Benchmarking the Setup of Updatable zk-SNARKs

Karim Baghery, Axel Mertens, and Mahdi Sedaghat

COSIC, KU Leuven, Leuven, Belgium
karim.baghery@kuleuven.be, axel.mertens@kuleuven.be,
ssedagha@esat.kuleuven.be

Abstract. Subversion-resistant zk-SNARKs allow the provers to verify the Structured Reference String (SRS), via an SRS Verification (SV) algorithm and bypass the need for a Trusted Third Party (TTP). Pairing-based zk-SNARKs with *updatable* and *universal* SRS are an extension of subversion-resistant ones which additionally allow the verifiers to update the SRS, via an SRS Updating (SU) algorithm, and similarly bypass the need for a TTP. In this paper, we examine the setup of these zk-SNARKs by benchmarking the efficiency of the SV and SU algorithms within the Arkworks library. The benchmarking covers a range of updatable zk-SNARKs, including Sonic, Plonk, Marlin, Lunar, and Basilisk. Our analysis reveals that relying solely on the standard Algebraic Group Model (AGM) may not be sufficient in practice, and we may need a model with weaker assumptions. Specifically, we find that while Marlin is secure in the AGM, additional elements need to be added to its SRS to formally prove certain security properties in the updatable CRS model. We demonstrate that the SV algorithms become inefficient for mid-sized circuits with over 20,000 multiplication gates and 100 updates. To address this, we introduce Batched SV algorithms (BSV) that leverage standard batching techniques and offer significantly improved performance. As a tool, we propose an efficient verification approach that allows the parties to identify a malicious SRS updater with logarithmic verification in the number of updates. In the case of Basilisk, for a circuit with 2^{20} multiplication gates, a 1000-time updated SRS can be verified in less than 30 sec, a malicious updater can be identified in less than 4 min (improvable by pre-computation), and each update takes less than 6 min.

1 Introduction

Let $\mathbf{R}$ be an NP relation which defines the language $\mathbf{L}$ of all statements, x, for which there exists a witness, w, s.t. $(\mathsf{x}, \mathsf{w}) \in \mathbf{R}$. A Non-Interactive Zero-Knowledge (NIZK) argument [24,11] for $\mathbf{R}$ allows an untrusted prover P, knowing w, to non-interactively convince a sceptical verifier V about the truth of a statement x, without leaking extra information about the witness w. Due to a wide range of applications, there has been a growing interest in recent years to develop NIZK proof systems, particularly those allowing for *succinct* proofs and efficient verifications, so-called zk-SNARKs (zero-knowledge Succinct Non-interactive Arguments of Knowledge) [34,25].

A zk-SNARK is expected to satisfy Zero-Knowledge (ZK) and Knowledge Soundness (KS). ZK ensures that V learns nothing beyond the truth of statement, x, from the proof. KS ensures that no malicious P can convince honest V of a false statement, unless he knows the witness. To achieve ZK and KS at the same time, zk-SNARKs rely on a Structured Reference String (SRS), which is supposed to be sampled by a Trusted Third Party (TTP), using the SRS generation algorithm SG [11]. Therefore, in the SRS model a zk-SNARK consists of three algorithms $(\mathsf{SG}, \mathsf{P}, \mathsf{V})$. In practice, finding a mutually TTP for executing the SG algorithm to generate the SRS can be challenging.

Mitigating the Trust on the Setup of zk-SNARKs. To relax the imposed trust on the setup of zk-SNARK, a line of research distributes the SG algorithm and constructed Multi-Party Computation (MPC) protocols to sample the SRS [10,12,29]. In such protocols, both P and V need to trust only 1 out of $i > 1$ participants.

In a different research direction, in 2016, Bellare et al. [8] built the first NIZK argument that can achieve ZK, even if its SRS was subverted, so-called Subversion ZK (Sub-ZK). In a Sub-ZK NIZK argument, the prover does not need to trust the SRS generator, instead, it needs to run an algorithm, so-called SRS Verification (SV), and verify the validity of SRS before using it. The SV algorithm uses some pairing equations to verify the well-formedness of SRS elements. Two subsequent works of [2,18] presented subversion-resistant zk-SNARKs that similarly come with an SV algorithm and can achieve Sub-ZK. In a Sub-ZK SNARK, consisting of four algorithms $(\mathsf{SG}, \mathsf{SV}, \mathsf{P}, \mathsf{V})$, the provers can verify the validity of SRS, by one-time executing the SV algorithm, and then bypass the need for a TTP. On the other side, the verifiers either need a TTP to generate the SRS, or they need to run an MPC protocol (e.g. [10,12]) to sample the SRS elements, which will relax the level of trust to 1 out of i (participants).

As an extension to the MPC approach and subversion-resistant zk-SNARKs, in 2018, Groth et al. [26] proposed a new model, so-called updatable SRS model, which allows the verifiers to also bypass the trust on a TTP. To this end, a V needs to update the SRS one time, using an SRS Updating (SU) algorithm, and also verify the validity of previous updates and the final SRS, using the SV algorithm. Roughly speaking, in a zk-SNARK with updatable SRS, which consists of five algorithms $(\mathsf{SG}, \mathsf{SU}, \mathsf{SV}, \mathsf{P}, \mathsf{V})$, to bypass the trust on a third party, a P needs to run the SV algorithm, and a V needs to run both SU and SV. In this model, the SRS is universal and can be used for various circuits within a bounded size. Then, Groth et al. [26] built the first zk-SNARK with universal and updatable SRS, but comes with $O(n^2)$ SRS size, where n is the number of multiplication gates in the circuit. In practice, this results in a huge SRS size, and impractical SU and SV algorithms.

Recently, there has been an impressive progress on designing Random Oracle-based zk-SNARKs with linear-size updatable SRS, shorter proofs, and more efficient provers and verifiers. Some of the known schemes that consecutively improve the initial scheme of [26] and the subsequent works are called, Sonic [32], Plonk [21], Marlin [14], Lunar [13], Basilisk [35], and Counting Vampires [31].

Table 2 in App. A.1, compares their efficiency in terms of computational costs of $(\mathsf{SG}, \mathsf{P}, \mathsf{V})$ and the SRS size. Currently, Counting Vampires [31] has the shortest proofs, i.e., two group elements less than Basilisk, but its SRS is 17× larger than the SRS of Basilisk, and this can result in a considerably slower setup phase. The SU and SV algorithms are two essential algorithms for achieving Sub-ZK and Updatable Knowledge Soundness (Upd-KS, KS in the updatable SRS model) and the employment of updatable zk-SNARKs. In order to achieve Sub-ZK and Upd-KS in the updatable SRS model, the underlying SRS *must be publicly verifiable* and *trapdoor extractable* [8,2,18,26]. Meaning that, the consistency of SRS elements should be publicly verifiable, and one should be able to extract the SRS trapdoors from the setup phase (e.g., by relying on a knowledge assumption). The initial scheme [26], and some follow-up generic constructions [3,7,6] come with SU and SV algorithms, under Bilinear Diffie-Hellman Knowledge of Exponent (BDH-KE) assumption. But their SV algorithm is identical for both P and V, which in case of verifying an i-time updated SRS, it brings $O(i)$ pairing operations as an overload for the P. In [31], authors have proposed an SV algorithm to achieve Sub-ZK in their construction. However, their SV algorithm can only be used by P (to achieve Sub-ZK), and it does not consider the verification of an i-time updated SRS, needed by V.

Our Contributions. The main objective of the current paper is to examine the efficiency of the setup phase in updatable zk-SNARKs, and evaluate their empirical performance, particularly in large-scale applications.

To this end, we first present a pair of $(\mathsf{SU}, \mathsf{SV})$ algorithms for each of the updatable zk-SNARKs including: Sonic [32], Plonk [21], Marlin [14], LunarLite [13] and Basilisk [35]. Similar to the earlier works [8,2,18,26], the proposed algorithms use pairing products and are tailored to each specific updatable zk-SNARK. As all the aforementioned zk-SNARKs can be instantiated in various ways, we focus on the pairing-based version of them with the shortest proof, which is commonly used for comparison in the literature. During the construction of the SU and SV algorithms, we noticed that relying only on the standard Algebraic Group Model (AGM) may not be enough in practice. In some cases, we may require a model with weaker assumptions, such as the AGM with *hashing* [30]. In fact, there might be a case that a zk-SNARK with monomial SRS is proven to achieve ZK and KS in the AGM model, but their SRS needs to be modified to achieve Sub-ZK and U-KS. The reason is that, to achieve Sub-ZK and Upd-KS the SRS needs to be publicly verifiable and trapdoor extractable [2,26]. In the rest, we show that the SRS of Marlin [14] is not *trapdoor extractable* as it is, but it can be made trapdoor extractable under the BDH-KE assumption, by adding a single group element to its SRS.

In the rest, we show that using the presented SU and SV algorithms, Sonic, Plonk, LunarLite and Basilisk also can achieve trapdoor extractability, under a subverted/maliciously updated SRS. Since all of them already are proven that satisfy ZK and KS, this implies that they also satisfy Sub-ZK and Upd-KS. Similar to the earlier works [8,2,18,14], our SV algorithms use pairing product equations to verify the SRS. But, differently our SV algorithms get an additional

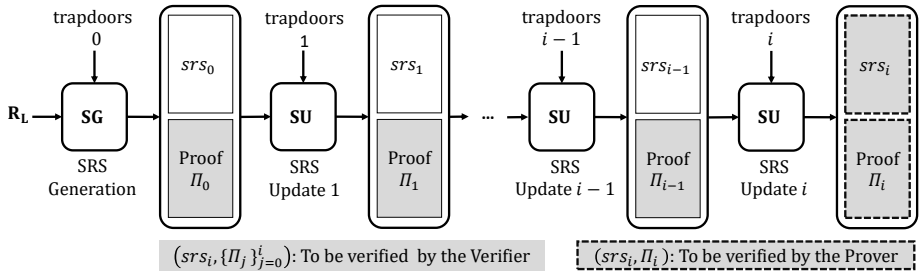

Fig. 1: Setup in the updatable zk-SNARKs: SG, SU, and SV by P or V.

input, denoted by party, which allows us to determine whether a P or V runs the algorithm. Due to achieving Sub-ZK and Upd-KS in the updatable zk-SNARKs, P only needs to verify the final (srs_i, Π_i), while V additionally needs to verify the intermediate proofs $\{\Pi_j\}_{j=0}^{i-1}$. Fig. 1 depicts a graphical representation of the setup phase in the pairing-based updatable zk-SNARKs, and highlights the parts that need to be verified by P or V. By running an SV algorithm, P needs to compute $O(n)$ pairings, where n is the number of multiplication gates in the circuit, and V requires to compute at least $O(n+i)$ pairings, where i is the number of updates done on the SRS. In practice, even for mid-size circuits (e.g. $n \geq 10^4$) with 100 updates, the SV algorithms can be very slow, consequently impractical.

Next, we use the standard batching techniques from [9] and propose a batched version of the SV algorithms, so-called BSV, for each of the studied updatable zk-SNARKs. Using the BSV algorithms, to verify an i-time updated SRS, P needs $O(n)$ exponentiations (with short exponents) and constant number of pairings, which is independent of the number of updates. A V needs to compute $O(n+i)$ exponentiations (with short exponents) and $O(i)$ pairings. Table 1, compares the efficiency of our proposed SU, SV and BSV algorithms for both P and V.

The schemes built in the updatable SRS model [26] can achieve security only with abort, if the parties do not verify the updated SRS after each update. Namely, by verifying the final SRS srs_i and the intermediate proofs $\{\Pi_j\}_{j=0}^{i}$ [26] the parties will abort the final SRS srs_i and would not be able to identify a mali-

Table 1: An efficiency comparison of our proposed SU, SV and BSV algorithms. SV_P: SV run by P, BSV_V: BSV run by V, E_l: Exponentiations in $\mathbb{G}_l$, $\bullet$: Pairing, m: #total (multiplication and addition) gates, n: #multiplication gates, k: #matrix elements with non-zero values describing the circuit, i: # SRS updates

	SG/SU		SV_P	SV_V	BSV_P			BSV_V		
Scheme	E_1	E_2	$\bullet$	$\bullet$	E_1	E_2	$\bullet$	E_1	E_2	$\bullet$
Sonic	$4n$	$4n$	$12n$	$12n+10i$	$8n$	$4n$	7	$8n+8i$	$6n+2i$	$4i+14$
Marlin	k	$\log k$	$2k+12$	$2k+9i+12$	$2k$	$\log k$	4	$2k+5i$	$2i+\log k$	$2i+9$
Plonk	$3m$	2	$6m$	$6m+4i$	$6m$	—	2	$6m+3i$	i	$i+3$
LunarLite	n	n	$3n$	$3n+4i+2$	$2n$	n	3	$2n+3i$	$n+i$	$i+3$
Basilisk	n	2	$2n$	$2n+4i$	$2n$	—	2	$2n+3i$	i	$i+3$

cious SRS generator/updater. To identify a malicious SRS generator/updater, if the parties (or a third party) verify each updated SRS $\{\mathsf{srs}_j\}_{j=0}^{i}$ (instead of only srs_i), then the verification of whole setup phase will be impractical. To deal with that, we introduce an efficient verification approach for identifying the malicious updater. For an i-time updated SRS, it allows the parties to identify the (first) malicious SRS updater with $\log i$ times running the BSV (or SV) algorithm. We discuss different optimizations that can speed up the proposed recursive SRS verification considerably, at the cost of some pre-computations and storage.

Finally, we present a comprehensive benchmark on the efficiency of our proposed SU, SV and BSV algorithms in the $\mathsf{Arkworks}$ library, which is written in Rust and currently is one of the most popular libraries programming zk-SNARKs. Full details of the benchmarking are reported in Sec. 5. In summary, for a particular circuit, by comparing the performance of BSV and SV algorithms, we observed that BSV can achieve up to $110-150\times$ better efficiency. In the case of Basilisk which has the most efficient setup phase, for a circuit with $n = 2^{20}$ multiplication gates, a 1000-time updated SRS can be verified in less than 30 sec. In the case that the verification of final SRS fails, using our proposed recursive verification approach, a malicious SRS updater can be identified in less than 4 min (or in less than 1 min by some pre-computations), and each party equipped with a multi-core CPU can update the SRS in less than 6 min. Our BSV_P algorithms are considerably faster than BSV_V ones, in case of a short SRS (e.g. $n \leq 30K$) and a large number of updates (e.g. $i \geq 200$).

Related Works. To mitigate the trust in the setup phase of zk-SNARKs, there are two key research directions. Either, by using an MPC protocol to sample the SRS [10,12,29], [1] or by directly constructing subversion-resistant [8,2,18,4] and updatable zk-SNARKs [26,32,3,7,6]. Our work is focused on the latter approach.

A bottleneck with the initial MPC protocols [10], is that the number of parties has to be known in advance. Bowe et al. [12] presented an MPC protocol for Groth16 [25] setup, which has two phases. The first phase is known as "Powers of Tau", which can be used to sample a universal SRS for all circuits up to a given size. In the second phase, given the universal SRS generated in the previous phase, parties generate a circuit-dependent SRS. In the Powers of Tau protocol, a coordinator is used to manage messages between the participants, however the output of the protocol is verifiable. Compared with the case one uses the Powers of Tau protocol [12], 1) our proposed algorithms do not need a random beacon, 2) our SV and BSV algorithms are constructed in the updatable SRS model which allows one to verify an i-time updated SRS considerably more efficient than i-time running their SRS verification algorithm. For verifying even one-time updated SRS, our proposed BSV algorithms can be more than $100\times$ faster than their verification algorithm, 3) our SV and BSV algorithms for the provers and verifiers are different, which allows the provers to verify a large-time updated SRS more efficient than verifiers. 4) our protocols can achieve identifiable security more efficiently (using a new recursive SRS verification approach).

In [29], Kohlweiss et al. presented a more efficient version of the Powers of Tau [12]. Their ceremony protocol [29] uses an RO-based proof system,

and comes with a BSV algorithm. Similar to previous $\mathsf{SG}, \mathsf{SU}, \mathsf{SV}$ and BSV algorithms, our algorithms do not use a random beacon or a random oracle. Similar to the earlier works on subversion-resistant or updatable NIZK arguments [2,18,26,4,3,7,6],we rely on particular knowledge assumptions. In comparison with the case that one uses the protocol proposed in [29], 1) our proposed algorithms (i.e., $\mathsf{SG}, \mathsf{SU}, \mathsf{SV}$, and BSV) do not rely on RO, 2) we have different SV (and BSV) algorithms for the provers and verifiers, which allow the provers to verify an updated SRS more efficient than the verifiers, 3) our constructions can achieve identifiable security.

In another related research direction, some studies have defined subversion-resistant and updatable commitments [5,16,22], and have proposed SV and SU algorithms for their studied (knowledge, vector, and polynomial) commitment schemes. Our proposed SV algorithm for Sonic can be considered as an extension of the one proposed in [5], which checks some extra terms and also allows the verifiers to verify an i-time updated SRS. Our SV algorithm for the verifiers in Basilisk is similar to the one proposed in [22], but our SV algorithm for the provers is more efficient. We also propose a batched version of SV algorithms that make them considerably more efficient in practice.

Organization. Sec. 2 introduces notations and preliminaries. In Sec. 3, we present SU and SV algorithms for each of the studied zk-SNARKs, and then prove that they all can achieve Sub-ZK and Upd-KS. We present more efficient and batched variants of the proposed SV algorithms, in Sec. 4. In Sec. 5, we benchmark the performance of our proposed algorithms, and also study identifiable security in the updatable SRS model. Finally, we conclude the paper in Sec. 6.

2 Preliminaries

Throughout, we suppose the security parameter of the scheme and its unary representation to be denoted by λ and 1^λ, respectively. For all positive functions $\varepsilon(\lambda)$, a mapping function $\mathsf{negl} : \mathbb{N} \rightarrow \mathbb{R}^+$ is called negligible function if there exists $\lambda_0 \in \mathbb{N}$ such that for all $\lambda > \lambda_0$ we have, $\mathsf{negl}(\lambda) < 1/\varepsilon(\lambda)$. We use $x \leftarrow\!\!\$\ X$ to denote x sampled uniformly according to the distribution X.

Let PPT and NUPPT denote probabilistic polynomial-time and non-uniform probabilistic polynomial-time, respectively. For an algorithm $\mathcal{A}$, let $\mathrm{im}(\mathcal{A})$ be the image of $\mathcal{A}$, i.e., the set of valid outputs of $\mathcal{A}$. Moreover, assume $\mathsf{RND}(\mathcal{A})$ denotes the random tape of $\mathcal{A}$, and $r \leftarrow\!\!\$\ \mathsf{RND}(\mathcal{A})$ denotes sampling of a randomizer r of sufficient length for $\mathcal{A}$'s needs. By $y \leftarrow \mathcal{A}(x; r)$ we mean given an input x and a randomizer r, $\mathcal{A}$ outputs y. For algorithms $\mathcal{A}$ and $\mathsf{Ext}_{\mathcal{A}}$, we write $(y \,\|\, y') \leftarrow (\mathcal{A} \,\|\, \mathsf{Ext}_{\mathcal{A}})(x; r)$ as a shorthand for "$y \leftarrow \mathcal{A}(x; r)$, $y' \leftarrow \mathsf{Ext}_{\mathcal{A}}(x; r)$".

We use additive and the bracket notation, i.e., in group $\mathbb{G}_\zeta$, $[a]_\zeta = a\,[1]_\zeta$, where $[1]_\zeta$ is the generator of $\mathbb{G}_\zeta$ for $\zeta \in \{1, 2, T\}$. A *bilinear group generator* $\mathsf{BGgen}(1^\lambda)$ returns $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, \hat{e}, [1]_1, [1]_2)$, where p (a large prime) is the order of cyclic abelian groups $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_T$. Finally, $\hat{e} : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ is an efficient non-degenerate bilinear pairing, s.t. $\hat{e}([a]_1, [b]_2) = [ab]_T$. Denote $[a]_1 \bullet [b]_2 = \hat{e}([a]_1, [b]_2)$.

