Mining for Privacy: How to Bootstrap a Snarky Blockchain

Thomas Kerber(✉), Aggelos Kiayias, and Markulf Kohlweiss

The University of Edinburgh and IOHK, Edinburgh, Scotland
papers@tkerber.org, {akiayias,mkohlwei}@ed.ac.uk

Abstract. Non-interactive zero-knowledge proofs, and more specifically succinct non-interactive zero-knowledge arguments (zk-SNARKs), have been proven to be the "Swiss army knife" of the blockchain and distributed ledger space, with a variety of applications in privacy, interoperability and scalability. Many commonly used SNARK systems rely on a *structured reference string*, the secure generation of which turns out to be their Achilles heel: If the randomness used for the generation is known, the soundness of the proof system can be broken with devastating consequences for the underlying blockchain system that utilises them. In this work we describe and analyse, for the first time, a blockchain mechanism that produces a secure SRS with the characteristic that security is shown under comparable conditions to the blockchain protocol itself. Our mechanism makes use of the recent discovery of *updateable* structured reference strings to perform this secure generation in a fully distributed manner. In this way, the SRS emanates from the normal operation of the blockchain protocol itself without the need of additional security assumptions or off-chain computation and/or verification. We provide concrete guidelines for the parameterisation of this setup which allows for the completion of a secure setup in a reasonable period of time. We also provide an incentive scheme that, when paired with the update mechanism, properly incentivises participants into contributing to secure reference string generation.

1 Introduction

In the domain of distributed ledgers, non-interactive zero-knowledge proofs have many interesting applications. In particular, they have been successfully used to introduce privacy into these inherently public peer-to-peer systems. Most notably, Zerocash [2] demonstrates their usefulness in the creation of private currencies. Beyond this, there are numerous suggestions [21,25,29] to apply the same technology to smart contracts for increased privacy. Beyond privacy, other applications of zero knowledge include blockchain interoperability, e.g., [17], and scalability, e.g., [9].

For the practical efficiency of these designs, two things are paramount: The succinctness of proofs, and the speed of verifying these proofs. The distributed

N. Borisov and C. Diaz (Eds.): FC 2021, LNCS 12674, pp. 497–514, 2021.
https://doi.org/10.1007/978-3-662-64322-8_24

nature of the ledgers mandates that a large number of users store and verify each proof made, rendering many zero-knowledge proof systems not fit for purpose.

Research into so-called zk-SNARKs [18–20,26,27] aims at optimising exactly these features, with proof sizes typically under a kilobyte, and verification times in the milliseconds. It is a well-known fact that non-interactive zero-knowledge requires some shared randomness, or a *common reference string*. For many succinct systems [18–20,26,27], a stronger property is necessary: Not only is a shared random value needed, but it must adhere to a specific *structure*. Such structured reference strings (or SRS) typically consist of related group elements: g^{x^i} for all $i \in \mathbb{Z}_n$, for instance.

The obvious way of sampling such a reference string from public randomness reveals the exponents used – and knowledge of these values breaks the soundness of the proof system itself. To make matters worse, the security of these systems typically relies (among others) on *knowledge of exponent* assumptions, which state that to create group elements related in such a way *requires* knowing the underlying exponents and hence any SRS sampler will have to "know" the exponents used and be trusted to erase them, becoming effectively a single point of failure for the underlying system. While secure multi-party computation can be, and has been, used to reduce the trust placed on such a setup process [31], the selection of the participants for the secure computation and the verification of the generation of the SRS by the MPC protocol retain an element of centralisation. Using an MPC setup remains a controversial element in the setup of a decentralised system that requires SNARKs.

Recent work has found succinct zero-knowledge proof systems with *updateable* reference strings [19,26]. In these systems, given a reference string, it is possible to produce an updated reference string, such that knowing the trapdoor of the new string requires both knowing the trapdoor of the old string, *and* knowing the randomness used in the update. [19] conjectured that a blockchain protocol may be used to securely generate such a reference string. Nevertheless, the exact blockchain mechanism that produces the SRS and the description of the security guarantees it can offer has, so far, remained elusive.

1.1 Our Contributions

In this work we describe and analyse, for the first time, a blockchain mechanism that produces a secure SRS with the characteristic that security is shown for similar conditions under which the blockchain protocol is proven to be secure. Notably different, we make implicit use of secure erasure, and require honest majority only during a specific initialisation period. The SRS then emanates from the normal operation of the blockchain protocol itself without the need of additional security assumptions or off-chain computation and/or verification.

We rely primarily on the *chain quality* property of "Nakamoto-style" ledgers [14] – distributed ledgers in which a randomised process selects which user may append a block to an already established chain. Such ledgers rely on an honest majority of hashing power (or some other resource) – and can be shown to

guarantee a chain quality property which suggests that any sufficiently long chain segment will have some blocks created by an honest user, cf. [14,15,28].

Our construction, described in Sect. 3 integrates reference string updates into the block creation process, but we face additional difficulties due to update calculation being a computationally heavy operation (albeit, contrary to brute-force hashing, useful). The issues arising from this are two fold. Firstly, an adversarial party can take shortcuts by supplying a low amount of entropy in their updates, and try to utilise this additional mining power to subvert the reference string which potentially has a large benefit for the adversary. Secondly, even non-colluding rational block creators may be incentivised to use bad randomness which would reduce or remove any security benefits of the updates. Our work addresses both of these issues.

We prove formally that our mechanism produces a secure reference string in the full version of this paper [22, Appendix F] by providing an analysis in the universal composition framework [10]. Furthermore, the full version of this paper [22, Section 4] demonstrates via experimental analysis how to concretely parameterise a proof-of-work ledger to ensure that an adversary which takes shortcuts (while honest users do not) will still fail in subverting the reference string. The concrete results provided in our experimental section can be used to inform the selection of parameters in order to run our reference string generation mechanism in live blockchain systems.

