

Proof-of-Stake Sidechains

Peter Gaži

IOHK

Bratislava, Slovakia

peter.gazi@iohk.io

Aggelos Kiayias

University of Edinburgh

IOHK

Edinburgh, UK

akiayias@inf.ed.ac.uk

Dionysis Zindros

University of Athens

IOHK

Athens, Greece

dionyziz@di.uoa.gr

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 liveness and safety 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.

Index Terms—blockchain, sidechains, distributed systems

I. INTRODUCTION

Blockchain protocols and their most prominent application so far, cryptocurrencies like Bitcoin [25], 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 [14] (cf. [15], [25]), while a promising alternative is to use proof-of-stake (cf. [4], [11], [18], [24]). Application data typically consists of *transactions* indicating some transfer of value as in the case of Bitcoin [25]. The transfer of value can be conditioned on arbitrary predicates called *smart contracts* such as, for example, in Ethereum [9], [29].

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.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

¹See e.g., <https://blockstream.com/technology/> and [1].

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. [19], [23], [31].

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 [17], [32]. 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.² The firewall property formalises and generalises the concept of a blockchain *firewall* which was described in high level in [1]. Informally the blockchain firewall suggests that no more money can ever

²To 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.

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 [18] that underlies the Cardano blockchain, which is currently one of the largest pure PoS blockchains by market capitalisation,³ nevertheless we also discuss how to modify our construction to operate for Ouroboros Praos [11], Ouroboros Genesis [2], Snow White [5] and Algorand [24].

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 [20]. 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

³See <https://coinmarketcap.com>.

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 [18], 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 [3], [8] 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.

Related work. Sidechains were first proposed as a high level concept in [1]. Notable proposed implementations of the concept are given in [21], [27]. 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 [30], Cosmos [7], Blockstream’s Liquid [12] and Interledger [28]. 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., [22], 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 [16], [17]; 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.

II. PRELIMINARIES

A. Our Model

We employ the model from [11], which is in turn based on [18] and [15]. The formalization we use below captures

both synchronous and semi-synchronous communication; as well as both semi-adaptive and fully adaptive corruptions.

1) *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 $r \in \{1, 2, \dots\}$. We consider a UC-style [10] 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 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.

2) *(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 .

3) *(Semi-)Synchronous Communication*: We employ the “Delayed Diffuse” functionality DDiffuse_Δ given in [11] 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 DDiffuse_Δ in Appendix A for completeness.

We refer to the setting where honest parties communicate via DDiffuse_Δ 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.⁴

⁴As an example, we will be assuming that a majority of a certain pool of stake is controlled by uncorrupted parties.

B. 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. $C[i]$ indicates the i^{th} block, starting from $C[0]$, the genesis block. $C[-i]$ indicates the i^{th} block from the end, with $C[-1]$ being the tip of the blockchain. $C[i : j]$ 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 $C\{A : Z\}$ 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 $C^P[t]$ 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 $\text{tx}_1, \text{tx}_2, \dots$ 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. \mathbf{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 $\tilde{\mathbf{L}}^P[t]$ 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 [15]:

Persistence. For any two honest parties P_1, P_2 and two time slots $t_1 \leq t_2$, it holds $\mathbf{L}^{P_1}[t_1] \preceq \mathbf{L}^{P_2}[t_2]$.

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 $\text{tx} \in \mathbf{L}^P[t]$.

For a ledger \mathbf{L} that satisfies persistence at time t , we denote by $\mathbf{L}^U[t]$ (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, $\text{length}(\mathbf{L})$ denotes the length of the ledger \mathbf{L} , i.e., the number of transactions it contains.

C. Underlying Proof-of-Stake Protocols

For conciseness we present our construction on a generic PoS protocol based on Ouroboros PoS [18]. As we outline in Appendix C, our construction can be easily adapted to other provably secure proof-of-stake protocols: Ouroboros Praos [11], Ouroboros Genesis [2], Snow White [5], and Algorand [24]. While a full 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 C (or the original papers) for details on the other protocols.

1) *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 [18] 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:

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 $j - 1$;

GetRandomness(j) returns the randomness η_j for epoch j as derived during epoch $j - 1$;

ValidateConsensusLevel(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;

PickWinningChain(C, \mathcal{C}) applies the chain-selection rule: from a set of chains $\{C\} \cup \mathcal{C}$ it chooses the longest one that does not fork from the current chain C more than k blocks in the past;

SlotLeader(U, j, sl, SD_j, η_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 EpochIndex (resp. SlotIndex) always returns the index of the current epoch (resp. slot), and the event NewEpoch (resp. 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 [18] 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.

III. 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 L_1, L_2, \dots, 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 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 monotone predicate (which can only go from *true* to *false*) evaluated on the whole execution of the respective ledger protocol.

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 $\text{merge}(\cdot)$ that merges a set of ledger states $\mathcal{L} = \{L_1, L_2, \dots, L_n\}$ into a single ledger state denoted by $\text{merge}(\mathcal{L})$. We will discuss concrete instantiations of $\text{merge}(\cdot)$ later, for now simply assume that some canonical way of merging all ledger states into one is given.

Informally, at any point during the execution, our security definition only provides guarantees to the subset \mathcal{S} of ledgers that have their security assumptions \mathbb{A}_i 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}_A the valid transaction set of A , i.e., the set of all syntactically valid transactions involving A . For a ledger L we denote by \mathcal{T}_L the set of transactions that can be included into L . For notational convenience, we define $\mathcal{T}_{A,L} \triangleq \mathcal{T}_A \cap \mathcal{T}_L$. Let $\text{Assets}(L)$ denote the set of all assets that are supported by L . Formally, $\text{Assets}(L) \triangleq \{A : \mathcal{T}_{A,L} \neq \emptyset\}$.*

We assume that each transaction pertains to a particular asset and belongs to a particular ledger, i.e., for distinct $A_1 \neq A_2$ and $L_1 \neq L_2$, we have that $\mathcal{T}_{A_1} \cap \mathcal{T}_{A_2} = \emptyset$ and $\mathcal{T}_{L_1} \cap \mathcal{T}_{L_2} = \emptyset$. 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 \parallel represent concatenation.