2.1 Updatable, Universal and Subversion-Resistant zk-SNARKs

We adopt the definition of subversion-resistant and updatable zk-SNARKs from [2,26]. Let $\mathcal{R}$ be a relation generator, such that $\mathcal{R}(1^\lambda)$ returns a polynomial-time decidable binary relation $\mathbf{R} = \{(\mathsf{x}, \mathsf{w})\}$, where x is the statement and w is the witness. We assume one can deduce λ from the description of $\mathbf{R}$. Let $\mathbf{L} = \{\mathsf{x} : \exists \mathsf{w} \mid (\mathsf{x}, \mathsf{w}) \in \mathbf{R}\}$ be an **NP**-language including all the statements which there exist corresponding witnesses in relation $\mathbf{R}$. A NIZK argument Ψ_{NIZK} in the updatable SRS model for $\mathcal{R}$ consists of the following PPT algorithms:

- $(\mathsf{srs}_0, \Pi_0) \leftarrow \mathsf{SG}(\mathbf{R})$: Given $\mathbf{R}$, the SRS generator SG first deduces the upper bound N on the relation size. Next, sample the trapdoor ts and then use it to generate srs_0 along with Π_0 as a proof of its well-formedness. Finally, return (srs_0, Π_0) as the output.
- $(\mathsf{srs}_i, \Pi_i) \leftarrow \mathsf{SU}(\mathsf{srs}_{i-1}, \{\Pi_j\}_{j=0}^{i-1})$: Given $(\mathsf{srs}_{i-1}, \{\Pi_j\}_{j=0}^{i-1})$, an SRS updater SU returns the pair of (srs_i, Π_i), where srs_i is the updated SRS and Π_i is a proof for correct updating.
- $(\perp/1) \leftarrow \mathsf{SV}(\mathsf{srs}_i, \{\Pi_j\}_{j=0}^{i}, \mathsf{party})$: Given a potentially updated srs_i, $\{\Pi_j\}_{j=0}^{i}$, SV, and $\mathsf{party} \in \{\mathsf{P}, \mathsf{V}\}$, return either $\perp$ (if srs_i is incorrectly formed or updated) or 1 (if srs_i is correctly formed or updated).
- $(\pi/\perp) \leftarrow \mathsf{P}(\mathbf{R}, \mathsf{srs}_i, \mathsf{x}, \mathsf{w})$: Given the tuple of $(\mathbf{R}, \mathsf{srs}_i, \mathsf{x}, \mathsf{w})$, such that $(\mathsf{x}, \mathsf{w}) \in \mathbf{R}$, P output an argument π. Otherwise, it returns $\perp$.
- $(0/1) \leftarrow \mathsf{V}(\mathbf{R}, \mathsf{srs}_i, \mathsf{x}, \pi)$: Given $(\mathbf{R}, \mathsf{srs}_i, \mathsf{x}, \pi)$, V verify the proof π and return either 0 (reject) or 1 (accept).

In the standard SRS model, a zk-SNARK for $\mathcal{R}$ has a tuple of algorithms $(\mathsf{SG}, \mathsf{P}, \mathsf{V})$ (and SG does not return the Π_0), while subversion-resistant constructions [8,2] additionally have an SV algorithm which is used to verify the well-formedness of the SRS elements to achieve Sub-ZK [8]. But as listed above, in the *updatable* SRS model, a NIZK argument additionally has an SU algorithm that allows the parties (more precisely, the verifiers) to update the SRS and add their own private shares to the SRS generation. Note that in the latest case, the algorithm SG does not necessarily need $\mathbf{R}$, and it only deduces security parameter 1^λ and the upper bound N from it. We highlight that, in comparison with previous definitions [26], our SV algorithm gets an additional input $\mathsf{party} \in \{\mathsf{P}, \mathsf{V}\}$. We later show that this allows us to build a more efficient SV algorithm for the prover. It is worth mentioning that in the updatable SRS model, there also exists a publicly computable deterministic algorithm Derive which given $(\mathbf{R}, \mathsf{srs}_i)$ outputs a specialized SRS for relation $\mathbf{R}$. The output elements of Derive all are in the span of the universal SRS, but they allow to build more efficient proof generation and verification algorithms.

In the subversion-resistant and updatable SRS model, a zk-SNARK is expected to satisfy *updatable completeness*, *Subversion-Zero-Knowledge* (Sub-ZK) and *Updatable Knowledge Soundness* (Upd-KS), of which the definitions are summarized below. In the definition of Sub-ZK, one requires the existence of a PPT simulator Sim consisting of algorithms $(\mathsf{Sim}_{\mathsf{SG}}, \mathsf{Sim}_{\mathsf{P}})$ that share state with each other. The idea is that it can be used to simulate the SRS and proofs without knowing the corresponding trapdoors.

The algorithm of proof simulation. $\pi \leftarrow \mathsf{Sim_P}(\mathbf{R}, \mathsf{srs}_i, \mathsf{ts}_i, \mathsf{x})$: For $\mathsf{SV}(\mathsf{srs}_i, \Pi_i) = 1$, given the tuple $(\mathbf{R}, \mathsf{srs}_i, \mathsf{ts}_i, \mathsf{x})$, where ts_i is the simulation trapdoor associated with the latest SRS, namely srs_i, outputs a simulated argument π.

Definition 1 (Perfect Updatable Completeness). *A non-interactive argument Ψ_{NIZK} is* perfectly updatable complete for $\mathcal{R}$*, if for all* $(\mathbf{R}) \in \mathrm{im}(\mathcal{R}(1^\lambda))$*, and* $(\mathsf{x}, \mathsf{w}) \in \mathbf{R}$*, the following probability is 1 on security parameter* λ*,*

$$\Pr\left[\begin{array}{l}(\mathbf{R}) \leftarrow \mathcal{R}(1^\lambda), (\mathsf{srs}_0, \Pi_0) \leftarrow \mathsf{SG}(\mathbf{R}), (\{\mathsf{srs}_j, \Pi_j\}_{j=1}^i) \leftarrow \mathcal{A}(\mathbf{R}, \mathsf{srs}_0), \\ \{\mathsf{SV}(\mathsf{srs}_j, \Pi_j, \mathsf{party}) = 1\}_{j=0}^i : (\mathsf{x}, \pi) \leftarrow \mathsf{P}(\mathbf{R}, \mathsf{srs}_i, \mathsf{x}, \mathsf{w}) \wedge \mathsf{V}(\mathbf{R}, \mathsf{srs}_i, \mathsf{x}, \pi) = 1\end{array}\right],$$

where Π_i is a proof for the correctness of the initial SRS generation or SRS updating. Note that in the above definition and all the following one, i is the index of final update, and without loss of generality, $\mathcal{A}$ can also first generate $\{\mathsf{srs}_j\}_{j=0}^{i-1}$ and then an honest updater updates srs_{i-1} to srs_i.

Definition 2 (Sub-ZK). *A NI argument Ψ is* computationally Sub-ZK for $\mathcal{R}$*, if for any PPT subvertor* Sub *there exists a PPT extractor* $\mathsf{Ext}_{\mathsf{Sub}}$*, s.t. for all* λ*, all* $\mathbf{R} \in \mathrm{im}(\mathcal{R}(1^\lambda))$*, and for any PPT* $\mathcal{A}$*, one has* $\varepsilon_0 \approx_\lambda \varepsilon_1$*, where*

$$\varepsilon_b = \Pr\left[\begin{array}{l}r \leftarrow\!\$\, \mathsf{RND}(\mathsf{Sub}), ((\mathsf{srs}, \Pi_{\mathsf{srs}}, \xi_{\mathsf{Sub}}) \,\|\, \mathsf{ts}) \leftarrow (\mathsf{Sub} \,\|\, \mathsf{Ext}_{\mathsf{Sub}})(\mathbf{R}; r) : \\ \mathsf{SV}(\mathsf{srs}, \Pi_{\mathsf{srs}}, \mathsf{party}) = 1 \wedge \mathcal{A}^{\mathsf{O}_b(\cdot,\cdot)}(\mathbf{R}, \mathsf{srs}, \mathsf{ts}, \xi_{\mathsf{Sub}}) = 1\end{array}\right].$$

Here, ξ_{Sub} is auxiliary information generated by subvertor Sub*, the* party *is set to be the prover, and the oracle* $\mathsf{O}_0(\mathsf{x}, \mathsf{w})$ *returns* $\perp$ *(reject) if* $(\mathsf{x}, \mathsf{w}) \notin \mathbf{R}$*, and otherwise it returns* $\mathsf{P}(\mathbf{R}, \mathsf{srs}, \mathsf{x}, \mathsf{w})$*. Similarly,* $\mathsf{O}_1(\mathsf{x}, \mathsf{w})$ *returns* $\perp$ *(reject) if* $(\mathsf{x}, \mathsf{w}) \notin \mathbf{R}$*, and otherwise it returns* $\mathsf{Sim}(\mathbf{R}, \mathsf{srs}, \mathsf{ts}, \mathsf{x})$*. Ψ is* perfectly Sub-ZK for $\mathcal{R}$ *if one requires that* $\varepsilon_0 = \varepsilon_1$*.*

Definition 3 (Updatable nBB Knowledge Soundness). *A non-interactive argument Ψ_{NIZK} is* updatable non-black-box knowledge sound for $\mathcal{R}$*, if for every PPT adversary* $\mathcal{A}$ *and any subvertor* Sub*, there exists a PPT extractor* $\mathsf{Ext}_{\mathcal{A}}$*, and the following probability is* $\mathsf{negl}(\lambda)$*,*

$$\Pr\left[\begin{array}{l}\mathbf{R} \leftarrow \mathcal{R}(1^\lambda), (\mathsf{srs}_0, \Pi_0) \leftarrow \mathsf{SG}(\mathbf{R}), r_s \leftarrow\!\$\, \mathsf{RND}(\mathsf{Sub}), \\ (\{\mathsf{srs}_j, \Pi_j\}_{j=1}^i, \xi_{\mathsf{Sub}}) \leftarrow \mathsf{Sub}(\mathsf{srs}_0, \Pi_0, r_s), \{\mathsf{SV}(\mathsf{srs}_j, \Pi_j, \mathsf{party}) = 1\}_{j=0}^i, \\ r_{\mathcal{A}} \leftarrow\!\$\, \mathsf{RND}(\mathcal{A}), ((\mathsf{x}, \pi) \,\|\, \mathsf{w}) \leftarrow (\mathcal{A} \,\|\, \mathsf{Ext}_{\mathcal{A}})(\mathbf{R}, \mathsf{srs}_i, \xi_{\mathsf{Sub}}; r_{\mathcal{A}}) : \\ (\mathsf{x}, \mathsf{w}) \notin \mathbf{R} \wedge \mathsf{V}(\mathbf{R}, \mathsf{srs}_i, \mathsf{x}, \pi) = 1\end{array}\right],$$

Here $\mathsf{RND}(\mathcal{A}) = \mathsf{RND}(\mathsf{Sub})$*, Π_{srs} is a proof for correctness of SRS generation or updating process, and the* party *is set to be the verifier.*

2.2 Assumptions

Definition 4 (Bilinear Diffie-Hellman Knowledge of Exponent (BDH-KE) Assumption [2]). *We say* BGgen *is* BDH-KE secure *for relation set $\mathcal{R}$ if*

for any λ, $\mathbf{R} \in \mathrm{im}(\mathcal{R}(1^\lambda))$, *and PPT adversary* $\mathcal{A}$, *there exists a PPT extractor* $\mathsf{Ext}_{\mathcal{A}}$, *such that, the following probability is* $\mathsf{negl}(\lambda)$,

$$\Pr\left[\begin{array}{l}(p,\mathbb{G}_1,\mathbb{G}_2,\mathbb{G}_T,\hat{e},[1]_1,[1]_2) \leftarrow \mathsf{BGgen}(1^\lambda), r \leftarrow\!\!\$\ \mathsf{RND}(\mathcal{A}),\\ ([\alpha_1]_1,[\alpha_2]_2 \parallel a) \leftarrow (\mathcal{A} \parallel \mathsf{Ext}_{\mathcal{A}})(\mathbf{R},r) : [\alpha_1]_1 \bullet [1]_2 = [1]_1 \bullet [\alpha_2]_2 \wedge a \neq \alpha_1\end{array}\right].$$

The BDH-KE assumption [2] is an asymmetric-pairing version of the original knowledge assumption [15]. We refer to App. A for some preliminaries on polynomial commitments that are used in the rest of paper.

3 SU and SV Algorithms for Updatable zk-SNARKs

In this section, we present a pair of SRS updating and SRS verification algorithms for each of the studied updatable zk-SNARKs, Sonic [32], Plonk [21], Marlin [14], LunarLite [13] and Basilisk [35].

The General Strategy. Our proposed SV and SU algorithms use pairing checks for SRS verification and the SRS elements are updated in a round-robin multiplicative manner. In comparison with the earlier works, we have a subtle change in the construction of SV algorithms, which allows the provers to verify an updated SRS more efficiently, especially in case of small circuits with a large number of updates. Recall that, a pairing-based zk-SNARK satisfies Sub-ZK if it can achieve ZK, even if its SRS is subverted (i.e., is generated by the adversary). In Sub-ZK zk-SNARKs [8,2,18,4], this is formalized and achieved by building an SV algorithm that verifies the *well-formedness* and *trapdoor extractability* of the SRS. The former guarantees that the whole SRS elements are consistent with each other, and the latter ensures that the (simulation) trapdoors of SRS can be extracted from an SRS subverter. Given the simulation trapdoors of SRS, the proofs are simulated as in the standard ZK. On the other side, a universal zk-SNARK is updatable [26] if its SRS can be sequentially updated by the parties, such that Upd-KS holds if at least one of the updates with SU or the initial SRS generation with SG is done honestly. To ensure that SRS generation/updating is done correctly, parties should return a knowledge assumption-based proof Π when running SG or SU algorithms. This proof is also known as the well-formedness proof of the SRS. In the presented SV algorithms, we use the fact that to achieve Sub-ZK, a P only needs to verify the final SRS. Without loss of generality, one can assume that the initial SRS generation and all the follow-up updates are done with a single adversary who can control all the updaters who run SU and the initial party who runs SG. However, to achieve Upd-KS without a TTP, a V needs to one-time run the SU and update the SRS, and also verify the final SRS and the correctness of all intermediate proofs, generated by all the updaters (See Fig. 1).

Next, in each subsection, we present an overview of a particular updatable zk-SNARK, and then describe its SRS Generation (SG) algorithm. Different from the original papers, in the description of SG algorithms, we also determine what constitutes a well-formedness proof that can be used to extract individual shares from the SRS generator/updaters, and more importantly, can be used to verify

the final SRS. The well-formedness proof is shown with Π which consists of two sets of elements $(\Pi^{\mathsf{Agg}}, \Pi^{\mathsf{Ind}})$, where Π^{Agg} can be interpreted as the aggregated elements necessary for verifying the well-formedness of final SRS, and Π^{Ind} can be interpreted as an individual proof for the correctness of updating using the secret shares, e.g. $\bar{x}$. The latter, also enables extracting the individual shares from a malicious SRS generator/updater in the proof of Upd-KS. Finally, we present SU and SV algorithms.

3.1 SU and SV Algorithms for Sonic

Sonic and its SG *algorithm.* The first proposed updatable zk-SNARK, presented by Groth et al. [26], came with explicit SU and SV algorithms, but its SRS size scales quadratically in the number of multiplication gates in the circuit that encodes the relation, which made the algorithms very slow. In a follow-up work, Maller et al. [32] proposed Sonic as the first updatable zk-SNARK with linear size SRS. The authors mostly focused on achieving a linear size SRS and more efficient P and V algorithms, and omitted the descriptions of SU and SV algorithms (and even SG which should determine the well-formedness proof) and mentioned that they can built as in [26]. For further details, we refer to the main paper [32]. We describe the SG algorithm of Sonic in Fig. 2.

SU *and* SV *Algorithms and Their Efficiency.* Fig. 3 describes the SU and SV algorithms for Sonic. As briefly mentioned before, the SRS update is done in a multiplicative manner, such that the updater multiplies a proper power of its secret shares $\bar{x}_i$ and $\bar{a}_i$ to the SRS elements. Similar to the SG algorithm, we also determine the elements of the well-formedness proof separately. Note that $[a]_T$ is omitted from updating, as due to the fact that $[a]_T := [1]_1 \bullet [a]_2$, it can finally be computed from the other SRS elements. The pairing checks inside SV chase two main goals. First, they check if all the individual proofs generated by the SRS generator and by all the follow-up SRS updaters are correct. If so, then it uses the elements of Π_i and verifies the final SRS, srs_i.

Efficiency. As it can be seen in Fig. 2 and 3, given the SU algorithm, similar to the SG algorithm, to update the SRS of size n in Sonic, one needs to compute $4n+2$ exponentiations in $\mathbb{G}_1$ and $4n+2$ exponentiations in $\mathbb{G}_2$. Using the SV

SRS Generation, $(\mathsf{srs}_0, \Pi_0) \leftarrow \mathsf{SG}(\mathbf{R})$**:** Given $\mathbf{R}$, first deduce the security parameter 1^λ and k, then obtain $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, \hat{e}, [1]_1, [1]_2) \leftarrow \mathsf{BGgen}(1^\lambda)$; after that act as follows:

- Sample $\bar{x}_0, \bar{a}_0 \leftarrow \mathbb{Z}_p^\star$, and set $x_0 := \bar{x}_0$ and $a_0 := \bar{a}_0$ which are the simulation trapdoor associated with srs_0;
- For $k = -n, \cdots, n$: compute $\left[x_0^k\right]_1, \left[x_0^k\right]_2, \left[a_0 x_0^k\right]_2$;
- For $k = -n, \cdots, -1, 1, \cdots, n$: compute $\left[a_0 x_0^k\right]_1$; Compute $[a_0]_T$;
- Set $\mathsf{srs}_0 := \left(\left(\left[x_0^k\right]_1, \left[x_0^k\right]_2, \left[a_0 x_0^k\right]_2\right)_{k=-n}^{n}, \left(\left[a_0 x_0^k\right]_1\right)_{k=-n, k\neq 0}^{n}, [a_0]_T\right)$, and the well-formedness proof $\Pi_0 := (\Pi_0^{\mathsf{Agg}}, \Pi_0^{\mathsf{Ind}}) := (([x_0]_1, [a_0 x_0]_1, [a_0]_2), ([x_0]_1, [x_0]_2, [a_0 x_0]_1, [a_0 x_0]_2, [a_0]_2))$;
- Return (srs_0, Π_0);

Fig. 2: SG algorithm for SONIC.

