Rate-1 Non-Interactive Arguments for Batch-NP and Applications *

Lalita Devadas
MIT

Rishab Goyal
University of Wisconsin-Madison

Yael Kalai
Microsoft Research and MIT

Vinod Vaikuntanathan
MIT

Abstract

We present a *rate*-1 construction of a publicly verifiable non-interactive argument system for batch-NP (also called a BARG), under the LWE assumption. Namely, a proof corresponding to a batch of k NP statements each with an m-bit witness, has size $m + \mathsf{poly}(\lambda, \log k)$.

In contrast, prior work either relied on non-standard knowledge assumptions, or produced proofs of size $m \cdot \mathsf{poly}(\lambda, \log k)$ (Choudhuri, Jain, and Jin, STOC 2021, following Kalai, Paneth, and Yang 2019).

We show how to use our rate-1 BARG scheme to obtain the following results, all under the LWE assumption:

- A multi-hop BARG scheme for NP.
- A multi-hop aggregate signature scheme (in the standard model).
- An incrementally verifiable computation (IVC) scheme for arbitrary T-time deterministic computations with proof size $\mathsf{poly}(\lambda, \log T)$.

Prior to this work, multi-hop BARGs were only known under non-standard knowledge assumptions or in the random oracle model; aggregate signatures were only known under indistinguishability obfuscation (and RSA) or in the random oracle model; IVC schemes with proofs of size $\mathsf{poly}(\lambda, T^{\epsilon})$ were known under a bilinear map assumption, and with proofs of size $\mathsf{poly}(\lambda, \log T)$ under non-standard knowledge assumptions or in the random oracle model.

*Devadas, Goyal and Vaikuntanathan were supported in part by DARPA under Agreement No. HR00112020023, a grant from the MIT-IBM Watson AI, a grant from Analog Devices, a Microsoft Trustworthy AI grant, and a Thornton Family Faculty Research Innovation Fellowship from MIT. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Government or DARPA. The work of Goyal was done when he was at MIT.

Contents

1 Introduction

Succinct non-interactive arguments (SNARGs) and arguments of knowledge (SNARKs) are not only objects of great importance in the theory of cryptographic proofs, but are also revolutionizing practical applications such as blockchains and cryptocurrencies [But21]. A vigorous and productive line of research has resulted in several constructions of SNARGs and SNARKs for NP (e.g., [Mic94, Gro10, Lip12, BCCT12, GGPR13, BCCT13, BCI+13], to name a few). However, despite decades of research, all constructions of SNARGs for general NP languages have relied on either the random oracle model or on non-falsifiable knowledge assumptions. While there are some limited negative results on constructing SNARGs in the standard model under falsifiable cryptographic assumptions [GW11], the possibility of achieving them remains tantalizing, indeed a holy grail of this line of research.

Meanwhile, a steady stream of results, starting from [KRR13, KRR14] has showed us how to achieve SNARGs in the standard model for increasingly more expressive classes of computations. The initial results in this direction compiled multi-prover interactive proofs into *privately verifiable* SNARGs [KRR13, KRR14, BHK17, BKK+18], resulting in privately verifiable SNARGs for P (and even for NTISP, the class of non-deterministic time and space bounded computations). More recent incarnations [KPY19, KPY20] of this line of work have resulted in *publicly verifiable* SNARGs for P (and NTISP) using hardness assumptions on groups that support bilinear maps. An alternate, seemingly completely different, route to publicly verifiable SNARGs has proceeded by the round-compression of interactive proofs via the Fiat-Shamir paradigm [FS87, CCH+19, JKKZ21]. Following this line of research, the recent breakthrough work of Choudhuri, Jain and Jin [CJJ21a] constructed a SNARG for P, under the learning with errors (LWE) assumption; and [KVZ21] achieved a SNARG for NTISP from the LWE assumption. The key ingredient in their constructions is the notion of a SNARG for batch-NP computations, also referred to as a batch argument or BARG.

BARGs are the central theme in our work. They have recently emerged as a powerful tool for constructing expressive SNARGs for various classes of computations. A BARG is a proof system for k NP statements where the size of the proof (resp. the verification time) is proportional to the size of a *single instance-witness pair* (resp. the time for a single NP verification). Indeed, [KVZ21, CJJ21a] showed a "BARG-to-SNARG compiler" that uses any BARG to construct a SNARG for P and one for NTISP. Together with the construction of BARGs from LWE [CJJ21a], we have SNARGs for P and NTISP from the LWE assumption. Together with the construction of BARGs from bilinear maps [KPY19, WW22, KLVW22], and from a combination of the decisional Diffie-Hellman (DDH) and quadratic residuosity (QR) assumptions [CJJ21b, KLVW22], the aforementioned BARG-to-SNARG compiler [CJJ21a, KVZ21] gives us SNARGs for P and NTISP from the corresponding assumptions. Batch proofs have also been studied in the statistical soundness setting; see, e.g., [RRR18, RR20]. In summary, the study of batch proofs and arguments shed new light on the long-running quest to construct succinct non-interactive arguments for NP.

Our work. In this paper, we advance the study of batch arguments for NP in several directions.

As our main technical contribution, we construct the first *rate*-1 BARG in the standard model where the size of the proof for k NP statements is $m + \mathsf{poly}(\lambda, \log k)$, where m is the length of a single witness and λ is the security parameter. That is, the ratio between the length of the proof to the length of a single witness is asymptotically 1. In all prior work [CJJ21a, CJJ21b, KVZ21, WW22], the BARG proof had a multiplicative overhead in the security parameter (whereas our overhead is additive). Short of attaining the holy grail of constructing a full-fledged SNARG for NP (with proof

length sublinear in the witness length), this is the optimal size one can achieve for BARGs. The soundness of our construction relies on the learning with errors (LWE) assumption.

Our BARG scheme has the following two additional properties (similar to the one from [CJJ21a]). First, it is *somewhere extractable* (we call such schemes seBARG) in the sense that given an appropriate trapdoor, one can extract a single witness from the BARG proof. In addition, it is an *index* seBARG, which means that if the k instances can be generated by a size-s circuit that on input i outputs the i-th instance (where s may be significantly smaller than k) then the verification time grows with s (and is otherwise independent of k).[1]

On the way to our construction, we define and construct a *fully local somewhere extractable hash function* (flSEH) family, a significant generalization of the notion of somewhere statistically binding (SSB) hashing defined by Hubáček and Wichs [HW15]. In SSB hashing, the hash function description specifies a (hidden) set of m input locations where the hash output is statistically binding. It is not hard to see that this implies the hash output is at least m bits long. Indeed, in all prior constructions [HW15, OPWW15a], the hash output had length $m \cdot \mathsf{poly}(\lambda)$. Our construction improves on this in several ways. First, our construction is *rate*-1, in the sense that the hash output consists of $m + \mathsf{poly}(\lambda)$ bits. Secondly, it is *extractable* in the sense that given a trapdoor associated to the hash function, one can extract the m statistically bound bits efficiently[2]; and finally, it is *fully locally openable* in the sense that the local opening of each bit has size $\mathsf{poly}(\lambda)$, *independent of* m. In contrast, the Hubáček-Wichs [HW15] notion of SSB hashing required a weaker version of local opening: the size of the local opening for a single bit could grow polynomially with m. A minute of thought reveals that this requirement is highly non-trivial: the local opening needs to be shorter than the hash function output which, as we observed above, grows with m. We view this primitive and the construction thereof to be as important as the result itself, and elaborate on it in Sections 1.2 and 2.3. Indeed, this primitive has already been used in [KLVW22] to compile any semi-succinct BARG scheme (where the BARG proof of k NP statements has size sublinear in k) into a (succinct) BARG scheme.

We show a number of applications of our main result. We show a construction of multi-hop seBARGs from LWE; a construction of multi-hop aggregate signatures from LWE (in the standard model); and a construction of incrementally verifiable computation for all deterministic computations from LWE. Previously, such results were known only in idealized models or under non-standard knowledge assumptions, with the exception of aggregate signatures which was known assuming indistinguishability obfuscation (iO) and the RSA assumption [HKW15].[3] We describe our contributions in more detail below.

1.1 Our Main Technical Result: Rate-1 seBARG for NP

Our main technical contribution is the construction of a *rate*-1 seBARG scheme. A somewhere extractable BARG scheme (seBARG) consists of a setup algorithm Gen; a prover algorithm $\mathcal{P}$; and a verifier algorithm $\mathcal{V}$. The setup algorithm Gen is given a security parameter 1^λ, a parameter k (which specifies the number of instances in a batch NP statement), an instance size n, and an index $i \in [k]$ (which specifies which witness should be extractable). It generates a common reference string

[1] From now on, when we refer to an seBARG, we always mean that it is an index seBARG.

[2] In fact, our construction turns out to have the stronger property of *local extraction*: that is, the time to extract a single bit is $\mathsf{poly}(\lambda)$, independent of m. However, we do not need this property for our constructions, and hence do not pursue it any further.

[3] We note that the scheme presented in [HKW15] is only selectively secure, whereas our scheme is adaptively secure.

crs and a trapdoor td. The prover algorithm $\mathcal{P}$ is given the crs and a set of instances $x_1, \ldots, x_k \in \mathcal{L}$ together with a set of corresponding witnesses $w_1, \ldots, w_k$, and generates a proof π. Finally, the verification algorithm $\mathcal{V}$ is given a crs, a set of instances $x_1, \ldots, x_k$, and a proof π, and outputs $0/1$ (indicating accept or reject).

A rate-1 seBARG scheme has the following properties (in addition to the usual notion of completeness).

1. *Semi-adaptive soundness.* For any index $i \in [k]$ and $\mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda, k, n, i)$, no probabilistic polynomial-time adversary can generate a tuple of k NP statements $x_1, \ldots, x_k \in \{0,1\}^n$ together with a proof π^*, where $x_i \notin \mathcal{L}$ and yet, the verifier accepts π^*. *Semi*-adaptivity here refers to the fact that the adversary picks i before seeing the common reference string.

2. *Somewhere proof-of-knowledge* is a strengthening of semi-adaptive soundness which requires an probabilistic polynomial-time extractor algorithm that, given a trapdoor for the crs, k NP statements, and an accepting proof π^*, extracts a witness w_i for the statement x_i.

3. *Index hiding.* The common reference string crs should hide the index $i \in [k]$ (which specifies which witness is extractable).

4. *Efficiency.* The algorithms $\mathsf{Gen}, \mathcal{P}$ and $\mathcal{V}$ are all probabilistic polynomial-time, and the proof π has size $m + \mathsf{poly}(\lambda, \log k) = m + \mathsf{poly}(\lambda)$. Moreover, if the instances $x_1, \ldots, x_k$ can be efficiently generated by a size-s circuit that on input i outputs x_i, then the run-time of $\mathcal{V}$ is $\mathsf{poly}(s, m, \lambda)$, as opposed to the potentially much larger $\mathsf{poly}(n, k, m, \lambda)$.

We note that, as remarked before, all known constructions of BARGs and seBARGs [CJJ21a, CJJ21b, WW22] have an inverse polynomial (in λ) rate. Our main result is the following.

Theorem 1.1 (Informal)**.** *Under the* LWE *assumption, for every* $\mathcal{L} \in \mathsf{NP}$*, there exists a rate-*1 seBARG *scheme* $(\mathsf{Gen}, \mathcal{P}, \mathcal{V})$*.*

We remark that our construction is more general: given a SNARG for an NP language with proofs of size $\ell(m)$ for statements with m-bit long witnesses, our construction produces an seBARG with proofs of size $\ell(m) + \mathsf{poly}(\lambda)$ for $k \leq 2^\lambda$ statements. Special-purpose NP languages with such non-trivial SNARGs exist, e.g. for the non-deterministic class NTISP. Our construction then "lifts" them into somewhere extractable batch arguments with similarly short proofs. We also note that short of constructing a non-trivial SNARG for NP, rate-1 seBARGs are the best one can do. We refer the reader to Section 3.2 for details.

1.2 Our Main Tool: Fully Local Somewhere Extractable Hash (SEH) Families

The main technical tool that we define, construct and use is that of a fully local somewhere extractable hash (flSEH) function.

Collision-resistant hash functions compress their input, therefore by definition lose information about it. Hubáček and Wichs [HW15] ask if there is a hash function that is nevertheless guaranteed to preserve *some* information about its input. For example, letting the input x be an n-bit string, we may wish to design a hash function h_I so that $h_I(x)$ remembers $x_I = (x_i)_{i \in I}$ for some subset $I \subseteq [n]$. That is, for every x, x' such that $x_I \neq x'_I$, $h_I(x) \neq h_I(x')$. In this sense, the hash function h_I is statistically binding on the locations $I \subseteq [n]$, and hence such a hash family is called a *somewhere*

statistically binding (SSB) hash function. It is not hard to see that for this to happen, the hash output must have size at least $|I|$. As stated, however, this is trivial to build: h_I could simply output x_I. To be non-trivial (and indeed, useful), an SSB hash function needs to have an additional *hiding* property, namely the descriptions of the hash functions h_I and $h_{I'}$ should be computationally indistinguishable whenever $|I| = |I'|$. Hubáček and Wichs [HW15] show how to construct an SSB hash function family using a leveled fully homomorphic encryption (FHE) scheme, thus relying on the LWE assumption. Subsequent work showed how to realize SSB hash functions from a wider class of assumptions [OPWW15a].

Our notion of *fully local somewhere extractable hash function* (flSEH) family is a significant generalization SSB hashing: it is *rate*-1, *extractable*, and *fully locally openable*.

For a reader familiar with the Hubáček-Wichs construction, it is not hard to see that their construction is *somewhere extractable*, that is, given a trapdoor associated with the hash function, one can extract the statistically bound value. In the Hubáček-Wichs construction, the trapdoor is simply the secret key of the FHE scheme. Their construction also has a *local opening* property, that is, there is an opening algorithm that, given (x, i) and a description of the hash function, outputs x_i together with a short "certificate" that certifies the correctness of x_i. The Hubáček-Wichs construction is a tree-based hash function, similar to a Merkle hash [Mer88]. Thus, an opening consists of all the hash values on the path in the tree from the leaf node i to the root, together with the hash values of all their siblings. The fact that each hash value is of size at least $|I|$ implies that the size of the opening is at least $|I|$. We refer to a hash family that satisfies these properties as an SEH family. Finally, the Hubáček-Wichs construction achieves *rate*-1, that is, the size of the hash output is $|I| + \mathsf{poly}(\lambda)$, if one uses a rate-1 FHE scheme such as the one from [BDGM19, GH19a, DGI+19a].

A *fully local* SEH family is a rate-1 SEH family with the crucial additional property of full locality. That is, the size of a local opening does not grow with $|I|$, the number of bits that we are statistically bound on. In other words, one should be able to open any bit of the input using an opening of size $\mathsf{poly}(\lambda)$ bits, independent of $|I|$. Additionally, verification of this opening should be possible in time $\mathsf{poly}(\lambda)$ as well. In other words, verification should take time smaller than the hash value itself! A minute of thought reveals that one has to significantly depart from the Merkle tree paradigm to construct such a hash function. Indeed, the main novelty and technical contribution of our construction is in building such a seemingly paradoxical hash function with *fully* local opening and verification algorithms. We refer the reader to Section 2.3 for details.

Theorem 1.2 (Informal)**.** *Under the* LWE *assumption, there exists a* rate-1 fully local SEH *family.*

1.3 Applications of seBARGs

We show two ways to strengthen seBARGs, resulting in the new notions of *multi-hop* seBARG and *hashed* (multi-hop) seBARG. We then show how to use rate-1 seBARGs to achieve these stronger notions. Finally, we show two applications of these stronger notions, the first to constructing a multi-hop aggregate signature scheme in the standard model, and the second to constructing an incrementally verifiable computation scheme in the standard model. All schemes are secure under the LWE assumption. We describe these contributions in more detail below.

Multi-Hop seBARGs. The ability to compose proofs enables mutually distrustful parties to perform distributed computations in a verifiable manner, and is especially important in decentralized applications such as blockchains. The importance of proof composition was in fact already realized

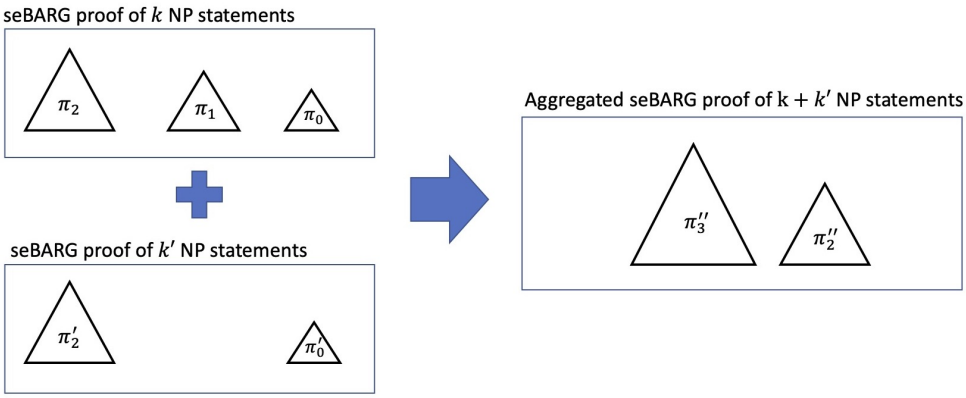


Figure 1: Each proof π_i (resp. π'_i) is an aggregation of 2^i many NP statements. A proof of k NP statements is thus a collection of seBARG proofs $(\pi_0, \ldots, \pi_{\log k})$ where each π_i is a proof of 2^i statements (if the i^{th} digit in the binary expansion of k is 1) or empty (if the i^{th} digit is 0). To aggregate two such proofs (for $k = 7$ and $k = 5$ NP statements respectively, as illustrated above on the left), we start by combining π_0 and π'_0 into a proof $\bar{\pi}_1$ that certifies the truth of two NP statements. We then aggregate π_1 and $\bar{\pi}_1$ into a proof π''_2 which certifies the truth of four NP statements. Finally, we aggregate π_2 and π'_2 into a proof π''_3 which certifies the truth of eight NP statements. The final proof consists of (π''_2, π''_3). Each proof π_i is a result of at most $i \leq \log k$ hops of aggregation.

in Micali's original work introducing SNARGs [Mic93], and has been extensively studied in the last two decades [Val08, CT10, BCCT13, BCTV14, COS20, KPY20, BCMS20, BCL+21], leading to powerful primitives such as Incrementally Verifiable Computation (IVC) [Val08] and Proof-Carrying Data (PCD) [CT10], and to many applications, such as enforcing language semantics [CTV13], verifiable MapReduce computations [CTV15], image authentication [NT16], PPAD hardness [Val08, BPR15, KPY20], and succinct blockchains [KB20, BMRS20, CCDW20].

We ask whether it is possible to compose BARG proofs. That is, given BARG proofs $\pi_1, \ldots, \pi_k$ of k *batches* of NP statements $\mathbf{x}_1 = (x_{1,1}, \ldots, x_{1,\ell_1}), \ldots, \mathbf{x}_k = (x_{k,1}, \ldots, x_{1,\ell_k})$, can we create a BARG proof π for the collection of these NP statements, namely that all of the NP statements

$$\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_k) = (x_{1,1}, \ldots, x_{1,\ell_1}, \ldots, x_{k,1}, \ldots, x_{k,\ell_k})$$

are true? Furthermore, the system should support an unbounded polynomial number of such iterative compositions without the proof size or the verification time growing by too much. We call such a system a multi-hop batch argument for NP, or a multi-hop BARG for short. If the system supports somewhere extraction, appropriately defined, it is called a multi-hop seBARG.

It is natural to try to realize a multi-hop BARG by BARGing many BARG proofs. In the example above, one could try to produce a BARG proof that there exist BARG proofs $\boldsymbol{\pi}_1, \ldots, \boldsymbol{\pi}_k$ certifying the truth of the collections of NP statements $\mathbf{x}_1, \ldots, \mathbf{x}_k$ respectively. The first issue with this approach is that each composition increases the length of the proofs. For example, if we use a BARG system where the proof length has a multiplicative overhead of $\mathsf{poly}(\lambda)$ (over the witness length) as is the case for all known BARG schemes [KPY19, CJJ21a, CJJ21b, WW22], then the size of the proofs after B hops grows to $\lambda^{\Omega(B)}$. By playing with the security parameter, one can make the number of hops slightly super-constant (see, e.g., [KPY19, KPY20]), but that hits the limit of what is possible.

Even if we had a constant-rate BARG (with rate > 1), we could handle at most a logarithmic (in the security parameter) number of hops. The second issue is one of proving soundness: the most direct way of proving soundness of the composed proof system is through extraction.

Rate-1 somewhere extractable BARGs (seBARGs) solve both these problems, giving us the following theorem.

Theorem 1.3 (Informal)**.** *Under the* LWE *assumption, there exists a multi-hop* seBARG *for* NP *statements with m-bit witnesses where the size of the proof after B hops is $m + B \cdot \mathsf{poly}(\lambda)$.*

Our proof of this theorem proceeds by showing a way to convert *any* rate-1 seBARG scheme into a multi-hop seBARG. We obtain the theorem by using our rate-1 seBARG from Theorem 1.1 which we construct under LWE. Each hop incurs an *additive* increase of $\mathsf{poly}(\lambda)$ to the proof length, and hence the total size after B hops is a modest $m + B \cdot \mathsf{poly}(\lambda)$ bits. Our construction does not require the batch size or the instance length to be fixed in advance, relying on the fact that the run-time of our rate-1 seBARG setup algorithm grows only poly-logarithmically with the batch size and the instance length.

Finally, we observe that while the dependence on the number of hops may seem limiting, a simple extension of our construction allows us to batch an *arbitrary* polynomial (and even more, upto an exponential) number of NP proofs. To do so, we maintain the invariant that the batched proof at any point corresponding to k NP statements consists of a sequence of seBARG proofs $(\pi_0, \pi_1, \ldots, \pi_\ell)$ with $\ell \leq \lfloor \log k \rfloor$, where π_i is either empty or a batched proof corresponding to exactly 2^i NP statements. It is not hard to see that two such batched proofs $(\pi_0, \ldots, \pi_\ell)$ and $(\pi'_0, \ldots, \pi'_\ell)$ can be combined together into a proof $(\pi''_0, \ldots, \pi''_\ell)$ maintaining the invariant. When batching a total of k NP statements, our batched proof consists of at most $\log k$ multi-hop seBARG proofs, each of which is a result of at most $\log k$ hops of aggregation and thus has size $O(\log k)$. Thus, the total proof size grows with $O(\log^2 k)$ (ignoring the dependence on the witness size m and the security parameter λ). See Figure 1 for an illustration.

We refer the reader to Section 6 for more details on the multi-hop seBARG definition and construction.

Multi-Hop *Hashed* seBARGs. To realize some of our applications, such as incrementally verifiable computation, we need a further generalization of (multi-hop) seBARGs, in which the verifier is given only a (somewhere extractable) hash v of the *instances*, rather than all the instances in the clear. The somewhere argument of knowledge property now states that if a proof π verifies with respect to a hash value v, then we can extract a valid witness ω from π for the instance x which is extractable from v. We call this a (multi-hop) hashed seBARG. A hashed seBARG is related to, but different from, an index seBARG: in the latter, there is a compressed representation of the instances from which each instance can be recovered quickly, whereas in the former, the instances can be arbitrary without necessarily a short representation. We believe that the notion of a hashed (multi-hop) seBARG is of independent interest.

We show how to construct a hashed multi-hop seBARG from a rate-1 seBARG. The hashed multi-hop seBARG proof combiner takes as input, instead of k batches of statements, k hash values $\mathsf{v}_1, \ldots, \mathsf{v}_k$ of depth d, and corresponding proofs $\pi_1, \ldots, \pi_k$ and outputs a hash value v of depth $d+1$ and corresponding proof π. The construction is similar to the regular multi-hop seBARG, except that now when aggregating, every witness includes a hash value v_i and an opening ρ_i of v_i w.r.t. v, in addition to the proof π_i. In the initial aggregation step, the witness is simply an NP witness w_i,

and v_i is simply x_i. In later aggregations, the witness becomes a seBARG proof π_i corresponding to a hash value v_i. We rely on the somewhere extraction property of the hash to ensure that the instance which we recursively extract from the proof, which goes along with the witness ω which we recursively extract from the proof, matches the instance x which we recursively extract from the hash value.

Theorem 1.4 (Informal)**.** *Under the* LWE *assumption, there exists a hashed multi-hop* seBARG *for* NP *statements with n-bit instances and m-bit witnesses where the size of the proof after B hops is $m + n \cdot B \cdot \mathsf{poly}(\lambda)$.*

We refer the reader to Section 6.3 for more details on hashed (multi-hop) seBARG.

Multi-Hop Aggregate Signature Schemes from LWE. As an immediate application of our multi-hop seBARG, we construct a multi-hop aggregate signature scheme.

Aggregate signatures were introduced by Boneh, Gentry, Lynn, and Shacham [BGLS03] to enable the compression of a sequence of signatures $\sigma_1, \ldots, \sigma_k$, where each σ_i is a signature of an arbitrary message m_i w.r.t. an arbitrary verification key vk_i, into a single *aggregated* signature $\widehat{\sigma}$ whose size is independent of k. While the original motivation for aggregate signatures was to compress certificate chains and to reduce cryptographic overhead in secure BGP, the notion has recently found a great deal of practical interest in the context of blockchains where they provide tangible savings in communication and space. Multi-hop aggregate signatures require the ability to aggregate several aggregate signatures, similar to multi-hop seBARGs.

The original construction of aggregate signatures [BGLS03] relied on bilinear maps and was proven secure in the random oracle model. A more recent construction [HKW15] uses indistinguishability obfuscation and the RSA assumption, and constructs an aggregate signature scheme secure in the standard model. Finally, it is a folklore observation that SNARKs for NP immediately give us an aggregate signature scheme. All these schemes support multi-hop aggregation. Thus, with the exception of [HKW15], all constructions of aggregate signature schemes rely either on random oracles, or on specialized knowledge-type assumptions.

In this work, we observe that our construction of multi-hop seBARGs immediately gives us a multi-hop aggregate signature scheme (in the standard model) secure under the LWE assumption.

Theorem 1.5 (Informal)**.** *Assuming the existence of rate-1* seBARG*s for* NP*, there exists a multi-hop aggregate signature scheme. Consequently, there exists a multi-hop aggregate signature scheme in the standard model under the hardness of* LWE*.*

Finally, we note that the "seBARG lens" gives us multi-hop aggregate signatures with several additional desirable properties. For example:

1. *Universal Aggregation:* The goal of universal signature aggregation [HKW15], is to be able to aggregate a collection of signatures produced from *any* signing algorithm (as long as the description of the verification algorithm is fixed). Universal signature aggregation allows real-world systems to continue to use existing signature schemes and public-key infrastructure, while supporting signature aggregation. The generality of the seBARG methodology lets us add the signature aggregation feature to any signature scheme by including the description of the verification algorithm as part of the instance. While [HKW15] and the SNARK-based construction give us universal signature aggregation, [BGLS03] does not.

2. *Local Verifiability:* A locally verifiable aggregate signature scheme [GV22] allows a verifier to check, given an aggregate signature $\widehat{\sigma}$ and a small advice that can be computed from all the (m_i, vk_i) pairs, whether a particular message m is in the aggregated set $(m_1, \ldots, m_k)$, *without having to know the entire list of messages.* Indeed, the signature verifier runs in time sublinear in k. Using a hashed (multi-hop) seBARG scheme, we show how to construct a (multi-hop) locally verifiable aggregate signature scheme.

We refer the reader to Section 7.1 for more details. While our constructions are not concretely efficient (as compared to, e.g., [BGLS03]), we believe that a refinement of the seBARG lens can potentially give us a concretely efficient LWE-based aggregate signature scheme. We leave the exploration of this line of thought for future work.

Incrementally Verifiable Computation from LWE. Consider the computation of a Turing machine $\mathcal{M}$ on an input x, a computation so long that no single person can finish it all by herself. Each person thus takes the intermediate state resulting from a partial computation, a Turing machine configuration conf_t at time t, and updates it by running ℓ more steps to get a configuration $\mathsf{conf}_{t+\ell}$. To certify that the computation has been done correctly, one needs to do more. Given a succinct proof π_t certifying that the t-th configuration is conf_t, one needs to efficiently compute a new succinct proof $\pi_{t+\ell}$ certifying that the $(t+\ell)$-th configuration is $\mathsf{conf}_{t+\ell}$. Importantly, the time to generate $(\mathsf{conf}_{t+\ell}, \pi_{t+\ell})$ should depend only on ℓ, and should be independent of t. In particular, the length of the proofs does not grow with t; at worst, we allow a poly-logarithmic in t growth. This is the notion of incrementally verifiable computation (IVC), a notion of great interest in settings where long ongoing computations are performed by a distributed network of mutually distrusting parties [Mic94, Val08, BCCT13].

All known constructions of IVC use full-fledged SNARKs for NP as a building block, and thus rely on non-falsifiable knowledge assumptions. Kalai, Paneth and Yang [KPY20] constructed a weak form of IVC for deterministic time-T computations with proofs of size $\mathsf{poly}(\lambda, T^\epsilon)$ for any constant (or even slightly sub-constant) $\epsilon > 0$, assuming the hardness of problems on elliptic curves that support bilinear maps. Roughly speaking, in their construction, $\pi_{t+\ell}$ simply consists of (π_t, π') where π' certifies the correctness of the computation from cf_t to $\mathsf{cf}_{t+\ell}$, and after $p(\lambda)$ many succinct proofs are accumulated (where λ is the security parameter), they are combined into a single succinct proof (using a seBARG), where each such combining step incurs a multiplicative blowup of $q(\lambda) \ll p(\lambda)$ to the proof size. Therefore, if this combining step is applied B times then the resulting proof increases by a factor of $q(\lambda)^B$ which allows for only a constant number of combination steps. Setting parameters appropriately, this results in a somewhat succinct proof of size T^ϵ for any constant $\epsilon > 0$, where T is the run-time of the computation. (We remark that a careful balancing act with the security parameter gives them slightly more, namely a slightly sub-constant ϵ, but they are inherently limited by this exponential growth of the proof in the depth B.)

We note that our rate-1 seBARG scheme allows us to do this combining step while paying only an *additive* $\mathsf{poly}(\lambda)$ blowup. In particular, using our seBARG, we can apply this combining step B times while incurring only an additive blowup of $B \cdot \mathsf{poly}(\lambda)$, and while relying only on the LWE assumption. This allows us to obtain an IVC scheme for *any* deterministic computation, even ones in EXP (e.g., by using a tree-like structure in which the depth, and thus the blowup, is only *logarithmic* in the run-time of the computation).

Theorem 1.6 (Informal)**.** *There exists an incrementally verifiable computation scheme for any*

deterministic computation under the LWE *assumption, where the proofs grow poly-logarithmically with the run-time of the computation.*

Our construction of an IVC scheme uses multi-hop seBARGs. In the first hop, we batch k instances, where the i-th instance is $(x, \mathsf{cf}_{i-1}, \mathsf{cf}_i)$. The witness for this statement is empty since verifying an instance can be done efficiently. It should be possible to verify the batched proof given only the initial configuration cf_0 and the last configuration computed so far cf_k, without knowing the intermediate configurations. This is crucial to obtain our desired efficiency gain. This is where our notion of a *hashed* multi-hop seBARG comes in. Using a hashed multi-hop seBARG, we can verify proofs without knowing all the corresponding instances, rather only their (somewhere statistically binding) hashes.

Continuing along these lines, we reach the second stage where we have k proofs $\pi_1, \ldots, \pi_k$ and k hash values $\mathsf{v}_1, \ldots, \mathsf{v}_k$, where π_i certifies that v_i is a hash of consecutive configurations starting with $\mathsf{cf}_{k(i-1)+1}$ and ending with cf_{ki}. We can feed these into the hashed multi-hop seBARG proof combiner and get an aggregated proof π, but we run into an issue along the "boundaries:" there is no guarantee that $\mathsf{cf}_{k(i-1)}$ actually transitions to $\mathsf{cf}_{k(i-1)+1}$ for $i \in \{2, \ldots, k\}$.

We handle this by overlapping the batches that we aggregate. In the first hop, we batch $2k$ instances at a time, so the second half of each batch overlaps with the first half of the next batch: The proof π_i actually certifies that $(\mathsf{v}_{i-1}, \mathsf{v}_i)$ is a hash of consecutive configurations starting with $\mathsf{cf}_{k(i-2)+1}$ and ending with cf_{ki}, so it indeed certifies that $\mathsf{cf}_{k(i-1)}$ actually transitions to $\mathsf{cf}_{k(i-1)+1}$. This way, the problematic "boundary" of any batch is completely contained within another batch. We refer the reader to Section 7.2 for more details on the IVC definition and construction.

2 Technical Overview

We sketch our main result that takes any seBARG scheme and converts it into a rate-1 seBARG, assuming a rate-1 additive homomorphic encryption scheme with certain properties (which in turn can be constructed based on LWE).

2.1 Main Ingredient: Flexible RAM SNARGs with Partial Input Soundness

The central tool that we use repeatedly in our construction is a flexible RAM SNARG with partial input soundness, a primitive that was very recently introduced and constructed in [KLVW22].

Remark 2.1. Our work and that of [KLVW22] are intertwined. An earlier (unpublished) version of this work did not use RAM SNARGs, and used *quasi-arguments* [KPY19] instead. Following [KLVW22], we noticed that their notion of *flexible* RAM SNARG *with partial input soundness* can be used to significantly simplify our construction and analysis. It is this simplified version that is presented in this paper.

The notion of a RAM delegation scheme was introduced and constructed in the work of Kalai and Paneth [KP16]. A (publicly-verifiable non-interactive) RAM delegation scheme, is a SNARG for RAM computations where a prover, given a crs, a time-T RAM machine M and an n-bit input x, can produce a short, $\mathsf{poly}(\lambda, \log T)$-bit, proof that $y = M(x)$. Given a $\mathsf{poly}(\lambda)$-bit "digest" of the input x, the verifier runs in time $\mathsf{poly}(\lambda, \log T)$, *independent of the input length* n, and either accepts or rejects the proof. The key difference from a SNARG for $\mathcal{P}$ is that the prover runtime is

proportional to the RAM runtime, but more importantly, that the verifier runtime, given the input digest, is independent of the input length and polylogarithmic in the RAM runtime.

For a formal definition, we refer the reader to Section 3.3.

For our work, the standard notion of a RAM SNARG is insufficient, and we need a RAM SNARG with two additional properties, as was defined and constructed in [KLVW22]: First, the RAM SNARG has to be *flexible* in the sense that any hash family with local opening can be used to digest the input. Second, it has a stronger soundness guarantee known as *partial input soundness*. In prior RAM SNARGs, soundness is guaranteed only if the adversary knows the entire input that is being digested. Partial input soundness guarantees that if the memory is digested using a somewhere extractable hash function that is extractable on a set of coordinates S, and if the RAM computation only reads coordinates in S, then soundness holds.

