbb{N}$. During an epoch j, the stake distribution that is used for slot leader election corresponds to the distribution recorded in the ledger up to a particular slot of epoch $j-1$, chosen in a way that guarantees that by the end of epoch $j-1$, there is consensus on the chain up to this slot. (More concretely, this is the latest slot of epoch $j-1$ that appears in the first $4k$ out of its total R slots.) Additionally, the *epoch randomness* η_j for epoch j is derived during the epoch $j-1$ via a *guaranteed-output delivery coin tossing* protocol that is executed by the epoch slot leaders, and is available after $10k$ slots of epoch $j-1$ have passed.

In our treatment, we will refer to the relevant parts of the above-described protocol as follows:

- $\mathsf{GetDistr}(j)$ returns the stake distribution SD_j to be used for epoch j, as recorded in the chain up to slot $4k$ of epoch $j-1$;
- $\mathsf{GetRandomness}(j)$ returns the randomness η_j for epoch j as derived during epoch $j-1$;
- $\mathsf{ValidateConsensusLevel}(\mathsf{C})$ checks the consensus-level validity of a given chain C: it verifies that all block hashes are correct, signatures are valid and belong to eligible slot leaders;
- $\mathsf{PickWinningChain}(\mathsf{C}, \mathcal{C})$ applies the chain-selection rule: from a set of chains $\{\mathsf{C}\} \cup \mathcal{C}$ it chooses the longest one that does not fork from the current chain C more than k blocks in the past;
- $\mathsf{SlotLeader}(U, j, sl, \mathsf{SD}_j, \eta_j)$ determines whether a party U is elected a slot leader for the slot sl of epoch j, given stake distribution SD_j and randomness η_j.

Moreover, the function $\mathsf{EpochIndex}$ (resp. $\mathsf{SlotIndex}$) always returns the index of the current epoch (resp. slot), and the event $\mathsf{NewEpoch}$ (resp. $\mathsf{NewSlot}$) denotes the start of a new epoch (resp. slot). Since we use these functions in a black-box manner, our construction can be readily adapted to PoS protocols with a similar structure that differ in the details of these procedures.

Ouroboros was shown in [20] to achieve both persistence and liveness under the following assumptions: (1) synchronous communication; (2) $2R$-semi-adaptive corruptions; (3) majority of stake in the stake distribution for each epoch is always controlled by honest parties during that epoch.

3 Defining Security of Pegged Ledgers

In this section we give the first formal definition of security desiderata for a system of pegged ledgers (popularly often called sidechains). We start by conveying its intuition and then proceed to the formal treatment.

We consider a setting where a set of parties run a protocol maintaining n ledgers $\mathbf{L}_1, \mathbf{L}_2, \ldots, \mathbf{L}_n$, each of the ledgers potentially carrying many different assets. (This protocol might of course be a combination

of subprotocols for each of the ledgers.) For each $i \in [n]$, we denote by $\mathbb{A}_i$ the security assumption required by $\mathbf{L}_i$: For example, $\mathbb{A}_i$ may denote that there has never been a majority of hashing power (or stake in a particular asset, on this ledger or elsewhere) under the control of the adversary; that a particular entity (in case of a centralized ledger) was not corrupted; and so on. We assume that all $\mathbb{A}_i$ are *monotone* in the sense that once violated, they cannot become true again. Formally, $\mathbb{A}_i$ is a sequence of events $\mathbb{A}_i[t]$ for each time slot t that satisfy $\neg\mathbb{A}_i[t] \Rightarrow \neg\mathbb{A}_i[t+1]$ for all t.

There is an a priori unlimited number of (types of) assets, each asset representing e.g. a different cryptocurrency. For simplicity we assume that assets of the same type are fungible, but our treatment easily covers also non-fungible assets. We will allow specific rules of behavior for each asset (called *validity languages*), and each asset behaves according to these rules on each of the ledgers where it is present.

We will fix an operator $\mathsf{merge}(\cdot)$ that merges a set of ledger states $\mathcal{L} = \{\mathsf{L}_1, \mathsf{L}_2, \ldots, \mathsf{L}_n\}$ into a single ledger state denoted by $\mathsf{merge}(\mathcal{L})$. We will discuss concrete instantiations of $\mathsf{merge}(\cdot)$ later, for now simply assume that some canonical way of merging all ledger states into one is given.

Informally, at any point t during the execution, our security definition only provides guarantees to the subset $\mathcal{S}$ of ledgers that have their security assumptions $\mathbb{A}_i[t]$ satisfied (and hence are all considered uncorrupted). We require that:

- each ledger in $\mathcal{S}$ individually maintains both persistence and liveness;
- for each asset A, when looking at the sequence of all A-transactions σ that occurred on the ledgers in $\mathcal{S}$ (sequentialized via the merge operator), there must exist a hypothetical sequence of A-transactions τ that could have happened on the compromised ledgers, such that the merge of σ and τ would be valid according to the validity language of A.

We now proceed to formalize the above intuition.

Definition 1 (Assets, syntactically valid transactions). *For an asset* A*, we denote by* $\mathcal{T}_{\mathsf{A}}$ *the* valid transaction set of A*, i.e., the set of all syntactically valid transactions involving* A*. For a ledger* $\mathbf{L}$ *we denote by* $\mathcal{T}_{\mathbf{L}}$ *the set of transactions that can be included into* $\mathbf{L}$*. For notational convenience, we define* $\mathcal{T}_{\mathsf{A},\mathbf{L}} \triangleq \mathcal{T}_{\mathsf{A}} \cap \mathcal{T}_{\mathbf{L}}$*. Let* $\mathsf{Assets}(\mathbf{L})$ *denote the set of all assets that are supported by* $\mathbf{L}$*. Formally,* $\mathsf{Assets}(\mathbf{L}) \triangleq \{\mathsf{A} : \mathcal{T}_{\mathsf{A},\mathbf{L}} \neq \emptyset\}$*.*

We assume that each transaction pertains to a particular asset and belongs to a particular ledger, i.e., for distinct $\mathsf{A}_1 \neq \mathsf{A}_2$ and $\mathbf{L}_1 \neq \mathbf{L}_2$, we have that $\mathcal{T}_{\mathsf{A}_1} \cap \mathcal{T}_{\mathsf{A}_2} = \emptyset$ and $\mathcal{T}_{\mathbf{L}_1} \cap \mathcal{T}_{\mathbf{L}_2} = \emptyset$. However, our treatment can be easily generalized to alleviate this restriction.

We now generically characterize the *validity* of a sequence of transactions involving a particular asset. This is captured individually for each asset via a notion of an asset's *validity language*, which is simply a set of words over the alphabet of this asset's transactions. The asset's validity language is meant to capture how the asset is mandated to behave in the system. Let ε denote the empty sequence and $\|$ represent concatenation.

Definition 2 (Asset validity language). *For an asset* A*, the* asset validity language of A *is any language* $\mathbb{V}_{\mathsf{A}} \subseteq \mathcal{T}_{\mathsf{A}}^*$ *that satisfies the following properties:*

1. ***Base.*** $\varepsilon \in \mathbb{V}_{\mathsf{A}}$.
2. ***Monotonicity.*** *For any* $w, w' \in \mathcal{T}_{\mathsf{A}}^*$ *we have* $w \notin \mathbb{V}_{\mathsf{A}} \Rightarrow w \,\|\, w' \notin \mathbb{V}_{\mathsf{A}}$.
3. ***Uniqueness of transactions.*** *Words from* $\mathbb{V}_{\mathsf{A}}$ *never contain the same transaction twice: for any* $\mathsf{tx} \in \mathcal{T}_{\mathsf{A}}$ *and any* $w_1, w_2, w_3 \in \mathcal{T}_{\mathsf{A}}^*$ *we have* $w_1 \,\|\, \mathsf{tx} \,\|\, w_2 \,\|\, \mathsf{tx} \,\|\, w_3 \notin \mathbb{V}_{\mathsf{A}}$.

The first condition in the definition above is trivial, the second one mandates the natural property that if a sequence of transactions is invalid, it cannot become valid again by adding further transactions. Finally, the third condition reflects a natural "uniqueness" property of transactions in existing implementations. While not necessary for our treatment, it allows for some simplifications.

The following definition allows us to focus on a particular asset or ledger within a sequence of transactions.

Definition 3 (Ledger state projection). *Given a ledger state* L*, we call a* projection of L with respect to a set $\mathcal{X}$ *(and denote by* $\pi_{\mathcal{X}}(\mathsf{L})$*) the ledger state that is obtained from* L *by removing all transactions not in* $\mathcal{X}$*. To simplify notation, we will use* π_{A} *and* $\pi_{\mathcal{I}}$ *as a shorthand for* $\pi_{\mathcal{T}_{\mathsf{A}}}$ *and* $\pi_{\bigcup_{i\in\mathcal{I}}\mathcal{T}_{\mathbf{L}_i}}$*, denoting the projection of the transactions of a ledger state with respect to particular asset* A *or a particular set of individual ledger indices. Naturally, for a language* $\mathbb{V}$ *we define the* projected language $\pi_{\mathcal{X}}(\mathbb{V}) := \{\pi_{\mathcal{X}}(w) : w \in \mathbb{V}\}$*, which contains all the sequences of transactions from the original language, each of them projected with respect to* $\mathcal{X}$*.*

The concept of *effect transactions* below captures ledger interoperability at the syntactic level.

Definition 4 (Effect Transactions). *For two ledgers* $\mathbf{L}$ *and* $\mathbf{L}'$*, the* effect mapping *is a mapping of the form* $\mathsf{effect}_{\mathbf{L}\to\mathbf{L}'} : \mathcal{T}_{\mathbf{L}} \to (\mathcal{T}_{\mathbf{L}'} \cup \{\bot\})$*. A transaction* $\mathsf{tx}' = \mathsf{effect}_{\mathbf{L}\to\mathbf{L}'}(\mathsf{tx}) \neq \bot$ *is called the* effect transaction *of the transaction* tx*.*

Intuitively, for any transaction $\mathsf{tx} \in \mathcal{T}_{\mathbf{L}}$, the corresponding transaction $\mathsf{effect}_{\mathbf{L}\to\mathbf{L}'}(\mathsf{tx}) \in \mathcal{T}_{\mathbf{L}'} \cup \{\bot\}$ identifies the necessary effect on ledger $\mathbf{L}'$ of the event of the inclusion of the transaction tx into the ledger $\mathbf{L}$. With foresight, in an implementation of a system of ledgers where a "pegging" exists, the transaction $\mathsf{effect}_{\mathbf{L}\to\mathbf{L}'}(\mathsf{tx})$ has to be eventually valid and includable in $\mathbf{L}'$ in response to the inclusion of tx in $\mathbf{L}$. Additionally, throughout the paper we assume that an effect transaction is always clearly identifiable as such, and its corresponding "sending" transaction can be derived from it; our instantiation does have this property.

We use a special symbol $\bot$ to indicate that the transaction tx does not necessitate any action on $\mathbf{L}'$ (this will be the case for most transactions). We will now be interested mostly in transactions that *do* require an action on the other ledger.

Definition 5 (Cross-Ledger Transfers). *For two ledgers* $\mathbf{L}$ *and* $\mathbf{L}'$ *and an effect mapping* $\mathsf{effect}_{\mathbf{L}\to\mathbf{L}'}(\cdot)$*, we refer to a transaction in* $\mathcal{T}_{\mathbf{L}}$ *that requires some effect on* $\mathbf{L}'$ *as a* $(\mathbf{L}, \mathbf{L}')$-cross-ledger transfer transaction *(or* cross-ledger transfer *for short). The set of all cross-ledger transfers is denoted by* $\mathcal{T}^{\mathsf{cl}}_{\mathbf{L},\mathbf{L}'} \subseteq \mathcal{T}_{\mathbf{L}}$*, formally* $\mathcal{T}^{\mathsf{cl}}_{\mathbf{L},\mathbf{L}'} \triangleq \{\mathsf{tx} \in \mathcal{T}_{\mathbf{L}} : \mathsf{effect}_{\mathbf{L}\to\mathbf{L}'}(\mathsf{tx}) \neq \bot\}$*.*

Given ledger states $\mathsf{L}_1, \mathsf{L}_2, \ldots, \mathsf{L}_n$, we need to consider a joint ordered view of the transactions in all these ledgers. This is provided by the merge operator. Intuitively, merge allows us to create a combined view of multiple ledgers, putting all of the transactions across multiple ledgers into a linear ordering. We expect that even if certain ledgers are missing from its input, merge is still able to produce a global ordering for the remaining ledgers. With foresight, this ability of the merge operator will enable us to reason about the situation when some ledgers fail: In that case, the respective inputs to the merge function will be missing. The merge function definition below depends on the effect mappings, we keep this dependence implicit for simpler notation.

Definition 6 (Merging ledger states). *The* $\mathsf{merge}(\cdot)$ *function is any mapping taking a subset of ledger states* $\mathcal{L} \subseteq \{\mathsf{L}_1, \mathsf{L}_2, \ldots, \mathsf{L}_n\}$ *and producing a ledger state* $\mathsf{merge}(\mathcal{L})$ *such that:*

1. ***Partitioning.*** *The ledger states in* $\mathcal{L}$ *are disjoint subsequences of* $\mathsf{merge}(\mathcal{L})$ *that cover the whole sequence* $\mathsf{merge}(\mathcal{L})$*.*
2. ***Topological soundness.*** *For any* $i \neq j$ *such that* $\mathsf{L}_i, \mathsf{L}_j \in \mathcal{L}$ *and any two transactions* $\mathsf{tx} \in \mathsf{L}_i$ *and* $\mathsf{tx}' \in \mathsf{L}_j$*, if* $\mathsf{tx}' = \mathsf{effect}_{\mathbf{L}_i\to\mathbf{L}_j}(\mathsf{tx})$ *then* tx *precedes* tx' *in* $\mathsf{merge}(\mathcal{L})$*.*

We will require that our validity languages are *correct* in the following sense.

Definition 7 (Correctness of $\mathbb{V}_{\mathsf{A}}$). *A validity language* $\mathbb{V}_{\mathsf{A}}$ *is* correct *with respect to a mapping* $\mathsf{merge}(\cdot)$*, if for any ledger states* $\mathcal{L} \triangleq (\mathsf{L}_1, \ldots, \mathsf{L}_n)$ *such that* $\pi_{\mathsf{A}}(\mathsf{merge}(\mathcal{L})) \in \mathbb{V}_{\mathsf{A}}$*, indices* $i \neq j$*, and any cross-ledger transfer* $\mathsf{tx} \in \mathsf{L}_i \cap \mathcal{T}^{\mathsf{cl}}_{\mathbf{L}_i,\mathbf{L}_j}$ *such that* $\mathsf{effect}_{\mathbf{L}_i\to\mathbf{L}_j}(\mathsf{tx}) = \mathsf{tx}' \neq \bot$ *is not in* L_j*, we have*

$$\pi_{\mathsf{A}}(\mathsf{merge}(\mathsf{L}_1, \ldots, \mathsf{L}_i, \ldots, \mathsf{L}_j \,\|\, \mathsf{tx}', \ldots, \mathsf{L}_n)) \in \mathbb{V}_{\mathsf{A}} .$$

The above definition makes sure that if a cross-ledger transfer of an asset A is included into some ledger $\mathbf{L}_i$ and mandates an effect transaction on $\mathbf{L}_j$, then the inclusion of this effect transaction will be consistent with $\mathbb{V}_{\mathsf{A}}$. Note that this does not yet guarantee that the effect transaction will indeed be included into $\mathbf{L}_j$, this will be provided by the liveness of $\mathbf{L}_j$ required below.

We are now ready to give our main security definition. In what follows, we call a *system-of-ledgers protocol* any protocol run by a (possibly dynamically changing) set of parties that maintains an evolving state of n ledgers $\{\mathbf{L}_i\}_{i\in[n]}$.

Definition 8 (Pegging security). *A system-of-ledgers protocol Π for $\{\mathbf{L}_i\}_{i\in[n]}$ is* pegging-secure *with liveness parameter $u \in \mathbb{N}$ with respect to:*

- *a set of assumptions $\mathbb{A}_i$ for ledgers $\{\mathbf{L}_i\}_{i\in[n]}$,*
- *a merge mapping $\mathsf{merge}(\cdot)$,*
- *validity languages $\mathbb{V}_{\mathsf{A}}$ for each $\mathsf{A} \in \bigcup_{i\in[n]} \mathsf{Assets}(\mathbf{L}_i)$,*

if for all PPT adversaries, all slots t and for $\mathcal{S}_t \triangleq \{i : \mathbb{A}_i[t] \text{ holds}\}$ we have that except with negligible probability in the security parameter:

Ledger persistence: *For each $i \in \mathcal{S}_t$, $\mathbf{L}_i$ satisfies the persistence property.*
Ledger liveness: *For each $i \in \mathcal{S}_t$, $\mathbf{L}_i$ satisfies the liveness property parametrized by u.*
Firewall: *For all $\mathsf{A} \in \bigcup_{i\in\mathcal{S}_t} \mathsf{Assets}(\mathbf{L}_i)$,*

$$\pi_{\mathsf{A}}\left(\mathsf{merge}\left(\{\mathbf{L}_i^{\cup}[t] \,:\, i \in \mathcal{S}_t\}\right)\right) \in \pi_{\mathcal{S}_t}(\mathbb{V}_{\mathsf{A}}) \,.$$

Intuitively, the firewall property above gives the following guarantee: If the security assumption of a particular sidechain has been violated, we demand that the sequence of transactions σ that appears in the still uncompromised ledgers is a valid projection of some word from the asset validity language onto these ledgers. This means that there exists a sequence of transactions τ that *could have happened* on the compromised ledgers, such that it would "justify" the current state of the uncompromised ledgers as a valid state. Of course, we don't know whether this sequence τ actually occurred on the compromised ledger, however, given that this ledger itself no longer provides any reliable state, this is the best guarantee we can still offer to the uncompromised ledgers.

Looking ahead, when we define a particular validity language for our concrete, fungible, constant-supply asset, we will see that this property will translate into the mainchain maintaining "limited liability" towards the sidechain: the amount of money transferred back from the sidechain can never exceed the amount of money that was previously moved towards the sidechain, because no plausible history of sidechain transactions can exist that would justify such a transfer.

4 Implementing Pegged Ledgers

We present a construction for pegged ledgers that is based on Ouroboros PoS [20], but also applicable to other PoS systems such as Snow White [6] and Algorand [26] (for a discussion of such adaptations, see Appendix B). Our protocol will implement a system of ledgers with pegging security according to Definition 8 under an assumption on the relative stake power of the adversary that will be detailed below.