SRS Update, $(\mathsf{srs}_i, \Pi_i) \leftarrow \mathsf{SU}(\mathsf{srs}_{i-1}, \{\Pi_{j-1}\}_{j=0}^{i-1})$**:** Given $(\mathsf{srs}_{i-1}, \{\Pi_{j-1}\}_{j=0}^{i-1})$,

- Parse $\mathsf{srs}_{i-1} := ((\left[x_{i-1}^k\right]_1, \left[x_{i-1}^k\right]_2, \left[a_{i-1}x_{i-1}^k\right]_2)_{k=-n}^{n}, (\left[a_{i-1}x_{i-1}^k\right]_1)_{k=-n,k\neq 0}^{n})$;
- Sample $\bar{x}_i, \bar{a}_i \leftarrow\!\$\ \mathbb{Z}_p^\star$, as the secret shares to be used for updating srs_{i-1}.
- For $k = -n, \cdots, n$: set $\left[x_i^k\right]_1 := \bar{x}_i^k \cdot \left[x_{i-1}^k\right]_1$; set $\left[x_i^k\right]_2 := \bar{x}_i^k \cdot \left[x_{i-1}^k\right]_2$; set $\left[a_i x_i^k\right]_2 := \bar{a}_i \bar{x}_i^k \cdot \left[a_{i-1}x_{i-1}^k\right]_2$;
- For $k = -n, \cdots, -1, 1, \cdots, n$: set $\left[a_i x_i^k\right]_1 := \bar{a}_i \bar{x}_i^k \cdot \left[a_{i-1}x_{i-1}^k\right]_1$;
- Set $\mathsf{srs}_i := ((\left[x_i^k\right]_1, \left[x_i^k\right]_2, \left[a_i x_i^k\right]_2)_{k=-n}^{n}, (\left[a_i x_i^k\right]_1)_{k=-n,k\neq 0}^{n}, [a_i]_T)$, and the well-formedness proof $\Pi_i := (\Pi_i^{\mathsf{Agg}}, \Pi_i^{\mathsf{Ind}}) := \left(\left([x_i]_1, [a_i x_i]_1, [a_i]_2\right), \left([\bar{x}_i]_1, [\bar{x}_i]_2, [\bar{a}_i \bar{x}_i]_1, [\bar{a}_i \bar{x}_i]_2, [\bar{a}_i]_2\right)\right)$;
- Return (srs_i, Π_i);

SRS Verify, $(\bot/1) \leftarrow \mathsf{SV}(\mathsf{srs}_i, (\Pi_j)_{j=0}^{i}, \mathsf{party})$**:** To verify (an i-time updated) $\mathsf{srs}_i := ((\left[x_i^k\right]_1, \left[x_i^k\right]_2, \left[a_i x_i^k\right]_2)_{k=-n}^{n}, (\left[a_i x_i^k\right]_1)_{k=-n,k\neq 0}^{n}, [a_i]_T)$, and $\Pi_j := \left(\Pi_j^{\mathsf{Agg}}, \Pi_j^{\mathsf{Ind}}\right) := \left(\left([x_j]_1, [a_j x_j]_1, [a_j]_2\right), \left([\bar{x}_j]_1, [\bar{x}_j]_2, [\bar{a}_j \bar{x}_j]_1, [\bar{a}_j \bar{x}_j]_2, [\bar{a}_j]_2\right)\right)$ for $j = 0, 1, \cdots, i$:

If $\mathsf{party} = \mathsf{P}$:

1. For $k = -n, \cdots, n$: check if $\left[x_i^k\right]_1 \bullet [1]_2 = [1]_1 \bullet \left[x_i^k\right]_2$;
2. For $k = -n+1, \cdots, n$: check if $\left[x_i^k\right]_1 \bullet [1]_2 = \left[x_i^{k-1}\right]_1 \bullet [x_i]_2$;
3. For $k = -n, \cdots, -1, 1, \cdots n$: check if $\left[a_i x_i^k\right]_1 \bullet [1]_2 = [1]_1 \bullet \left[a_i x_i^k\right]_2 = \left[x_i^k\right]_1 \bullet [a_i]_2$;

If $\mathsf{party} = \mathsf{V}$:

- If $i = 0$: srs_0 is sampled by $\mathsf{verifier}$, and it does not need to be verified.
- If $i \geq 1$:
  1. Check that $[x_0]_1 = [\bar{x}_0]_1$, $[a_0 x_0]_1 = [\bar{a}_0 \bar{x}_0]_1$, and $[a_0]_2 = [\bar{a}_0]_2$;
  2. For $j = 0, 1, \cdots, i$: check if $[\bar{x}_j]_1 \bullet [1]_2 = [1]_1 \bullet [\bar{x}_j]_2$
  3. For $j = 0, 1, \cdots, i$: check if $[\bar{a}_j \bar{x}_j]_1 \bullet [1]_2 = [1]_1 \bullet [\bar{a}_j \bar{x}_j]_2 = [\bar{x}_j]_1 \bullet [\bar{a}_j]_2$;
  4. For $j = 1, 2, \cdots, i$: check if $[x_j]_1 \bullet [1]_2 = [x_{j-1}]_1 \bullet [\bar{x}_j]_2$;
  5. For $j = 1, 2, \cdots, i$: check if $[a_j x_j]_1 \bullet [1]_2 = [x_j]_1 \bullet [a_j]_2 = [a_{j-1} x_{j-1}]_1 \bullet [\bar{a}_j \bar{x}_j]_2$;
  6. For $k = -n, \cdots, n$: check if $\left[x_i^k\right]_1 \bullet [1]_2 = [1]_1 \bullet \left[x_i^k\right]_2$;
  7. For $k = -n+1, \cdots, n$: check if $\left[x_i^k\right]_1 \bullet [1]_2 = \left[x_i^{k-1}\right]_1 \bullet [x_i]_2$;
  8. For $k = -n, \cdots, -1, 1, \cdots n$: check if $\left[a_i x_i^k\right]_1 \bullet [1]_2 = [1]_1 \bullet \left[a_i x_i^k\right]_2 = \left[x_i^k\right]_1 \bullet [a_i]_2$;

Return 1 if all the checks passed, otherwise return $\bot$.

Fig. 3: SU and SV algorithms for SONIC.

algorithm described in Fig. 3, to verify an i-time updated SRS, $i \geq 1$, a prover needs to compute $12n - 1$ pairing operations (importantly, independent of the number of updates), while a verifier needs to compute $12n + 10i + 4$ pairings.

Security Proofs. In [32, Theorem 6.1, 6.2], authors proved that assuming the ability to extract a trapdoor for the subverted/updated SRS (without proving it), Sonic satisfies Sub-ZK and KS. The following lemmas prove that using the SG, SU and SV algorithms (given in Fig.2 and 3), under the BDH-KE assumption, one can extract the simulation trapdoors from a subverted/updated SRS.

Lemma 1 (Trapdoor Extraction from a Subverted SRS). *Given the algorithm in Fig. 2 and 3, suppose that there exists a PPT adversary $\mathcal{A}$ that outputs a (srs_i, Π_i) such that $\mathsf{SV}(\mathsf{srs}_i, \Pi_i, \mathsf{P}) = 1$ with non-negligible probability.*

Then, by the BDH-KE assumption (given in Def. 4) there exists a PPT extractor $\mathsf{Ext}_{\mathcal{A}}$ *given the random tape of* $\mathcal{A}$ *as input, outputs* (x_i, a_i) *such that running* SG *with* (x_i, a_i) *results in* (srs_i, Π_i).

Proof. An SRS, srs_i, and proof, Π_i, that passes verification is structured as if it were computed by $\mathsf{SG}(\mathbf{R})$; i.e., there exist values $(x_i, a_i) \in \mathbb{F}_p^2$ such that Π_i includes $([x_i]_1, [a_i x_i]_1, [a_i]_2)$ and srs_i includes $\Big(\big([x_i^k]_1, [x_i^k]_2, [a_i x_i^k]_2\big)_{k=-n}^{n}, \big([a_i x_i^k]_1\big)_{k=-n,k\neq 0}^{n}\Big)$. Note that, for $k = 1$, one can deduce $([x_i]_1, [a_i x_i]_1, [x_i]_2, [a_i]_2, [a_i x_i]_2)$ from srs_i.

Let $\mathcal{A}$ be an adversary that outputs (srs_i, Π_i). We then define algorithms $\mathcal{A}_{x_i}$ and $\mathcal{A}_{a_i}$, that each run $(\mathsf{srs}_i, \Pi_i) \leftarrow \mathcal{A}(\mathbf{R})$, parse Π as above, and returns $([x_i]_1, [x_i]_2)$ and $([a_i x_i]_1, [a_i]_2)$, respectively. According to the BDH-KE assumption (given in Def. 4) there exist PPT extractors $\mathsf{Ext}_{\mathcal{A}_{x_i}}$ and $\mathsf{Ext}_{\mathcal{A}_{a_i}}$ that, given the randomness of $\mathcal{A}_{x_i}$ and $\mathcal{A}_{a_i}$, output some $x_i, a_i \in \mathbb{F}_p$ that can be used to generate $([x_i]_1, [a_i x_i]_1, [x_i]_2, [a_i]_2, [a_i x_i]_2)$. By combining $\mathsf{Ext}_{\mathcal{A}_{x_i}}$ and $\mathsf{Ext}_{\mathcal{A}_{a_i}}$, we obtain a full extractor for $\mathcal{A}$. From the rest of checks within the SV algorithm one concludes that all the SRS elements are consistent and the SRS is well-formed. □

The following lemma shows that SRS trapdoors can be extracted from an updated SRS. To this end, we first recall a corollary from [23].

Corollary 1. *In the updatable SRS model, single adversarial updates imply full updatable security [23, Lemma 6].*

Lemma 2 (Trapdoor Extraction from an Updated SRS). *Given the algorithm in Fig. 2 and 3, suppose that there exists a PPT* $\mathcal{A}$ *such that given* $(\mathsf{srs}_0, \pi_0) \leftarrow \mathsf{SG}(\mathbf{R})$, $\mathcal{A}$ *returns an updated SRS* (srs_1, π_1), *where* $\mathsf{SV}(\mathsf{srs}_1, \Pi_1, \mathsf{V}) = 1$ *with a non-negligible probability. Then, the BDH-KE assumption implies that there exists a PPT extractor* $\mathsf{Ext}_{\mathcal{A}}$ *that, given the randomness of* $\mathcal{A}$ *as input, outputs* $(\bar{x}_1, \bar{a}_1)$ *that are used to update* srs_0 *and generate* (srs_1, Π_1).

Proof. According to the Corollary 1, we consider the case that $\mathcal{A}$ updates the SRS only once, but similar to [23, Lemma 6], it can be generalized. Parse Π_0 as a tuple $(([x_0]_1, [a_0 x_0]_1, [a_0]_2), ([x_0]_1, [x_0]_2, [a_0 x_0]_1, [a_0 x_0]_2, [a_0]_2))$ and srs_0 as $\big([x_0^k]_1, [x_0^k]_2, [a_0 x_0^k]_2\big)_{k=-n}^{n}$ and $\big[a_0 x_0^k\big]_1$ for $k = -n, \cdots, -1, 1, \cdots, n$.

We consider an adversary $\mathcal{A}$, that given (srs_0, Π_0), returns an updated SRS, srs_1, which contains $\big([x_1^k]_1, [x_1^k]_2, [a_1 x_1^k]_2\big)_{k=-n}^{n}$ and $\big[a_1 x_1^k\big]_1$ for $k = -n, \cdots, -1, 1, \cdots, n$, and a proof π_1 for correct updating as containing $(([x_1]_1, [a_1 x_1]_1, [a_1]_2), ([\bar{x}_1]_1, [\bar{x}_1]_2, [\bar{a}_0 \bar{x}_0]_1, [\bar{a}_0 \bar{x}_0]_2, [\bar{a}_0]_2))$. If the SRS verification accepts the updated SRS, namely if $\mathsf{SV}(\mathsf{srs}_1, \Pi_1, \mathsf{V}) = 1$, then the following equations hold, 1) $[\bar{x}_1]_1 \bullet [1]_2 = [1]_1 \bullet [\bar{x}_1]_2$, 2) $[\bar{a}_1 \bar{x}_1]_1 \bullet [1]_2 = [1]_1 \bullet [\bar{a}_1 \bar{x}_1]_2 = [\bar{x}_1]_1 \bullet [\bar{a}_1]_2$, 3) $[x_1]_1 \bullet [1]_2 = [x_0]_1 \bullet [\bar{x}_1]_2$, 4) $[a_1 x_1]_1 \bullet [1]_2 = [x_1]_1 \bullet [a_1]_2 = [a_0 x_0]_1 \bullet [\bar{a}_1 \bar{x}_1]_2$. So from the equations 1) and 2), under the BDH-KE assumption, there exist extractors $\mathsf{Ext}_{\bar{x}_1}$ and $\mathsf{Ext}_{\bar{a}_1}$ that output $\bar{a}_1$ and $\bar{x}_1$. If $\bar{a}_1$ and $\bar{x}_1$ are non-zero, then from the rest of verification equations within SV algorithm (e.g., $[x_1]_1 \bullet [1]_2 = [x_0]_1 \bullet [\bar{x}_1]_2$), one can conclude that $x_1 = \bar{x}_1 x_0$, $a_1 = \bar{a}_1 a_0$, and the SRS is well-formed. □

3.2 SU and SV Algorithms for Marlin

Marlin. As a follow-up work to Sonic and a concurrent work to Plonk, Chiesa et al. proposed Marlin [14], which is comparable to Plonk in performance and outperforms Sonic. Compared to Sonic, Marlin reduces P's computational cost by a factor of $10\times$ and improves V's time by a factor of $4\times$ without compromising the constant-size property of proofs. To this end, the authors first propose an information-theoretic model called Algebraic Holographic Proof (AHP), which is an interactive protocol between algebraic P and V. The verifier performs a small number of queries on an encoding of the circuit instead of receiving the entire circuit description. At the end, the verifier makes a number of queries to the proofs provided by the prover and then performs low-degree tests to be convinced about the validity of proof and the encoding of the circuit. Then, they proposed a transformation that uses PCs with Fiat-Shamir transformation [17] and compiles any public coin AHP for sparse Rank 1 Constraint System (R1CS) instances into a preprocessing zk-SNARK with universal and updatable SRS. To build Marlin, authors first proposed two PC schemes, which one is proven to be secure under a concrete knowledge assumption, and the other one is built in the Algebraic Group Model (AGM) [14, Appendix B]. The scheme built in the AGM model achieves a better efficiency and requires a single group element to commit to a polynomial (instead of two in the initial construction). Marlin is a zk-SNARK which is obtained by instantiating their transformation by the AGM-based PC scheme. Both their PC schemes are proven to be secure (complete, hiding, extractable, as defined in App. A.2) under a *trusted setup* [14, Lemmas B.5-B.15], and later, the AGM-based one is used to obtain updatable zk-SNARK Marlin.

Achieving Sub-ZK and Upd-KS in Marlin. Marlin uses a universal SRS and assuming that the simulation trapdoors are provided to the ZK simulator, it is proven to achieve ZK and KS in the AGM. In [14, Remark 7.1], authors argue that their constructions have updatable SRS because of using monomial terms in the SRS, and thus fall within the framework of [26]. The SRS of Marlin, which is equivalent to the SRS of their AGM-based PC scheme, consists of $\mathsf{srs} := \left(\left(\left[x^k\right]_1, \left[\gamma x^k\right]_1\right)_{k=0}^{n}, [1]_2, [x]_2\right)$ group elements. This SRS is shown to be sufficient for their PC scheme. Note that a standard PC scheme, is constructed under a trusted setup, and there is no guarantee that it will remain secure under a subverted SRS or a maliciously updated SRS. Therefore, once we use the SRS of a PC scheme (with a trusted setup) to build a Sub-ZK zk-SNARK with updatable SRS, we need to ensure that the SRS of resulting zk-SNARK is well-formed and trapdoor-extractable [23]. Since Marlin is proven to satisfy KS under the above SRS srs, therefore, to prove that it also achieves Upd-KS, we need to show that the SRS trapdoors can be extracted from a subverted or a (maliciously) updated SRS. However, one may notice that in practice an adversary, capable of hashing to an elliptic curve, can produce the SRS $([x]_1, [\gamma x]_1, [1]_2, [x]_2)$ without knowing γ. For instance, it can sample a group element from $\mathbb{G}_1$, without knowing its exponent, and then use a known x to compute $([x]_1, [\gamma x]_1, [1]_2, [x]_2)$ for an unknown γ. A malicious SRS updater can perform a similar attack.

One may argue that Marlin (and some follow-up schemes) is proven in the original AGM [19], which adversaries are purely algebraic and do not have the capability to create random group elements without knowing their discrete logarithms. This argument is valid, but the problem still exists in practice and such constructions may not achieve Sub-ZK be default, as an adversary can use elliptic curve hashing [27] to sample random group elements without knowing the exponents. To deal with such concerns, earlier Sub-ZK SNARKs [2,30] used and are proven in more realistic models, namely the Generic Group Model (GGM) with *hashing* [2] and the AGM with *hashing* [30]. The "with hashing" parts mean that the adversary is allowed to sample random group elements without knowing the exponents, say using the elliptic curve hashing [27]. Considering the discussed issue, one can see that to achieve Sub-ZK/Upd-KS in updatable zk-SANRKs, including Marlin, a more realistic option is to prove them in the more realistic variant of AGM, namely AGM with hashing [30], and also explicitly construct the extraction algorithms requited in the games of Sub-ZK/Upd-KS. It is worth to mention that, by chance, the SRS of Kate et al.'s polynomial commitment scheme [28] is well-formed and without further modification, its SRS can achieve trapdoor extractability under BDH-KE assumption. This is the reason that the updatable zk-SNARKs that directly use Kate et al.'s PC scheme [28], e.g., Lunar or Basilisk, do not face with the mentioned issue. In the rest, we focus on constructing a concrete extraction algorithm which is necessary to prove the Sub-ZK and Upd-KS of Marlin. As we argued above, γ cannot be extracted from the original SRS of Marlin, and we need to slightly modify its SRS to achieve trapdoor extractability and prove Sub-ZK and Upd-KS.

Marlin with a Trapdoor Extractable SRS. To deal with the discussed issue, the solution is to force the adversary to add a proof of knowledge of γ to the SRS, such that the simulator would be able to extract γ from a maliciously generated SRS. In earlier works [8,2,23], this is simply achieved by forcing the SRS generator to return γ in two different groups. Then, relying on the BDH-KE assumption one can extract γ from a maliciously generated SRS. Consequently, we slightly modify the SRS of Marlin and add a single group element $[\gamma x]_2$ to it. Then, we show that in the modified version, the SRS trapdoors can be extracted from a subverted/updated SRS, which would allow to prove Sub-ZK/Upd-KS.

SRS Generation, $(\mathsf{srs}_0, \Pi_0) \leftarrow \mathsf{SG}(\mathbf{R})$**:** Given $\mathbf{R}$, first deduce the security parameter 1^λ and obtain $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, \hat{e}, [1]_1, [1]_2) \leftarrow \mathsf{BGgen}(1^\lambda)$; then act as follows:

- Sample $\bar{x}_0, \bar{\gamma}_0 \leftarrow \mathbb{Z}_p^\star$, and set $x_0 := \bar{x}_0$, and $\gamma_0 := \bar{\gamma}_0$ which are the trapdoors of srs_0;
- For $k = 0, \cdots, n$: compute $\left[x_0^k\right]_1, \left[\gamma_0 x_0^k\right]_1$;
- Compute $[x_0]_2$, and $[x_0\gamma_0]_2$;
- Set $\mathsf{srs}_0 := ((\left[x_0^k\right]_1, \left[\gamma_0 x_0^k\right]_1)_{k=0}^n, [x_0]_2, [x_0\gamma_0]_2)$, and the well-formedness proof $\Pi_0 := (\Pi_0^{\mathsf{Agg}}, \Pi_0^{\mathsf{Ind}}) := (([\gamma_0]_1, [x_0\gamma_0]_1, [x_0]_2), ([\bar{x}_0]_1, [\bar{\gamma}_0]_1, [\bar{x}_0]_2, [\bar{x}_0\bar{\gamma}_0]_2))$;
- Return (srs_0, Π_0);

Fig. 4: Slightly modified SG algorithm of Marlin. The term $[x_0\gamma_0]_2$ is added to SRS and proof to make the SRS well-formed and achieve trapdoor extractability.

SRS Update, $(\mathsf{srs}_i, \Pi_i) \leftarrow \mathsf{SU}(\mathsf{srs}_{i-1}, \{\Pi_{j-1}\}_{j=0}^{i-1})$**:** Given $(\mathsf{srs}_{i-1}, \{\Pi_{j-1}\}_{j=0}^{i-1})$,

- Parse $\mathsf{srs}_{i-1} := ((\left[x_{i-1}^k\right]_1, \left[\gamma_{i-1}x_{i-1}^k\right]_1)_{k=0}^n, [x_{i-1}]_2, [x_{i-1}\gamma_{i-1}]_2)$;
- Sample $\bar{x}_i, \bar{\gamma}_i \leftarrow \mathbb{Z}_p^\star$ as the secret shares to use for updating srs_{i-1}.
- For $k = 0, \cdots, n$: set $\left[x_i^k\right]_1 := \bar{x}_i^k \cdot \left[x_{i-1}^k\right]_1$, $\left[\gamma_i x_i^k\right]_1 := \bar{\gamma}_i \bar{x}_i^k \cdot \left[\gamma_{i-1}x_{i-1}^k\right]_1$;
- set $[x_i]_2 := \bar{x}_i \cdot [x_{i-1}]_2$ and $[x_i\gamma_i]_2 := \bar{x}_i\bar{\gamma}_i \cdot [x_i\gamma_i]_2$;
- Set $\mathsf{srs}_i := ((\left[x_i^k\right]_1, \left[\gamma_i x_i^k\right]_1)_{k=0}^n, [x_i]_2, [x_i\gamma_i]_2)$, and the well-formedness proof $\Pi_i := (\Pi_i^{\mathsf{Agg}}, \Pi_i^{\mathsf{Ind}}) := (([\gamma_i]_1, [x_i\gamma_i]_1, [x_i]_2), ([\bar{x}_i]_1, [\bar{\gamma}_i]_1, [\bar{x}_i]_2, [\bar{x}_i\bar{\gamma}_i]_2))$;
- Return (srs_i, Π_i);

SRS Verify, $(\perp/1) \leftarrow \mathsf{SV}(\mathsf{srs}_i, (\Pi_j)_{j=0}^i, \mathsf{party})$**:** To verify (an i-time updated) $\mathsf{srs}_i := ((\left[x_i^k\right]_1, \left[\gamma_i x_i^k\right]_1)_{k=0}^n, [x_i]_2, [x_i\gamma_i]_2)$, and $\Pi_j := (\Pi_j^{\mathsf{Agg}}, \Pi_j^{\mathsf{Ind}}) := (([\gamma_j]_1, [x_j\gamma_j]_1, [x_j]_2), ([\bar{x}_j]_1, [\bar{\gamma}_j]_1, [\bar{x}_j]_2, [\bar{x}_j\bar{\gamma}_j]_2))$; for $j = 0, 1, \cdots, i$:

If $\mathsf{party} = \mathsf{P}$:

1. For $k = 1, \cdots, n$: check if $\left[x_i^k\right]_1 \bullet [1]_2 = \left[x_i^{k-1}\right]_1 \bullet [x_i]_2$;
2. For $k = 1, \cdots, n$: check if $\left[\gamma_i x_i^k\right]_1 \bullet [1]_2 = \left[\gamma_i x_i^{k-1}\right]_1 \bullet [x_i]_2$;
3. Check if $[x_i\gamma_i]_1 \bullet [1]_2 = [1]_1 \bullet [\gamma_i x_i]_2$;

If $\mathsf{party} = \mathsf{V}$:

- If $i = 0$: srs_0 is sampled by $\mathsf{verifier}$, and it does not need to be verified.
- If $i \geq 1$:
  1. Check if $[\gamma_0]_1 = [\bar{\gamma}_0]_1$ and $[x_0]_2 = [\bar{x}_0]_2$;
  2. For $j = 0, 1, \cdots, i$: check if $[\bar{x}_j]_1 \bullet [1]_2 = [1]_1 \bullet [\bar{x}_j]_2$ and $[1]_1 \bullet [\bar{x}_j\bar{\gamma}_j]_2 = [\bar{\gamma}_j]_1 \bullet [\bar{x}_j]_2$.
  3. For $j = 1, 2, \cdots, i$: check if $[1]_1 \bullet [x_j]_2 = [\bar{x}_j]_1 \bullet [x_{j-1}]_2$, $[x_j\gamma_j]_1 \bullet [1]_2 = [x_{j-1}\gamma_{j-1}]_1 \bullet [\bar{x}_j\bar{\gamma}_j]_2 = [\gamma_j]_1 \bullet [x_j]_2$;
  4. For $k = 1, \cdots, n$: check if $\left[x_i^k\right]_1 \bullet [1]_2 = \left[x_i^{k-1}\right]_1 \bullet [x_i]_2$;
  5. For $k = 1, \cdots, n$: check if $\left[\gamma_i x_i^k\right]_1 \bullet [1]_2 = \left[\gamma_i x_i^{k-1}\right]_1 \bullet [x_i]_2$;
  6. Check if $[x_i\gamma_i]_1 \bullet [1]_2 = [1]_1 \bullet [\gamma_i x_i]_2$;

Return 1 if all the checks passed, otherwise return $\perp$.