Such a flexible RAM SNARG with partial input soundness was constructed in [KLVW22] from any seBARG scheme (and SEH hash) and was used to boost the succinctness of seBARG proofs.

2.2 Our Rate-1 seBARG Scheme

The high level intuition behind our construction is the following. Suppose a prover is given $x_1, \ldots, x_k$ and corresponding witnesses $\omega_1, \ldots, \omega_k$, and wishes to convince the verifier that $x_1, \ldots, x_k$ are all in the language. The basic idea is to simply hash all the witnesses using a rate-1 SSB hash function, so that the hash value v is statistically binding on a single witness ω_{i^*}. The seBARG proof will consist of v together with a succinct proof π that v is obtained by hashing valid witnesses for $x_1, \ldots, x_k$. Importantly, to ensure that the proof (v, π) is of rate-1, i.e., of length $m + \mathsf{poly}(\lambda)$ where m is the length of a single witness, the succinct proof π must be of length $\mathsf{poly}(\lambda)$, since v is already of length $m + \mathsf{poly}(\lambda)$.

Constructing such a succinct proof is the main technical challenge and novelty of our rate-1 seBARG construction. Jumping ahead, we note that we do not know how to construct such a succinct proof if v is computed using any rate-1 SSB hash family, and we need to use a (rate-1) *fully local* SEH hash family (flSEH), a primitive described in Section 1.2. Constructing a rate-1 flSEH hash family is a grand challenge of its own, and we elaborate on it in Section 2.3 and Section 4.

At first it may seem that this approach, of constructing a succinct proof that v is a hash of valid witnesses, is doomed to fail since it requires a SNARG for NP, which is currently out of our reach. Yet, on a closer look, we notice that a SNARG for all of NP is not necessary. The reason is that a witness ω_{i^*} for x_{i^*} can be efficiently extracted from the hash value v given the trapdoor td. Indeed it is this hash value that saves the day!

In order to prove that v is a commitment to valid witnesses, we draw inspiration from the works on RAM delegation (starting with [KP16]), and in particular, we use the recent notion of a *flexible* RAM SNARG *with partial input soundness* [KLVW22]. Recall that a flexible RAM SNARG allows us to digest the memory with any hash family with local opening, and in particular with a fully local SEH (flSEH) hash family. The partial input soundness property guarantees that if the hash value v is extractable on w_{i^*} and the RAM program only touches the memory locations corresponding to w_{i^*} then soundness holds.

Given such a RAM SNARG scheme, we proceed as follows: in our seBARG scheme, the prover generates for every $i \in [k]$ a succinct RAM SNARG π_i for the RAM computation that checks that the i-th witness in memory is a valid witness of x_i. The size of each such proof grows with the size of a local opening, and moreover, with the time it takes to verify a local opening. This is precisely where the need for a rate-1 flSEH hash family comes in, as it allows each proof π_i to be of size $\mathsf{poly}(\lambda)$.

Note that if the prover would send $(\pi_1, \ldots, \pi_k)$ to the verifier then we could argue semi-adaptive soundness by sampling a hash key where the extractability is on the witness corresponding to the instance x_{i^*} which is not in the language, and then use the partial input soundness to argue that π_{i^*} would be rejecting with overwhelming probability. The issue is that we want our seBARG to be succinct and hence cannot afford to send all the proofs $(\pi_1, \ldots, \pi_k)$. Instead we will send a seBARG proof (which need not be rate-1) that for every $i \in [k]$ there exists a RAM SNARG π_i that is a valid proof w.r.t. the digest v.

2.3 Our Fully Local SEH Family

We next describe at a high-level the ideas behind our construction of a rate-1 fully local SEH (flSEH) family.

Rate-1 SEH family. As a first step, we focus on achieving the rate-1 property (without the fully-local property). Our construction uses as a building block the SEH family of Hubáček and Wichs [HW15], which is a tree-like construction that can be instantiated using any leveled fully homomorphic encryption (FHE) scheme [BV11, BGV12]. We note that in their construction, the hash key hk consists of encryptions of the indices $\{i_1 \ldots, i_m\}$ where the hash function is statistically bound, and the hash value $\mathsf{Hash}(\mathsf{hk}, x)$ consists of a bit-by-bit encryption of $x_{i_1}, \ldots, x_{i_m}$.

To make this construction rate-1, we use a rate-1 levelled FHE [BDGM19, GH19b, DGI+19b] as the underlying encryption. We use the fact that such an FHE scheme can convert any set of (evaluated) ciphertexts $\mathsf{ct}_1, \ldots, \mathsf{ct}_m$, each encrypting a single bit b_i, into a ciphertext v of size $m + \mathsf{poly}(\lambda)$ encrypting $(b_1, \ldots, b_m)$. Thus, the idea is to output v (which is rate-1), as opposed to outputting the bit-wise ciphertexts $\mathsf{ct}_1, \ldots, \mathsf{ct}_m$. Unfortunately, by outputting only v we lose the local opening property. This is fixed as follows.

Obtaining fully local opening. Recall that our goal is to provide a local opening to any bit of the input that can be verified in time $\mathsf{poly}(\lambda)$ (i.e., significantly less than m). Thus, an opening needs to be verified without even reading v! To reach this goal, we send in addition to v a somewhere extractable hash of the ciphertexts $\mathsf{ct}_1, \ldots, \mathsf{ct}_m$, denoted by h. Importantly h is statistically binding on only one ciphertext ct_i and hence is of size $\mathsf{poly}(\lambda)$.

We use v to extract the m coordinates, and use h to open any desired coordinate. We elaborate on the opening procedure below, and mention that to verify the validity of an opening we use only h (which is succinct) and do not use v. This allows us to obtain our fully-local guarantee.

For this approach to work we need a mechanism that checks consistency between v and h. Thus our hash value actually contains three components $(\mathsf{v}, \mathsf{h}, \pi)$, where π is a succinct proof that certifies the consistency between ct and h. The main technical burden is in constructing such a proof of consistency. Note that to obtain our promised rate-1 construction this proof must be of size $\mathsf{poly}(\lambda)$. We defer the description of this proof, and we first discuss the local opening (which is substantially simpler).

Fully local opening using h. In the construction of Hubáček and Wichs [HW15], an opening to a bit b consists of the bit b together with m openings, $\rho_1, \ldots, \rho_m$, where each ρ_j corresponds to the output ciphertext ct_j. Moreover, each ρ_j is of size $\mathsf{poly}(\lambda)$ and can be verified in time $\mathsf{poly}(\lambda)$ given ct_j. To obtain full succinctness, rather that sending all these openings $\rho_1, \ldots, \rho_m$ (together with

local openings of $\mathsf{ct}_1, \ldots, \mathsf{ct}_m$), we give a somewhere extractable BARG proof that for every $j \in [m]$ there exists a triplet $(\rho_j, \mathsf{ct}_j, o_j)$ where ρ_j is a valid opening to b w.r.t. ct_j and that o_j is a valid opening to ct_j w.r.t. h.

Thus, our construction uses not only the SEH family corresponding to a rate-1 FHE scheme, but also uses a somewhere extractable BARG scheme.

Proving consistency between ct and h. The remaining ingredient is the succinct proof π that certifies the consistency between v and h. Namely, we need to prove that there exist ciphertexts $\mathsf{ct}_1, \ldots, \mathsf{ct}_m$ that hash to h and their corresponding rate-1 ciphertext is indeed v. Note that this is an NP computation and SNARGs for NP are beyond our reach, and is harder than our goal that we started with. Thus, it seems that we are back to square one.

However, we observe that this NP computation has special properties that will allow us to construct a SNARG for it. The initial observation is that the NP witness is hashed (via h), and thus one may hope to use RAM delegation here. Unfortunately, the soundness of RAM delegation is not strong enough, since it provides soundness only if the adversary knows the memory that is digested. In our setting the memory is $\mathsf{ct}_1, \ldots, \mathsf{ct}_m$ and we have no means of extracting these ciphertexts from the adversary. What saves the day is that we can extract one ciphertext ct_i (since the underlying hash is somewhere extractable).

Indeed, to construct this SNARG we use a RAM SNARG from the recent work of [KLVW22]. This RAM SNARG has two properties that are crucial in our setting. First, it is *flexible* in the sense that the memory can be digested using any hash family with local opening, and in particular using any hash family that is also locally extractable. Second, it has a stronger soundness guarantee, known as *partial input soundness* which guarantees correctness if the RAM computation only reads the memory from locations in $S \subseteq [N]$ (where N is the memory length) and the hash function is extractable on the set S.

At first, it is not clear that this soundness guarantee is helpful since the RAM computation that computes v from the memory $(\mathsf{ct}_1, \ldots, \mathsf{ct}_m)$ reads the entire memory. Yet, if we use the scheme from [BDGM19] (which is based on LWE) then v is computed in the following way, which does allow us to make use of this underlying RAM SNARG.

1. First, each ciphertext ct_j is expanded into a vector ciphertext, denote by ct'_j.
2. Second, all these vector ciphertexts are added (over a finite field) to obtain a ciphertext $\mathsf{ct}' = \sum_{j=1}^{m} \mathsf{ct}'_j$.
3. Finally, each bit of v is computed separately, and depends only on a few bits from ct'.

Note that by the partial input soundness guarantee one can compute a RAM SNARG for each bit of ct'_j that certifies its correctness w.r.t. h, and correctness is guaranteed if h is extractable on ct_j. Denote all the proofs corresponding to ct'_j by π'_j.

We append to the hash value a somewhere extractable hash of

$$(\mathsf{ct}'_1, \pi'_1, \ldots, \mathsf{ct}'_m, \pi'_m)$$

If Item 2 did not exist and each bit of v depended only on a few bits $(\mathsf{ct}'_1, \ldots, \mathsf{ct}'_m)$ then we would be done since we could simply add a BARG asserting that each bit of v is computed correctly, where each witness is the relevant bits from $(\mathsf{ct}'_1, \ldots, \mathsf{ct}'_m)$ and the corresponding proofs and all their openings.

Unfortunately, Item 2 does exist which causes an additional complication, stemming from the fact that this computation is not local (as opposed to Item 1 and 3 which are local). What saves the say is the fact that the computation in Item 2 is simply addition, which can be computed in a tree-like manner where each node computation is a local computations in the nodes of the layer below. Thus, we add $\log m$ hash values $\mathsf{h}_1, \ldots, \mathsf{h}_{\log m}$, where h_i is the hash of all the computations at layer i of the tree along of RAM proofs of correctness, where each computation depends only on a few coordinates of the memory at layer $i-1$ of the tree, which allows us to rely on the partial input soundness guarantee.

To summarize, in our construction of a flSEH hash family, the hash value consists of the following ingredients:

1. A hash value h, which is a somewhere extractable hash of $(\mathsf{ct}_1, \ldots, \mathsf{ct}_m)$, which in turn is the hash value from the construction of [HW15]. The size of h is $\mathsf{poly}(\lambda)$.
2. A rate-1 ciphertext v, which is a rate-1 version of $(\mathsf{ct}_1, \ldots, \mathsf{ct}_m)$, of size $m + \mathsf{poly}(\lambda)$.
3. A proof of consistency between h and v which consists of hash values $(\mathsf{h}', \mathsf{h}_0, \ldots, \mathsf{h}_{\log m})$ together with a seBARG proof. Each of these hash values hashes ciphertexts and proofs of consistency, and the final seBARG proof proves consistency between $\mathsf{h}_{\log m}$ and v and that all the proofs hashed in $\mathsf{h}_{\log m}$ are accepting.

3 Preliminaries

Notations. We use PPT to denote probabilistic polynomial-time, and denote the set of all positive integers up to n as $[n] := \{1, \ldots, n\}$. Also, we use $[0, n]$ to denote the set of all non-negative integers up to n, i.e. $[0, n] := \{0\} \cup [n]$. Throughout this paper, unless specified, all polynomials we consider are positive polynomials. For any finite set S, $x \leftarrow S$ denotes a uniformly random element x from the set S. Similarly, for any distribution $\mathcal{D}$, $x \leftarrow \mathcal{D}$ denotes an element x drawn from the distribution $\mathcal{D}$.

3.1 Somewhere Extractable Hash (SEH) Families

In what follows we recall the definition of a somewhere extractable (SEH) hash family based on prior works [HW15, OPWW15b].

Syntax. A SEH hash family consists of algorithms

$$(\mathsf{Gen}, \mathsf{Hash}, \mathsf{Open}, \mathsf{Verify}, \mathsf{Extract})$$

with the following syntax:

$\mathsf{Gen}(1^\lambda, N, I) \to (\mathsf{hk}, \mathsf{td})$. This is a probabilistic poly-time setup algorithm that takes as input a security parameter 1^λ in unary, a message length N, and a subset $I \subseteq [N]$. It outputs a hash key hk along with trapdoor td.

$\mathsf{Hash}(\mathsf{hk}, x) \to \mathsf{v}$. This is a deterministic poly-time algorithm that takes as input a hash key hk generated by $\mathsf{Gen}(1^\lambda, N, I)$ and an input $x \in \{0,1\}^N$, and outputs a hash value $\mathsf{v} \in \{0,1\}^{|I| \cdot \mathsf{poly}(\lambda)}$.

$\mathsf{Open}(\mathsf{hk}, x, j) \to (b, \rho)$**.** This is a deterministic poly-time algorithm that takes as input a hash key hk generated by $\mathsf{Gen}(1^\lambda, N, I)$, an input $x \in \{0,1\}^N$ and an index $j \in [N]$, and outputs a bit $b \in \{0,1\}$ and an opening $\rho \in \{0,1\}^{\leq |I| \cdot \mathsf{poly}(\lambda)}$.

$\mathsf{Verify}(\mathsf{hk}, \mathsf{v}, j, b, \rho)$**.** This is a deterministic poly-time algorithm that takes as input a hash key hk generated by $\mathsf{Gen}(1^\lambda, N, I)$, a hash value $\mathsf{v} \in \{0,1\}^{|I| \cdot \mathsf{poly}(\lambda)}$, an index $j \in [N]$, a bit $b \in \{0,1\}$ and an opening $\rho \in \{0,1\}^{\leq |I| \cdot \mathsf{poly}(\lambda)}$, and outputs 1 (accept) or 0 (reject).

$\mathsf{Extract}(\mathsf{td}, \mathsf{v}, j) \to u$. This is a deterministic poly-time extraction algorithm that takes as input a trapdoor td generated by $\mathsf{Gen}(1^\lambda, N, I)$, a hash value v, and index $j \in [|I|]$ and outputs a bit u.

We sometimes use the notation $\mathsf{Extract}(\mathsf{td}, \mathsf{v}) = (\mathsf{Extract}(\mathsf{td}, \mathsf{v}, j))_{j \in [|I|]}$.

Definition 3.1 (SEH)**.** *A* SEH *hash family* $(\mathsf{Gen}, \mathsf{Hash}, \mathsf{Open}, \mathsf{Verify}, \mathsf{Extract})$ *is required to satisfy the following properties:*

Efficiency. *The size of the hash key* hk *and the hash value* v *is at most* $|I| \cdot \mathsf{poly}(\lambda)$*.*

Index hiding. *For any poly-size adversary* $\mathcal{A}$*, any polynomial* $N = N(\lambda)$*, and any* $I_0, I_1 \subseteq [N]$ *such that* $|I_0| = |I_1|$*, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for every* $\lambda \in \mathbb{N}$*,*

$$\Pr\left[\mathcal{A}(\mathsf{hk}) = b : \begin{array}{l} b \leftarrow \{0,1\} \\ (\mathsf{hk}, \mathsf{td}) \leftarrow \mathsf{Gen}(1^\lambda, N, I_b) \end{array}\right] \leq \frac{1}{2} + \mathsf{negl}(\lambda),$$

Opening completeness. *For any* $\lambda \in \mathbb{N}$*, any* $N \leq 2^\lambda$*, any subset* $I \subseteq [N]$*, any index* $j \in [N]$*, and any* $x \in \{0,1\}^N$*,*

$$\Pr\left[\begin{array}{l} b = x_j \\ \wedge\ \mathsf{Verify}(\mathsf{hk}, \mathsf{v}, j, b, \rho) = 1 \end{array} : \begin{array}{l} (\mathsf{hk}, \mathsf{td}) \leftarrow \mathsf{Gen}(1^\lambda, N, I), \\ \mathsf{v} = \mathsf{Hash}(\mathsf{hk}, x), \\ (b, \rho) = \mathsf{Open}(\mathsf{hk}, x, j), \end{array}\right] = 1.$$

Somewhere statistically binding w.r.t. opening. *For any* $\lambda \in \mathbb{N}$*, any* $N \leq 2^\lambda$*, any subset* $I \subseteq [N]$*, any index* $i^* \in I$*, and any (all powerful) adversary* $\mathcal{A}$*, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for every* $\lambda \in \mathbb{N}$*,*

$$\Pr\left[\begin{array}{l} \mathsf{Verify}(\mathsf{hk}, \mathsf{v}, i^*, b, \rho) = 1 \\ \wedge\ b \neq b_{i^*} \end{array} : \begin{array}{l} (\mathsf{hk}, \mathsf{td}) \leftarrow \mathsf{Gen}(1^\lambda, N, I), \\ (\mathsf{v}, b, \rho) \leftarrow \mathcal{A}(\mathsf{hk}), \\ (b_i)_{i \in I} = \mathsf{Extract}(\mathsf{td}, \mathsf{v}) \end{array}\right] \leq \mathsf{negl}(\lambda).$$

Extraction correctness. *For any* $\lambda \in \mathbb{N}$*, any* $N \leq 2^\lambda$*, any subset* $I \subseteq [N]$*, and any* $x \in \{0,1\}^N$*, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for every* $\lambda \in \mathbb{N}$*,*

$$\Pr\left[(x_i)_{i \in I} \neq \mathsf{Extract}(\mathsf{td}, \mathsf{v}, I) \ : \begin{array}{l} (\mathsf{hk}, \mathsf{td}) \leftarrow \mathsf{Gen}(1^\lambda, N, I) \\ \mathsf{v} \leftarrow \mathsf{Hash}(\mathsf{hk}, x) \end{array}\right] \leq \mathsf{negl}(\lambda).$$

Remark 3.1. Note that the index hiding property and the somewhere statistically binding w.r.t. opening property of a SEH hash family, implies the following property:

Computational binding w.r.t opening. For any poly-size adversary $\mathcal{A}$ any polynomial $N = N(\lambda)$ and any $I \subseteq [N]$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that for every $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{ll} \forall\, b \in \{0,1\} & : \quad (\mathsf{hk}, \mathsf{td}) \leftarrow \mathsf{Gen}(1^\lambda, N, I), \\ \mathsf{Verify}(\mathsf{hk}, \mathsf{v}, i, b, \rho_b) = 1 & \quad (i, \mathsf{v}, \rho_0, \rho_1) \leftarrow \mathcal{A}(\mathsf{hk}) \end{array}\right] \leq \mathsf{negl}(\lambda).$$

Lemma 3.2. *[KLVW22] Let* $(\mathsf{E.Gen}, \mathsf{Enc}, \mathsf{Dec})$ *be any encryption scheme that is either levelled fully homomorphic or rate-1 additively homomorphic. Then there is a somewhere extractable hash family with algorithms* $\mathsf{Gen}, \mathsf{Hash}$ *that satisfies the following:*

1. *For every* $\lambda \in \mathbb{N}$*, any* $N = N(\lambda) \leq 2^\lambda$*, any* $m \leq N$*, any* $I = \{i_1, \ldots, i_m\} \subseteq [N]$*, and any* $(\mathsf{hk}, \mathsf{td}) \leftarrow \mathsf{Gen}(1^\lambda, N, I)$*,*

$$\mathsf{hk} = (\mathsf{pk}, \mathsf{hk}_1, \ldots, \mathsf{hk}_m)$$

where for every $j \in [m]$*,* $\mathsf{hk}_j \in \{0,1\}^{\mathsf{poly}(\lambda)}$ *is an encryption of the index* $i_j \in [N]$ *w.r.t.* pk*, and* td *is the secret key corresponding to* pk*.*

In particular, if I *consists of* m *consecutive indices* $I = \{i+1, \ldots, i+m\}$ *then one can set* $\mathsf{hk} = (\mathsf{pk}, \mathsf{hk}_1)$*, and* $(\mathsf{hk}_2, \ldots, \mathsf{hk}_m)$ *can be computed efficiently from* hk*. In this case,* Gen *runs in time* $\mathsf{poly}(\lambda)$*.*

2. *There exists a negligible function* negl *such that for any* $\lambda \in \mathbb{N}$*, any* $N = N(\lambda) \leq 2^\lambda$*, any* $m \leq N$*, any* $I = \{i_1, \ldots, i_m\} \subseteq [N]$*, any* $j \in [m]$*, and any* $x \in \{0,1\}^N$*,*

$$\Pr\left[\mathsf{Dec}(\mathsf{td}, \mathsf{v}_j) = x_j \quad : \quad \begin{array}{l} (\mathsf{hk}, \mathsf{td}) \leftarrow \mathsf{Gen}(1^\lambda, N, I), \\ (\mathsf{v}_1, \ldots, \mathsf{v}_m) = \mathsf{Hash}(\mathsf{hk}, x) \end{array}\right] \leq \mathsf{negl}(\lambda).$$

In particular, for every $j \in [m]$*,* $\mathsf{v}_j \in \{0,1\}^{\mathsf{poly}(\lambda)}$*.*

3.2 Somewhere Extractable Batch Arguments (seBARGs)

Syntax. A (publicly verifiable and non-interactive) somewhere extractable batch argument scheme seBARG for an NP language $\mathcal{L}$ consists of the following polynomial time algorithms:

$\mathsf{Gen}(1^\lambda, k, n, i^*) \to (\mathsf{crs}, \mathsf{td})$. This is a probabilistic algorithm that takes as input a security parameter 1^λ, number of instances k, input length n, and an index $i^* \in [k]$. It runs in time at most $\mathsf{poly}(\lambda, n, \log k)$ and outputs a crs crs along with a trapdoor td.

$\mathcal{P}(\mathsf{crs}, x_1, \ldots, x_k, \omega_1, \ldots, \omega_k) \to \pi$. This is a prover algorithm takes as input a crs, k instances $x_1, \ldots, x_k$ and corresponding witnesses $\omega_1, \ldots, \omega_k$, and outputs a proof π.

$\mathcal{V}(\mathsf{crs}, x_1, \ldots, x_k, \pi) \to 0/1$. The verification algorithm takes as input a crs, k instances x_i for $i \in [k]$, and a proof π. It outputs a bit to signal whether the proof is valid or not.

Definition 3.3 (seBARG)**.** *A somewhere-extractable batch argument scheme (*seBARG*)* $(\mathsf{Gen}, \mathcal{P}, \mathcal{V})$ *for* $\mathcal{L}$ *is required to satisfy the following properties:*

Efficiency. *The size of the CRS and the proof is at most* $\mathsf{poly}(\lambda, \log k, n, m)$*, where* m *is the witness length.*

Completeness. *For any $\lambda \in \mathbb{N}$, and any $k = k(\lambda)$, $n = n(\lambda)$ of size at most 2^λ, any k instances $x_1, \ldots, x_k \in \mathcal{L}$, and their corresponding witnesses $\omega_1, \ldots, \omega_k \in \{0,1\}^m$, and any index $i^* \in [k]$,*

$$\Pr\left[\mathcal{V}(\mathsf{crs}, x_1, \ldots, x_k, \pi) = 1 : \begin{array}{l}(\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{Gen}(1^\lambda, k, n, i^*), \\ \pi \leftarrow \mathcal{P}(\mathsf{crs}, x_1, \ldots, x_k, \omega_1, \ldots, \omega_k)\end{array}\right] = 1.$$

Index hiding. *For any poly-size adversary $\mathcal{A}$, any polynomials $k = k(\lambda)$ and $n = n(\lambda)$, and any indices $i_0, i_i \in [k]$ there exists a negligible function $\mathsf{negl}(\cdot)$ such that for every $\lambda \in \mathbb{N}$,*

$$\Pr\left[\mathcal{A}(\mathsf{crs}) = b : \begin{array}{l} b \leftarrow \{0,1\}, \\ (\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{Gen}(1^\lambda, k, n, i_b)\end{array}\right] \leq \frac{1}{2} + \mathsf{negl}(\lambda).$$

Somewhere argument of knowledge. *There exists a PPT extractor $\mathcal{E}$ such that for any poly-size adversary $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that for any polynomials $k = k(\lambda)$ and $n = n(\lambda)$, and any index $i^* \in [k]$, for every $\lambda \in \mathbb{N}$,*

$$\Pr\left[\begin{array}{l}\mathcal{V}(\mathsf{crs}, x_1, \ldots, x_k, \pi) = 1 \\ \wedge\ \omega^* \textit{ is not a valid witness for } x_{i^*} \in \mathcal{L}\end{array} : \begin{array}{l}(\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{Gen}(1^\lambda, k, n, i^*) \\ (x_1, \ldots, x_k, \pi) = \mathcal{A}(\mathsf{crs}) \\ \omega^* \leftarrow \mathcal{E}\left(\mathsf{td}, \{x_i\}_{i \in [k]}, \pi\right)\end{array}\right] \leq \mathsf{negl}(\lambda).$$

Remark 3.2. We note that the somewhere argument of knowledge property implies the following *semi-adaptive soundness* property which asserts that for any poly-size adversary $\mathcal{A}$, any polynomials $k = k(\lambda)$ and $n = n(\lambda)$, and any index $i^* \in [k]$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that for every $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l}\mathcal{V}(\mathsf{crs}, x_1, \ldots, x_k, \pi) = 1 \\ \wedge\ x_{i^*} \notin \mathcal{L}\end{array} : \begin{array}{l}(\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{Gen}(1^\lambda, k, n, i^*) \\ (x_1, \ldots, x_k, \pi) = \mathcal{A}(\mathsf{crs})\end{array}\right] \leq \mathsf{negl}(\lambda).$$

Remark 3.3. In Section 4, we use a seBARG scheme which is somewhere extractable on a set of indices $I = \{i_1, \ldots, i_\ell\}$, where the size of the CRS and proof and runtime of the verifier are allowed to grow with ℓ. This is trivially achievable using any generic seBARG scheme $(\mathsf{Gen}, \mathcal{P}, \mathcal{V})$ which is somewhere extractable on one index by creating ℓ separate parallel CRS and corresponding proof segments. The setup algorithm, given $1^\lambda, k, n, I$, samples $\mathsf{crs}_j \leftarrow \mathsf{Gen}(1^\lambda, k, n, i_j)$ for each $j \in [\ell]$ and outputs $\mathsf{crs} = (\mathsf{crs}_j)_{j \in [\ell]}$. The prover, given $\mathsf{crs}, x_1, \ldots, x_k, \omega_1, \ldots, \omega_k$, computes $\pi_j \leftarrow \mathcal{P}(\mathsf{crs}_j, x_1, \ldots, x_k, \omega_1, \ldots, \omega_k)$ for each $j \in [\ell]$ and outputs $\pi = (\pi_j)_{j \in [\ell]}$. The verifier, given $\mathsf{crs}, x_1, \ldots, x_k, \pi$, outputs 1 if and only if $\mathcal{V}(\mathsf{crs}_j, x_1, \ldots, x_k, \pi_j)$ for each $j \in [\ell]$. The extractor, given $\mathsf{td}, x_1, \ldots, x_k, \pi$, computes $\omega_j^* \leftarrow \mathcal{E}(\mathsf{td}, x_1, \ldots, x_k, \pi_j)$ for each $j \in [\ell]$ and outputs $(\omega_j^*)_{j \in [\ell]}$.

Definition 3.4. *A seBARG scheme $(\mathsf{Gen}, \mathcal{P}, \mathcal{V})$ is said to be an* index seBARG *if the run-time of $\mathcal{V}$ on instances $x_1, \ldots, x_k$, where $x_i = (x, i)$ for every $i \in [k]$, is at most $\mathsf{poly}(\lambda, |x|, \log k)$.*

Theorem 3.5 ([CJJ21a])**.** *Assuming the hardness of the Learning with Errors* (LWE) *problem, there exists an index seBARG scheme with proof length $|\pi| = m \cdot \mathsf{poly}(\lambda, \log k)$.*

Rate-1 BARGs. In this work, we construct a rate-1 seBARG under the LWE assumption. Moreover, our crs grows only poly-logarithmically in n. Informally, in a rate-1 BARG the proof size is the size of a single witness with only an additive overhead. Formally, we define it as follows:

Definition 3.6. *A* seBARG *scheme* $(\mathsf{Gen}, \mathcal{P}, \mathcal{V})$ *is said to be* rate-1 *if the following two conditions are satisfied:*

1. Gen *runs in time* $\mathsf{poly}(\lambda, \log n, \log k)$,[4] *and outputs* crs *of size* $|\mathsf{crs}| = \mathsf{poly}(\lambda, \log k, \log n)$.
2. *The proof generated by* $\mathcal{P}(\mathsf{crs}, x_1, \ldots, x_k, \omega_1, \ldots, \omega_k)$ *is of length* $m + \mathsf{poly}(\lambda, \log k)$.

3.3 Flexible RAM SNARGs with Partial Input Soundness

In this work we use a flexible RAM SNARG with partial input soundness, as defined in [KLVW22]. Such a RAM SNARG is for RAM computations that only read from memory (and do not write). However, it is flexible with respect to the hash family used to digest the memory, and works with any hash family with local opening. Jumping ahead, the reason we need this flexibility is that our rate-1 seBARG construction in Section 5, uses a RAM SNARG which digests its memory using a rate-1 fully-local somewhere extractable hash function (defined and constructed in Section 4).

Importantly, this RAM SNARG achieves a soundness guarantee known as *partial input soundness*, which is stronger than the soundness achieved in the recent works of [KPY19, CJJ21a]. In what follows, for the sake of simplicity, we define a flexible RAM SNARG, where the flexibility is only with respect to somewhere extractable hash family (see Section 3.1), as opposed to any hash family with local opening.

Syntax. Let $\mathcal{R}$ be a RAM machine. A **flexible** (publicly verifiable and non-interactive) RAM SNARG for $\mathcal{R}$ is associated with a somewhere extractable hash family (Section 3.1)

$$\mathsf{SEH} = (\mathsf{SEH.Gen}, \mathsf{SEH.Hash}, \mathsf{SEH.Open}, \mathsf{SEH.Verify}, \mathsf{SEH.Extract}),$$

and consists of the following algorithms:

$\mathsf{Gen}(1^\lambda, T) \to \mathsf{crs}$. This is a probabilistic poly-time setup algorithm that takes as input a security parameter 1^λ and a time bound T, and outputs a common reference string crs.

$\mathsf{Digest}(\mathsf{hk}, x) \to \mathsf{v}$. This is a deterministic polynomial-time algorithm that takes as input an SEH hash key hk, generated by $\mathsf{SEH.Gen}(1^\lambda, N, I)$ for some $N \in \mathbb{N}$, and a bit string $x \in \{0,1\}^N$, and outputs the digest $\mathsf{v} = \mathsf{SEH.Hash}(\mathsf{hk}, x)$ of size $|I| \cdot \mathsf{poly}(\lambda)$.

$\mathcal{P}(\mathsf{crs}, \mathsf{hk}, x_{\mathsf{imp}}, x_{\mathsf{exp}}) \to (b, \pi)$. This is a deterministic polynomial-time prover that takes as input a crs, a hash key hk, and an input $x = (x_{\mathsf{imp}}, x_{\mathsf{exp}})$ which consists of a (long) implicit input x_{imp} and a (short) explicit input x_{exp}, and outputs a bit $b = \mathcal{R}(x) \in \{0,1\}$ and a proof π.

$\mathcal{V}(\mathsf{crs}, \mathsf{hk}, \mathsf{v}, x_{\mathsf{exp}}, b, \pi) \to \{0,1\}$. This is a deterministic polynomial-time verifier that takes as input a crs, an SEH hash key hk, a digest v of the (long) implicit input, a (short) explicit input x_{exp}, a bit $b \in \{0,1\}$, and a proof π, and outputs 1 (accept) or 0 (reject).

Remark 3.4. Sometimes the goal of the prover $\mathcal{P}$ is to prove that the RAM computation is accepting (i.e., outputs 1). In this case we omit the bit b (which is set to 1), and simply think of $\mathcal{V}$ as taking as input the tuple $(\mathsf{crs}, \mathsf{hk}, \mathsf{v}, x_{\mathsf{exp}}, \pi)$. This will indeed be the case in Sections 4 and 5.

[4] We note that we require that Gen runs in time poly-logarithmically in n. This may not be an inherent requirement for a rate-1 BARG. We add this requirement since we can achieve it, and we use it in Section 7 where we construct a multi-hop seBARG scheme and an aggregate signature scheme.

Definition 3.7. *A flexible* RAM SNARG

$$(\mathsf{Gen}, \mathsf{Digest}, \mathcal{P}, \mathcal{V})$$

associated with a somewhere extractable hash family (Section 3.1)

$$\mathsf{SEH} = (\mathsf{SEH.Gen}, \mathsf{SEH.Hash}, \mathsf{SEH.Open}, \mathsf{SEH.Verify}),$$

satisfies the following properties:

1. **Efficiency.** *The length of the* RAM *proof π is at most $\mathsf{poly}(\lambda, T_V)$ where T_V is a bound on the run-time of* SEH.Verify.

   In particular, if the hash key can be partitioned into $\mathsf{hk} = (\mathsf{hk}_\ell, \mathsf{hk}_s)$ and the hash value can be partitioned into $\mathsf{v} = (\mathsf{v}_\ell, \mathsf{v}_s)$, and SEH.Verify *takes as input only $(\mathsf{hk}_s, \mathsf{v}_s)$, together with (j, b, ρ), then $|\pi| \leq \mathsf{poly}(|\mathsf{hk}_s|, |\mathsf{v}_s|, |\rho|, \lambda)$, and thus the run-time of $\mathcal{V}$ is at most $\mathsf{poly}(|\mathsf{hk}_s|, |\mathsf{v}_s|, |\rho|, \lambda)$.*

   Remark 3.5. Jumping ahead, we note that if the underlying SEH hash family is a *fully local* SEH family (defined in Section 4) then $|\mathsf{hk}_s|, |\mathsf{v}_s|, |\rho| \leq \mathsf{poly}(\lambda)$ in which case the length of π and the run-time of $\mathcal{V}$ is at most $\mathsf{poly}(\lambda)$.