The main challenge in implementing pegged ledgers is to facilitate secure cross-chain transfers. We consider two approaches to such transfers and refer to them as *direct observation* or *cross-chain certification*. Consider two pegged ledgers $\mathbf{L}_1$ and $\mathbf{L}_2$. Direct observation of $\mathbf{L}_1$ means that every node of $\mathbf{L}_2$ follows and validates $\mathbf{L}_1$; it is easy to see that this enables transfers from $\mathbf{L}_1$ to $\mathbf{L}_2$. On the other hand, cross-chain certification of $\mathbf{L}_2$ means that $\mathbf{L}_1$ contains appropriate cryptographic information sufficient to validate data issued by the nodes following $\mathbf{L}_2$. This allows transfers of assets from $\mathbf{L}_2$, as long as they are certified, to be accepted by $\mathbf{L}_1$-nodes without following $\mathbf{L}_2$. The choice between direct observation and cross-chain certification can be made independently for each direction of transfers between $\mathbf{L}_1$ and $\mathbf{L}_2$, any of the 4 variants is possible (cf. Figure 1).

Another aspect of implementing pegged ledgers in the PoS context is the choice of stake distribution that underlies the PoS on each of the chains. We again consider two options, which we call *independent staking* and *merged staking*. In independent staking, blocks on say $\mathbf{L}_1$ are "produced by" coins from $\mathbf{L}_1$ (in other words, the block-creating rights on $\mathbf{L}_1$ are attributed based on the stake distribution recorded on $\mathbf{L}_1$ only). In contrast, with merged staking, blocks on $\mathbf{L}_1$ are produced either by coins on $\mathbf{L}_1$, or coins on $\mathbf{L}_2$ that have, via their staking key, declared support of $\mathbf{L}_1$ (but otherwise remain on $\mathbf{L}_1$); see Figure 1. Also here, all 4 combinations are possible.

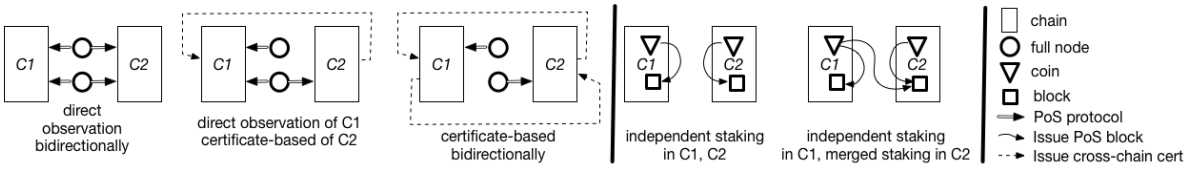

Fig. 1: Deployment options for PoS Sidechains.

In our construction we choose an exemplary configuration between two ledgers $\mathbf{L}_1$ and $\mathbf{L}_2$, so that direct observation is applied to $\mathbf{L}_1$, cross-chain certification to $\mathbf{L}_2$, independent-staking in $\mathbf{L}_1$ and merged staking in $\mathbf{L}_2$. As a result, all stakeholders in $\mathbf{L}_2$ also keep track of chain development on $\mathbf{L}_1$ (and hence run a full node for $\mathbf{L}_1$) while the opposite is not necessary, i.e., $\mathbf{L}_1$ stakeholders can be oblivious of transactions and blocks being added to $\mathbf{L}_2$. This illustrates the two basic possibilities of pegging and can be easily adapted to any other of the configurations between two ledgers in Figure 1.

In order to reflect the asymmetry between the two chains in our exemplary construction we will refer to $\mathbf{L}_1$ as the "mainchain" **MC**, and to $\mathbf{L}_2$ as the "sidechain" **SC**. To elaborate further on this concrete asymmetric use case, we also fully specify how the sidechain can be initialized from scratch, assuming that the mainchain already exists.

The pegging with the sidechain will be provided with respect to a specific asset of **MC** that will be created on **MC**. Note that **MC** as well as **SC** may carry additional assets but for simplicity we will assume that staking and pegging is accomplished only via this single primary asset.

The presentation of the construction is organized as follows. First, in Section 4.1 we introduce a novel cryptographic primitive, *ad-hoc threshold multisignature (ATMS)*, which is the fundamental building block for cross-chain certification. Afterwards, in Section 4.3 we use it as a black box to build secure pegged ledgers with respect to concrete instantiations of the functions merge and effect and a validity language $\mathbb{V}_{\mathfrak{A}}$ for asset $\mathfrak{A}$ given in Section 4.2. Finally, we discuss specific instantiations of ATMS in Section 5.

4.1 Ad-Hoc Threshold Multisignatures

We introduce a new primitive, *ad-hoc threshold multisignatures (ATMS)*, which borrow properties from multisignatures and threshold signatures and are ad-hoc in the sense that signers need to be selected on the fly from an existing key set. In Section 4.3 we describe how ATMS are useful for periodically updating the "anchor of trust" that the mainchain parties have w.r.t. the sidechain they are not following.

ATMS are parametrized by a threshold t. On top of the usual digital signatures functionality, ATMS also provide a way to: (1) aggregate the public keys of a subset of these parties into a single aggregate public key avk; (2) check that a given avk was created using the right sequence of individual public keys; and (3) aggregate $t' \geq t$ individual signatures from t' of the parties into a single aggregate signature that can then be verified using avk, which is impossible if less than t individual signatures are used.

The definition of an ATMS is given below.

Definition 9. *A t-ATMS is a tuple of algorithms $\Pi = (\mathsf{PGen}, \mathsf{Gen}, \mathsf{Sig}, \mathsf{Ver}, \mathsf{AKey}, \mathsf{ACheck}, \mathsf{ASig}, \mathsf{AVer})$ where:*

$\mathsf{PGen}(1^\kappa)$ *is the parameter generation algorithm that takes the security parameter* 1^κ *and returns system parameters* $\mathcal{P}$.

$\mathsf{Gen}(\mathcal{P})$ *is the key-generation algorithm that takes* $\mathcal{P}$ *and produces a public/private key pair* (vk_i, sk_i) *for the party invoking it.*

$\mathsf{Sig}(sk_i, m)$ *is the signature algorithm as in an ordinary signature scheme: it takes a private key and a message and produces a (so-called* local*) signature* σ.

$\mathsf{Ver}(m, pk_i, \sigma)$ *is the verification algorithm that takes a public key, a message and a signature and returns* true *or* false.

$\mathsf{AKey}(\mathcal{VK})$ *is the key aggregation algorithm that takes a sequence of public keys* $\mathcal{VK}$ *and aggregates them into an* aggregate public key *avk.*

$\mathsf{ACheck}(\mathcal{VK}, avk)$ *is the aggregation-checking algorithm that takes a public key sequence* $\mathcal{VK}$ *and an aggregate public key avk and returns* true *or* false*, determining whether* $\mathcal{VK}$ *were used to produce avk.*

$\mathsf{ASig}(m, \mathcal{VK}, \langle (vk_1, \sigma_1), \cdots, (vk_d, \sigma_d) \rangle)$ *is the signature-aggregation algorithm that takes a message* m*, a sequence of public keys* $\mathcal{VK}$ *and a sequence of* d *pairs* $\langle (vk_1, \sigma_1), \cdots, (vk_d, \sigma_d) \rangle$ *where each* σ_i *is a local signature on* m *verifiable by* vk_i *and each* vk_i *is in a distinct position within* $\mathcal{VK}$*,* ASig *combines these into a multisignature* σ *that can later be verified with respect to the aggregate public key avk produced from* $\mathcal{VK}$ *(as long as* $d \geq t$*, see below).*

$\mathsf{AVer}(m, avk, \sigma)$ *is the aggregate-signature verification algorithm that takes a message* m*, an aggregate public key avk, and a multisignature* σ*, and returns* true *or* false.

Definition 10 (ATMS correctness). *Let* Π *be a* t*-ATMS scheme initialized with* $\mathcal{P} \leftarrow \mathsf{PGen}(1^\kappa)$*, let* $(vk_1, sk_1), \cdots, (vk_n, sk_n)$ *be a sequence of keys generated via* $\mathsf{Gen}(\mathcal{P})$*, let* $\mathcal{VK}$ *be a sequence containing (not necessarily unique) keys from the above and avk be generated by invoking* $avk \leftarrow \mathsf{AKey}(\mathcal{VK})$*. Let* m *be any message and let* $\langle (vk_1, \sigma_1), \cdots, (vk_d, \sigma_d) \rangle$ *be any sequence of key/signature pairs provided that* $d \geq t$ *and every* vk_i *appears in a unique position in the sequence* $\mathcal{VK}$*, where* σ_i *is generated as* $\sigma_i = \mathsf{Sig}(sk_i, m)$*. Let* $\sigma \leftarrow \mathsf{ASig}(m, \mathcal{VK}, \langle (vk_1, \sigma_1), \cdots, (vk_d, \sigma_d) \rangle)$*. The scheme* Π *is* correct *if for every such message and sequence the following hold:*

1. $\mathsf{Ver}(m, vk_i, \sigma_i)$ *is* true *for all* i*;*
2. $\mathsf{ACheck}(\mathcal{VK}, avk)$ *is* true*;*
3. $\mathsf{AVer}(m, avk, \sigma)$ *is* true.

We define the security of an ATMS in the definition below, via a cryptographic game given in Algorithm 1.

Definition 11 (Security). *A* t*-ATMS scheme* $\Pi = (\mathsf{PGen}, \mathsf{Gen}, \mathsf{Sig}, \mathsf{Ver}, \mathsf{AKey}, \mathsf{ACheck}, \mathsf{ASig}, \mathsf{AVer})$ *is* secure *if for any PPT adversary* $\mathcal{A}$ *and any polynomial* p *there exists some negligible function* negl *such that* $\Pr[\mathsf{ATMS}_{\Pi,\mathcal{A}}(\kappa, p(\kappa)) = 1] < \mathsf{negl}(\kappa)$.

The quantity q in the ATMS game counts how many keys the adversary is in control of among her chosen keys keys which will be used for aggregate-signature verification. The sequence keys can contain both adversarially-generated keys as well as some of the keys $\mathcal{VK}$ honestly generated by the challenger. The variable q counts the number of adversarially controlled keys in keys. This includes those keys in keys for which the adversary has obtained a signature for the message in question (through the use of the oracle $\mathcal{O}^{\mathsf{sig}}(\cdot)$) or which the adversary has corrupted completely (through the use of the oracle $\mathcal{O}^{\mathsf{cor}}(\cdot)$), as well as those keys which have been generated by the adversary herself and therefore are not in $\mathcal{VK}$.

It is straightforward to see that if Π is a secure ATMS, then the tuple $(\mathsf{PGen}, \mathsf{Gen}, \mathsf{Sig}, \mathsf{Ver})$ is a $\mathsf{EUF\text{-}CMA}$-secure signature scheme.

Looking ahead, note that since the AKey algorithm is only invoked with the public keys of the participants, it can be invoked by anyone, not just the parties who hold the respective secret keys, as long as the public portion of their keys is published. Furthermore, notice that the above games allow the adversary to generate more public/private key pairs of their own and combine them at will.

Having defined the ATMS primitive, we will now describe a sidechain construction that uses it. Concrete instantiations of the ATMS primitive are presented in Section 5.

Algorithm 1 The game $\mathsf{ATMS}_{\Pi,\mathcal{A}}$

The game is parameterized by a security parameter κ and an integer $p(\kappa)$.

$\mathcal{VK} \leftarrow \epsilon; \mathcal{SK} \leftarrow \epsilon; Q^{\mathsf{sig}} \leftarrow \emptyset; Q^{\mathsf{cor}} \leftarrow \emptyset$
$\mathcal{P} \leftarrow \mathsf{PGen}(1^\kappa)$
$(m, \sigma, avk, \mathsf{keys}) \leftarrow \mathcal{A}^{\mathcal{O}^{\mathsf{gen}}, \mathcal{O}^{\mathsf{sig}}(\cdot,\cdot), \mathcal{O}^{\mathsf{cor}}(\cdot)}(\mathcal{P})$
$q \leftarrow 0$
for vk in keys **do**
  if $vk \notin \mathcal{VK} \vee vk \in Q^{\mathsf{sig}}[m] \cup Q^{\mathsf{cor}}$ **then**
    $q \leftarrow q + 1$
  end if
end for
return $\mathsf{AVer}(m, avk, \sigma) \wedge \mathsf{ACheck}(\mathsf{keys}, avk) \wedge q < t$

Algorithm 2 The oracle $\mathcal{O}^{\mathsf{gen}}$

function $\mathcal{O}^{\mathsf{gen}}$
  $(vk, sk) \leftarrow \mathsf{Gen}(\mathcal{P})$
  $\mathcal{VK} \leftarrow \mathcal{VK} \,\|\, vk$
  $\mathcal{SK} \leftarrow \mathcal{SK} \,\|\, sk$
  return vk
end function

Algorithm 3 The oracle $\mathcal{O}^{\mathsf{sig}}$

function $\mathcal{O}^{\mathsf{sig}}(i, m)$
  $Q^{\mathsf{sig}}[m] \leftarrow Q^{\mathsf{sig}}[m] \cup \{\mathcal{VK}[i]\}$
  return $\mathsf{Sig}(\mathcal{SK}[i], m)$
end function

Algorithm 4 The oracle $\mathcal{O}^{\mathsf{cor}}$

function $\mathcal{O}^{\mathsf{cor}}(i)$
  $Q^{\mathsf{cor}} \leftarrow Q^{\mathsf{cor}} \cup \{\mathcal{VK}[i]\}$
  return $\mathcal{SK}[i]$
end function

Fig. 2: The ATMS security game $\mathsf{ATMS}_{\Pi,\mathcal{A}}$.

4.2 A Concrete Asset $\mathfrak{A}$

We now present an example of a simple fungible asset with fixed supply, which we denote $\mathfrak{A}$, and describe its validity language $\mathbb{V}_{\mathfrak{A}}$. This will be the asset (and validity language) considered in our construction and proof. While $\mathbb{V}_{\mathfrak{A}}$ is simple and natural, it allows us to exhibit the main features of our security treatment and illustrate how it can be applied to more complex languages such as those capable of capturing smart contracts; we omit such extensions in this version. Note that our language is account-based, but a UTXO-based validity language can be considered in a similar manner.

Instantiating $\mathbb{V}_{\mathbf{A}}$. The validity language $\mathbb{V}_{\mathfrak{A}}$ for the asset $\mathfrak{A}$ considers two ledgers: the mainchain ledger $\mathbf{L}_0 \triangleq \mathbf{MC}$ and the sidechain ledger $\mathbf{L}_1 \triangleq \mathbf{SC}$. For this asset, every transaction $\mathsf{tx} \in \mathcal{T}_{\mathfrak{A}}$ has the form $\mathsf{tx} = (\mathsf{txid}, \mathsf{lid}, (\mathsf{send}, \mathsf{sAcc}), (\mathsf{rec}, \mathsf{rAcc}), v, \sigma)$, where:

- txid is a transaction identifier that prevents replay attacks. We assume that txid contains sufficient information to identify lid by inspection and that this is part of syntactic transaction validation.
- $\mathsf{lid} \in \{0, 1\}$ is the ledger index where the transaction belongs.
- $\mathsf{send} \in \{0, 1\}$ is the index of the sender ledger $\mathbf{L}_{\mathsf{send}}$ and sAcc is an account on this ledger, this is the sender account. For simplicity, we assume that sAcc is the public key of the account.
- $\mathsf{rec} \in \{0, 1\}$ is the index of the recipient ledger $\mathbf{L}_{\mathsf{rec}}$ and rAcc is an account (again represented by a public key) on this ledger, this is the recipient account. We allow either $\mathbf{L}_{\mathsf{send}} = \mathbf{L}_{\mathsf{rec}}$, which denotes a *local transaction*, or $\mathbf{L}_{\mathsf{send}} \neq \mathbf{L}_{\mathsf{rec}}$, which denotes a *remote transaction* (i.e., a cross-ledger transfer).
- v is the amount to be transferred.
- σ is the signature of the sender, i.e. made with the private key corresponding to the public key sAcc on the plaintext $(\mathsf{txid}, (\mathsf{send}, \mathsf{sAcc}), (\mathsf{rec}, \mathsf{rAcc}), v)$.

The correctness of lid is enforced by the ledgers, i.e., for both $i \in \{0, 1\}$ the set $\mathcal{T}_{\mathfrak{A},\mathbf{L}_i}$ only contains transactions with $\mathsf{lid} = i$. Note that although we sometimes notationally distinguish between an account and the public

key that is associated with it, for simplicity we will assume that these are either identical or can always be derived from one another (this assumption is not essential for our construction).

The membership-deciding algorithm for $\mathbb{V}_{\mathfrak{A}}$ is presented in Algorithm 5. It processes the sequence of transactions $(\mathsf{tx}_1, \mathsf{tx}_2, \ldots, \mathsf{tx}_m)$ given to it as input in their order. Assuming transactions are syntactically valid, the function verifies for each transaction tx_i the freshness of txid, validity of the signature, and availability of sufficient funds on the sending account. For an intra-ledger transaction (i.e., one that has $\mathsf{send} = \mathsf{rec}$), these are all the performed checks.

More interestingly, $\mathbb{V}_{\mathfrak{A}}$ also allows for cross-ledger transfers. Such transfers are expressed by a pair of transactions in which $\mathsf{send} \neq \mathsf{rec}$. The first transaction appears in $\mathsf{lid} = \mathsf{send}$, while the second transaction appears in $\mathsf{lid} = \mathsf{rec}$. The two transactions are identical except for this change in lid (this is the only exception to the txid-freshness requirement). Every receiving transaction has to be preceded by a matching sending transaction. Cross-chain transactions have to, similarly to intra-ledger transactions, conform to laws of balance conservation.

Note that $\mathbb{V}_{\mathfrak{A}}$ does not require that every "sending" cross-ledger transaction on the sender ledger is matched by a "receiving" transaction on the receiving ledger. Hence, if the asset $\mathfrak{A}$ is sent from ledger $\mathbf{L}_{\mathsf{send}}$ but has not yet arrived on $\mathbf{L}_{\mathsf{rec}}$ then validity for this asset is *not* violated. All the validity language ensures is that appending the $\mathsf{sidechain\_receive}$ transaction to the rec will eventually be a valid way to extend the receiving ledger, as long as the $\mathsf{sidechain\_send}$ transaction has been included in send.

Instantiating $\mathsf{effect}_{\mathbf{L}_i \to \mathbf{L}_j}$. For the simple asset $\mathfrak{A}$ outlined above, every cross-ledger transfer is a "sending" transaction tx with $\mathbf{L}_{\mathsf{lid}} = \mathbf{L}_{\mathsf{send}} \neq \mathbf{L}_{\mathsf{rec}}$ appearing in $\mathbf{L}_{\mathsf{send}}$, and its effect transaction is a "receiving" transaction tx' with $\mathbf{L}_{\mathsf{lid}} = \mathbf{L}_{\mathsf{rec}} \neq \mathbf{L}_{\mathsf{send}}$ in $\mathbf{L}_{\mathsf{rec}}$ that is otherwise identical (except for the different $\mathsf{lid}' = 1 - \mathsf{lid}$). Hence, we define $\mathsf{effect}_{\mathbf{L}_{\mathsf{send}} \to \mathbf{L}_{\mathsf{rec}}}(\mathsf{tx}) = \mathsf{tx}'$ exactly for all these transactions and no other.