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

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

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 $\pi_{\mathcal{X}}(L)$) 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 $\pi_{\mathcal{T}_A}$ and $\pi_{\bigcup_{i \in \mathcal{I}} \mathcal{T}_{L_i}}$, 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 L and L' , the effect mapping is a mapping of the form $\text{effect}_{L \rightarrow L'} : \mathcal{T}_L \rightarrow (\mathcal{T}_{L'} \cup \{\perp\})$. A transaction $\text{tx}' = \text{effect}_{L \rightarrow L'}(\text{tx}) \neq \perp$ is called the effect transaction of the transaction tx .*

Intuitively, for any transaction $\text{tx} \in \mathcal{T}_L$, the corresponding transaction $\text{effect}_{L \rightarrow L'}(\text{tx}) \in \mathcal{T}_{L'} \cup \{\perp\}$ identifies the necessary effect on ledger L' of the event of the inclusion of the transaction tx into the ledger L . With foresight, in an implementation of a system of ledgers where a “pegging” exists, the transaction $\text{effect}_{L \rightarrow L'}(\text{tx})$ has to be eventually valid and includable in L' in response to the inclusion of tx in 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 \perp to indicate that the transaction tx does not necessitate any action on 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 L and L' and an effect mapping $\text{effect}_{L \rightarrow L'}(\cdot)$, we refer to a transaction in \mathcal{T}_L that requires some effect on L' as a (L, L') -cross-ledger transfer transaction (or cross-ledger transfer for short). The set of all cross-ledger transfers is denoted by $\mathcal{T}_{L, L'}^{\text{cl}} \subseteq \mathcal{T}_L$, formally $\mathcal{T}_{L, L'}^{\text{cl}} \triangleq \{\text{tx} \in \mathcal{T}_L : \text{effect}_{L \rightarrow L'}(\text{tx}) \neq \perp\}$.*

Given ledger states L_1, L_2, \dots, 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 $\text{merge}(\cdot)$ function is any mapping taking a subset of ledger states $\mathcal{L} \subseteq \{L_1, L_2, \dots, L_n\}$ and producing a ledger state $\text{merge}(\mathcal{L})$ such that:*

- 1) **Partitioning.** *The ledger states in \mathcal{L} are disjoint sub-sequences of $\text{merge}(\mathcal{L})$ that cover the whole sequence $\text{merge}(\mathcal{L})$.*
- 2) **Topological soundness.** *For any $i \neq j$ such that $L_i, L_j \in \mathcal{L}$ and any two transactions $\text{tx} \in L_i$ and $\text{tx}' \in L_j$, if $\text{tx}' = \text{effect}_{L_i \rightarrow L_j}(\text{tx})$ then tx precedes tx' in $\text{merge}(\mathcal{L})$.*

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

Definition 7 (Correctness of \mathbb{V}_A). *A validity language \mathbb{V}_A is correct with respect to a mapping $\text{merge}(\cdot)$, if for any ledger states $\mathcal{L} \triangleq (L_1, \dots, L_n)$ such that $\pi_A(\text{merge}(\mathcal{L})) \in \mathbb{V}_A$, indices $i \neq j$, and any cross-ledger transfer $\text{tx} \in L_i \cap \mathcal{T}_{L_i, L_j}^{\text{cl}}$ such that $\text{effect}_{L_i \rightarrow L_j}(\text{tx}) = \text{tx}' \neq \perp$ is not in L_j , we have*

$$\pi_A(\text{merge}(L_1, \dots, L_i, \dots, L_j \parallel \text{tx}', \dots, L_n)) \in \mathbb{V}_A.$$

The above definition makes sure that if a cross-ledger transfer of an asset A is included into some ledger L_i and mandates an effect transaction on L_j , then the inclusion of this effect transaction will be consistent with \mathbb{V}_A . Note that this does not yet guarantee that the effect transaction will indeed be included into L_j , this will be provided by the liveness of 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 $\{L_i\}_{i \in [n]}$.

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

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

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

Ledger persistence: *For each $i \in \mathcal{S}_t$, L_i satisfies the persistence property.*

Ledger liveness: *For each $i \in \mathcal{S}_t$, L_i satisfies the liveness property parametrized by u .*

Firewall: *For all $A \in \bigcup_{i \in \mathcal{S}_t} \text{Assets}(L_i)$,*

$$\pi_A(\text{merge}(\{L_i^{\cup}[t] : i \in \mathcal{S}_t\})) \in \pi_{\mathcal{S}_t}(\mathbb{V}_A).$$

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.

IV. IMPLEMENTING PEGGED LEDGERS

We present a construction for pegged ledgers that is based on Ouroboros PoS [18], but also applicable to other PoS systems such as Snow White [5] and Algorand [24] (for a discussion of such adaptations, see Appendix C). 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 L_1 and L_2 . Direct observation of L_1 means that every node of L_2 follows and validates L_1 ; it is easy to see that this enables transfers from L_1 to L_2 . On the other hand, cross-chain certification of L_2 means that L_1 contains appropriate cryptographic information sufficient to validate data issued by the nodes following L_2 . This allows transfers of assets from L_2 , as long as they are certified, to be accepted by L_1 -nodes without following L_2 . The choice between direct observation and cross-chain certification can be made independently for each direction of transfers between L_1 and 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 L_1 are “produced by” coins from L_1 (in other words, the block-creating rights on L_1 are attributed based on the stake distribution recorded on L_1 only). In contrast, with merged staking, blocks on L_1 are produced either by coins on L_1 , or coins on L_2 that have, via their staking key, declared support of L_1 (but otherwise remain on L_2); see Figure 1. Also here, all 4 combinations are possible.

In our construction we choose an exemplary configuration between two ledgers L_1 and L_2 , so that direct observation is applied to L_1 , cross-chain certification to L_2 , independent-staking in L_1 and merged staking in L_2 . As a result, all

stakeholders in L_2 also keep track of chain development on L_1 (and hence run a full node for L_1) while the opposite is not necessary, i.e., L_1 stakeholders can be oblivious of transactions and blocks being added to 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 L_1 as the “mainchain” MC, and to 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 IV-A we introduce a novel cryptographic primitive, *ad-hoc threshold multisignature (ATMS)*, which is the fundamental building block for cross-chain certification. Afterwards, in Section IV-C 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 IV-B. Finally, we discuss specific instantiations of ATMS in Section V.

A. 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 IV-C 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 $\Pi = (PGen, Gen, Sig, Ver, AKey, ACheck, ASig, AVer)$ where:

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

$Gen(\mathcal{P})$ 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.

$Sig(sk_i, m)$ 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 .

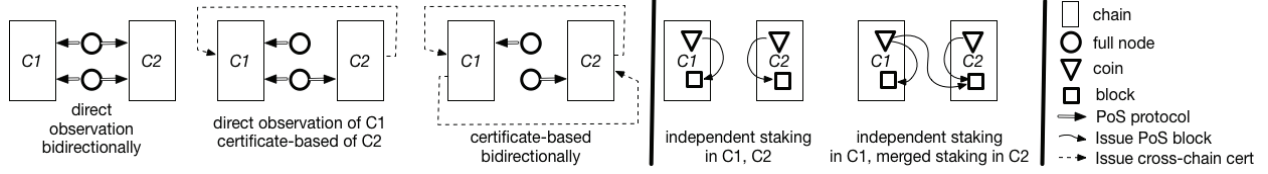


Fig. 1: Deployment options for PoS Sidechains.