2. **Completeness.** *For any $\lambda, N \in \mathbb{N}$ such that $N \leq T(N) \leq 2^\lambda$ and any $x = (x_{\mathsf{imp}}, x_{\mathsf{exp}}) \in \{0,1\}^N$ such that $\mathcal{R}(x)$ halts within T time steps, and any $I \in [|x_{\mathsf{imp}}|]$ we have that*

$$\Pr\left[\begin{array}{l|l} \mathcal{V}(\mathsf{crs}, \mathsf{hk}, \mathsf{v}, x_{\mathsf{exp}}, b, \pi) = 1 & \mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda), \\ \wedge\ b = \mathcal{R}(x) & (\mathsf{hk}, \mathsf{td}) \leftarrow \mathsf{SEH.Gen}(1^\lambda, |x_{\mathsf{imp}}|, I), \\ & (b, \pi) = \mathcal{P}(\mathsf{crs}, \mathsf{hk}, x), \\ & \mathsf{v} = \mathsf{Digest}(\mathsf{hk}, x_{\mathsf{imp}}) \end{array}\right] = 1.$$

3. **Partial-input soundness**: *For any poly-size adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ and any polynomial $T = T(\lambda)$ there exists a negligible function $\mathsf{negl}(\cdot)$ such that for every $\lambda \in \mathbb{N}$,*

$$\Pr\left[\begin{array}{l|l} \mathcal{V}(\mathsf{crs}, \mathsf{hk}, \mathsf{v}, x_{\mathsf{exp}}, b^*, \pi) = 1 & \mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda), \\ \wedge\ \mathcal{R}(x_{\mathsf{imp}}, x_{\mathsf{exp}}, T) = 1 - b^* & (1^N, I) = \mathcal{A}_1(\mathsf{crs}), \\ \quad \textit{and does not read any} & (\mathsf{hk}, \mathsf{td}) \leftarrow \mathsf{SEH.Gen}(1^\lambda, N, I), \\ \quad \textit{location in } [N] \setminus I & \left(\mathsf{v}, x_{\mathsf{exp}}, b^*, \pi\right) = \mathcal{A}_2(\mathsf{crs}, \mathsf{hk}), \\ & (b_j)_{j \in I} = \mathsf{SEH.Extract}(\mathsf{td}, \mathsf{v}, I), \\ & \textit{define } x_{\mathsf{imp}} \in \{0,1\}^N : \\ & \forall j \in I, (x_{\mathsf{imp}})_j = b_j; \forall j \in [N] \setminus I, (x_{\mathsf{imp}})_j = 0 \end{array}\right] \leq \mathsf{negl}(\lambda).$$

Theorem 3.8 ([KLVW22])**.** *Assuming the Learning with Errors (*LWE*) assumption, there exists a flexible* RAM SNARG*.*

3.4 Homomorphic Encryption with Ciphertext Compression

Below we recall the notion of a homomorphic encryption scheme, and define ciphertext compression algorithms. These ciphertext compression algorithms are inspired by the rate-1 homomorphic encryption scheme of Brakerski et al. [BDGM19], and are essential in our construction of a rate-1 fully-local somewhere extractable hash family (defined in section 4). Our compressed ciphertexts have desired locality properties that are essential to our fully-local somewhere extractable hash family.

Syntax. A homomorphic encryption scheme HE for a circuit class $\mathcal{C}$ and message space $\{0,1\}$ consists of the following PPT algorithms:

$\mathsf{Gen}(1^\lambda) \to (\mathsf{pk}, \mathsf{sk})$. This is a setup algorithm takes as input the security parameter λ, and outputs a public-secret key pair $(\mathsf{pk}, \mathsf{sk})$.

$\mathsf{Enc}(\mathsf{pk}, b) \to \mathsf{ct}$. The encryption algorithm takes as input a public key pk, message bit $b \in \{0,1\}$, and outputs a ciphertext ct.

$\mathsf{Eval}(\mathsf{pk}, C, (\mathsf{ct}_1, \ldots, \mathsf{ct}_n)) \to (\mathsf{ct}'_1, \ldots, \mathsf{ct}'_\ell)$. The evaluation algorithm is a deterministic algorithm that takes as input a public key pk, description of a circuit $C \in \mathcal{C}$, and n ciphertexts ct_i for $i \in [n]$, where n is the input length of the circuit C. It outputs a sequence of evaluated ciphertexts $(\mathsf{ct}'_1, \ldots, \mathsf{ct}'_\ell)$, where ℓ is the output length of the circuit C.

$\mathsf{Dec}(\mathsf{sk}, \mathsf{ct}) \to b$. The decryption algorithm is a deterministic algorithm that takes as input a secret key sk, a (possibly evaluated) ciphertext ct, and outputs a message bit b.

(Throughout the sequel, we naturally define the decryption algorithm to take as input a sequence of ciphertexts, and output a sequence of message bits as output.)

Definition 3.9 (Correctness and Compactness of HE)**.** *The encryption scheme* HE *is said to be* correct *if for any security parameter* $\lambda \in \mathbb{N}$*, any circuit* $C \in \mathcal{C}$ *with input and output lengths* n *and* ℓ *(respectively), and any sequence of* n *messages* $b_1, \ldots, b_n \in \{0,1\}$*,*

$$\Pr[\mathsf{Dec}(\mathsf{sk}, \mathsf{Eval}(\mathsf{pk}, C, (\mathsf{ct}_1, \ldots, \mathsf{ct}_n))) = C(b_1, \ldots, b_n)] = 1$$

where the probability is over $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda)$ *and* $\mathsf{ct}_i \leftarrow \mathsf{Enc}(\mathsf{pk}, b_i)$ *for* $i \in [n]$*.*

The encryption scheme HE *is said to be* compact *if the bit-length of the evaluated ciphertext is at most* $\mathsf{poly}(\lambda, \ell)$*, i.e. its size does not depend on circuit size.*

Compressing ciphertexts. In our construction of a rate-1 fully-local SEH (in Section 4), we use a homomorphic encryption scheme that has a "compression" algorithm which takes a set of ℓ evaluated ciphertexts (each encrypting a single bit) and compresses them into a single rate-1 ciphertext encrypting these ℓ bits. Moreover, we require this "compression" algorithm to have a specific form which is needed in order to construct the rate-1 fully-local SEH scheme.

The compression algorithm consists of three parts: $\mathsf{Compress}_1, \mathsf{LinEval}, \mathsf{Compress}_2$, which we elaborate on below.[5] We note that in order to compress ℓ ciphertexts we need to generate new "compression" keys (that depend on ℓ), via an algorithm $\mathsf{CompGen}$ and we need a new decryption algorithm called $\mathsf{CompDec}$. We require this decryption algorithm to be local (as defined below).

$\mathsf{CompGen}(\mathsf{pk}, \mathsf{sk}, 1^\ell) \to (\mathsf{pk}_\mathbf{c}, \mathsf{sk}_\mathbf{c})$. The PPT compression key generator algorithm takes as input a public-secret key pair $(\mathsf{pk}, \mathsf{sk})$, and the compression parameter ℓ, and it outputs a new public-secret key pair $(\mathsf{pk}_\mathbf{c}, \mathsf{sk}_\mathbf{c})$ that enables ciphertext compression.

(Here ℓ should be regarded as the maximum number of one-bit ciphertexts that can be compressed to a rate-1 encryption.)

[5] We note that the properties we specify below are quite specific. We could have specified more general properties but we chose simplicity over generality.

$\mathsf{Compress}_1(\mathsf{pk}_{\mathbf{c}}, \mathsf{ct}, \ell, i) \to \mathsf{ct}_{\mathbf{c}_1}$. The first part of the compression algorithm is a deterministic poly-time algorithm that takes as input a compression public key $\mathsf{pk}_{\mathbf{c}}$, a (non-compressed) HE ciphertext ct encrypting a single bit b, a length value $\ell \in \mathbb{N}$, and an index $i \in [\ell]$. It outputs a "processed" ciphertext $\mathsf{ct}_{\mathbf{c}_1}$, which encrypts the ℓ-bit vector $b \cdot e_i \in \{0,1\}^\ell$ where $e_i \in \{0,1\}^\ell$ is 1 on its i'th coordinate and zero on all other coordinates.

This processed ciphertext has the form

$$\mathsf{ct}_{\mathbf{c}_1} = (\mathsf{sub\text{-}ct}_0, \mathsf{sub\text{-}ct}_1, \ldots, \mathsf{sub\text{-}ct}_\ell),$$

where $\mathsf{sub\text{-}ct}_0$ is a "preamble" consisting of $\mathsf{poly}(\lambda)$ field elements in a finite field $\mathbb{F}_q$ of size $q \leq 2^\lambda$, and each $\mathsf{sub\text{-}ct}_i$ for $i \in [\ell]$ consists of a single element in $\mathbb{F}_q$.[6]

$\mathsf{LinEval}(\mathsf{pk}, \mathsf{ct}_{\mathbf{c}_1}^{(1)}, \ldots, \mathsf{ct}_{\mathbf{c}_1}^{(\ell)}) \to \mathsf{ct}'_{\mathbf{c}_1}$. This linear evaluation algorithm is a deterministic poly-time algorithm that takes as input a public key pk, and ℓ processed ciphertexts (each encrypting an ℓ-bit vector), and outputs a single processed ciphertext (encrypting an ℓ-bit vector).

This algorithm simply adds the ℓ ciphertexts $\mathsf{ct}_{\mathbf{c}_1}^{(1)}, \ldots, \mathsf{ct}_{\mathbf{c}_1}^{(\ell)}$ coordinate-wise over $\mathbb{F}_q$. Importantly, if each $\mathsf{ct}_{\mathbf{c}_1}^{(i)}$ encrypts the vector $b_i \cdot e_i \in \{0,1\}^\ell$ then the output ciphertext $\mathsf{ct}'_{\mathbf{c}_1}$ encrypts the vector $(b_1, \ldots, b_\ell)$.

$\mathsf{Compress}_2(\mathsf{ct}_{\mathbf{c}_1}) \to \mathsf{ct}_{\mathbf{c}}$. The final compression algorithm is a deterministic poly-time algorithm that takes as input a processed ciphertext $\mathsf{ct}_{\mathbf{c}_1}$ and outputs a fully compressed ciphertext $\mathsf{ct}_{\mathbf{c}}$.

Importantly, this algorithm can be executed locally in the sense that one can run it on each sub-ciphertext separately. Namely, $\mathsf{Compress}_2(\mathsf{ct}_{\mathbf{c}_1})$ can be computed as follows:

1. Parse $\mathsf{ct}_{\mathbf{c}_1} = (\mathsf{sub\text{-}ct}_0, \mathsf{sub\text{-}ct}_1, \ldots, \mathsf{sub\text{-}ct}_\ell)$.
2. For every $j \in [\ell]$, compute $\mathsf{sub\text{-}ct}_{\mathbf{c},j} = \mathsf{Compress}_2(\mathsf{sub\text{-}ct}_j) \in \{0,1\}$.
   (For the sake of simplicity, we are overloading the notation of $\mathsf{Compress}_2$.)
3. Output $\mathsf{ct}_{\mathbf{c}} = (\mathsf{sub\text{-}ct}_0, \mathsf{sub\text{-}ct}_{\mathbf{c},1} \ldots, \mathsf{sub\text{-}ct}_{\mathbf{c},\ell})$.

$\mathsf{CompDec}(\mathsf{sk}_{\mathbf{c}}, \mathsf{ct}_{\mathbf{c}}) \to (b_1, \ldots, b_\ell)$. This is a deterministic poly-time decryption algorithm that decrypts compressed ciphertexts. It takes as input a compression secret key $\mathsf{sk}_{\mathbf{c}}$ and a compressed ciphertext $\mathsf{ct}_{\mathbf{c}}$ and it outputs its decryption $(b_1, \ldots, b_\ell)$.

Importantly, this algorithm is local in the following sense: Parsing $\mathsf{ct}_{\mathbf{c}} = (\mathsf{sub\text{-}ct}_0, \mathsf{sub\text{-}ct}_{\mathbf{c},1} \ldots, \mathsf{sub\text{-}ct}_{\mathbf{c},\ell})$ (and overloading notation),

$$\mathsf{CompDec}(\mathsf{sk}_{\mathbf{c}}, \mathsf{ct}_{\mathbf{c}}) = (\mathsf{CompDec}(\mathsf{sk}_{\mathbf{c}}, \mathsf{sub\text{-}ct}_0, \mathsf{sub\text{-}ct}_1), \ldots, \mathsf{CompDec}(\mathsf{sk}_{\mathbf{c}}, \mathsf{sub\text{-}ct}_0, \mathsf{sub\text{-}ct}_\ell)) .$$

Definition 3.10. *[Correctness of Compressed Encryption Scheme] A homomorphic encryption scheme* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval}, \mathsf{Dec})$ *for a circuit class* $\mathcal{C}$*, with compression algorithms*

$$(\mathsf{CompGen}, \mathsf{Compress}_1, \mathsf{LinEval}, \mathsf{Compress}_2, \mathsf{CompDec})$$

as defined above, is said to be correct and compact if the encryption scheme is correct and compact (as per definition 3.9), and there exists a negligible function negl *such that for any security parameter*

[6] We note that $\mathbb{F}_q$ is the underlying field used in this encryption scheme.

$\lambda \in \mathbb{N}$, *parameters* $\ell, n \leq 2^\lambda$, *key pair* $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda)$, *compressed key pair* $(\mathsf{pk}_\mathbf{c}, \mathsf{sk}_\mathbf{c}) \leftarrow \mathsf{CompGen}(\mathsf{pk}, \mathsf{sk}, 1^\ell)$, *any circuit* $C \in \mathcal{C}$ *with input and output lengths* n *and* ℓ *respectively, any sequence of* n *bit messages* $b_1, \ldots, b_n \in \{0,1\}$, *ciphertexts* $\mathsf{ct}_i \leftarrow \mathsf{Enc}(\mathsf{pk}, b_i)$ *for* $i \in [n]$, *evaluated ciphertext*

$$(\mathsf{ct}'_1, \ldots, \mathsf{ct}'_\ell) = \mathbf{ct}' \leftarrow \mathsf{Eval}(\mathsf{pk}, C, (\mathsf{ct}_1, \ldots, \mathsf{ct}_n)),$$

partially compressed ciphertexts

$$\left(\mathsf{sub\text{-}ct}_0^{(i)}, \mathsf{sub\text{-}ct}_1^{(i)}, \ldots, \mathsf{sub\text{-}ct}_\ell^{(i)}\right) = \mathsf{ct}_{\mathbf{c}_1}^{(i)} \leftarrow \mathsf{Compress}_1(\mathsf{pk}_\mathbf{c}, \mathsf{ct}'_i, \ell, i)$$

for $i \in [\ell]$, *letting*

$$\mathsf{ct}'_{\mathbf{c}_1} = \mathsf{LinEval}(\mathsf{pk}, \mathsf{ct}_{\mathbf{c}_1}^{(1)}, \ldots, \mathsf{ct}_{\mathbf{c}_1}^{(\ell)}),$$

and $\mathsf{ct}_\mathbf{c} = \mathsf{Compress}_2(\mathsf{ct}'_{\mathbf{c}_1})$, *it holds that*

$$\Pr[\mathsf{CompDec}(\mathsf{sk}_\mathbf{c}, \mathsf{ct}_\mathbf{c}) = C(b_1, \ldots, b_n)] \geq 1 - \mathsf{negl}(\lambda).$$

where the probability is over the sampling of all the ciphertexts above (including the ciphertexts $\mathsf{ct}_1, \ldots, \mathsf{ct}_n$*).*

Security. For security of our encryption scheme with compression algorithms, we consider an extended notion of semantic security, where the attacker gets both the regular and compression public keys, and it still can not distinguish between two honestly computed ciphertexts.

Definition 3.11 (Semantic Security)**.** *A homomorphic encryption scheme* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval}, \mathsf{Dec})$ *with compression algorithms* $(\mathsf{CompGen}, \mathsf{Compress}_1, \mathsf{LinEval}, \mathsf{Compress}_2, \mathsf{LocDec}_2)$ *is said to be secure if for every poly-size adversary* $\mathcal{A}$ *there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for all* $\lambda \in \mathbb{N}$, *the following holds:*

$$\Pr\left[\mathcal{A}(\mathsf{pk}, \mathsf{pk}_\mathbf{c}, \mathsf{ct}) = b : \begin{array}{l}(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda),\ 1^\ell \leftarrow \mathcal{A}(\mathsf{pk}) \\ (\mathsf{pk}_\mathbf{c}, \mathsf{sk}_\mathbf{c}) \leftarrow \mathsf{CompGen}(\mathsf{pk}, \mathsf{sk}, 1^\ell) \\ b \leftarrow \{0,1\},\ \mathsf{ct} \leftarrow \mathsf{Enc}(\mathsf{pk}, b)\end{array}\right] \leq \frac{1}{2} + \mathsf{negl}(\lambda).$$

Later in appendix A, we construct such a homomorphic encryption scheme with compression algorithms by relying on the rate-1 FHE construction by Brakerski et al. [BDGM19]. Below we state the main theorem that we prove in appendix A.

Theorem 3.12. *Assuming the Learning with Errors* (LWE) *assumption, there exists a homomorphic encryption scheme* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval}, \mathsf{Dec})$ *for all circuits of depth* $\mathsf{poly}(\lambda)$ *(where the public key grows with* $\mathsf{poly}(\lambda)$*), with compression algorithms* $(\mathsf{CompGen}, \mathsf{Compress}_1, \mathsf{LinEval}, \mathsf{Compress}_2, \mathsf{CompDec})$ *satisfying all the properties defined above, including correctness, compactness, and security (see Definitions 3.9 to 3.11).*

4 Rate-1 Fully-Local SEH (flSEH) Families

4.1 Definition

Syntax. A rate-1 fully-local somewhere extractable hash family consists of the following polynomial time algorithms:

$\mathsf{Gen}(1^\lambda, N, I) \to (\mathsf{crs}_\ell, \mathsf{crs}_s, \mathsf{td})$. This is a probabilistic poly-time setup algorithm that takes as input security parameter 1^λ in unary, message length N, and subset $I \subset [N]$. It runs in time $\mathsf{poly}(\lambda, |I|, \log N)$ and outputs a long crs $\mathsf{crs}_\ell \in \{0,1\}^{|I| \cdot \mathsf{poly}(\lambda)}$, a short crs $\mathsf{crs}_s \in \{0,1\}^{\mathsf{poly}(\lambda)}$, and a trapdoor td.

Moreover, if the set I consists of consecutive indices $I = \{i^* + 1, \ldots, i^* + m\}$ then Gen runs in time $\mathsf{poly}(\lambda, \log N, \log m)$ and outputs $(\mathsf{crs}, \mathsf{td})$ both of size $\mathsf{poly}(\lambda, \log N, \log m)$.[7]

$\mathsf{Hash}(\mathsf{crs}_\ell, \mathsf{crs}_s, \boldsymbol{x}) \to (\mathsf{v}, \mathsf{rt})$. This is a deterministic poly-time hash algorithm that takes as input (long) crs_ℓ, (short) crs crs_s, and message $\boldsymbol{x} \in \{0,1\}^N$, and outputs a long hash value $\mathsf{v} \in \{0,1\}^{|I| + \mathsf{poly}(\lambda)}$ and a short digest $\mathsf{rt} \in \{0,1\}^{\mathsf{poly}(\lambda)}$.

Remark 4.1. We note that we can always assume w.l.o.g. that crs_ℓ contains crs_s, in which case Hash can take as input only crs_ℓ. However, in our construction (in Section 4.2) crs_ℓ does not include crs_s and hence we give Hash both crs_ℓ and crs_s.

$\mathsf{Validate}(\mathsf{crs}_s, \mathsf{v}, \mathsf{rt}) \to 0/1$. This is a deterministic poly-time validation algorithm that takes as input (short) crs_s, hash value v, and (short) digest rt, and outputs 1 (accept) or 0 (reject).

$\mathsf{Open}(\mathsf{crs}_\ell, \mathsf{crs}_s, \boldsymbol{x}, i) \to (x_i, \rho)$. This is a deterministic poly-time opening algorithm that takes as input (long) crs_ℓ, (short) crs crs_s, message $\boldsymbol{x} \in \{0,1\}^N$, and index $i \in [N]$, and outputs a bit $x_i \in \{0,1\}$ and a local opening ρ of length $\mathsf{poly}(\lambda, \log N)$.

$\mathsf{Verify}(\mathsf{crs}_s, \mathsf{rt}, i, b, \rho) \to 0/1$. This is a deterministic poly-time verification algorithm that takes as input (short) crs_s, (short) digest rt (produced by Hash), index $i \in [N]$, bit b, and local opening ρ, and outputs 1 (accept) or 0 (reject).[8]

$\mathsf{Extract}(\mathsf{td}, \mathsf{v}) \to u$. This is a deterministic poly-time extraction algorithm that takes as input trapdoor td, and hash value v, and outputs a string $u \in \{0,1\}^{|I|}$.

A rate-1 fISEH family is required to satisfy similar properties to those of a SEH family adapted to the fISEH setting. The adapted properties are described formally below.

Definition 4.1 (rate-1 fISEH)**.** *A rate-1* fISEH *family is required to have the following properties:*

Index hiding. *For any poly-size adversary $\mathcal{A}$, any polynomial $N = N(\lambda)$, and any subsets $I_0, I_1 \in [N]$ such that $|I_0| = |I_1|$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that for every $\lambda \in \mathbb{N}$,*

$$\Pr\left[\mathcal{A}(\mathsf{crs}_\ell, \mathsf{crs}_s) = b : \begin{array}{l} b \leftarrow \{0,1\} \\ (\mathsf{crs}_\ell, \mathsf{crs}_s, \mathsf{td}) \leftarrow \mathsf{Gen}(1^\lambda, N, I_b) \end{array}\right] \leq \frac{1}{2} + \mathsf{negl}(\lambda).$$

Completeness. *For any $\lambda \in \mathbb{N}$, any $N \leq 2^\lambda$, any subset $I \subset [N]$, any index $i \in [N]$, and any $\boldsymbol{x} \in \{0,1\}^N$,*

$$\Pr\left[\begin{array}{l} b = x_i \\ \wedge\ \mathsf{Verify}(\mathsf{crs}_s, \mathsf{rt}, i, b, \rho) = 1 \\ \wedge\ \mathsf{Validate}(\mathsf{crs}_s, \mathsf{v}, \mathsf{rt}) = 1 \end{array} : \begin{array}{l} (\mathsf{crs}_\ell, \mathsf{crs}_s, \mathsf{td}) \leftarrow \mathsf{Gen}(1^\lambda, N, I), \\ (\mathsf{v}, \mathsf{rt}) = \mathsf{Hash}(\mathsf{crs}_\ell, \mathsf{crs}_s, \boldsymbol{x}), \\ (b, \rho) = \mathsf{Open}(\mathsf{crs}_\ell, \mathsf{crs}_s, \boldsymbol{x}, i), \end{array}\right] = 1.$$

[7] In this case, there is no reason to distinguish between crs_ℓ and crs_s since crs is short.

[8] Note that the run-time of Verify is $\mathsf{poly}(\lambda)$ assuming $N \leq 2^\lambda$.

Somewhere statistically binding w.r.t. opening. *For any $\lambda \in \mathbb{N}$, any $N \leq 2^\lambda$, any subset $I \subseteq [N]$, any index $i \in I$, and any (all powerful) adversary $\mathcal{A}$, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for every $\lambda \in \mathbb{N}$,*

$$\Pr\left[\begin{array}{ll} \mathsf{Verify}(\mathsf{crs}_s, \mathsf{rt}, i, b, \rho) = 1 & (\mathsf{crs}_\ell, \mathsf{crs}_s, \mathsf{td}) \leftarrow \mathsf{Gen}(1^\lambda, N, I), \\ \wedge\ \mathsf{Validate}(\mathsf{crs}_s, \mathsf{v}, \mathsf{rt}) = 1 \quad : & (\mathsf{v}, \mathsf{rt}, b, \rho) \leftarrow \mathcal{A}(\mathsf{crs}_\ell, \mathsf{crs}_s), \\ \wedge\ b \neq b_i & (b_j)_{j \in I} = \mathsf{Extract}(\mathsf{td}, \mathsf{v}) \end{array}\right] \leq \mathsf{negl}(\lambda).$$

Extraction correctness. *For any $\lambda \in \mathbb{N}$, any $N \leq 2^\lambda$, any subset $I \subseteq [N]$, and any $\boldsymbol{x} \in \{0,1\}^N$, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for every $\lambda \in \mathbb{N}$,*

$$\Pr\left[\begin{array}{ll} x_I \neq \mathsf{Extract}(\mathsf{td}, \mathsf{v}) \quad : & (\mathsf{crs}_\ell, \mathsf{crs}_s, \mathsf{td}) \leftarrow \mathsf{Gen}(1^\lambda, N, I) \\ & \mathsf{v} \leftarrow \mathsf{Hash}(\mathsf{crs}_\ell, \mathsf{crs}_s, \boldsymbol{x}) \end{array}\right] \leq \mathsf{negl}(\lambda).$$

4.2 Construction

Our construction of a flSEH family uses the following ingredients:

1. A rate-1 linearly homomorphic or levelled fully homomorphic encryption scheme

   $$(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval}, \mathsf{Dec})$$

   with compression algorithms

   $$(\mathsf{CompGen}, \mathsf{Compress}_1, \mathsf{LinEval}, \mathsf{Compress}_2, \mathsf{CompDec})$$

   (see Section 3.4). Recall that each ciphertext is a vector over a finite field, denoted by $\mathbb{F}_q$.

2. A somewhere extractable hash family

   $$\mathsf{SEH} = (\mathsf{SEH.Gen}, \mathsf{SEH.Hash}, \mathsf{SEH.Open}, \mathsf{SEH.Verify}, \mathsf{SEH.Extract})$$

   constructed w.r.t. the rate-1 encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval}, \mathsf{Dec})$ from Item 1 (see Section 3.1 and Lemma 3.2).

3. Two flexible RAM SNARG schemes with partial input soundness (see Section 3.3):

   $$(\mathsf{RAM}_0.\mathsf{Gen}, \mathsf{RAM}_0.\mathsf{Digest}, \mathsf{RAM}_0.\mathcal{P}, \mathsf{RAM}_0.\mathcal{V})$$

   and

   $$(\mathsf{RAM}_1.\mathsf{Gen}, \mathsf{RAM}_1.\mathsf{Digest}, \mathsf{RAM}_1.\mathcal{P}, \mathsf{RAM}_1.\mathcal{V}),$$

   where the first is w.r.t. the RAM machine $\mathcal{R}_0$ that runs in time T_0 and the second is w.r.t. the RAM machine $\mathcal{R}_1$ that runs in time T_1.

   (a) The RAM machine $\mathcal{R}_0$ is associated with the encryption scheme from Item 1. It takes as input a sequence of ciphertexts $(\mathsf{ct}_1, \ldots, \mathsf{ct}_m)$ each encrypting a single bit, the corresponding public key pk, indices $j \in [m]$ and $k \in [m']$, where m' denotes the number of field elements in the vector $\mathsf{Compress}_1(\mathsf{pk}, \mathsf{ct}_j, m, j)$, and a field element $c \in \mathbb{F}_q$, and outputs 1 if and only if c is the k'th element of $\mathsf{Compress}_1(\mathsf{pk}_\mathbf{c}, \mathsf{ct}_j, m, j)$.
   Recall that $m' = m + \mathsf{poly}(\lambda)$. We let

   $$\Delta(\lambda) \triangleq m' - m = \mathsf{poly}(\lambda) \tag{1}$$

(b) The RAM machine $\mathcal{R}_1$ is specified in Item 6b of the Hash algorithm construction below.

4. Two somewhere extractable BARG schemes (see Section 3.2)

$$(\mathsf{seBARG}_0.\mathsf{Gen}, \mathsf{seBARG}_0.\mathcal{P}, \mathsf{seBARG}_0.\mathcal{V})$$

and

$$(\mathsf{seBARG}_1.\mathsf{Gen}, \mathsf{seBARG}_1.\mathcal{P}, \mathsf{seBARG}_1.\mathcal{V})$$

where the first is for the NP language $\mathcal{L}_0$, specified in Item 8 of the Hash algorithm construction below, and the second is for the NP language $\mathcal{L}_1$, specified in Item 5 of the Open algorithm construction below.

We are now ready to define our flSEH family.

$\mathsf{Gen}(1^\lambda, N, I) \to (\mathsf{crs}_\ell, \mathsf{crs}_s, \mathsf{td})$. This poly-time algorithm does the following:

In what follows we let $m = |I|$ and we assume without loss of generality that m is a power of 2, and for every $t \in [0, \log m]$ we denote by $m_t = m/2^t$.

1. Generate $(\mathsf{hk}', \mathsf{td}') \leftarrow \mathsf{SEH.Gen}(1^\lambda, N, I)$.
2. Parse $\mathsf{hk}' = (\mathsf{pk}, \mathsf{hk}'_1, \ldots, \mathsf{hk}'_m)$.
   Recall that $\mathsf{td}' = \mathsf{sk}$ where sk is the secret key corresponding to pk, and $n \triangleq |\mathsf{hk}'_j| \leq \mathsf{poly}(\lambda)$ for every $j \in [m]$ (see Lemma 3.2). In addition, recall that each hash value corresponding to hk' consists of $m = |I|$ ciphertexts w.r.t. pk, each encrypting a single bit, where the j'th ciphertext is a function of pk and hk'_j. By padding, we can assume w.l.o.g. that the size of each of these ciphertexts is also n.
3. Generate $(\mathsf{pk}_\mathbf{c}, \mathsf{sk}_\mathbf{c}) \leftarrow \mathsf{CompGen}(\mathsf{pk}, \mathsf{sk}, 1^m)$.
4. Set arbitrarily $\alpha \in [m]$. For example, set $\alpha = 1$.
5. Generate $(\mathsf{hk}, \mathsf{td}) \leftarrow \mathsf{SEH.Gen}(1^\lambda, m \cdot n, I_\alpha)$, where $I_\alpha = \{(\alpha - 1) \cdot n + 1, \ldots, \alpha \cdot n\}$.
6. Let $\mathsf{h}' = \mathsf{SEH.Hash}\left(\mathsf{hk}, (\mathsf{hk}'_1, \ldots, \mathsf{hk}'_m)\right)$.
7. For every $t \in [0, \log m]$, generate $(\mathsf{hk}_t, \mathsf{td}_t) \leftarrow \mathsf{SEH.Gen}(1^\lambda, m' \cdot m_t \cdot n_t, J_{t,\alpha})$, where the value n_t and the set of coordinates $J_{t,\alpha} \subseteq [m' \cdot m_t \cdot n_t]$ of size $\mathsf{poly}(\lambda)$ are specified in Item 6a of the Hash algorithm construction below.
8. Generate $\mathsf{RAM}_0.\mathsf{crs} \leftarrow \mathsf{RAM}_0.\mathsf{Gen}(1^\lambda, T_0)$.
9. Generate $\mathsf{RAM}_1.\mathsf{crs} \leftarrow \mathsf{RAM}_1.\mathsf{Gen}(1^\lambda, T_1)$.
10. Generate $(\mathsf{seBARG}_0.\mathsf{crs}, \mathsf{seBARG}_0.\mathsf{td}) \leftarrow \mathsf{seBARG}_0.\mathsf{Gen}(1^\lambda, m', n', S_\alpha)$, where recall that $m' = \Delta(\lambda) + m$ is the number of field elements in the vector $\mathsf{Compress}_1(\mathsf{pk}, \mathsf{ct}_j, m, j)$, $n' \leq \mathsf{poly}(\lambda)$ is specified in Item 8 of the Hash algorithm construction below, and $S_\alpha = \{1, \ldots, \Delta(\lambda)\} \cup \{\Delta(\lambda) + \alpha\} \subseteq [m']$.
11. Generate $(\mathsf{seBARG}_1.\mathsf{crs}, \mathsf{seBARG}_1.\mathsf{td}) \leftarrow \mathsf{seBARG}_1.\mathsf{Gen}(1^\lambda, m, n'', \alpha)$, where $n'' \leq \mathsf{poly}(\lambda)$ is specified in Item 5 of the Open algorithm construction below.
12. Let $\mathsf{crs}_\ell = \mathsf{hk}'$ and

$$\mathsf{crs}_s = \left(\mathsf{pk}, \mathsf{pk}_\mathbf{c}, \mathsf{hk}, \mathsf{h}', \{\mathsf{hk}_t\}_{t=0}^{\log m}, \mathsf{RAM}_0.\mathsf{crs}, \mathsf{RAM}_1.\mathsf{crs}, \mathsf{seBARG}_0.\mathsf{crs}, \mathsf{seBARG}_1.\mathsf{crs}\right).$$

13. Output $(\mathsf{crs}_\ell, \mathsf{crs}_s, \mathsf{td}^*)$, where $\mathsf{td}^* = \mathsf{sk}_{\mathbf{c}}$.

Remark 4.2. Note that if $I = \{i^*+1, \ldots, i^*+m\}$ then hk' can be computed efficiently from pk and an encryption of i^*, so Gen runs in time $\mathsf{poly}(\lambda, \log N)$ (see Lemma 3.2).

$\mathsf{Hash}(\mathsf{crs}_\ell, \mathsf{crs}_s, \boldsymbol{x}) \to (\mathsf{v}, \mathsf{rt})$. This poly-time algorithm does the following:

1. Parse $\mathsf{crs}_\ell = \mathsf{hk}' = (\mathsf{pk}, \mathsf{hk}'_1, \ldots, \mathsf{hk}'_m)$ and

$$\mathsf{crs}_s = \left(\mathsf{pk}, \mathsf{pk}_{\mathbf{c}}, \mathsf{hk}, \mathsf{h}', \{\mathsf{hk}_t\}_{t=0}^{\log m}, \mathsf{RAM}_0.\mathsf{crs}, \mathsf{RAM}_1.\mathsf{crs}, \mathsf{seBARG}_0.\mathsf{crs}, \mathsf{seBARG}_1.\mathsf{crs}\right).$$

2. Compute

$$(\mathsf{ct}_1, \ldots, \mathsf{ct}_m) = \mathsf{SEH.Hash}\left(\mathsf{hk}', \boldsymbol{x}\right).$$

   Denoting by $I = \{i_1, \ldots, i_m\} \subseteq [N]$, note that ct_j is a ciphertext encrypting the bit x_{i_j} w.r.t. pk (see Lemma 3.2).

3. Compute

$$\mathsf{h} = \mathsf{SEH.Hash}\left(\mathsf{hk}, (\mathsf{ct}_1, \ldots, \mathsf{ct}_m)\right).$$

   Note that $|\mathsf{h}| \leq \mathsf{poly}(\lambda)$.