Fig. 5: SV and SU algorithms for Marlin with the slightly modified SRS.

We describe the modified SG algorithm of Marlin in Fig. 4, and the new added element is shown with gray background.

SU and SV Algorithms and Their Efficiency. In Fig. 5, we describe our constructed SU and SV algorithms for Marlin with the modified SRS. As the other cases, the SRS update is multiplicative, and at the end, the updater also gives a well-formedness proof which includes the new element $[\bar{x}_i\bar{\gamma}_i]_2$. The new element allows one to verify the well-formedness of the final SRS as well as the validity of intermediate proofs. The SV algorithm verifies if $\{\Pi_j\}_{j=0}^i$ are valid and the final SRS, srs_i, is well-formed.

Using the SU algorithm in Fig. 5, similar to the SG algorithm (in Fig. 4), to update the SRS of size n in Marlin, one needs to compute 2 exponentiations in $\mathbb{G}_2$ and $2n + 1$ exponentiations in $\mathbb{G}_1$. Using the SV algorithm described in Fig. 5, to verify an i-time updated SRS, $i \geq 1$, a prover needs to compute $4n + 2$ pairing operations, while a verifier needs to compute $4n + 2 + 9i + 4$ pairings.

Security Proofs. Relying on the fact that the underlying PC scheme is secure, Marlin, is proven to achieve ZK and KS in the AGM model [14, Theorem 8.1, 8.3 and 8.4]. Our evaluations show that our minimal modification to their PC scheme does not compromise the security of the original scheme (see App. B.1 for further details). Moreover, in the rest, we show that using the presented SG, SU and SV algorithms (given in Fig. 4 and 5), under the BDH-KE assumption (as in [30]), it is also possible to extract the simulation trapdoors from a subverted/updated SRS and achieve Sub-ZK and Upd-KS in the AGM.

Lemma 3 (Trapdoor Extraction from a Subverted SRS). *Given the algorithms in Fig. 4 and 5, suppose that there exists a PPT adversary $\mathcal{A}$ that outputs (srs_i, Π_i) such that $\mathsf{SV}(\mathsf{srs}_i, \Pi_i, \mathsf{P}) = 1$ with a non-negligible probability. Then, by the BDH-KE assumption (given in Def. 4) there exists a PPT extractor $\mathsf{Ext}_{\mathcal{A}}$ that, given the random tape of $\mathcal{A}$ as input, outputs (x_i, γ_i) such that running SG with (x_i, γ_i) results in (srs_i, Π_i).*

Proof. The proof is analogue to the proof of Lemma 1. □

Lemma 4 (Trapdoor Extraction from an Updated SRS). *Given the algorithms in Fig. 4 and 5, suppose that there exists a PPT $\mathcal{A}$ such that given $(\mathsf{srs}_0, \Pi_0) \leftarrow \mathsf{SG}(\mathbf{R})$, $\mathcal{A}$ returns an updated SRS (srs_1, Π_1) s.t. $\mathsf{SV}(\mathsf{srs}_1, \Pi_1, \mathsf{V}) = 1$, with a non-negligible probability. Then, the BDH-KE assumption implies that there exists a PPT extractor $\mathsf{Ext}_{\mathcal{A}}$ that, given the randomness of $\mathcal{A}$ as input, outputs $(\bar{x}_1, \bar{\gamma}_1)$ that are used to update srs_0 and generate (srs_1, Π_1).*

Proof. The proof is analogue to the proof of Lemma 2. □

3.3 SU and SV Algorithms for LunarLite

LunarLite and its SG *algorithm.* In 2021, Campanelli et al. proposed Lunar [13] and compared to Marlin the authors describe several improvements. As we mentioned above, Marlin is built over sparse R1CS instances while Campanelli et al. define a new and simpler version of R1CS, known as R1CS-lite, with only two characterizing matrices instead of three s.t. one of the matrices can be the identity matrix. R1CS-lite with almost the same complexity as R1CS can be utilized to express the language of circuit satisfiability. Meanwhile, the property of two-matrix instances enables the authors to achieve more efficient and simpler zk-SNARKs. In addition, Campanelli et al. demonstrate an efficient method to prove PC soundness with partial opening rather than opening all the commitments. Note that in Marlin and Lunar like Plonk, the prover to commit to vectors utilizes the Lagrange interpolation basis. For further details, we refer to the main paper [13]. In Fig. 6, we describe the SG algorithm of LunarLite, which is the most efficient instantiate of Lunar in term of proof size. Lunar has another variant, so called LunarLite2x, which has slightly shorter SRS, but results in slightly longer proofs.

SRS Generation, $(\mathsf{srs}_0, \Pi_0) \leftarrow \mathsf{SG}(\mathbf{R})$**:** Given $\mathbf{R}$, first deduce the security parameter 1^λ and obtain $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, \hat{e}, [1]_1, [1]_2) \leftarrow \mathsf{BGgen}(1^\lambda)$; then act as follows:
- Sample $x_0 := \bar{x}_0 \leftarrow \mathbb{Z}_p^\star$, which is the trapdoor associated with srs_0;
- For $k = 1, \cdots, n$: compute $\left[x_0^k\right]_1, \left[x_0^k\right]_2$;
- Set $\mathsf{srs}_0 := \left(\left[x_0^k\right]_1, \left[x_0^k\right]_2\right)_{k=0}^n$, and the well-formedness proof $\Pi_0 := (\Pi_0^{\mathsf{Agg}}, \Pi_0^{\mathsf{Ind}}) := \left([x_0]_1, \left([\bar{x}_0]_1, [\bar{x}_0]_2\right)\right)$;
- Return (srs_0, Π_0);

Fig. 6: SG algorithm for LunarLite.

SRS Update, $(\mathsf{srs}_i, \Pi_i) \leftarrow \mathsf{SU}\left(\mathsf{srs}_{i-1}, \{\Pi_{j-1}\}_{j=0}^{i-1}\right)$**:** Given $\left(\mathsf{srs}_{i-1}, \{\Pi_{j-1}\}_{j=0}^{i-1}\right)$,
- Parse $\mathsf{srs}_{i-1} := \left(\left[x_{i-1}^k\right]_1, \left[x_{i-1}^k\right]_2\right)_{k=0}^n$;
- Sample $\bar{x}_i \leftarrow \mathbb{Z}_p^\star$, as the secret share used for updating srs_{i-1}.
- For $k = 1, \cdots, n$: set $\left[x_i^k\right]_1 := \bar{x}_i^k \cdot \left[x_{i-1}^k\right]_1$; and $\left[x_i^k\right]_2 := \bar{x}_i^k \cdot \left[x_{i-1}^k\right]_2$;
- Set $\mathsf{srs}_i := \left(\left[x_i^k\right]_1, \left[x_i^k\right]_2\right)_{k=0}^n$, and $\Pi_i := \left(\Pi_i^{\mathsf{Agg}}, \Pi_i^{\mathsf{Ind}}\right) := \left([x_i]_1, \left([\bar{x}_i]_1, [\bar{x}_i]_2\right)\right)$.
- Return (srs_i, Π_i);

SRS Verify, $(\bot/1) \leftarrow \mathsf{SV}\left(\mathsf{srs}_i, (\Pi_j)_{j=0}^i, \mathsf{party}\right)$**:** To verify (an i-time updated) $\mathsf{srs}_i := \left(\left[x_i^k\right]_1, \left[x_i^k\right]_2\right)_{k=0}^n$, and $\Pi_j := \left(\Pi_j^{\mathsf{Agg}}, \Pi_j^{\mathsf{Ind}}\right) := \left([x_j]_1, \left([\bar{x}_j]_1, [\bar{x}_j]_2\right)\right)$ for $j = 0, 1, \cdots, i$:

If $\mathsf{party} = \mathsf{P}$:

1. For $k = 1, 2, \cdots, n$: check if $\left[x_i^k\right]_1 \bullet [1]_2 = [1]_1 \bullet \left[x_i^k\right]_2 = \left[x_i^{k-1}\right]_1 \bullet [x_i]_2$;

If $\mathsf{party} = \mathsf{V}$:

- If $i = 0$: srs_0 is sampled by verifier, and it does not need to be verified.
- If $i \geq 1$:
  1. Check that $[x_0]_1 = [\bar{x}_0]_1$;
  2. For $j = 0, 1, \cdots, i$: check if $[\bar{x}_j]_1 \bullet [1]_2 = [1]_1 \bullet [\bar{x}_j]_2$;
  3. For $j = 1, 2, \cdots, i$: check if $[x_j]_1 \bullet [1]_2 = [x_{j-1}]_1 \bullet [\bar{x}_j]_2$;
  4. For $k = 1, \cdots, n$: check if $\left[x_i^k\right]_1 \bullet [1]_2 = [1]_1 \bullet \left[x_i^k\right]_2 = \left[x_i^{k-1}\right]_1 \bullet [x_i]_2$;

Return 1 if all the checks passed, otherwise return $\bot$.

Fig. 7: SU and SV algorithms for LunarLite.

SU and SV Algorithms and Their Efficiency. Fig. 7 illustrates our constructed SU and SV algorithms for updatable zk-SNARK LunarLite. In order to update the SRS of size n in LunarLite, one needs to execute the SU algorithm described in Fig. 7 that similar to the SRS generation algorithm, it requires $n+1$ exponentiations in $\mathbb{G}_1$ and $n+1$ exponentiations in $\mathbb{G}_2$. Using the SV algorithm (given in Fig 7), to verify an i-time updated SRS, $i \geq 1$, a prover needs to compute $3n$ pairing operations (importantly, independent of the value of i), while a verifier needs to compute $3n + 4i + 2$ pairings.

Security Proofs. In [13], authors proved that different versions of Lunar, including LunarLite can achieve ZK and KS. However, similar to other constructions, they did not explicitly prove Sub-ZK and Upd-KS. For example, to prove ZK, they assumed that the simulation trapdoor x is provided to the simulator. The same as other constructions, next, we prove that using the SG, SU and SV algorithms (given in Fig.6 and 7), under the BDH-KE assumption, we can extract

the simulation trapdoors for LunarLite from a subverted or updated SRS, which implies that LunarLite meets Sub-ZK and Upd-KS.

Lemma 5 (Trapdoor Extraction from a Subverted SRS). *Given the algorithms in Fig. 6 and 7, suppose that there exists a PPT adversary $\mathcal{A}$ that outputs a (srs_i, Π_i) such that $\mathsf{SV}(\mathsf{srs}_i, \Pi_i, \mathsf{P}) = 1$ with non-negligible probability. Then, by the BDH-KE assumption (given in Def. 4) there exists a PPT extractor $\mathsf{Ext}_{\mathcal{A}}$ that, given the random tape of $\mathcal{A}$ as input, outputs x_i such that running SG with x_i results in (srs_i, Π_i).*

Proof. An srs_i and Π_i that passes verification, namely $\mathsf{SV}(\mathsf{srs}_1, \Pi_1, \mathsf{P}) = 1$, is structured as if it were computed by $\mathsf{SG}(\mathbf{R})$; i.e., there exist values $x_i \in \mathbb{F}_p$ s.t. Π_i includes $[x_i]_1$ and srs_i includes $(\left[x_i^k\right]_1, \left[x_i^k\right]_2)_{k=0}^n$. Therefore, for $k = 1$, one can obtain $([x_i]_1, [x_i]_2)$.

Let $\mathcal{A}$ be an adversary (or subverter) that outputs (srs_i, Π_i). We then define algorithm $\mathcal{A}_{x_i}$, that runs $(\mathsf{srs}_i, \Pi_i) \leftarrow \mathcal{A}(\mathbf{R})$, parse Π_i as above, and returns $([x_i]_1, [x_i]_2)$. Under the BDH-KE assumption (given in Def. 4) there exists a PPT extractor $\mathsf{Ext}_{\mathcal{A}_{x_i}}$ that, given the randomness of $\mathcal{A}_{x_i}$, outputs a $x_i \in \mathbb{F}_p$ that can be used to generate $([x_i]_1, [x_i]_2)$. This gives an extractor for $\mathcal{A}$. From the rest of pairing checks within the SV algorithm, one concludes that all SRS elements are consistent and the SRS is well-formed. □

Lemma 6 (Trapdoor Extraction from an Updated SRS). *Given the algorithms in Fig. 6 and 7, suppose that there exists a PPT adversary $\mathcal{A}$ such that given $(\mathsf{srs}_0, \pi_0) \leftarrow \mathsf{SG}(\mathbf{R})$, $\mathcal{A}$ returns an updated SRS, (srs_1, π_1), where $\mathsf{SV}(\mathsf{srs}_1, \Pi_1, \mathsf{V}) = 1$ with a non-negligible probability. Then, the BDH-KE assumption implies that there exists a PPT extractor $\mathsf{Ext}_{\mathcal{A}}$, given the randomness of $\mathcal{A}$ as input, outputs $\bar{x}_1$ that are used to update srs_0 and generate (srs_1, Π_1).*

Proof. In Corollary 1, we consider the case where $\mathcal{A}$ updates the SRS only once. However, as in [23, Lemma 6], this case is generalizable. Parse Π_0 as containing $([x_0]_1, ([x_0]_1, [x_0]_2))$ and srs_0 includes $(\left[x_0^k\right]_1, \left[x_0^k\right]_2)_{k=0}^n$.

We consider an adversary $\mathcal{A}$, that given (srs_0, Π_0), returns an updated SRS, srs_1, which contains $(\left[x_1^k\right]_1, \left[x_1^k\right]_2)_{k=0}^n$, and a proof $\pi_1 = ([x_1]_1, ([\bar{x}_1]_1, [\bar{x}_1]_2))$ for correct updating. If the SV algorithm accepts the updated SRS, say $\mathsf{SV}(\mathsf{srs}_1, \Pi_1, \mathsf{V}) = 1$, then the following equations hold, 1) $[\bar{x}_1]_1 \bullet [1]_2 = [1]_1 \bullet [\bar{x}_1]_2$, 2) $[x_1]_1 \bullet [1]_2 = [x_0]_1 \bullet [\bar{x}_1]_2$, 3) $[a_1]_1 \bullet [1]_2 = [1]_1 \bullet [x_1]_2$. If the equation 1) holds, under the BDH-KE assumption, there exists an extractor $\mathsf{Ext}_{\bar{x}_1}$ that outputs $\bar{x}_1$. If $\bar{x}_1$ is non-zero, then from the other one concludes that $x_1 = \bar{x}_1 x_0$, and the SRS is well-formed. □

3.4 SU and SV Algorithms for Plonk and Basilisk

Plonk and Basilisk and their SG *algorithms.* As a subsequent work on Sonic [32], in 2019, Gabizon et al. [21] designed Plonk as an updatable and universal zk-SNARK that can be run in two modes: either with a small proof (large SRS) or with a long proof (short SRS). Although Plonk relies neither on bivariate

SRS Generation, $(\mathsf{srs}_0, \Pi_0) \leftarrow \mathsf{SG}(\mathbf{R})$**:** Given $\mathbf{R}$, first deduce the security parameter 1^λ and obtain $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, \hat{e}, [1]_1, [1]_2) \leftarrow \mathsf{BGgen}(1^\lambda)$; then act as follows:
- Sample $x_0 \leftarrow \mathbb{Z}_p$, which is the simulation trapdoor associated with srs_0; chooses an arbitrary $u_0 \in \mathbb{Z}_p^\star$;
- Compute $[x_0]_2$ and $\left[x_0^k\right]_1$ for $k = 1, 2, \cdots, n$;
- Set $\mathsf{srs}_0 := \left(\left(\left[x_0^k\right]_1\right)_{k=1}^n, [x_0]_2, \boxed{u_0}\right)$, and $\Pi_0 := (\Pi_0^{\mathsf{Agg}}, \Pi_0^{\mathsf{Ind}}) := \left([x_0]_1, ([x_0]_1, [x_0]_2)\right)$;
- Return (srs_0, Π_0);

Fig. 8: SG algorithm for Basilisk (and Plonk without the elements u_0).

polynomials nor sparse matrices that leads to a more general type of constraints, its SRS depends both on addition and multiplication gates for any given circuit. While Sonic commits to vectors using standard interpolation basis, Plonk uses Lagrange interpolation basis. As a subsequent work on Sonic, Plonk, Marlin and LunarLite, in 2021, Rafols and Zapico [35] presented Basilisk updatable zk-SNARK. They, first defined a novel information theoretical interactive technique called Checkable Subspace Sampling (CSS) arguments in which P shows that a vector is sampled from a subspace based on V's coin. To be more precise, for a given matrix M, both P and V agree on a polynomial $F(x)$ which encodes a row v within M's rows space. This method is efficient because, in spite of the fact that the coefficients of the linear combination defining v are sampled according to V's coin, there is no need to perform a linear number of operations in order to check that $F(x)$ is well-formed. There are a number of trade-offs associated with universal and updatable zk-SNARK resulting from the CSS proof systems constructed. The most efficient instantiation is called Basilisk that is built for a limited constraint system in which R1CS instances' matrices have a small constant number of elements per row (it is equivalent to arithmetic circuits of bounded fan-out). For further details about Plonk and Basilisk, we refer to their main papers [21,35]. Plonk and Basilisk have almost the same SG algorithms with a similar SRS elements, except that in the case of Basilisk, there exists an extra element u_i in the SRS and the SRS is generated with a smaller upper bound on the size of relation. The reason is that the constraint system used in Plonk, encodes both the addition and multiplication gates that leads to a longer SRS. We investigate and construct the SG, SU and SV algorithms for Basilisk, but with minimal changes they can be adapted and be used for Plonk. We start by describing the SG algorithm of Basilisk in Fig. 8.

SU and SV Algorithms and Their Efficiency. Fig. 9 describes our constructed SU and SV algorithms for updatable zk-SNARK Basilisk. By removing the parts related to the element u_i, the algorithms can also be used for Plonk. To one time updating the SRS of Basilisk, one would need to execute the SU algorithm that requires to compute $n + 1$ exponentiations in $\mathbb{G}_1$ and 2 exponentiations in $\mathbb{G}_2$. On the other side, to verify an i-time updated SRS, a prover would need to compute $2n$ pairing operations (independent of the number of updates), while a verifier would need to compute $4i + 2n + 2$ pairings.