Instantiating $\mathsf{merge}(\cdot)$. It is easy to construct a canonical function $\mathsf{merge}(\cdot)$ once we see its inputs not only as ledger states (i.e., sequences of transactions) but we also exploit the additional structure of the blockchains carrying those ledgers. The *canonical merge* of the set of ledger states $\mathcal{L}$ is the lexicographically minimum topologically sound merge, in which transactions of ledger L_i are compared favourably to transactions in L_j if $i < j$. However, note that the construction we provide below will work for any topologically sound merge function.

One can easily observe the following statement.

Proposition 1. *The validity language $\mathbb{V}_{\mathfrak{A}}$ is correct (according to Definition 7) with respect to the* merge *function defined above.*

4.3 The Sidechain Construction

We now describe the procedures for running a sidechain in the configuration outlined at the beginning of this section: with independent staking on **MC** and merged staking on **SC**; direct observation of **MC** and cross-chain certification of **SC**. We describe the sidechain's creation, maintenance, and the way assets can be transferred to it and back. The protocol we describe below is quite complex, we hence choose to describe different parts of the protocol in differing levels of detail. This level is always chosen with the intention to allow the reader to easily fill in the details. A graphical depiction of our construction that can serve as a reference is given in Figure 3.

Notation. Where applicable, we denote the analogues of the mainchain objects on the sidechain with an additional overline. In our pseudocode, we use the statement "**post** tx **to** $\mathbf{L}$" to refer to the action of broadcasting the transaction tx to the maintainers of the ledger $\mathbf{L}$ so that they include it in the ledger eventually as prescribed by the protocol. Unless indicated otherwise, we also denote by MC (resp. SC) the

Algorithm 5 The transaction sequence validator (membership-deciding algorithm for $\mathbb{V}_{\mathfrak{A}}$).

```
1: function valid-seq(tx)
2:     BALANCE ← Initial stake distribution; seen ← ∅
3:     ▷ Traverse transactions in order
4:     for tx ∈ tx do
5:         ▷ Destructure tx into its constituents
6:         (txid, lid, (send, sAcc), (rec, rAcc), v, σ) ← tx
7:         if ¬valid(σ) then
8:             return false
9:         end if
10:        if lid = send then
11:            ▷ Replay protection
12:            if seen[txid] ≠ 0 then
13:                return false
14:            end if
15:            ▷ Law of conservation
16:            if BALANCE[send][sAcc] − v < 0 then
17:                return false
18:            end if
19:        else
20:            ▷ The case lid = rec ≠ send
21:            if seen[txid] ≠ 1 then
22:                return false
23:            end if
24:            ▷ Cross-ledger validity
25:            tx′ ← effect^{-1}_{L_{(1−lid)}→L_lid}(tx)
26:            if tx′ has not appeared before then
27:                return false
28:            end if
29:        end if
30:        if seen[txid] = 0 then
31:            ▷ Update sender balance when money departs
32:            BALANCE[send][sAcc] −= v
33:        end if
34:        ▷ Update receiver balance when money arrives
35:        if (seen[txid] = 0 ∧ send = rec)∨
              (seen[txid] = 1 ∧ send ≠ rec)   then
36:            BALANCE[rec][rAcc] += v
37:        end if
38:        seen[txid]+ = 1
39:    end for
40:    return true
41: end function
```

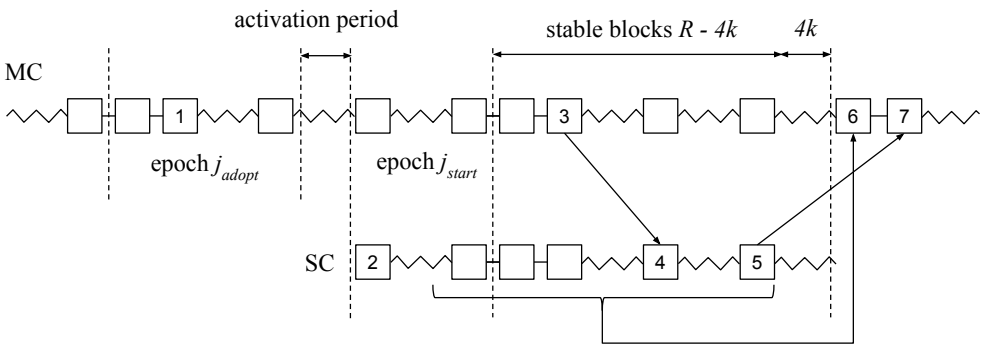

Fig. 3: Our sidechain construction. Blocks are shown as rectangles. Adjacent blocks connect with straight lines. Squiggly lines indicate some blocks are omitted. **MC** is at the top, **SC** at the bottom. Epochs are separated by dashed lines. $e_{j_{\text{adopt}}}$ is the epoch of first signalling; $e_{j_{\text{start}}}$ is the activation epoch. Blocks of interest: 1. The first block signalling **SC** awareness; 2. The **SC** genesis block; 3. A $\mathsf{tx}_{\mathsf{send}}$ transaction for a deposit; 4. A $\mathsf{tx}_{\mathsf{rec}}$ transaction for a deposit; 5. A $\mathsf{tx}_{\mathsf{send}}$ transaction for withdrawal; 6. A sc_cert transaction signalling trust transition within **SC** and certifying pending withdrawals; 7. A $\mathsf{tx}_{\mathsf{rec}}$ transaction for withdrawal, certified in a sc_cert transaction e.g. in block 6.

current *ledger state* of the ledger **MC** (resp. **SC**) as viewed by the party executing the protocol. Similarly, we denote by $\mathsf{C}_{\mathbf{MC}}$ (resp. $\mathsf{C}_{\mathbf{SC}}$) the currently held *chain* corresponding to the ledger **MC** (resp. **SC**). Hence, for example MC always represents the state stored in the stable part of the chain $\mathsf{C}_{\mathbf{MC}}$.

Helper Transactions and Data. The construction uses a set of *helper transactions* which can be included in both blockchains, but do not get reported in the respective ledgers. These helper transactions store the appropriate metadata which is implementation-specific and allow the pegging functionality to be maintained. The transaction types sidechain_support, sidechain_certificate, sidechain_success and sidechain_failure, whose nature will be detailed later, are of this kind. Moreover, our concrete implementation of pegged ledgers extends certain transactions with additional information (such as Merkle-tree inclusion proofs) that are, for convenience, understood to be stripped off these transactions when the blockchain is interpreted as a ledger.

Initialisation. The creation of a new sidechain **SC** starts by any of the stakeholders of the mainchain adopting the code that implements the sidechain. This action does not require the stakeholders to put stake on the sidechain but merely to run the code to support it (e.g. by installing a pluggable module into their client software). In the following this is referred to as "adopting the sidechain" and captured by the predicate SidechainAdoption. The adoption is announced at the mainchain by a special transaction detailed below. Each sidechain is identified by a unique identifier $\mathsf{id}_{\mathbf{SC}}$.

Let j_{adopt} denote the epoch on **MC** when the first adoption transaction has appeared; the sidechain **SC** – if its activation succeeds as discussed below – will start at the beginning of some later epoch j_{start} and will have its slots and epochs synchronized with **MC**. The software module implementing the sidechain comes with a set of deterministic rules describing the requirements for the successful activation of the sidechain, as well as for determining j_{start}. These rules are sidechain-specific and are captured in a predicate ActivationSuccess and a function ActivationEpoch, respectively. One typical such example is the following: the sidechain starts at the beginning of **MC**-epoch j_{start} for the smallest j_{start} that satisfies: (i) $j_{\mathsf{start}} - j_{\mathsf{adopt}} > c_1$; (ii) at least c_2-fraction of stake on **MC** is controlled by stakeholders that have adopted **SC**; for some constants c_1,c_2. Additionally, if such a successful activation does not occur until a failure condition captured by a predicate ActivationFailure is met (e.g. until a predetermined period of $c_3 > c_1$ epochs has passed), the sidechain initialization is aborted.

The activation process then follows the steps outlined below, the detailed description is given in Algorithm 6).

Algorithm 6 Sidechain initialisation procedures.

The algorithm is run by every stakeholder U that adopted the sidechain. We denote by (vk, sk) its public and private keys.

upon $\mathsf{SidechainAdoption}(\mathsf{id}_{\mathbf{SC}})$ **do**
  $\mathsf{sidechain\_state}[\mathsf{id}_{\mathbf{SC}}] \leftarrow \mathsf{initializing}$
  $(vk', sk') \leftarrow \mathsf{Gen}(\mathcal{P})$
  $\sigma \leftarrow \mathsf{Sig}_{sk}(\mathsf{sidechain\_support}, \mathsf{id}_{\mathbf{SC}}, vk, vk')$
  post $(\mathsf{sidechain\_support}, \mathsf{id}_{\mathbf{SC}}, vk, vk', \sigma)$ **to MC**
end upon
upon MC.NewEpoch() **do**
  $j \leftarrow$ **MC**.EpochIndex()
  if $\mathsf{sidechain\_state}[\mathsf{id}_{\mathbf{SC}}] = \mathsf{initializing}$ **then**
    if ActivationFailure() **then**
      $\mathsf{sidechain\_state}[\mathsf{id}_{\mathbf{SC}}] \leftarrow \mathsf{failed}$
      post $\mathsf{sidechain\_failure}(\mathsf{id}_{\mathbf{SC}})$ **to MC**
    else if ActivationSuccess() **then**
      $\mathsf{sidechain\_state}[\mathsf{id}_{\mathbf{SC}}] \leftarrow \mathsf{initialized}$
      $j_{\mathsf{start}} \leftarrow \mathsf{ActivationEpoch}()$
      Post $\mathsf{sidechain\_success}(\mathsf{id}_{\mathbf{SC}})$ to **MC**
    end if
  end if
  if $\mathsf{sidechain\_state}[\mathsf{id}_{\mathbf{SC}}] = \mathsf{initialized} \wedge j = j_{\mathsf{start}}$ **then**
    $\bar{\eta}_{j_{\mathsf{start}}} \leftarrow H(\mathsf{id}_{\mathbf{SC}}, \eta_{j_{\mathsf{start}}})$
    $\mathcal{VK}_{j_{\mathsf{start}}} \leftarrow 2k$ last slot leaders of $e_{j_{\mathsf{start}}}$ in **SC**
    $avk^{j_{\mathsf{start}}} \leftarrow \mathsf{AKey}(\mathcal{VK}_{j_{\mathsf{start}}})$
    $\overline{\mathsf{G}} \leftarrow \left(\mathsf{id}_{\mathbf{SC}}, \overline{\mathsf{SD}}_{j_{\mathsf{start}}}, \bar{\eta}_{j_{\mathsf{start}}}, \mathcal{P}, avk^{j_{\mathsf{start}}}\right)$
    $\mathsf{C}_{\mathbf{SC}} \leftarrow (\overline{\mathsf{G}})$
  end if
end upon

First, every stakeholder U_i of **MC** (holding a key pair (vk, sk)) that supports the sidechain posts a special transaction $(\mathsf{sidechain\_support}, \mathsf{id}_{\mathbf{SC}}, vk, vk')$, signed by sk into the mainchain. Here vk' is a public key from an ATMS key pair freshly generated by U_i; its role is explained in Section 4.3 below.

If the sidechain activation succeeds, then during the first slot of epoch j_{start} the stakeholders of **MC** that support **SC** construct the genesis block $\overline{\mathsf{G}} = (\mathsf{id}_{\mathbf{SC}}, \overline{\mathsf{SD}}_{j_{\mathsf{start}}}, \bar{\eta}_{j_{\mathsf{start}}} \triangleq H(\mathsf{id}_{\mathbf{SC}}, \eta_{j_{\mathsf{start}}}), \mathcal{P}, avk^{j_{\mathsf{start}}})$ for **SC**. $\eta_{j_{\mathsf{start}}}$ is the randomness for leader election on **MC** in epoch j_{start} (derived on **MC** in epoch $j_{\mathsf{start}} - 1$). It is reused to compute the initial sidechain randomness $\bar{\eta}_{j_{\mathsf{start}}}$ as well, further $\bar{\eta}_{j'}$ for $j' > j_{\mathsf{start}}$ are determined independently on **SC** using the Ouroboros coin-tossing protocol.[8] Furthermore, $\mathcal{P}$ and $avk^{j_{\mathsf{start}}}$ are public parameters and an aggregated public key of an ATMS scheme; their creation and role is discussed in Section 4.3 below. Note that $\overline{\mathsf{G}}$ is defined mostly for notational compatibility, as $\overline{\mathsf{SD}}_{j_{\mathsf{start}}}$ is empty at this point anyway. $\overline{\mathsf{G}}$ can be constructed as soon as $\eta_{j_{\mathsf{start}}}$ is known and stable.

The stakeholders that adopted **SC** post into **MC** a transaction $\mathsf{sidechain\_success}(\mathsf{id}_{\mathbf{SC}})$ to signify that **SC** has been initialized. If the sidechain creation expires, then, after the first block of the next epoch after expiration occurs, the stakeholders of **MC** that supported **SC** post the transaction $\mathsf{sidechain\_failure}(\mathsf{id}_{\mathbf{SC}})$ to **MC**. We assume that both predicates ActivationSuccess and ActivationFailure can be evaluated based on the state of **MC** only, and hence spurious success/failure transactions will be considered invalid.

[8] This can be interpreted as using **MC** to implement the setup functionality needed to bootstrap **SC**.

Maintenance. Once the sidechain is created, both the mainchain and the sidechain need to be maintained by their respective set of stakeholders (detailed below) running their respective instance of the Ouroboros protocol.

In the case of the mainchain, the maintenance procedure is given in Algorithm 7. This algorithm is run by all stakeholders controlling stake that is recorded on the mainchain. Each stakeholder, on every new slot, collects all the candidate **MC**-chains from the network (modelled via the Diffuse functionality) and filters them for both consensus-level validity (using $\mathbf{MC}$.ValidateConsensusLevel) and transaction validity (using the $\text{VERIFIER}_{\mathbf{MC}}$ predicate given in Algorithm 8). Out of the remaining valid chains, he chooses his new state $\mathsf{C}_{\mathbf{MC}}$ via PickWinningChain. Then the stakeholder evaluates whether he is an eligible leader for this slot, basing its selection on the stake distribution SD_j and randomness η_j, which are determined once per epoch in accordance with the Ouroboros protocol. If the stakeholder finds out he is a slot leader, he creates a new block B by including all transactions currently valid with respect to $\mathsf{C}_{\mathbf{MC}}$ (as per the predicate $\text{VERIFYTX}_{\mathbf{MC}}$ given also in Algorithm 8), appends it to the chain $\mathsf{C}_{\mathbf{MC}}$ and diffuses the result[9] for other parties to adopt.

Algorithm 7 Mainchain maintenance procedures.

The algorithm is run by every stakeholder U with stake on **MC** in every epoch $j \geq j_{\mathsf{start}}$, sk denotes the secret key of U. An analogous mainchain-maintaining procedure was running also before j_{start} and is omitted.

upon $\mathbf{MC}$.NewSlot() **do**
  $sl \leftarrow \mathbf{MC}$.SlotIndex()
  ▷ *First slot of a new epoch*
  if $sl \mod R = 1$ **then**
    $j \leftarrow \mathbf{MC}$.EpochIndex()
    $\mathsf{SD}_j \leftarrow \mathbf{MC}$.GetDistr($j$)
    $\eta_j \leftarrow \mathbf{MC}$.GetRandomness($j$)
  end if
  $\mathcal{C} \leftarrow$ chains received via Diffuse
  ▷ *Consensus-level validation*
  $\mathcal{C}_{\mathsf{valid}} \leftarrow \mathsf{Filter}(\mathcal{C}, \mathbf{MC}.\mathsf{ValidateConsensusLevel})$
  ▷ *Transaction-level validation*
  $\mathcal{C}_{\mathsf{validtx}} \leftarrow \mathsf{Filter}(\mathcal{C}_{\mathsf{valid}}, \text{VERIFIER}_{\mathbf{MC}}(\cdot))$
  ▷ *Apply chain selection rule*
  $\mathsf{C}_{\mathbf{MC}} \leftarrow \mathbf{MC}.\mathsf{PickWinningChain}(\mathsf{C}_{\mathbf{MC}}, \mathcal{C}_{\mathsf{validtx}})$
  ▷ *Decide slot leadership based on* SD_j *and* η_j
  if $\mathbf{MC}.\mathsf{SlotLeader}(U, j, sl, \mathsf{SD}_j, \eta_j)$ **then**
    $\mathsf{prev} \leftarrow H(\mathsf{C}_{\mathbf{MC}}[-1])$
    $\mathbf{tx}_{\mathsf{state}} \leftarrow$ transaction sequence in $\mathsf{C}_{\mathbf{MC}}$
    $\mathbf{tx} \leftarrow$ current transactions in mempool
    $\mathbf{tx}_{\mathsf{valid}} \leftarrow \text{VERIFYTX}_{\mathbf{MC}}(\mathbf{tx}_{\mathsf{state}} \,\|\, \mathbf{tx})[|\mathbf{tx}_{\mathsf{state}}| :]$
    $\sigma \leftarrow \mathsf{Sig}_{sk}(\mathsf{prev}, \mathbf{tx}_{\mathsf{valid}})$
    $B \leftarrow (\mathsf{prev}, \mathbf{tx}_{\mathsf{valid}}, \sigma)$
    $\mathsf{C}_{\mathbf{MC}} \leftarrow \mathsf{C}_{\mathbf{MC}} \,\|\, B$
    Diffuse($\mathsf{C}_{\mathbf{MC}}$)
  end if
end upon

The maintenance procedure for **SC** is similar, hence we only describe here how it differs from Algorithm 7. Most importantly, it is executed by all stakeholders who have adopted **SC**, irrespectively of whether they own any stake on **SC**. Recall that the slots and epochs of the **SC**-instance of Ouroboros are aligned with the slots and epochs of **MC**.

[9] As in [20,13], we simplify our presentation by diffusing the complete chains, although a practical implementation would only diffuse the block B.