4. Compute v which is the compressed ciphertext corresponding to $(\mathsf{ct}_1, \ldots, \mathsf{ct}_m)$ and $\mathsf{pk}_{\mathbf{c}}$, as follows:
   (a) For every $j \in [m]$ compute

$$(\mathsf{ct}_j)_{\mathbf{c}_1} = \mathsf{Compress}_1\left(\mathsf{pk}_{\mathbf{c}}, \mathsf{ct}_j, m, j\right).$$

   Note that $(\mathsf{ct}_j)_{\mathbf{c}_1}$ is a ciphertext encrypting the vector $x_{i_j} \cdot e_j \in \{0,1\}^m$.
   (b) Compute

$$\mathsf{ct}_{\mathbf{c}_1} = \mathsf{LinEval}\left(\mathsf{pk}, (\mathsf{ct}_1)_{\mathbf{c}_1}, \ldots, (\mathsf{ct}_m)_{\mathbf{c}_1}\right).$$

   (c) Compute

$$\mathsf{v} = \mathsf{Compress}_2(\mathsf{ct}_{\mathbf{c}_1}).$$

   Note that $\mathsf{v} = (\mathsf{v}_1, \ldots, \mathsf{v}_{m'})$ is a ciphertext encrypting the string $x_I \in \{0,1\}^m$, where $\mathsf{v}_k \in \mathbb{F}_q$ for every $k \in [\Delta(\lambda)]$, and $\mathsf{v}_k \in \{0,1\}$ for every $k \in [\Delta(\lambda)+1, m']$.[9]

   In Item 5-Item 9 below we compute a proof that h and v are consistent.

5. For every $j \in [m]$ and $k \in [m']$ let

$$c^{(0)}_{j,k} = k\text{-th element of } (\mathsf{ct}_j)_{\mathbf{c}_1}, \text{ and}$$

$$\pi^{(0)}_{j,k} = \mathsf{RAM}_0.\mathcal{P}\left(\mathsf{RAM}_0.\mathsf{crs}, \mathsf{hk}, (\mathsf{ct}_1, \ldots, \mathsf{ct}_m), (\mathsf{pk}, j, k, c^{(0)}_{j,k})\right).$$

   Note that $\pi^{(0)}_{j,k}$ is a RAM proof that $c^{(0)}_{j,k}$ is the k'th element of $\mathsf{Compress}_1(\mathsf{pk}_{\mathbf{c}}, \mathsf{ct}_j, m, j)$.

6. For $t = 0, \ldots, \log m$:

[9] Using our notation in Section 3.4, $\mathsf{sub\text{-}ct}_0 = (\mathsf{v}_1, \ldots, \mathsf{v}_{\Delta(\lambda)}) \in \mathbb{F}_q^{\Delta(\lambda)}$ and $\mathsf{sub\text{-}ct}_{\mathbf{c},j} = \mathsf{v}_{\Delta(\lambda)+j} \in \{0,1\}$ for every $j \in [m]$.

(a) Let

$$X^{(t)} = \left\{c_{j,k}^{(t)}, \pi_{j,k}^{(t)}\right\}_{j\in[m_t],k\in[m']} \tag{2}$$

and compute

$$\mathsf{h}_t = \mathsf{SEH.Hash}\left(\mathsf{hk}_t, X^{(t)}\right)$$

where hk_t is defined to hash messages of length $m' \cdot m_t \cdot n_t$ and be statistically binding on the coordinates in $J_{t,\alpha}$, where n_t and $J_{t,\alpha}$ are defined as follows:

$$n_t = \log q + \left|\pi_{j,k}^{(t)}\right|$$

where $\left|\pi_{j,k}^{(t)}\right|$ is the number of bits in the proof $\pi_{j,k}^{(t)}$, and $J_{t,\alpha}$ is defined by first letting

$$J'_{t,\alpha} = \left\{\lceil \alpha/2^t \rceil - 1, \lceil \alpha/2^t \rceil, \lceil \alpha/2^t \rceil + 1\right\} \cap [m_t].$$

and letting $J_{t,\alpha}$ be the set of coordinates corresponding to $\left(c_{j,k}^{(t)}, \pi_{j,k}^{(t)}\right)$ for $j \in J'_{t,\alpha}$ and $k \in S_\alpha$.[10] Note that $|J_{t,\alpha}| \leq \mathsf{poly}(\lambda)$.

(b) If $t < \log m$: For every $j \in [m_{t+1}]$ and $k \in [m']$, compute

$$c_{j,k}^{(t+1)} = c_{2j-1,k}^{(t)} + c_{2j,k}^{(t)},$$

where addition is over the finite field $\mathbb{F}_q$ used in the underlying encryption scheme (see Definition 3.10), and compute

$$\pi_{j,k}^{(t+1)} = \mathsf{RAM}_1.\mathcal{P}\left(\mathsf{RAM}_1.\mathsf{crs}, \mathsf{hk}_t, X^{(t)}, \left(\mathsf{hk}, \mathsf{h}, \{\mathsf{hk}_\ell, \mathsf{h}_\ell\}_{\ell=0}^{t-1}, \mathsf{pk}, j, k, c_{j,k}^{(t+1)}\right)\right).$$

RAM_1 is defined w.r.t. the RAM machine $\mathcal{R}_1$ that takes as input $X^{(t)}$ as in Equation (2), hash keys $(\mathsf{hk}, \mathsf{hk}_0, \ldots, \mathsf{hk}_{t-1})$, hash values $(\mathsf{h}, \mathsf{h}_0, \mathsf{h}_1, \ldots, \mathsf{h}_{t-1})$, the public key pk defined by hk', indices $j \in [m_{t+1}]$ and $k \in [m']$, and field element $c_{j,k}^{(t+1)}$, and outputs 1 if and only if the following conditions are satisfied:

i. $c_{j,k}^{(t+1)} = c_{2j-1,k}^{(t)} + c_{2j,k}^{(t)}$ over $\mathbb{F}_q$.

ii. For every $b \in \{0,1\}$, $\pi_{2j-b,k}^{(t)}$ is a valid proof. Namely, if $t = 0$, then

$$\mathsf{RAM}_0.\mathcal{V}\left(\mathsf{RAM}_0.\mathsf{crs}, \mathsf{hk}, \mathsf{h}, \left(\mathsf{pk}, 2j-b, k, c_{2j-b,k}^{(t)}\right), \pi_{2j-b,k}^{(t)}\right) = 1$$

and if $t \geq 1$, then

$$\mathsf{RAM}_1.\mathcal{V}\left(\mathsf{RAM}_1.\mathsf{crs}, \mathsf{hk}_{t-1}, \mathsf{h}_{t-1}, \left(\mathsf{hk}, \mathsf{h}, \{\mathsf{hk}_\ell, \mathsf{h}_\ell\}_{\ell=0}^{t-2}, \mathsf{pk}, 2j-b, k, c_{2j-b,k}^{(t)}\right), \pi_{2j-b,k}^{(t)}\right) = 1.$$

Note that the RAM proof $\pi_{j,k}^{(t+1)}$ certifies the correctness of the k'th element of $c_j^{(t+1)}$ w.r.t. $(\mathsf{h}, \mathsf{h}_0, \mathsf{h}_1, \ldots, \mathsf{h}_t, \mathsf{pk})$.

[10] Recall that $m_t = 2^{\log m - t}$ and $S_\alpha = \{1, \ldots, \Delta(\lambda)\} \cup \{\Delta(\lambda) + \alpha\} \subseteq [m']$.

7. Note that

$$c^{(\log m)} = \mathsf{LinEval}\left(\mathsf{pk}, (\mathsf{ct}_1)_{\mathbf{c}_1}, \ldots, (\mathsf{ct}_m)_{\mathbf{c}_1}\right) = \sum_{j=1}^{m} (\mathsf{ct}_j)_{\mathbf{c}_1}$$

where addition is coordinate-wise over $\mathbb{F}_q$, and

$$\mathsf{h}_{\log m} = \mathsf{SEH.Hash}\left(\mathsf{hk}_{\log m}, \left(c_k^{(\log m)}, \pi_k^{(\log m)}\right)_{k\in[m']}\right).$$

8. Compute

$$\mathsf{seBARG}_0.\pi = \mathsf{seBARG}_0.\mathcal{P}\left(\mathsf{seBARG}_0.\mathsf{crs}, \begin{array}{l}\text{instances} = \left(\mathsf{pk}, \mathsf{v}_k, \mathsf{hk}, \mathsf{h}, \{\mathsf{hk}_t, \mathsf{h}_t\}_{t=0}^{\log m}, k\right)_{k\in[m']}, \\ \text{witnesses} = \left(c_k^{(\log m)}, \pi_k^{(\log m)}, \rho_k\right)_{k\in[m']}\end{array}\right)$$

where seBARG_0 is defined w.r.t. the NP language $\mathcal{L}_0$ that has instances of length

$$n' = \left|\left(\mathsf{pk}, \mathsf{v}_k, \mathsf{hk}, \mathsf{h}, \{\mathsf{hk}_t, \mathsf{h}_t\}_{t=0}^{\log m}, k\right)\right|,$$

and a valid witness $\left(c_k^{(\log m)}, \pi_k^{(\log m)}, \rho_k\right)$ satisfies the following conditions:

(a) ρ_k is a valid opening of $\left(c_k^{(\log m)}, \pi_k^{(\log m)}\right)$ w.r.t. $(\mathsf{hk}_{\log m}, \mathsf{h}_{\log m})$. Namely,

$$\mathsf{SEH.Verify}\left(\mathsf{hk}_{\log m}, \mathsf{h}_{\log m}, J_k, \left(c_k^{(\log m)}, \pi_k^{(\log m)}\right), \rho_k\right) = 1,$$

where J_k are the coordinates corresponding to $\left(c_k^{(\log m)}, \pi_k^{(\log m)}\right)$ in $\left(c_k^{(\log m)}, \pi_k^{(\log m)}\right)_{k\in[m']}$.

(b) $\pi_k^{(\log m)}$ is a valid proof. Namely,

$$\mathsf{RAM}_1.\mathcal{V}\left(\mathsf{RAM}_1.\mathsf{crs}, \mathsf{hk}_{\log m}, \mathsf{h}_{\log m}, \left(\mathsf{hk}, \mathsf{h}, \{\mathsf{hk}_t, \mathsf{h}_t\}_{t=0}^{\log m-1}, \mathsf{pk}, 1, k, c_k^{(\log m)}\right), \pi_k^{(\log m)}\right) = 1.$$

(c) $c_k^{(\log m)}$ is consistent with v_k. Namely, if $k \in [m' - m]$, then

$$\mathsf{v}_k = c_k^{(\log m)}$$

and if $k > m' - m$, then

$$\mathsf{v}_k = \mathsf{Compress}_2\left(c_k^{(\log m)}\right).$$

9. Set $\pi = (\mathsf{h}_0, \mathsf{h}_1, \ldots, \mathsf{h}_{\log m}, \mathsf{seBARG}_0.\pi)$.
10. Output $(\mathsf{v}, \mathsf{rt} = (\mathsf{h}, \pi))$.

$\mathsf{Validate}(\mathsf{crs}_s, \mathsf{v}, \mathsf{rt}) \to 0/1$. This poly-time algorithm does the following:

1. Parse

$$\mathsf{crs}_s = \left(\mathsf{pk}, \mathsf{pk}_{\mathbf{c}}, \mathsf{hk}, \mathsf{h}', \{\mathsf{hk}_t\}_{t=0}^{\log m}, \mathsf{RAM}_0.\mathsf{crs}, \mathsf{RAM}_1.\mathsf{crs}, \mathsf{seBARG}_0.\mathsf{crs}, \mathsf{seBARG}_1.\mathsf{crs}\right),$$

parse $\mathsf{rt} = (\mathsf{h}, \pi)$ where

$$\pi = (\mathsf{h}_0, \mathsf{h}_1, \ldots, \mathsf{h}_{\log m}, \mathsf{seBARG}_0.\pi),$$

and parse

$$\mathsf{v} = (\mathsf{v}_1, \ldots, \mathsf{v}_{m'})$$

where recall that $\mathsf{v}_j \in \mathbb{F}_q$ for every $j \in [m' - m]$, and $\mathsf{v}_j \in \{0,1\}$ for every $j \in [m' - m + 1, m']$.

2. Output 1 if and only if

$$\mathsf{seBARG}_0.\mathcal{V}\left(\mathsf{seBARG}_0.\mathsf{crs}, \left(\mathsf{pk}, \mathsf{v}_k, \mathsf{hk}, \mathsf{h}, \{\mathsf{hk}_t, \mathsf{h}_t\}_{t=0}^{\log m}, k\right)_{k\in[m']}, \mathsf{seBARG}_0.\pi\right) = 1.$$

$\mathsf{Open}(\mathsf{crs}_\ell, \mathsf{crs}_s, \boldsymbol{x}, i) \to (x_i, \rho)$. This poly-time algorithm does the following:

1. Parse $\mathsf{crs}_\ell = \mathsf{hk}' = (\mathsf{pk}, \mathsf{hk}'_1, \ldots, \mathsf{hk}'_m)$ and

$$\mathsf{crs} = \left(\mathsf{pk}, \mathsf{pk}_{\mathbf{c}}, \mathsf{hk}, \mathsf{h}', \{\mathsf{hk}_t\}_{t=0}^{\log m}, \mathsf{RAM}_0.\mathsf{crs}, \mathsf{RAM}_1.\mathsf{crs}, \mathsf{seBARG}_0.\mathsf{crs}, \mathsf{seBARG}_1.\mathsf{crs}\right).$$

2. Compute

$$(\mathsf{ct}_1, \ldots, \mathsf{ct}_m) = \mathsf{SEH.Hash}\left(\mathsf{hk}', \boldsymbol{x}\right).$$

3. Compute

$$(b, \rho'_1, \ldots, \rho'_m) = \mathsf{SEH.Open}(\mathsf{hk}', \boldsymbol{x}, i).$$

Note that $(\rho'_1, \ldots, \rho'_m)$ is an opening of x_i w.r.t. the hash value $(\mathsf{ct}_1, \ldots, \mathsf{ct}_m)$. Importantly, for every $j \in [m]$, one can verify ρ'_j given only $(\mathsf{pk}, \mathsf{hk}'_j, \mathsf{ct}_j)$. Namely,

$$\mathsf{SEH.Verify}((\mathsf{pk}, \mathsf{hk}'_j), \mathsf{ct}_j, i, b, \rho'_j) = 1.$$

4. For every $j \in [m]$ let

$$o_j = \mathsf{SEH.Open}\left(\mathsf{hk}, (\mathsf{ct}_1, \ldots, \mathsf{ct}_m), [(j-1)\cdot n + 1, j \cdot n]\right)$$

and let

$$o'_j = \mathsf{SEH.Open}\left(\mathsf{hk}, (\mathsf{hk}'_1, \ldots, \mathsf{hk}'_m), [(j-1)\cdot n + 1, j \cdot n]\right)$$

where $n = |\mathsf{ct}_j|$.
Note that o_j is an opening of ct_j w.r.t. the hash value $\mathsf{h} = \mathsf{SEH.Hash}(\mathsf{hk}, (\mathsf{ct}_1, \ldots, \mathsf{ct}_m))$, and o'_j is an opening of hk'_j w.r.t. the hash value $\mathsf{h}' = \mathsf{SEH.Hash}(\mathsf{hk}, (\mathsf{hk}'_1, \ldots, \mathsf{hk}'_m))$.

5. Compute

$$\mathsf{seBARG}_1.\pi = \mathsf{seBARG}_1.\mathcal{P}\left(\mathsf{seBARG}_1.\mathsf{crs}, \begin{array}{l}\text{instances} = (\mathsf{hk}, \mathsf{pk}, \mathsf{h}', \mathsf{h}, i, b, j)_{j\in[m]}, \\ \text{witnesses} = (\mathsf{ct}_j, \mathsf{hk}'_j, \rho'_j, o_j, o'_j)_{j\in[m]}\end{array}\right)$$

where seBARG_1 is defined w.r.t. the NP language $\mathcal{L}_1$ that has instances of length

$$n'' = \left|\left(\mathsf{hk}, \mathsf{pk}, \mathsf{h}', \mathsf{h}, i, b, j\right)\right|$$

and a valid witness $(\mathsf{ct}_j, \mathsf{hk}'_j, \rho'_j, o_j, o'_j)$ satisfies the following conditions:

(a) ρ'_j is a valid opening of b w.r.t. $(\mathsf{pk}, \mathsf{hk}'_j, \mathsf{ct}_j)$. Namely,

$$\mathsf{SEH.Verify}\left(\left(\mathsf{pk}, \mathsf{hk}'_j\right), \mathsf{ct}_j, i, b, \rho'_j\right) = 1.$$

(b) o_j is a valid opening of ct_j w.r.t. $(\mathsf{hk}, \mathsf{h})$. Namely,

$$\mathsf{SEH.Verify}\left(\mathsf{hk}, \mathsf{h}, [(j-1)\cdot n + 1, j \cdot n], \mathsf{ct}_j, o_j\right) = 1.$$

(c) o'_j is a valid opening of hk'_j w.r.t. $(\mathsf{hk}, \mathsf{h}')$. Namely,

$$\mathsf{SEH.Verify}\left(\mathsf{hk}, \mathsf{h}', [(j-1)\cdot n + 1, j \cdot n], \mathsf{hk}'_j, o'_j\right) = 1.$$

6. Output $(b, \rho = \mathsf{seBARG}_1.\pi)$.

$\mathsf{Verify}(\mathsf{crs}_s, \mathsf{rt}, i, b, \rho) \to 0/1$. This poly-time algorithm does the following:

1. Parse

$$\mathsf{crs}_s = \left(\mathsf{pk}, \mathsf{pk}_{\mathbf{c}}, \mathsf{hk}, \mathsf{h}', \{\mathsf{hk}_t\}_{t=0}^{\log m}, \mathsf{RAM}_0.\mathsf{crs}, \mathsf{RAM}_1.\mathsf{crs}, \mathsf{seBARG}_0.\mathsf{crs}, \mathsf{seBARG}_1.\mathsf{crs}\right),$$

and parse $\mathsf{rt} = (\mathsf{h}, \pi)$, $\rho = \mathsf{seBARG}_1.\pi$.

2. Output

$$\mathsf{seBARG}_1.\mathcal{V}(\mathsf{seBARG}_1.\mathsf{crs}, (\mathsf{hk}, \mathsf{pk}, \mathsf{h}', \mathsf{h}, i, b, j)_{j\in[m]}, \mathsf{seBARG}_1.\pi).$$

$\mathsf{Extract}(\mathsf{td}, \mathsf{v}) \to \{0,1\}^m$. This poly-time algorithm does the following:

1. Output $\mathsf{Dec}(\mathsf{td}, \mathsf{v})$.[11]

4.3 Analysis

Theorem 4.2. *Assuming hardness of the* LWE *problem, the construction defined in Section 4.2 is a rate-1 fully-local somewhere extractable hash family as in Section 4.1.*

Proof of Theorem 4.2.

Efficiency. Follows immediately from the efficiency of the underlying primitives, namely, the efficiency of the underlying rate-1 linearly homomorphic encryption scheme, the SEH scheme, the RAM SNARGs, and the seBARG schemes. If $I = \{i^* + 1, \ldots, i^* + m\}$ for some $i^* \in [N]$, then Gen runs in time $\mathsf{poly}(\lambda, \log N)$ (see Remark 4.2).

Index hiding. Follows immediately from the index hiding property of the underlying SEH hash family (see Definition 3.1).

Completeness. Follows immediately from the completeness property of the underlying $\mathsf{RAM}_0, \mathsf{RAM}_1$ schemes (Definition 3.7), the completeness of the underlying $\mathsf{seBARG}_0, \mathsf{seBARG}_1$ schemes (Definition 3.3), and the opening completeness of the underlying SEH family (Definition 3.1).

[11] Recall that the SEH hash family we used has the property that the trapdoor key td is the secret key corresponding to the underlying encryption scheme (see Lemma 3.2).

Somewhere statistically binding w.r.t. opening. Fix any (possibly all-powerful) adversary $\mathcal{A}$, any polynomial $N = N(\lambda)$, subset $I \subseteq [N]$, and index $i^* \in I$. We need to prove that there exists a negligible function μ such that for every $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{ll} \mathsf{Verify}(\mathsf{crs}_s, \mathsf{rt}, i^*, b, \rho) = 1 & (\mathsf{crs}_\ell, \mathsf{crs}_s, \mathsf{td}') \leftarrow \mathsf{Gen}(1^\lambda, N, I), \\ \wedge\ \mathsf{Validate}(\mathsf{crs}_s, \mathsf{v}, \mathsf{rt}) = 1 \quad : & (\mathsf{v}, \mathsf{rt}, b, \rho) \leftarrow \mathcal{A}(\mathsf{crs}_\ell, \mathsf{crs}_s), \\ \wedge\ b \neq b_{i^*} & (b_i)_{i \in I} = \mathsf{Extract}(\mathsf{td}, \mathsf{v}) \end{array}\right] \leq \mu(\lambda). \tag{3}$$

Parse $I = \{i_1, \ldots, i_m\}$ and suppose that $i^* = i_\alpha$. Let Gen_α be identical to Gen except that it sets α as above, as opposed to setting $\alpha = 1$. In addition, Gen_α does not output only td' as the trapdoor, but rather outputs all the trapdoors generated during the Gen algorithm. Namely, it outputs

$$\mathsf{td}^* = \left(\mathsf{td}', \mathsf{td}, \{\mathsf{td}_t\}_{t=0}^{\log m}, \mathsf{seBARG}_0.\mathsf{td}, \mathsf{seBARG}_1.\mathsf{td}\right).$$

Below we define the extraction algorithm $\mathcal{E}$, which uses td^* and calls as subroutines the polynomial time extractors $\mathsf{SEH.Extract}$, $\mathsf{seBARG}_0.\mathcal{E}$ and $\mathsf{seBARG}_1.\mathcal{E}$:

$\mathcal{E}(\mathsf{td}^*, \mathsf{crs}_s, \mathsf{v}, \mathsf{rt}, b, \rho)$

Parse $\mathsf{td}^* = \left(\mathsf{td}', \mathsf{td}, \{\mathsf{td}_t\}_{t=0}^{\log m}, \mathsf{seBARG}_0.\mathsf{td}, \mathsf{seBARG}_1.\mathsf{td}\right)$.

Parse $\mathsf{crs}_s = \left(\mathsf{pk}, \mathsf{pk}_{\mathbf{c}}, \mathsf{hk}, \mathsf{h}', \{\mathsf{hk}_t\}_{t=0}^{\log m}, \mathsf{RAM}_0.\mathsf{crs}, \mathsf{RAM}_1.\mathsf{crs}, \mathsf{seBARG}_0.\mathsf{crs}, \mathsf{seBARG}_1.\mathsf{crs}\right)$.

Parse $\mathsf{rt} = (\mathsf{h}, \pi)$, $\pi = (\mathsf{h}_0, \ldots, \mathsf{h}_{\log m}, \mathsf{seBARG}_0.\pi)$, $\rho = \mathsf{seBARG}_1.\pi$.

Compute $\mathsf{hk}'_\alpha = \mathsf{SEH.Extract}(\mathsf{td}, \mathsf{h}')$.

Compute $\mathsf{ct}_\alpha = \mathsf{SEH.Extract}(\mathsf{td}, \mathsf{h})$.

for $t \in [0, \log m]$: Compute $\left(c_{j,k}^{(t)}, \pi_{j,k}^{(t)}\right)_{j \in J'_{t,\alpha}, k \in S_\alpha} = \mathsf{SEH.Extract}(\mathsf{td}_t, \mathsf{h}_t)$.

Compute $\omega_0 = \mathsf{seBARG}_0.\mathcal{E}(\mathsf{seBARG}_0.\mathsf{td}, \left(\mathsf{pk}, \mathsf{v}_k, \mathsf{hk}, \mathsf{h}, \{\mathsf{hk}_t, \mathsf{h}_t\}_{t=0}^{\log m}, k\right)_{k \in [m']}, \mathsf{seBARG}_0.\pi)$.

Compute $\omega_1 = \mathsf{seBARG}_1.\mathcal{E}(\mathsf{seBARG}_1.\mathsf{td}, (\mathsf{hk}, \mathsf{pk}, \mathsf{h}', \mathsf{h}, i^*, b, j)_{j \in [m]}, \mathsf{seBARG}_1.\pi)$.

return $\left(\mathsf{hk}'_\alpha, \mathsf{ct}_\alpha, \left\{\left(c_{j,k}^{(t)}, \pi_{j,k}^{(t)}\right)_{j \in J'_{t,\alpha}, k \in S_\alpha}\right\}_{t=0}^{\log m}, \omega_0, \omega_1\right)$

Recall that $J'_{t,\alpha} = \left\{\lceil \alpha/2^t \rceil - 1, \lceil \alpha/2^t \rceil, \lceil \alpha/2^t \rceil + 1\right\} \cap [m_t]$ and $S_\alpha = \{1, \ldots, \Delta(\lambda)\} \cup \{\Delta(\lambda) + \alpha\}$. We define the following experiment:

EXPT_α

$(\mathsf{crs}_\ell, \mathsf{crs}_s, \mathsf{td}^*) \leftarrow \mathsf{Gen}_\alpha(1^\lambda, N, I)$

$(\mathsf{v}, \mathsf{rt}, b, \rho) \leftarrow \mathcal{A}(\mathsf{crs}_\ell, \mathsf{crs}_s)$

$(b_i)_{i \in I} = \mathsf{Extract}(\mathsf{td}, \mathsf{v})$

$\left(\mathsf{hk}'_\alpha, \mathsf{ct}_\alpha, \left\{\left(c_{j,k}^{(t)}, \pi_{j,k}^{(t)}\right)_{j \in J'_{t,\alpha}, k \in S_\alpha}\right\}_{t=0}^{\log m}, \omega_0, \omega_1\right) = \mathcal{E}(\mathsf{td}^*, \mathsf{crs}_s, \mathsf{v}, \mathsf{rt}, b, \rho)$

By the index hiding property of the underlying SEH family and the underlying seBARG schemes, to prove Equation (3) it suffices to prove that there exists a negligible function μ such that for every $\lambda \in \mathbb{N}$,

$$\Pr_{\mathsf{EXPT}_\alpha}\left[\ b \neq b_{i_\alpha}\ \wedge\ \mathsf{Verify}(\mathsf{crs}_s, \mathsf{rt}, i_\alpha, b, \rho) = 1\ \wedge\ \mathsf{Validate}(\mathsf{crs}_s, \mathsf{v}, \mathsf{rt}) = 1\ \right] \leq \mu(\lambda). \tag{4}$$

Recall that $\mathsf{Verify}(\mathsf{crs}_s, \mathsf{rt}, i_\alpha, b, \rho) = 1 \ \wedge \ \mathsf{Validate}(\mathsf{crs}_s, \mathsf{v}, \mathsf{rt}) = 1$ if and only if

$$\mathsf{seBARG}_0.\mathcal{V}\left(\mathsf{seBARG}_0.\mathsf{crs}, \left(\mathsf{pk}, \mathsf{v}_k, \mathsf{hk}, \mathsf{h}, \{\mathsf{hk}_t, \mathsf{h}_t\}_{t=0}^{\log m}, k\right)_{k\in[m']}, \mathsf{seBARG}_0.\pi\right) = 1$$

and

$$\mathsf{seBARG}_1.\mathcal{V}\left(\mathsf{seBARG}_1.\mathsf{crs}, \left(\mathsf{hk}, \mathsf{pk}, \mathsf{h}', \mathsf{h}, i_\alpha, b, j\right)_{j\in[m]}, \mathsf{seBARG}_1.\pi\right) = 1.$$

In what follows, we parse

$$\omega_0 = \left(c'^{(\log m)}_k, \pi'^{(\log m)}_k, \rho_k\right)_{k\in S_\alpha} \text{ and } \omega_1 = \left(\mathsf{ct}'_\alpha, \mathsf{hk}''_\alpha, \rho'_\alpha, o_\alpha, o'_\alpha\right).$$

The somewhere argument of knowledge property of the underlying $\mathsf{seBARG}_0, \mathsf{seBARG}_1$ schemes (Definition 3.3) implies that to prove Equation (4) it suffices to prove that there exists a negligible function μ such that for every $\lambda \in \mathbb{N}$,

$$\Pr_{\mathsf{EXPT}_\alpha}\left[\begin{array}{l} b \neq b_{i_\alpha} \\ \wedge\ \forall\ k \in S_\alpha : \left(c'^{(\log m)}_k, \pi'^{(\log m)}_k, \rho_k\right) \text{ is a valid witness for} \\ \qquad\qquad \left(\mathsf{pk}, \mathsf{v}_k, \mathsf{hk}, \mathsf{h}, \{\mathsf{hk}_t, \mathsf{h}_t\}_{t=0}^{\log m}, k\right) \in \mathcal{L}_0 \\ \wedge\ \left(\mathsf{ct}'_\alpha, \mathsf{hk}''_\alpha, \rho'_\alpha, o_\alpha, o'_\alpha\right) \text{ is a valid witness for } (\mathsf{hk}, \mathsf{pk}, \mathsf{h}', \mathsf{h}, i_\alpha, b, \alpha) \in \mathcal{L}_1 \end{array}\right] \leq \mu(\lambda). \quad (5)$$

Recall that $\left(\mathsf{ct}'_\alpha, \mathsf{hk}''_\alpha, \rho'_\alpha, o_\alpha, o'_\alpha\right)$ is a valid witness for $(\mathsf{hk}, \mathsf{pk}, \mathsf{h}', \mathsf{h}, i_\alpha, b, \alpha) \in \mathcal{L}_1$ if and only if $\rho'_\alpha, o_\alpha, o'_\alpha$ are valid openings for $b, \mathsf{ct}'_\alpha, \mathsf{hk}''_\alpha$ respectively (see Item 5). The somewhere statistically binding w.r.t opening property of the underlying SEH family (Definition 3.1) implies that there exists a negligible function negl such that for every $\lambda \in \mathbb{N}$,

$$\Pr_{\mathsf{EXPT}_\alpha}\left[\begin{array}{l} (\rho'_\alpha, o_\alpha, o'_\alpha) \text{ are valid openings for } \left(b, \mathsf{ct}'_\alpha, \mathsf{hk}''_\alpha\right) \text{ respectively} \\ \wedge\ \left((b \neq \mathsf{SEH.Extract}(\mathsf{td}', \mathsf{ct}'_\alpha)) \ \vee \ (\mathsf{ct}'_\alpha \neq \mathsf{ct}_\alpha) \ \vee \ (\mathsf{hk}''_\alpha \neq \mathsf{hk}'_\alpha)\right) \end{array}\right] \leq \mathsf{negl}(\lambda).$$

This implies that for every $\lambda \in \mathbb{N}$,

$$\Pr_{\mathsf{EXPT}_\alpha}\left[\begin{array}{l} \left(\mathsf{ct}'_\alpha, \mathsf{hk}''_\alpha, \rho'_\alpha, o_\alpha, o'_\alpha\right) \text{ is a valid witness for } (\mathsf{hk}, \mathsf{pk}, \mathsf{h}', \mathsf{h}, i_\alpha, b, \alpha) \in \mathcal{L}_1 \\ \wedge\ b \neq \mathsf{SEH.Extract}(\mathsf{td}', \mathsf{ct}_\alpha) \end{array}\right] \leq \mathsf{negl}(\lambda). \quad (6)$$

It remains to argue that there exists a negligible function μ' such that for every $\lambda \in \mathbb{N}$,

$$\Pr_{\mathsf{EXPT}_\alpha}\left[\begin{array}{l} \forall\ k \in S_\alpha : \quad \left(c'^{(\log m)}_k, \pi'^{(\log m)}_k, \rho_k\right) \text{ is a valid witness for} \\ \qquad\qquad \left(\mathsf{pk}, \mathsf{v}_k, \mathsf{hk}, \mathsf{h}, \{\mathsf{hk}_t, \mathsf{h}_t\}_{t=0}^{\log m}, k\right) \in \mathcal{L}_0 \\ \wedge\ \mathsf{SEH.Extract}(\mathsf{td}', \mathsf{ct}_\alpha) \neq b_{i_\alpha} \end{array}\right] \leq \mu'(\lambda).$$

Let

$$\left(c^{(\log m)}_k, \pi^{(\log m)}_k\right) \triangleq \left(c^{(\log m)}_{j,k}, \pi^{(\log m)}_{j,k}\right)_{j\in J'_{\log m,\alpha}}$$

where $\left(c^{(\log m)}_{j,k}, \pi^{(\log m)}_{j,k}\right)_{j\in J'_{\log m,\alpha}}$ was extracted by $\mathcal{E}$ (note that $J'_{\log m,\alpha} = \{1\}$ so the sizes align).