SRS Update, $(\mathsf{srs}_i, \Pi_i) \leftarrow \mathsf{SU}\left(\mathsf{srs}_{i-1}, \{\Pi_{j-1}\}_{j=0}^{i-1}\right)$**:** Given $\left(\mathsf{srs}_{i-1}, \{\Pi_{j-1}\}_{j=0}^{i-1}\right)$,

- Parse $\mathsf{srs}_{i-1} := \left(\left(\left[x_{i-1}^k\right]_1\right)_{k=1}^n, [x_{i-1}]_2, u_{i-1}\right)$;
- Sample $\bar{x}_i \leftarrow \mathbb{Z}_p$; $\boxed{\text{Chooses an arbitrary } \bar{u}_i \in \mathbb{Z}_p^\star}$;
- Set $\boxed{u_i := \bar{u}_i \cdot u_{i-1}}$; $[x_i]_2 = \bar{x}_i \cdot [x_{i-1}]_2$; and $\left[x_i^k\right]_1 := \bar{x}_i^k \cdot \left[x_{i-1}^k\right]_1$ for $k = 1, 2, \cdots, n$;
- Set $\mathsf{srs}_i := \left(\left(\left[x_i^k\right]_1\right)_{k=1}^n, [x_i]_2, \boxed{u_i}\right)$, and $\Pi_i := (\Pi_i^{\mathsf{Agg}}, \Pi_i^{\mathsf{Ind}}) := ([x_i]_1, ([\bar{x}_i]_1, [\bar{x}_i]_2))$;
- Return (srs_i, Π_i);

SRS Verify, $(\bot/1) \leftarrow \mathsf{SV}\left(\mathsf{srs}_i, (\Pi_j)_{j=0}^i, \mathsf{party}\right)$**:** To verify (an i-time updated) $\mathsf{srs}_i := \left(\left(\left[x_i^k\right]_1\right)_{k=1}^n, [x_i]_2, \boxed{u_i}\right)$, and $\Pi_j := (\Pi_j^{\mathsf{Agg}}, \Pi_j^{\mathsf{Ind}}) := \left([x_j]_1, \left([\bar{x}_j]_1, [\bar{x}_j]_2\right)\right)$ for $j = 0, 1, \cdots, i$:

If $\mathsf{party} = \mathsf{P}$:

1. For $k = 1, 2, \cdots, n$: check if $\left[x_i^k\right]_1 \bullet [1]_2 = \left[x_i^{k-1}\right]_1 \bullet \left[x_i^1\right]_2$;
2. $\boxed{\text{Check if } u_i \in \mathbb{Z}_p^\star;}$

If $\mathsf{party} = \mathsf{V}$:

- If $i = 0$: srs_0 is sampled by $\mathsf{verifier}$, and it does not need to be verified.
- If $i \geq 1$:
  1. Check that $[x_0]_1 = [\bar{x}_0]_1$;
  2. For $j = 0, 1, \cdots, i$: check if $[\bar{x}_j]_1 \bullet [1]_2 = [1]_1 \bullet [\bar{x}_j]_2$;
  3. For $j = 1, 2, \cdots, i$: check if $[x_j]_1 \bullet [1]_2 = [x_{j-1}]_1 \bullet [\bar{x}_j]_2$;
  4. For $k = 1, 2, \cdots, n$: check if $\left[x_i^k\right]_1 \bullet [1]_2 = \left[x_i^{k-1}\right]_1 \bullet \left[x_i^1\right]_2$;
  5. $\boxed{\text{Check if } u_i \in \mathbb{Z}_p^\star;}$

Return 1 if all the checks passed, otherwise return $\bot$.

Fig. 9: SU and SV algorithms for Basilisk (and Plonk without $\boxed{\text{the elements } u_i}$).

Using a minimally modified versions of the algorithms given in Fig. 8 and 9, to update the SRS of Plonk, one would need to compute $3m+1$ exponentiations in $\mathbb{G}_1$ and 2 exponentiations in $\mathbb{G}_2$, where m is the number of total (addition and multiplication) gates in the circuit. To verify an i-time updated SRS, a prover would need to compute $6m$ pairing operations (independent of the number of updates), while a verifier would need to compute $4i + 6m + 2$ pairings.

Security Proofs. Similar to Lunar [13], the authors of Plonk and Basilisk have proved that their schemes achieve ZK and KS. Similarly, by assuming that the simulation trapdoor x is provided to the simulator. Next, we prove that using the SG, SU and SV algorithms (given in Fig.8 and 9), under the BDH-KE assumption, one can extract the trapdoor x from a subverted or updated SRS of Basilisk and Plonk, that shows that they both satisfy Sub-ZK and Upd-KS.

Lemma 7 (Trapdoor Extraction from a Subverted SRS). *Given the algorithms in Fig. 8 and 9, suppose that there exists a PPT adversary $\mathcal{A}$ that outputs a (srs_i, Π_i) such that $\mathsf{SV}(\mathsf{srs}_i, \Pi_i, \mathsf{P}) = 1$ with a non-negligible probability. Then, by the BDH-KE assumption (given in Def. 4) there exists a PPT extractor $\mathsf{Ext}_{\mathcal{A}}$ that, given the random tape of $\mathcal{A}$ as input, outputs x_i such that running SG with x_i results in (srs_i, Π_i).*

Proof. The proof is analogue to the proof of Lemma 5. □

Lemma 8 (Trapdoor Extraction from an Updated SRS). *Given the algorithms in Fig. 8 and 9, suppose that there exists a PPT adversary $\mathcal{A}$ such that given $(\mathsf{srs}_0, \pi_0) \leftarrow \mathsf{SG}(\mathbf{R})$, $\mathcal{A}$ returns an updated SRS (srs_1, π_1), where $\mathsf{SV}(\mathsf{srs}_1, \Pi_1, \mathsf{V}) = 1$ with a non-negligible probability. Then, the BDH-KE assumption implies that there exists a PPT extractor $\mathsf{Ext}_{\mathcal{A}}$ that, given the randomness of $\mathcal{A}$ as input, outputs $\bar{x}_1$ that are used to update srs_0 and generate (srs_1, Π_1).*

Proof. The proof is analogue to the proof of Lemma 6. □

4 Batched SRS Verification Algorithms

By now, we presented SU and SV algorithms for Sonic, Marlin, Plonk, LunarLite and Basilisk, that allow the parties to update/verify the SRS and bypass the need for a TTP. However, when running an SV algorithm, the prover needs to compute at least $O(n)$ pairing operations, where n denotes the number of multiplication gates in the circuit. On the other hand, the verifier needs to compute $O(n+i)$ pairings, where i represents the number of updates done on the SRS. Consequently, even for circuits of moderate size (e.g., $n \geq 10^4$) with a considerable number of updates (e.g., $i = 100$), the efficiency of these algorithms can be severely impacted. In Section 5, we will provide concrete numerical examples to further illustrate this inefficiency.

To make them practical, we use batching techniques from [9] and construct a Batched version of the SV algorithms, BSV in short, which allow the provers to verify the SRS by $O(n)$ exponentiations (mostly, short-exponent) and constant pairings, and the verifiers by $O(n+i)$ exponentiations (mostly, short-exponent) and $O(i)$ pairings. To build the BSV algorithms, we use a corollary of the Schwartz-Zippel lemma stating that if $\sum_{i=1}^{s-1} t_i X_i + X_s = 0$ is a polynomial in $\mathbb{Z}_q[t_i]$ with coefficients $X_1, \ldots, X_s$, $t_i \leftarrow_r \{1, \ldots, 2^\kappa\}$ for $i < s$, then $X_i = 0$ for each i, with probability $1 - 1/2^\kappa$. Namely, if $\sum_{i=1}^{s-1} t_i([a_i]_1 \bullet [b_i]_2) = \sum_{i=1}^{s-1} t_i [c]_T$ for uniformly random t_i, then w.h.p., $[a_i]_1 \bullet [b_i]_2 = [c]_T$ for each individual $i = 1, 2, \cdots, s-1$. In Sec. 5, we show that the BSV algorithms can be considerably faster than SV algorithms (at the soundness error rate 2^{-80}, where 80 is a statistical security parameter) and even faster at the soundness error rate 2^{-40}.

It is worth to mention that, using the batching technique comes at the cost of a small loss of soundness: even if the batched equation verifies, there is a probability of at most $2^{-\kappa}$ that one of the non-batched (original) equations was false. In other words, the BSV algorithms will become probabilistic, and they will accept incorrect SRSs with negligible probability. Therefore, once using the BSV algorithms, one needs to modify some of our security results from Sec. 3 to achieve statistical SRS trapdoor extractability. Next, we describe a BSV algorithm for each of the studied schemes, and then discuss their efficiency.

Batched SRS Verification, $(\perp/1) \leftarrow \mathsf{BSV}(\mathsf{srs}_i, (\Pi_j)_{j=0}^{i}, \mathsf{party})$**:**

To verify (an i-time updated) $\mathsf{srs}_i := \Big(\big([x_i^k]_1, [x_i^k]_2, [a_i x_i^k]_2\big)_{k=-n}^{n}, \big([a_i x_i^k]_1\big)_{k=-n,k\neq 0}^{n}, [a_i]_T\Big)$, and $\Pi_j := \Big(\Pi_j^{\mathsf{Agg}}, \Pi_j^{\mathsf{Ind}}\Big) := \big(([x_j]_1, [a_j x_j]_1, [a_j]_2), ([\bar{x}_j]_1, [\bar{x}_j]_2, [\bar{a}_j \bar{x}_j]_1, [\bar{a}_j \bar{x}_j]_2, [\bar{a}_j]_2)\big)$ for $j = 0, 1, \cdots, i$:

If $\mathsf{party} = \mathsf{P}$:

1. Sample $\Big\{t_k, \hat{t}_k \leftarrow \mathbb{Z}_p^\star\Big\}_{k=-n}^{n}$;
2. Check if $\big(\sum_{k=-n}^{n} t_k \cdot [x_i^k]_1\big) \bullet [1]_2 = [1]_1 \bullet \big(\sum_{k=-n}^{n} t_k \cdot [x_i^k]_2\big)$;
3. Check if $\big(\sum_{k=-n+1}^{n} t_k \cdot [x_i^k]_1\big) \bullet [1]_2 = \big(\sum_{k=-n+1}^{n} t_k \cdot [x_i^{k-1}]_1\big) \bullet [x_i]_2$;
4. Check if $\Big(\sum_{k=-n,k\neq 0}^{n} \hat{t}_k \cdot [a_i x_i^k]_1\Big) \bullet [1]_2 = [1]_1 \bullet \Big(\sum_{k=-n,k\neq 0}^{n} \hat{t}_k \cdot [a_i x_i^k]_2\Big) = \Big(\sum_{k=-n,k\neq 0}^{n} \hat{t}_k \cdot [x_i^k]_1\Big) \bullet [a_i]_2$;

If $\mathsf{party} = \mathsf{V}$:

1. Sample $\big\{r_{1,j}, r_{2,j}, r_{3,j}, r_{4,j} \leftarrow \mathbb{Z}_p^\star\big\}_{j=0}^{i}$; and $\Big\{t_k, \hat{t}_k \leftarrow \mathbb{Z}_p^\star\Big\}_{k=-n}^{n}$;
2. Check that $[x_0]_1 = [\bar{x}_0]_1$, $[a_0 x_0]_1 = [\bar{a}_0 \bar{x}_0]_1$, and $[a_0]_2 = [\bar{a}_0]_2$;
3. Check if $\Big(\sum_{j=0}^{i} r_{1,j} \cdot [\bar{x}_j]_1\Big) \bullet [1]_2 = [1]_1 \bullet \Big(\sum_{j=0}^{i} r_{1,j} [\bar{x}_j]_2\Big)$;
4. Check if $\Big(\sum_{j=0}^{i} r_{2,j} \cdot [\bar{a}_j \bar{x}_j]_1\Big) \bullet [1]_2 = [1]_1 \bullet \Big(\sum_{j=0}^{i} r_{2,j} [\bar{a}_j \bar{x}_j]_2\Big) = \sum_{j=0}^{i} \big(r_{2,j} \cdot [\bar{x}_j]_1 \bullet [\bar{a}_j]_2\big)$;
5. Check if $\Big(\sum_{j=1}^{i} r_{3,j} \cdot [x_j]_1\Big) \bullet [1]_2 = \sum_{j=1}^{i} \big(r_{3,j} \cdot [x_{j-1}]_1 \bullet [\bar{x}_j]_2\big)$;
6. Check if $\Big(\sum_{j=1}^{i} r_{4,j} \cdot [a_j x_j]_1\Big) \bullet [1]_2 = \sum_{j=1}^{i} \big(r_{4,j} \cdot [x_j]_1 \bullet [a_j]_2\big) = \sum_{j=1}^{i} \big(r_{4,j} \cdot [a_{j-1} x_{j-1}]_1 \bullet [\bar{a}_j \bar{x}_j]_2\big)$;
7. Check if $\big(\sum_{k=-n}^{n} t_k \cdot [x_i^k]_1\big) \bullet [1]_2 = [1]_1 \bullet \big(\sum_{k=-n}^{n} t_k \cdot [x_i^k]_2\big)$;
8. Check if $\big(\sum_{k=-n+1}^{n} t_k \cdot [x_i^k]_1\big) \bullet [1]_2 = \big(\sum_{k=-n+1}^{n} t_k \cdot [x_i^{k-1}]_1\big) \bullet [x_i]_2$;
9. Check if $\Big(\sum_{k=-n,k\neq 0}^{n} \hat{t}_k \cdot [a_i x_i^k]_1\Big) \bullet [1]_2 = [1]_1 \bullet \Big(\sum_{k=-n,k\neq 0}^{n} \hat{t}_k \cdot [a_i x_i^k]_2\Big) = \Big(\sum_{k=-n,k\neq 0}^{n} \hat{t}_k \cdot [x_i^k]_1\Big) \bullet [a_i]_2$;

Return 1 if all the checks passed, otherwise return $\perp$.

Fig. 10: BSV: The Batched version of SV algorithm for Sonic.

Batched SV Algorithm for Sonic Fig. 10, describes the batched version of Sonic's SV algorithm (given in Fig. 3). Using the proposed BSV algorithm, to verify an i-time updated SRS of size n: 1) a prover will compute 7 pairings, $4n$ exponentiations in $\mathbb{G}_2$, and $8n-1$ exponentiations in $\mathbb{G}_1$, 2) and a verifier would need to calculate $4i+14$ pairings, $6n+2i+1$ exponentiations in $\mathbb{G}_2$, and $8n+8i+2$ exponentiations in $\mathbb{G}_1$.

Batched SV Algorithm for Marlin. Our proposed BSV algorithm for Marin is described in Fig. 11. Using the BSV algorithm, to verify an i-time updated SRS of size n: 1) a prover will compute 3 pairings and $4n$ exponentiations (mostly, short exponent) in $\mathbb{G}_1$, 2) and a verifier would need to calculate $4i+9$ pairings, $3i+2$ exponentiations in $\mathbb{G}_2$, and $4n+6i+2$ exponentiations in $\mathbb{G}_1$.

Batched SV Algorithm for LunarLite. The batched SV algorithm for LunarLite is shown in Fig. 12. Using the proposed BSV algorithm, to verify an

Batched SRS Verification, $(\perp/1) \leftarrow \mathsf{BSV}(\mathsf{srs}_i, (\Pi_j)_{j=0}^{i}, \mathsf{party})$**:** To verify (an i-time updated) $\mathsf{srs}_i := \left(\left(\left[x_i^k\right]_1, \left[\gamma_i x_i^k\right]_1\right)_{k=0}^{n}, [x_i]_2, [x_i\gamma_i]_2\right)$, and $\Pi_j := (\Pi_j^{\mathsf{Agg}}, \Pi_j^{\mathsf{Ind}}) := (([\gamma_i]_1, [x_i\gamma_i]_1, [x_i]_2), ([\bar{x}_i]_1, [\bar{\gamma}_i]_1, [\bar{x}_i]_2, [\bar{x}_i\bar{\gamma}_i]_2))$; for $j = 0, 1, \cdots, i$:

If $\mathsf{party} = \mathsf{P}$:

1. Sample $\{t_{1,k}, t_{2,k} \leftarrow \mathbb{Z}_p^\star\}_{k=1}^{n}$;
2. Check if $\left([\gamma_i x_i]_1 + \sum_{k=1}^{n}(t_{1,k} \cdot \left[x_i^k\right]_1 + t_{2,k} \cdot \left[\gamma_i x_i^k\right]_1)\right) \bullet [1]_2 = (\sum_{k=1}^{n}(t_{1,k} \cdot \left[x_i^{k-1}\right]_1 + t_{2,k} \cdot \left[\gamma_i x_i^{k-1}\right]_1)) \bullet [x_i]_2 + [1]_1 \bullet [\gamma_i x_i]_2$;

If $\mathsf{party} = \mathsf{V}$:

- If $i = 0$: srs_0 is sampled by $\mathsf{verifier}$, and it does not need to be verified.
- If $i \geq 1$, act as follows,
  1. Sample $\left\{r_{1,j}, r_{2,j}, r_{3,j}, r_{4,j} \leftarrow \mathbb{Z}_p^\star\right\}_{j=0}^{i}$; and $\left\{t_k, \hat{t}_k \leftarrow \mathbb{Z}_p^\star\right\}_{k=1}^{n}$;
  2. Check if $[\gamma_0]_1 = [\bar{\gamma}_0]_1$ and $[x_0]_2 = [\bar{x}_0]_2$;
  3. Check if $(\sum_{j=0}^{i} r_{1,j} \cdot [\bar{x}_j]_1) \bullet [1]_2 = [1]_1 \bullet (\sum_{j=0}^{i} r_{1,j} [\bar{x}_j]_2)$;
  4. Check if $[1]_1 \bullet (\sum_{j=0}^{i} r_{2,j} \cdot [\bar{\gamma}_j\bar{x}_j]_2) = \sum_{j=0}^{i}(r_{2,j} [\bar{\gamma}_j]_1 \bullet [\bar{x}_j]_2)$;
  5. Check if $[1]_1 \bullet (\sum_{j=1}^{i} r_{3,j} \cdot [x_j]_2) = \sum_{j=1}^{i}(r_{3,j} \cdot [\bar{x}_j]_1 \bullet [x_{j-1}]_2)$;
  6. Check if $(\sum_{j=1}^{i} r_{4,j} \cdot [\gamma_j x_j]_1) \bullet [1]_2 = \sum_{j=1}^{i}(r_{4,j} \cdot [\gamma_{j-1}x_{j-1}]_1 \bullet [\bar{\gamma}_j\bar{x}_j]_2) = \sum_{j=1}^{i}(r_{4,j} \cdot [\gamma_j]_1 \bullet [x_j]_2)$;
  7. Check if $\left([\gamma_i x_i]_1 + \sum_{k=1}^{n}(t_k \cdot \left[x_i^k\right]_1 + \hat{t}_k \cdot \left[\gamma_i x_i^k\right]_1)\right) \bullet [1]_2 = \left(\sum_{k=1}^{n}\left(t_k \cdot \left[x_i^{k-1}\right]_1 + \hat{t}_k \cdot \left[\gamma_i x_i^{k-1}\right]_1\right)\right) \bullet [x_i]_2 + [1]_1 \bullet [\gamma_i x_i]_2$;

Return 1 if all the checks passed, otherwise return $\perp$.

Fig. 11: BSV: The Batched version of SV algorithm for Marlin.

Batched SRS Verification, $(\perp/1) \leftarrow \mathsf{BSV}(\mathsf{srs}_i, (\Pi_j)_{j=0}^{i}, \mathsf{party})$**:** To verify (an i-time updated) $\mathsf{srs}_i := \left(\left[x_i^k\right]_1, \left[x_i^k\right]_2\right)_{k=1}^{n}$, and $\Pi_j := (\Pi_j^{\mathsf{Agg}}, \Pi_j^{\mathsf{Ind}}) := ([x_j]_1, ([\bar{x}_j]_1, [\bar{x}_j]_2))$ for $j = 0, 1, \cdots, i$:

If $\mathsf{party} = \mathsf{P}$:

1. Sample $t_k \leftarrow \mathbb{Z}_p^\star$ for $k = 2, \cdots, n$;
2. Check if $([x_i]_1 + \sum_{k=2}^{n} t_k \cdot \left[x_i^k\right]_1) \bullet [1]_2 = [1]_1 \bullet ([x_i]_1 + \sum_{k=2}^{n} t_k \cdot \left[x_i^k\right]_2) = ([1]_1 + \sum_{k=2}^{n} t_k \left[x_i^{k-1}\right]_1) \bullet [x_i]_2$;

If $\mathsf{party} = \mathsf{V}$:

- If $i = 0$: srs_0 is sampled by $\mathsf{verifier}$, and it does not need to be verified.
- If $i \geq 1$, act as follows,
  1. Sample $t_j, s_j \leftarrow \mathbb{Z}_p^\star$ for $j = 1, \cdots, i$; and $h_k \leftarrow \mathbb{Z}_p^\star$ for $k = 1, \cdots, n$;
  2. Check if $[x_0]_1 = [\bar{x}_0]_1$;
  3. Check if $([\bar{x}_0]_1 + \sum_{j=1}^{i}(t_j [\bar{x}_j]_1 + s_j [x_j]_1) + 2\sum_{k=1}^{n} h_k \left[x_i^k\right]_1) \bullet [1]_2 = [1]_1 \bullet ([\bar{x}_0]_2 + \sum_{j=1}^{i} t_j [\bar{x}_j]_2 + \sum_{k=1}^{n} h_k \left[x_i^k\right]_2) + \sum_{j=1}^{i}(s_j [x_{j-1}]_1 \bullet [\bar{x}_j]_2) + (\sum_{k=1}^{n} h_k \left[x_i^{k-1}\right]_1) \bullet [x_i]_2$

Return 1 if all the checks passed, otherwise return $\perp$.