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

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

$ACheck(\mathcal{VK}, avk)$ 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 .

$ASig(m, \mathcal{VK}, \langle (vk_1, \sigma_1), \dots, (vk_d, \sigma_d) \rangle)$ is the signature-aggregation algorithm that takes a message m , a sequence of public keys \mathcal{VK} and a sequence of d pairs $\langle (vk_1, \sigma_1), \dots, (vk_d, \sigma_d) \rangle$ 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).

$AVer(m, avk, \sigma)$ 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 $\mathcal{P} \leftarrow PGen(1^\kappa)$, let $(vk_1, sk_1), \dots, (vk_n, sk_n)$ be a sequence of keys generated via $Gen(\mathcal{P})$, let \mathcal{VK} be a sequence containing (not necessarily unique) keys from the above and avk be generated by invoking $avk \leftarrow AKey(\mathcal{VK})$. Let m be any message and let $\langle (vk_1, \sigma_1), \dots, (vk_d, \sigma_d) \rangle$ 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 $\sigma_i = Sig(sk_i, m)$. Let $\sigma \leftarrow ASig(m, \mathcal{VK}, \langle (vk_1, \sigma_1), \dots, (vk_d, \sigma_d) \rangle)$. The scheme Π is correct if for every such message and sequence the following hold:

- 1) $Ver(m, vk_i, \sigma_i)$ is true for all i ;
- 2) $ACheck(\mathcal{VK}, avk)$ is true;
- 3) $AVer(m, avk, \sigma)$ 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 $\Pi = (PGen, Gen, Sig, Ver, AKey, ACheck, ASig, AVer)$ is secure if for any PPT adversary \mathcal{A} and any polynomial p there exists some negligible function $negl$ such that $\Pr[ATMS_{\Pi, \mathcal{A}}(\kappa, p(\kappa)) = 1] < negl(\kappa)$.

Algorithm 1 The ATMS game

```

1: function ATMS( $\kappa, p$ )
2:    $\mathcal{VK} \leftarrow \epsilon; SK \leftarrow \epsilon; Q^{sig} \leftarrow \emptyset; Q^{cor} \leftarrow \emptyset$ 
3:    $\mathcal{P} \leftarrow PGen(1^\kappa)$ 
4:    $(m, \sigma, avk, keys) \leftarrow \mathcal{A}^{O^{gen}, O^{sig}(\cdot, \cdot), O^{cor}(\cdot)}(\mathcal{P})$ 
5:    $q \leftarrow 0$ 
6:   for  $vk$  in  $keys$  do
7:     if  $vk \notin \mathcal{VK} \vee vk \in Q^{sig}[m] \cup Q^{cor}$  then
8:        $q \leftarrow q + 1$ 
9:     end if
10:  end for
11:  return  $AVer(m, avk, \sigma) \wedge ACheck(keys, avk) \wedge q < t$ 
12: end function
13: function  $O^{gen}$ 
14:    $(vk, sk) \leftarrow Gen(\mathcal{P})$ 
15:    $\mathcal{VK} \leftarrow \mathcal{VK} \parallel vk$ 
16:    $SK \leftarrow SK \parallel sk$ 
17:   return  $vk$ 
18: end function
19: function  $O^{sig}(i, m)$ 
20:    $Q^{sig}[m] \leftarrow Q^{sig}[m] \cup \{\mathcal{VK}[i]\}$ 
21:   return  $Sig(SK[i], m)$ 
22: end function
23: function  $O^{cor}(i)$ 
24:    $Q^{cor} \leftarrow Q^{cor} \cup \{\mathcal{VK}[i]\}$ 
25:   return  $SK[i]$ 
26: end function

```

It is straightforward to see that if Π is a secure ATMS, then the tuple $(PGen, Gen, Sig, Ver)$ is a EUF-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 V.

B. 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.

1) *Instantiating $\mathbb{V}_{\mathfrak{A}}$* : The validity language $\mathbb{V}_{\mathfrak{A}}$ for the asset \mathfrak{A} considers two ledgers: the mainchain ledger $L_0 \triangleq \text{MC}$ and the sidechain ledger $L_1 \triangleq \text{SC}$. For this asset, every transaction $\text{tx} \in \mathcal{T}_{\mathfrak{A}}$ has the form $\text{tx} = (\text{txid}, \text{lid}, (\text{send}, \text{sAcc}), (\text{rec}, \text{rAcc}), v, \sigma)$, 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.
- $\text{lid} \in \{0, 1\}$ is the ledger index where the transaction belongs.
- $\text{send} \in \{0, 1\}$ is the index of the sender ledger L_{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.
- $\text{rec} \in \{0, 1\}$ is the index of the recipient ledger L_{rec} and rAcc is an account (again represented by a public key) on this ledger, this is the recipient account. We allow either $L_{\text{send}} = L_{\text{rec}}$, which denotes a *local transaction*, or $L_{\text{send}} \neq L_{\text{rec}}$, 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 $(\text{txid}, (\text{send}, \text{sAcc}), (\text{rec}, \text{rAcc}), v)$.

The correctness of lid is enforced by the ledgers, i.e., for both $i \in \{0, 1\}$ the set $\mathcal{T}_{\mathfrak{A}, L_i}$ only contains transactions with $\text{lid} = i$. 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 $\mathbb{V}_{\mathfrak{A}}$ (detailed as Algorithm 7 in Appendix B) processes the sequence of transactions $(\text{tx}_1, \text{tx}_2, \dots, \text{tx}_m)$ 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 $\text{send} = \text{rec}$), 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 $\text{send} \neq \text{rec}$. The first transaction appears in $\text{lid} = \text{send}$, while the second transaction appears in $\text{lid} = \text{rec}$. 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 L_{send} but has not yet arrived on L_{rec} then validity for this asset is *not* violated. All the validity language ensures is that appending the sidechain_receive transaction to the rec will eventually be a valid way to extend the receiving ledger, as long as the sidechain_send transaction has been included in send.

2) *Instantiating $\text{effect}_{L_i \rightarrow L_j}$* : For the simple asset \mathfrak{A} outlined above, every cross-ledger transfer is a “sending” transaction tx with $L_{\text{lid}} = L_{\text{send}} \neq L_{\text{rec}}$ appearing in L_{send} , and its effect transaction is a “receiving” transaction tx' with $L_{\text{lid}} = L_{\text{rec}} \neq L_{\text{send}}$ in L_{rec} that is otherwise identical (except for the different $\text{lid}' = 1 - \text{lid}$). Hence, we define $\text{effect}_{L_{\text{send}} \rightarrow L_{\text{rec}}}(\text{tx}) = \text{tx}'$ exactly for all these transactions and no other.