Algorithm 8 The **MC** verifier.

```
1: function VERIFYTX_MC(tx)
2:     bal ← initial stake; avk ← initial aggregate key
3:     seen ← ∅; pool ← 0; pfs_mtrs ← ∅; pfs_used ← ∅
4:     for tx ∈ tx do
5:         if type(tx) = sc_cert then
6:             (m, σ) ← tx
7:             if ¬AVer(m, avk, σ) then
8:                 continue
9:             end if
10:            (txs_root, avk′) ← m
11:            avk ← avk′
12:            pfs_mtrs[txs_root] ← true
13:        else
14:            (txid, lid, (send, sAcc), (rec, rAcc), v, σ) ← tx
15:            m ← (txid, lid, (send, sAcc), (rec, rAcc), v)
16:            if ¬Ver(m, sAcc, σ) ∨ seen[txid] ≠ 0 then
17:                continue
18:            end if
19:            if lid = send then
20:                if bal[sAcc] − v < 0 then
21:                    continue
22:                end if
23:                bal[sAcc] −= v
24:            else if send ≠ rec then
25:                π ← tx.π
26:                (mtr, inclusion_pf) ← π
27:                if π ∈ pfs_used ∨ mtr ∉ pfs_mtrs∨ ¬MTR-VER(mtr, inclusion_pf) then
28:                    continue
29:                end if
30:            end if
31:            if lid = rec then
32:                bal[rAcc] += v
33:            end if
34:            if send ≠ rec then
35:                if lid = send then
36:                    pool −= v
37:                else
38:                    pool += v
39:                end if
40:            end if
41:        end if
42:        seen ← seen ‖ tx
43:    end for
44:    return seen
45: end function
46: function VERIFIER_MC(C_mc)
47:    tx ← ∅
48:    for B ∈ C_mc do
49:        for tx ∈ B do
50:            tx ← tx ‖ tx
51:        end for
52:    end for
53:    return tx ≠ VERIFYTX_MC(tx)
54: end function
```

Algorithm 9 The **SC** transaction verifier.

```
1: function VERIFYTX_SC(tx)
2:     bal[MC] ← Initial MC stake
3:     bal[SC] ← Initial SC stake
4:     mc_outgoing_tx ← ∅; seen ← ∅
5:     for tx ∈ tx do
6:         (txid, lid, (send, sAcc), (rec, rAcc), v, σ, t) ← tx
7:         m ← (txid, lid, (send, sAcc), (rec, rAcc), v)
8:         if ¬Ver(sAcc, m, σ) ∨ seen[txid] ≠ 0 then
9:             continue
10:        end if
11:        if lid = send then
12:            if bal[send][sAcc] − v < 0 then
13:                continue
14:            end if
15:            if lid = MC ∧ send ≠ rec then
16:                mc_outgoing_tx[txid] ← t + 2k
17:            end if
18:        end if
19:        if lid = rec then
20:            if send ≠ rec then
21:                ▷ Effect pre-image tx immature
22:                if t < mc_outgoing_tx[txid] then
23:                    continue
24:                end if
25:            end if
26:            bal[rec][rAcc] += v
27:        end if
28:        if lid = send then
29:            bal[send][sAcc] −= v
30:        end if
31:        seen ← seen ‖ tx
32:    end for
33:    return seen
34: end function
```

Algorithm 10 The **SC** verifier.

```
1: function VERIFIER_SC(C_sc, C_mc)
2:     tx ← ANNOTATETX_SC(C_sc, C_mc)
3:     return tx ≠ VERIFYTX_SC(tx)
4: end function
```

Algorithm 11 The **SC** transaction annotation.

```
1: function ANNOTATETX_SC(C_sc, C_mc)
2:     tx ← ∅
3:     for each time slot t do
4:         tx′ ← ϵ
5:         if C_sc has a block generated at slot t then
6:             B ← the block in C_sc generated at t
7:             for tx ∈ B do
8:                 tx′ ← tx′ ‖ tx
9:             end for
10:        end if
11:        if C_mc has a block generated at slot t then
12:            B ← the block in C_mc generated at t
13:            for tx ∈ B do
14:                tx′ ← tx′ ‖ tx
15:            end for
16:        end if
17:        for tx ∈ tx′ do
18:            ▷ Mark the time of each tx in tx′
19:            tx.t ← t
20:        end for
21:        tx ← tx ‖ tx′
22:    end for
23:    return tx
24: end function
```

The first difference is that all ocurrences of **MC** and $\mathsf{C}_{\mathbf{MC}}$ are naturally replaced by **SC** and $\mathsf{C}_{\mathbf{SC}}$, respectively. This also means that the validity of received chains (resp. transactions), determined on line 13 (resp. 21), is decided based on predicate $\text{VERIFIER}_{\mathbf{SC}}(\cdot, \mathsf{C}_{\mathbf{MC}})$ (resp. $\text{VERIFYTX}_{\mathbf{SC}}(\cdot)$) instead of the predicate $\text{VERIFIER}_{\mathbf{MC}}(\cdot)$ (resp. $\text{VERIFYTX}_{\mathbf{MC}}(\cdot)$). Additionally, note that $\text{VERIFYTX}_{\mathbf{SC}}$ must be called with a sequence of transactions containing both the transactions in **SC** as well as the transactions in **MC** interspersed and timestamped, similarly to the way done in Line 2 of Algorithm 10. This is straightforward to implement, as the sidechain maintainers also directly observe the mainchain. The predicates $\text{VERIFYTX}_{\mathbf{SC}}$ and $\text{VERIFIER}_{\mathbf{SC}}$ are given in Algorithms 9 and 10, respectively.

Second, instead of the stake distribution SD_j determined on line 6, a different distribution $\overline{\mathsf{SD}}^*_j$ is determined to be used for slot leader selection in the j-th epoch of the sidechain. The distribution $\overline{\mathsf{SD}}^*$ contains all stake belonging to stakeholders that have adopted **SC**, irrespectively of whether this stake is located on **MC** or **SC** (we call such stake **SC**-*aware*). It can be obtained by combining the distribution $\overline{\mathsf{SD}}$ as recorded in **SC** with the distribution of **SC**-aware stake on **MC** (which is known to **SC**-maintainers via direct observation of **MC**). Note that the distribution used for epoch j reflects the stake distribution of **SC**-aware stake in the

past, namely by slot $4k$ of epoch $j-1$, just as in **MC**. Naturally, this also implies that the fourth parameter for the $\mathsf{SlotLeader}$ predicate on line 17 is $\overline{\mathsf{SD}}_j^*$ instead of SD_j.

Finally, the block construction procedure on line 23 is adjusted so that in the last $2k$ slots of each epoch, the created blocks on the sidechain also contain an additional ATMS signature of a so-called sidechain certificate (how this certificate is constructed and used will be described below). Hence, whenever $sl \mod R > 10k$, line 23 is replaced by $B \leftarrow (\mathsf{prev}, \mathbf{tx}_{\mathsf{valid}}, \sigma, \sigma_{\mathsf{sc\_cert}_{j+1}})$ where $\sigma_{\mathsf{sc\_cert}_{j+1}} = \mathsf{Sig}_{sk}(\mathsf{sc\_cert}_{j+1})$ and j is the current epoch index.

Depositing to SC. Once **SC** is initialized, cross-chain transfers to it can be made from **MC**. A cross-chain transfer operation in this case consists of two transactions $\mathsf{tx}_{\mathsf{send}}$ and $\mathsf{tx}_{\mathsf{rec}}$ that both have $\mathsf{send} = \mathbf{MC}$, $\mathsf{rec} = \mathbf{SC}$, and all other fields are also identical, except that each tx_i for $i \in \{\mathsf{send}, \mathsf{rec}\}$ contains $\mathsf{lid} = i$. The *sending transaction* $\mathsf{tx}_{\mathsf{send}}$ is meant to be included in **MC**, while the *receiving transaction* $\mathsf{tx}_{\mathsf{rec}}$ is meant to be included in **SC**.

Whenever a stakeholder on **MC** that has adopted **SC** wants to transfer funds to **SC**, she diffuses $\mathsf{tx}_{\mathsf{send}}$ with the correct receiving account on **SC** and the desired amount. Honest slot leaders in **MC** include these transactions into their blocks just like any intra-chain transfer transactions. Maintainers of **MC** keep account of a variable $\mathsf{pool}_{\mathbf{SC}}$, initially set to zero. Whenever a $\mathsf{tx}_{\mathsf{send}}$ is included into **MC**, they increase $\mathsf{pool}_{\mathbf{SC}}$ by the amount of this transaction.

When $\mathsf{tx}_{\mathsf{send}}$ becomes stable in **MC** (i.e., appears in MC, this happens at most $2k$ slots after its inclusion), the stakeholder creates and diffuses the corresponding $\mathsf{tx}_{\mathsf{rec}}$ which credits the respective amount of coins to rAcc in **SC**, to be included into **SC**. In practice, this is akin to a coinbase transaction, as the money was not transferred from an existing **SC** account.

Note that depositing from **MC** to **SC** is relatively fast; it merely requires a reliable inclusion of $\mathsf{tx}_{\mathsf{send}}$ into **MC** and consequently of $\mathsf{tx}_{\mathsf{rec}}$ into **SC**, as guaranteed by the liveness of the underlying Ouroboros instances. The depositing algorithm code is shown in Algorithm 12.

Algorithm 12 Depositing from **MC** to **SC**.

The algorithm is run by a stakeholder U in control of the secret key sk corresponding to the account sAcc on **MC**.

function $\mathsf{Send}(\mathsf{sAcc}, \mathsf{rAcc}, v)$ ▷ Send v from sAcc on **MC** to rAcc on **SC**
  $\mathsf{txid} \xleftarrow{\$} \{0,1\}^k$
  $\sigma \leftarrow \mathsf{Sig}_{sk}(\mathsf{txid}, \mathbf{MC}, (\mathbf{MC}, \mathsf{sAcc}), (\mathbf{SC}, \mathsf{rAcc}), v)$
  $\mathsf{tx}_{\mathsf{send}} \leftarrow (\mathsf{txid}, \mathbf{MC}, (\mathbf{MC}, \mathsf{sAcc}), (\mathbf{SC}, \mathsf{rAcc}), v, \sigma)$
  post $\mathsf{tx}_{\mathsf{send}}$ **to MC**
end function
function $\mathsf{Receive}(\mathsf{txid}, \mathsf{sAcc}, \mathsf{rAcc}, v)$
  wait until $\mathsf{tx}_{\mathsf{send}} \in \mathsf{MC}$ ▷ MC is the stable part of **MC**
  $\sigma \leftarrow \mathsf{Sig}_{sk}(\mathsf{txid}, \mathbf{SC}, (\mathbf{MC}, \mathsf{sAcc}), (\mathbf{SC}, \mathsf{rAcc}), v)$
  $\mathsf{tx}_{\mathsf{rec}} \leftarrow (\mathsf{txid}, \mathbf{SC}, (\mathbf{MC}, \mathsf{sAcc}), (\mathbf{SC}, \mathsf{rAcc}), v, \sigma)$
  post $\mathsf{tx}_{\mathsf{rec}}$ **to SC**
end function

Withdrawing to MC. The withdrawal operation is more cumbersome than the depositing operation since not all nodes of **MC** have adopted (i.e., are aware of and follow) the sidechain **SC**. As transactions, the withdrawals have the same structure as deposits, consisting of $\mathsf{tx}_{\mathsf{send}}$ and $\mathsf{tx}_{\mathsf{rec}}$, with the only difference that now they both have $\mathsf{send} = \mathbf{SC}$ and $\mathsf{rec} = \mathbf{MC}$. The sending transaction will be handled in the same way as in the case of deposits, but the receiving transaction requires a different certificate-based treatment, as detailed below.

Whenever a stakeholder in **SC** wishes to withdraw coins from **SC** to **MC**, she creates and diffuses the respective transaction $\mathsf{tx}_{\mathsf{send}}$ with the correct transfer details as before. If $\mathsf{tx}_{\mathsf{send}}$ is included in a block that belongs in one of the first $R-4k$ slots of some epoch then let j_{send} denote the index of this epoch, otherwise let j_{send} denote the index of the following epoch. The stakeholder then waits for the end of the epoch $e_{j_{\mathsf{send}}}$ to pass and $e_{j_{\mathsf{send}}+1}$ to begin.

At the beginning of $e_{j_{\mathsf{send}}+1}$, a special transaction called *sidechain certificate* $\mathsf{sc\_cert}_{j_{\mathsf{send}}+1}$ is generated by the maintainers of **SC**. It contains: (i) a Merkle-tree commitment to all withdrawal transactions $\mathsf{tx}_{\mathsf{send}}$ that were included into **SC** during last $4k$ slots of epoch $j_{\mathsf{send}}-1$ and the first $R-4k$ slots of epoch j_{send} (as these all are already stable by slot $R-2k$ of epoch j_{send}); (ii) other information allowing the maintainers of **MC** to inductively validate the certificate in every epoch. The construction of $\mathsf{sc\_cert}$ is detailed below, for now assume that the transaction provides a proof that the included information about withdrawal transactions is correct. The transaction $\mathsf{sc\_cert}$ is broadcast into the **MC** network to be included into **MC** at the beginning of $e_{j_{\mathsf{send}}+1}$ by the first honest slot leader.

The stakeholder who wishes to withdraw their money into **MC** now creates and diffuses the transaction $\mathsf{tx}_{\mathsf{rec}}$ to be included in **MC**. This transaction is only included into **MC** if it is considered valid, which means: (1) it is properly signed; (2) it contains a Merkle inclusion proof confirming its presence in some already included sidechain certificate; (3) its amount is less or equal to the current value of $\mathsf{pool}_{\mathbf{SC}}$. If included, **MC**-maintainers decrease the value of $\mathsf{pool}_{\mathbf{SC}}$ by the amount of this transaction. The code of the withdrawal algorithm is illustrated in Algorithm 13.

Algorithm 13 Withdrawing from **SC** to **MC**.

The algorithm is run by a stakeholder U in control of the secret key sk corresponding to the account sAcc on **SC**.

function $\mathsf{Send}(\mathsf{sAcc}, \mathsf{rAcc}, v)$ ▷ Send v from sAcc on **SC** to rAcc on **MC**
  $\mathsf{txid} \xleftarrow{\$} \{0,1\}^k$
  $\sigma \leftarrow \mathsf{Sig}_{sk}(\mathsf{txid}, \mathbf{SC}, (\mathbf{SC}, \mathsf{sAcc}), (\mathbf{MC}, \mathsf{rAcc}), v)$
  $\mathsf{tx}_{\mathsf{send}} \leftarrow (\mathsf{txid}, \mathbf{SC}, (\mathbf{SC}, \mathsf{sAcc}), (\mathbf{MC}, \mathsf{rAcc}), v, \sigma)$
  post $\mathsf{tx}_{\mathsf{send}}$ **to SC**
end function
function $\mathsf{Receive}(\mathsf{txid}, \mathsf{sAcc}, \mathsf{rAcc}, v)$
  wait until $\mathsf{tx}_{\mathsf{send}} \in \mathsf{C}_{\mathbf{SC}}$
  $j' \leftarrow$ epoch where $\mathsf{C}_{\mathbf{SC}}$ contains $\mathsf{tx}_{\mathsf{send}}$
  if ($\mathsf{tx}_{\mathsf{send}}$ included in slot $sl \leq R-4$ of $e_{j'}$) **then**
    $j_{\mathsf{send}} \leftarrow j'$
  else
    $j_{\mathsf{send}} \leftarrow j'+1$
  end if
  wait until $\mathsf{sc\_cert}_{j_{\mathsf{send}}+1} \in \mathsf{C}_{\mathbf{MC}}$
  $\pi \leftarrow$ Merkle-tree proof of $\mathsf{tx}_{\mathsf{send}}$ in $\mathsf{sc\_cert}_{j_{\mathsf{send}}+1}$
  $\sigma \leftarrow \mathsf{Sig}_{sk}(\mathsf{txid}, \mathbf{MC}, (\mathbf{SC}, \mathsf{sAcc}), (\mathbf{MC}, \mathsf{rAcc}), v, \pi)$
  $\mathsf{tx}_{\mathsf{rec}} \leftarrow (\mathsf{txid}, \mathbf{MC}, (\mathbf{SC}, \mathsf{sAcc}), (\mathbf{MC}, \mathsf{rAcc}), v, \pi, \sigma)$
  post $\mathsf{tx}_{\mathsf{rec}}$ **to MC**
end function

The certificate transaction. We now describe the construction of the $\mathsf{sc\_cert}$ transaction, also called the *sidechain certificate*, formally described in Algorithm 14). The role of the certificate produced by the end of epoch $j-1$ to be included in **MC** at the beginning of epoch j (denoted $\mathsf{sc\_cert}_j$) is to attest all the withdrawals that had their sending transactions included into **SC** in either the last $4k$ slots of e_{j-2} or the first $R-4k$ slots of e_{j-1}. To maintain a chain of trust for the **MC** maintainers that cannot verify these transactions by observing **SC**, we make use of ad-hoc threshold multisignatures introduced in Section 4.1.

Algorithm 14 Constructing sidechain certificate $\mathsf{sc\_cert}$.

The algorithm is run by every **SC**-maintainer at the end of each epoch, j denotes the index of the ending epoch.

function $\mathsf{ConstructSCCert}(j)$
  $T \leftarrow$ last $4k$ slots of e_{j-1} and first $R-4k$ slots of e_j
  $\mathbf{tx} \leftarrow$ transactions included in **SC** during T
  $\mathsf{pending}_{j+1} \leftarrow \{\mathsf{tx} \in \mathbf{tx} : \mathsf{tx.send} \neq \mathsf{tx.rec}\}$
  $\mathcal{VK}_{j+1} \leftarrow$ keys of last $2k$ **SC** slot leaders in e_{j+1}
  $avk^{j+1} \leftarrow \mathsf{AKey}(\mathcal{VK}_{j+1})$
  $m \leftarrow \left(\left\langle \mathsf{pending}_{j+1} \right\rangle, avk^{j+1}\right)$
  $\mathcal{VK}_j \leftarrow$ keys of last $2k$ **SC** slot leaders for e_j
  $\sigma_{j+1} \leftarrow \mathsf{ASig}\left(m, \{(vk_i, \sigma_i)\}_{i=1}^{d}, \mathcal{VK}_j\right)$
  $\mathsf{sc\_cert}_{j+1} \leftarrow \left(\left\langle \mathsf{pending}_{j+1} \right\rangle, avk^{j+1}, \sigma_{j+1}\right)$
  return $\mathsf{sc\_cert}_{j+1}$
end function

Namely, the $\mathsf{sc\_cert}_j$ transaction also contains an aggregate key avk^j of an ATMS, and is signed by the previous aggregate key avk^{j-1} included in $\mathsf{sc\_cert}_{j-1}$.

$\mathsf{sc\_cert}_j$ is generated by **SC**-maintainers and contains:

- **The epoch index** j.
- **The pending transactions from SC to MC.** Let **tx** be the sequence of all transactions which are included in **SC** during either the last $4k$ slots of e_{j-2} or the first $R-4k$ slots of e_j. All transactions in **tx** that have $\mathbf{SC} = \mathsf{send} \neq \mathsf{rec} = \mathbf{MC}$ are picked up and combined into a list $\mathsf{pending}_j$ (sorted in the same order as in **SC**). Let $\left\langle \mathsf{pending}_j \right\rangle$ denote a Merkle-tree commitment to this list.
- **The new ATMS key** avk^j**.** The key is created from the public keys of the slot leaders of the last $2k$ slots of the epoch j, using threshold $k+1$. Hence, it allows to verify whether a particular signature comes from $k+1$ out of these $2k$ keys.
- **Signature valid with respect to** avk^{j-1}**.**

The full $\mathsf{sc\_cert}_j$ is therefore a tuple $\left(j, \langle \mathsf{pending}_j \rangle, avk^j, \sigma_j\right)$, where σ_j is an ATMS signature on the preceding elements that verifies using avk^{j-1}.