Recall that $\left(c'^{(\log m)}_k, \pi'^{(\log m)}_k, \rho_k\right)$ is a valid witness for $\left(\mathsf{pk}, \mathsf{v}_k, \mathsf{hk}, \mathsf{h}, \{\mathsf{hk}_t, \mathsf{h}_t\}_{t=0}^{\log m}, k\right) \in \mathcal{L}_0$ if and only if (1) ρ_k is a valid opening of $\left(c'^{(\log m)}_k, \pi'^{(\log m)}_k\right)$ w.r.t. $\left(\mathsf{hk}_{\log m}, \mathsf{h}_{\log m}\right)$, (2) $\pi'^{(\log m)}_k$ is a

valid RAM proof, and (3) $c'^{(\log m)}_k$ is consistent with v_k (see Item 8). The somewhere statistically binding w.r.t. opening property of the underlying SEH family (Definition 3.1) implies that there exists a negligible function negl such that for every $\lambda \in \mathbb{N}$,

$$\Pr_{\mathsf{EXPT}_\alpha}\left[\begin{array}{l} \forall\ k \in S_\alpha: \quad \rho_k \text{ is a valid opening of } \left(c'^{(\log m)}_k, \pi'^{(\log m)}_k\right) \\ \wedge\ \exists\ k \in S_\alpha: \quad \left(c'^{(\log m)}_k, \pi'^{(\log m)}_k\right) \neq \left(c^{(\log m)}_k, \pi^{(\log m)}_k\right) \end{array}\right] \leq \mathsf{negl}(\lambda).$$

Also note that if $c^{(\log m)}_k$ is consistent with v_k for all $k \in S_\alpha$, then

$$\mathsf{CompDec}\left(\mathsf{sk}_{\mathbf{c}}, \left(\mathsf{Compress}_2\left(c^{(\log m)}_k\right)\right)_{k\in S_\alpha}\right) = \mathsf{CompDec}\left(\mathsf{sk}_{\mathbf{c}}, (\mathsf{v}_k)_{k\in S_\alpha}\right) = b_{i_\alpha}.$$

Thus to prove Equation (5) it suffices to argue that there exists a negligible function μ' such that for every $\lambda \in \mathbb{N}$,

$$\Pr_{\mathsf{EXPT}_\alpha}\left[\begin{array}{l} \pi^{(\log m)}_k \text{ is a valid proof} \\ \wedge\ \ \mathsf{SEH.Extract}(\mathsf{td}', \mathsf{ct}_\alpha) \neq \mathsf{CompDec}\left(\mathsf{sk}_{\mathbf{c}}, \left(\mathsf{Compress}_2\left(c^{(\log m)}_k\right)\right)_{k\in S_\alpha}\right) \end{array}\right] \leq \mu'(\lambda). \quad (7)$$

The partial-input soundness property of the underlying RAM_1 scheme (applied $\log m - t$ times) implies that there exists a negligible function negl such that for every $\lambda \in \mathbb{N}, t \in [0, \log m], k \in S_\alpha$, and $j \in J'_{t,\alpha}$,

$$\Pr_{\mathsf{EXPT}_\alpha}\left[\begin{array}{l} \pi^{(\log m)}_k \text{ is a valid RAM proof} \\ \wedge\ \pi^{(t)}_{j,k} \text{ is not a valid RAM proof} \end{array}\right] \leq (\log m - t)\cdot \mathsf{negl}(\lambda).$$

Definition 4.3. *For every $t \in [\log m]$ and $j \in J'_{t,\alpha}$ we say that $\left(c^{(t)}_{j,k}\right)_{k\in S_\alpha}$ is invalid if $j \neq \lceil \frac{\alpha}{2^t} \rceil$ and*

$$\mathsf{CompDec}\left(\mathsf{sk}_{\mathbf{c}}, \left(\mathsf{Compress}_2\left(c^{(t)}_{j,k}\right)\right)_{k\in S_\alpha}\right) \neq 0.$$

Claim 4.3.1. *There exists a negligible function negl such that for every $\lambda \in \mathbb{N}, t \in [0, \log m]$ and $j \in J'_{t,\alpha}$,*

$$\Pr_{\mathsf{EXPT}_\alpha}\left[\begin{array}{ll} \left(\pi^{(t)}_{j,k}\right)_{k\in S_\alpha} & \textit{are valid RAM proofs} \\ \wedge\ \left(c^{(t)}_{j,k}\right)_{k\in S_\alpha} & \textit{is invalid} \end{array}\right] \leq \mathsf{negl}(\lambda).$$

We note that Claim 4.3.1 implies Equation (7) since if $\left\{\left(c^{(t)}_{j,k}\right)_{j\in J'_{t,\alpha}, k\in S_\alpha}\right\}_{t=0}^{\log m}$ are all valid then the correctness of the underlying encryption scheme (Definition 3.10) implies that

$$\begin{aligned}\mathsf{CompDec}\left(\mathsf{sk}_{\mathbf{c}}, \left(\mathsf{Compress}_2\left(c^{(\log m)}_k\right)\right)_{k\in S_\alpha}\right) &= \mathsf{CompDec}\left(\mathsf{sk}_{\mathbf{c}}, \left(\mathsf{Compress}_2\left(c^{(0)}_{\alpha,k}\right)\right)_{k\in S_\alpha}\right) \\ &= \mathsf{SEH.Extract}(\mathsf{td}', \mathsf{ct}_\alpha)\end{aligned}$$

except with negligible probability, since all the other ciphertexts we add up encrypt 0. Thus, to conclude the proof it remains to prove Claim 4.3.1.

Proof of Claim 4.3.1. We proceed by induction.

- **Base case ($t = 0$).** Recall that for $k \in S_\alpha$, $\pi^{(0)}_{j,k}$ certifies that $c^{(0)}_{j,k}$ is the k-th element of $\mathsf{Compress}_1(\mathsf{pk}_{\mathbf{c}}, \mathsf{ct}_j, m, j)$. The partial-input soundness property of the underlying RAM_0 scheme (Definition 3.7) implies that

$$\Pr_{\mathsf{EXPT}_\alpha}\left[\begin{array}{l}\left(\pi^{(0)}_{j,k}\right)_{k\in S_\alpha} \text{ are valid RAM proofs} \\ \wedge\ \exists\ k \in S_\alpha : c^{(0)}_{j,k} \neq \left(\mathsf{Compress}_1(\mathsf{pk}_{\mathbf{c}}, \mathsf{ct}_j, m, j)\right)_k\end{array}\right] \leq \mathsf{negl}(\lambda).$$

  Recall that $\mathsf{Compress}_1(\mathsf{pk}_{\mathbf{c}}, \mathsf{ct}_j, m, j)$ is a ciphertext encrypting the vector $\mathsf{SEH.Extract}(\mathsf{td}', \mathsf{ct}_j) \cdot e_j$, which is 0 everywhere except at coordinate j. The correctness of the encryption scheme implies that for all $j \neq \alpha$,

$$\mathsf{CompDec}\left(\mathsf{sk}_{\mathbf{c}}, \left(\mathsf{Compress}_2\left(\left(\mathsf{Compress}_1(\mathsf{pk}_{\mathbf{c}}, \mathsf{ct}_j, m, j)\right)_k\right)\right)_{k\in S_\alpha}\right) = 0,$$

  so we can conclude the claim for $t = 0$.

- **Inductive step.** Recall that $\pi^{(t)}_{j,k}$ certifies that $c^{(t)}_{j,k} = c^{(t-1)}_{2j-1,k} + c^{(t-1)}_{2j,k}$, where addition is over the finite field $\mathbb{F}_q$, and that $\pi^{(t-1)}_{2j-1,k}, \pi^{(t-1)}_{2j,k}$ are valid RAM proofs. The partial-input soundness property of the underlying RAM_1 scheme implies that for every $\lambda \in \mathbb{N}, j \in J_{t,\alpha}$,

$$\Pr_{\mathsf{EXPT}_\alpha}\left[\begin{array}{l}\left(\pi^{(t)}_{j,k}\right)_{k\in S_\alpha} \text{ are valid RAM proofs} \\ \wedge\ \left(\begin{array}{l}\left(c^{(t)}_{j,k}\right)_{k\in S_\alpha} \neq \left(c^{(t-1)}_{2j-1,k}\right)_{k\in S_\alpha} + \left(c^{(t-1)}_{2j,k}\right)_{k\in S_\alpha} \\ \qquad \vee\ \exists\ k \in S_\alpha : \pi^{(t-1)}_{2j-1,k} \text{ or } \pi^{(t-1)}_{2j,k} \text{ is not a valid RAM proof}\end{array}\right)\end{array}\right] \leq \mathsf{negl}(\lambda), \tag{8}$$

  where addition is coordinate-wise over the finite field $\mathbb{F}_q$. The inductive hypothesis for $t-1$ implies that

$$\Pr_{\mathsf{EXPT}_\alpha}\left[\begin{array}{l}\left(\pi^{(t-1)}_{2j-1,k}, \pi^{(t-1)}_{2j,k}\right)_{k\in S_\alpha} \text{ are valid RAM proofs} \\ \wedge\ \left(c^{(t-1)}_{2j-1,k}\right)_{k\in S_\alpha} \text{ or } \left(c^{(t-1)}_{2j,k}\right)_{k\in S_\alpha} \text{ is invalid}\end{array}\right] \leq \mathsf{negl}(\lambda). \tag{9}$$

  Note that if $\left(c^{(t-1)}_{2j-1,k}\right)_{k\in S_\alpha}$ and $\left(c^{(t-1)}_{2j,k}\right)_{k\in S_\alpha}$ are valid, then

$$\begin{aligned} j \neq \left\lceil \frac{\alpha}{2^t} \right\rceil &\implies 2j-1 \neq \left\lceil \frac{\alpha}{2^{t-1}} \right\rceil \text{ and } 2j \neq \left\lceil \frac{\alpha}{2^{t-1}} \right\rceil \\ &\implies \mathsf{CompDec}\left(\mathsf{sk}_{\mathbf{c}}, \left(\mathsf{Compress}_2\left(c^{(t-1)}_{2j-1,k}\right)\right)_{k\in S_\alpha}\right) = 0 \\ &\qquad \text{and } \mathsf{CompDec}\left(\mathsf{sk}_{\mathbf{c}}, \left(\mathsf{Compress}_2\left(c^{(t-1)}_{2j,k}\right)\right)_{k\in S_\alpha}\right) = 0 \\ &\implies \mathsf{CompDec}\left(\mathsf{sk}_{\mathbf{c}}, \left(\mathsf{Compress}_2\left(c^{(t-1)}_{2j-1,k} + c^{(t-1)}_{2j,k}\right)\right)_{k\in S_\alpha}\right) = 0, \end{aligned}$$

  so $\left(c^{(t)}_{j,k}\right)_{k\in S_\alpha} = \left(c^{(t-1)}_{2j-1,k}\right)_{k\in S_\alpha} + \left(c^{(t-1)}_{2j,k}\right)_{k\in S_\alpha}$ is valid. Combining this fact with Equations (8) and (9), we can conclude the claim for t.

□

Extraction correctness. Follows immediately from the correctness of the encryption scheme (Definition 3.10).

□

5 Rate-1 seBARGs

In this section we construct a rate-1 seBARG (Definitions 3.3 and 3.6) for any NP language $\mathcal{L} = \{\mathcal{L}_n\}_{n\in\mathbb{N}}$ with witnesses of length $m = m(n)$. Our construction of a rate-1 seBARG uses the following ingredients:

1. A rate-1 fISEH family (see Section 4)

$$(\mathsf{H.Gen}, \mathsf{H.Hash}, \mathsf{H.Extract}, \mathsf{H.Digest}, \mathsf{H.Open}, \mathsf{H.Verify}).$$

2. A flexible RAM SNARG scheme with partial input soundness (see Section 3.3)

$$(\mathsf{RAM.Gen}, \mathsf{RAM.Digest}, \mathsf{RAM}.\mathcal{P}, \mathsf{RAM}.\mathcal{V})$$

   as in Theorem 3.8 w.r.t. the RAM machine $\mathcal{R}$ which takes an implicit input $(\omega_1, \ldots, \omega_k)$ and an explicit input (x_i, i) and outputs 1 if and only if ω_i is a valid witness for $x_i \in \mathcal{L}$. Let T denote the run-time of $\mathcal{R}$.

3. An index seBARG scheme with proof length $|\mathsf{seB}.\pi| = m \cdot \mathsf{poly}(\lambda, \log k)$ as in Theorem 3.5 (see Section 3.2)

$$(\mathsf{seB.Gen}, \mathsf{seB}.\mathcal{P}, \mathsf{seB}.\mathcal{V})$$

   for the NP language

$$\mathcal{L}' = \left\{ (\mathsf{RAM.crs}, \mathsf{H.crs}, \mathsf{rt}, x_i, i) \;\middle|\; \begin{array}{l} \exists\ \mathsf{RAM}.\pi \in \{0,1\}^{\mathsf{poly}(\lambda)} \text{ s.t.} \\ \mathsf{RAM}.\mathcal{V}(\mathsf{RAM.crs}, \mathsf{H.crs}, \mathsf{rt}, (x_i, i), \mathsf{RAM}.\pi) = 1 \end{array} \right\}.$$

We are now ready to construct our rate-1 seBARG for $\mathcal{L}$.

$\mathsf{Gen}(1^\lambda, k, n, i^*) \to (\mathsf{crs}, \mathsf{td})$. This poly-time algorithm does the following:

1. Generate $(\mathsf{H.crs}, \mathsf{H.td}) \leftarrow \mathsf{H.Gen}\left(1^\lambda, k \cdot m, I\right)$ where $I = \{(i^* - 1) \cdot m + 1 \ldots, i^* \cdot m\}$.[12]
2. Generate $\mathsf{RAM.crs} \leftarrow \mathsf{RAM.Gen}(1^\lambda, T)$.
3. Generate $(\mathsf{seB.crs}, \mathsf{seB.td}) \leftarrow \mathsf{seB.Gen}(1^\lambda, k, n', i^*)$ where $n' = |(\mathsf{RAM.crs}, \mathsf{H.crs}, \mathsf{rt}, i, x_i)|$.
4. Output $\mathsf{crs} = (\mathsf{H.crs}, \mathsf{RAM.crs}, \mathsf{seB.crs})$ and $\mathsf{td} = (\mathsf{H.td}, \mathsf{seB.td})$.

$\mathcal{P}(\mathsf{crs}, x_1, \ldots, x_k, \omega_1, \ldots, \omega_k) \to \Pi$. This poly-time algorithm does the following:

[12] Recall that a fISEH hash family has the property that if I consists of consecutive indices (which is the case here) then Gen runs in time $\mathsf{poly}(\lambda, \log k, \log m)$, and outputs a single (shrot) crs, denoted above by H.crs.

1. Parse $\mathsf{crs} = (\mathsf{H.crs}, \mathsf{RAM.crs}, \mathsf{seB.crs})$.
2. Compute $(\mathsf{v}, \mathsf{rt}) = \mathsf{H.Hash}(\mathsf{H.crs}, (\omega_1, \ldots, \omega_k))$.
3. For every $i \in [k]$ let $\mathsf{RAM}.\pi_i = \mathsf{RAM}.\mathcal{P}\left(\mathsf{RAM.crs}, \mathsf{H.crs}, (\omega_1, \ldots, \omega_k), (x_i, i)\right)$.
4. Compute $\mathsf{seB}.\pi \leftarrow \mathsf{seB}.\mathcal{P}\left(\mathsf{seB.crs}, \begin{array}{l}\text{instances} = (\mathsf{RAM.crs}, \mathsf{H.crs}, \mathsf{rt}, x_i, i)_{i\in[k]}, \\ \text{witnesses} = (\mathsf{RAM}.\pi_i)_{i\in[k]}\end{array}\right)$.
5. Output $\Pi = (\mathsf{v}, \mathsf{rt}, \mathsf{seB}.\pi)$.

$\mathcal{V}(\mathsf{crs}, x_1, \ldots, x_k, \Pi) \to 0/1$. This poly-time algorithm does the following:

1. Parse $\mathsf{crs} = (\mathsf{H.crs}, \mathsf{RAM.crs}, \mathsf{seB.crs})$.
2. Parse $\Pi = (\mathsf{v}, \mathsf{rt}, \mathsf{seB}.\pi)$.
3. Output 1 if and only if
$$\mathsf{H.Validate}(\mathsf{H.crs}, \mathsf{v}, \mathsf{rt}) = 1$$
and
$$\mathsf{seB}.\mathcal{V}(\mathsf{seB.crs}, (\mathsf{RAM.crs}, \mathsf{H.crs}, \mathsf{rt}, x_i, i)_{i\in[k]}, \mathsf{seB}.\pi) = 1.$$

Theorem 5.1. *Assuming hardness of the* LWE *problem, the construction defined in Section 5 is a rate-*1 seBARG *as in Definitions 3.3 and 3.6.*

Proof of Theorem 5.1.

Efficiency. We first note that our construction satisfies the efficiency requirement.

1. The fact that Gen is a poly-time algorithm follows from the efficiency guarantee of the key generation algorithm of the underlying primitives: the fISEH family, the RAM delegation scheme (Definition 3.7), and the seB scheme (Definition 3.3).

2. The fact that the proof $\Pi = (\mathsf{v}, \mathsf{rt}, \mathsf{seB}.\pi)$ is of length $m + \mathsf{poly}(\lambda)$ follows from the fact that the underlying fISEH family has rate-1, which implies that $|\mathsf{v}| = m + \mathsf{poly}(\lambda)$, $|\mathsf{rt}| \leq \mathsf{poly}(\lambda)$, and the time to verify an opening is $\mathsf{poly}(\lambda)$, together with the efficiency guarantees of the underlying RAM and seB schemes, which ensure that $|\mathsf{RAM}.\pi_i| \leq \mathsf{poly}(\lambda)$ and thus $|\mathsf{seB}.\pi| \leq \mathsf{poly}(\lambda)$.

   Moreover, Π can be generated in polynomial time, which follows from the polynomial running time of the underlying algorithms.

Index hiding. The index hiding condition follows directly from the index hiding property of the underlying fISEH family (Definition 4.1) and the seB scheme (Definition 3.3).

Completeness. The completeness condition follows immediately from the completeness condition of the underlying RAM delegation scheme (Definition 3.7), the completeness condition of the underlying seB scheme (Definition 3.3), and the validation/opening completeness condition of the underlying fISEH scheme (Definition 4.1).

Somewhere argument of knowledge. We define a PPT extractor $\mathcal{E}$ that takes as input a tuple $(\mathsf{td}, (x_1, \ldots, x_k), \Pi)$, and does the following:

$\mathcal{E}(\mathsf{td}, (x_1, \ldots, x_k), \Pi)$

parse $\Pi = (\mathsf{v}, \mathsf{rt}, \pi), \mathsf{td} = (\mathsf{H.td}, \mathsf{seB.td})$

return $\omega = \mathsf{H.Extract}(\mathsf{H.td}, \mathsf{v})$

Suppose for the sake of contradiction that there exists a poly-size adversary $\mathcal{A}$, polynomials $k = k(\lambda)$ and $n = n(\lambda)$, an index $i^* \in [k]$, and a non-negligible function ϵ, such that for every $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l} \mathcal{V}(\mathsf{crs}, x_1, \ldots, x_k, \Pi) = 1 \\ \wedge\ \omega_{i^*} \text{ is not a valid witness for } x_{i^*} \in \mathcal{L} \end{array} : \begin{array}{l} (\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{Gen}(1^\lambda, k, n, i^*), \\ (x_1, \ldots, x_k, \Pi) \leftarrow \mathcal{A}(\mathsf{crs}), \\ \omega_{i^*} \leftarrow \mathcal{E}(\mathsf{td}, x_1, \ldots, x_k, \Pi) \end{array}\right] \geq \epsilon(\lambda). \tag{10}$$

We abbreviate $\boldsymbol{x} = (x_1, \ldots, x_k)$. Recall that

$$\mathcal{V}(\mathsf{crs} = (\mathsf{H.crs}, \mathsf{RAM.crs}, \mathsf{seB.crs}), \boldsymbol{x}, \Pi = (\mathsf{v}, \mathsf{rt}, \mathsf{seB}.\pi)) = 1$$

if and only if

$$\mathsf{H.Validate}(\mathsf{H.crs}, \mathsf{v}, \mathsf{rt}) = 1$$

and

$$\mathsf{seB}.\mathcal{V}(\mathsf{seB.crs}, (\mathsf{RAM.crs}, \mathsf{H.crs}, \mathsf{rt}, x_i, i)_{i\in[k]}, \mathsf{seB}.\pi) = 1.$$

Thus, Equation (10) implies that for every $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l} \mathsf{H.Validate}(\mathsf{H.crs}, \mathsf{v}, \mathsf{rt}) = 1 \\ \wedge\ \mathsf{seB}.\mathcal{V}(\mathsf{seB.crs}, \\ \qquad (\mathsf{RAM.crs}, \mathsf{H.crs}, \mathsf{rt}, x_i, i)_{i\in[k]}, \mathsf{seB}.\pi) = 1 \\ \wedge\ \omega_{i^*} \text{ is not a valid witness for } x_{i^*} \in \mathcal{L} \end{array} : \begin{array}{l} (\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{Gen}(1^\lambda, k, n, i^*), \\ (\boldsymbol{x}, \Pi = (\mathsf{v}, \mathsf{rt}, \mathsf{seB}.\pi)) = \mathcal{A}(\mathsf{crs}) \end{array}\right] \geq \epsilon(\lambda). \tag{11}$$

Consider the following poly-size adversary $\mathsf{seB}.\mathcal{A}$ for the seB scheme.

$\mathsf{seB}.\mathcal{A}(\mathsf{seB.crs})$

$(\mathsf{H.crs}, \mathsf{H.td}) \leftarrow \mathsf{H.Gen}(1^\lambda, k \cdot m, I)$

$\mathsf{RAM.crs} \leftarrow \mathsf{RAM.Gen}(1^\lambda, T)$

$\mathsf{crs} = (\mathsf{H.crs}, \mathsf{RAM.crs}, \mathsf{seB.crs})$

$(\boldsymbol{x}, \Pi = (\mathsf{v}, \mathsf{rt}, \mathsf{seB}.\pi)) = \mathcal{A}(\mathsf{crs})$

return $\left(\mathsf{seB}.\boldsymbol{x} = (\mathsf{RAM.crs}, \mathsf{H.crs}, \mathsf{rt}, x_i, i)_{i\in[k]}, \mathsf{seB}.\pi\right)$

Recall that seB proves that for all $i \in [k]$ there exists a short proof $\mathsf{RAM}.\pi_i$ such that $\mathsf{RAM}.\mathcal{V}(\mathsf{RAM.crs}, \mathsf{H.crs}_s, \mathsf{rt}, (x_i, i), \mathsf{RAM}.\pi_i) = 1$. By the somewhere argument of knowledge property (Definition 3.3), the seB scheme has a PPT extractor $\mathsf{seB}.\mathcal{E}$ that satisfies that there exists a negligible function μ such that for every $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l} \mathsf{seB}.\mathcal{V}(\mathsf{seB.crs}, \mathsf{seB}.\boldsymbol{x}, \mathsf{seB}.\pi) = 1 \\ \wedge\ \mathsf{RAM}.\mathcal{V}(\mathsf{RAM.crs}, \mathsf{H.crs}, \\ \qquad \mathsf{rt}, (x_{i^*}, i^*), \mathsf{RAM}.\pi^*) \neq 1 \end{array} : \begin{array}{l} (\mathsf{seB.crs}, \mathsf{seB.td}) \leftarrow \mathsf{seB.Gen}(1^\lambda, k, n', i^*), \\ (\mathsf{seB}.\boldsymbol{x}, \mathsf{seB}.\pi) = \mathsf{seB}.\mathcal{A}(\mathsf{seB.crs}), \\ \mathsf{RAM}.\pi^* = \mathsf{seB}.\mathcal{E}(\mathsf{seB.td}, \mathsf{seB}.\boldsymbol{x}, \mathsf{seB}.\pi) \end{array}\right] \leq \mu(\lambda) \tag{12}$$

Combining Equations (11) and (12) we conclude that there exists a non-negligible function $\epsilon'(\lambda) = \epsilon(\lambda) - \mu(\lambda)$ such that for every $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l} \omega_{i^*} \text{ is not a valid witness for } x_{i^*} \in \mathcal{L} \\ \wedge\ \mathsf{H.Validate}(\mathsf{H.crs}, \mathsf{v}, \mathsf{rt}) = 1 \\ \wedge\ \mathsf{RAM}.\mathcal{V}(\mathsf{RAM.crs}, \mathsf{H.crs}, \\ \qquad \mathsf{rt}, (x_{i^*}, i^*), \mathsf{RAM}.\pi^*) = 1 \end{array} : \begin{array}{l} (\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{Gen}(1^\lambda, k, n, i^*), \\ (\boldsymbol{x}, \Pi = (\mathsf{v}, \mathsf{rt}, \mathsf{seB}.\pi)) = \mathcal{A}(\mathsf{crs}), \\ \mathsf{RAM}.\pi^* = \mathsf{seB}.\mathcal{E}(\mathsf{seB.td}, \boldsymbol{x}, \mathsf{seB}.\pi) \end{array}\right] \geq \epsilon'(\lambda). \quad (13)$$

Consider the following poly-size adversary $\mathsf{RAM}.\mathcal{A}$ for the RAM delegation scheme.

$\mathsf{RAM}.\mathcal{A}(\mathsf{RAM.crs}, \mathsf{H.crs})$

$(\mathsf{seB.crs}, \mathsf{seB.td}) \leftarrow \mathsf{seB.Gen}(1^\lambda, k, n', i^*)$

$\mathsf{crs} = (\mathsf{H.crs}, \mathsf{RAM.crs}, \mathsf{seB.crs})$

$(\boldsymbol{x}, \Pi = (\mathsf{v}, \mathsf{rt}, \mathsf{seB}.\pi)) = \mathcal{A}(\mathsf{crs})$

$\mathsf{RAM}.\pi^* = \mathsf{seB}.\mathcal{E}(\mathsf{seB.td}, (\mathsf{RAM.crs}, \mathsf{H.crs}, \mathsf{rt}, x_i, i)_{i\in[k]}, \mathsf{seB}.\pi)$

return $((\mathsf{v}, \mathsf{rt}), (x_{i^*}, i^*), \mathsf{RAM}.\pi^*)$

Recall that the RAM machine $\mathcal{R}$ associated with the RAM SNARG scheme takes as input $(\omega_1, \ldots, \omega_k), (x_i, i)$ and outputs 1 if and only if ω_i is a valid witness for $x_i \in \mathcal{L}$. Equation (13) implies that for every $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l} \mathcal{R}((\omega_1, \ldots, \omega_k), (x_{i^*}, i^*)) = 0 \\ \wedge\ \mathsf{H.Validate}(\mathsf{H.crs}, \mathsf{v}, \mathsf{rt}) = 1 \\ \wedge\ \mathsf{RAM}.\mathcal{V}(\mathsf{RAM.crs}, \mathsf{H.crs}, \\ \qquad \mathsf{rt}, (x_{i^*}, i^*), \mathsf{RAM}.\pi^*) = 1 \end{array} : \begin{array}{l} (\mathsf{H.crs}, \mathsf{H.td}) \leftarrow \mathsf{H.Gen}(1^\lambda, k \cdot m, I), \\ \mathsf{RAM.crs} \leftarrow \mathsf{RAM.Gen}(1^\lambda, T), \\ ((\mathsf{v}, \mathsf{rt}), (x_{i^*}, i^*), \mathsf{RAM}.\pi^*) \\ \qquad = \mathsf{RAM}.\mathcal{A}(\mathsf{RAM.crs}, \mathsf{H.crs}), \\ \omega_{i^*} = \mathsf{H.Extract}(\mathsf{H.td}, \mathsf{v}), \\ \forall\ i \in [k] \setminus \{i^*\} : \omega_i = 0^m \end{array}\right] \geq \epsilon'(\lambda)$$

which contradicts partial-input soundness of the RAM delegation scheme (Definition 3.7).

□

6 Multi-Hop seBARGs

In this section, we extend the notion of a seBARG to the *multi-hop* setting, and describe a generic construction based on any rate-1 seBARG. In the multi-hop setting, seBARGs can be further batched with other seBARGs (or even a single instance-witness pair) succinctly. The number of *hops* (i.e., number of times seBARGs can be successively batched) can be any polynomial, and the batch size in each hop can be set arbitrarily. Each hop increases the proof size by an *additive* $\mathsf{poly}(\lambda)$ factor.

In Section 7 we show how to use a multi-hop seBARG to construct a multi-hop aggregate signature scheme, and how to construct an IVC scheme.

6.1 Definition

To formally capture the notion of multi-hop seBARGs, we make the following syntactic and semantic changes to seBARGs, which includes adding a new "proof combiner" procedure that we call AggProve.

Syntax.

$\mathsf{Gen}(1^\lambda, d, (i_1, \ldots, i_d)) \to (\mathsf{crs}, \mathsf{td})$**.** Gen is a PPT machine. It no longer takes as input the number of instances k being batched, nor the length of the instances n.[13] Rather, it takes as input the security parameter 1^λ, the maximum number of hops $d \in [2^\lambda]$ (i.e., the number of batch-compositions), and a sequence of d extraction indices $I = (i_1, \ldots, i_d) \in [2^\lambda]^d$. It outputs crs which consists of d strings $\mathsf{crs} = (\mathsf{crs}_1, \ldots, \mathsf{crs}_d)$, along with a trapdoor td which consists of d strings $\mathsf{td} = (\mathsf{td}_1, \ldots, \mathsf{td}_d)$.

We note that in the multi-hop setting, we need to define a separate extraction index for each hop. That is, instead of having a single extraction index (as in a standard seBARG), it takes as input d extraction indices $I = (i_1, \ldots, i_d)$. Also, since the batch size in each hop is unbounded, each index $i_j \in [2^\lambda]$ for $j \in [d]$. The j'th extraction index i_j is interpreted as saying that, from an accepting seBARG proof π created via j-hops/compositions, we can efficiently extract the i_j'th witness (which is an another accepting seBARG proof π' created via $(j-1)$-hops/compositions). Thus, the sequence of extraction indices I defines the sequence in which multi-hop seBARGs can be recursively extracted.

$\mathcal{P}(\mathsf{crs}_1, x_1, \ldots, x_k, \omega_1, \ldots, \omega_k) \to \pi$. The prover $\mathcal{P}$ is a poly-time algorithm that takes as input crs_1 (which is the first string in crs), any (unbounded) number of instance-witness pairs of arbitrary size (since the batch size nor the instance size are no longer fixed at setup time). It runs in time $\mathsf{poly}\left(\lambda, (|x_i|, |w_i|)_{i \in [k]}\right)$ and outputs a proof π.

$\mathsf{AggProve}\left((\mathsf{crs}_i)_{i \in [d']}, X^{(1)}, \ldots, X^{(\ell)}, \pi^{(1)}, \ldots, \pi^{(\ell)}\right) \to \pi$**.** This proof combiner is a poly-time algorithm that takes as input the $(\mathsf{crs})_{i \in [d']}$, a sequence of arbitrarily many *instance-trees* $X^{(1)}, \ldots, X^{(\ell)}$ (as defined below), of maximal depth $d'-1$, and corresponding $\mathsf{multi\text{-}BARG}$ proofs $\pi^{(1)}, \ldots, \pi^{(\ell)}$. It outputs a (combined) $\mathsf{multi\text{-}BARG}$ proof π.

Definition 6.1. *An instance-tree X is a tree of varying arity, where each leaf node v is associated with an instance x_v, and each intermediate node u corresponds to a* $\mathsf{multi\text{-}BARG}$ *proof that certifies the validity of the sub-tree rooted at u (though these* $\mathsf{multi\text{-}BARG}$ *proofs are not included in X).*

We note that the depth of an instance-tree X represents the number of hops taken in order to compute a proof π for X. Therefore, the proof combiner takes as input a sequence of instance-trees along with their corresponding $\mathsf{multi\text{-}BARG}$ proofs, where each instance-tree encodes not only the instances in the leaves, but also the history of how the previous proofs were combined, including the number of hops taken so far.

$\mathcal{V}((\mathsf{crs}_i)_{i \in [d']}, X, \pi) \to \{0, 1\}$**.** The verifier $\mathcal{V}$ is a poly-time algorithm that takes as input $(\mathsf{crs}_i)_{i \in [d']}$, an instance-tree X of depth d', and a (combined) proof π, and outputs 0/1 (corresponding to reject or accept).

Notation 6.2. *In what follows, when we refer to a tree T we always think of a tree where each of its leaves, denoted by v, is associated with a parameter $n_v = n_v(\lambda)$.*[14] *When we refer to a poly-size*

[13] We note that it was unnecessary for Gen to take k and n as input to begin with, since it could have produced a crs corresponding to every $k, n \in \{2^i\}_{i \in [\lambda]}$, and by padding we can assume w.l.o.g. that indeed k, n are powers of 2. Indeed, originally Gen took k, n as input only for the sake of simplicity.

[14] Different leaves can be associated with different parameters.

tree, we mean that the number of nodes in the tree is $\leq \mathsf{poly}(\lambda)$ *and* $n_v = n_v(\lambda) \leq \mathsf{poly}(\lambda)$ *for every leaf* v*. For any tree* T*, we let* $\mathsf{path}(T)$ *be the set that consists of all the possible paths from the root to a leaf in* T*. We say that an instance-tree* X *is consistent with* T *if* X *has the exact same tree structure as* T *and for each leaf* $v \in T$ *the instance* x_v *in* X *(corresponding to the leaf* v*) is of size* n_v*. For any instance-tree* X *that is consistent with* T *and any* $(i_1, \ldots, i_d) \in \mathsf{path}(T)$ *we let* $X_{i_1,\ldots,i_d}$ *denote the instance in the leaf corresponding to the path* $(i_1, \ldots, i_d)$*.*

In addition, for any instance-trees $X^{(1)}, \ldots, X^{(\ell)}$ *we denote by* $X = \left(X^{(1)}, \ldots, X^{(\ell)}\right)$ *the instance-tree that combines all the* ℓ *instance-trees* $X^{(1)}, \ldots, X^{(\ell)}$ *by adding a root with arity* ℓ*, whose* i*'th child is the root of* $X^{(i)}$*.*

Definition 6.3. *A rate-*1 *multi-hop* seBARG *for an* NP *language* $\mathcal{L}$ *is required to satisfy all the properties in definitions 3.3 and 3.6, adapted to our syntax (all stated below for the sake of completeness), with an additional completeness and compactness guarantee for* combined *proofs.*

Efficiency. *For every* $i \in [d]$*, the size of* $(\mathsf{crs}_i, \mathsf{td}_i)$ *is at most* $\mathsf{poly}(\lambda)$*, and the size of a (combined) proof corresponding to an instance-tree* X *of depth* d' *is at most* $m + d' \cdot \mathsf{poly}(\lambda, \log|X|)$*, where* m *is the maximal witness length of all the leaf instances in* X*.*

Completeness. *For any* $\lambda \in \mathbb{N}$ *any* $d \in [2^\lambda]$*, any instance-tree* X *of size* $\leq 2^\lambda$ *and depth* $d' \leq d$*, and any corresponding valid witness* W

$$\Pr\left[\begin{array}{l|ll} \mathcal{V}\left((\mathsf{crs}_i)_{i\in[d']}, X, \pi\right) = 1 & : & (\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{Gen}(1^\lambda, d, (i_1, \ldots, i_d)), \\ & & \textit{parse } \mathsf{crs} = (\mathsf{crs}_i)_{i\in[d]}, \\ & & \pi \leftarrow \mathsf{AggProve}\left((\mathsf{crs}_i)_{i\in[d']}, X, W\right) \end{array}\right] = 1$$

where $\mathsf{AggProve}\left((\mathsf{crs}_i)_{i\in[d']}, X, W\right)$ *is defined by induction on* d' *as follows:*

If $d' = 1$ *then parse* $X = (x_1, \ldots, x_\ell)$ *and* $W = (w_1 \ldots, w_\ell)$ *and output*

$$\mathsf{AggProve}(\mathsf{crs}_1, X, W) = \mathcal{P}(\mathsf{crs}_1, x_1, \ldots, x_\ell, w_1 \ldots, w_\ell).$$

If $d' > 1$ *then parse* $X = \left(X^{(1)}, \ldots, X^{(\ell)}\right)$ *and* $W = \left(W^{(1)}, \ldots, W^{(\ell)}\right)$*, where* $W^{(i)}$ *is the witness corresponding to the sub-tree instance* $X^{(i)}$*. Denote by* d_i *the depth of* $X^{(i)}$*.*

For every $i \in [\ell]$ *compute by induction*

$$\pi^{(i)} = \mathsf{AggProve}\left((\mathsf{crs}_j)_{j\in[d_i]}, X^{(i)}, W^{(i)}\right)$$

and output

$$\pi = \mathsf{AggProve}\left((\mathsf{crs}_i)_{i\in[d']}, X^{(1)}, \ldots, X^{(\ell)}, \pi^{(1)}, \ldots, \pi^{(\ell)}\right).$$

Index hiding. *For any poly-size adversary* $\mathcal{A}$*, any polynomial* $d = \mathsf{poly}(\lambda)$*, any poly-size tree* T *of depth* $d' \leq d$*, and any sets of indices* $I_0 = (i_{0,1}, \ldots, i_{0,d}), I_1 = (i_{1,1}, \ldots, i_{1,d}) \in \mathsf{path}(T)$*, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for every* $\lambda \in \mathbb{N}$*,*

$$\Pr\left[\mathcal{A}(\mathsf{crs}) = b : \begin{array}{l} b \leftarrow \{0,1\}, \\ (\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{Gen}(1^\lambda, d, I_b) \end{array}\right] \leq \frac{1}{2} + \mathsf{negl}(\lambda).$$

Somewhere argument of knowledge. *There exists a* PPT *extractor* $\mathcal{E}$ *such that for any poly-size adversary* $\mathcal{A}$*, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for any polynomial* $d = \mathsf{poly}(\lambda)$*, any poly-size tree* T *of depth* $d' \leq d$*, and any set of indices* $(i_1, \ldots, i_{d'}) \in \mathsf{path}(T)$ *and* $i_{d'+1} \ldots, i_d \in [2^\lambda]$*, for every* $\lambda \in \mathbb{N}$*,*

$$\Pr\left[\begin{array}{ll} \mathcal{V}(\mathsf{crs}, X, \pi) = 1 & (\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{Gen}(1^\lambda, d, (i_1, \ldots, i_d)) \\ \wedge\ X \textit{ is consistent with } T & : \ (X, \pi) = \mathcal{A}(\mathsf{crs}) \\ \wedge\ w^* \textit{ is not a valid witness for } X_{i_1,\ldots,i_{d'}} \in \mathcal{L} & w^* \leftarrow \mathcal{E}\left(\mathsf{td}, X, \pi\right) \end{array}\right] \leq \mathsf{negl}(\lambda).$$

Remark 6.1. We note that the above somewhere argument of knowledge property implies the following semi-adaptive soundness property:

For any poly-size adversary $\mathcal{A}$, any polynomial $d = \mathsf{poly}(\lambda)$, any poly-size tree T of depth $d' \leq d$, and any set of indices $(i_1, \ldots, i_{d'}) \in \mathsf{path}(T)$ and $i_{d'+1} \ldots, i_d \in [2^\lambda]$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that for every $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{ll} \mathcal{V}(\mathsf{crs}, X, \pi) = 1 & (\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{Gen}(1^\lambda, d, (i_1, \ldots, i_d)) \\ \wedge\ X \text{ is consistent with } T & : \ (X, \pi) = \mathcal{A}(\mathsf{crs}) \\ \wedge\ X_{i_1,\ldots,i_{d'}} \notin \mathcal{L} & \end{array}\right] \leq \mathsf{negl}(\lambda).$$

6.2 Construction and Analysis

In this section we construct a multi-hop seBARG from any rate-1 seBARG (Definitions 3.3 and 3.6). We do this in two steps. First, we show how to convert the rate-1 seBARG into a an intermediate primitive that we call a rate-1 *single-hop* seBARG, denoted by single-seBARG, and then we show how to convert a rate-1 single-seBARG into a *multi-hop* seBARG.