3) *Instantiating $\text{merge}(\cdot)$* : It is easy to construct a canonical function $\text{merge}(\cdot)$ 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.*

C. 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 2.

1) *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 L**” to refer to the action of broadcasting the transaction tx to the maintainers of the ledger L so that they include it

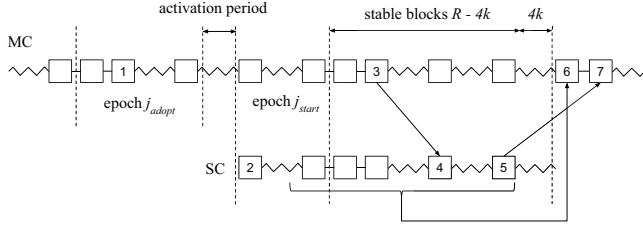


Fig. 2: Our sidechain construction. Blocks are shown as rectangles. Adjacent blocks connect with straight lines. Squiggly lines indicate some blocks are omitted. MC is at the top, SC at the bottom. Epochs are separated by dashed lines. $e_{j_{\text{adopt}}}$ is the epoch of first signalling; $e_{j_{\text{start}}}$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

in the ledger eventually as prescribed by the protocol. Unless indicated otherwise, we also denote by MC (resp. SC) the current *ledger state* of the ledger MC (resp. SC) as viewed by the party executing the protocol. Similarly, we denote by C_{MC} (resp. C_{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 C_{MC} .

2) *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.

3) *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 id_{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) $j_{\text{start}} - j_{\text{adopt}} > c_1$; (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 2 in Appendix B. First, every stakeholder U_i of MC (holding a key pair (vk, sk)) that supports the sidechain posts a special transaction (*sidechain_support*, $\text{id}_{\text{SC}}, vk, vk'$), 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 IV-C8 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 $\bar{G} = (\text{id}_{\text{SC}}, \overline{\text{SD}}_{j_{\text{start}}}, \bar{\eta}_{j_{\text{start}}} \triangleq H(\text{id}_{\text{SC}}, \eta_{j_{\text{start}}}), \mathcal{P}, \text{avk}^{j_{\text{start}}})$ for SC. $\eta_{j_{\text{start}}}$ is the randomness for leader election on MC in epoch j_{start} (derived on MC in epoch $j_{\text{start}} - 1$). It is reused to compute the initial sidechain randomness $\bar{\eta}_{j_{\text{start}}}$ as well, further $\bar{\eta}_{j'}$ for $j' > j_{\text{start}}$ are determined independently on SC using the Ouroboros coin-tossing protocol.⁵ Furthermore, \mathcal{P} and $\text{avk}^{j_{\text{start}}}$ are public parameters and an aggregated public key of an ATMS scheme; their creation and role is discussed in Section IV-C8 below. Note that \bar{G} is defined mostly for notational compatibility, as $\overline{\text{SD}}_{j_{\text{start}}}$ is empty at this point anyway. \bar{G} can be constructed as soon as $\eta_{j_{\text{start}}}$ is known and stable.

The stakeholders that adopted SC post into MC a transaction *sidechain_success*(id_{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 *sidechain_failure*(id_{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.

4) *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 below, with its exact code shown in Appendix B

⁵This can be interpreted as using MC to implement the setup functionality needed to bootstrap SC.

as Algorithm 3. 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 $\text{MC.ValidateConsensusLevel}$) and transaction validity (using the $\text{VERIFIER}_{\text{MC}}$ predicate given in Algorithm 8). Out of the remaining valid chains, he chooses his new state C_{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 C_{MC} (as per the predicate $\text{VERIFYTX}_{\text{MC}}$ given also in Algorithm 8), appends it to the chain C_{MC} and diffuses the result⁶ for other parties to adopt.

The maintenance procedure for SC is similar, hence we only describe here how it differs from Algorithm 3. 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.

The first difference is that all occurrences of MC and C_{MC} are naturally replaced by SC and C_{SC} , respectively. This also means that the validity of received chains (resp. transactions), determined on line 13 (resp. 21), is decided based on predicate $\text{VERIFIER}_{\text{SC}}(\cdot, C_{\text{MC}})$ (resp. $\text{VERIFYTX}_{\text{SC}}(\cdot)$) instead of the predicate $\text{VERIFIER}_{\text{MC}}(\cdot)$ (resp. $\text{VERIFYTX}_{\text{MC}}(\cdot)$). Additionally, note that $\text{VERIFYTX}_{\text{SC}}$ 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 $\text{VERIFYTX}_{\text{SC}}$ and $\text{VERIFIER}_{\text{SC}}$ are given in Algorithm 10 in Appendix B.

Second, instead of the stake distribution SD_j determined on line 6, a different distribution $\overline{\text{SD}}_j^*$ is determined to be used for slot leader selection in the j -th epoch of the sidechain. The distribution $\overline{\text{SD}}^*$ 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 $\overline{\text{SD}}$ 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 SlotLeader predicate on line 17 is $\overline{\text{SD}}_j^*$ 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 $sl \bmod R > 10k$, line 23 is replaced by $B \leftarrow (\text{prev}, \vec{\text{tx}}_{\text{valid}}, \sigma, \sigma_{\text{sc_cert}_{j+1}})$ where $\sigma_{\text{sc_cert}_{j+1}} = \text{Sig}_{sk}(\text{sc_cert}_{j+1})$ and j is the current epoch index.

5) *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 tx_{send} and tx_{rec} that both have $\text{send} = \text{MC}$, $\text{rec} = \text{SC}$, and all other fields are also identical, except that each tx_i for $i \in \{\text{send}, \text{rec}\}$ contains $\text{lid} = i$. The *sending transaction* tx_{send} is meant to be included in MC, while the *receiving transaction* tx_{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 tx_{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 pool_{SC} , initially set to zero. Whenever a tx_{send} is included into MC, they increase pool_{SC} by the amount of this transaction.

When tx_{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 tx_{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 tx_{send} into MC and consequently of tx_{rec} into SC, as guaranteed by the liveness of the underlying Ouroboros instances. The depositing algorithm code is shown in Algorithm 4 in Appendix B.

6) *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 tx_{send} and tx_{rec} , with the only difference that now they both have $\text{send} = \text{SC}$ and $\text{rec} = \text{MC}$. 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 tx_{send} with the correct transfer details as before. If tx_{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 $e_{j_{\text{send}}}$ to pass and $e_{j_{\text{send}}+1}$ to begin.