We further introduce an incentive scheme in Sect. 4, which ensures that rational participants in the protocol, who intend to maximise their profits, will avoid low-entropy attacks. In short, the incentive mechanism mandates that a random fraction of update contributors in the final chain will be asked to reveal their trapdoor, which will be verified to be the output of a random oracle by the underlying ledger rules. Only if a user can demonstrate that their update is indeed random do they receive a suitably determined reward for their effort. Careful choice of the reward assignment enables us to demonstrate that rational participants will utilise high entropy exponents, thus contributing to the SRS computation.

1.2 Related Work

Beyond the obvious relation to the works introducing updateable reference strings in [19,26] (most notably Sonic [26], which we follow closely in our instantiation in the full version of this paper [22, Appendix A]), there have been attempts of practically answering the question of how to securely generate reference strings. These have been in a setting where the string is *not* updateable.

Notably [5] describes the mechanism used by Sprout, the first version of Zcash, during the initial setup of the cryptocurrency's SRS. It uses multi-party computation to generate a reference string, with a root of trust on the initial group of people participating. Due to performance constraints on the MPC protocol, the set of parties participating is relatively small, although only the honesty of a single participating party is required.

For the Sapling version of Zcash, a different approach was used when their reference string was replaced (due to an upgrade of the zero-knowledge statement, and proof system used). Their second CRS generation mechanism, described in [6] uses a multiple-phase round-robin mechanism to generate a reference string for Groth's zk-SNARK [18]. They utilise a random beacon to ensure the uniform distribution of the result, and a coordinator to perform deterministic auxiliary computations.

A great deal of work has also gone into the design of non-interactive zero-knowledge which does not require structure in it's references, such as DARK [8], STARKs [1], and Bulletproofs [7]. While these pose a promising alternative which does not require the techniques used in this work, leveraging updatability of reference strings may permit greater efficiency without additional security assumptions, and may be useful in instantiating generic constructions, such as the polynomial commitments-based Halo Infinite [3].

2 Updateable Structured Reference Strings

While updateable structured reference strings (uSRSs) are modelled in the works we are building on [26, Section 3.2], we model their security in the setting of universal composability (UC) [10]. Here, a uSRS is a reference string with an underlying trapdoor τ, which has had a structure function S imposed on it. $S(\tau)$ is the reference string itself, while τ is not revealed to the adversary. In the full version of this paper [22, Appendix A], we prove that Sonic [26] (with small modifications for extraction, as described in Subsect. 2.2), satisfies all the properties we require in this section. Our main proof is independent of the Sonic protocol however, and applies to any updateable reference string scheme satisfying the properties laid out in the rest of this section.

2.1 Standard Requirements

A uSRS scheme $\mathcal{S}$ consists of a trapdoor domain T, an initial trapdoor τ_0, a set P of permissible (and invertible) permutations over T (i.e. bijective functions whose domain and codomain is T), and a structure function S with the domain T. We require P to include the identity function id, and to be closed under function composition: $\forall p_1, p_2 \in P : p_1 \circ p_2 \in P$. An efficient permutation lifting $\dagger$ should exist, such that for any permutation $p \in P$ and $\tau \in T$, $p^\dagger(S(\tau)) = S(p(\tau))$. Finally, there must exist algorithms $\rho \leftarrow \text{ProveUpd}(S(\tau), p)$ and $b \leftarrow \text{VerifyUpd}(S(\tau), \rho, S(p(\tau)))$ for creating and verifying update proofs respectively. The format of these update proofs is not specified, however the following constraints must be met:

1. **Correctness.** Applying an honestly generated update proof will verify: $\forall p \in P, \tau \in T : \text{VerifyUpd}(S(\tau), \text{ProveUpd}(S(\tau), p), S(p(\tau)))$.
2. **Structure preservation.** Applying *any* valid update is equivalent to applying *some* permutation $p \in P$ on the trapdoor: $\forall \rho, \tau, \text{srs}' : \text{VerifyUpd}(S(\tau), \rho, \text{srs}') \implies \exists p \in P : \text{srs}' = S(p(\tau))$.

3. **Update uniformity.** Applying a random permutation is equivalent to selecting a new random trapdoor: Let D be the uniform distribution over T, and for all $\tau \in T$, let D_τ be the uniform distribution over the multiset $\{\ p(\tau) \mid p \in P\ \}$. Then $\forall \tau \in T : D = D_\tau$.

We define a corresponding UC functionality $\mathcal{F}_{\text{uSRS}}$, which provides a reference string $S(p(\tau_{\mathcal{H}}))$, which the adversary can influence by providing the permutation $p \in P$, given only $S(\tau_{\mathcal{H}})$ as input, for a randomly sampled $\tau_{\mathcal{H}} \in T$.

Functionality $\mathcal{F}_{\text{uSRS}}$

The updateable structured reference string functionality $\mathcal{F}_{\text{uSRS}}$ allows the adversary to update a reference string by applying a permutation from a set of permissible permutations P.

The functionality is parameterised by a trapdoor domain T, a structure function S, and a set of permissible permutations P over T.

State variables and initialisation values.

Variable	Description
$\tau_{\mathcal{H}} := \bot$	The honest part of the trapdoor
$\tau := \bot$	The trapdoor

When receiving a message HONEST-SRS *from* $\mathcal{A}$*:*

if $\tau_{\mathcal{H}} = \bot$ **then let** $\tau_{\mathcal{H}} \xleftarrow{R} T$
return $S(\tau_{\mathcal{H}})$

When receiving a message SRS *from a party* ϕ*:*

query $\mathcal{A}$ **with** (PERMUTE, ϕ) and **receive the reply** p
if $\tau = \bot$ **then**
  assert $p \in P \wedge \tau_{\mathcal{H}} \neq \bot$
  let $\tau \leftarrow p(\tau_{\mathcal{H}})$
return $S(\tau)$

We believe this functionality to be of independent interest, and it is not explicitly tied to our implementation. Notably, while we use a distributed ledger as a weak form of a broadcast channel, other broadcasts can be considered without modification to this functionality. While, as presented, the functionality does not dictate any specific usage, we conjecture that when parameterised with an appropriate structure function and permutation set it can be used to securely instantiate updateable SRS-based SNARKs, such as Sonic [26], Marlin [11], or Plonk [13]. Due to the UC setting, this would require additional lifting to enable UC knowledge extraction, such as that of C∅C∅ [24].