Single-hop seBARG**.** A single-seBARG scheme is very similar to a seBARG scheme, the only difference being that it enables batching instances and witnesses of varying lengths, and these lengths, as well as batch size k, can be decided at batching time (and are no longer fixed at setup time). The reason why a rate-1 single-seBARG can be directly obtained from a rate-1 seBARG scheme follows from the following two observations about a rate-1 seBARG scheme.

Observation 1. The fact that the running time of a rate-1 seBARG.Gen grows only *poly-logarithmically* with the number of instances k and the input length n (see Definition 3.6) implies that one could generically convert this seBARG into another seBARG scheme in which the number of instances and the input length are not a priori bounded. To see this, we first observe that by standard padding we can consider only k and n which are powers of 2. Therefore, we can run seBARG.Gen λ^2 times, one for each $k, n \in \{2^i\}_{i \in [\lambda]}$. The prover can then compute the seBARG proof using the crs corresponding to appropriate k, n. This gives a single-hop seBARG scheme for k and n that are not fixed in advance.

Observation 2. Our second observation is that any rate-1 seBARG can be converted into one that allows batching instances of unequal lengths, by padding all the instances and witnesses to be of the same length.

By combining the above two observations we can convert our rate-1 seBARG scheme into a rate-1 single-seBARG scheme, which enables batching instances of varying lengths where the length of these instances as well as the batch size are not a priori fixed.

The construction of our rate-1 multi-hop seBARG. Fix any NP language $\mathcal{L}$. We construct a rate-1 multi-hop seBARG scheme for $\mathcal{L}$, denoted by multi-BARG, which uses as a building block a rate-1 single-seBARG scheme for the NP language $\mathcal{L}'$. This NP language $\mathcal{L}'$ contains $\mathcal{L}$ and is defined recursively below. Loosely speaking, any instance in $\mathcal{L}'$ is an instance-tree X of some depth d', along with $(\mathsf{crs}_i)_{i\in[d']}$ (if $d'=0$ then there is no crs associated with the instance) and a valid witness is a valid aggregated proof corresponding to $(\mathsf{crs}_i)_{i\in[d']}$ (where if $d'=0$ then a valid proof is simply a valid witness corresponding to $\mathcal{L}$).

$\mathsf{Gen}(1^\lambda, d, I) \to (\mathsf{crs}, \mathsf{td})$. This PPT algorithm parses $I=(i_1,\ldots,i_d)$, it samples for every $j\in[d]$

$$(\mathsf{crs}_j, \mathsf{td}_j) \leftarrow \text{single-seBARG.}\mathsf{Gen}(1^\lambda, i_j),$$

and it outputs $\mathsf{crs}=(\mathsf{crs}_1,\ldots,\mathsf{crs}_d)$ and $\mathsf{td}=(\mathsf{td}_1,\ldots,\mathsf{td}_d)$.

$\mathcal{P}(\mathsf{crs}_1, x_1, \ldots, x_k, \omega_1, \ldots, \omega_k) \to \pi$. This poly-time algorithm outputs

$$\pi \leftarrow \text{single-seBARG.}\mathcal{P}(\mathsf{crs}_1, x_1, \ldots, x_k, \omega_1, \ldots, \omega_k).$$

$\mathsf{AggProve}\left(\mathsf{crs}, X^{(1)},\ldots,X^{(\ell)}, \pi^{(1)},\ldots,\pi^{(\ell)}\right) \to \pi$. For every $i\in[\ell]$ denote by d_i the depth of $X^{(i)}$ and let $d'=\max\{d_i\}+1$. If $d'>d$ then abort. Otherwise, output

$$\pi \leftarrow \text{single-seBARG.}\mathcal{P}\left(\mathsf{crs}_{d'}, \left((\mathsf{crs}_j)_{j\in[d_i]}, X^{(i)}\right)_{i\in[\ell]}, \left(\pi^{(i)}\right)_{i\in[\ell]}\right)$$

where a valid witness corresponding to $\left((\mathsf{crs}_j)_{j\in[d_i]}, X^{(j)}\right)\in\mathcal{L}'$ is $\pi^{(i)}$ such that

$$\mathcal{V}\left((\mathsf{crs}_j)_{j\in[d_i]}, X^{(i)}, \pi^{(i)}\right)=1$$

$\mathcal{V}(\mathsf{crs}, X, \pi) \to 0/1$. Parse $\mathsf{crs}=(\mathsf{crs}_1,\ldots,\mathsf{crs}_{d'})$ and parse $X=\left(X^{(1)},\ldots,X^{(\ell)}\right)$. Let $d_i\in[d]$ be the depth of the instance-tree $X^{(i)}$. If $d'\neq\max\{d_i\}_{i[\ell]}+1$ then output 0. Else, output

$$\text{single-seBARG.}\mathcal{V}\left(\mathsf{crs}_{d'}, \left((\mathsf{crs}_j)_{j\in[d_i]}, X^{(i)}\right)_{i\in[\ell]}, \pi\right).$$

Remark 6.2 (The language $\mathcal{L}'$)**.** At first there may appear to be a circularity in the definition of the NP language $\mathcal{L}'$, since on the one hand its definition depends on the multi-hop multi-BARG scheme, and on the other hand, multi-BARG uses as a building block a rate-1 single-seBARG for $\mathcal{L}'$. The reason this is not an issue is that both $\mathcal{L}'$ and multi-BARG are formally defined by induction on the depth d' of the instance-trees, starting with $d'=0$. Namely, we define by induction for every $d'\in\mathbb{N}$, $\mathcal{L}'_{d'}$ and $\mathcal{V}_{d'}$, where $\mathcal{L}'_{d'}$ contains only instance-trees of depth at most d' (where $\mathcal{L}'_0=\mathcal{L}$), and $\mathcal{V}_{d'}$ takes as input only an instance-tree of depth d', and it runs single-seBARG.$\mathcal{V}$ with instance-trees of depth $d'-1$ (where $\mathcal{L}_{d'-1}$ is already defined by induction).

Theorem 6.4. *If* single-seBARG *is a rate-1 single-hop* seBARG *scheme then* multi-BARG *is a rate-1 multi-hop* seBARG *scheme (definition 6.3).*

Proof. In what follows we prove that multi-BARG satisfies all the desired properties:

Efficiency. Follows immediately from the fact that the underlying single-seBARG is rate-1.

Completeness. Follows directly from the completeness of the underlying single-seBARG scheme.

Index-hiding. Follows immediately from the index-hiding property of the underlying single-seBARG scheme (via a standard hybrid argument).

Somewhere argument of knowledge. Let single-seBARG.$\mathcal{E}$ denote the PPT extractor corresponding to the underlying single-seBARG scheme. We define an extractor $\mathcal{E}$ that given (td, X, π) does the following:

1. Parse $\mathsf{td} = (\mathsf{td}_1, \ldots, \mathsf{td}_d)$ and suppose that td corresponds to indices $(i_1, \ldots, i_d)$.
2. Parse $X = \left(X^{(1)}, \ldots, X^{(\ell)}\right)$, and let d' denote the depth of X.
3. For every $j \in [d']$ we denote by d_j the depth of the subtree $X^{(i_{d'}, \ldots, i_{d'+1-j})}$.

   Let $j^* \in [d']$ be the smallest index such that $d_j = 0$ for for every $j \geq j^*$ (i.e., j^* is the length of the path $(i_{d'}, \ldots, i_1)$ in X until we reach a leaf).
4. Define
$$\pi^{(i_{d'})} = \text{single-seBARG}.\mathcal{E}(\mathsf{td}_{d'}, X, \pi).$$
   Intuitively, if π is a valid multi-BARG proof for X then with overwhelming probability $\pi^{(i_{d'})}$ is a valid multi-BARG proof for the instance-tree $X^{(i_{d'})}$.
5. For every $j \in [j^* - 1]$ we define (by induction, starting with $j = 1$)
$$\pi^{(i_{d'}, \ldots i_{d'-j})} = \text{single-seBARG}.\mathcal{E}\left(\mathsf{td}_{d_j}, X^{(i_{d'}, \ldots, i_{d'+1-j})}, \pi^{(i_{d'}, \ldots i_{d'-j+1})}\right)$$
   where $X^{(i_{d'}, \ldots, i_{d'-j})}$ is defined to be the subtree of X obtained by going down from the root on the path $(i_{d'}, \ldots, i_{d'-j})$.

Fix any poly-size adversary $\mathcal{A}$ and any polynomial $d = \mathsf{poly}(\lambda)$. In what follows we argue by induction that for every $d' \leq d$ there exists a negligible function $\mu_{d'}$, such that for any poly-size tree T of depth d', and any set of indices $(i_1, \ldots, i_{d'}) \in \mathsf{path}(T)$ and $i_{d'+1} \ldots, i_d \in [2^\lambda]$ it holds that for every $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{ll} \mathcal{V}((\mathsf{crs}_j)_{j\in[d']}, X, \pi) = 1 & (\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{Gen}(1^\lambda, d, (i_1, \ldots, i_d)) \\ \wedge\ X \text{ is consistent with } T & : \ (X, \pi) = \mathcal{A}(\mathsf{crs}) \\ \wedge\ w^* \text{ is not a valid witness for } X_{i_1,\ldots,i_{d'}} \in \mathcal{L} & w^* \leftarrow \mathcal{E}\left(\mathsf{td}, X, \pi\right) \end{array}\right] \leq \mu_{d'}(\lambda). \tag{14}$$

Base case: $d' = 1$: This follows immediately from the somewhere argument of knowledge of the underlying single-seBARG scheme.

Induction step: Suppose that Equation (14) holds for every $j < d'$ and we prove that it holds for d'. To this end, the somewhere argument of knowledge of the underlying single-seBARG scheme implies that there exists a negligible function ν such that for every $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l} \mathcal{V}((\mathsf{crs}_j)_{j\in[d']}, X, \pi) = 1 \\ \wedge\ X \text{ is consistent with } T \\ \wedge\ \mathcal{V}\left((\mathsf{crs}_j)_{j\in[d_1]}, X^{(i_{d'})}, \pi^{(i_{d'})}\right) = 1 \end{array} : \begin{array}{l} (\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{Gen}(1^\lambda, d, (i_1, \ldots, i_d)) \\ (X, \pi) = \mathcal{A}(\mathsf{crs}) \\ \pi^{(i_{d'})} \leftarrow \text{single-seBARG.}\mathcal{E}\left(\mathsf{td}_{d'}, X, \pi\right) \\ \text{parse } \mathsf{crs} = (\mathsf{crs}_1, \ldots, \mathsf{crs}_d) \end{array}\right] \leq \nu(\lambda). \quad (15)$$

Consider the poly-size adversary $\mathcal{A}'$ that given crs does the following:

1. Parse $\mathsf{crs} = (\mathsf{crs}_1, \ldots, \mathsf{crs}_d)$.
2. Generate $(\mathsf{crs}^*, \mathsf{td}^*) \leftarrow \text{single-seBARG.}\mathsf{Gen}(1^\lambda, i_{d'})$.
3. Let $\mathsf{crs}^* = (\mathsf{crs}_1, \ldots, \mathsf{crs}_{d'-1}, \mathsf{crs}^*, \mathsf{crs}_{d'+1}, \ldots, \mathsf{crs}_d)$.
4. Compute $(X, \pi) = \mathcal{A}(\mathsf{crs}^*)$.
5. Compute $\pi^{(i_{d'})} = \text{single-seBARG.}\mathcal{E}(\mathsf{td}^*, X, \pi)$.
6. Output $\left(X^{(i_{d'})}, \pi^{(i_{d'})}\right)$.

By the induction hypothesis there exists a negligible function μ such that for every $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l} \mathcal{V}\left((\mathsf{crs}_j)_{j\in[d_1]}, X^{(i_{d'})}, \pi^{(i_{d'})}\right) = 1 \\ \wedge\ X^{(i_{d'})} \text{ is consistent with } T^{(i_{d'})} \\ \wedge\ \mathcal{V}\left((\mathsf{crs}_j)_{j\in[d_2]}, X^{(i_{d'}, i_{d'-1})}, \pi^{(i_{d'}, i_{d'-1})}\right) = 1 \end{array} : \begin{array}{l} (\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{Gen}(1^\lambda, d, (i_1, \ldots, i_d)) \\ \left(X^{(i_{d'})}, \pi^{(i_{d'})}\right) = \mathcal{A}^*(\mathsf{crs}) \\ \pi^{(i_{d'}, i_{d'-1})} \leftarrow \mathcal{E}\left(\mathsf{td}^*, X^{(i_{d'})}, \pi^{(i_{d'})}\right) \end{array}\right] \leq \nu(\lambda).$$

This, together with Equation (15), implies the induction step (i.e., Equation (14)).

□

6.3 Hashed Multi-Hop seBARGs

6.3.1 Definition

A hashed multi-BARG is one where the verifier is required to verify the proof without being given the instances (in the clear), and instead being given only a hash of the instances. In Section 7 we use a hashed multi-BARG to construct an IVC scheme. For this application, it suffices to consider hashed multi-BARG where all the instances are of the same size n and where the number of instances being BARGed is a fixed parameter k. Therefore, for simplicity we fix n and k in setup time. We also consider a hashed multi-BARG where the number of hops is at most λ. This suffices for our IVC application since there we will combine the seBARGs in a tree-like structure with fan-out k, so a depth λ tree is sufficient to accumulate k^λ many proofs.

We note that one could define the more general notion where n and k can vary adaptively (as in Section 6.1), and we chose simplicity over generality.

Syntax. A hashed multi-BARG scheme, corresponding to an NP language $\mathcal{L}$, is associated with a rate-1 SEH family

$$\mathsf{SEH} = (\mathsf{SEH.Gen}, \mathsf{SEH.Hash}, \mathsf{SEH.Open}, \mathsf{SEH.Verify}, \mathsf{SEH.Extract}).$$

Remark 6.3. For simplicity (and w.l.o.g.) we assume that each hash value v computed by SEH.Hash includes a depth parameter $d \in \mathbb{N}$ which corresponds to the number of times SEH.Hash was applied in order to compute v. Namely, we assume that there exists a poly-time computable function $d(\cdot)$ such that for every hk generated by SEH.Gen, and for every $\mathsf{v}_1, \ldots, \mathsf{v}_N$,

$$\mathsf{SEH.Hash}(\mathsf{hk}, \mathsf{v}_1, \ldots, \mathsf{v}_N) = \mathsf{v}$$

where $d(\mathsf{v}) = \max\{d(\mathsf{v}_i)\}_{i\in[N]} + 1$, and we also assume that if v_i is a "fresh" input then it is encoded so that $d(\mathsf{v}_i) = 0$. This depth parameter d indicates the number of times that SEH.Hash was applied (i.e., the number of hops made) to compute the hash value.

A multi-BARG is also associated with the following algorithms:

$\mathsf{Gen}(1^\lambda, n, k, (i_1, \ldots, i_\lambda)) \to (\mathsf{crs}, \mathsf{hk}, \mathsf{td})$. This is a PPT algorithm that takes as input a security parameter 1^λ, an instance size n, the number of instances k, and a sequence of λ extraction indices $i_1, \ldots, i_\lambda \in [k]$. It outputs a $\mathsf{crs} = (\mathsf{crs}_d)_{d\in[\lambda]}$, a hash key hk and a trapdoor td which consists of two parts, denoted by $\mathsf{td} = (\mathsf{seBARG.td}, \mathsf{SEH.td})$, one corresponding to crs and the other corresponding to hk.

$\mathcal{P}(\mathsf{crs}, \mathsf{hk}, x_1, \ldots, x_k, \omega_1, \ldots, \omega_k) \to (\mathsf{v}, \pi)$. This is a poly-time prover algorithm that takes as input crs, a hash key hk, instances $x_1, \ldots, x_k$ each of size n, and corresponding witnesses $\omega_1, \ldots, \omega_k$, and outputs a hash value v and a proof π.

$\mathsf{AggProve}\left(\mathsf{crs}, \mathsf{hk}, \mathsf{v}^{(1)}, \ldots, \mathsf{v}^{(k)}, \pi^{(1)}, \ldots, \pi^{(k)}\right) \to (\mathsf{v}, \pi)$**.** This proof combiner is a poly-time algorithm that takes as input crs, a hash key hk, a sequence of hash values $\mathsf{v}^{(1)}, \ldots, \mathsf{v}^{(k)}$, along with corresponding proofs $\pi^{(1)}, \ldots, \pi^{(k)}$, and outputs a new hash value v and a corresponding proof π.

$\mathcal{V}(\mathsf{crs}, \mathsf{hk}, \mathsf{v}, \pi) \to 0/1$. The verifier $\mathcal{V}$ is a poly-time algorithm that takes as input $(\mathsf{crs}_j)_{j\in[d]}$, a hash key hk, a hash value v s.t. $d(\mathsf{v}) = d$, and a proof π and outputs a bit to signal whether the proof is valid or not.

Remark 6.4. The fact that in a hashed multi-BARG the parameters n and k are fixed in advance, implies that the instance tree is a k-ary tree where all the leaves correspond to instances of size n. Since this tree is fixed (except its depth which is some $d \in [\lambda]$) we do not need to include it as an input (and all we need is to include its depth d which we assume is encoded in v). We note that in a multi-BARG we did need to include it as part of the input since n and k were adaptively chosen and were changing from hop to hop.

The definition of a hashed multi-BARG is similar to that of a seBARG (adapted to our syntax), and the main change is to the somewhere argument of knowledge property.

Definition 6.5. *A rate-1 hashed* multi-BARG *for an* NP *language $\mathcal{L}$ is required to satisfy the following properties.*

Efficiency. *The output* (v, π) *generated by the prover, or the aggregate prover, is of size at most* $n \cdot \mathsf{poly}(\lambda) + m$ *where* n *is the instance size and* m *is the witness size.*[15]

Completeness. *For any* $\lambda \in \mathbb{N}$ *any* $n, k, \ell \in [2^\lambda]$ *such that* $k \geq 2$*, any* ℓ *instances* $x_1, \ldots, x_\ell$ *each of size* n*, and corresponding valid witnesses* $w_1, \ldots, w_\ell$*, and any* $i_1, \ldots, i_\lambda \in [k]$*,*

$$\Pr\left[\begin{array}{l} \mathcal{V}\left(\mathsf{crs}, \mathsf{hk}, \mathsf{v}, \pi\right) = 1 \\ \end{array} : \begin{array}{l} (\mathsf{crs}, \mathsf{hk}, \mathsf{td}) \leftarrow \mathsf{Gen}(1^\lambda, n, k, (i_1, \ldots, i_\lambda)), \\ (\mathsf{v}, \pi) \leftarrow \mathsf{AggProve}\left(\mathsf{crs}, x_1, \ldots, x_\ell, w_1, \ldots, w_\ell\right) \end{array}\right] = 1$$

where $\mathsf{AggProve}\left(\mathsf{crs}, x_1, \ldots, x_\ell, w_1, \ldots, w_\ell\right)$ *is defined recursively, as follows: First assume that* ℓ *is a multiple of* k *(this can be achieved via padding). If* $\ell = k$ *then*

$$\mathsf{AggProve}\left(\mathsf{crs}, \mathsf{hk}, x_1, \ldots, x_\ell, w_1, \ldots, w_\ell\right) \triangleq \mathcal{P}\left(\mathsf{crs}, \mathsf{hk}, x_1, \ldots, x_\ell, w_1, \ldots, w_\ell\right).$$

Otherwise, first compute for every $i \in \left[\frac{\ell}{k}\right]$

$$\left(\mathsf{v}^{(i)}, \pi^{(i)}\right) = \mathcal{P}(\mathsf{crs}, \mathsf{hk}, x_{(i-1)\cdot k+1}, \ldots, x_{i\cdot k}, w_{(i-1)\cdot k+1}, \ldots, w_{i\cdot k})$$

and then compute $\mathsf{AggProve}\left(\mathsf{crs}, \mathsf{hk}, \mathsf{v}^{(1)}, \ldots, \mathsf{v}^{(\ell/k)}, \pi^{(1)}, \ldots, \pi^{(\ell/k)}\right)$*, recursively.*

Index hiding. *For any poly-size adversary* $\mathcal{A}$*, any polynomials* $n = n(\lambda)$ *and* $k = k(\lambda)$*, and any sets of indices* $I_0 = (i_{0,1}, \ldots, i_{0,\lambda}), I_1 = (i_{1,1}, \ldots, i_{1,\lambda}) \in \mathsf{path}(T)$*, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for every* $\lambda \in \mathbb{N}$*,*

$$\Pr\left[\mathcal{A}(\mathsf{crs}, \mathsf{hk}) = b : \begin{array}{l} b \leftarrow \{0,1\}, \\ (\mathsf{crs}, \mathsf{hk}, \mathsf{td}) \leftarrow \mathsf{Gen}(1^\lambda, n, k, I_b) \end{array}\right] \leq \frac{1}{2} + \mathsf{negl}(\lambda).$$

Somewhere argument of knowledge. *There exists a* PPT *extractor* $\mathcal{E}$ *such that for any poly-size adversary* $\mathcal{A}$*, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for any polynomials* $n = n(\lambda)$ *and* $k = k(\lambda)$*, and any sequence of indices* $I = (i_1, \ldots, i_\lambda) \in [k]^\lambda$*, for every* $\lambda \in \mathbb{N}$*,*

$$\Pr\left[\begin{array}{l} \mathcal{V}(\mathsf{crs}, \mathsf{hk}, \mathsf{v}, \pi) = 1 \\ \wedge \left(x \neq \mathsf{SEH.Extract}(\mathsf{SEH.td}, \mathsf{v}, i)\right. \\ \quad \left.\vee\ w \text{ is not a valid witness for } x \in \mathcal{L}\right) \end{array} : \begin{array}{l} (\mathsf{crs}, \mathsf{hk}, \mathsf{td}) \leftarrow \mathsf{Gen}(1^\lambda, n, k, I) \\ (\mathsf{v}, \pi) = \mathcal{A}(\mathsf{crs}, \mathsf{hk}) \\ (x, w) \leftarrow \mathcal{E}\left(\mathsf{td}, \mathsf{v}, \pi\right) \\ \text{parse } \mathsf{td} = (\mathsf{seBARG.td}, \mathsf{SEH.td}) \end{array}\right] \leq \mathsf{negl}(\lambda).$$

6.3.2 Construction and Analysis

Fix any NP language $\mathcal{L}$ and any rate-1 somewhere extractable hash family

$$\mathsf{SEH} = (\mathsf{SEH.Gen}, \mathsf{SEH.Hash}, \mathsf{SEH.Open}, \mathsf{SEH.Verify}, \mathsf{SEH.Extract}).$$

We construct a hashed multi-BARG scheme for $\mathcal{L}$ w.r.t. the SEH family. Our scheme uses as a building block a rate-1 seBARG scheme

$$\mathsf{seBARG} = (\mathsf{seBARG.Gen}, \mathsf{seBARG}.\mathcal{P}, \mathsf{seBARG}.\mathcal{V}, \mathsf{seBARG.Extract})$$

for the NP language $\mathcal{L}'$ defined recursively below.[16]

[15] Our scheme allows for at most λ hops so we do not need to incorporate the depth as an additional parameter.

[16] The language $\mathcal{L}'$ is somewhat similar to the NP language $\mathcal{L}'$ defined in the construction of the multi-BARG scheme in Section 6.2, except that here the instance-tree is hashed.

$\mathsf{Gen}(1^\lambda, n, k, (i_1, \ldots, i_\lambda)) \to (\mathsf{crs}, \mathsf{hk}, \mathsf{td})$. This PPT algorithm does the following:

1. For every $d \in [\lambda]$ sample $(\mathsf{hk}_d, \mathsf{SEH.td}_d) \leftarrow \mathsf{SEH.Gen}(1^\lambda, n_{d-1} \cdot k, I_{i_d})$, where $n_0 = n$ and for every $d \in [\lambda]$, n_d is the size of hash values outputted by $\mathsf{SEH.Hash}(\mathsf{hk}_d, \cdot)$, i.e., hash values of depth d, and
$$I_{i_d} \triangleq \{n_{d-1} \cdot (i_d - 1) + 1, \ldots, n_{d-1} \cdot i_d\}.$$
Note that $n_d \leq n_{d-1} + \mathsf{poly}(\lambda)$ and thus $n_\lambda \leq n + \mathsf{poly}(\lambda)$.
2. For every $d \in [\lambda]$ sample $(\mathsf{crs}_d, \mathsf{td}_d) \leftarrow \mathsf{seBARG.Gen}(1^\lambda, n'_d, k, i_d)$, where n'_d for $d \in [\lambda]$ is defined below.
3. Let $\mathsf{crs} = (\mathsf{crs}_1, \ldots, \mathsf{crs}_\lambda)$, $\mathsf{hk} = (\mathsf{hk}_1, \ldots, \mathsf{hk}_\lambda)$, and $\mathsf{td} = (\mathsf{td}_1, \ldots, \mathsf{td}_\lambda, \mathsf{SEH.td}_1, \ldots, \mathsf{SEH.td}_\lambda)$.
4. Output $(\mathsf{crs}, \mathsf{hk}, \mathsf{td})$.

$\mathcal{P}(\mathsf{crs}, \mathsf{hk}, x_1, \ldots, x_k, \omega_1, \ldots, \omega_k) \to \pi$. This poly-time algorithm does the following:

1. Parse $\mathsf{hk} = (\mathsf{hk}_1, \ldots, \mathsf{hk}_\lambda)$.
2. Compute $\mathsf{v} = \mathsf{SEH.Hash}(\mathsf{hk}_1, (x_1, \ldots, x_k))$.
Note that $|\mathsf{v}| \leq n + \mathsf{poly}(\lambda)$.
3. For every $i \in [k]$ compute $\rho_i = \mathsf{SEH.Open}(\mathsf{hk}_1, (x_1, \ldots, x_k), I_i)$, where I_i corresponds to the coordinates of x_i, i.e.,
$$I_i \triangleq \{n \cdot (i-1) + 1, \ldots, n \cdot i\}.$$
Note that $|\rho_i| \leq \mathsf{poly}(\lambda)$ (see Lemma 3.2 in Section 3.1).
4. Parse $\mathsf{crs} = (\mathsf{crs}_1, \ldots, \mathsf{crs}_\lambda)$.
5. Compute $\pi = \mathsf{seBARG}.\mathcal{P}\left(\mathsf{crs}_1, (\mathsf{hk}_1, \mathsf{v}, i)_{i\in[k]}, (x_i, \rho_i, w_i)_{i\in[k]}\right)$, where instances of $\mathcal{L}'$ have length
$$n'_1 = |(\mathsf{hk}_1, \mathsf{v}, i)| \leq n + \mathsf{poly}(\lambda)$$
and a valid witness (x_i, ρ_i, w_i) satisfies the following conditions:
$$\mathsf{SEH.Verify}(\mathsf{hk}_1, \mathsf{v}, I_i, x_i, \rho_i) = 1 \quad \wedge \quad (x_i, w_i) \in \mathcal{R}_\mathcal{L}.$$
The fact that seBARG and SEH both have rate 1 implies that π is of size
$$|\pi| \leq |(x_i, \rho_i, w_i)| + \mathsf{poly}(\lambda) \leq n + m + \mathsf{poly}(\lambda),$$
as desired.
6. Output (v, π).

$\mathsf{AggProve}\left(\mathsf{crs}, \mathsf{hk}, \mathsf{v}^{(1)}, \ldots, \mathsf{v}^{(k)}, \pi^{(1)}, \ldots, \pi^{(k)}\right) \to (\mathsf{v}, \pi)$. This poly-time algorithm does the following:

1. Parse $\mathsf{hk} = (\mathsf{hk}_1, \ldots, \mathsf{hk}_\lambda)$.
2. Check that $d\left(\mathsf{v}^{(i)}\right) = d\left(\mathsf{v}^{(j)}\right)$ for every $i, j \in [k]$ and that this depth, denoted by d, is smaller than λ. If this is not the case then abort.

3. Compute $\mathsf{v} = \mathsf{SEH.Hash}\left(\mathsf{hk}_{d+1}, \left(\mathsf{v}^{(1)}, \ldots, \mathsf{v}^{(k)}\right)\right)$. Note that

$$|\mathsf{v}| \leq \left|\mathsf{v}^{(i)}\right| + \mathsf{poly}(\lambda) = n_d + \mathsf{poly}(\lambda) \leq n + \mathsf{poly}(\lambda).$$

4. For every $i \in [k]$ compute $\rho_i = \mathsf{SEH.Open}\left(\mathsf{hk}_{d+1}, \left(\mathsf{v}^{(1)}, \ldots, \mathsf{v}^{(k)}\right), I_i\right)$, where (as above) I_i corresponds to the coordinates of $\mathsf{v}^{(i)}$; i.e.,

$$I_i \triangleq \{n_d \cdot (i-1) + 1, \ldots, n_d \cdot i\}$$

and $|\rho_i| \leq \mathsf{poly}(\lambda)$.

5. Parse $\mathsf{crs} = (\mathsf{crs}_1, \ldots, \mathsf{crs}_\lambda)$.
6. Compute

$$\pi = \mathsf{seBARG}.\mathcal{P}\left(\mathsf{crs}_{d+1}, \begin{array}{l} \text{instances} = \left((\mathsf{crs}_j)_{j\in[d]}, (\mathsf{hk}_j)_{j\in[d+1]}, \mathsf{v}, i\right)_{i\in[k]} \\ \text{witnesses} = \left(\mathsf{v}^{(i)}, \rho_i, \pi^{(i)}\right)_{i\in[k]} \end{array}\right),$$

where instances of $\mathcal{L}'$ have length

$$n'_{d+1} = \left|(\mathsf{crs}_j)_{j\in[d]}, (\mathsf{hk}_j)_{j\in[d+1]}, \mathsf{v}, i\right| \leq n + \mathsf{poly}(\lambda)$$

and a valid witness $\left(\mathsf{v}^{(i)}, \rho_i, \pi^{(i)}\right)$ satisfies the following conditions:

(a) ρ_i is a valid opening of $\mathsf{v}^{(i)}$ w.r.t. $(\mathsf{hk}_{d+1}, \mathsf{v})$. Namely,

$$\mathsf{SEH.Verify}(\mathsf{hk}_{d+1}, \mathsf{v}, I_i, \mathsf{v}^{(i)}, \rho_i) = 1$$

(b) $\pi^{(i)}$ is a valid proof w.r.t. $\mathsf{v}^{(i)}$. Namely,

$$\mathsf{seBARG}.\mathcal{V}\left(\mathsf{crs}_d, \left((\mathsf{crs}_j)_{j\in[d-1]}, (\mathsf{hk}_j)_{j\in[d]}, \mathsf{v}^{(i)}, j\right)_{j\in[k]}, \pi^{(i)}\right) = 1.$$

Inductively we have that $\left|\pi^{(i)}\right| \leq n \cdot d \cdot \mathsf{poly}(\lambda) + m$. The fact that seBARG and SEH both have rate 1 implies that π is of size

$$\begin{aligned} |\pi| &= \left|\left(\mathsf{v}^{(i)}, \rho_i, \pi^{(i)}\right)\right| + \mathsf{poly}(\lambda) \leq n_d + \mathsf{poly}(\lambda) + (n \cdot d \cdot \mathsf{poly}(\lambda) + m) \\ &\leq n \cdot (d+1) \cdot \mathsf{poly}(\lambda) + m \end{aligned} \tag{16}$$

as desired.

7. Output (v, π).

$\mathcal{V}(\mathsf{crs}, \mathsf{hk}, \mathsf{v}, \pi) \to 0/1$**.** This polynomial time algorithm does the following:

1. Parse $\mathsf{crs} = (\mathsf{crs}_1, \ldots, \mathsf{crs}_\lambda)$ and $\mathsf{hk} = (\mathsf{hk}_1, \ldots, \mathsf{hk}_\lambda)$.
2. Let $d \in [\lambda]$ be the depth of v. If $d \geq \lambda$ then output 0.
3. Output $\mathsf{seBARG}.\mathcal{V}\left(\mathsf{crs}_{d+1}, \left(Z^{(1)}, \ldots, Z^{(k)}\right), \pi\right)$, where $Z^{(i)} = \left((\mathsf{crs}_j)_{j\in[d]}, (\mathsf{hk}_j)_{j\in[d+1]}, \mathsf{v}, i\right)$.