The certificate $\mathsf{sc\_cert}_{j+1}$ is constructed as follows: Both the stake distribution $\overline{\mathsf{SD}}^*_{j+1}$ and the **SC**-randomness $\bar{\eta}_{j+1}$ (and hence also the slot leader schedule for **SC** in epoch $j+1$) are determined by the states of the blockchains **MC** and **SC** by the end of slot $10k$ of epoch j. Therefore, during the last $2k$ slots of epoch j, the $2k$ elected slot leaders for these slots can already include a (local) signature on (their proposal of) $\mathsf{sc\_cert}_{j+1}$ into the blocks they create. Given the deterministic construction of $\mathsf{sc\_cert}_{j+1}$, all valid blocks ending up in the part of **SC**-chain belonging to the last $2k$ slots of epoch j will contain a local signature on the same $\mathsf{sc\_cert}_{j+1}$, and by the chain growth property of the underlying blockchain, there will be at least $k+1$ of them. Therefore, any party observing **SC** can now combine these signatures into an ATMS that can be later verified using the ATMS key avk^j, it can hence create the complete certificate $\mathsf{sc\_cert}_{j+1}$ and serve it to the maintainers of **MC** for inclusion.

Transitioning trust. As already outlined above, our construction uses ATMS to maintain the authenticity of the sidechain certificates from epoch to epoch. We now describe this inductive process in greater detail.

Initially, during the setup of the sidechain, $\mathcal{P} \leftarrow \mathsf{PGen}(1^\kappa)$ is ran. Stakeholders generate their keys by invoking $(sk_i, vk_i) \leftarrow \mathsf{Gen}(\mathcal{P})$. In case $\mathsf{Gen}(\cdot)$ is a probabilistic algorithm, it is run in a derandomized fashion with its coins fixed to the output of a PRNG that is seeded by $H(\mathsf{ats\_init}, \eta_{j_{\mathsf{start}}})$ where "$\mathsf{ats\_init}$" is a fixed label and H is a hash function. This ensures that $\mathcal{P}$ will be uniquely determined and will still be unpredictable. We note that this process is only suitable for ATMS that employ public-coin parameters; our ATMS constructions in Section 5 are only of this type.

For the induction base, $\mathcal{P}$ is published as part of the Genesis block $\overline{\mathsf{G}}$. Each time an **MC** stakeholder U_i posts the $\mathsf{sidechain\_support}$ message to **MC**, he also includes an ATMS key vk_i. Subsequently, when the **SC** is initialised, the stake distribution $\overline{\mathsf{SD}}^*_{j_{\mathsf{start}}}$ is known to the **MC** participants. Hence, based on $\overline{\mathsf{SD}}^*_{j_{\mathsf{start}}}$ and $\bar{\eta}_{j_{\mathsf{start}}}$, these can determine the last $2k$ slot leaders of epoch j_{start} in **SC**, we will refer to them as the j_{start}-th *trust committee*. (In general, the j-th *trust committee* for $j \geq j_{\mathsf{start}}$ will be the set of last $2k$ slot leaders in epoch j.) **SC**-maintainers (that also follow **MC**) can also determine the j_{start}-th trust committee and therefore create $avk^{j_{\mathsf{start}}}$ from their public keys and insert it into the genesis block $\overline{\mathsf{G}}$ of **SC**. They can also serve it as a special transaction to the **MC**-maintainers to include into the mainchain. The correctness of $avk^{j_{\mathsf{start}}}$ can be readily verified by anyone following the mainchain using the procedure ACheck of the used ATMS.

For the induction step, consider an epoch $j > j_{\mathsf{start}}$ and assume that there exists an ATMS key of the previous epoch avk^{j-1}, known to the mainchain maintainers. Every honest **SC** slot leader among the last $2k$ slot leaders of **SC** epoch $j-1$ will produce a local signature s_i^j on the message $m = (j, \langle \mathsf{pending}_j \rangle, avk_j)$ using their private key sk_i^{j-1} by running $\mathsf{Sig}(sk_i^{j-1}, m)$, and include this signature into the block they create. The rest of the **SC** maintainers will verify that the epoch index, avk^j and $\langle \mathsf{pending}_j \rangle$ are correct (by ensuring $\mathsf{ACheck}(\mathcal{VK}^j, avk^j)$ is *true* for $\mathcal{VK}$ denoting the public keys of the last $2k$ slot leaders on **SC** for epoch j, and by recomputing the Merkle tree commitment $\langle \mathsf{pending}_j \rangle$) and that s_i^j is valid by running $\mathsf{Ver}(m, vk_i^{j-1}, s_i^j)$, otherwise the block is considered invalid. Thanks to the chain growth property of the underlying Ouroboros protocol, after the last $2k$ slots of epoch $j-1$ the honest sidechain maintainers will all observe at least $k+1$ signatures among the $\{s_i^j : i \in [2k]\}$ desired ones. They then combine all of these local signatures into an aggregated ATMS signature $\sigma^j \leftarrow \mathsf{ASig}(m, \{(s_i^j, vk_i^{j-1})\}, \mathsf{keys}^j)$. This combined signature is then diffused as part of $\mathsf{sc\_cert}_j$ on the mainchain network. The mainchain maintainers verify that it has been signed by the sidechain maintainers by checking that $\mathsf{AVer}(m, avk^{j-1}, \sigma^j)$ evaluates to *true* and include it in a mainchain block. This effectively hands over control to the new committee.

5 Constructing Ad-Hoc Threshold Multisignatures

In this section we give several ways to instantiate the ATMS primitive. We order them by increasing succinctness but also increasing complexity. We defer full proofs that our constructions satisfy ATMS correctness and security (as per Definitions 10 and 11) to later versions of this paper.

5.1 Plain ATMS

Given a EUF-CMA-secure signature scheme, combining signatures and keys can be implemented by plain concatenation. Subsequently, combined verification requires all signatures to be verified individually. This illustrates that the ATMS primitive is easy to realize if no concern is given to succinctness. The size of these aggregate signatures and aggregate keys is *quadratic* in the security parameter κ: for the aggregate key $2k$ individual keys of size κ bits each are concatenated (with $k = \Theta(\kappa)$), while the aggregate signature consists of at least $k+1$ individual signatures of size κ bits.

5.2 Multisignature-based ATMS

The previous construction can be improved by employing an appropriate multisignature scheme. In the construction below, we consider the multisignature scheme Π_{MGS} from [8]. We make use of a homomorphic property of this scheme: any d individual signatures $\sigma_1, \ldots, \sigma_d$ created using secret keys belonging to (not necessarily unique) public keys $vk_1, \ldots, vk_d$ can be combined into a multisignature $\sigma = \prod_{i=1}^{d} \sigma_i$ that can then be verified using an aggregated public key $avk = \prod_{i=1}^{d} vk_i$.

Our multisignature-based t-ATMS construction works as follows: the procedures PGen, Gen, Sig and Ver work exactly as in Π_{MGS}. Given a set S, denote by $\langle S \rangle$ a Merkle-tree commitment to the set S created in some arbitrary, fixed, deterministic way. Procedure AKey, given a sequence of public keys $\mathcal{VK} = \{vk_i\}_{i=1}^{n}$

returns $avk = (\prod_{i=1}^{n} vk_i, \langle \mathcal{VK} \rangle)$. Since AKey is deterministic, $\mathsf{ACheck}(\mathcal{VK}, avk)$ simply recomputes it to verify avk. ASig takes the message m, d pairs of signatures with their respective public keys $\{\sigma_i, vk_i\}_{i=1}^{d}$ and $n-d$ additional public keys $\{\widehat{vk}_i\}_{i=1}^{n-d}$ and produces an aggregate signature

$$\sigma = \left(\prod_{i=1}^{d} \sigma_i, \{\widehat{vk}_i\}_{i=1}^{n-d}, \{\pi_{\widehat{vk}_i}\}_{i=1}^{n-d} \right) \tag{1}$$

where $\pi_{\widehat{vk}_i}$ denotes the (unique) inclusion proof of $\widehat{vk}_i$ in the Merkle commitment $\left\langle \{vk_i\}_{i=1}^{d} \cup \{\widehat{vk}_i\}_{i=1}^{n-d} \right\rangle$. Finally, the procedure AVer takes a message m, an aggregate key avk, and an aggregate signature σ parsed as in (1), and does the following: (a) verifies that each of the public keys $\widehat{vk}_i$ indeed belongs to a different leaf in the commitment $\langle \mathcal{VK} \rangle$ in avk using membership proofs $\pi_{\widehat{vk}_i}$; (b) computes avk' by dividing the first part of avk by $\prod_{i=1}^{n-d} \widehat{vk}_i$; (c) returns *true* if and only if $d \geq t$ and the first part of σ verifies as a Π_{MGS}-signature under avk'.

Note that the scheme Π_{MGS} requires vk_i to be accompanied by a (non-interactive) proof-of-possession (POP) [28] of the respective secret key. This POP can be appended to the public key and verified when the key is communicated in the protocol. For conciseness, we omit these proofs-of-knowledge from the description (but we include them in the size calculation below).

Asymptotic Complexity. This provides an improvement in our use case over the plain scheme: In the optimistic case where each of the $2k$ committee members create their local signatures, both the aggregate key avk and the aggregate signature σ are *linear* in the security parameter, which is optimal. If $r < k$ of the keys do *not* provide their local signatures, the construction falls back to being *quadratic* in the worst case if $r = k - 1$. However, for the practically relevant case where $r \ll k$ and almost all slot leaders produced a signature, this construction is clearly preferable.

Concrete space requirements. Concrete signature sizes in this scheme for practical parameters could be as follows. We set $k = 2160$ (as is done in the Cardano implementations of [20]) and for the signature of [8] we have in bits: $|vk_i| = 272$, $|\sigma_i| = 528$ (N. Di Prima, V. Hanquez, personal communication, 16 Mar 2018), with $|vk_i + POP| = |vk_i| + |\sigma_i| = 800$ bits. Assuming 256-bit hash function is used for the Merkle tree construction, the size of the data which needs to be included in **MC** in the optimistic case during an epoch transition is $|avk| + |\sigma| + |\langle \mathsf{pending} \rangle| = |vk_i + POP| + 2|H(\cdot)| + |\sigma_i| = 800 + 512 + 528 = 1840$ bits per epoch. In a case where 10% of participants fail to sign, the size will be $|avk| + |\sigma| = |vk_i + POP| + 2|H(\cdot)| + |\sigma_i| + 0.1 \cdot 2 \cdot k(|vk_i + POP| + \log(k)|H(\cdot)) = 800 + 512 + 528 + 432 \cdot (500 + 12 \cdot 256) = 1544944$, or about 190 KB per epoch (which is approximately 5 days).

5.3 ATMS From Proofs of Knowledge

While the aggregate signatures construction seems sufficient for practice, it still requires a $\mathsf{sc\_cert}$ transaction that is in the worst case quadratic in the security parameter. The approach below, based on proofs of knowledge, improves on that.

We define $avk \leftarrow \mathsf{AKey}(\mathcal{VK})$ to be the root of a Merkle tree that has $\mathcal{VK}$ at its leaves. Let $\mathsf{Sig}, \mathsf{Ver}$ come from any secure signature scheme. In our ATMS, the local signature is equal to $s_i = \mathsf{Sig}(sk_i, m)$, where sk_i is the secret key that corresponds to the vk_i verification key. Letting $S' = \{s_i\}$ be the signatures generated by a sequence $\mathcal{VK}'$ containing keys in $\mathcal{VK}$, the $\mathsf{ASig}(\mathcal{VK}, S', m)$ algorithm reconstructs the Merkle tree from $\mathcal{VK}$ and determines the membership proof π_i for each $vk_i \in \mathcal{VK}'$. Regarding the non-interactive argument of knowledge, the statement of interest is (avk, m) with witness $\{\pi_i, (s_i, vk_i)\}_{i \in S'}$ such that for all i we have that $\mathsf{Ver}(vk_i, m, s_i) = 1$ and π_i is a valid Merkle tree proof pointing to a unique leaf for every i. π_i demonstrates that vk_i is in avk. We also require $|S'| \geq t$. It is possible to construct succinct proofs for this statement using SNARKs [7] or even without any trusted setup using e.g., STARKs [4] or Bulletproofs [10] in the Random Oracle model [3]. In both cases the actual size of the resulting signature will be at most logarithmic in k, while in the case of STARKs the verifier will also have time complexity logarithmic in k.

6 Security

In this section we give a formal argument establishing that the construction from Section 4 achieves pegging security of Definition 8.

6.1 Assumptions

Let $\mathbb{A}_{\mathsf{hm}}(\mathbf{L})[t]$ denote the honest-majority assumption for an Ouroboros ledger $\mathbf{L}$. Namely, $\mathbb{A}_{\mathsf{hm}}(\mathbf{L})[t]$ postulates that in all slots $t' \leq t$, the majority of stake in the stake distribution used to sample the slot leader for slot t' in $\mathbf{L}$ is controlled by honest parties (note that the distribution in question is SD and $\overline{\mathsf{SD}}^*$ for **MC** and **SC**, respectively). Specifically, the adversary is restricted to $(1-\epsilon)/2$ relative stake for some fixed $\epsilon > 0$.

The assumption $\mathbb{A}_{\mathbf{MC}}$ we consider for **MC** is precisely $\mathbb{A}_{\mathbf{MC}}[t] \triangleq \mathbb{A}_{\mathsf{hm}}(\mathbf{MC})[t]$, while the assumption $\mathbb{A}_{\mathbf{SC}}$ for **SC** is $\mathbb{A}_{\mathbf{SC}}[t] \triangleq \mathbb{A}_{\mathbf{MC}}[t] \wedge \mathbb{A}_{\mathsf{hm}}(\mathbf{SC})[t]$. The reason that $\mathbb{A}_{\mathbf{SC}}[t] \Rightarrow \mathbb{A}_{\mathbf{MC}}[t]$ is that **SC** uses merged staking and hence cannot provide any security guarantees if the stake records on **MC** get corrupted. It is worth noting that it is possible to program **SC** to wean off **MC** and switch to independent staking; in such case the assumption for **SC** will transition to $\mathbb{A}_{\mathsf{hm}}(\mathbf{SC})$ (now with respect to $\overline{\mathsf{SD}}$) after the weaning slot and the two chains will become sidechains of each other.

Remark 1. We note that the assumption of honest majority in the distribution out of which leaders are *sampled* is one of two related ways of stating this requirement. The distribution from which sampling is performed corresponds to the actual stake distribution near the end of the previous epoch. Hence, the actual stake may have since shifted and may no longer be honest. Had we wanted to formulate this assumption in terms of the actual (current) stake distribution, we would have to state two different assumptions: (1) that the current actual stake has honest majority with some gap σ; and (2) that the rate of stake shifting is bounded by σ for the duration of (roughly) 2 epochs. From these two assumptions, one can conclude that the distribution from which leaders are elected is currently controlled by an honest majority. The latter approach was taken for example in [20].

6.2 Proof Overview

Proving our construction secure requires some case analysis. We summarize the intuition behind this endeavour before we proceed with the formal treatment.

The proof of Theorem 1 that shows that our construction from Section 4 has pegging security with overwhelming probability will be established as follows. We will borrow the fact that our construction achieves persistence and liveness from the original analysis [20] and state them as Lemma 1. The main challenge will be to establish the firewall property, which is done in Lemma 7. These properties together establish pegging security as required by Definition 8.

To show that the firewall property holds, we perform a case analysis, looking at the two cases of interest: when both **MC** and **SC** are secure (i.e., when $\mathbb{A}_{\mathbf{MC}} \wedge \mathbb{A}_{\mathbf{SC}}$ holds), and when only **MC** is secure while the security assumption of **SC** has been violated. As discussed above, the case where **SC** is secure and the security of **MC** has been violated cannot occur per definition of $\mathbb{A}_{\mathbf{MC}}$ and $\mathbb{A}_{\mathbf{SC}}$, and so examining this case is not necessary.

First, we examine the case where both **MC** and **SC** are secure, but only concern ourselves with *direct observation* transactions, or transactions that can be verified without relying on sidechain certificates. We show that such transactions will always be correctly verified in this case.

Next, we establish that, when only **MC** is secure, it is impossible for the **MC** maintainers to accept a view inconsistent with the validity language, and hence the firewall property is maintained in the case of a sidechain failure.

Finally, the heart of the proof is a computational reduction (using the above partial results) showing how, given an adversary that breaks the firewall property, there must exist a receiving transaction on **MC** which breaks the validity of the scheme. Given such a transaction, we can construct an adversary against either the security of the underlying ATMS scheme or the collision resistance of the underlying hash function.

6.3 Liveness and Persistence

We begin by stating the persistence and liveness guarantees of our construction, they both follow directly from the guarantees shown for the standalone Ouroboros blockchain in [20].

Lemma 1 (Persistence and Liveness). *Consider the construction of Section 4 with the assumptions* $\mathbb{A}_{\mathbf{SC}}, \mathbb{A}_{\mathbf{MC}}$*. For all slots* t*, if* $\mathbb{A}_{\mathbf{SC}}[t]$ *(resp.* $\mathbb{A}_{\mathbf{MC}}[t]$*) holds, then* **SC** *(resp.* **MC***) satisfies persistence and liveness up to slot* t *with overwhelming probability in* k*.*

We now restate the Common Prefix property of blockchains for future reference. If the Common Prefix property holds, then Persistence can be derived along the lines of [20].

Definition 12 (Common Prefix). *For every honest party* P_1 *and* P_2 *both maintaining the same ledger (i.e., either both maintaining* **MC***, or both maintaining* **SC***) and for every slot* r_1 *and* r_2 *such that* $r_1 \leq r_2 \leq t$*, let* C_1 *be the adopted chain of* P_1 *at slot* r_1 *and* C_2 *be the adopted chain of* P_2 *at slot* r_2*. The* k-common prefix property *for slot* t *states that* $\mathsf{C}_2[: |\mathsf{C}_1[: -k]|] = \mathsf{C}_1[: -k]$*.*

6.4 The Firewall Property and MC-Receiving Transactions

Recall that the transactions in $\mathcal{T}_{\mathfrak{A}}$ can be partitioned into several classes with different validity-checking procedures. First, there are *local* transactions (where $\mathsf{send} = \mathsf{rec} = \mathsf{lid}$) and *sending* transactions (with $\mathsf{lid} = \mathsf{send} \neq \mathsf{rec}$). Then we have *receiving* transactions (with $\mathsf{send} \neq \mathsf{rec} = \mathsf{lid}$), which can be split into **SC**-*receiving* transactions ($\mathsf{send} \neq \mathsf{rec} = \mathsf{lid} = \mathbf{SC}$) and **MC**-*receiving* transactions ($\mathsf{send} \neq \mathsf{rec} = \mathsf{lid} = \mathbf{MC}$).