2.2 Simulation Requirements

In addition to the basic properties of correctness, structure preservation, and update uniformity, any simulator wishing to help realise $\mathcal{F}_{\text{uSRS}}$ via updates will need to have access to two additional properties:

1. **Update proof simulation.** From an initial SRS $S(\tau)$ for which the simulator knows the trapdoor, it can produce a valid update to any (correctly structured) SRS. Formally: $\exists \mathcal{S}_\rho \forall \tau_1, \tau_2 \in T$: VerifyUpd$(S(\tau_1), \mathcal{S}_\rho(\tau_1, S(\tau_2)), S(\tau_2))$, where $\mathcal{S}_\rho$ is a PPT algorithm.
2. **Permutation extraction.** The simulator must be capable of extracting the permutation p underlying any valid adversarial update proof.

The most natural method to achieve permutation extraction would be using white-box extractors, as the updates themselves typically rely on some form of knowledge assumption, such as knowledge-of-exponent. However, white-box extractors cannot be used in UC proofs. Instead, we will assume that the update proof is proven to correspond to a specific trapdoor through a lower-level NIZK. Crucially, this lower-level NIZK should not require a *structured* reference string, and rely only on a common random string, or a random oracle. Fortunately, it is not subject to stringent efficiency requirements as the full version of this paper [22, Section 4] demonstrates.

Specifically, we assume that the basic update proof ρ is a statement in a NIZK relation $\mathcal{R}$ where the witness is an encoding of the corresponding permutation p. We require each update proof to have one and only one corresponding permutation, formally expressed by requiring $\mathcal{R}$ to be a bijection. This results in a straightforward modification to the ProveUpd and VerifyUpd algorithms that permits the extraction of the underlying permutations even in the UC setting: ProveUpd also creates a NIZK proof π of (ρ, p), and returns (ρ, π), While VerifyUpd returns true only if this newly embedded NIZK proof also verifies.

The addition of this NIZK trivially preserves all security properties including correctness, due to the definition of $\mathcal{R}$:

Definition 1. *A uSRS scheme is permutation extractable if the relation*

$$\mathcal{R} := \{(\text{ProveUpd}(S(\tau), p), p) \mid \tau \in T, p \in P\}$$

is a bijection, and in NP.

We show in [22, Appendix A] that the relation required for the case of Sonic [26] can be efficiently constructed, and leave the question of how to achieve extraction without the reliance on a further NIZK to future work.

3 Building uSRS from Chain Quality

This section shows how to securely initialise a uSRS using a distributed ledger by requiring block creators to perform updates on an evolving uSRS during an initial setup period. After waiting for agreement on the final uSRS, it can be safely used. To formally model this approach, we discuss the ideal and real worlds used in our simulation proof. Both worlds have access to a ledger, however the ideal world's ledger is independent of the reference string (which is instead provided by the independent $\mathcal{F}_{\text{uSRS}}$ functionality), while the real world's ledger is programmed to generate it using updates.

3.1 High-Level Overview

This basic premise of this paper relies on Nakamoto-style ledgers' basic means of operation: Different users can extend a chain of blocks if they can satisfy some condition, with this condition being associated with a type of hardness which ensures attackers are limited in the number of extensions they can perform. Given such a structure, we associate a uSRS update with each block prior to a time δ_1. This time is selected such that the security properties of the ledger ensure at least one of the blocks is honest in each competitive chain at this point.

In our modelling, we construct this from a ledger functionality with an additional *leadership state*, which is derived from information miners embed in their blocks. Specifically for our case, these encode uSRS updates. We leave this sufficiently general to allow other uses as well. The basic idea is to show that a ledger which performs uSRS updates in its leadership state is equivalent to one which doesn't, but is accompanied by the $\mathcal{F}_{\text{uSRS}}$ functionality. They make up our real and ideal worlds respectively. After time δ_1, users wait a further time period δ_2 until common prefix ensures that all parties agree on the reference string.

While ledger functionalities are often treated as global, our approach effectively constructs one ledger from another – the ledger is not a dependency of our protocol, but a component. In this context, globality is irrelevant, as the environment already has direct access to the functionality. We expect protocols building on the ledger to use it in a global fashion, however. The same is not true for the uSRS – most usages will likely rely on the simulator being able to extract its trapdoor.

3.2 Our Ledger Abstraction

Our construction of the updateable structured reference string functionality relies heavily on the properties of *common prefix*, *chain quality*, and *chain growth* defined in the "Bitcoin backbone" analysis by Garay et al. [14], for Nakamoto-style consensus algorithms. Despite our use in the section title, we make use of all three properties, not just that of chain quality. We emphasise chain quality, as it is the property central to ensuring an honest update has occurred. We briefly and informally restate the three properties:

- **Common prefix.** Given the current chains Π_1 and Π_2 of two parties, and removing k blocks from the first, it is a prefix of the second: $\Pi_1^{\lceil k} \prec \Pi_2$.
- **Chain quality.** For any party's current chain Π, any consecutive l blocks in this chain will include μ blocks created by an honest party.
- **Chain growth.** If a party's chain is of length c, then s time slots later, it will be at least of length $c + \gamma$.

These parameters determine the length of the two phases of our protocol. In the first phase, we construct the reference string itself from the liveness parameter (assuming $\mu \geq 1$), and in the second phase, we wait until this reference string has propagated to all users. The length of the first phase is at least $\delta_1 \geq \lceil l\gamma^{-1} \rceil s$,

and that of the second at least $\delta_2 \geq \lceil k\gamma^{-1} \rceil s$. Combined, they make up the total uSRS generation delay $\delta \geq (\lceil l\gamma^{-1} \rceil + \lceil k\gamma^{-1} \rceil)s$.