Theorem 6.6. *The above construction is a hashed* multi-BARG *scheme (according to Definition 6.5).*

Proof. In what follows we prove that our hashed multi-BARG satisfies all the desired properties:

Efficiency. Follows from the construction, together with the fact that the underlying seBARG scheme and SEH family are both rate-1.

Completeness. Follows directly from the completeness of the underlying seBARG scheme and the opening completeness of the underlying SEH family. and from the fact that the underlying single-seBARG scheme is rate-1.

Index-hiding. Follows immediately from the index-hiding property of the underlying single-seBARG scheme and the SEH family (via a standard hybrid argument).

Somewhere argument of knowledge. Let $\mathsf{seBARG}.\mathcal{E}$ denote the PPT extractor corresponding to the underlying seBARG scheme. We define an extractor $\mathcal{E}$ that given $(\mathsf{td}, \mathsf{v}, \pi)$ does the following:

1. Parse $\mathsf{td} = (\mathsf{td}_1, \ldots, \mathsf{td}_\lambda, \mathsf{SEH.td}_1, \ldots, \mathsf{SEH.td}_\lambda)$.
2. Let $d \in [\lambda]$ be the depth of v, and denote by $\mathsf{v}^{(d)} = \mathsf{v}$ and $\pi^{(d)} = \pi$.
3. For every $i \in [d-1]$ compute by backward induction the triplet

$$\left(\mathsf{v}^{(i)}, \rho_i, \pi^{(i)}\right) = \mathsf{seBARG}.\mathcal{E}\left(\mathsf{td}_{i+1}, \left(Z^{(j)}\right)_{j\in[k]}, \pi^{(i+1)}\right)$$

   where

$$Z^{(j)} = \left((\mathsf{crs}_\ell)_{\ell\in[d]}, (\mathsf{hk}_\ell)_{\ell\in[d+1]}, \mathsf{v}^{(i+1)}, j\right)$$

   and let $(x, \rho, w) = \mathsf{seBARG}.\mathcal{E}\left(\mathsf{td}_1, (\mathsf{hk}_1, \mathsf{v}_1, j)_{j\in[k]}, \pi^{(i)}\right)$.
4. Output (x, w).

We need to argue that for any poly-size adversary $\mathcal{A}$ there exists a negligible function negl such that for every $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l} \mathcal{V}(\mathsf{crs}, \mathsf{hk}, \mathsf{v}, \pi) = 1 \\ \wedge\ x = \mathsf{SEH.Extract}(\mathsf{SEH.td}, \mathsf{v}, i) \\ \wedge\ w \text{ is not a valid witness for } x \in \mathcal{L} \end{array} : \begin{array}{l} (\mathsf{crs}, \mathsf{hk}, \mathsf{td}) \leftarrow \mathsf{Gen}(1^\lambda, n, k, (i_1, \ldots, i_\lambda)) \\ (\mathsf{v}, \pi) = \mathcal{A}(\mathsf{crs}, \mathsf{hk}) \\ (x, w) \leftarrow \mathcal{E}(\mathsf{td}, \mathsf{v}, \pi) \\ \text{parse } \mathsf{td} = (\mathsf{seBARG.td}, \mathsf{SEH.td}) \end{array}\right] \leq \mathsf{negl}(\lambda). \tag{17}$$

We prove this by induction on the depth d of v output by $\mathcal{A}$.

Induction base: Fix any poly-size adversary that always outputs (v, π) such that $d(\mathsf{v}) = 1$. In this case, the equation above follows immediately from the somewhere argument of knowledge property of the underlying seBARG scheme together with the somewhere statistically binding w.r.t. opening property of the underlying SEH family.

Induction step: Suppose that Equation (17) holds for every poly-size $\mathcal{A}$ that outputs (v, π) such that $d(\mathsf{v}) < d$, and we prove that it holds for any poly-size $\mathcal{A}$ that outputs (v, π) such that $d(\mathsf{v}) \leq d$. Recall that $\mathcal{E}(\mathsf{td}, \mathsf{v}, \pi)$ first computes

$$\left(\mathsf{v}^{(d-1)}, \rho_{d-1}, \pi^{(d-1)}\right) = \mathsf{seBARG}.\mathcal{E}\left(\mathsf{td}_d, \left(Z^{(j)}\right)_{j\in[k]}, \pi\right)$$

where

$$Z^{(j)} = \left((\mathsf{crs}_\ell)_{\ell\in[d]}, (\mathsf{hk}_\ell)_{\ell\in[d+1]}, \mathsf{v}, j\right)$$

By the somewhere argument of knowledge property of the underlying seBARG scheme there exists a negligible function μ such that for every $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l} \mathcal{V}(\mathsf{crs}, \mathsf{hk}, \mathsf{v}, \pi) = 1 \\ \wedge \left(\mathsf{v}^{(d-1)}, \rho_{d-1}, \pi^{(d-1)}\right) \text{ is not a valid witness for } : \\ Z^{(i_d)} \in \mathcal{L}' \end{array} \begin{array}{c} (\mathsf{crs}, \mathsf{hk}, \mathsf{td}) \leftarrow \mathsf{Gen}(1^\lambda, n, k, (i_1, \ldots, i_\lambda)) \\ \text{parse } \mathsf{td} = (\mathsf{seBARG.td}, \mathsf{SEH.td}). \end{array}\right] \leq \mu(\lambda)$$

By the definition of $\mathcal{L}'$ this implies that for every $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l} \mathcal{V}(\mathsf{crs}, \mathsf{hk}, \mathsf{v}, \pi) = 1 \\ \wedge \Big(\mathsf{SEH.Verify}\left(\mathsf{hk}, \mathsf{v}, I_{i_d}, \mathsf{v}^{(d-1)}, \rho_{d-1}\right) = 0 \\ \vee\ \mathsf{seBARG}.\mathcal{V}\left(\mathsf{crs}_d, \left(Y^{(1)}, \ldots, Y^{(k)}\right), \pi^{(d-1)}\right) = 0\Big) \end{array} \begin{array}{c} : \\ \end{array} \begin{array}{c} (\mathsf{crs}, \mathsf{hk}, \mathsf{td}) \leftarrow \mathsf{Gen}(1^\lambda, n, k, (i_1, \ldots, i_\lambda)) \\ \text{parse } \mathsf{td} = (\mathsf{seBARG.td}, \mathsf{SEH.td}) \end{array}\right] \leq \mu(\lambda)$$

where

$$Y^{(j)} = \left((\mathsf{crs}_\ell)_{\ell\in[d-1]}, (\mathsf{hk}_\ell)_{\ell\in[d]}, \mathsf{v}^{(d-1)}, j\right).$$

This together with our induction hypothesis implies Equation (17), as desired. □

Remark 6.5. The above construction can be generalized to prove statements which involve multiple of the hash values being aggregated (but still only one of the proofs being aggregated). Suppose $\mathsf{AggProve}$ is given hash values $\mathsf{v}_1, \ldots, \mathsf{v}_{k'}$ of depth d and proofs $\pi_1, \ldots, \pi_k$ where each proof π_i is verified against hash values $(\mathsf{v}_j)_{j\in J_i}$ for a subset $J_i \subseteq [k']$ of size $\ell \geq 1$. $\mathsf{AggProve}$ produces a hash value $\mathsf{v} = \mathsf{SEH.Hash}\left(\mathsf{hk}_{d+1}, (\mathsf{v}_1, \ldots, \mathsf{v}_{k'})\right)$ and proof π with witnesses $\pi_i, (\mathsf{v}_j, \rho_j)_{j\in J_i}$, where ρ_j is an opening of v_j w.r.t. $(\mathsf{hk}_{d+1}, \mathsf{v})$.

Where in the original construction we have that the length of π was bounded by $n \cdot (d+1) \cdot \mathsf{poly}(\lambda) + m$ (see Equation (16)), here we have that

$$\begin{aligned} |\pi| &= |\pi_i| + \ell \cdot (n_d + \mathsf{poly}(\lambda)) \leq (\ell \cdot n \cdot d \cdot \mathsf{poly}(\lambda) + m) + \ell \cdot n + \mathsf{poly}(\lambda) \\ &\leq \ell \cdot n \cdot (d+1) \cdot \mathsf{poly}(\lambda) + m \end{aligned}$$

The bound on the length of v remains the same, as it is still only extractable on one hash value of depth d. Thus we preserve succinctness for $\ell = O(1)$.

Note that we still only have the somewhere argument of knowledge property for one instance of the NP language $\mathcal{L}$. We will use this generalization when constructing incrementally verifiable computation in Section 7.2.

7 Applications

7.1 Aggregate Signatures

Our first application is to show how multi-hop seBARGs can be used to build an unbounded (multi-hop, multi-signer) aggregate signature scheme.

7.1.1 Definition

The notion of aggregate signatures as introduced by Boneh, Gentry, Lynn, and Shacham [BGLS03] is a digital signature scheme that comes with two poly-time algorithms Aggregate and AggVerify, where Aggregate is used to aggregate an arbitrary polynomial number of message-signature pairs $\{(m_i, \sigma_i)\}_i$ generated using verification keys $\{\mathsf{vk}_i\}_i$, into a shorter aggregate signature $\widehat{\sigma}$, and AggVerify can be used to verify such aggregate signatures with respect to the sequence of messages $(m_1, \ldots, m_\ell)$ and the verification keys $(\mathsf{vk}_1, \ldots, \mathsf{vk}_\ell)$.

It is desirable that aggregated signatures have the property that they can be further aggregated; i.e., that aggregation can be performed in multiple hops or sequentially, and all the key-message-signature tuples need not be available at once. Indeed, many existing schemes in the literature give multi-hop aggregation by default, yet this notion was not formalized. In this work, we give a formal definition of a multi-hop aggregate signature scheme. First, we recall the syntax and desired properties from (single-hop) aggregate signatures as defined in the literature, and later describe the formal syntax for multi-hop signature aggregation.

Syntax. A (single-hop, multi-signer) aggregate signature scheme $\mathcal{S}$ for message space $\{\{0,1\}^\lambda\}_{\lambda\in\mathbb{N}}$ consists of the following polynomial time algorithms:

$\mathsf{CRS}(1^\lambda) \to \mathsf{crs}$. The CRS generation algorithm samples global parameters crs.

All the remaining algorithms take crs *as input, and for ease of notation we do not write it explicitly.*

$\mathsf{Setup}(1^\lambda) \to (\mathsf{vk}, \mathsf{sk})$. The setup algorithm, on input the security parameter λ, outputs a pair of signing and verification keys $(\mathsf{vk}, \mathsf{sk})$.

$\mathsf{Sign}(\mathsf{sk}, m) \to \sigma$. The signing algorithm takes as input a signing key sk and a message $m \in \{0,1\}^\lambda$, and computes a signature σ.

$\mathsf{Verify}(\mathsf{vk}, m, \sigma) \to 0/1$. The verification algorithm takes as input a verification key vk, a message $m \in \{0,1\}^\lambda$, and a signature σ. It outputs a bit to signal whether the signature is valid or not.

$\mathsf{Aggregate}(\{(\mathsf{vk}_i, m_i, \sigma_i)\}_i) \to \widehat{\sigma}/\bot$. The signature aggregation algorithm takes as input a sequence of tuples, each containing a verification key vk_i, a message m_i, a signature σ_i, and it outputs either an aggregated signature $\widehat{\sigma}$ or a special abort symbol $\bot$.

$\mathsf{AggVerify}(\{(\mathsf{vk}_i, m_i)\}_i, \widehat{\sigma}) \to 0/1$. The aggregated verification algorithm takes as input a sequence of tuples, each containing a verification key vk_i, a message m_i, and it outputs a bit to signal whether the aggregated signature $\widehat{\sigma}$ is valid or not.

Correctness and Compactness. An aggregate signature scheme is said to be correct and compact if for all $\lambda, \ell, N \in \mathbb{N}$, parameters $\mathsf{crs} \leftarrow \mathsf{CRS}(1^\lambda)$, verification-signing key pairs $(\mathsf{vk}_j, \mathsf{sk}_j) \leftarrow \mathsf{Setup}(1^\lambda)$ for $j \in [N]$, messages m_i for $i \in [\ell]$, every key mapping function $\pi : [\ell] \to [N]$,[17] and every signature $\sigma_i \leftarrow \mathsf{Sign}(\mathsf{sk}_{\pi(i)}, m_i)$ for $i \in [\ell]$, the following holds:

Correctness of signing. For all $i \in [\ell]$, $\mathsf{Verify}(\mathsf{vk}_{\pi(i)}, m_i, \sigma_i) = 1$.

Correctness of aggregation. If $\widehat{\sigma} = \mathsf{Aggregate}\left(\{\left(\mathsf{vk}_{\pi(i)}, m_i, \sigma_i\right)\}_i\right)$, then

$$\mathsf{AggVerify}\left(\{\left(\mathsf{vk}_{\pi(i)}, m_i\right)\}_i, \widehat{\sigma}\right) = 1.$$

Compactness of aggregation. $|\widehat{\sigma}| \leq \mathsf{poly}(\lambda, \log \ell)$. That is, the size of an aggregated signature is bounded by a fixed polynomial in λ, and only logarithmically depends on the number of aggregations ℓ. (Since ℓ is always less than 2^λ, thus one could also simply write $\mathsf{poly}(\lambda)$ instead.)

Security. For security, there is plain and aggregated unforgeability.

Definition 7.1 (Unforgeability)**.** *A signature scheme* $(\mathsf{Setup}, \mathsf{Sign}, \mathsf{Verify})$ *is said to be a secure signature scheme if for every admissible PPT attacker* $\mathcal{A}$*, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for all* $\lambda \in \mathbb{N}$*, the following holds*

$$\Pr\left[\mathsf{Verify}(\mathsf{vk}, m^*, \sigma^*) = 1 : \begin{array}{l} \mathsf{crs} \leftarrow \mathsf{CRS}(1^\lambda), (\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda) \\ (m^*, \sigma^*) \leftarrow \mathcal{A}^{\mathsf{Sign}(\mathsf{sk}, \cdot)}(1^\lambda, \mathsf{vk}) \end{array}\right] \leq \mathsf{negl}(\lambda),$$

and $\mathcal{A}$ *is admissible as long as it did not query* m^* *to the* Sign *oracle.*

Definition 7.2 (Aggregated Unforgeability)**.** *An aggregate signature scheme* $(\mathsf{CRS}, \mathsf{Setup}, \mathsf{Sign}, \mathsf{Verify}, \mathsf{Aggregate}, \mathsf{AggVerify})$ *is said to be a secure aggregate signature scheme if for every admissible PPT attacker* $\mathcal{A}$*, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for all* $\lambda \in \mathbb{N}$*, the following holds*

$$\Pr\left[\mathsf{AggVerify}\left(\{(\mathsf{vk}_i^*, m_i^*)\}_{i \in [\ell]}, \widehat{\sigma}^*\right) = 1 : \begin{array}{l} \mathsf{crs} \leftarrow \mathsf{CRS}(1^\lambda), (\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda) \\ \left(\{(\mathsf{vk}_i^*, m_i^*)\}_{i \in [\ell]}, \widehat{\sigma}^*\right) \leftarrow \mathcal{A}^{\mathsf{Sign}(\mathsf{sk}, \cdot)}(1^\lambda, \mathsf{vk}) \end{array}\right] \leq \mathsf{negl}(\lambda),$$

where $\mathcal{A}$ *is admissible if there exists* $i \in [\ell]$ *such that* $\mathsf{vk}_i^* = \mathsf{vk}$ *and* m_i^* *was not queried by* $\mathcal{A}$ *to the* $\mathsf{Sign}(\mathsf{sk}, \cdot)$ *oracle.*

Extended syntax for multi-hop aggregation. In the case of multi-hop aggregation, the syntax of aggregation and aggregated verification is strengthened as follows.

$\mathsf{Aggregate}\left(\{(T_i^{\mathsf{vk}, m}, \sigma_i)\}_i\right) \to \widehat{\sigma}/\perp$. The signature aggregation algorithm takes as input a sequence of (verification-)key-message trees along with a (possibly aggregated) signature σ_i. And, it outputs either an aggregated signature $\widehat{\sigma}$ or a special abort symbol $\perp$.

[17] The goal behind the key mapping function is to allow aggregation of multiple signatures that come from a single-signer. That is, an aggregator might want to aggregate signatures from multiple signers, where each signer might be contributing more than one signature potentially.

Remark 7.1. *Similar to the case of multi-hop BARGs, we define a (verification-)key-message tree $T^{\mathsf{vk},m}$ of key-message pairs as a tree of varying arity each of whose leaf nodes is associated with a single key-message pair, and each intermediate node corresponds to an aggregated signature that certifies the validity for all key-message pairs associated with its children. As before, the signatures at the intermediate nodes are not stored in the tree. (See definition 6.1.)*

$\mathsf{AggVerify}\left(T^{\mathsf{vk},m}, \widehat{\sigma}\right) \to 0/1$. The aggregated verification algorithm takes as input a key-message tree $T^{\mathsf{vk},m}$ of key-message pairs, and an aggregate signature $\widehat{\sigma}$. It outputs a bit to signal whether the aggregated signature $\widehat{\sigma}$ is valid or not.

Correctness and Compactness of Multi-Hop Aggregation. The correctness and compactness of an aggregate signature scheme can be naturally extended to the multi-hop setting. In addition to the properties discussed previously, we require the following to hold:

Correctness of multi-hop aggregation. For any sequence of accepting aggregate signatures and key-message trees σ_i and $T_i^{\mathsf{vk},m}$ for $i \in [\ell]$, we have that the aggregated signature $\widehat{\sigma}$ will also be an accepting signature for the key-message tree $T^{\mathsf{vk},m} = (T_1^{\mathsf{vk},m}, \ldots, T_\ell^{\mathsf{vk},m})$. More formally, we have that for any $\lambda \in \mathbb{N}$, $\ell \in [2^\lambda]$, and any σ_i, and key-message tree $T_i^{\mathsf{vk},m}$ such that $\mathsf{AggVerify}(T_i^{\mathsf{vk},m}, \sigma_i) = 1$, we have that

$$\Pr\left[\mathsf{AggVerify}(T^{\mathsf{vk},m}, \widehat{\sigma}) = 1 \ : \ T^{\mathsf{vk},m} = (T_1^{\mathsf{vk},m}, \ldots, T_\ell^{\mathsf{vk},m}),\ \widehat{\sigma} \leftarrow \mathsf{Aggregate}(\{(T_i^{\mathsf{vk},m}, \sigma_i)\}_i)\right] = 1.$$

Compactness of multi-hop aggregation. $|\widehat{\sigma}| = \max_i |\sigma_i| + \mathsf{poly}(\lambda, \log \ell)$. That is, the size of an aggregated signature is bounded by a fixed polynomial in λ, and only logarithmically depends on the number of aggregations ℓ. (Since ℓ is always less than 2^λ, thus one could also simply write $\mathsf{poly}(\lambda)$ instead.)

Security of Multi-Hop Aggregation. Multi-hop aggregated unforgeability is defined as follows.

Definition 7.3 (Multi-Hop Aggregated Unforgeability)**.** *An multi-hop aggregate signature scheme* $(\mathsf{CRS}, \mathsf{Setup}, \mathsf{Sign}, \mathsf{Verify}, \mathsf{Aggregate}, \mathsf{AggVerify})$ *is said to be a secure multi-hop aggregate signature scheme if for every admissible PPT attacker $\mathcal{A}$, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for all $\lambda \in \mathbb{N}$, the following holds*

$$\Pr\left[\mathsf{AggVerify}\left(T^*, \widehat{\sigma}^*\right) = 1 : \begin{array}{l} \mathsf{crs} \leftarrow \mathsf{CRS}(1^\lambda), (\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda) \\ (T^*, \widehat{\sigma}^*) \leftarrow \mathcal{A}^{\mathsf{Sign}(\mathsf{sk}, \cdot)}(1^\lambda, \mathsf{vk}) \end{array}\right] \leq \mathsf{negl}(\lambda),$$

where $\mathcal{A}$ is admissible if there exists a leaf node (vk_i^*, m_i^*) in T^* such that $\mathsf{vk}_i^* = \mathsf{vk}$ and m_i^* was not queried by $\mathcal{A}$ to the $\mathsf{Sign}(\mathsf{sk}, \cdot)$ *oracle.*

7.1.2 Construction and Analysis

We construct a multi-hop aggregate signature scheme from any multi-hop seBARG scheme $\mathsf{seBARG} = (\mathsf{seBARG.Gen}, \mathsf{seBARG}.\mathcal{P}, \mathsf{seBARG.AggProve}, \mathsf{seBARG}.\mathcal{V})$ and any signature scheme $\mathsf{S} = (\mathsf{S.Setup}, \mathsf{S.Sign}, \mathsf{S.Verify})$. Below we describe our multi-hop aggregate signature scheme Agg.

$\mathsf{CRS}(1^\lambda) \to \mathsf{crs}$. The CRS generator samples the CRS for the multi-hop seBARG scheme for language $\mathcal{L}_{\mathsf{S}}$ (eq. (18)) as

$$(\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{seBARG.Gen}(1^\lambda, d = \lambda, I = (1, \ldots, 1)).$$

That is, it sets the maximum number of hops to be λ, and sets the extraction indices to be all ones by default. (Refer to remark 7.2 for details on λ hops being sufficient for unbounded signature aggregation.)

$$\mathcal{L}_{\mathsf{S}} = \{x = (m, \mathsf{vk}) \ : \ \exists \sigma \text{ s.t. } \mathsf{S.Verify}(\mathsf{vk}, m, \sigma) = 1\}. \tag{18}$$

It outputs crs as the CRS.

(As noted previously, for ease of notation, we do not write it explicitly, but all the algorithms take crs as an additional input.)

$\mathsf{Setup}(1^\lambda) \to (\mathsf{vk}, \mathsf{sk})$. This is the same as the regular signature setup algorithm. That is, it outputs the verification-signing key pair as $(\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{S.Setup}(1^\lambda)$.

$\mathsf{Sign}(\mathsf{sk}, m) \to \sigma$. This is the same as the regular signature signing algorithm. That is, it outputs the signature as $\sigma \leftarrow \mathsf{S.Sign}(\mathsf{sk}, m)$.

$\mathsf{Verify}(\mathsf{vk}, m, \sigma) \to 0/1$. This is the same as the regular signature verification algorithm. That is, it checks whether $\mathsf{S.Verify}(\mathsf{vk}, m, \sigma) = 0$ or 1, and outputs whatever it outputs.

$\mathsf{Aggregate}\left(\{(T_i^{\mathsf{vk},m}, \sigma_i)\}_{i \in [\ell]}\right) \to \widehat{\sigma}/\perp$. The signature aggregator simply runs the proof combiner algorithm $\mathsf{seBARG.AggProve}$ on the CRS, and sequence of key-message tree and signature pairs. Concretely, it computes the aggregated signature as

$$\widehat{\sigma} \leftarrow \mathsf{seBARG.AggProve}(\mathsf{crs}, T_1^{\mathsf{vk},m}, \ldots, T_\ell^{\mathsf{vk},m}, \sigma_1, \ldots, \sigma_\ell).$$

Here we abuse notation, and if all the input signatures σ_i are plain (unaggregated) signatures, then we run the prover algorithm $\mathsf{seBARG}.\mathcal{P}$. However, for ease of exposition, we write it to be the proof combiner above.

$\mathsf{AggVerify}\left(T^{\mathsf{vk},m}, \widehat{\sigma}\right) \to 0/1$. This is the same as the seBARG verification algorithm. That is, it checks whether $\mathsf{seBARG}.\mathcal{V}(\mathsf{crs}, T^{\mathsf{vk},m}, \widehat{\sigma}) = 0$ or 1, and outputs whatever it outputs.

Correctness, Compactness, and Security. Below we prove the following.

Theorem 7.4. *If the signature scheme* S *is unforgeable as per definition 7.1, and batch argument scheme* seBARG *is rate-1 multi-hop scheme (definition 6.3), then the above scheme* Agg *is a multi-hop aggregate signature scheme satisfying plain and aggregated unforgeability as per definitions 7.1 and 7.3.*

First, we argue correctness and compactness of our multi-hop aggregate signatures, and later talk about its security.

Correctness, Compactness and Multi-Hop Aggregation. The correctness of signing follows directly from the correctness of the signature scheme S, as the $\mathsf{Setup}, \mathsf{Sign}, \mathsf{Verify}$ algorithms in our aggregate signature scheme Agg are same as $\mathsf{S.Setup}, \mathsf{S.Sign}, \mathsf{S.Verify}$. Next, the correctness

of aggregation also follows from that the correctness of the multi-hop BARGs and the signature scheme.

For compactness, first note that each aggregate signature $\widehat{\sigma}$ is simply a batched proof. Thus, in the base case, when the aggregation algorithm takes as input plain (unaggregated) signatures, then the size of the resulting aggregate signature $\widehat{\sigma}$ will simply be the size of a plain signature (which is a fixed polynomial in the security parameter) plus another polynomial term in the security parameter (since the length of each instance is also a fixed polynomial as it only contains a single key-message pair). Thus, our multi-hop aggregate signature scheme is fully compact. Below we briefly talk about how a fixed number of λ hops is sufficient for unbounded aggregation.

Remark 7.2 (Unbounded Aggregation using Tree-Based Aggregation)**.** Note that in the above description we only set the number of hops supported by the BARG scheme to be λ. However, this does not limit the number of times an aggregator can aggregate signatures. This is because we consider that each aggregator will combine proofs in a more efficient tree-based fashion where the proof combiner algorithm is only run when the input aggregate signatures are of equal length.

More concretely, in any application, we consider that each aggregate signature $\widehat{\sigma}$ will be stored as a length-λ sequence of batched proofs. Here the i-th element of the sequence is interpreted as an aggregate signature for 2^i key-message pairs. Now, whenever two aggregate signatures will be combined, then only the proofs corresponding to the same level i will be combined together. Since this is purely a data structure management task, we decided to simply our above exposition, and avoid maintaining this structure as part of the actual scheme.

Next, we argue the unforgeability of our scheme.

Lemma 7.5. *If the signature scheme* S *satisfies unforgeability, the aggregate signature scheme* Agg *satisfies plain unforgeability. In addition, if the multi-hop batch argument scheme* seBARG *satisfies the index-hiding and somewhere argument of knowledge properties, then the scheme* Agg *is aggregated-unforgeable.*

Proof. Plain Unforgeability follows directly from the unforgeability of the signature scheme S.

Aggregated unforgeability relies on the index hiding and somewhere argument of knowledge properties of seBARG, and unforgeability of the signature scheme S. Let $\mathcal{A}$ be a PPT attacker that breaks aggregated unforgeability property. That is, $\mathcal{A}$ finds a valid forgery $(T^*, \widehat{\sigma}^*)$ with non-negligible probability $\epsilon = \epsilon(\lambda)$.

Here and through the rest of the proof, we consider that the the arity of every node in the tree T^* is *exactly two*, except the parents of the leaf nodes. That is, the all except the last layer of tree is a binary tree. Note that this does not conflict with our scheme, but is in line with remark 7.2. That is, the aggregator only runs the proof combiner algorithm on a batch size of two, but the initial batch size could still be arbitrary. Although we could consider that the arity of all nodes is exactly two and it would not affect the compactness, we allow a larger arity for the parents of leaf nodes for simplicity.

With this consideration, note that by definition of a valid forgery, there must exist a set of indices $(i_1^*, \ldots, i_\lambda^*) \in [2^\lambda]^\lambda$ such that the key-message pair in the tree T^* corresponding to the path $I^* = (i_1^*, \ldots, i_\lambda^*)$ is (vk, m^*) where vk is the challenge verification key, and m^* was not queried by $\mathcal{A}$ to the challenger. (Also, all but the first index i_1^* are just a single bit due to the condition that arity of T^* is exactly two at all but the last layer.) While there might exist multiple such paths, we use I^* to denote the lexicographically first such path with this property. We start by defining simple hybrid experiments to complete the proof.

Experiment 0. This is the aggregate unforgeability security game, and we say the output of the experiment is 1 if and only if $\mathcal{A}$ wins.

Experiment 1. Let N be the maximum running time of $\mathcal{A}$and therefore the maximum number of leaves in T^*. In this experiment, the challenger samples λ random indices $i_1, \ldots, i_\lambda$ at the beginning of the game as $i_1 \leftarrow [N]$ and $i_j \leftarrow \{1, 2\}$ for $j > 1$. And, it plays the rest of aggregate unforgeability game with $\mathcal{A}$ as is. In the end, we say the output of the experiment is 1 if and only if $\mathcal{A}$ wins *and* $I^* = (i_1, \ldots, i_{j^*}, 1, \ldots, 1)$ for some j^*. (That is, I^* matches the guessed path. Since $\mathcal{A}$ is a PPT machine, thus the tree T^* is of polynomial size.)

Experiment 2. In this experiment, the challenger samples λ random indices $i_1, \ldots, i_\lambda$ as before, and samples the CRS as follows:

$$(\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{seBARG.Gen}(1^\lambda, d = \lambda, I = (i_1, \ldots, i_\lambda)).$$

It plays the rest of aggregate unforgeability game with $\mathcal{A}$ as is. In the end, we say the output of the experiment is 1 if and only if $\mathcal{A}$ wins and $I^* = (i_1, \ldots, i_{j^*}, 1, \ldots, 1)$ for some j^*.

Experiment 3. This is same as previous experiment, except if $\mathcal{A}$ wins and I^* matches the guessed indices $(i_1, \ldots, i_{j^*})$, then the challenger runs $\mathcal{E}$, the PPT extractor for the seBARG scheme, as follows:

$$\sigma^* = \mathcal{E}(\mathsf{td}, T^*, \widehat{\sigma}^*).$$

And, we say the output of the experiment is 1 if and only if (m^*, σ^*) is a valid forgery w.r.t. challenge key vk.

Let $\text{EXPT}_i^{\mathcal{A}}(1^\lambda)$ denote the output of the experiment i. Next, we prove the following sequence of claims regarding the above experiments.

Claim 7.5.1. *For every adversary $\mathcal{A}$, for every $\lambda \in \mathbb{N}$, we have that*

$$\Pr\left[\text{EXPT}_1^{\mathcal{A}}(1^\lambda)\right] \geq \frac{\Pr\left[\text{EXPT}_0^{\mathcal{A}}(1^\lambda)\right]}{N \cdot 2^{j^*-1}}.$$

Proof. This is an information theoretic argument, and follows directly from the fact that the challenger samples λ random indices as its guesses, and the first guess is correct with probability at least $1/N$, while remaining $(j^* - 1)$ guesses are correct with probability at least $1/2$. Basically, with probability at least $\frac{1}{N \cdot 2^{j^* 01}}$, its guess will be correct, and the output of the experiment will be kept as 1 when $\mathcal{A}$ wins. □

Claim 7.5.2. *If the batch argument scheme* seBARG *satisfies index hiding, then for every PPT $\mathcal{A}$ playing the above aggregated unforgeability game, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for all $\lambda \in \mathbb{N}$, we have that*

$$\Pr\left[\text{EXPT}_1^{\mathcal{A}}(1^\lambda)\right] - \Pr\left[\text{EXPT}_2^{\mathcal{A}}(1^\lambda)\right] \leq \mathsf{negl}(\lambda).$$

Proof. This follows directly from the index hiding property of the seBARG. □

Claim 7.5.3. *If the batch argument scheme* seBARG *satisfies somewhere argument of knowledge property, then for every PPT $\mathcal{A}$ playing the above aggregated unforgeability game, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for all $\lambda \in \mathbb{N}$, we have that*

$$\Pr\left[\text{EXPT}_2^{\mathcal{A}}(1^\lambda)\right] - \Pr\left[\text{EXPT}_3^{\mathcal{A}}(1^\lambda)\right] \leq \mathsf{negl}(\lambda).$$

Proof. This follows directly from the somewhere argument of knowledge property of the seBARG. □

Claim 7.5.4. *If the signature scheme* S *satisfies unforgeability, then for every* $\mathcal{A}$ *playing the above aggregated unforgeability game, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for all* $\lambda \in \mathbb{N}$*, we have that*

$$\Pr\left[\text{EXPT}_3^{\mathcal{A}}(1^\lambda)\right] \leq \mathsf{negl}(\lambda).$$

Proof. Suppose $\Pr\left[\text{EXPT}_3^{\mathcal{A}}(1^\lambda)\right] = \epsilon$ for some non-negligible probability $\epsilon = \epsilon(\lambda)$. We now describe a reduction algorithm $\mathcal{B}$ that uses $\mathcal{A}$ to break the unforgeability property of the signature scheme S with probability at least ϵ.

The challenger corresponding to scheme S samples a verification-signing key pair, and sends vk to the reduction algorithm $\mathcal{B}$. $\mathcal{B}$ then samples λ random indices $i_1, \ldots, i_\lambda$ as described in the experiments above, and samples the CRS components as follows:

$$(\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{seBARG.Gen}(1^\lambda, d = \lambda, I = (i_1, \ldots, i_\lambda)).$$

It then sets the CRS as crs, and sends crs, vk to the adversary. $\mathcal{B}$ then answers $\mathcal{A}$'s signing queries by forwarding them to the signature scheme challenger, and relaying the challenger's response back to $\mathcal{A}$. Finally, $\mathcal{A}$ outputs a forgery $(T^*, \widehat{\sigma}^*)$, and sends it to $\mathcal{B}$.

The reduction algorithm $\mathcal{B}$ computes the forgery as $\sigma^* = \mathcal{E}(\mathsf{td}, T^*, \widehat{\sigma}^*)$, and submits m^* and σ^* as its forgery, where (vk, m^*) is the key-message pair corresponding to path I^* in T^*.

Note that $\mathcal{B}$ wins the unforgeability game with the challenger for signature scheme S with probability ϵ. This is because, by definition of experiment 3, the experiment outputs 1 if and only if (m^*, σ^*) is a valid forgery w.r.t. challenge key vk. Thus, the claim follows. □

Combining all the above claims, we obtain that $\Pr\left[\text{EXPT}_0^{\mathcal{A}}(1^\lambda)\right] \leq \mathsf{negl}(\lambda)$ as $N \cdot 2^{j^*-1}$ is a polynomial in the security parameter as the adversary outputs at most $N \cdot 2^{j^*-1}$ messages along with its forgery. Thus, proof of aggregate unforgeability follows.

This concludes the security proof.

□

7.2 Incrementally Verifiable Computation

In this section, we construct an incrementally verifiable computation (IVC) scheme for deterministic computations with proof size $\mathsf{poly}(\lambda, \log T)$ using a rate-1 hashed multi-hop BARG scheme (see Section 6) associated with a rate-1 fully-local somewhere extractable hash function. Below we recall the definition of IVC.