At the beginning of $e_{j_{\text{send}}+1}$, a special transaction called *sidechain certificate* $\text{sc_cert}_{j_{\text{send}}+1}$ is generated by the maintainers of SC. It contains: (i) a Merkle-tree commitment to all withdrawal transactions tx_{send} that were included into SC during last $4k$ slots of epoch $j_{\text{send}} - 1$ and the first $R - 4k$

⁶ As in [11], [18], we simplify our presentation by diffusing the complete chains, although a practical implementation would only diffuse the block B .

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 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 sc_cert is broadcast into the MC network to be included into MC at the beginning of $e_{j_{\text{send}}+1}$ by the first honest slot leader.

The stakeholder who wishes to withdraw their money into MC now creates and diffuses the transaction tx_{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 pool_{SC} . If included, MC-maintainers decrease the value of pool_{SC} by the amount of this transaction. The code of the withdrawal algorithm is illustrated in Algorithm 5 in Appendix B.

7) *The certificate transaction*: We now describe the construction of the sc_cert transaction, also called the *sidechain certificate*, formally described in Algorithm 6 in Appendix B. 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 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 IV-A. Namely, the 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 sc_cert_{j-1} .

sc_cert_j is generated by SC-maintainers and contains:

- **The epoch index j .**
- **The pending transactions from SC to MC.** Let $\vec{\text{tx}}$ 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 $\vec{\text{tx}}$ that have $\text{SC} = \text{send} \neq \text{rec} = \text{MC}$ are picked up and combined into a list pending_j (sorted in the same order as in SC). Let $\langle \text{pending}_j \rangle$ 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 sc_cert_j is therefore a tuple $(j, \langle \text{pending}_j \rangle, \text{avk}^j, \sigma_j)$, where σ_j is an ATMS signature on the preceding elements that verifies using avk^{j-1} .

The certificate sc_cert_{j+1} is constructed as follows: Both the stake distribution $\overline{\text{SD}}_{j+1}^*$ 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) sc_cert_{j+1} into the blocks they create. Given the deterministic construction of 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 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 sc_cert_{j+1} and serve it to the maintainers of MC for inclusion.

8) *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, $\mathcal{P} \leftarrow \text{PGen}(1^\kappa)$ is ran. Stakeholders generate their keys by invoking $(sk_i, vk_i) \leftarrow \text{Gen}(\mathcal{P})$. In case $\text{Gen}(\cdot)$ 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 $H(\text{ats_init}, \eta_{j_{\text{start}}})$ where “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 V are only of this type.

For the induction base, \mathcal{P} is published as part of the Genesis block \bar{G} . Each time an MC stakeholder U_i posts the sidechain_support message to MC, he also includes an ATMS key vk_i . Subsequently, when the SC is initialised, the stake distribution $\overline{\text{SD}}_{j_{\text{start}}}^*$ is known to the MC participants. Hence, based on $\overline{\text{SD}}_{j_{\text{start}}}^*$ and $\bar{\eta}_{j_{\text{start}}}$, 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 $j \geq j_{\text{start}}$ 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 $\text{avk}^{j_{\text{start}}}$ from their public keys and insert it into the genesis block \bar{G} of SC. They can also serve it as a special transaction to the MC-maintainers to include into the mainchain. The correctness of $\text{avk}^{j_{\text{start}}}$ can be readily verified by anyone following the mainchain using the procedure ACHECK of the used ATMS.

For the induction step, consider an epoch $j > j_{\text{start}}$ 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 $m = (j, \langle \text{pending}_j \rangle, \text{avk}^j)$ using their private key sk_i^{j-1} by running $\text{Sig}(sk_i^{j-1}, m)$, and include this signature into the block they create. The rest of the SC maintainers will verify that the epoch index, avk^j and $\langle \text{pending}_j \rangle$ are correct (by ensuring $\text{ACheck}(\mathcal{VK}^j, \text{avk}^j)$ 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 $\langle \text{pending}_j \rangle$) and that s_i^j

is valid by running $\text{Ver}(m, vk_i^{j-1}, s_i^j)$, 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 $\sigma^j \leftarrow \text{ASig}(m, \{(s_i^j, vk_i^{j-1})\}, \text{keys}^j)$. This combined signature is then diffused as part of sc_cert_j on the mainchain network. The mainchain maintainers verify that it has been signed by the sidechain maintainers by checking that $\text{AVer}(m, avk^{j-1}, \sigma^j)$ evaluates to *true* and include it in a mainchain block. This effectively hands over control to the new committee.

V. CONSTRUCTING AD-HOC THRESHOLD MULTISIGNATURES

We now present an instantiation of the ATMS primitive based on the multisignature scheme Π_{MGS} from [6]. Note that other constructions are possible: we defer a discussion of a plain signature-based ATMS, as well as an ATMS based on proofs of knowledge, to the full version of this paper.

We make use of a homomorphic property of Π_{MGS} : any d individual signatures $\sigma_1, \dots, \sigma_d$ created using secret keys belonging to (not necessarily unique) public keys vk_1, \dots, 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, $\text{ACheck}(\mathcal{VK}, avk)$ 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 $\{vk_i\}_{i=1}^{n-d}$ and produces an aggregate signature

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

where $\pi_{\widehat{vk_i}}$ denotes the (unique) inclusion proof of $\widehat{vk_i}$ in the Merkle commitment $\langle \{vk_i\}_{i=1}^d \cup \{\widehat{vk_i}\}_{i=1}^{n-d} \rangle$. 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 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) [26] 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).

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 $r = k-1$.

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 [18]) and for the signature of [6] we have in bits: $|vk_i| = 272$, $|\sigma_i| = 528$ (N. Di Prima, V. Hanquez, personal communication, 16 Mar 2018), with $|vk_i + \text{POP}| = |vk_i| + |\sigma_i| = 800$ 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 $|avk| + |\sigma| + |\langle \text{pending} \rangle| = |vk_i + \text{POP}| + 2|H(\cdot)| + |\sigma_i| = 800 + 512 + 528 = 1840$ bits per epoch. In a case where 10% of participants fail to sign, the size will be $|avk| + |\sigma| = |vk_i + \text{POP}| + 2|H(\cdot)| + |\sigma_i| + 0.1 \cdot 2 \cdot k(|vk_i + \text{POP}| + \log(k)|H(\cdot))) = 800 + 512 + 528 + 432 \cdot (500 + 12 \cdot 256) = 1544944$, or about 190 KB per epoch (which is approximately 5 days).

VI. SECURITY

Our main theorem below states that under natural assumptions, the construction from Section IV implements a secure 2-way peg according to Definition 8.