We assume a ledger which guarantees the backbone properties. While we do not prove any specific existing proof-of-work ledger (or those based on a different leader-selection mechanism) formally UC-realise this specific formalisation, we argue all ledgers with "Nakamoto-style" (as opposed to BFT-style) consensus do so.. Both ledger and argument are presented in the full version of this paper [22, Appendix B]. Our functionality further depends on a *global clock* $\mathcal{G}_{\text{clock}}$, defined in [22, Appendix E.1]. For the purposes of this paper, it is sufficient that this is a beacon providing monotonically increasing values representing the current time to any party requesting them.

In addition to this, we assume each block created can contain additional information, provided by its creator (the "miner"), which can be aggregated to construct a "leader state". Each created block is associated with an *update* a, and the ledger is parameterised by two procedures, Gen, and Apply, which describe the honest selection of updates, and the semantics of updates respectively. Looking forward, these utilise ProveUpd and VerifyUpd internally, although the formalism is sufficiently general to allow usage of the leader state for other, parallel purposes. The exact parameters differ in our ideal and real world, with the ideal world "hiding" the uSRS updates. Additionally, the real world adds time-sensitivity: It does nothing to the SRS after the setup period. Gen is randomised, takes a leader state σ and the current time t as inputs, and produces an update a. Apply takes a leader state σ, an update a, and an update time t, and returns a successor state σ': $\sigma' = \text{Apply}(\sigma, (a, t))$. For a chain, the leader state may be computed by sequentially applying all updates in the chain, starting from an initial state $\varnothing$.

The adversary controls when and which party creates a new block, as well as the transactions each new block contains (provided it does not violate the backbone properties). For transactions created by a corrupted party, the adversary can further control the block's timestamp (within the reasonable limits of not being in the future, and being after the previous block), and the desired update a itself. For honest parties updates, Gen is used instead.
The UC interfaces our ledger provides are:

- SUBMIT. Submitting new transactions for the ledger.
- READ. Reading the confirmed sequence of transactions.
- PROJECTION. Reading the current chain's sequence of (potentially unconfirmed) transactions.
- LEADER-STATE. Reading the confirmed leader state.
- ADVANCE. The adversary switches a party to a longer chain.
- EXTEND. The adversary instructs a party to create a block.

While this ledger abstraction is not the focus of this paper, we believe it to be of independent interest in cases where finer control over miner's actions, or better access to the competing chains is desired.

3.3 The Ideal World

Our ideal world consists of two functionalities, composed in parallel (by which we mean: the environment may address either, and they do not interact). The first is a variant of $\mathcal{F}_{\text{uSRS}}$, with the modification that it cannot be addressed by honest parties before δ time slots have passed. Formally, this modification is made with a wrapper functionality $\mathcal{W}_{\text{delay}}(\mathcal{F}, \delta)$, described in [22, Appendix E.4].

The second is the Nakamoto-style ledger functionality, parameterised with arbitrary leader-state generation and application procedures which are also partially used in the hybrid world: Gen = GenIdeal and Apply = ApplyIdeal, and the following ledger parameters:

1. A common prefix parameter k.
2. Chain quality parameters μ and l.
3. Chain growth parameters γ and s.

Formally then, our ideal world consists of the pair $(\mathcal{W}_{\text{delay}}(\delta, \mathcal{F}_{\text{uSRS}}), \mathcal{F}_{\text{nakLedger}}^{\text{ideal}})$, as well as the global functionality $\mathcal{G}_{\text{clock}}$.

3.4 The Hybrid World

In our hybrid world, we use a uSRS scheme $\mathcal{S}$, with algorithms ProveUpd, VerifyUpd, the structure function S, permissible permutations P, permutation lifting $\dagger$, initial trapdoor τ_0. The hybrid world consists of a separate Nakamoto-style ledger $\mathcal{F}_{\text{nakLedger}}^{\text{real}}$, a NIZK functionality $\mathcal{F}_{\text{NIZK}}^{\mathcal{R}}$, and the global clock $\mathcal{G}_{\text{clock}}$. The ledger is then parameterised by the same chain parameters as those in the ideal world, and the following leader-state procedures:

procedure $\text{Apply}((\text{srs}, \sigma^{\text{ideal}}), ((\text{srs}', \rho, \pi, a^{\text{ideal}}), t))$
  if $\text{srs} = \varnothing$ **then let** $\text{srs} \leftarrow S(\tau_0)$
  if $t \leq \delta_1 \wedge \text{VerifyUpd}(\text{srs}, \rho, \text{srs}')$ **then**
    send $(\text{VERIFY}, \rho, \pi)$ **to** $\mathcal{F}_{\text{NIZK}}^{\mathcal{R}}$ and **receive the reply** b
    if b **then**
      let $\text{srs} \leftarrow \text{srs}'$
  return $(\text{srs}, \text{ApplyIdeal}(\sigma^{\text{ideal}}, a^{\text{ideal}}, t))$
procedure $\text{Gen}((\text{srs}, \sigma^{\text{ideal}}), t)$
  if $t > \delta_1$ **then**
    return $(\epsilon, \epsilon, \epsilon, \text{GenIdeal}(\sigma^{\text{ideal}}, t))$
  else
    let $p \xleftarrow{R} P; \rho \leftarrow \text{ProveUpd}(\text{srs}, p)$
    send (PROVE, ρ, p) **to** $\mathcal{F}_{\text{NIZK}}^{\mathcal{R}}$ and **receive the reply** π
    return $(p^{\dagger}(\text{srs}), \rho, \pi, \text{GenIdeal}(\sigma^{\text{ideal}}, t))$

Note that these parameterising algorithms use $\mathcal{F}_{\text{NIZK}}^{\mathcal{R}}$, and are therefore the reason the ledger depends on this hybrid functionality.