Fig. 12: BSV: The Batched version of SV algorithm for LunarLite.

i-time updated SRS of size n, 1) a P will compute 3 pairings, $n-1$ exponentiations in $\mathbb{G}_2$, and $2n-2$ exponentiations in $\mathbb{G}_1$, 2) and a V would need to compute $i+3$ pairings, $n+i$ exponentiations in $\mathbb{G}_2$, and $2n+3i$ exponentiations in $\mathbb{G}_1$.

Batched SRS Verification, $(\perp/1) \leftarrow \mathsf{BSV}(\mathsf{srs}_i, (\Pi_j)_{j=0}^{i}, \mathsf{party})$**:** To verify (an i-time updated) $\mathsf{srs}_i := ((\left[x_i^k\right]_1)_{k=1}^{n}, [x_i]_2, u_i)$, and $\Pi_j := (\Pi_j^{\mathsf{Agg}}, \Pi_j^{\mathsf{Ind}}) := ([x_j]_1, ([\bar{x}_j]_1, [\bar{x}_j]_2))$ for $j = 0, 1, \cdots, i$:

If $\mathsf{party} = \mathsf{P}$:

1. Sample $t_k \leftarrow \mathbb{Z}_p^\star$ for $k = 2, \cdots, n$;
2. Check if $([x_i]_1 + \sum_{k=2}^{n} t_k \cdot \left[x_i^k\right]_1) \bullet [1]_2 = ([1]_1 + \sum_{k=2}^{n} t_k \left[x_i^{k-1}\right]_1) \bullet [x_i]_2$;
3. Check if $u_i \in \mathbb{Z}_p^\star$;

If $\mathsf{party} = \mathsf{V}$:

- If $i = 0$: srs_0 is sampled by $\mathsf{verifier}$, and it does not need to be verified.
- If $i \geq 1$, act as follows,
  1. Sample $t_j, s_j \leftarrow \mathbb{Z}_p^\star$ for $j = 1, \cdots, i$; and $h_k \leftarrow \mathbb{Z}_p^\star$ for $k = 1, \cdots, n$;
  2. Check that $[x_0]_1 = [\bar{x}_0]_1$;
  3. Check if $([\bar{x}_0]_1 + \sum_{j=1}^{i}(t_j [\bar{x}_j]_1 + s_j [x_j]_1) + \sum_{k=1}^{n} h_k \left[x_i^k\right]_1) \bullet [1]_2 = [1]_1 \bullet ([\bar{x}_0]_2 + \sum_{j=1}^{i} t_j [\bar{x}_j]_2) + \sum_{j=1}^{i}(s_j [x_{j-1}]_1 \bullet [\bar{x}_j]_2) + (\sum_{k=1}^{n} h_k \left[x_i^{k-1}\right]_1) \bullet [x_i]_2$
  4. Check if $u_i \in \mathbb{Z}_p^\star$;

Return 1 if all the checks passed, otherwise return $\perp$.

Fig. 13: The BSV algorithm for Basilisk (and Plonk without checking u_i).

Batched SV Algorithm for Basilisk and Plonk. Similar to the previous cases, in Fig. 13, we propose a probabilistic batched variant of the Basilisk's SV algorithm (given in Fig. 9). A slightly modified version of this algorithm can be used for Plonk, just by removing the checks related to u_i. However, one should pay attention that Plonk and Basilisk are using two difference constraint systems, and the value of n will be different once one uses the same BSV for both. In summary, by running the BSV algorithm given in Fig. 13, to verify an i-time updated SRS of size n: 1) a prover will compute 2 pairings and $2n - 2$ exponentiations in $\mathbb{G}_1$, 2) and a verifier would need to compute $i + 3$ pairings, i exponentiations in $\mathbb{G}_2$, and $2n + 3i$ exponentiations in $\mathbb{G}_1$.

5 Performance Analysis and Identifiable Security

5.1 Implementation Results

Next, we evaluate the efficiency of the presented algorithms using a prototype implementation in Arkworks library [1], which is a Rust library for developing and programming with zk-SNARKs. We have made the source code of our benchmarks publicly available to the research community for reproducibility and further experimentation [2]. For benchmarking Sonic, Plonk, LunarLite, and Basilisk we use the algorithms constructed in Sec. 3 and Sec. 4. But in case of Marlin, we use a variant of it, which is implemented in Arkworks [3]. The original paper

[1] Available on `https://github.com/arkworks-rs`.

[2] Our open-source implementations can be accessed on the Git page at `https://github.com/Baghery/BMS23`.

[3] Available on `https://github.com/arkworks-rs/marlin`.

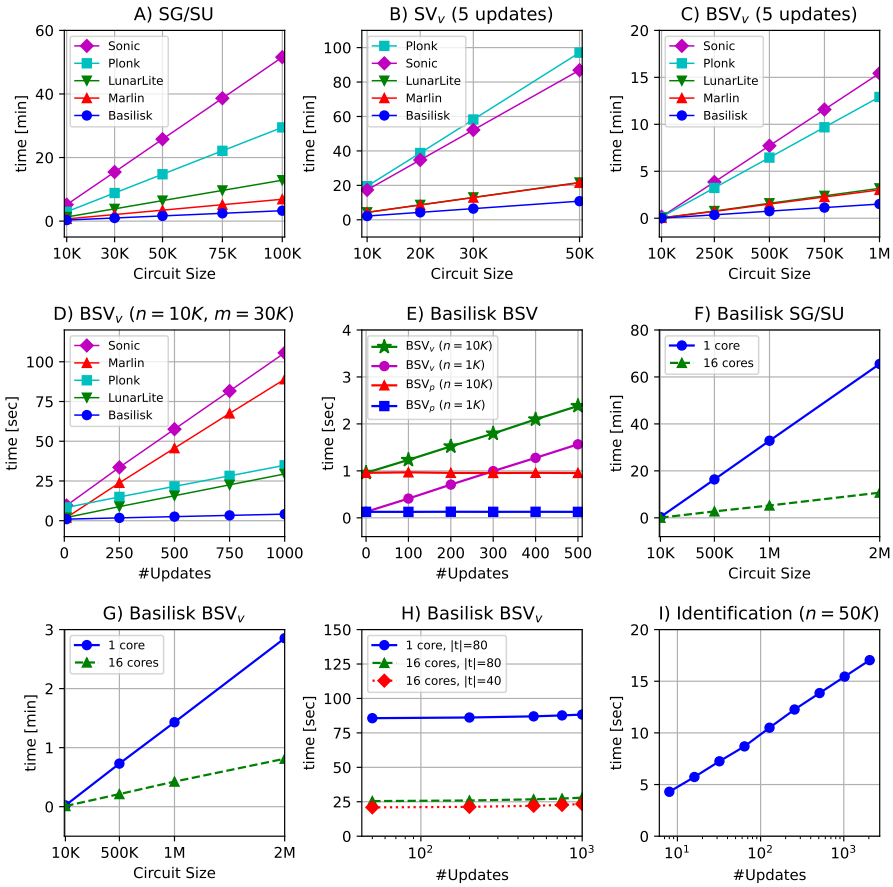

Fig. 14: A) SG or SU, B) $\mathsf{SV_V}$ for a fixed $i = 5$, C) $\mathsf{BSV_V}$ for a fixed $i = 5$, D) $\mathsf{BSV_V}$ for a fixed circuit size ($n = 10K, m = 30K$), E) Comparison of Basilisk's $\mathsf{BSV_P}$ and $\mathsf{BSV_V}$ for a fixed $n = 10K$, F) Basilisk's SG/SU with multi-threading, G) Basilisk's $\mathsf{BSV_V}$ with multi-threading, H) Basilisk's $\mathsf{BSV_V}$ with $n = 10^6$, and different security parameters in batching, I) Identifying a malicious SRS updater with recursive verification in Basilisk.

does not explain this variant, which uses a different PC scheme to reduce proof size, which is a variant of the scheme proposed in [20]. We built the associated $(\mathsf{SG}, \mathsf{SU}, \mathsf{SV}, \mathsf{BSV})$ algorithms for that version in App. B.2.

Our empirical analysis are done with the elliptic curves BLS12-381 that is estimated to achieve between 117 and 120 bits security [33]. All experiments are done on a desktop machine with Ubuntu 20.4.2 LTS, an Intel Core i9-9900 processor at base frequency 3.1 GHz, and 128GB of memory. All algorithms first are executed in the single-thread mode, while later we show that they all can be parallelized and executed in the multi-thread mode. We also report the benchmarks for Basilisk in the multi-thread mode, with 16 threads. For the benchmarks, we report the running times of all the proposed algorithms, for an arithmetic circuit with different circuit sizes, and by circuit size we mean sum of the multiplica-

tion and the addition gates. For Plonk, whose constraint system encodes both multiplication and addition gates, we set the number of addition gates $2\times$ the number of multiplication gates. This choice was based on the evaluation done in the original paper [21]. Motivated by the blockchains and large-scale applications, we also report the SRS verification/updating times for a big number of users and large circuits. All times are expressed in seconds or minutes. In the execution of the BSV algorithms, we first sample some vectors $\vec{t_i}$ of random numbers from the range $[1\,..\,2^{80}]$, the time of sampling randomnesses are not included in the run times of BSV algorithm, as they can be pre-computed. Based on earlier results, one can re-use the same randomness for different verification equations, and zk-SNARKs.

The graphs in Fig. 14 summarize our implementation results based on different criteria for all our studied zk-SNARKs. In the rest, we go through them sequentially and explain the key points. The plot A compares the run times of SG and SU in the single-thread mode, for all the studied zk-SNARKs. Naturally, the shorter SRS, the faster SG and SU algorithms. The plot B presents the run times of SV algorithm executed by V, for a 5-time updated SRS and various circuit sizes. As it can be seen, standard SV algorithms can be very slow for even small circuits, e.g. circuits of sizes $< 50K$ (this is why we are not giving its timings for $n > 50K$). In this case, since the size of SRS ($n = 50K$), is considerably larger than the number of updates ($i = 5$), then the run times of $\mathsf{SV_P}$ and $\mathsf{SV_V}$ are almost the same, therefore $\mathsf{SV_P}$ is omitted from the plot.

The plot C illustrates the efficiency of BSV algorithm run by V, for $i = 5$ (5-time updated SRS) and different circuit sizes. One can see that they are considerably faster than standard SV algorithms, and in some cases they are very efficient even for large circuits. e.g. circuits of sizes $> 1M$. Similar to the last plot, in this setting again the run times of $\mathsf{SV_P}$ and $\mathsf{SV_V}$ are very close. In plot D, we set the circuit size fixed ($n = 10K$ multiplication gates, $m = 30K$ total gates) and plot the run times of $\mathsf{BSV_V}$ algorithms for different number of updates. Similar to the previous plots, we observe that the setup phase of Basilisk can be considerably faster than other schemes. Therefore, in the rest of benchmarks, we mainly used Basilisk's algorithms. In plot E, we compare the run times of Basilisk's BSV algorithm executed by the prover ($\mathsf{BSV_P}$) and verifier ($\mathsf{BSV_V}$), for a circuit with $n = 1K$ or $10K$ multiplication gates, and different numbers of updates. As it can be seen, for the cases that a small circuit is updated many times, $\mathsf{BSV_P}$ can be significantly faster, independent of the number of updates. The plot shows that $\mathsf{BSV_P}$ for $n = 10K$, is as efficient as $\mathsf{BSV_V}$ for $n = 1K$ and $i \approx 300$.

By now all evaluations are done in the single-thread mode. In the rest, in both plots F and G, we execute the algorithms of Basilisk in the multi-thread mode and re-evaluate the efficiency of SU (or SG) and $\mathsf{BSV_V}$, for various circuits and different number of updates. We observed that, the SRS of Basilisk for a particular circuit with 2M multiplication gates, can be generated/updated in about 11 min, and verified in less than 1 minute. As mentioned before, within the BSV algorithms, the randomness vector $\vec{t_i}$ are sampled from $[1\,..\,2^{80}]$ which

assures that the batching causes security gap not bigger than 2^{-80}. This is a conservative approach. In plot H, we compare the run times of BSV_V for Basilisk in the case that the coordinates of $\vec{t_i}$ were chosen from $[1\,..\,2^{40}]$. This makes the SRS verification even faster, but at the cost of a bigger error rate, i.e., 2^{-40}.

5.2 Identifiable Security in the Updatable SRS Model

In the updatable SRS model [26], the initial SRS generator and the follow-up SRS updaters attach a proof to each updated SRS, and the parties do not store every updated SRS but only update proofs. At the end, each party runs the SV (or BSV) algorithm once to verify the validity of proofs in a chain and then uses the final proof to check the well-formedness of the final SRS (see Fig. 1). In lemmas 1-8, we also observed that after the final update on SRS, it is sufficient that all the participants in the SRS generation/updating phases run the SV (or BSV) algorithm only once. In the rest, this case is referred to as the *optimistic case* or *optimistic verification*. As we observed in Fig. 14, in this case the setup phase of updatable zk-SNARKs can be significantly fast, and can easily be scaled for a large number of users (e.g. thousands of parties), without the need for a third party. However, then the parties would only abort a maliciously updated SRS at the end, without identifying a malicious party. This can lead to repeat the SRS generation and updates all over again. Note that, the SV (and BSV) algorithm verifies the proofs $\mathit{\Pi}_0$ till $\mathit{\Pi}_i$, and the final SRS srs_i. If a malicious SRS generator/updater generates a valid proof but an invalid SRS, it cannot be detected by just verifying the proofs. To deal with this concern, a naive solution is to verify the SRS after each update (by either all the participants or a TTP) and identify the malicious party. In practice, the above approach would be impractical for large scale applications.

Identifying a Malicious Updater with Logarithmic Verification. Next, we describe an efficient approach to identify a malicious party in the updatable SRS model. To this end, the parties need to store all the transcripts, as in current ceremonies, and then recursively run the BSV (or SV) algorithm for one SRS and a smaller set of proofs. More precisely, parties would run the BSV (or SV) algorithm of the target zk-SANRK, with a single SRS and $\frac{i}{2^1}, \frac{i}{2^2}, \cdots, \frac{i}{2^{\log i}}$ proofs, respectively. Note that with this approach, only $\lceil \log i \rceil + 1$ of SRSs are verified (e.g., boldface SRSs $\mathsf{srs}_{15}, \mathsf{srs}_7, \mathsf{srs}_{11}, \mathsf{srs}_9, \mathsf{srs}_{10}$ in Fig. 15), instead of i. As in practice, the circuit size is considerably higher than the number of SRS updates, e.g. 2^{22} vs. 100 in current ceremonies, therefore the run time of SV (and BSV) is dominated by the size of SRS, rather than the number of updates. Due to this fact, in practice, the proposed verification approach can be considerably faster than the naive solution. Fig. 15, presents an example of such recursive execution of BSV algorithms for $i = 15$. We also evaluate the performance of this approach with a sample implementation. The plot I in Fig. 14, illustrates the required time to identify a malicious updater in Basilisk's setup for different number of updates with the SRS of a circuit with $n = 50K$ multiplication gates. As it can be seen, for 2000-time updated SRS of length $50K$, the first malicious updater can be

Fig. 15: Recursive execution of BSV to identify a malicious SRS updater.

identified in less than 20 sec. In similar settings, where $n >> i$, the identification time would be independent of the precise position of the malicious updater, and it will take an approximate run time of $\log i$ times that of a single BSV.

As an optimization, one may notice that once a verifier runs the BSV algorithm on the final SRS, e.g. $\mathsf{BSV}_0^{(16)}$ in the mentioned example, we already compute the batched form of the proof elements required in all the follow-up steps of the recursive search, as e.g. $\sum_{i=0}^{15} t_i [x_i]_1 = \sum_{i=0}^{7} t_i [x_i]_1 + \sum_{i=8}^{15} t_i [x_i]_1 = \sum_{i=0}^{3} t_i [x_i]_1 + \sum_{i=4}^{7} t_i [x_i]_1 + \sum_{i=8}^{11} t_i [x_i]_1 + \sum_{i=12}^{15} t_i [x_i]_1$. By storing a proper set of batched proofs, one can speed up the follow-up executions of BSV. This optimization is more effective in cases that the circuit size is small but the SRS is updated many times. As another optimization, one can precompute the batched version of the checks on some intermediate SRSs, e.g. $\mathsf{srs}_{11}, \mathsf{srs}_7, \mathsf{srs}_3$, and speed-up the run times of BSV algorithms in the follow-up steps. Note that our BSV and SV algorithms, by default verifies all the proofs for $j = 0$ till the final SRS srs_i, i.e. $j = i$. In the recursive execution, we need to run the BSV (or SV) algorithm for a particular set of SRSs and proofs. In those cases, one can feed proper starting and finishing indexes to the BSV (or SV) algorithms. For instance, to check the SRS srs_{11} and the set of proofs $\{\Pi_8, \Pi_9, \Pi_{10}, \Pi_{11}\}$ one needs to run the algorithms for $j = 8$ till $j = 11$, which will verify a batched variant of $(\Pi_8, \Pi_9, \Pi_{10}, \Pi_{11})$ and the final SRS srs_{11}.

In practice, if the values of i and n will be huge, it might happen that the setup phase would take a long time, especially if a malicious update occurs during the earlier updates. To minimize the run time, as well as to gain the benefits of the optimistic verification, an effective solution would be to verify the updated SRS after a particular number of updates, i.e. one would need to verify the updated SRS every k updates, where $1 < k < i$. Basically, the idea is rather than verifying every update (the slowest case), or all i updates once (the fastest case), the parties will verify the SRS after each k updates. If the verification of srs_k was successful, then the parties will continue with updating the SRS. If not, they would use the recursive search approach (given in Fig. 15) to find the first malicious updater and then will continue the SRS update from there (without the malicious updater).

Since the entire described procedure is accountable, in practice one can minimize the risk of a malicious SRS update significantly by enforcing a high penalty for a malicious SRS updater.

6 Conclusion

In this study, we examined the setup phase of updatable zk-SNARKs. We constructed the necessary algorithms, namely $(\mathsf{SG}, \mathsf{SU}, \mathsf{SV})$, for the setup phase of various updatable zk-SANRKs, including Sonic, Plonk, Marlin, Lunar, and Basilisk. To make SV algorithms practical, we also presented a batched version of them, called BSV. We constructed the algorithms for the most efficient version of each zk-SNARK, in terms of proof size. However, the proposed algorithm can be adapted to their different versions. Our results show that in a few cases, to achieve better efficiency in the setup phase, one option would be to use a version of the studied schemes, with a shorter SRS but slightly larger proofs and slower provers. For instance, Lunar [13] has a version, so-called LunarLite2x, which has the same SRS as Basilisk, therefore can be as efficient as Basilisk in the setup phase, but in cost of slightly longer proofs and slower prover. In another example, we observed that Counting Vampires [31] has only two fewer group elements than Basilisk in the proof, but its SRS size is 17× larger and such an SRS can result in a prolonged setup.

Meanwhile, we observed that to achieve Sub-ZK/Upd-KS in updatable zk-SANRKs, a more realistic model for security proofs could be the AGM with hashing [30], rather than the original AGM [19].

Moreover, we showed that pairing-based updatable zk-SNARKs, or other primitives constructed in the updatable SRS model, by default achieve security with abort, and the parties cannot identify a malicious SRS generator/updater. A naive solution to deal with this concern is verifying the SRS after each update (either by the parties or a third party), but it can be impractical in a large-scale application. To make it practical, we proposed an efficient recursive verification approach, that allows the parties to identify a malicious SRS updater by a logarithmic number of SRS verification (instead of linear) in the number of updates. We believe our proposed approach to achieve identifiable security, can also be used in the MPC SRS generation protocols [29], as well as in other cryptographic primitives (like commitments, signatures, encryptions) constructed in the updatable SRS model [16,3,7,22,6].