Let $\mathbb{A}_{\text{hm}}(\mathbf{L})$ denote the honest-majority assumption for an Ouroboros ledger \mathbf{L} . Namely, $\mathbb{A}_{\text{hm}}(\mathbf{L})$ postulates that in each slot t , the majority of stake in the stake distribution used to sample slot leader for slot t in \mathbf{L} is controlled by honest parties (note that the distribution in question is SD and $\overline{\text{SD}}^*$ for MC and SC , respectively). The assumption \mathbb{A}_{MC} we consider for MC is precisely $\mathbb{A}_{\text{hm}}(\text{MC})$, while the assumption \mathbb{A}_{SC} for SC is $\mathbb{A}_{\text{hm}}(\text{MC}) \wedge \mathbb{A}_{\text{hm}}(\text{SC})$. The reason that $\mathbb{A}_{\text{SC}} \Rightarrow \mathbb{A}_{\text{MC}}$ 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 $\mathbb{A}_{\text{hm}}(\text{SC})$ (now with respect to $\overline{\text{SD}}$) after the weaning slot and the two chains will become sidechains of each other.

Theorem 1. *Consider the synchronous setting with $2R$ -semiadaptive corruptions defined in Section II-A. The construction of Section IV, using a secure ATMS and a collision resistant hash function, is pegging secure with liveness parameter $u = 2k$ with respect to assumptions \mathbb{A}_{MC} and \mathbb{A}_{SC} , and merge, effect and \mathbb{V}_{A} defined in Section IV-B.*

The proof of Theorem 1 is deferred to the full version of the paper.

Acknowledgements. This research was partially supported by H2020 project PRIVILEGE #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. Gazi, 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] 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.
- [4] 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>.
- [5] I. Bentov, R. Pass, and E. Shi. Snow white: Provably secure proofs of stake. *IACR Cryptology ePrint Archive*, 2016:919, 2016.
- [6] 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.
- [7] E. Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, 2016.
- [8] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. Bulletproofs: Short proofs for confidential transactions and more.
- [9] V. Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 2014.
- [10] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, Oct. 2001.
- [11] 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.
- [12] J. Dille, 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.
- [13] 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.
- [14] 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.
- [15] 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.
- [16] 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.
- [17] A. Kiayias, A. Miller, and D. Zindros. Non-interactive proofs of proof-of-work, 2017.
- [18] 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.
- [19] 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.
- [20] 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.
- [21] S. D. Lerner. Drivechains, sidechains and hybrid 2-way peg designs, 2016.
- [22] 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.
- [23] 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.
- [24] S. Micali. ALGORAND: the efficient and democratic ledger. *CoRR*, abs/1607.01341, 2016.
- [25] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.
- [26] 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>.
- [27] P. Sztorc. Drivechain - the simple two way peg, November 2015. <http://www.truthcoin.info/blog/drivechain/>.
- [28] S. Thomas and E. Schwartz. A protocol for interledger payments. <https://interledger.org/interledger.pdf>.
- [29] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151:1–32, 2014.
- [30] G. Wood. Polkadot: Vision for a heterogeneous multi-chain framework, 2016.
- [31] 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>.
- [32] A. Zamyatin, N. Stifter, A. Judmayer, P. Schindler, E. Weippl, W. Knotenbelt, 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 II-A we employ the “Delayed Diffuse” functionality of [11], which we now describe in detail for completeness. The functionality is parameterized by $\Delta \in \mathbb{N}$ and denoted DDiffuse_Δ . It keeps rounds, executing one round per slot. DDiffuse_Δ interacts with the environment \mathcal{Z} , stakeholders U_1, \dots, U_n and adversary \mathcal{A} , working as follows for each round: DDiffuse_Δ 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 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 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 delayed_i or (b) removed from the string delayed_i in this round is delivered to the inbox of party P_i . If a message currently present in delayed_i was originally diffused Δ slots ago, the functionality removes it from delayed_i and appends it to the inbox of party P_i .

Upon receiving $(\text{Create}, U, \mathcal{C})$ from the environment, the functionality spawns a new stakeholder with chain \mathcal{C} as its initial local chain (as in [11], [18]).

B. Omitted Algorithms

Here we spell out the procedures for initializing the sidechain (Alg. 2), maintaining the mainchain (Alg. 3), depositing funds to SC (Alg. 4) and withdrawing them back to MC (Alg. 5), as well as creating a sidechain certificate (Alg. 6). Additionally, we also describe the membership-deciding procedure for the validity language $\mathbb{V}_{\mathcal{A}}$ in Alg. 7, and the auxiliary verifier algorithms for the mainchain and the sidechain in Algorithms 8 and 10, respectively.

Algorithm 2 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.

```

1: upon SidechainAdoption(idSC) do
2:   sidechain_state[idSC]  $\leftarrow$  initializing
3:    $(vk', sk') \leftarrow \text{Gen}(\mathcal{P})$ 
4:    $\sigma \leftarrow \text{Sig}_{sk}(\text{sidechain\_support}, \text{id}_{SC}, vk, vk')$ 
5:   post  $(\text{sidechain\_support}, \text{id}_{SC}, vk, vk', \sigma)$  to MC
6: end upon
7: upon MC.NewEpoch() do
8:    $j \leftarrow \text{MC.EpochIndex}()$ 
9:   if sidechain_state[idSC] = initializing then
10:    if ActivationFailure() then
11:      sidechain_state[idSC]  $\leftarrow$  failed
12:    post sidechain_failure(idSC) to MC
13:    else if ActivationSuccess() then
14:      sidechain_state[idSC]  $\leftarrow$  initialized
15:       $j_{\text{start}} \leftarrow \text{ActivationEpoch}()$ 
16:      Post sidechain_success(idSC) to MC
17:    end if
18:  end if
19:  if sidechain_state[idSC] = initialized  $\wedge j = j_{\text{start}}$  then
20:     $\bar{\eta}_{j_{\text{start}}} \leftarrow H(\text{id}_{SC}, \eta_{j_{\text{start}}})$ 
21:     $\mathcal{VK}_{j_{\text{start}}} \leftarrow 2k$  last slot leaders of  $e_{j_{\text{start}}}$  in SC
22:     $avk_{j_{\text{start}}} \leftarrow \text{AKey}(\mathcal{VK}_{j_{\text{start}}})$ 
23:     $\bar{G} \leftarrow (\text{id}_{SC}, \bar{SD}_{j_{\text{start}}}, \bar{\eta}_{j_{\text{start}}}, \mathcal{P}, avk_{j_{\text{start}}})$ 
24:     $C_{SC} \leftarrow (\bar{G})$ 
25:  end if
26: end upon

```

C. 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 II-C: Ouroboros Praos [11], Ouroboros Genesis [2], Snow White [5], and Algorand [24]. Here we assume some familiarity with the considered protocols and refer the interested reader to the original papers for details.