Key here is that once a block is received after the initial chain quality period, any reference string update it may declare is no longer carried out – at this point the uSRS is not necessarily stable, as the chain may still be reorganised, but should not change for this particular chain. Further, these procedures always

mimic the ideal-world behaviour, extending it rather than replacing it. This demonstrates the composability of allowing block leaders to produce updates: One system using updates for security does not impact other parallel uses of the leadership state.

There is little additional work to be done to UC-emulate the ideal-world behaviour, besides ensuring that queries are routed appropriately, especially how the reference string is queried in the hybrid world. We describe this with a small "adaptor" protocol in the full version of this paper [22, Appendix C], LEDGER-ADAPTOR. This forwards most queries, and treats uSRS queries as querying the appropriate part of the leader state after time δ, and by ignoring them before. Formally, our real world consists of the global clock $\mathcal{G}_{\text{clock}}$, and the system LEDGER-ADAPTOR$(\delta, \mathcal{F}_{\text{nakLedger}}^{\text{real}}(\mathcal{F}_{\text{NIZK}}^{\mathcal{R}}))$.

3.5 Alternative Usage of $\mathcal{G}_{\text{clock}}$

In both worlds, $\mathcal{G}_{\text{clock}}$ is used to determine the cutoff point after which the reference string is deemed secure. A simple alternative to this usage of the clock is to instead rely on the length of the chain for this purpose. We did not make this choice as it complicates the ideal world: The delay wrapper would have to communicate with the ideal world ledger, and query it for the length of parties' chains. We do not regard a clock as a significant additional assumption, however little of the remainder of this paper differs if chain lengths are used instead. Even in this case, a clock is present to guarantee liveness, although it is used only to constrain the adversary.

3.6 UC Emulation

Our security is derived through UC-emulation, stated in the following theorem:

Theorem 1. *For any updateable reference string scheme $\mathcal{S}$, satisfying correctness, structure preservation, update uniformity, update simulation with $\mathcal{S}_\rho$, and permutation extraction,* LEDGER-ADAPTOR *(in the $(\mathcal{F}_{\text{nakLedger}}^{\text{real}}, \mathcal{F}_{\text{NIZK}}^{\mathcal{R}})$-hybrid world, parameterised as in Subsect. 3.4) UC-emulates the pair of functionalities $(\mathcal{F}_{\text{nakLedger}}^{\text{ideal}}, \mathcal{W}_{\text{delay}}(\delta, \mathcal{F}_{\text{uSRS}}))$, parameterised as in Subsect. 3.3, in the presence of the global clock functionality $\mathcal{G}_{\text{clock}}$, with the simulator $\mathcal{S}_{\text{LEDGER-ADAPTOR}}$.*

A full security proof and simulator may be found in the full version of this paper [22, Appendix F & D].

4 Low-Entropy Update Mitigation

While our analysis indicates that in a Byzantine, honest majority setting, our protocol produces a trustworthy reference string, it also asks participants to dedicate computational resources to updates. It follows that in a rational setting, players need to be properly incentivised to follow the protocol. We emphasise

that the rational setting is not the focus of this paper, and optimistically, in a setting where the majority of miners are rational and a small fraction honest, the few honest blocks are sufficient to eliminate the issue described in this section.

For Sonic, a protocol deviation exists that breaks the security of the reference string: By choosing the exponent in a specific low-entropy fashion, (e.g., $y = 2^l$) the computation of the update, which primarily relies on repeated squaring, can be done significantly faster. More generally, some permutations in P may be more efficiently computable. In more detail, instead of using a random permutation p, a specific choice is made that eases the computation of srs' – in the most extreme case, for any uSRS scheme, the update for $p = \mathrm{id}$ is trivial.

4.1 Proposed Construction

In order to facilitate a mitigation for this class of attacks, we will need to assume an additional property of the underlying ledger, in particular it must provide a "resettable" randomness beacon: With each ADVANCE operation (where adversary must be restricted in how often it may do such ADVANCE queries), a random beacon value is sampled in a variable bcn and is associated with the corresponding block. Beacons of this kind are often easily available, for instance by hashing the proof-of-work [4], and are inherent in many proof-of-stake designs. Prior work [12] demonstrates that such beacon values allow for the adversary to bias them only by "resetting" it at most a certain number of times, say t, before they are fixed by entering the ledger's confirmed state, with the exact value of t depending on the chain parameters.

We can then amend Gen to derive its random values from the random oracle, by sending the query $(\mathrm{bcn}, \mathrm{nonce})$ to $\mathcal{F}_{\mathrm{RO}}$, where nonce is a randomly selected nonce, and bcn is the previous block's beacon value. The response is used to index the set of trapdoor permutations P, choosing the result p, and the nonce is stored by miners locally, and kept private. We adapt the Phase 1 period δ_1 so that at least $l' := l(1-\theta)^{-1} + c$ blocks will be produced, where θ and c are new security parameters (to be discussed below). Next, after Phase 2 ends, we can be sure that the beacon value associated with the end of Phase 1 has been reset at most t times.

We extract from bcn l' biased coins, each with probability θ. For each block, if the corresponding coin is 1, it is required to reveal its randomness within a period of time at least as long as the liveness parameter. Specifically, a party which created one of the selected blocks may reveal its nonce. If its update matches this nonce, the party receives an additional reward of value R times the standard block reward.

While this requires a stricter chain quality property, with the ledger functionality instead enforcing that one of these l non-opened updates are honest, we sketch why this property still holds in the next section.