As the lemma below observes, if a transaction violates the firewall property in a certain situation, it must be a **MC**-receiving transaction.

Lemma 2. *Consider an execution of the protocol of Section 4 at slot* t *in which* **MC** *and* **SC** *satisfy persistence. Suppose*

$$\mathsf{L} = \mathsf{merge}\left(\{\mathsf{L}^{\cup}_{\mathbf{MC}}[t], \mathsf{L}^{\cup}_{\mathbf{SC}}[t]\}\right) \notin \mathbb{V}_{\mathfrak{A}}$$

and suppose that $\mathcal{S}_t = \{\mathbf{SC}, \mathbf{MC}\}$*. Let* L' *be the minimum prefix of* L *such that* $\mathsf{L}' \notin \mathbb{V}_{\mathfrak{A}}$*. Then* $\mathsf{L}' \neq \varepsilon$ *and* $\mathsf{tx} \triangleq \mathsf{L}'[-1]$ *is an* **MC***-receiving transaction.*

Proof. The *base* property of the validity language implies $\mathsf{L}' \neq \varepsilon$, hence tx exists. Due to the minimality of L', Algorithm 5 returns *false* for L' but *true* for $\mathsf{L}'[: -1]$. Since it processes transactions sequentially, it must return *false* during the processing of tx. Suppose for contradiction that tx is not an **MC**-receiving transaction; let us call such a transaction *direct* in this proof.

Algorithm 5 can output *false* while processing a direct transaction in the following cases: (a) in Line 17 when there is a Conservation Law violation; (b) in Line 8 when there is a signature validation failure; (c) in Line 13 when tx is a replay of a previous transaction; (d) in Line 22 when tx is a replay, or (e) in Line 27 when the pre-image transaction has not yet been processed. Hence, tx falls under one of these violations.

Due to persistence and the definition of $\mathsf{L}^{\cup}_{\mathbf{MC}}[t]$ and $\mathsf{L}^{\cup}_{\mathbf{SC}}[t]$, there exists an **MC** maintainer $P_{\mathbf{MC}}$ and an **SC** maintainer $P_{\mathbf{SC}}$, such that $\mathsf{L}^{P_{\mathbf{MC}}}_{\mathbf{MC}}[t] = \mathsf{L}^{\cup}_{\mathbf{MC}}[t]$ and $\mathsf{L}^{P_{\mathbf{SC}}}_{\mathbf{SC}}[t] = \mathsf{L}^{\cup}_{\mathbf{SC}}[t]$, respectively. Due to the *partitioning property* of merge, tx will be in $\mathsf{L}^{P_{\mathsf{lid}(\mathsf{tx})}}_{\mathsf{lid}(\mathsf{tx})}[t]$. We separately consider the two possibilities for $\mathsf{lid}(\mathsf{tx})$.

Case 1: $\mathsf{lid}(\mathsf{tx}) = \mathbf{MC}$. In this case, the only violations that a direct tx can attain are (a), (b) and (c), as the cases (d) and (e) for $\mathsf{lid}(\mathsf{tx}) = \mathbf{MC}$ do not pertain to a direct transaction. $P_{\mathbf{MC}}$ has reported $\mathsf{L}^{P_{\mathbf{MC}}}_{\mathbf{MC}}[t]$ as its adopted state, hence $\mathsf{L}^{P_{\mathbf{MC}}}_{\mathbf{MC}}[t]$ is a fixpoint of $\textsc{verifytx}_{\mathbf{MC}}$ (as $\textsc{verifytx}_{\mathbf{MC}}$ checks for a fixpoint). The execution of $\textsc{verifytx}_{\mathbf{MC}}$ included every transaction in $\mathsf{L}^{P_{\mathbf{MC}}}_{\mathbf{MC}}[t]$. Therefore, $\textsc{verifytx}_{\mathbf{MC}}$ has accepted every transaction in every iteration until the last iteration, which processes tx. Consider, now, what happened in the last iteration of the execution of $\textsc{verifytx}_{\mathbf{MC}}$. In that iteration, $\textsc{verifytx}_{\mathbf{MC}}$ checks the validity of σ, the Conservation Law, and transaction replay. In all cases (a), (b) and (c), $\textsc{verifytx}_{\mathbf{MC}}$ will reject tx. But this could not have happened, as $\mathsf{L}^{P_{\mathbf{MC}}}_{\mathbf{MC}}[t]$ is a fixpoint, and we have a contradiction.

Case 2: $\mathsf{lid}(\mathsf{tx}) = \mathbf{SC}$. Let C_{mc} and C_{sc} be the **MC** and respectively **SC** chain adopted by $P_{\mathbf{SC}}$ at slot t (and recall that $P_{\mathbf{SC}}$ maintains both chains). Let $\mathsf{C}_{\mathsf{mc}}{}'$ be the chain adopted by $P_{\mathbf{MC}}$ at slot t. As before,

$\text{ANNOTATETX}_{\mathbf{SC}}(\mathsf{C_{mc}}, \mathsf{C_{sc}})$ must be a fixpoint of $\text{VERIFYTX}_{\mathbf{SC}}$ (as $\text{VERIFYTX}_{\mathbf{SC}}$ checks for a fixpoint). As in the previous case, tx cannot violate (a), (b), (c) and in this case nor (d), as this would constitute a fixpoint violation. Hence tx is an effect transaction and we will examine whether tx constitutes a violation of (e).

Let $\mathsf{tx}^{-1} \triangleq \mathsf{effect}^{-1}_{\mathbf{MC}\to\mathbf{SC}}(\mathsf{tx})$. Since tx is accepted by $\text{VERIFYTX}_{\mathbf{SC}}$ on input $\text{ANNOTATETX}_{\mathbf{SC}}(\mathsf{C_{mc}}, \mathsf{C_{sc}})$, we deduce that there exists some block $B \in \mathsf{C_{mc}}[:-k]$ with $\mathsf{tx}^{-1} \in B$. But $\mathsf{C_{mc}}'[:-k]$ is the longest stable chain among $\mathbf{MC}$ maintainers (due to $\mathsf{L}^{\cup}_{\mathbf{MC}}[t] = \mathsf{L}^{P_{\mathbf{MC}}}_{\mathbf{MC}}[t]$), hence $\mathsf{C_{mc}}[:-k]$ is its prefix. Therefore $B \in \mathsf{C_{mc}}'[:-k]$. Hence, $\mathsf{tx}^{-1} \in \mathbf{L}^{P_{\mathbf{MC}}}_{\mathbf{MC}}[t]$. Due to the *partioning property* of merge, tx^{-1} must appear in the output of $\mathsf{merge}\left(\{\mathsf{L}^{P_{\mathbf{MC}}}_{\mathbf{MC}}[t], \mathsf{L}^{P_{\mathbf{SC}}}_{\mathbf{SC}}[t]\}\right)$. Due to the *topological soundness* of merge, tx^{-1} must appear *before* tx in $\mathsf{merge}\left(\{\mathsf{L}^{P_{\mathbf{MC}}}_{\mathbf{MC}}[t], \mathsf{L}^{P_{\mathbf{SC}}}_{\mathbf{SC}}[t]\}\right)$. Hence, it cannot be the case that (e) is violated, as the pre-image transaction exists. □

6.5 Firewall Property During Sidechain Failure

We now turn our attention to the case where the sidechain has suffered a "catastrophic failure" and so $\mathcal{S}_t = \{\mathbf{MC}\}$. We describe why a catastrophic failure in the sidechain does not violate the firewall property. To do this, we need to illustrate that, given a transaction sequence L which is accepted by the $\mathbf{MC}$ verifier, we can "fill in the gaps" with transactions from $\mathbf{SC}$ in order to produce a new transaction sequence $\mathbf{tx}$ which is valid with respect to $\mathbb{V}_{\mathfrak{A}}$.

We prove this constructively in Lemma 4. The construction of such a sequence is described in Algorithm 15. The algorithm accepts a transaction sequence $\mathsf{L} \subseteq \mathcal{T}_{\mathbf{MC}}$ valid according to $\text{VERIFIER}_{\mathbf{MC}}$ and produces a transaction sequence $\mathbf{tx} \in \mathbb{V}_{\mathfrak{A}}$ satisfying $\pi_{\mathbf{MC}}(\mathbf{tx}) = \mathsf{L}$, as desired.

The algorithm works by mapping each $\mathsf{tx} \in \mathsf{L}$ to one or more transactions in $\mathbf{tx}$. The mapping is done by calling $\mathsf{plausibility\text{-}map}(\mathsf{tx})$ for each transaction individually. Hence each transaction in $\mathbf{tx}$ has a specific preimage transaction in L, which can be shared by other transactions in $\mathbf{tx}$. The mapping is performed as follows. If tx is a local transaction, then it is simply copied over, otherwise some extra transactions are included. Specifically, if it's an sending transaction tx, then first tx is included, and subsequently the funds are recovered by a corresponding transaction tx_1 on $\mathbf{SC}$, the effect transaction of tx. The funds are afterwards moved to a pool address pool_{pk} by a transaction tx_2. (Note that for this, we assume that the receiving account public key has a correspnding private key, as this key is needed to sign tx_2. As we are only demonstrating the existence of $\mathbf{tx}$, Algorithm 15 does not need to be efficient and so assuming the existence of the private key is sufficient.) On the other hand, if it is an ($\mathbf{MC}$-)receiving transaction tx, the reverse procedure is followed. First, the funds are collected by tx_2 from the pool address pool_{pk} and moved into the $\mathbf{SC}$ address which will be used for the upcoming remote transaction. Then tx_1 moves the funds out of $\mathbf{SC}$ so that they can be collected by the corresponding tx on $\mathbf{MC}$. In the first case, the transaction sequence is $(\mathsf{tx}, \mathsf{tx}_1, \mathsf{tx}_2)$ and in the second case the sequence is $(\mathsf{tx}_2, \mathsf{tx}_1, \mathsf{tx})$. Note that, in both cases, tx and tx_1 are identical, except for the fact that tx is recorded on $\mathbf{MC}$ while tx_1 is recorded on $\mathbf{SC}$; the latter is the effect (or pre-image, respectively) of the former.

The simple intuition behind this construction is that, in the plausible history $\mathbf{tx}$ produced by Algorithm 15, the account pool_{pk} is holding all the money of the sidechain. More specifically, the balance that is maintained in the variable $\mathsf{balances}[\mathbf{SC}][\mathsf{pool}_{pk}]$ is identical to the pool variable maintained by the $\mathbf{MC}$ verifier. This invariant is made formal in Lemma 3.

Lemma 3 (Plausible balances). *Let $\mathsf{L} \in \mathcal{T}^{*}_{\mathfrak{A},\mathbf{MC}}$ and $\mathbf{tx} \leftarrow \mathit{plausible}(\mathsf{L})$. Consider an execution of Algorithm 5 on $\mathbf{tx}$ and an execution of $\text{VERIFIER}_{\mathbf{MC}}$ on L. Let $\mathsf{tx} \in \mathsf{L}$. Call $\mathit{pool}_{\mathsf{tx}}$ the value of the pool variable maintained by $\text{VERIFIER}_{\mathbf{MC}}$ prior to processing tx in its main* for *loop; call $\mathit{balances}[\mathbf{SC}][\mathit{pool}_{pk}]_{\mathsf{tx}}$ the value of the $\mathit{balances}[\mathbf{SC}][\mathit{pool}_{pk}]$ variable prior to the iteration of its main* for *loop which processes the first item of $\mathsf{plausibility\text{-}map}(\mathsf{tx})$. For all $\mathsf{tx} \in \mathsf{L}$, the following invariant will hold: $\mathit{pool}_{\mathsf{tx}} = \mathit{balances}[\mathbf{SC}][\mathit{pool}_{pk}]_{\mathsf{tx}}$.*

Proof. By direct inspection of the two algorithms, observe that the $\mathsf{balances}[\mathbf{SC}][\mathsf{pool}_{pk}]$ are updated by Algorithm 5 only when $\mathsf{send}(\mathsf{tx}_a) \neq \mathsf{rec}(\mathsf{tx}_a)$. The balances are increased when $\mathsf{send}(\mathsf{tx}_a) = \mathbf{MC}$ (due to $\mathsf{tx}_2 \in \mathsf{plausibility\text{-}map}(\mathsf{tx})$ at Line 19 of Algorithm 15) and decreased when $\mathsf{send}(\mathsf{tx}_a) = \mathbf{SC}$ (due to $\mathsf{tx}_2 \in$

plausibility-map(tx) at Line 25 of Algorithm 15). Exactly the same accounting is performed by VERIFIER$_{\mathbf{MC}}$ when the respective tx is processed. □

Algorithm 15 The plausible transaction sequence generator.

```
1: (pool_sk, pool_pk) ← Gen(1^λ)
2: function plausible(L)
3:     tx ← ε
4:     for tx ∈ L do
5:         tx ← tx || plausibility-map(tx)
6:     end for
7:     return tx
8: end function
9: function plausibility-map(tx)
10:    ▷ Destructure tx into its constituents
11:    (txid, lid, (send, sAcc), (rec, rAcc), v, σ) ← tx
12:    if send = rec then
13:        return (tx)
14:    end if
15:    if send = MC then
16:        tx_1 ← effect_{MC→SC}(tx)
17:        Construct a valid σ_2 using the private key corresponding to rAcc
18:        Generate a fresh txid_2
19:        tx_2 ← (txid_2, SC, (SC, rAcc), (SC, pool_pk), v, σ_2)
20:        return (tx, tx_1, tx_2)
21:    end if
22:    if send = SC then
23:        Construct a valid σ_2 using pool_sk
24:        Generate a fresh txid_2
25:        tx_2 ← (txid_2, SC, (SC, pool_pk), (SC, sAcc), v, σ_2)
26:        tx_1 ← effect^{-1}_{SC→MC}(tx)
27:        return (tx_2, tx_1, tx)
28:    end if
29: end function
```

We now prove the correctness of Algorithm 15 in Lemma 4.

Lemma 4 (Plausibility). *For all* $\mathsf{L} \in \mathcal{T}^*_{\mathfrak{A},\mathbf{MC}}$*, if* VERIFYTX$_{\mathbf{MC}}(\mathsf{L}) = \mathsf{L}$ *then* $\mathbf{tx} \leftarrow$ *plausible*(L) *will satisfy* $\mathbf{tx} \in \mathbb{V}_{\mathfrak{A}}$.

Proof. Suppose for contradiction that $\mathbf{tx} \notin \mathbb{V}_{\mathfrak{A}}$ and let $\mathbf{tx}'$ be the minimum prefix of $\mathbf{tx}$ such that $\mathbf{tx}' \notin \mathbb{V}_{\mathfrak{A}}$. From the validity language *base* property we have that $\mathbf{tx}' \neq \varepsilon$ and so it must have at least one element. Let $\mathsf{tx} \triangleq \mathbf{tx}'[-1]$ and let $\mathsf{tx_L} \in \mathsf{L}$ be the input to plausibility-map which caused tx to be included in $\mathbf{tx}$ in the execution of plausible in Algorithm 15. Since Algorithm 5 processes transactions sequentially, and by the minimality of $\mathbf{tx}'$, it must return *false* when tx is processed.

We distinguish the following cases for $\mathsf{tx_L}$:

Case 1: Local transaction: $\mathsf{send}(\mathsf{tx_L}) = \mathsf{rec}(\mathsf{tx_L})$. Then $\mathsf{tx} = \mathsf{tx_L}$ and $\mathsf{send}(\mathsf{tx}) = \mathsf{lid}(\mathsf{tx})$. Since L is a fixpoint of VERIFYTX$_{\mathbf{MC}}$, tx must (a) have a valid signature σ, (b) not be a replay transaction, and (c) respect the Conservation Law. As $\mathsf{tx_L}$ is a local transaction satisfying all of (a), (b) and (c), therefore $\mathbf{tx}' \in \mathbb{V}_{\mathfrak{A}}$, which is a contradiction.

Case 2: Sending transaction: $\mathsf{send}(\mathsf{tx_L}) = \mathbf{MC}$ and $\mathsf{rec}(\mathsf{tx_L}) = \mathbf{SC}$. In this case, let $(\mathsf{tx_L}, \mathsf{tx}_1, \mathsf{tx}_2) =$ plausibility-map$(\mathsf{tx_L})$. If $\mathsf{tx} = \mathsf{tx_L}$, then tx is a sending transaction and we can apply the same reasoning to

argue that it will respect properties (a), (b) and (c). But those are the only violations for which Algorithm 5 can reject an sending transaction, and hence $\mathbf{tx}' \in \mathbb{V}_{\mathfrak{A}}$, which is a contradiction.

If $\mathsf{tx} = \mathsf{tx}_1$, then Algorithm 5 must return *true*. To see this, consider the cases when Algorithm 5 returns *false*: (d) a replay failure in Line 22, which cannot occur as tx_L has been accepted by $\text{VERIFYTX}_{\mathbf{MC}}$ and so $\text{VERIFYTX}_{\mathbf{MC}}$ must have *seen* tx_L only once while Algorithm 5 must be seeing it for exactly the second time; or (e) a mismatch failure in Line 22 which cannot occur as tx_1 is constructed identical to tx_L.

If $\mathsf{tx} = \mathsf{tx}_2$ then $\mathsf{send}(\mathsf{tx}) = \mathsf{rec}(\mathsf{tx})$. This transaction cannot cause Algorithm 5 to return *false*. To see this, consider the cases when Algorithm 5 returns *false*: (a) a signature failure in Line 8 cannot occur because σ_2 was constructed correctly and the signature scheme is correct; (b) a replay failure in Line 13 cannot occur because txid_2 is fresh; (c) a conservation failure in Line 17 cannot occur because the immediately preceding transaction $\mathbf{tx}'[-2]$ supplies sufficient balance.

Case 3: Receiving transaction: $\mathsf{send}(\mathsf{tx}_\mathsf{L}) = \mathbf{SC}$ and $\mathsf{rec}(\mathsf{tx}_\mathsf{L}) = \mathbf{MC}$. In this case, let $(\mathsf{tx}_2, \mathsf{tx}_1, \mathsf{tx}_\mathsf{L}) = \mathsf{plausibility\text{-}map}(\mathsf{tx}_\mathsf{L})$. The argument for $\mathsf{tx} = \mathsf{tx}_\mathsf{L}$ and $\mathsf{tx} = \mathsf{tx}_1$ is as in *Case 2*. For the case of $\mathsf{tx} = \mathsf{tx}_2$, the same argument as before holds for a signature validity and for replay protection. It suffices to show that the conservation law is not violated. This is established in Lemma 3 by the invariant that $\mathsf{pool}_{\mathsf{tx}_\mathsf{L}} = \mathsf{balances}[\mathbf{SC}][\mathsf{pool}_{pk}]_{\mathsf{tx}_\mathsf{L}}$ that holds prior to processing tx_2, as it is the first transaction of a triplet produced by $\mathsf{plausibility\text{-}map}$. As $\text{VERIFYTX}_{\mathbf{MC}}(\mathsf{L}) = \mathsf{L}$ then therefore $\mathsf{pool}_{\mathsf{tx}_\mathsf{L}} - v \geq 0$ and so $\mathsf{balances}[\mathbf{SC}][\mathsf{pool}_{pk}]_{\mathsf{tx}_\mathsf{L}} - \mathit{amount} \geq 0$ and Algorithm 5 returns *true*.