1) *Ouroboros Praos and Ouroboros Genesis*: These protocols [2], [11] 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 II-A) and this result extends to Ouroboros Genesis.

Algorithm 3 Mainchain maintenance procedures.

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

```

1: upon MC.NewSlot() do
2:    $sl \leftarrow \text{MC.SlotIndex}()$ 
3:    $\triangleright$  First slot of a new epoch
4:   if  $sl \bmod R = 1$  then
5:      $j \leftarrow \text{MC.EpochIndex}()$ 
6:      $SD_j \leftarrow \text{MC.GetDistr}(j)$ 
7:      $\eta_j \leftarrow \text{MC.GetRandomness}(j)$ 
8:   end if
9:    $\mathcal{C} \leftarrow$  chains received via Diffuse
10:   $\triangleright$  Consensus-level validation
11:   $C_{\text{valid}} \leftarrow \text{Filter}(\mathcal{C}, \text{MC.ValidateConsensusLevel})$ 
12:   $\triangleright$  Transaction-level validation
13:   $C_{\text{validtx}} \leftarrow \text{Filter}(C_{\text{valid}}, \text{VERIFIER}_{\text{MC}}(\cdot))$ 
14:   $\triangleright$  Apply chain selection rule
15:   $C_{\text{MC}} \leftarrow \text{MC.PickWinningChain}(C_{\text{MC}}, C_{\text{validtx}})$ 
16:   $\triangleright$  Decide slot leadership based on  $SD_j$  and  $\eta_j$ 
17:  if MC.SlotLeader( $U, j, sl, SD_j, \eta_j$ ) then
18:     $\text{prev} \leftarrow H(C_{\text{MC}}[-1])$ 
19:     $\vec{\text{tx}}_{\text{state}} \leftarrow$  transaction sequence in  $C_{\text{MC}}$ 
20:     $\vec{\text{tx}} \leftarrow$  current transactions in mempool
21:     $\vec{\text{tx}}_{\text{valid}} \leftarrow \text{VERIFYTX}_{\text{MC}}(\vec{\text{tx}}_{\text{state}} \parallel \vec{\text{tx}})[|\vec{\text{tx}}_{\text{state}}|:]$ 
22:     $\sigma \leftarrow \text{Sig}_{sk}(\text{prev}, \vec{\text{tx}}_{\text{valid}})$ 
23:     $B \leftarrow (\text{prev}, \vec{\text{tx}}_{\text{valid}}, \sigma)$ 
24:     $C_{\text{MC}} \leftarrow C_{\text{MC}} \parallel B$ 
25:    Diffuse( $C_{\text{MC}}$ )
26:  end if
27: end upon

```

Algorithm 4 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.

```

1: function Send(sAcc, rAcc,  $v$ )
2:    $\text{txid} \xleftarrow{\$} \{0, 1\}^k$ 
3:    $\sigma \leftarrow \text{Sig}_{sk}(\text{txid}, \text{MC}, (\text{MC}, \text{sAcc}), (\text{SC}, \text{rAcc}), v)$ 
4:    $\text{tx}_{\text{send}} \leftarrow (\text{txid}, \text{MC}, (\text{MC}, \text{sAcc}), (\text{SC}, \text{rAcc}), v, \sigma)$ 
5:   post  $\text{tx}_{\text{send}}$  to MC
6: end function
7: function Receive(txid, sAcc, rAcc,  $v$ )
8:   wait until  $\text{tx}_{\text{send}} \in \text{MC}$ 
9:    $\sigma \leftarrow \text{Sig}_{sk}(\text{txid}, \text{SC}, (\text{MC}, \text{sAcc}), (\text{SC}, \text{rAcc}), v)$ 
10:   $\text{tx}_{\text{rec}} \leftarrow (\text{txid}, \text{SC}, (\text{MC}, \text{sAcc}), (\text{SC}, \text{rAcc}), v, \sigma)$ 
11:  post  $\text{tx}_{\text{rec}}$  to SC
12: end function

```

Algorithm 5 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.

```
1: function Send( $sAcc, rAcc, v$ )  $\triangleright$  Send  $v$  from  $sAcc$  on SC to  $rAcc$  on MC
2:    $txid \xleftarrow{\$} \{0, 1\}^k$ 
3:    $\sigma \leftarrow \text{Sig}_{sk}(txid, SC, (SC, sAcc), (MC, rAcc), v)$ 
4:    $tx_{send} \leftarrow (txid, SC, (SC, sAcc), (MC, rAcc), v, \sigma)$ 
5:   post  $tx_{send}$  to SC
6: end function
7: function Receive( $txid, sAcc, rAcc, v$ )
8:   wait until  $tx_{send} \in C_{SC}$ 
9:    $j' \leftarrow$  epoch where  $C_{SC}$  contains  $tx_{send}$ 
10:  if ( $tx_{send}$  included in slot  $sl \leq R - 4$  of  $e_{j'}$ ) then
11:     $j_{send} \leftarrow j'$ 
12:  else
13:     $j_{send} \leftarrow j' + 1$ 
14:  end if
15:  wait until  $sc\_cert_{j_{send}+1} \in C_{MC}$ 
16:   $\pi \leftarrow$  Merkle-tree proof of  $tx_{send}$  in  $sc\_cert_{j_{send}+1}$ 
17:   $\sigma \leftarrow \text{Sig}_{sk}(txid, MC, (SC, sAcc), (MC, rAcc), v, \pi)$ 
18:   $tx_{rec} \leftarrow (txid, MC, (SC, sAcc), (MC, rAcc), v, \pi, \sigma)$ 
19:  post  $tx_{rec}$  to MC
20: end function
```

Algorithm 6 Constructing sidechain certificate sc_cert .

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

```
1: function ConstructSCCert( $j$ )
2:    $T \leftarrow$  last  $4k$  slots of  $e_{j-1}$  and first  $R - 4k$  slots of  $e_j$ 
3:    $\vec{tx} \leftarrow$  transactions included in SC during  $T$ 
4:    $pending_{j+1} \leftarrow \{tx \in \vec{tx} : tx.send \neq tx.rec\}$ 
5:    $\mathcal{VK}_{j+1} \leftarrow$  keys of last  $2k$  SC slot leaders in  $e_{j+1}$ 
6:    $avk^{j+1} \leftarrow \text{AKey}(\mathcal{VK}_{j+1})$ 
7:    $m \leftarrow (\langle pending_{j+1} \rangle, avk^{j+1})$ 
8:    $\mathcal{VK}_j \leftarrow$  keys of last  $2k$  SC slot leaders for  $e_j$ 
9:    $\sigma_{j+1} \leftarrow \text{ASig}(m, \{(vk_i, \sigma_i)\}_{i=1}^d, \mathcal{VK}_j)$ 
10:   $sc\_cert_{j+1} \leftarrow (\langle pending_{j+1} \rangle, avk^{j+1}, \sigma_{j+1})$ 
11:  return  $sc\_cert_{j+1}$ 
12: end function
```

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, [13]) 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

Algorithm 7 The transaction sequence validator (membership-deciding algorithm for $\mathbb{V}_{\mathcal{A}}$).