4.2 Security Intuition

Consider now a rational miner with hashing power α. We know that, at best, using an underlying blockchain like Bitcoin, the relative rewards such a miner may expect are at most $\alpha/(1-\alpha)$ in expectation; this assumes a selfish mining strategy that wins all network races against the other rational participants. Now consider a miner who uses low entropy exponents to save on computational power on created blocks and, as a result, boosts their hashing power α to an increased relative hashing power of $\alpha' > \alpha$. The attacker can further try to influence the blockchain by forking and selectively disclosing blocks which has the effect of resetting the bcn value to a preferred one. To see that the impact of this is minimal, we prove the following lemma.

Lemma 1. *Consider a mapping $\rho \mapsto \{0,1\}^{l'}$ that generates l' independent biased coin flips, each with probability θ, when ρ is uniformly selected. Consider any fixed $n \leq l'$ positions and suppose an adversary gets to choose any one out of t independent draws of the mapping's random input with the intention to increase the number of successes in the n positions. The probability of obtaining more than $n(1+\epsilon)\theta$ successes is* $\exp(-\Omega(\epsilon^2\theta n) + \ln t)$.

Proof. In case $t = 1$, result follows from a Chernoff bound on the event E defined as obtaining more than $n(1+\epsilon)\theta$ successes, and has probability $\exp(-\Omega(\epsilon^2\theta n))$. Given that each reset is an independent draw of the same experiment, by applying a union bound we obtain the lemma's statement. □

The optimal strategy of a miner utilising low-entropy attacks is to minimise the number of blocks of other miners are chosen, to increase its relative reward. Lemma 1 demonstrates that at most a factor of $(1+\epsilon)^{-1}$ damage can be done in this way. Regardless of whether a miner utilises low-entropy attacks or not, their optimal strategy beyond this is selfish mining, in the low-entropy attack mining in expectation $l'\alpha'/(1-\alpha')$ blocks [14]. A rational miner utilising low-entropy attacks will not gain any additional rewards, while a miner not doing so will gain at least $l'\alpha/(1-\alpha)(1+\epsilon)^{-1}\theta R$ rewards from revealing their randomness, by Lemma 1. It follows that for a rational miner, this strategy can be advantageous to plain selfish mining only in case:

$$\frac{\alpha'}{1-\alpha'} > (1+\theta(1+\epsilon)^{-1}R)\frac{\alpha}{1-\alpha}$$

If we assume a miner can increase their effective hash rate by a factor of c, using low-entropy exponents, then their advantage in the low entropy case is $\alpha' = \alpha c/(\alpha c + \beta)$, where $\beta = 1-\alpha$ is the relative mining power of all other miners. If follows that the miner benefits if and only if:

$$\begin{aligned} &\tfrac{\alpha c}{\alpha c+\beta} \cdot \tfrac{\alpha c+\beta}{\beta} > (1+\theta(1+\epsilon)^{-1}R)\tfrac{\alpha}{\beta} \\ \iff\ & c > 1+\theta(1+\epsilon)^{-1}R \end{aligned}$$

If we adopt a sufficiently large intended time interval between blocks it is possible to bound the relative savings of a selfish miner using low-entropy exponents;

following the parameterisation of the full version's simulation [22, Section 4.2], if a selfish miner using such exponents can improve their hashing power by at most a multiplicative factor c then we can mitigate such attack by setting R to $(c-1)/(\theta(1+\epsilon)^{-1})$.

5 Discussion

While the clean generation of a new reference string from a ledger protocol is itself useful, real-world situations are likely to be more complex. In this section we discuss practical adjustments that may be made.

5.1 Upgrading Reference Strings

As distributed ledgers are typically long-lived, and may well outlive any reference string used within it – or have been running before a reference string was needed. Indeed, the Zcash protocol has seen upgrades in its reference string. A reference string being replaced with a new one is innocuous without further context, however it is important to consider how they are usually used in zero-knowledge proofs. If the proof they are used in is stateless, upgrading from an insecure to a secure reference string behaves as one may naively expect: It ensures that after the upgrade, security properties hold.

In the example of Zcash, which runs a variant of the Zerocash [2] protocol, the situation is more muddy. Zerocash makes *stateful* zero-knowledge proofs. Suppose a user is sceptical of the security of the initial setup – and there is good reason to be [30] – but is convinced the second reference string is secure. Is such a user able to use Zcash with confidence in its security?

Had Zcash not had safeguards in place, the answer would be no. While the protocol may operate as intended currently, and the user can be convinced of that, due to the stateful nature of the proofs, the user cannot be convinced of the correctness of this state. The Zcash cryptocurrency did employ similar safeguards to those we outline below. We stress the importance of such here, as not every project may have the same foresight.

Specifically, for a Zerocash-based system, an original reference string's backdoor could have been used to create mismatched transactions, and to effectively "mint" large coins illicitly. This process is undetectable at the time, and the minted coins would persist across a reference string upgrade. Our fictitious user may therefore be rightfully suspicious as to the value of any coins he is sold – they may be a part of an almost infinite pool!

Such an attack, once carried out (especially against a currency) is hard to recover from – it is impossible to identify "legitimate" owners of the currency, even if the private transaction history were deanonymised, and the culprit identified. The culprit may have traded whatever he created already. Simply invalidating the transaction would therefore harm those he traded with, not himself. In an extreme case, if he traded one-to-one with legitimate owners of the currency, he would succeed in effectively stealing the honest users funds. If such an attack

is identified, the community has two unfortunate options: Annul the funds of potentially legitimate users, or accept a potentially large amount of inflation.

We may assume a less grim scenario however: Suppose we are *reasonably confident* in the security of our old reference string, but we are *more confident* of the new one. Is it possible to convince users that we have genuinely upgraded our security? We suggest the usage of a type of *firewalling* property. Such properties are common in the domain of cross-chain transfers [17], and are designed to prevent a catastrophic failure on one chain damaging another.