Finally, our empirical analysis showed that the algorithms are practical for large-scale applications, and among the current updatable zk-SNARKs, Basilisk (and the Lunarlite2x variant of Lunar) can have the fastest setup phase. Counting Vampires, Sonic and Plonk can have a very slow setup phase, which is mainly because of having a very long SRS or using a specific constraint system (i.e., Plonk) that encodes both addition and multiplication gates.

Acknowledgement. This work has been supported in part by the Defense Advanced ResearchProjects Agency (DARPA) under contract No. HR001120C0085, by the FWO underan Odysseus project GOH9718N, by the Research Council KU Leuven C1 on Security and Privacy for Cyber-Physical Systems and the Internet of Things with contract number C16/15/058, and by CyberSecurity Research Flanders with reference number VR20192203.

References

1. Behzad Abdolmaleki, Karim Baghery, Helger Lipmaa, Janno Siim, and Michal Zajac. UC-secure CRS generation for SNARKs. In Johannes Buchmann, Abderrahmane Nitaj, and Tajje eddine Rachidi, editors, *AFRICACRYPT 19*, volume 11627 of *LNCS*, pages 99–117. Springer, Heidelberg, July 2019.
2. Behzad Abdolmaleki, Karim Baghery, Helger Lipmaa, and Michal Zajac. A subversion-resistant SNARK. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part III*, volume 10626 of *LNCS*, pages 3–33. Springer, Heidelberg, December 2017.
3. Behzad Abdolmaleki, Sebastian Ramacher, and Daniel Slamanig. Lift-and-shift: Obtaining simulation extractable subversion and updatable SNARKs generically. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1987–2005. ACM Press, November 2020.
4. Karim Baghery. Subversion-resistant simulation (knowledge) sound NIZKs. In Martin Albrecht, editor, *17th IMA International Conference on Cryptography and Coding*, volume 11929 of *LNCS*, pages 42–63. Springer, Heidelberg, December 2019.
5. Karim Baghery. Subversion-resistant commitment schemes: Definitions and constructions. In Konstantinos Markantonakis and Marinella Petrocchi, editors, *Security and Trust Management - 16th International Workshop, STM 2020, Guildford, UK, September 17-18, 2020, Proceedings*, volume 12386 of *Lecture Notes in Computer Science*, pages 106–122. Springer, 2020.
6. Karim Baghery and Navid Ghaedi Bardeh. Updatable NIZKs from non-interactive zaps. In Alastair R. Beresford, Arpita Patra, and Emanuele Bellini, editors, *CANS 22*, volume 13641 of *LNCS*, pages 23–43. Springer, Heidelberg, November 2022.
7. Karim Baghery and Mahdi Sedaghat. Tiramisu: Black-box simulation extractable NIZKs in the updatable CRS model. In Mauro Conti, Marc Stevens, and Stephan Krenn, editors, *CANS 21*, volume 13099 of *LNCS*, pages 531–551. Springer, Heidelberg, December 2021.
8. Mihir Bellare, Georg Fuchsbauer, and Alessandra Scafuro. NIZKs with an untrusted CRS: Security in the face of parameter subversion. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part II*, volume 10032 of *LNCS*, pages 777–804. Springer, Heidelberg, December 2016.
9. Mihir Bellare, Juan A. Garay, and Tal Rabin. Batch Verification with Applications to Cryptography and Checking. In Claudio L. Lucchesi and Arnaldo V. Moura, editors, *LATIN 1998*, volume 1380 of *LNCS*, pages 170–191, Campinas, Brazil, April 20–24, 1998. Springer, Heidelberg.
10. Eli Ben-Sasson, Alessandro Chiesa, Matthew Green, Eran Tromer, and Madars Virza. Secure sampling of public parameters for succinct zero knowledge proofs. In *2015 IEEE Symposium on Security and Privacy*, pages 287–304. IEEE Computer Society Press, May 2015.

11. Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 103–112. ACM, 1988.
12. Sean Bowe, Ariel Gabizon, and Ian Miers. Scalable multi-party computation for zk-SNARK parameters in the random beacon model. Cryptology ePrint Archive, Report 2017/1050, 2017. `https://eprint.iacr.org/2017/1050`.
13. Matteo Campanelli, Antonio Faonio, Dario Fiore, Anaïs Querol, and Hadrián Rodríguez. Lunar: A toolbox for more efficient universal and updatable zkSNARKs and commit-and-prove extensions. In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT 2021, Part III*, volume 13092 of *LNCS*, pages 3–33. Springer, Heidelberg, December 2021.
14. Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Psi Vesely, and Nicholas Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. Cryptology ePrint Archive, Report 2019/1047, 2019. `https://eprint.iacr.org/2019/1047`.
15. Ivan Damgård. Towards practical public key systems secure against chosen ciphertext attacks. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 445–456. Springer, Heidelberg, August 1992.
16. Vanesa Daza, Carla Ràfols, and Alexandros Zacharakis. Updateable inner product argument with logarithmic verifier and applications. In Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, and Vassilis Zikas, editors, *PKC 2020, Part I*, volume 12110 of *LNCS*, pages 527–557. Springer, Heidelberg, May 2020.
17. Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194. Springer, Heidelberg, August 1987.
18. Georg Fuchsbauer. Subversion-zero-knowledge SNARKs. In Michel Abdalla and Ricardo Dahab, editors, *PKC 2018, Part I*, volume 10769 of *LNCS*, pages 315–347. Springer, Heidelberg, March 2018.
19. Georg Fuchsbauer, Eike Kiltz, and Julian Loss. The algebraic group model and its applications. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 33–62. Springer, Heidelberg, August 2018.
20. Ariel Gabizon. AuroraLight: Improved prover efficiency and SRS size in a sonic-like system. Cryptology ePrint Archive, Report 2019/601, 2019. `https://eprint.iacr.org/2019/601`.
21. Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953, 2019. `https://eprint.iacr.org/2019/953`.
22. Chaya Ganesh, Hamidreza Khoshakhlagh, Markulf Kohlweiss, Anca Nitulescu, and Michał Zając. What makes fiat–shamir zksnarks (updatable srs) simulation extractable? In Clemente Galdi and Stanislaw Jarecki, editors, *Security and Cryptography for Networks*, pages 735–760, Cham, 2022. Springer International Publishing.
23. Sanjam Garg, Mohammad Mahmoody, Daniel Masny, and Izaak Meckler. On the round complexity of OT extension. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 545–574. Springer, Heidelberg, August 2018.
24. Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on computing*, 18(1):186–208, 1989.

25. Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 305–326. Springer, Heidelberg, May 2016.
26. Jens Groth, Markulf Kohlweiss, Mary Maller, Sarah Meiklejohn, and Ian Miers. Updatable and universal common reference strings with applications to zk-SNARKs. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 698–728. Springer, Heidelberg, August 2018.
27. Thomas Icart. How to hash into elliptic curves. In Shai Halevi, editor, *CRYPTO 2009*, volume 5677 of *LNCS*, pages 303–316. Springer, Heidelberg, August 2009.
28. Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 177–194. Springer, Heidelberg, December 2010.
29. Markulf Kohlweiss, Mary Maller, Janno Siim, and Mikhail Volkhov. Snarky ceremonies. In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT 2021, Part III*, volume 13092 of *LNCS*, pages 98–127. Springer, Heidelberg, December 2021.
30. Helger Lipmaa. A unified framework for non-universal SNARKs. In Goichiro Hanaoka, Junji Shikata, and Yohei Watanabe, editors, *PKC 2022, Part I*, volume 13177 of *LNCS*, pages 553–583. Springer, Heidelberg, March 2022.
31. Helger Lipmaa, Janno Siim, and Michal Zajac. Counting vampires: From univariate sumcheck to updatable ZK-SNARK. In Shweta Agrawal and Dongdai Lin, editors, *ASIACRYPT 2022, Part II*, volume 13792 of *LNCS*, pages 249–278. Springer, Heidelberg, December 2022.
32. Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 2111–2128. ACM Press, November 2019.
33. NCC. Zcash overwinter consensus and sapling cryptography review. `https://research.nccgroup.com/wp-content/uploads/2020/07/NCC_Group_Zcash2018_Public_Report_2019-01-30_v1.3.pdf`, 2019.
34. Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, pages 238–252. IEEE Computer Society Press, May 2013.
35. Carla Ràfols and Arantxa Zapico. An algebraic framework for universal and updatable SNARKs. Cryptology ePrint Archive, Report 2021/590, 2021. `https://eprint.iacr.org/2021/590`.

A Appendix

A.1 Comparison of Updatable zk-SNARKs

Table 2, compares the efficiency of some known universal and updatable zk-SNARKs.

A.2 Polynomial Commitments for a Single Degree Bound

PCs are a key primitive to build an efficient updatable zk-SNARK. In a PC scheme, a prover commits to a polynomial, and later opens it to an evaluation of the polynomial at a random point, chosen by the verifier [28]. Next, we recall the definition of PC schemes that support a single degree bound chosen in the setup phase [14]. Over a field family, $\mathcal{F}$, a PC scheme, PC, for a single degree bound and a single evaluation point consists of the PPT algorithms $\mathsf{PC} = (\mathsf{KG}, \mathsf{Com}, \mathsf{Open}, \mathsf{Vfy})$, defined as follows:

– $(\mathsf{ck}, \mathsf{vk}) \leftarrow \mathsf{KG}(1^\lambda, D)$: Given λ in its unary representation, and a maximum degree bound $D \in \mathbb{N}$ as inputs, it then samples and returns a key pair $(\mathsf{ck}, \mathsf{vk})$. Note that the keys contain the description of a finite field $\mathbb{F} \in \mathcal{F}$.

Table 2: An efficiency comparison of the existing updatable and universal zk-SNARKs. The pairing operation is shown by P. Additionally, m: #total gates, m_0: #public input wires, n: #multiplication gates, k: #matrix elements with non-zero values describing the circuit.

		$\lvert\mathsf{srs}\rvert$	$\lvert\pi\rvert$	SG	P	V
Sonic [32]	$\mathbb{G}_1$	$4n-1$	20	$4n-1$	$273n$	$7P$
	$\mathbb{G}_2$	$4n$	–	$4n$	–	
	$\mathbb{F}$	–	16	–	$O(k\log(k))$	$O(m_0+\log(k))$
Plonk [21]	$\mathbb{G}_1$	$3m$	7	$3m$	$11m$	$2P$
	$\mathbb{G}_2$	1	–	–	–	
	$\mathbb{F}$	–	7	–	$O(m\log(m))$	$O(m_0+\log(m))$
Marlin [14]	$\mathbb{G}_1$	$3k$	13	$3k$	$14n+8k$	$2P$
	$\mathbb{G}_2$	2	–	–	–	
	$\mathbb{F}$	–	8	–	$O(k\log(k))$	$O(m_0+\log(k))$
LunarLite [13]	$\mathbb{G}_1$	k	10	k	$8n+3k$	$7P$
	$\mathbb{G}_2$	k	–	k	–	
	$\mathbb{F}$	–	2	–	$O(k\log(k))$	$O(m_0+\log(k))$
Basilisk [35]	$\mathbb{G}_1$	n	6	n	$6n$	$2P$
	$\mathbb{G}_2$	1	–	–	–	
	$\mathbb{F}$	–	2	–	$O(n\log(n))$	$O(m_0+\log(n))$
Vampires [31]	$\mathbb{G}_1$	$12n+k$	4	$12n+k$	$20n+2k$	$5\mathbb{G}_1+6P$
	$\mathbb{G}_2$	$4n+k$	–	$4n+k$	–	21
	$\mathbb{F}$	–	2	–	$O(k\log(k))$	$O(m_0+\log(n))$

- $c \leftarrow \mathsf{Com}(\mathsf{ck}, \mathbf{p}, \mathbf{w})$: Given ck, a set of univariate polynomials $\mathbf{p} = [p_i]_{i=1}^n$ over the finite field $\mathbb{F}$ s.t. $\deg(p_i) \leq D$ for all $i \in [n]$ as inputs, it returns commitments $c = [c_i]_{i=1}^n$ to the set of polynomials $\mathbf{p}$. The randomnesses $\mathbf{w} = [w_i]_{i=1}^n$ are used when the commitments c are meant to be hiding.
- $\pi \leftarrow \mathsf{Open}(\mathsf{ck}, \mathbf{p}, z, \zeta; \mathbf{w})$: Given ck, set of univariate polynomials $\mathbf{p} = [p_i]_{i=1}^n$, evaluation point $z \in \mathbb{F}$, and opening challenge ζ as inputs, it outputs an evaluation proof π. Note that the used randomnesses $\mathbf{w}$ should be identical to the ones in the commitment algorithm.
- $(1/0) \leftarrow \mathsf{Vfy}(\mathsf{vk}, \mathbf{c}, z, \mathbf{v}, \pi, \zeta)$: Given the verification key vk, commitments $\mathbf{c} = [c_i]_{i=1}^n$, evaluation point $z \in \mathbb{F}$, supposed evaluations $\mathbf{v} = [v_i]_{i=1}^n$, evaluation proof π, and opening challenge ζ as inputs, it then outputs either 1, if the proof π confirms that, for each $i \in [n]$, the polynomial committed in c_i has degree at most D and evaluates to v_i at z. Otherwise it returns 0.

A (perfectly) hiding PC scheme satisfies the completeness, hiding, and extractability properties defined below.

Definition 5 (Completeness). *For all degree bounds $D \in \mathbb{N}$ and efficient adversary $\mathcal{A}$, a PC, PC, satisfies* Completeness *property, if:*

$$\Pr\left[\begin{matrix}(\mathsf{ck}, \mathsf{vk}) \leftarrow \mathsf{KG}(1^\lambda, D), (\mathbf{p}, z, \zeta, \mathbf{w}) \leftarrow \mathcal{A}(\mathsf{ck}, \mathsf{vk}), \mathbf{v} \leftarrow \mathbf{p}(z), \\ \pi \leftarrow \mathsf{Open}(\mathsf{ck}, \mathbf{p}, z, \zeta; \mathbf{w}) : \mathsf{Vfy}(\mathsf{vk}, \mathbf{c}, z, \mathbf{v}, \pi, \zeta) = 1 \ \wedge \deg(\mathbf{p}) \leq D\end{matrix}\right] = 1 \ .$$

Definition 6 (Extractability). *A PC PC, is called* extractable *if for all upper bound $D \in \mathbb{N}$ and PPT adversaries $\mathcal{A}$, there exists an efficient extractor Ext s.t. for every efficient public-coin challenger $\mathcal{C}$, efficient query sampler $\mathcal{Q}$, PPT adversaries $\mathcal{B} = (\mathcal{B}_1, \mathcal{B}_2)$ and a round bound $q \in \mathbb{N}$,*

$$\Pr\left[\begin{matrix}(\mathsf{ck}, \mathsf{vk}) \leftarrow \mathsf{KG}(1^\lambda, D), \{\rho_i = \mathcal{C}(\mathsf{ck}, \mathsf{vk}, i), \\ \mathbf{c}_i \leftarrow \mathcal{A}(\mathsf{ck}, \mathsf{vk}, [\rho_j]_{j=1}^i), \mathbf{p}_i \leftarrow \mathsf{Ext}(\mathsf{ck}, \mathsf{vk}, [\rho_j]_{j=1}^i), \}_{i \in [q]}, \\ Q \leftarrow \mathcal{Q}(\mathsf{ck}, \mathsf{vk}, [\rho_j]_{j=1}^q), (\mathbf{v}, \mathsf{st}) \leftarrow \mathcal{B}_1(\mathsf{ck}, \mathsf{vk}, [\rho_j]_{j=1}^q, Q), \\ \textit{Sample opening challenge } \zeta, \pi \leftarrow \mathcal{B}_2(\mathsf{st}, \zeta) : \\ \mathsf{Vfy}(\mathsf{vk}, \mathbf{c}, z, \mathbf{v}, \pi, \zeta) = 1 \ \wedge \ \deg(\mathbf{p}) \leq D \ \wedge \ \mathbf{v} = \mathbf{p}(z)\end{matrix}\right] \geq 1 - \mathsf{negl}(\lambda) \ ,$$

where $[c_i]_{i=1}^n := [\mathbf{c}_i]_{i=1}^q$, $[p_i]_{i=1}^n := [\mathbf{p}_i]_{i=1}^q$, $[d_i]_{i=1}^n := [\mathbf{d}_i]_{i=1}^q$. For a Q as the form of $T \times \{z\}$ s.t. $T \subseteq [n]$ and $z \in \mathbb{F}$ we set $\mathbf{c} := [c_i]_{i \in T}$, $\mathbf{p} := [p_i]_{i \in T}$ and $\mathbf{d} := [d_i]_{i \in T}$.

Definition 7 (Simulation-based Hiding). *A PC PC, satisfies (simulation-based) hiding if there exists a polynomial-time simulator $\mathsf{Sim} = (\mathsf{Sim}_{\mathsf{KG}}, \mathsf{Sim}_{\mathsf{Com}}, \mathsf{Sim}_{\mathsf{Open}})$ such that, for every maximum degree bound $D \in \mathbb{N}$, round bound $q \in \mathbb{N}$ and NUPPT adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3)$, we have,*

$$\left|\Pr[\mathsf{Real}(1^\lambda, D, \mathcal{A}) = 1] - \Pr[\mathsf{Ideal}(1^\lambda, D, \mathcal{A}) = 1]\right| \leq \mathsf{negl}(\lambda) \ .$$

$\mathbf{Real}(1^\lambda, D, \mathcal{A})$:	$\mathbf{Ideal}(1^\lambda, D, \mathcal{A})$:
$(\mathsf{ck}, \mathsf{vk}) \leftarrow \mathsf{KG}(1^\lambda, D)$;	$(\mathsf{ck}, \mathsf{vk}, \tau) \leftarrow \mathsf{Sim}_{\mathsf{KG}}(1^\lambda, D)$
Let $c_0 := \perp$; *for* $i \in [q]$:	*Let* $c_0 := \perp$; *for* $i \in [q]$:
- $(\mathbf{p}_i, h_i) \leftarrow \mathcal{A}_1(\mathsf{ck}, \mathsf{vk}, [c_j]_{j=0}^{i-1})$;	- $(\mathbf{p}_i, h_i) \leftarrow \mathcal{A}_1(\mathsf{ck}, \mathsf{vk}, , [c_j]_{j=0}^{i-1})$;
- *if* $h_i = 0$: *sample* $\mathbf{w}_i \leftarrow\!\!\$\ \mathbb{F}$;	- *if* $h_i = 0$: *sample* $\mathbf{w}_i \leftarrow\!\!\$\ \mathbb{F}$; *and* *compute* $c_i \leftarrow \mathsf{Sim}_{\mathsf{Com}}(\tau, \lvert\mathbf{p}_i\rvert; \mathbf{w}_i)$;
- *else: set* $\mathbf{w}_i := \perp$;	- *else: set* $\mathbf{w}_i := \perp$; *and compute* *real* $c_i \leftarrow \mathsf{Com}(\mathsf{ck}, \mathbf{p}_i; \mathbf{w}_i)$;
- $c_i \leftarrow \mathsf{Com}(\mathsf{ck}, \mathbf{p}_i, \mathbf{w}_i)$;	$c := [c_i]_{i=1}^q$, $\mathbf{p} := [\mathbf{p}_i]_{i=1}^q$, $\mathbf{w} := [\mathbf{w}_i]_{i=1}^q$;
$c := [c_i]_{i=1}^q$, $\mathbf{p} := [\mathbf{p}_i]_{i=1}^q$, $\mathbf{w} := [\mathbf{w}_i]_{i=1}^q$;	*Zero out hidden polynomials:* $\mathbf{p}' := [h_i \mathbf{p}_i]_{i=1}^q$;
$\left([Q_j]_{j=1}^t, [\zeta_j]_{j=1}^t, \mathsf{st}\right) \leftarrow \mathcal{A}_2(\mathsf{ck}, \mathsf{vk}, c)$;	$\left([Q_j]_{j=1}^t, [\zeta_j]_{j=1}^t, \mathsf{st}\right) \leftarrow \mathcal{A}_2(\mathsf{ck}, \mathsf{vk}, c)$;
for $i \in [t]$:	*for* $i \in [t]$: $\pi_j \leftarrow \mathsf{Sim}_{\mathsf{open}}(\tau, \mathbf{p}', \mathbf{p}(Q_j), Q_j, \zeta_j; \mathbf{w})$;
$\pi_j \leftarrow \mathsf{open}(\mathsf{ck}, \mathbf{p}, Q_j, \zeta_j; \mathbf{w})$;	*return* $b \leftarrow \mathcal{A}_3(\mathsf{st}, [\pi_j]_{j=1}^t)$.
return $b \leftarrow \mathcal{A}_3(\mathsf{st}, [\pi_j]_{j=1}^t)$.	