All three cases result in a contradiction, concluding the proof. □

Lemma 5 (SC failure firewall). *Consider any execution of the construction of Section 4 in which persistence holds for* $\mathbf{MC}$*. For all slots* t *such that* $\mathcal{S}_t = \{\mathbf{MC}\}$ *we have that*

$$\mathsf{merge}(\{\mathbf{L}^{\cup}_{\mathbf{MC}}[t]\}) \in \pi_{\{\mathbf{MC}\}}(\mathbb{V}_{\mathfrak{A}}) \,.$$

Proof. From the assumption that persistence holds, there exists some $\mathbf{MC}$ party P for which $\mathbf{L}^P_{\mathbf{MC}}[t] = \mathbf{L}^{\cup}_{\mathbf{MC}}[t]$. Additionally, $\mathsf{merge}(\{\mathbf{L}^{\cup}_{\mathbf{MC}}[t]\}) = \mathbf{L}^{\cup}_{\mathbf{MC}}[t]$ due to the *partitioning* property. It suffices to show that there exists some $\mathbf{tx} \in \mathbb{V}_{\mathfrak{A}}$ such that $\pi_{\{\mathbf{MC}\}}(\mathbf{tx}) = \mathbf{L}^P_{\mathbf{MC}}[t]$. Let $\mathbf{tx} \leftarrow \mathsf{plausible}(\mathbf{L}^P_{\mathbf{MC}}[t])$. We have $\text{VERIFIER}_{\mathbf{MC}}(\mathbf{L}^P_{\mathbf{MC}}[t]) = \mathit{true}$, so apply Lemma 4 to obtain that $\mathbf{tx} \in \mathbb{V}_{\mathfrak{A}}$.

To see that $\pi_{\{\mathbf{MC}\}}(\mathbf{tx}) = \mathbf{L}^P_{\mathbf{MC}}[t]$, note that Algorithm 15 for input L includes all $\mathsf{tx} \in \mathsf{L}$ in the same order as in its input. Furthermore, all $\mathsf{tx} \in \mathbf{tx}$ such that $\mathsf{tx} \notin \mathsf{L}$ have $\mathsf{lid}(\mathsf{tx}) = \mathbf{SC}$ and so are excluded from the projection. □

6.6 General Firewall Property

In preparation for establishing the full firewall property, we state the following simple technical lemma.

Lemma 6 (Honest subsequence). *Consider any set* S *of* $2k$ *consecutive slots prior to slot* t *in an execution of an Ouroboros ledger* $\mathbf{L}$ *such that* $\mathbb{A}_{\mathsf{hm}}(\mathbf{L})[t]$ *holds. Then* $k+1$ *slots of* S *are honest, except with negligible probability.*

Proof (sketch). If the adversary controlled at least k out of any $2k$ consecutive slots, he could use them to produce an alternative k-blocks long chain for this interval without any help from the honest parties, resulting in a violation of common prefix and hence persistence (cf. Lemma 1). □

We are now ready to prove our key lemma, showing that our construction satisfies the firewall property.

Lemma 7 (Firewall). *For all PPT adversaries* $\mathcal{A}$*, the construction of Section 4 with a secure ATMS and a collision-resistant hash function satisfies the firewall property with respect to assumptions* $\mathbb{A}_{\mathbf{MC}}, \mathbb{A}_{\mathbf{SC}}$ *with overwhelming probability in* k*.*

Proof. Let $\mathcal{A}$ be an arbitrary PPT adversary against the firewall property, and $\mathcal{Z}$ be an arbitrary environment for the execution of $\mathcal{A}$. We will construct the following PPT adversaries:

1. $\mathcal{A}_1$ is an adversary against ATMS.
2. $\mathcal{A}_2$ is a collision adversary against the hash function.

We first describe the construction of these adversaries.

The adversary $\mathcal{A}_1$. $\mathcal{A}_1$ simulates the execution of $\mathcal{A}$ and $\mathcal{Z}$ and of two populations of maintainers for two blockchains, **MC** and **SC**, which run the protocol Π (either the **MC** or the **SC**-maintainer part respectively) and spawns parties according to the mandates of the environment $\mathcal{Z}$ as follows. For all parties that are spawned as **MC** maintainers, $\mathcal{A}_1$ generates keys internally by invoking the Gen algorithm of the ATMS scheme. For all parties that are spawned as **SC** maintainers, $\mathcal{A}_1$ uses the oracle $\mathcal{O}^{\mathsf{gen}}$ to produce the public keys vk_i.

Whenever $\mathcal{A}$ requests that a (block or transaction) signature in **SC** is created, $\mathcal{A}_1$ invokes its oracle $\mathcal{O}^{\mathsf{sig}}$ to obtain the respective signature to provide to $\mathcal{A}$. When $\mathcal{A}$ requests that a **MC** signature is created, $\mathcal{A}_1$ uses its own generated private key to sign by invoking the Sig algorithm of the ATMS scheme. If $\mathcal{A}$ requests the corruption of a certain party P^*, then $\mathcal{A}_1$ reveals P^*'s private key to $\mathcal{A}$ as follows: If P^* is a **MC** maintainer, then the secret key is directly available to $\mathcal{A}_1$, so it is immediately returned. Otherwise, if P^* is a **SC** maintainer, then $\mathcal{A}_1$ obtains the secret key of P^* by invoking the oracle $\mathcal{O}^{\mathsf{cor}}$.

For every time slot t of the execution, $\mathcal{A}_1$ inspects all pairs $(P_{\mathbf{MC}}, P_{\mathbf{SC}})$ of honest parties such that $P_{\mathbf{MC}}$ is a **MC** maintainer and $P_{\mathbf{SC}}$ is a **SC** maintainer such that $\mathsf{L}_{\mathbf{MC}}^{P_{\mathbf{MC}}}[t] = \mathsf{L}_{\mathbf{MC}}^{\cup}[t]$ and $\mathsf{L}_{\mathbf{SC}}^{P_{\mathbf{SC}}}[t] = \mathsf{L}_{\mathbf{SC}}^{\cup}[t]$ (if such parties exist). Let $\mathsf{L}_1 = \mathsf{L}_{\mathbf{MC}}^{P_{\mathbf{MC}}}[t]$ and $\mathsf{L}_2 = \mathsf{L}_{\mathbf{SC}}^{P_{\mathbf{SC}}}[t]$. The adversary obtains the stable portion of the honestly adopted chain, namely $\mathsf{C}_1 = \mathsf{C}^{P_{\mathbf{MC}}}[t][: -k]$ and the transactions included in C_1, namely L_1' (note that $\mathsf{L}_1' \neq \mathsf{L}_1$ if L_1' contains certificate transactions). $\mathcal{A}_1$ examines whether $\mathsf{L} = \mathsf{merge}(\mathsf{L}_1, \mathsf{L}_2) \notin \mathbb{V}_{\mathfrak{L}}$, to deduce whether $\mathcal{A}$ has succeeded. Note that both the evaluation of merge on arbitrary states and the verification of inclusion in $\mathbb{V}_{\mathfrak{L}}$ are efficiently computable and hence $\mathcal{A}_1$ can execute them. If $\mathcal{A}_1$ is not able to find such a time slot t and parties $P_{\mathbf{MC}}, P_{\mathbf{SC}}$, it returns FAILURE (in the latter part of this proof, we will argue that all $\mathcal{A}_1$ failures occur with negligible probability conditioned on the event that $\mathcal{A}$ is successful, unless $\mathcal{A}_2$ is successful).

Otherwise it obtains the minimum t for which this holds and the L for this t. Because of the *base property* of the validity language, we have that $\epsilon \in \mathbb{V}_{\mathfrak{L}}$ and therefore $\mathsf{L} \neq \epsilon$. Let L^* be the minimum prefix of L such that $\mathsf{L}^* \notin \mathbb{V}_{\mathfrak{L}}$ and let $\mathsf{tx} = \mathsf{L}^*[-1]$. If tx has $\mathsf{send}(\mathsf{tx}) \neq \mathbf{SC}$ or $\mathsf{lid}(\mathsf{tx}) \neq \mathbf{MC}$, then $\mathcal{A}_1$ returns FAILURE. Now therefore $\mathsf{send}(\mathsf{tx}) = \mathbf{SC}$ and $\mathsf{lid}(\mathsf{tx}) = \mathbf{MC}$ (and so $\mathsf{tx} \in \mathsf{L}_1$). Hence, tx references a certain certificate transaction, say tx'. Due to the algorithm executed by **MC** maintainers for validation, we will have that $\mathsf{tx}' \in \mathsf{L}_1'\{: \mathsf{tx}\}$.

Let $\mathbf{tx}^*$ be the subsequence of L_1' containing all certificate transactions up to and including tx'. We will argue that there must exist some ATMS forgery among one of the certificate transactions in $\mathbf{tx}^*$. $\mathcal{A}_1$ looks at every transaction $\mathsf{sc\_cert}_j \in \mathbf{tx}^*$ (and note that it will correspond to a unique epoch e_j). $\mathsf{sc\_cert}_j$ contains a message $m = (j, \langle \mathsf{pending}_j \rangle, avk^j)$ and a signature σ_j. $\mathcal{A}_1$ extracts the epoch e_j in which $\mathsf{sc\_cert}_j$ was confirmed in C_1 (and note that we must have $j > 0$). $\mathcal{A}_1$ collects the public keys elected for the last $2k$ slots of epoch e_{j-1} according to the view of $P_{\mathbf{SC}}$ into a set keys_{j-1} and similarly for keys_j. $\mathcal{A}_1$ collects the pending cross-chain transactions of e_{j-1} according to the view of $P_{\mathbf{SC}}$ into $\mathsf{pending}_j'$, and creates the respective Merkle-tree commitment $\langle \mathsf{pending}_j' \rangle$. $\mathcal{A}_1$ checks whether the following *certificate violation* condition holds:

$$\mathsf{AVer}(m, avk^{j-1}, \sigma_j) \wedge \mathsf{ACheck}(\mathsf{keys}_{j-1}, avk^{j-1}) \wedge \left(\neg \mathsf{ACheck}(\mathsf{keys}_j, avk^j) \vee \langle \mathsf{pending}_j \rangle \neq \langle \mathsf{pending}_j' \rangle\right) \quad (2)$$

where avk^{j-1} is extracted from $\mathsf{sc\_cert}_{j-1}$ according to the view of $P_{\mathbf{SC}}$, unless $j = 1$ in which case avk^0 is known. If the condition (2) holds for no j then $\mathcal{A}_1$ returns FAILURE, otherwise it denotes by j^* the minimum j for which (2) holds and outputs the tuple $(m, \sigma_{j^*}, avk^{j^*-1}, \mathsf{keys}^{j^*-1})$.

The adversary $\mathcal{A}_2$. Like $\mathcal{A}_1$, $\mathcal{A}_2$ simulates the execution of $\mathcal{A}$ including two populations of maintainers and spawns parties according to the mandates of the environment $\mathcal{Z}$. For *all* these parties, $\mathcal{A}_2$ generates keys internally. When $\mathcal{A}$ requests that a transaction is created, $\mathcal{A}_2$ provides the signature with its respective private key. If $\mathcal{A}$ requests the corruption of a certain party, say P^*, then $\mathcal{A}_2$ provides the respective private key to $\mathcal{A}$.

For every time slot t of the execution, $\mathcal{A}_2$ inspects all pairs of honest parties such that $P_{\mathbf{MC}}$ is a $\mathbf{MC}$ maintainer and $P_{\mathbf{SC}}$ is a $\mathbf{SC}$ maintainer such that $\mathsf{L}_{\mathbf{MC}}^{P_{\mathbf{MC}}}[t] = \mathsf{L}_{\mathbf{MC}}^{\cup}[t]$ and $\mathsf{L}_{\mathbf{SC}}^{P_{\mathbf{SC}}}[t] = \mathsf{L}_{\mathbf{SC}}^{\cup}[t]$ and obtains the variables $\mathsf{L}_1, \mathsf{L}_2, \mathsf{C}_1, \mathsf{L}_1'$ as before. $\mathcal{A}_2$ examines whether $\mathsf{L} = \mathsf{merge}(\mathsf{L}_1, \mathsf{L}_2) \notin \mathbb{V}_{\mathfrak{A}}$, to deduce whether $\mathcal{A}$ has succeeded. If $\mathcal{A}_2$ is not able to find such a time slot t and parties $P_{\mathbf{MC}}, P_{\mathbf{SC}}$, it returns FAILURE. Let tx be as in $\mathcal{A}_1$. If $\mathsf{send}(\mathsf{tx}) \neq \mathbf{SC}$ or $\mathsf{lid}(\mathsf{tx}) \neq \mathbf{MC}$, then $\mathcal{A}_2$ returns FAILURE. Then tx references a certain certificate transaction $\mathsf{sc\_cert}_j = (j, \langle \mathsf{pending}_j \rangle, avk^j, \sigma_j)$ and uses a Merkle tree proof π which proves the inclusion of tx in $\mathsf{pending}_j$. If $\mathsf{sc\_cert}_j \notin \mathsf{L}_1'$, then $\mathcal{A}_2$ returns FAILURE. When $\mathsf{sc\_cert}_j$ was accepted by $P_{\mathbf{SC}}$, $\mathsf{pending}_j$ included a set of transactions $\mathbf{tx}$ in the view of $P_{\mathbf{SC}}$. If $\mathsf{tx} \in \mathbf{tx}$, then $\mathcal{A}_2$ returns FAILURE. Otherwise, the Merkle tree $\langle \mathsf{pending}_j \rangle$ was constructed from $\mathbf{tx}$, but a proof-of-inclusion π for $\mathsf{tx} \notin \mathbf{tx}$ was created. From this proof, $\mathcal{A}_2$ extracts a hash collision and returns it.

Probability analysis. Define the following events:

- SC-FORGE$[t]$: $\mathcal{A}$ is successful at slot t, i.e., $\pi_{\mathfrak{A}}\left(\mathsf{merge}(\{\mathsf{L}_i^{\cup}[t] : i \in \mathcal{S}_t\})\right) \notin \pi_{\mathcal{S}_t}(\mathbb{V}_{\mathfrak{A}})$.
- ATMS-FORGE: $\mathcal{A}_1$ finds an index j^* for which the condition (2) occurs.
- HASH-COLLISION: $\mathcal{A}_2$ finds a hash function collision.

Note that ledger states in the protocol only contain $\mathfrak{A}$-transactions, hence $\pi_{\mathfrak{A}}$ is the identity function and SC-FORGE$[t]$ is equivalent to $\mathsf{merge}\left(\{\mathbf{L}_i^{\cup}[t] : i \in \mathcal{S}_t\}\right) \notin \pi_{\mathcal{S}_t}(\mathbb{V}_{\mathfrak{A}})$. We will now show that for every t, the probability Pr[SC-FORGE$[t]$] is negligible. We distinguish two cases:

Case 1: $\mathcal{S}_t = \{\mathbf{MC}, \mathbf{SC}\}$. In this case Persistence holds for both $\mathbf{MC}$ and $\mathbf{SC}$, and $\pi_{\mathcal{S}_t}$ is the identity function. We deal with this case in two successive claims (both implicitly conditioning on being in Case 1). First we show that, if SC-FORGE$[t]$ occurs, then one of ATMS-FORGE, HASH-COLLISION occurs. Therefore applying a union bound, we will have that:

$$\Pr[\text{SC-FORGE}[t]] \leq \Pr[\text{ATMS-FORGE}] + \Pr[\text{HASH-COLLISION}] \ .$$

Second, we show that Pr[ATMS-FORGE] is negligible (and the negligibility of Pr[HASH-COLLISION] follows from our assumption that the hash function is collision resistant).

Claim 1a: SC-FORGE$[t] \Rightarrow$ ATMS-FORGE $\vee$ HASH-COLLISION.

Because persistence holds in both $\mathbf{MC}$ and $\mathbf{SC}$, we know that there exist two parties $P_{\mathbf{MC}}, P_{\mathbf{SC}}$ such that at slot t we have that $\mathbf{L}_{\mathbf{MC}}^{P_{\mathbf{MC}}}[t] = \mathbf{L}_{\mathbf{MC}}^{\cup}[t]$ and $\mathbf{L}_{\mathbf{SC}}^{P_{\mathbf{SC}}}[t] = \mathbf{L}_{\mathbf{SC}}^{\cup}[t]$, respectively. Therefore SC-FORGE$[t]$ implies

$$\mathsf{merge}(\{\mathbf{L}_{\mathbf{MC}}^{P_{\mathbf{MC}}}[t], \mathbf{L}_{\mathbf{SC}}^{P_{\mathbf{SC}}}[t]\}) \notin \mathbb{V}_{\mathfrak{A}} \ .$$

Let $\mathsf{tx}, \mathsf{tx}'$ be as in the definition of $\mathcal{A}_1$. By Lemma 2 and using $\mathbf{MC}$ and $\mathbf{SC}$ persistence, tx will exist and be an $\mathbf{MC}$-receiving transaction. Hence, $\mathsf{send}(\mathsf{tx}) = \mathbf{SC}$ and $\mathsf{rec}(\mathsf{tx}) = \mathsf{lid}(\mathsf{tx}) = \mathbf{MC}$. Therefore, tx' will also exist. If $\mathcal{A}_1$ finds the index j^* for which (2) is satisfied, then ATMS-FORGE has occured and the claim is established, so let us assume otherwise. Hence, for each certificate $\mathsf{sc\_cert}_j$ containing a message $m = (j, \langle \mathsf{pending}_j \rangle, avk^j)$, it holds that

$$\left(\mathsf{AVer}(m, avk^{j-1}, \sigma_j) \wedge \mathsf{ACheck}(\mathsf{keys}_{j-1}, avk^{j-1})\right) \Rightarrow \left(\mathsf{ACheck}(\mathsf{keys}_j, avk^j) \wedge \langle \mathsf{pending}_j \rangle = \langle \mathsf{pending}_j' \rangle\right) \ . \quad (3)$$

Therefore, we have a chain of certificates, each of which is signed with a valid key avk^{j-1} and attests to the validity of the next key avk^j. For all of these certificates, $\mathsf{AVer}(m, avk^{j-1}, \sigma_j)$ holds, as it has been verified by $P_{\mathbf{MC}}$. Furthermore, by an induction argument (where the base case comes from the construction of avk^0 and the induction step follows from (3)) we have $\mathsf{ACheck}(\mathsf{keys}_{j-1}, avk^{j-1})$ as well.