For monetary transfers, the firewall would guarantee an upper-bound of funds was not exceeded. Proving the firewall property is preserved is easy if a small loss of privacy is accepted – each private coin being re-minted before it can be used after the upgrade, during which time its value must be declared. Assuming everything operates fine, and the firewall property is not violated, users interacting with the post-firewall state can be confident as to the upper bound of funds available. Further, attacks on the system can be identified: If an attacker mints too many coins, eventually the firewall property will be violated, indicating that too many coins were in circulation – bringing the question of how to handle this situation with it. We believe that a firewall property does however give peace of mind to users of the system, and is a practical means to assuage concerns about the security of a system which once had a questionable reference string.

In Zcash, a soft form of such firewalling is available, in that funds are split across several "pools", each of which uses a different proving mechanism. The total value of each pool can be observed, and values under zero would be considered a cause for alarm, and rejected. Zcash use the terminology "turnstiles" [32], and no attacks have been observed through them.

A further consideration for live systems is that as the full version's simulation [22, Section 4.2] shows, the time required strongly depends on the frequency between blocks. This may conflict with other considerations for selecting the block time – a potential solution for this is to only perform updates on "superblocks": blocks which meet a higher proof-of-work (or other selection mechanism) criteria than usual.

5.2 The Root of Trust

An important question for all protocols in the distributed ledger setting is whether a user entering the system at some point during its runtime can be convinced to trust in its security. Early proof-of-stake protocols, such as [23], did poorly at this, and were subject to "stake-bleeding" attacks [16] for instance – effectively meaning new users could not safely join the network.

For reference strings, if a newly joining user is prepared to accept that the honest majority assumption holds, they may trust the security of the reference string, as per Theorem 1. There is a curious difference to the security of the consensus protocol however: to trust the consensus – at least for proof-of-work based protocols – it is most important to trust a *current* honest majority, as these protocols are assumed to be able to recover from dishonest majorities at some point in their past. The security of the reference string on the other hand

only relies on assuming honest majority during the initial δ time units. This may become an issue if a large period of time passes – why should someone trust the intentions of users during a different age?

In practice, it may make sense to "refresh" a reference string regularly to renew faith in it. It is tempting to instead continuously perform updates, however as noted in Subsect. 5.1, this does not necessarily increase faith in a stateful system, although is can remove the "historical" part from the honest majority requirement when used with stateless proofs.

Most subversion attacks are detectable – they require lengthy forks which are unlikely to occur during a legitimate execution. In an optimistic case, where no attack is attempted, this may provide an additional level of confirmation: if there are no widespread claims of large forks during the initial setup, then the reference string is likely secure (barring large-scale out-of-band censorship). A flip side to this is that it may be a lot easier to sow doubt, however, as there is no way to *prove* this: A malicious actor could create a fork long after the initial setup, and claim that it is evidence of an attack to undermine the credibility of the system.

5.3 Applications to Non-updateable SNARKs

Updateable SNARK schemes have two distinct advantages which our protocol makes use of: First, they have an explicit update procedure which allows a party ϕ to replace a reference string whose security depends on some assumption A, with one whose security depends on $A \vee (\phi$ is honest). Second, they can survive with a partially biased reference string, a fact which we don't use directly in this paper, however the functionality $\mathcal{F}_{\text{uSRS}}$ we provide permits rejection sampling, encoding it into the ideal world.

The lack of an update algorithm can be resolved for some zk-SNARKs, such as [18], by the existence of a weaker property: In two phases, the reference string can be constructed with (potentially different) parties performing round-robin updates (also group exponentiations) in each phase. This approach is also detailed in [6], and it implies a natural translation to our protocol, in which the first phase is replaced with two phases of the same length, performing the first and second phase updates respectively.

The security of partially biased references strings has not been sufficiently analysed for non-updateable SNARKs, however this weakness can be mitigated. Following [6], it is possible to use a pure random beacon (as opposed to the reset-table one used in Sect. 4) to create a "pure" reference string from the "impure" one presented so far. To sketch the design: The random beacon would be queried after time δ, and the randomness used to select a trapdoor permutation over the reference string. This would then be applied by each party independently, arriving at the same – randomly distributed – reference string.

As this is not required for updateable SRS schemes, we did not perform this analysis in depth. However the approach to the simulation would be to perform the SRS generation identically, and then program the random beacon to invert all permutations applied to the honest reference string. Since this includes the

one honest permutation applied on every honest update, this is indistinguishable from a random value to the adversary. It is worth noting that the requirement of a random beacon is on the stronger side of requirements, especially as it should itself not allow adversarial influence to provide the desired advantage. Approaches using block hashes for randomness introduce exactly the limited influence which we are attempting to remove!

Acknowledgements. The second and third author were partially supported by the EU Horizon 2020 project PRIVILEDGE #780477. We thank Eduardo Morais for providing data on the required depth of reference strings for Zcash's Sapling protocol.