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

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

Algorithm 8 The MC verifier.

```

1: function VERIFYTXMC( $\vec{tx}$ )
2:    $bal \leftarrow$  initial stake;  $avk \leftarrow$  initial aggregate key
3:    $seen \leftarrow \emptyset$ ;  $pool \leftarrow 0$ ;  $pfs\_mtrs \leftarrow \emptyset$ ;  $pfs\_used \leftarrow \emptyset$ 
4:   for  $tx \in \vec{tx}$  do
5:     if  $type(tx) = sc\_cert$  then
6:        $(m, \sigma) \leftarrow tx$ 
7:       if  $\neg AVer(m, avk, \sigma)$  then
8:         continue
9:       end if
10:       $(txs\_root, avk') \leftarrow m$ 
11:       $avk \leftarrow avk'$ 
12:       $pfs\_mtrs[txs\_root] \leftarrow true$ 
13:    else
14:       $(txid, lid, (send, sAcc), (rec, rAcc), v, \sigma) \leftarrow tx$ 
15:       $m \leftarrow (txid, lid, (send, sAcc), (rec, rAcc), v)$ 
16:      if  $\neg Ver(m, sAcc, \sigma) \vee seen[txid] \neq 0$  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  $send \neq rec$  then
25:         $\pi \leftarrow tx.\pi$ 
26:         $(mtr, inclusion\_pf) \leftarrow \pi$ 
27:        if  $\pi \in pfs\_used \vee mtr \notin pfs\_mtrs \vee$ 
28:           $\neg MTR-VER(mtr, inclusion\_pf)$  then
29:            continue
30:          end if
31:        if  $lid = rec$  then
32:           $bal[rAcc] += v$ 
33:        end if
34:        if  $send \neq rec$  then
35:          if  $lid = send$  then
36:             $pool -= v$ 
37:          else
38:             $pool += v$ 
39:          end if
40:        end if
41:      end if
42:       $seen \leftarrow seen \parallel tx$ 
43:    end for
44:    return  $seen$ 
45: end function
46: function VERIFIERMC( $C_{mc}$ )
47:    $\vec{tx} \leftarrow \emptyset$ 
48:   for  $B \in C_{mc}$  do
49:     for  $tx \in B$  do
50:        $\vec{tx} \leftarrow \vec{tx} \parallel tx$ 
51:     end for
52:   end for
53:   return  $\vec{tx} \neq VERIFYTX_{MC}(\vec{tx})$ 
54: end function

```

Algorithm 9 The SC transaction annotation.

```

1: function ANNOTATETXSC( $C_{sc}, C_{mc}$ )
2:    $\vec{tx} \leftarrow \emptyset$ 
3:   for each time slot  $t$  do
4:      $\vec{tx}' \leftarrow \epsilon$ 
5:     if  $C_{sc}$  has a block generated at slot  $t$  then
6:        $B \leftarrow$  the block in  $C_{sc}$  generated at  $t$ 
7:       for  $tx \in B$  do
8:          $\vec{tx}' \leftarrow \vec{tx}' \parallel tx$ 
9:       end for
10:    end if
11:    if  $C_{mc}$  has a block generated at slot  $t$  then
12:       $B \leftarrow$  the block in  $C_{mc}$  generated at  $t$ 
13:      for  $tx \in B$  do
14:         $\vec{tx}' \leftarrow \vec{tx}' \parallel tx$ 
15:      end for
16:    end if
17:    for  $tx \in \vec{tx}'$  do
18:       $\triangleright$  Mark the time of each  $tx$  in  $\vec{tx}'$ 
19:       $tx.t \leftarrow t$ 
20:    end for
21:     $\vec{tx} \leftarrow \vec{tx} \parallel \vec{tx}'$ 
22:  end for
23:  return  $\vec{tx}$ 
24: end function

```

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 [11] 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 discussion.

Construction of Pegged Ledgers. The main difference compared to our treatment of Ouroboros would be in the construction of the sidechain certificate (cf. Section IV-C7). 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;

Algorithm 10 The SC verifier and its respective transaction verifier.

```

1: function VERIFIERSC(Csc, Cmc)
2:    $\vec{tx} \leftarrow \text{ANNOTATE}_{TX_{SC}}(C_{sc}, C_{mc})$ 
3:   return  $\vec{tx} \neq \text{VERIFY}_{TX_{SC}}(\vec{tx})$ 
4: end function
5: function VERIFYTXSC( $\vec{tx}$ )
6:   bal[MC]  $\leftarrow$  Initial MC stake
7:   bal[SC]  $\leftarrow$  Initial SC stake
8:   mc_outgoing_tx  $\leftarrow \emptyset$ ; seen  $\leftarrow \emptyset$ 
9:   for tx  $\in \vec{tx}$  do
10:    (txid, lid, (send, sAcc), (rec, rAcc), v,  $\sigma$ , t)  $\leftarrow$  tx
11:    m  $\leftarrow$  (txid, lid, (send, sAcc), (rec, rAcc), v)
12:    if  $\neg \text{Ver}(sAcc, m, \sigma) \vee \text{seen}[txid] \neq 0$  then
13:      continue
14:    end if
15:    if lid = send then
16:      if bal[send][sAcc] - v < 0 then
17:        continue
18:      end if
19:      if lid = MC  $\wedge$  send  $\neq$  rec then
20:        mc_outgoing_tx[txid]  $\leftarrow t + 2k$ 
21:      end if
22:    end if
23:    if lid = rec then
24:      if send  $\neq$  rec then
25:         $\triangleright$  Effect pre-image tx immature
26:        if t < mc_outgoing_tx[txid] then
27:          continue
28:        end if
29:      end if
30:      bal[rec][rAcc] += v
31:    end if
32:    if lid = send then
33:      bal[send][sAcc] -= v
34:    end if
35:    seen  $\leftarrow$  seen  $\parallel$  tx
36:  end for
37:  return seen
38: end function

```

- 3) a Merkle commitment to the SC stake distribution \overline{SD}_j ;
- 4) a list of Q public keys;
- 5) Q inclusion proofs (with respect to \overline{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 [2], [11]), 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].

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.

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{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{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.