B More On Marlin Updatable zk-SNARK

B.1 Marlin's PCs with a Slightly Modified SRS

In [14], authors presented two PC schemes in the GGM and AGM, and then used them to instantiate their general transformation and obtain updatable zk-SNARK. Marlin is obtained by instantiating their general transformation with their AGM-based scheme. However, in Sec. 3.2, we showed that in order to use the AGM-based one, to build a updatable zk-SNARK, one needs to slightly modify its SRS and add a single group element to its SRS. The same issue also exists in their GGM-based PC.

In this section, we describe a new variant of their AGM-based construction that has an additional group element, namely $[x\gamma]_2$, in the SRS, and allows us to prove Sub-ZK and Upd-KS in Marlin. A similar modification needs to be done on their GGM-based PC scheme, while using it in updatable zk-SNARKs. Next, we summarize the modified version of their AGM-based PC scheme, which the only difference is that it has an additional element in the SRS.

– $(\mathsf{ck}, \mathsf{vk}) \leftarrow \mathsf{KG}(1^\lambda, D)$: Given a λ in its unary representation, it runs the bilinear pairing generator $\mathsf{BG} := (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, \hat{e}, [1]_1, [1]_2) \leftarrow \mathsf{BGgen}(1^\lambda)$, and samples the random integers $x, \gamma \leftarrow\!\!\$\ \mathbb{F}_p$ and then computes:

$$\Gamma := \begin{pmatrix} [1]_1 & [x]_1 & [x^2]_1 & \dots & [x^D]_1 \\ [\gamma]_1 & [\gamma x]_1 & [\gamma x^2]_1 & \dots & [\gamma x^D]_1 \end{pmatrix}.$$

It sets $\mathsf{ck} := (\mathsf{BG}, \Gamma)$ and $\mathsf{vk} := \left(D, \mathsf{BG}, [\gamma]_1, [x]_2, [x_0\gamma_0]_2\right)$ and then returns $(\mathsf{ck}, \mathsf{vk})$ as output. Note that the element $[x_0\gamma_0]_2$ is not needed to commit and verify openings, but it is useful to verify ck and also achieve trapdoor extractability in the updatable SRS model.

- $c \leftarrow \mathsf{Com}(\mathsf{ck}, \mathbf{p}, \mathbf{w})$: Given commitment key ck, for any univariate polynomial $p_i \in \mathbf{p}$ for $i \in [n]$ of degree $\deg(p_i) \leq D$ it computes the commitment $c_i := [p_i(x)]_1 + [\gamma \bar{p}_i(x)]_1$. It then returns $c := [c_i]_{i\in[n]}$ as output.
- $\pi \leftarrow \mathsf{Open}(\mathsf{ck}, \mathbf{p}, z, \zeta; \mathbf{w})$: Given ck, a single evaluation point $z \in \mathbb{F}_p$ and opening challenge $\zeta \in \mathbb{F}_p$, it computes the linear combination of polynomials $p(X) := \sum_{i=1}^{n} \zeta^i p_i(X)$ and $\bar{p}(X) := \sum_{i=1}^{n} \zeta^i \bar{p}_i(X)$ and the witness polynomials $w(X) := \frac{p(X)-p(z)}{X-z}$ and $\bar{w}(X) := \frac{\bar{p}(X)-\bar{p}(z)}{X-z}$. It then sets $\mathbf{w} := [w(x)]_1 + [\gamma \bar{w}(x)]_1$ and $\bar{v} := \bar{p}(z)$ and then returns the evaluation proof $\pi := (\mathbf{w}, \bar{v})$ as output.
- $(1/0) \leftarrow \mathsf{Vfy}(\mathsf{vk}, \mathbf{c}, z, \mathbf{v}, \pi, \zeta)$: Given the verification key vk, commitments $\mathbf{c} = [c_i]_{i=1}^{n}$, evaluation point $z \in \mathbb{F}_p$ and opening challenge $\zeta \in \mathbb{F}_p$, it calculates the linear combination $C := \sum_{i=1}^{n} \zeta^i c_i$ and the lineanr combination of evaluation as $v := \sum_{i=1}^{n} \zeta^i v_i$ and check the validty of the following pairing product equation:

$$(C - [v]_1 - [\gamma \bar{v}]_1) \bullet [1]_2 == \mathbf{w} \bullet ([x]_2 - [z]_2) \ .$$

  If it holds, then it accepts the proof; otherwise it rejects the proof π.

In [14, Theorem B.12, Lemmas B.13-B.15], authors showed that under Strong Diffie–Hellman (SDH) assumption, the above PC scheme satisfies hiding, binding, and extractability against algebraic adversaries. By minimal modification to their security proofs it can be shown that the modified scheme still achieves hiding, binding, and extractability in the AGM.

B.2 On the Setup of Marlin with the Shortest Proofs

In [14], authors presented two PC schemes in the GGM and AGM, where the AGM-based one is more efficient. In Sec. 3.2, we described the $\mathsf{SG}, \mathsf{SU}, \mathsf{SV}$ and BSV algorithms for Marlin, once it is instantiated with a minimally modified version of the AGM-based one. However, in practice, to obtain shorter proofs, in the latest version of Marlin's implementation [4], authors use a different PC scheme, that can be considered as a variant of the scheme proposed in [32], which is optimized in [20]. Next, in Fig. 16, we present the SG, and SU algorithms for the mentioned PC, which we use in our implementations and comparisons [5]. Note that similar to the PC schemes proposed in Marlin, the SRS of the implemented PC scheme was not trapdoor extractable, and similar to previous cases we add $[\gamma x]_2$ to the SRS, which allows to extract the simulation trapdoors. Also in Fig. 17, we describe the SV and BSV algorithms corresponding to the SG and SU algorithms described in Fig. 16.

[4] Available on `https://github.com/arkworks-rs/marlin`.

[5] Specifically, the details about the SRS elements of this variant are provided with Pratyush Mishra, a co-author of Marlin, in a private communication.

SRS Generation, $(\mathsf{srs}_0, \Pi_0) \leftarrow \mathsf{SG}(\mathbf{R})$**:** Given $\mathbf{R}$, first deduce the security parameter 1^λ and obtain $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, \hat{e}, [1]_1, [1]_2) \leftarrow \mathsf{BGgen}(1^\lambda)$; then act as follows:

- Sample $\bar{x}_0, \bar{\gamma}_0 \leftarrow \mathbb{Z}_p^\star$, and set $x_0 := \bar{x}_0$, and $\gamma_0 := \bar{\gamma}_0$ which are the trapdoors of srs_0;
- For $k = 0, \cdots, n$: compute $\left[x_0^k\right]_1$;
- For $k = 0, \cdots, 5$: compute $\left[\gamma_0 x_0^k\right]_1$;
- For $k = 1, 2, 4, 8, \cdots, \log n$: compute $\left[x_0^{-k}\right]_2$;
- Compute $[x_0]_2$, and $[x_0\gamma_0]_2$;
- Set $\mathsf{srs}_0 := \left(\left(\left[x_0^k\right]_1\right)_{k=0}^{n}, \left(\left[\gamma_0 x_0^k\right]_1\right)_{k=0}^{5}, \left[x_0^{-l}\right]_2, [x_0]_2, [x_0\gamma_0]_2\right)$, for $l = 1, 2, 4, \cdots, \log n$, and the well-formedness proof $\Pi_0 := (\Pi_0^{\mathsf{Agg}}, \Pi_0^{\mathsf{Ind}}) := (([\gamma_0]_1, [x_0\gamma_0]_1, [x_0]_2), ([\bar{x}_0]_1, [\bar{\gamma}_0]_1, [\bar{x}_0]_2, [\bar{x}_0\bar{\gamma}_0]_2))$;
- Return (srs_0, Π_0);

SRS Update, $(\mathsf{srs}_i, \Pi_i) \leftarrow \mathsf{SU}(\mathsf{srs}_{i-1}, \{\Pi_{j-1}\}_{j=0}^{i-1})$**:** Given $(\mathsf{srs}_{i-1}, \{\Pi_{j-1}\}_{j=0}^{i-1})$,

- Parse $\mathsf{srs}_{i-1} := \left(\left(\left[x_{i-1}^k\right]_1\right)_{k=0}^{n}, \left(\left[\gamma_{i-1}x_{i-1}^k\right]_1\right)_{k=0}^{5}, \left[x_{i-1}^{-l}\right]_2, [x_{i-1}]_2, [x_{i-1}\gamma_{i-1}]_2\right)$, for $l = 1, 2, 4, \cdots, \log n$;
- Sample $\bar{x}_i, \bar{\gamma}_i \leftarrow \mathbb{Z}_p^\star$ as the secret shares to use for updating srs_{i-1}.
- For $k = 0, \cdots, n$: set $\left[x_i^k\right]_1 := \bar{x}_i^k \cdot \left[x_{i-1}^k\right]_1$, $\left[\gamma_i x_i^k\right]_1 := \bar{\gamma}_i \bar{x}_i^k \cdot \left[\gamma_{i-1}x_{i-1}^k\right]_1$;
- For $k = 0, \cdots, 5$: set $\left[\gamma_i x_i^k\right]_1 := \bar{\gamma}_i \bar{x}_i^k \cdot \left[\gamma_{i-1}x_{i-1}^k\right]_1$;
- For $k = 1, 2, 4, \cdots, \log n$: set $\left[x_i^{-k}\right]_2 := \bar{x}_i^{-k} \cdot \left[x_{i-1}^{-k}\right]_2$;
- set $[x_i]_2 := \bar{x}_i \cdot [x_{i-1}]_2$ and $[x_i\gamma_i]_2 := \bar{x}_i\bar{\gamma}_i \cdot [x_{i-1}\gamma_{i-1}]_2$;
- Set $\mathsf{srs}_i := \left(\left(\left[x_i^k\right]_1\right)_{k=0}^{n}, \left(\left[\gamma_i x_i^k\right]_1\right)_{k=0}^{5}, \left[x_i^{-l}\right]_2, [x_i]_2, [x_i\gamma_i]_2\right)$, for $l = 1, 2, 4, \cdots, \log n$, and the well-formedness proof $\Pi_i := (\Pi_i^{\mathsf{Agg}}, \Pi_i^{\mathsf{Ind}}) := (([\gamma_i]_1, [x_i\gamma_i]_1, [x_i]_2), ([\bar{x}_i]_1, [\bar{\gamma}_i]_1, [\bar{x}_i]_2, [\bar{x}_i\bar{\gamma}_i]_2))$;
- Return (srs_i, Π_i);

Fig. 16: SG and SU algorithms for the implemented variant of Marlin.

On the Efficiency of SU*,* SV*, and* BSV *Algorithms in Marlin:* Using the SU algorithm described in Fig. 16, similar to the SG algorithm, in order to update the SRS of size n in Marlin, one needs to compute $\log\log n$ exponentiations in $\mathbb{G}_2$ and $n+6$ exponentiations in $\mathbb{G}_1$. Using the SV algorithm described in Fig 17, to verify an i-time updated SRS, $i \geq 1$, a prover needs to compute $2n+\log\log n+12$ pairing operations (importantly, independent of the value of i), while a verifier needs to compute $2n + \log\log n + 9i + 16$ pairings. To verify an i-time updated SRS of size n more efficiently, using the BSV algorithm (given in Fig. 17),

- a prover will compute 4 pairings, $\log\log n$ exponentiations in $\mathbb{G}_2$, and $2n + \log\log n + 10$ exponentiations in $\mathbb{G}_1$,
- and a verifier would need to compute $2i + 9$ pairings, $2i + \log\log n$ exponentiations in $\mathbb{G}_2$, and $2n + \log\log n + 5i + 10$ exponentiations in $\mathbb{G}_1$,

which in practice is considerably faster than the standard SV algorithm.

SRS Verify, $(\bot/1) \leftarrow \mathsf{SV}(\mathsf{srs}_i, (\Pi_j)_{j=0}^{i}, \mathsf{party})$**:** To verify (an i-time updated) $\mathsf{srs}_i := \Big(\big([x_i^k]_1\big)_{k=0}^{n}, \big([\gamma_i x_i^k]_1\big)_{k=0}^{5}, [x_i^{-l}]_2, [x_i]_2, [x_i\gamma_i]_2\Big)$, for $l = 1, 2, 4, \cdots, \log n$, and $\Pi_j := (\Pi_j^{\mathsf{Agg}}, \Pi_j^{\mathsf{Ind}}) := \big(\big([\gamma_j]_1, [x_j\gamma_j]_1, [x_j]_2\big), \big([\bar{x}_j]_1, [\bar{\gamma}_j]_1, [\bar{x}_j]_2, [\bar{x}_j\bar{\gamma}_j]_2\big)\big)$; for $j = 0, 1, \cdots, i$:
If $\mathsf{party} = \mathsf{P}$:
1. For $k = 1, \cdots, n$: check if $[x_i^k]_1 \bullet [1]_2 = [x_i^{k-1}]_1 \bullet [x_i]_2$;
2. For $k = 1, \cdots, 5$: check if $[\gamma_i x_i^k]_1 \bullet [1]_2 = [\gamma_i x_i^{k-1}]_1 \bullet [x_i]_2$;
3. For $k = 1, 2, 4, \cdots, \log n$: check if $\Big[x_i^{(\log n)-k}\Big]_1 \bullet [1]_2 = \Big[x_i^{\log n}\Big]_1 \bullet [x_i^{-k}]_2$;
4. Check if $[x_i\gamma_i]_1 \bullet [1]_2 = [1]_1 \bullet [\gamma_i x_i]_2$;

If $\mathsf{party} = \mathsf{V}$:
- If $i = 0$: srs_0 is sampled by V, and it does not need to be verified.
- If $i \geq 1$:
  1. Check if $[\gamma_0]_1 = [\bar{\gamma}_0]_1$ and $[x_0]_2 = [\bar{x}_0]_2$;
  2. For $j = 0, 1, \cdots, i$: check if $[\bar{x}_j]_1 \bullet [1]_2 = [1]_1 \bullet [\bar{x}_j]_2$ and $[1]_1 \bullet [\bar{x}_j\bar{\gamma}_j]_2 = [\bar{\gamma}_j]_1 \bullet [\bar{x}_j]_2$.
  3. For $j = 1, 2, \cdots, i$: check if $[1]_1 \bullet [x_j]_2 = [\bar{x}_j]_1 \bullet [x_{j-1}]_2$, $[x_j\gamma_j]_1 \bullet [1]_2 = [x_{j-1}\gamma_{j-1}]_1 \bullet [\bar{x}_j\bar{\gamma}_j]_2 = [\gamma_j]_1 \bullet [x_j]_2$;
  4. For $k = 1, \cdots, n$: check if $[x_i^k]_1 \bullet [1]_2 = [x_i^{k-1}]_1 \bullet [x_i]_2$;
  5. For $k = 1, \cdots, 5$: check if $[\gamma_i x_i^k]_1 \bullet [1]_2 = [\gamma_i x_i^{k-1}]_1 \bullet [x_i]_2$;
  6. For $k = 1, 2, 4, \cdots, \log n$: check if $\Big[x_i^{(\log n)-k}\Big]_1 \bullet [1]_2 = \Big[x_i^{\log n}\Big]_1 \bullet [x_i^{-k}]_2$;
  7. Check if $[x_i\gamma_i]_1 \bullet [1]_2 = [1]_1 \bullet [\gamma_i x_i]_2$;

Return 1 if all the checks passed, otherwise return $\bot$.

Batched SRS Verification, $(\bot/1) \leftarrow \mathsf{SV}(\mathsf{srs}_i, (\Pi_j)_{j=0}^{i}, \mathsf{party})$**:** To verify (an i-time updated) $\mathsf{srs}_i := \Big(\big([x_i^k]_1\big)_{k=0}^{n}, \big([\gamma_i x_i^k]_1\big)_{k=0}^{5}, [x_i^{-l}]_2, [x_i]_2, [x_i\gamma_i]_2\Big)$, for $l = 1, 2, 4, \cdots, \log n$, and $\Pi_j := (\Pi_j^{\mathsf{Agg}}, \Pi_j^{\mathsf{Ind}}) := (([\gamma_j]_1, [x_j\gamma_j]_1, [x_j]_2), ([\bar{x}_j]_1, [\bar{\gamma}_j]_1, [\bar{x}_j]_2, [\bar{x}_j\bar{\gamma}_j]_2))$; for $j = 0, 1, \cdots, i$:
If $\mathsf{party} = \mathsf{P}$:
1. Sample $\big\{t_{1,k} \leftarrow \mathbb{Z}_p^\star\big\}_{k=1}^{n}$, $\big\{t_{2,k} \leftarrow \mathbb{Z}_p^\star\big\}_{k=1}^{5}$, $\big\{t_{3,k} \leftarrow \mathbb{Z}_p^\star\big\}_{k=1,2,4,\cdots \log n}$;
2. Check if $([\gamma_i x_i]_1 + \sum_{k=1}^{n}(t_{1,k} \cdot [x_i^k]_1) + \sum_{k=1}^{5}(t_{2,k} \cdot [\gamma_i x_i^k]_1) + \sum_{k=1}^{\log n}(t_{3,k} \cdot \Big[x_i^{(\log n)-k}\Big]_1)) \bullet [1]_2 = (\sum_{k=1}^{n}(t_{1,k} \cdot [x_i^{k-1}]_1) + \sum_{k=1}^{5}(t_{2,k} \cdot [\gamma_i x_i^{k-1}]_1)) \bullet [x_i]_2 + \Big[x_i^{\log n}\Big]_1 \bullet (\sum_{k=1,2,4,\cdots,\log n}(t_{3,k} \cdot [x_i^{-k}]_2)) + [1]_1 \bullet [\gamma_i x_i]_2$;

If $\mathsf{party} = \mathsf{V}$:
- If $i = 0$: srs_0 is sampled by $\mathsf{verifier}$, and it does not need to be verified.
- If $i \geq 1$, act as follows,
  1. Sample $\{r_{1,j}, r_{2,j}, r_{3,j}, r_{4,j} \leftarrow \mathbb{Z}_p^\star\}_{j=0}^{i}$; and $\{t_{1,k} \leftarrow \mathbb{Z}_p^\star\}_{k=1}^{n}$, $\{t_{2,k} \leftarrow \mathbb{Z}_p^\star\}_{k=1}^{5}$, $\{t_{3,k} \leftarrow \mathbb{Z}_p^\star\}_{k=1,2,4,\cdots,\log n}$;
  2. Check if $[\gamma_0]_1 = [\bar{\gamma}_0]_1$ and $[x_0]_2 = [\bar{x}_0]_2$;
  3. Check if $(\sum_{j=0}^{i} r_{1,j} \cdot [\bar{x}_j]_1) \bullet [1]_2 = [1]_1 \bullet (\sum_{j=0}^{i} r_{1,j} [\bar{x}_j]_2)$;
  4. Check if $[1]_1 \bullet (\sum_{j=0}^{i} r_{2,j} \cdot [\bar{\gamma}_j\bar{x}_j]_2) = \sum_{j=0}^{i}(r_{2,j} [\bar{\gamma}_j]_1 \bullet [\bar{x}_j]_2)$;
  5. Check if $[1]_1 \bullet (\sum_{j=1}^{i} r_{3,j} \cdot [x_j]_2) = \sum_{j=1}^{i}(r_{3,j} \cdot [\bar{x}_j]_1 \bullet [x_{j-1}]_2)$;
  6. Check if $(\sum_{j=1}^{i} r_{4,j} \cdot [\gamma_j x_j]_1) \bullet [1]_2 = \sum_{j=1}^{i}(r_{4,j} \cdot [\gamma_{j-1}x_{j-1}]_1 \bullet [\bar{\gamma}_j\bar{x}_j]_2) = \sum_{j=1}^{i}(r_{4,j} \cdot [\gamma_j]_1 \bullet [x_j]_2)$;
  7. Check if $([\gamma_i x_i]_1 + \sum_{k=1}^{n}(t_{1,k} \cdot [x_i^k]_1) + \sum_{k=1}^{5}(t_{2,k} \cdot [\gamma_i x_i^k]_1) + \sum_{k=1}^{\log n}(t_{3,k} \cdot \Big[x_i^{(\log n)-k}\Big]_1)) \bullet [1]_2 = (\sum_{k=1}^{n}(t_{1,k} \cdot [x_i^{k-1}]_1) + \sum_{k=1}^{5}(t_{2,k} \cdot [\gamma_i x_i^{k-1}]_1)) \bullet [x_i]_2 + \Big[x_i^{\log n}\Big]_1 \bullet (\sum_{k=1,2,4,\cdots,\log n}(t_{3,k} \cdot [x_i^{-k}]_2)) + [1]_1 \bullet [\gamma_i x_i]_2$;

Return 1 if all the checks passed, otherwise return $\bot$.

Fig. 17: SV and an BSV algorithms for Marlin with the shortest proofs.