References

1. Ben-Sasson, E., Bentov, I., Horesh, Y., Riabzev, M.: Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Report 2018/046 (2018). https://eprint.iacr.org/2018/046
2. Ben-Sasson, E., et al.: Zerocash: decentralized anonymous payments from bitcoin. In: 2014 IEEE Symposium on Security and Privacy, pp. 459–474. IEEE Computer Society Press, May 2014
3. Boneh, D., Drake, J., Fisch, B., Gabizon, A.: Halo infinite: recursive zk-SNARKS from any additive polynomial commitment scheme. Cryptology ePrint Archive, Report 2020/1536 (2020). https://eprint.iacr.org/2020/1536
4. Bonneau, J., Clark, J., Goldfeder, S.: On bitcoin as a public randomness source. Cryptology ePrint Archive, Report 2015/1015 (2015). http://eprint.iacr.org/2015/1015
5. Bowe, S., Gabizon, A., Green, M.D.: A multi-party protocol for constructing the public parameters of the Pinocchio zk-SNARK. In: FC 2018. LNCS, vol. 10958, pp. 64–77. Springer, Heidelberg (2019). https://doi.org/10.1007/978-3-662-58820-8_5
6. Bowe, S., Gabizon, A., Miers, I.: Scalable multi-party computation for zk-SNARK parameters in the random beacon model. Cryptology ePrint Archive, Report 2017/1050 (2017). http://eprint.iacr.org/2017/1050
7. Bünz, B., Bootle, J., Boneh, D., Poelstra, A., Wuille, P., Maxwell, G.: Bulletproofs: short proofs for confidential transactions and more. In: 2018 IEEE Symposium on Security and Privacy, pp. 315–334. IEEE Computer Society Press, May 2018
8. Bünz, B., Fisch, B., Szepieniec, A.: Transparent SNARKs from DARK compilers. In: Canteaut, A., Ishai, Y. (eds.) EUROCRYPT 2020. LNCS, vol. 12105, pp. 677–706. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45721-1_24
9. Buterin, V.: On-chain scaling to potentially 500 tx/sec through mass tx validation. https://ethresear.ch/t/on-chain-scaling-to-potentially-500-tx-sec-through-mass-tx-validation/3477
10. Canetti, R.: Universally composable security: a new paradigm for cryptographic protocols. In: 42nd FOCS, pp. 136–145. IEEE Computer Society Press, October 2001
11. Chiesa, A., Hu, Y., Maller, M., Mishra, P., Vesely, N., Ward, N.: Marlin: preprocessing zkSNARKs with universal and updatable SRS. In: Canteaut, A., Ishai, Y. (eds.) EUROCRYPT 2020. LNCS, vol. 12105, pp. 738–768. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45721-1_26

12. David, B., Gaži, P., Kiayias, A., Russell, A.: Ouroboros Praos: an adaptively-secure, semi-synchronous proof-of-stake blockchain. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018. LNCS, vol. 10821, pp. 66–98. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78375-8_3
13. Gabizon, A., Williamson, Z.J., Ciobotaru, O.: PlonK: permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953 (2019). https://eprint.iacr.org/2019/953
14. Garay, J., Kiayias, A., Leonardos, N.: The bitcoin backbone protocol: analysis and applications. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015. LNCS, vol. 9057, pp. 281–310. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46803-6_10
15. Garay, J., Kiayias, A., Leonardos, N.: The bitcoin backbone protocol with chains of variable difficulty. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017. LNCS, vol. 10401, pp. 291–323. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63688-7_10
16. Gaži, P., Kiayias, A., Russell, A.: Stake-bleeding attacks on proof-of-stake blockchains. Cryptology ePrint Archive, Report 2018/248 (2018). https://eprint.iacr.org/2018/248
17. Gazi, P., Kiayias, A., Zindros, D.: Proof-of-stake sidechains. In: 2019 IEEE Symposium on Security and Privacy, pp. 139–156. IEEE Computer Society Press, May 2019
18. Groth, J.: On the size of pairing-based non-interactive arguments. In: Fischlin, M., Coron, J.-S. (eds.) EUROCRYPT 2016. LNCS, vol. 9666, pp. 305–326. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49896-5_11
19. Groth, J., Kohlweiss, M., Maller, M., Meiklejohn, S., Miers, I.: Updatable and universal common reference strings with applications to zk-SNARKs. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018. LNCS, vol. 10993, pp. 698–728. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96878-0_24
20. Groth, J., Maller, M.: Snarky signatures: minimal signatures of knowledge from simulation-extractable SNARKs. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017. LNCS, vol. 10402, pp. 581–612. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63715-0_20
21. Juels, A., Kosba, A.E., Shi, E.: The ring of gyges: investigating the future of criminal smart contracts. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) ACM CCS 2016, pp. 283–295. ACM Press, October 2016
22. Kerber, T., Kiayias, A., Kohlweiss, M.: Mining for privacy: how to bootstrap a snarky blockchain. Cryptology ePrint Archive, Report 2020/401 (2020). https://eprint.iacr.org/2020/401
23. Kiayias, A., Russell, A., David, B., Oliynykov, R.: Ouroboros: a provably secure proof-of-stake blockchain protocol. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017. LNCS, vol. 10401, pp. 357–388. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63688-7_12
24. Kosba, A., et al.: C∅c∅: a framework for building composable zero-knowledge proofs. Cryptology ePrint Archive, Report 2015/1093 (2015). https://eprint.iacr.org/2015/1093
25. Kosba, A.E., Miller, A., Shi, E., Wen, Z., Papamanthou, C.: Hawk: the blockchain model of cryptography and privacy-preserving smart contracts. In: 2016 IEEE Symposium on Security and Privacy, pp. 839–858. IEEE Computer Society Press, May 2016

26. Maller, M., Bowe, S., Kohlweiss, M., Meiklejohn, S.: Sonic: zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM CCS 2019, pp. 2111–2128. ACM Press, November 2019
27. Parno, B., Howell, J., Gentry, C., Raykova, M.: Pinocchio: nearly practical verifiable computation. In: 2013 IEEE Symposium on Security and Privacy, pp. 238–252. IEEE Computer Society Press, May 2013
28. Pass, R., Seeman, L., Shelat, A.: Analysis of the blockchain protocol in asynchronous networks. In: Coron, J.-S., Nielsen, J.B. (eds.) EUROCRYPT 2017. LNCS, vol. 10211, pp. 643–673. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-56614-6_22
29. Steffen, S., Bichsel, B., Gersbach, M., Melchior, N., Tsankov, P., Vechev, M.T.: zkay: specifying and enforcing data privacy in smart contracts. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM CCS 2019, pp. 1759–1776. ACM Press, November 2019
30. Swihart, J., Winston, B., Bowe, S.: Zcash counterfeiting vulnerability successfully remediated. ECC Blog, February 2019. https://electriccoin.co/blog/zcash-counterfeiting-vulnerability-successfully-remediated/
31. Zcash. Parameter generation (2018). https://z.cash/technology/paramgen/
32. Zcash. Address and value pools in Zcash (2019). https://zcash.readthedocs.io/en/latest/rtd_pages/addresses.html#turnstiles