As tx' is a certificate transaction which appears last in the above chain (with some index $\mathsf{sc\_cert}_k$), the above implication also holds for tx', and so does its premise $\mathsf{AVer}(m, avk^{k-1}, \sigma_k) \wedge \mathsf{ACheck}(\mathsf{keys}_{k-1}, avk^{k-1})$. Therefore, the conclusion of the implication $\langle \mathsf{pending}_k \rangle = \langle \mathsf{pending}_k' \rangle$ holds. However, the sending transaction corresponding to tx has been proven to belong to the Merkle Tree $\langle \mathsf{pending}_k \rangle$ (as verified by $P_{\mathbf{MC}}$), but does not belong to $\mathsf{pending}_k'$ (by the selection of tx). This constitutes a Merkle Tree collision, which translates to a hash collision. The construction of $\mathcal{A}_2$ outputs exactly this collision, and in this case we deduce that $\mathcal{A}_2$ is successful and HASH-COLLISION follows.

Claim 1b: Pr[ATMS-FORGE] is negligible.
Suppose that ATMS-FORGE occurs. We will argue that, in this case, $\mathcal{A}_1$ will have computed an ATMS forgery, which is a negligible event by the assumption that the used ATMS is secure.

From the assumption that ATMS-FORGE has occurred, at epoch e_j we have that $\mathsf{AVer}(m, avk^{j-1}, \sigma_j)$ and $\mathsf{ACheck}(\mathsf{keys}_{j-1}, avk^{j-1})$, but $\neg\mathsf{ACheck}(\mathsf{keys}_j, avk^j)$ or $\langle\mathsf{pending}_j\rangle \neq \langle\mathsf{pending}'_j\rangle$. From Lemma 6 and using $\mathbb{A}_{\mathsf{hm}}(\mathbf{SC})[t]$, we deduce that in the last $2k$ slots of epoch e_{j-1}, at least $k+1$ must be honest. Since e_j is the earliest epoch in which this occurs, this means that keys_{j-1} corresponds to the last $2k$ slot leaders of epoch e_{j-1}, and all honest parties agree on the same $2k$ slot leaders. Hence, in the ATMS game, the number of keys in keys corrupted by the adversary through the use of the oracle $\mathcal{O}^{\mathsf{cor}}(\cdot)$ is less than k. Furthermore, since $\neg\mathsf{ACheck}(\mathsf{keys}_j, avk^j)$ or $\langle\mathsf{pending}_j\rangle \neq \langle\mathsf{pending}'_j\rangle$, the message m contains either an invalid future aggregate key, an invalid Merkle Tree root of outgoing cross-chain transactions, or both. Hence, no honest party will sign the message m for this epoch and therefore $|Q^{\mathsf{sig}}[m]| = 0$. Hence $q < k$, and $\mathcal{A}_1$ wins the ATMS security game.

Case 2: $\mathcal{S}_t \neq \{\mathbf{MC}, \mathbf{SC}\}$. If $\mathbf{MC} \notin \mathcal{S}_t$ then, since $\mathbb{A}_{\mathbf{MC}}[t] \Rightarrow \mathbb{A}_{\mathbf{SC}}[t]$, we have $\mathcal{S}_t = \emptyset$ and $\neg$SC-FORGE$[t]$, as $\epsilon \in \mathbb{V}_{\mathfrak{A}}$ by the *base property*. It remains to consider the case $\mathcal{S}_t = \{\mathbf{MC}\}$. Using $\mathbf{MC}$ persistence, by Lemma 5 we obtain $\mathsf{merge}(\{\mathbf{L}^{\cup}_{\mathbf{MC}}[t]\}) \in \pi_{\{\mathbf{MC}\}}(\mathbb{V}_{\mathfrak{A}})$, and hence SC-FORGE$[t]$ did not occur.

From the two above cases, we conclude that for every t, Pr[SC-FORGE$[t]$] $\leq \mathsf{negl}$. As the total number of slots is polynomial, we have shown that with overwhelming probability, we have that for all slots t and for all $\mathsf{A} \in \bigcup_{i \in \mathcal{S}_t} \mathsf{Assets}(\mathbf{L}_i)$, $\pi_{\mathsf{A}}\left(\mathsf{merge}\left(\{\mathbf{L}^{\cup}_i[t] \,:\, i \in \mathcal{S}_t\}\right)\right) \in \pi_{\mathcal{S}_t}(\mathbb{V}_{\mathsf{A}})$, concluding the proof. □

Lemmas 1 and 7 together directly imply the following theorem.

Theorem 1 (Pegging Security). *Consider the synchronous setting as defined in Section 2.1 with $2R$-semiadaptive corruptions as defined in Section 2.1. The construction of Section 4 using a secure ATMS and a collision resistant hash function is pegging secure with liveness parameter $u = 2k$ with respect to assumptions $\mathbb{A}_{\mathbf{MC}}$ and $\mathbb{A}_{\mathbf{SC}}$ defined above, and* merge, effect *and $\mathbb{V}_{\mathfrak{A}}$ defined in Section 4.2.*

Acknowledgements. This research was partially supported by H2020 project PRIVILEDGE #780477.

References

1. A. Back, M. Corallo, L. Dashjr, M. Friedenbach, G. Maxwell, A. Miller, A. Poelstra, J. Timón, and P. Wuille. Enabling blockchain innovations with pegged sidechains. 2014. `http://www.opensciencereview.com/papers/123/enablingblockchain-innovations-with-pegged-sidechains`.
2. C. Badertscher, P. Gaži, A. Kiayias, A. Russell, and V. Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. Cryptology ePrint Archive, Report 2018/378, 2018. `https://eprint.iacr.org/2018/378`. To appear at ACM CCS 2018.
3. M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In V. Ashby, editor, *ACM CCS 93*, pages 62–73. ACM Press, Nov. 1993.
4. E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *Manuscript.(2017). Slides at https://people. eecs. berkeley. edu/˜ alexch/docs/pcpip_bensasson. pdf*, 2017.
5. I. Bentov, R. Pass, and E. Shi. Snow white: Provably secure proofs of stake. Cryptology ePrint Archive, Report 2016/919, 2016. `http://eprint.iacr.org/2016/919`.
6. I. Bentov, R. Pass, and E. Shi. Snow white: Provably secure proofs of stake. *IACR Cryptology ePrint Archive*, 2016:919, 2016.
7. N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In S. Goldwasser, editor, *ITCS 2012*, pages 326–349. ACM, Jan. 2012.
8. A. Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-Diffie-Hellman-group signature scheme. In Y. Desmedt, editor, *PKC 2003*, volume 2567 of *LNCS*, pages 31–46. Springer, Heidelberg, Jan. 2003.
9. E. Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, 2016.

10. B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. Bulletproofs: Short proofs for confidential transactions and more.
11. V. Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 2014.
12. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, Oct. 2001.
13. B. David, P. Gazi, A. Kiayias, and A. Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In J. B. Nielsen and V. Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 66–98. Springer, Heidelberg, Apr. / May 2018.
14. J. Dilley, A. Poelstra, J. Wilkins, M. Piekarska, B. Gorlick, and M. Friedenbach. Strong federations: An interoperable blockchain solution to centralized third party risks. *CoRR*, abs/1612.05491, 2016.
15. Y. Dodis and A. Yampolskiy. A verifiable random function with short proofs and keys. In S. Vaudenay, editor, *PKC 2005*, volume 3386 of *LNCS*, pages 416–431. Springer, Heidelberg, Jan. 2005.
16. C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In E. F. Brickell, editor, *CRYPTO'92*, volume 740 of *LNCS*, pages 139–147. Springer, Heidelberg, Aug. 1993.
17. J. A. Garay, A. Kiayias, and N. Leonardos. The bitcoin backbone protocol: Analysis and applications. In E. Oswald and M. Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 281–310. Springer, Heidelberg, Apr. 2015.
18. A. Kiayias, N. Lamprou, and A.-P. Stouka. Proofs of proofs of work with sublinear complexity. In J. Clark, S. Meiklejohn, P. Y. A. Ryan, D. S. Wallach, M. Brenner, and K. Rohloff, editors, *FC 2016 Workshops*, volume 9604 of *LNCS*, pages 61–78. Springer, Heidelberg, Feb. 2016.
19. A. Kiayias, A. Miller, and D. Zindros. Non-interactive proofs of proof-of-work, 2017.
20. A. Kiayias, A. Russell, B. David, and R. Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In J. Katz and H. Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 357–388. Springer, Heidelberg, Aug. 2017.
21. E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford. OmniLedger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy*, pages 583–598. IEEE Computer Society Press, May 2018.
22. J. A. Kroll, I. C. Davey, and E. W. Felten. The economics of bitcoin mining, or bitcoin in the presence of adversaries. In *The Twelfth Workshop on the Economics of Information Security (WEIS 2013), Washington DC*, June 10-11 2013.
23. S. D. Lerner. Drivechains, sidechains and hybrid 2-way peg designs, 2016.
24. C. Li, T. Hwang, and N. Lee. Threshold-multisignature schemes where suspected forgery implies traceability of adversarial shareholders. In A. D. Santis, editor, *Advances in Cryptology - EUROCRYPT '94, Workshop on the Theory and Application of Cryptographic Techniques, Perugia, Italy, May 9-12, 1994, Proceedings*, volume 950 of *Lecture Notes in Computer Science*, pages 194–204. Springer, 1994.
25. L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena. A secure sharding protocol for open blockchains. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 17–30. ACM, 2016.
26. S. Micali. ALGORAND: the efficient and democratic ledger. *CoRR*, abs/1607.01341, 2016.
27. S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.
28. T. Ristenpart and S. Yilek. The power of proofs-of-possession: Securing multiparty signatures against rogue-key attacks. Cryptology ePrint Archive, Report 2007/264, 2007. `https://eprint.iacr.org/2007/264`.
29. P. Sztorc. Drivechain - the simple two way peg, November 2015. `http://www.truthcoin.info/blog/drivechain/`.
30. S. Thomas and E. Schwartz. A protocol for interledger payments. https://interledger.org/interledger.pdf.
31. G. Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151:1–32, 2014.
32. G. Wood. Polkadot: Vision for a heterogeneous multi-chain framework, 2016.
33. M. Zamani, M. Movahedi, and M. Raykova. RapidChain: A fast blockchain protocol via full sharding. Cryptology ePrint Archive, Report 2018/460, 2018. `https://eprint.iacr.org/2018/460`.
34. A. Zamyatin, N. Stifter, A. Judmayer, P. Schindler, E. Weippl, W. Knottenbelt, and A. Zamyatin. A wild velvet fork appears! inclusive blockchain protocol changes in practice. In *International Conference on Financial Cryptography and Data Security*. Springer, 2018.

Appendix

A The Diffuse Functionality

In the model described in Section 2.1 we employ the "Delayed Diffuse" functionality of [13], which we now describe in detail for completeness. The functionality is parameterized by $\Delta \in \mathbb{N}$ and denoted $\mathsf{DDiffuse}_\Delta$. It keeps rounds, executing one round per slot. $\mathsf{DDiffuse}_\Delta$ interacts with the environment $\mathcal{Z}$, stakeholders $U_1, \ldots, U_n$ and adversary $\mathcal{A}$, working as follows for each round: $\mathsf{DDiffuse}_\Delta$ maintains an incoming string for each party P_i that participates. A party, if activated, can fetch the contents of its incoming string, hence it behaves as a mailbox. Furthermore, parties can give an instruction to the functionality to diffuse a message. Activated parties can diffuse once per round.

When the adversary $\mathcal{A}$ is activated, it can: (a) read all inboxes and all diffuse requests and deliver messages to the inboxes in any order; (b) for any message m obtained via a diffuse request and any party P_i, $\mathcal{A}$ may move m into a special string $\mathsf{delayed}_i$ instead of the inbox of P_i. $\mathcal{A}$ can decide this individually for each message and each party; (c) for any party P_i, $\mathcal{A}$ can move any message from the string $\mathsf{delayed}_i$ to the inbox of P_i.

At the end of each round, the functionality ensures that every message that was either (a) diffused in this round and not put to the string $\mathsf{delayed}_i$ or (b) removed from the string $\mathsf{delayed}_i$ in this round is delivered to the inbox of party P_i. If a message currently present in $\mathsf{delayed}_i$ was originally diffused Δ slots ago, the functionality removes it from $\mathsf{delayed}_i$ and appends it to the inbox of party P_i.

Upon receiving $(\mathsf{Create}, U, \mathcal{C})$ from the environment, the functionality spawns a new stakeholder with chain $\mathcal{C}$ as its initial local chain (as in [20,13]).

B Adaptation to Other Proof-of-Stake Blockchains

Our construction can be adapted to work with other provably secure proof-of-stake blockchains discussed in Section 2.3: Ouroboros Praos [13], Ouroboros Genesis [2], Snow White [6], and Algorand [26]. Here we assume some familiarity with the considered protocols and refer the interested reader to the original papers for details.

B.1 Ouroboros Praos and Ouroboros Genesis

These protocols [13,2] are strongly related and differ from each other only in the chain-selection rule they use, which is irrelevant for our discussion here, hence we consider both of the protocols simultaneously. Ouroboros Praos was shown secure in the semi-synchronous model with fully adaptive corruptions (cf. Section 2.1) and this result extends to Ouroboros Genesis. Despite sharing the basic structure with Ouroboros, they differ in several significant points which we now outline.

The slot leaders are elected differently: Namely, each party for each slot evaluates a verifiable random function (VRF, [15]) using the secret key associated with their stake, and providing as inputs to the VRF both the slot index and the epoch randomness. If the VRF output is below a certain threshold that depends on the party's stake, then the party is an eligible slot leader for that slot, with the same consequences as in Ouroboros. Each leader then includes into the block it creates the VRF output and a proof of its validity to certify her eligibility to act as slot leader. The probability of becoming a slot leader is roughly proportional to the amount of stake the party controls, however now it is independent for each slot and each party, as it is evaluated locally by each stakeholder for herself. This local nature of the leader election implies that there will inevitably be some slots with no, or several, slot leaders. In each epoch j, the stake distribution used in Praos and Genesis for slot leader election corresponds to the distribution recorded in the ledger up to the last block of epoch $j-2$. Additionally, the *epoch randomness* η_j for epoch j is derived as a hash of additional VRF-values included into blocks from the first two thirds of epoch $j-1$ for this purpose by the respective slot leaders. Finally, the protocols use *key-evolving signatures* for block signing, and in each slot the honest parties are mandated to update their private key, contributing to their resilience to adaptive corruptions.

Ouroboros Praos was shown [13] to achieve persistence and liveness under weaker assumptions than Ouroboros, namely: (1) Δ-semi-synchronous communication (where Δ affects the security bounds but is unknown to the protocol); (2) majority of the stake is always controlled by honest parties. In particular, Ouroboros Praos is secure in face of fully adaptive corruptions without any corruption delay. Ouroboros Genesis provides the same guarantees as Praos, as well as several other features that will not be relevant for our present discusion.

Construction of Pegged Ledgers. The main difference compared to our treatment of Ouroboros would be in the construction of the sidechain certificate (cf. Section 4.3). The need for a modification is caused by the private, local leader selection using VRFs in these protocols, which makes it impossible to identify the set of slot leaders for the suffix of an epoch at the beginning of this epoch, as done for Ouroboros.

The sidechain certificate included in **MC** at the beginning of epoch j would hence contain the following, for parameters Q and T specified below:

1. the epoch index;
2. a Merkle commitment to the list of withdrawals as in the case of Ouroboros;
3. a Merkle commitment to the **SC** stake distribution $\overline{\mathsf{SD}}_j$;
4. a list of Q public keys;
5. Q inclusion proofs (with respect to $\overline{\mathsf{SD}}_{j-1}$ contained in the previous certificate) and Q VRF-proofs certifying that these Q keys belong to slot leaders of Q out of the last T slots in epoch $j-1$;
6. Q signatures from the above Q public keys on the above; these can be replaced by a single aggregate signature to save space on **MC**.

The parameters Q and T have to be chosen in such a way that with overwhelming probability, there will be a chain growth of at least Q blocks during the last T slots of epoch $j-1$, but the adversary controls Q slots in this period only with negligible probability (and hence at least one of the signatures will have to come from an honest slot leader). The existence of such constants for $T = \Theta(k)$ was shown in [2].

While the above sidechain certificate is larger (and hence takes more space on **MC**) than the one we propose for Ouroboros, a switch to Ouroboros Praos or Genesis would also bring several advantages. First off, both constructions would give us security in the semi-synchronous model with fully adaptive corruptions (as shown in [13,2]), and the use of Ouroboros Genesis would allow newly joining players to bootstrap from the mainchain genesis block only—without the need for a trusted checkpoint—as discussed extensively in [2].

B.2 Snow White

The high-level structure of Snow White execution is similar to the protocols we have already discussed: it contains epochs, committees that are sampled for each epoch based on the stake distribution recorded in the blockchain prior to that epoch, and randomness used for this sampling produced by hashing special nonce values included in previous blocks. Hence, our construction can be adapted to work with Snow White-based blockchains in a straightforward manner.

B.3 Algorand

Algorand does not aim for the so-called eventual consensus. Instead it runs a full Byzantine Agreement protocol for each block before moving to the next block, hence blocks are immediately finalized. Consider a setting with **MC** and **SC** both running Algorand. The main difficulty to address when constructing pegged ledgers is the continuous authentication of the sidechain certificate constructed by **SC**-maintainers for **MC** (other aspects, such as deposits from **MC** to **SC** work analogously to what we described above). As Algorand does not have epochs, and creating and processing a sidechain certificate for each block is overly demanding, a natural choice is to introduce a parameter R and execute this process only once every R blocks. Namely, every R blocks, the **SC**-maintainers produce a certificate that the **MC**-maintainers insert into the mainchain. This certificate most importantly contains:

1. a Merkle commitment to the list of withdrawals in the most recent R-block period;

2. a Merkle commitment to the full, most recent stake distribution $\overline{\mathsf{SD}}_j$ on **SC**;
3. a sufficient number of signatures from a separate committee certifying the above information, together with proofs justifying the membership of the signature's creators in the committee.

This additional committee is sampled from $\overline{\mathsf{SD}}_{j-1}$ (the stake distribution committed to in the previous sidechain certificate) via Algorand's private sortition mechanism such that the expected size of the committee is large enough to ensure honest supermajority (required for Algorand's security) translates into a strong honest majority within the committee. Note that the sortition mechanism also allows for a succinct proof of membership in the committee. The members of the committee then insert their individual signatures (signing the first two items in the certificate above) into the **SC** blockchain during the period of R blocks preceding the construction of the certificate. All the remaining mechanics of the pegged ledgers are a direct analogy of our construction above.