7.2.1 Definition

Language $\mathcal{L}_{\mathcal{M}}$. For any deterministic Turing machine $\mathcal{M}$ with run-time $T = T(n)$ and configuration size $S = S(n)$, any input $z \in \{0,1\}^n$, and any $t \in [T(n)]$, we denote by $\mathcal{M}(z; 1^t) \in \{0,1\}^{S(n)}$ the configuration of $\mathcal{M}$ when executed on input z after t steps. In this paper, we consider the following language.

$$\mathcal{L}_{\mathcal{M}} = \left\{(z, t, c) \;:\; t \in [T(|z|)] \;\wedge\; \mathcal{M}(z; 1^t) = c \in \{0,1\}^{S(|z|)}\right\} \tag{19}$$

Notation 7.6. *We use* $\mathsf{nxt\text{-}cnfg}_{\mathcal{M}}(\cdot)$ *as the shorthand denoting the computation of the next configuration from any given* valid *intermediate configuration of the Turing machine* $\mathcal{M}$ *and an input* z*. That is,* $c_0 = \mathsf{nxt\text{-}cnfg}_{\mathcal{M}}(\emptyset, z)$ *is the starting configuration, and* $c_i = \mathsf{nxt\text{-}cnfg}_{\mathcal{M}}(c_{i-1}, z)$ *for* $i \geq 1$*. It follows by induction that* $c_i = \mathcal{M}(z; 1^i)$ *for* $i \geq 1$*. We refer to* c_0 *as the initial configuration of the machine, and* z *represents the fixed input tape of the Turing machine.*

IVC Syntax and Definition. An incrementally verifiable computation (IVC) scheme for a Turing machine $\mathcal{M}$ consists of the following (probabilistic) polynomial time algorithms:

$\mathsf{Gen}(1^\lambda, n, 1^S) \to \mathsf{crs}$**.** The setup algorithm is a randomized algorithm that takes as input a security parameter 1^λ, the input length n, the maximum configuration size $S = S(n)$ in unary, and outputs a common reference string crs.

$\mathcal{P}(\mathsf{crs}, z, 1^t) \to (c_t, \pi_t)$**.** The prover algorithm takes as input a common reference string crs, an input $z \in \{0,1\}^n$, and the number of time steps $t \in \mathbb{N}$ in unary. It outputs the configuration $c_t = \mathcal{M}(z; 1^t)$ and a proof π_t.

$\mathsf{Update}(\mathsf{crs}, z, c_{t-1}, \pi_{t-1}) \to (c_t, \pi_t)$**.** The update algorithm takes as input a CRS crs, an input $z \in \{0,1\}^n$, an intermediate configuration c_{t-1}, and a proof π_{t-1}. It outputs the next configuration c_t and an updated proof π_t.

$\mathcal{V}(\mathsf{crs}, x = (z, t, c_t), \pi) \to 0/1$**.** The verifier algorithm takes as input a CRS crs, an instance $x = (z, t, c_t)$, and a proof π. It outputs a bit to signal whether the proof is valid or not.

Definition 7.7. *An incremental verifiable computation scheme* $(\mathsf{Gen}, \mathcal{P}, \mathsf{Update}, \mathcal{V})$ *for* $\mathcal{M}$ *is required to satisfy the following properties:*

Efficiency. *The running time of the setup algorithm is at most* $\mathsf{poly}(\lambda, S, \log n, \log T(n))$*, the run-time of verifier and update algorithm is at most* $\mathsf{poly}(\lambda, S, n, \log T(n))$*, and the prover runs in time at most* $t \cdot \mathsf{poly}(\lambda, S, n, \log T(n))$*.*

Completeness. *For every* $\lambda, n \in \mathbb{N}$ *any* $z \in \{0,1\}^n$*, and any time step* $t \in [T(n)]$*,*

$$\Pr\left[\mathcal{V}(\mathsf{crs}, (z,t,c), \pi) = 1 \quad : \quad \begin{array}{l} \mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda, n, 1^{S(n)}) \\ (c, \pi) \leftarrow \mathcal{P}(\mathsf{crs}, z, 1^t) \end{array} \right] = 1.$$

Soundness. *For any poly-size adversary* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ *there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for every* $\lambda \in \mathbb{N}$*,*

$$\Pr\left[\begin{array}{l} z \in \{0,1\}^n \wedge \ x \notin \mathcal{L}_{\mathcal{M}} \\ \wedge\ \mathcal{V}(\mathsf{crs}, x, \pi) = 1 \end{array} \quad : \quad \begin{array}{l} 1^n \leftarrow \mathcal{A}_1(1^\lambda), \\ \mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda, n, 1^{S(n)}), \\ (x = (z, 1^t, c_t), \pi) \leftarrow \mathcal{A}_2(\mathsf{crs}) \end{array} \right] \leq \mathsf{negl}(\lambda).$$

An *incremental verifiable computation* (IVC) scheme for $\mathcal{M}$ is simply an updatable SNARG for $\mathcal{L}_{\mathcal{M}}$.

7.2.2 Construction and Analysis

Let

$$(\mathsf{multi\text{-}BARG.Gen}, \mathsf{multi\text{-}BARG}.\mathcal{P}, \mathsf{multi\text{-}BARG.AggProve}, \mathsf{multi\text{-}BARG}.\mathcal{V})$$

be a hashed multi-BARG scheme associated with the rate-1 SEH hash family

$$(\mathsf{SEH.Gen}, \mathsf{SEH.Hash}, \mathsf{SEH.Open}, \mathsf{SEH.Verify}, \mathsf{SEH.Extract})$$

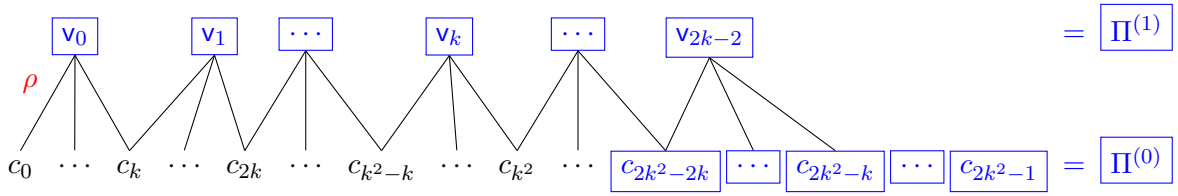
corresponding to the NP language

$$\mathcal{L}_{\mathsf{nxt\text{-}cnfg},\mathcal{M}} = \left\{ (z, c, c') \mid c' = \mathsf{nxt\text{-}cnfg}_{\mathcal{M}}(c, z) \right\}.$$

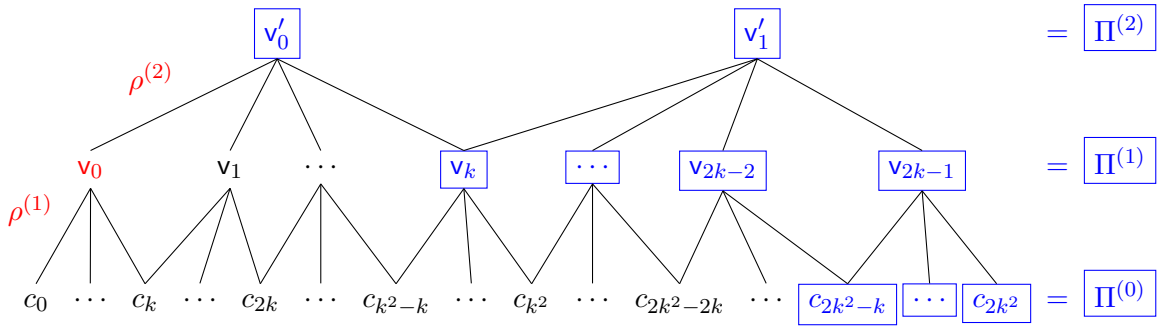
Notation. We specify our multi-BARG interface using the generality of Remark 6.5. Given instances $x_i = (z, c_{i-1}, c_i)$ for $i \in [2k]$, $\mathsf{multi\text{-}BARG}.\mathcal{P}$ outputs a segmented hash value $(\mathsf{v}, \mathsf{v}')$ of depth 1 where $\mathsf{v} = \mathsf{SEH.Hash}(\mathsf{hk}_1, z, c_0, c_1, \ldots, c_k)$ and $\mathsf{v}' = \mathsf{SEH.Hash}(\mathsf{hk}_1, z, c_k, c_{k+1}, \ldots, c_{2k})$ (and a corresponding proof). That is, although each configuration c_i is included in two instances, it is only included once in the string being hashed (except c_k). v and v' are each statistically binding on the input z and two configurations. Similarly, given hash values $(\mathsf{v}_{i-1}, \mathsf{v}_i)$ for $i \in [2k]$ of depth d (and corresponding proofs), $\mathsf{multi\text{-}BARG.AggProve}$ outputs a segmented hash value $(\mathsf{v}, \mathsf{v}')$ of depth $d+1$ where $\mathsf{v} = \mathsf{SEH.Hash}(\mathsf{hk}_{d+1}, \mathsf{v}_0, \mathsf{v}_1, \ldots, \mathsf{v}_k)$ and $\mathsf{v}' = \mathsf{SEH.Hash}(\mathsf{hk}_{d+1}, \mathsf{v}_k, \mathsf{v}_{k+1}, \ldots, \mathsf{v}_{2k})$ (and a corresponding proof). That is, although each hash value v_i is included in two instances, it is only included once in the string being hashed (except v_k). v and v' are each statistically binding on one hash value of depth d.

Overview. We aggregate in a tree-like structure, and only aggregate configurations (leaves of the tree) or hash values (non-leaf nodes of the tree) in batches of $2k$. We store all the unaggregated configurations/hash values at each level of the tree while waiting to accumulate enough configurations/hash values to aggregate. In our construction, a proof $\pi = \left(\rho, \Pi^{(0)}, \Pi^{(1)}, \ldots, \Pi^{(\lambda)}\right)$ contains an opening ρ, a list $\Pi^{(0)}$ of configurations, and, for every $d \in \lambda$, a list $\Pi^{(d)}$ of hash values of depth d (and corresponding proofs). Initially, ρ and the lists are empty. As we keep updating π for more timesteps, whenever a list reaches length $2k$, we aggregate, add the aggregated hash value/proof to the list one level higher, and then erase the first half of the list. The second half of the list will be aggregated again once the list grows (so we can ensure that we're always aggregating consecutive batches of configurations). We also maintain ρ so that it always opens the "leftmost" hash value, i.e., the first hash value stored in the highest nonempty level, to the initial configuration c_0.

For example, after updating the proof π for timestep $2k^2 - 1$, the hash tree looks like this (only the blue boxes are stored in π):



Note that $\Pi^{(1)}$ also contains proofs $\{\pi_i\}_{i=1}^{2k-1}$ for instances $\{(\mathsf{v}_{i-1}, \mathsf{v}_i)\}_{i=1}^{2k-1}$, which are omitted from the above figure for simplicity. After updating the proof π for the next timestep $2k^2$, the hash tree looks like this (again, only the blue boxes are stored in π):



Note that $\Pi^{(1)}$ also contains proofs $\{\pi_i\}_{i=k+1}^{2k}$ for instances $\{(\mathsf{v}_{i-1}, \mathsf{v}_i)\}_{i=k+1}^{2k}$, and $\Pi^{(2)}$ also contains a proof π' for instance $(\mathsf{v}'_0, \mathsf{v}'_1)$, which are omitted from the above figure for simplicity.

We are now ready to define our IVC scheme.

$\mathsf{Gen}(1^\lambda, n, 1^S) \to \mathsf{crs}$. This poly-time algorithm does the following:

1. Set $i_1, \ldots, i_\lambda \in [2k]$ arbitrarily. For example, set $i_1 = \cdots = i_\lambda = 1$.
2. Sample $(\mathsf{multi\text{-}BARG.crs}, \mathsf{hk}, \mathsf{td}) \leftarrow \mathsf{multi\text{-}BARG.Gen}(1^\lambda, n + 2S, 2k, (i_1, \ldots, i_\lambda))$.
3. Output $\mathsf{crs} = (\mathsf{multi\text{-}BARG.crs}, \mathsf{hk})$.

$\mathcal{P}(\mathsf{crs}, z, 1^t) \to (c_t, \pi_t)$. This poly-time algorithm does the following:

1. Let $c_0 = \mathsf{nxt\text{-}cnfg}_{\mathcal{M}}(\emptyset, z)$.
2. Let $\pi_0 = \big(\rho = \emptyset, \Pi^{(0)} = (c_0), \Pi^{(1)} = \emptyset, \ldots, \Pi^{(\lambda)} = \emptyset\big)$.
3. For $i \in [t]$, compute $(c_i, \pi_i) = \mathsf{Update}(\mathsf{crs}, z, c_{i-1}, \pi_{i-1})$.
4. Output (c_t, π_t).

$\mathsf{Update}(\mathsf{crs}, z, c_{t-1}, \pi_{t-1}) \to (c_t, \pi_t)$. This poly-time algorithm does the following:

1. Parse $\mathsf{crs} = (\mathsf{multi\text{-}BARG.crs}, \mathsf{hk} = (\mathsf{hk}_1, \ldots, \mathsf{hk}_\lambda))$.
2. Compute the next configuration $c_t = \mathsf{nxt\text{-}cnfg}_{\mathcal{M}}(c_{t-1}, z)$.
3. Parse $\pi_{t-1} = \big(\rho, \Pi^{(0)}, \ldots, \Pi^{(\lambda)}\big)$.
4. Append c_t to $\Pi^{(0)}$.
5. If $\Pi^{(0)}$ is not full, i.e., contains $< 2k + 1$ configurations, go to Item 11.
6. Parse the list $\Pi^{(0)} = (c_{t-2k}, \ldots, c_{t-1}, c_t)$.
7. Compute $(\mathsf{v}_*, \mathsf{v}'_*, \pi_*) = \mathsf{multi\text{-}BARG}.\mathcal{P}(\mathsf{crs}, (z, c_{t-2k+i-1}, c_{t-2k+i})_{i\in[2k]})$.
8. Compute $\rho^{(1)} = \mathsf{SEH.Open}(\mathsf{hk}_1, (z, c_{t-2k}, \ldots, c_{t-k}), [n + S])$. Note that the set $[n + S]$ corresponds to the coordinates of (z, c_{t-2k}).
9. Erase the first half of $\Pi^{(0)}$, i.e., let $\Pi^{(0)} = (c_{t-k}, \ldots, c_{t-1}, c_t)$.
10. For $d = 1, \ldots, \lambda$:

- If $\Pi^{(d)}$ is empty, i.e., v_* is now the leftmost hash value, then let $\Pi^{(d)} = (\mathsf{v}_*)$ and append $(\rho^{(d)}, \mathsf{v}_*)$ to ρ.
- Append v'_* and π_* to $\Pi^{(d)}$.
- If $\Pi^{(d)}$ is not full, i.e., contains $< 2k+1$ hash values of depth d, go to Item 11.
- Parse the list $\Pi^{(d)} = (\mathsf{v}_0, \mathsf{v}_1, \ldots, \mathsf{v}_{2k}, \pi_1, \ldots, \pi_{2k})$. Recall that for $i \in [2k]$, π_i is a proof for instance $(\mathsf{v}_{i-1}, \mathsf{v}_i)$.
- Compute $((\mathsf{v}_*, \mathsf{v}'_*, \pi_*)) = \mathsf{multi\text{-}BARG.AggProve}(\mathsf{crs}, (\mathsf{v}_{i-1}, \mathsf{v}_i)_{i\in[2k]}, (\pi_i)_{i\in[2k]})$.
- Compute $\rho^{(d+1)} = \mathsf{SEH.Open}(\mathsf{hk}_{d+1}, (\mathsf{v}_0, \ldots, \mathsf{v}_k), [|\mathsf{v}_0|])$.
- Erase the first half of $\Pi^{(d)}$, i.e., let $\Pi^{(d)} = (\mathsf{v}_k, \mathsf{v}_{k+1}, \ldots, \mathsf{v}_{2k}, \pi_{k+1}, \ldots, \pi_{2k})$

11. Output $\left(c_t, \pi_t = \left(\rho, \Pi^{(0)}, \Pi^{(1)}, \ldots, \Pi^{(\lambda)}\right)\right)$.

$\mathcal{V}(\mathsf{crs}, x = (z, t, c), \pi) \to 0/1$. This poly-time algorithm does the following:

1. Parse $\mathsf{crs} = (\mathsf{multi\text{-}BARG.crs}, \mathsf{hk})$.
2. Parse $\pi = \left(\rho, \Pi^{(0)}, \ldots, \Pi^{(\lambda)}\right)$.
3. Parse $\Pi^{(0)} = (c_{t-\ell}, \ldots, c_{t-1}, c_t)$ for some $\ell \geq 1$.
4. If $c_t \neq c$, output 0.
5. For $i \in [\ell]$, if $c_{t-i+1} \neq \mathsf{nxt\text{-}cnfg}_{\mathcal{M}}(c_{t-i}, z)$, output 0.
6. For $d = 1, \ldots, \lambda$:
   - Parse $\Pi^{(d)} = (\mathsf{v}_0, \mathsf{v}_1, \ldots, \mathsf{v}_\ell, \pi_1, \ldots, \pi_\ell)$ for some $\ell \geq 1$.
   - For $i \in [\ell]$, if $\mathsf{multi\text{-}BARG}.\mathcal{V}(\mathsf{crs}, (\mathsf{v}_{i-1}, \mathsf{v}_i), \pi_i) \neq 1$, output 0.
   - If $\Pi^{(d+1)} = \emptyset$, break.
7. Let $c_0 = \mathsf{nxt\text{-}cnfg}_{\mathcal{M}}(\emptyset, z)$. Note that v_0 is the leftmost hash value.
8. If $\mathsf{SEH.Verify}(\mathsf{hk}, \mathsf{v}_0, [n+S], \rho, (z, c_0)) = 0$[18], output 0.
9. Output 1.

Theorem 7.8. *The above construction is an IVC scheme (Definition 7.7).*

Proof of Theorem 7.8.

Efficiency. Follows from efficiency of the underlying SEH and multi-BARG schemes, as in Remark 6.5.

Completeness. Follows from the completeness of the underlying multi-BARG scheme.

[18] We are abusing notation here – ρ is actually a sequence of openings and hash values which can be verified using the sequence of hash keys.

Soundness. Suppose towards contradiction that there exists a poly-size adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ and a non-negligible function ϵ such that for every $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l|l} z \in \{0,1\}^n \wedge \ x \notin \mathcal{L}_{\mathcal{M}} & 1^n \leftarrow \mathcal{A}_1(1^\lambda), \\ \wedge\ \mathcal{V}(\mathsf{crs}, x, \pi) = 1 & \mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda, n, 1^{S(n)}), \\ & (x = (z, 1^t, c_t), \pi) \leftarrow \mathcal{A}_2(\mathsf{crs}) \end{array}\right] \geq \epsilon(\lambda). \tag{20}$$

Let $c_0^z = \mathsf{nxt\text{-}cnfg}_{\mathcal{M}}(\emptyset, z)$ and $c_i^z = \mathsf{nxt\text{-}cnfg}_{\mathcal{M}}(c_{i-1}^z, z)$ for $i \in [t]$ be $\mathcal{M}$'s true configurations when run for t timesteps on input z. For $I = (i_1, \ldots, i_\lambda \in [2k])$, let Gen_I be identical to Gen except that it sets $i_1, \ldots, i_\lambda$ as above, as opposed to setting $i_1 = \cdots = i_\lambda = 1$. In addition, Gen_I outputs the multi-BARG trapdoor $\mathsf{td} = (\mathsf{seBARG.td}, \mathsf{SEH.td})$. By the index hiding property of the multi-BARG scheme, Equation (20) implies that for any I,

$$\Pr\left[\begin{array}{l|l} z \in \{0,1\}^n \wedge \ c_t \neq c_t^z & 1^n \leftarrow \mathcal{A}_1(1^\lambda), \\ \wedge\ \mathcal{V}(\mathsf{crs}, x, \pi) = 1 & (\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{Gen}_I(1^\lambda, n, 1^{S(n)}), \\ & (x = (z, 1^t, c_t), \pi) \leftarrow \mathcal{A}_2(\mathsf{crs}) \end{array}\right] \geq \epsilon(\lambda). \tag{21}$$

We define an extractor $\mathsf{Extract}$ which takes as input the multi-BARG trapdoor $\mathsf{td} = (\mathsf{seBARG.td}, \mathsf{SEH.td})$, an instance $x = (z, t, c_t)$, a proof $\pi = \left(\Pi^{(0)}, \Pi^{(1)}, \ldots, \Pi^{(\lambda)}\right)$, and a timestep $i \in [t]$. If configurations c_{i-1} and c_i are stored in $\Pi^{(0)}$, $\mathsf{Extract}$ outputs (z, c_{i-1}, c_i). If not, $\mathsf{Extract}$ locates the hash value v_i which covers timestep i (i.e., v_i is a depth d hash of configurations including c_{i-1} and c_i), and outputs $\mathsf{SEH.Extract}(\mathsf{SEH.td}, \mathsf{v}_i)$.

To contradict Equation (21), it suffices to show by induction that for all $i \in \{0, \ldots, t\}$, if we set $I = (i_1, \ldots, i_\lambda \in [2k])$ so that the hash value v_i which covers timestep i is statistically binding on configurations (c_{i-1}, c_i), then

$$\Pr\left[\begin{array}{l|l} z \in \{0,1\}^n \wedge \ \mathcal{V}(\mathsf{crs}, x, \pi) = 1 & 1^n \leftarrow \mathcal{A}_1(1^\lambda), \\ \wedge\ c_i \neq c_i^z & (\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{Gen}_I(1^\lambda, n, 1^{S(n)}), \\ & (x = (z, 1^t, c_t), \pi) \leftarrow \mathcal{A}_2(\mathsf{crs}), \\ & (z', c_{i-1}, c_i) = \mathsf{Extract}(\mathsf{td}, x, \pi, i) \end{array}\right] \leq \mathsf{negl}(\lambda).$$

Induction base. Since $\mathcal{V}$ verifies that ρ opens the leftmost hash value to the true initial configuration, the somewhere statistically binding property of SEH implies that

$$\Pr\left[\begin{array}{l|l} z \in \{0,1\}^n \wedge \ \mathcal{V}(\mathsf{crs}, x, \pi) = 1 & 1^n \leftarrow \mathcal{A}_1(1^\lambda), \\ \wedge\ (z' \neq z \vee c_0 \neq c_0^z) & (\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{Gen}_I(1^\lambda, n, 1^{S(n)}), \\ & (x = (z, 1^t, c_t), \pi) \leftarrow \mathcal{A}_2(\mathsf{crs}), \\ & (z', c_0, c_1) = \mathsf{Extract}(\mathsf{td}, x, \pi, 1) \end{array}\right] \leq \mathsf{negl}(\lambda),$$

i.e., the probability that c_0 is incorrect is $\mathsf{negl}(\lambda)$.

Induction step. Since the event that

$$z \in \{0,1\}^n \ \wedge \ \left(z' \neq z \ \vee \ c_{i-1} \neq c_{i-1}^z\right) \ \wedge \ \mathcal{V}(\mathsf{crs}, x, \pi) = 1$$

is detectable by an efficient distinguisher, the inductive hypothesis and the index hiding property of multi-BARG imply that

$$\Pr\left[\begin{array}{l|l} z \in \{0,1\}^n \wedge \ \mathcal{V}(\mathsf{crs}, x, \pi) = 1 & 1^n \leftarrow \mathcal{A}_1(1^\lambda), \\ \wedge\ \left(z' \neq z \vee c_{i-1} \neq c_{i-1}^z\right) & (\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{Gen}_I(1^\lambda, n, 1^{S(n)}), \\ & (x = (z, 1^t, c_t), \pi) \leftarrow \mathcal{A}_2(\mathsf{crs}), \\ & (z', c_{i-1}, c_i) = \mathsf{Extract}(\mathsf{td}, x, \pi, i) \end{array}\right] \leq \mathsf{negl}(\lambda).$$

Let multi-BARG.$\mathcal{E}$ be the hashed multi-hop BARG extractor given by the somewhere argument of knowledge property. We define an extractor $\mathsf{Extract}'$ which behaves identically to $\mathsf{Extract}$, except that if configurations c_{i-1} and c_i are not stored in $\Pi^{(0)}$, $\mathsf{Extract}'$ locates the hash value v_i which covers timestep i (i.e., v_i is a depth d hash of configurations including c_{i-1} and c_i), immediately previous hash value v_{i-1}, and corresponding proof π_i and outputs multi-BARG.$\mathcal{E}(\mathsf{seBARG.td}, (\mathsf{v}_{i-1}, \mathsf{v}_i), \pi_i)$. The somewhere argument of knowledge property of multi-BARG implies that

$$\Pr\left[\begin{array}{l|l} z \in \{0,1\}^n \wedge \ \mathcal{V}(\mathsf{crs}, x, \pi) = 1 & 1^n \leftarrow \mathcal{A}_1(1^\lambda), \\ \wedge \Big((z', c_{i-1}, c_i) \neq \mathsf{Extract}(\mathsf{td}, x, \pi, i) & (\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{Gen}_I(1^\lambda, n, 1^{S(n)}), \\ \quad \vee\ c_i \neq \mathsf{nxt\text{-}cnfg}_{\mathcal{M}}(c_{i-1}, z)\Big) & (x = (z, 1^t, c_t), \pi) \leftarrow \mathcal{A}_2(\mathsf{crs}), \\ & (z', c_{i-1}, c_i) = \mathsf{Extract}'(\mathsf{td}, x, \pi, i) \end{array}\right] \leq \mathsf{negl}(\lambda),$$

Note that c_i is only incorrect if c_{i-1} is incorrect or c_i is not the next configuration after c_{i-1}. Thus the probability that c_i is incorrect is at most $\mathsf{negl}(\lambda) + \mathsf{negl}(\lambda) = \mathsf{negl}(\lambda)$.

□

References

[BCCT12] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In Shafi Goldwasser, editor, *Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012*, pages 326–349. ACM, 2012. 1

[BCCT13] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for SNARKS and proof-carrying data. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 111–120. ACM, 2013. 1, 5, 8

[BCI+13] Nir Bitansky, Alessandro Chiesa, Yuval Ishai, Rafail Ostrovsky, and Omer Paneth. Succinct non-interactive arguments via linear interactive proofs. In Amit Sahai, editor, *Theory of Cryptography - 10th Theory of Cryptography Conference, TCC 2013, Tokyo, Japan, March 3-6, 2013. Proceedings*, volume 7785 of *Lecture Notes in Computer Science*, pages 315–333. Springer, 2013. 1

[BCL+21] Benedikt Bünz, Alessandro Chiesa, William Lin, Pratyush Mishra, and Nicholas Spooner. Proof-carrying data without succinct arguments. In *Advances in Cryptology - CRYPTO 2021 - 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16-20, 2021, Proceedings, Part I*, pages 681–710, 2021. 5

[BCMS20] Benedikt Bünz, Alessandro Chiesa, Pratyush Mishra, and Nicholas Spooner. Recursive proof composition from accumulation schemes. In *Theory of Cryptography - 18th International Conference, TCC 2020, Durham, NC, USA, November 16-19, 2020, Proceedings, Part II*, pages 1–18, 2020. 5

[BCTV14] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. In *CRYPTO 2014*, pages 276–294, 2014. 5

[BDGM19] Zvika Brakerski, Nico Döttling, Sanjam Garg, and Giulio Malavolta. Leveraging linear decryption: Rate-1 fully-homomorphic encryption and time-lock puzzles. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019, Part II*, volume 11892 of *LNCS*, pages 407–437. Springer, Heidelberg, December 2019. 4, 11, 12, 18, 21, 67, 68, 69

[BGLS03] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures. In *Proceedings of Eurocrypt '03*, volume 2656 of LNCS, pages 416–432, 2003. 7, 8, 50

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *ITCS 2012*, pages 309–325. ACM, January 2012. 11

[BHK17] Zvika Brakerski, Justin Holmgren, and Yael Tauman Kalai. Non-interactive delegation and batch NP verification from standard computational assumptions. In Hamed Hatami, Pierre McKenzie, and Valerie King, editors, *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 474–482. ACM, 2017. 1

[BKK+18] Saikrishna Badrinarayanan, Yael Tauman Kalai, Dakshita Khurana, Amit Sahai, and Daniel Wichs. Succinct delegation for low-space non-deterministic computation. In Ilias Diakonikolas, David Kempe, and Monika Henzinger, editors, *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25-29, 2018*, pages 709–721. ACM, 2018. 1

[BMRS20] Joseph Bonneau, Izaak Meckler, Vanishree Rao, and Evan Shapiro. Coda: Decentralized cryptocurrency at scale. Cryptology ePrint Archive, Report 2020/352, 2020. https://ia.cr/2020/352. 5

[BPR15] Nir Bitansky, Omer Paneth, and Alon Rosen. On the cryptographic hardness of finding a nash equilibrium. In Venkatesan Guruswami, editor, *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 1480–1498. IEEE Computer Society, 2015. 5

[But21] Vitalik Buterin. An approximate introduction to how zk-snarks are possible, 2021. 1

[BV11] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In Rafail Ostrovsky, editor, *52nd FOCS*, pages 97–106. IEEE Computer Society Press, October 2011. 11

[CCDW20] Weikeng Chen, Alessandro Chiesa, Emma Dauterman, and Nicholas P. Ward. Reducing participation costs via incremental verification for ledger systems. Cryptology ePrint Archive, Report 2020/1522, 2020. https://ia.cr/2020/1522. 5

[CCH+19] Ran Canetti, Yilei Chen, Justin Holmgren, Alex Lombardi, Guy N. Rothblum, Ron D. Rothblum, and Daniel Wichs. Fiat-Shamir: from practice to theory. In Moses Charikar and Edith Cohen, editors, *51st ACM STOC*, pages 1082–1090. ACM Press, June 2019. 1

[CJJ21a] Arka Rai Choudhuri, Abhihsek Jain, and Zhengzhong Jin. Snargs for $\mathcal{P}$ from LWE. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*, pages 68–79. IEEE, 2021. 1, 2, 3, 5, 16, 17

[CJJ21b] Arka Rai Choudhuri, Abhishek Jain, and Zhengzhong Jin. Non-interactive batch arguments for np from standard assumptions. Cryptology ePrint Archive, Paper 2021/807, 2021. https://eprint.iacr.org/2021/807. 1, 3, 5

[COS20] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. In *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part I*, pages 769–793, 2020. 5

[CT10] Alessandro Chiesa and Eran Tromer. Proof-carrying data and hearsay arguments from signature cards. In *Innovations in Computer Science - ICS 2010, Tsinghua University, Beijing, China, January 5-7, 2010. Proceedings*, pages 310–331, 2010. 5

[CTV13] Stephen Chong, Eran Tromer, and Jeffrey A. Vaughan. Enforcing language semantics using proof-carrying data. *IACR Cryptol. ePrint Arch.*, page 513, 2013. 5

[CTV15] Alessandro Chiesa, Eran Tromer, and Madars Virza. Cluster computing in zero knowledge. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, pages 371–403, 2015. 5

[DGI+19a] Nico Döttling, Sanjam Garg, Yuval Ishai, Giulio Malavolta, Tamer Mour, and Rafail Ostrovsky. Trapdoor hash functions and their applications. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part III*, volume 11694 of *Lecture Notes in Computer Science*, pages 3–32. Springer, 2019. 4

[DGI+19b] Nico Döttling, Sanjam Garg, Yuval Ishai, Giulio Malavolta, Tamer Mour, and Rafail Ostrovsky. Trapdoor hash functions and their applications. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 3–32. Springer, Heidelberg, August 2019. 11

[FS87] 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. 1

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 626–645. Springer, Heidelberg, May 2013. 1

[GH19a] Craig Gentry and Shai Halevi. Compressible FHE with applications to PIR. In Dennis Hofheinz and Alon Rosen, editors, *Theory of Cryptography - 17th International*

Conference, TCC 2019, Nuremberg, Germany, December 1-5, 2019, Proceedings, Part II, volume 11892 of *Lecture Notes in Computer Science*, pages 438–464. Springer, 2019. 4

[GH19b] Craig Gentry and Shai Halevi. Compressible FHE with applications to PIR. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019, Part II*, volume 11892 of *LNCS*, pages 438–464. Springer, Heidelberg, December 2019. 11

[Gro10] Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 321–340. Springer, Heidelberg, December 2010. 1

[GV22] Rishab Goyal and Vinod Vaikuntanathan. Locally verifiable signature and key aggregation, 2022. 8

[GW11] Craig Gentry and Daniel Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In Lance Fortnow and Salil P. Vadhan, editors, *43rd ACM STOC*, pages 99–108. ACM Press, June 2011. 1

[HKW15] Susan Hohenberger, Venkata Koppula, and Brent Waters. Universal signature aggregators. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 3–34. Springer, 2015. 2, 7

[HW15] Pavel Hubacek and Daniel Wichs. On the communication complexity of secure function evaluation with long output. In Tim Roughgarden, editor, *ITCS 2015*, pages 163–172. ACM, January 2015. 2, 3, 4, 11, 13

[JKKZ21] Ruta Jawale, Yael Tauman Kalai, Dakshita Khurana, and Rachel Yun Zhang. Snargs for bounded depth computations and PPAD hardness from sub-exponential LWE. In Samir Khuller and Virginia Vassilevska Williams, editors, *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*, pages 708–721. ACM, 2021. 1

[KB20] Assimakis Kattis and Joseph Bonneau. Proof of necessary work: Succinct state verification with fairness guarantees. Cryptology ePrint Archive, Report 2020/190, 2020. https://ia.cr/2020/190. 5

[KLVW22] Yael Kalai, Alex Lombardi, Vinod Vaikuntanathan, and Daniel Wichs. Boosting batch arguments and ram delegation. Unpublished manuscript, 2022. 1, 2, 9, 10, 12, 15, 17, 18

[KP16] Yael Tauman Kalai and Omer Paneth. Delegating RAM computations. In *Theory of Cryptography - 14th International Conference, TCC 2016-B, Beijing, China, October 31 - November 3, 2016, Proceedings, Part II*, pages 91–118, 2016. 9, 10

[KPY19] Yael Tauman Kalai, Omer Paneth, and Lisa Yang. How to delegate computations publicly. In Moses Charikar and Edith Cohen, editors, *51st ACM STOC*, pages 1115–1124. ACM Press, June 2019. 1, 5, 9, 17

[KPY20] Yael Tauman Kalai, Omer Paneth, and Lisa Yang. Delegation with updatable unambiguous proofs and PPAD-hardness. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part III*, volume 12172 of *LNCS*, pages 652–673. Springer, Heidelberg, August 2020. 1, 5, 8

[KRR13] Yael Tauman Kalai, Ran Raz, and Ron D. Rothblum. Delegation for bounded space. In *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 565–574, 2013. 1

[KRR14] Yael Tauman Kalai, Ran Raz, and Ron D. Rothblum. How to delegate computations: the power of no-signaling proofs. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 485–494, 2014. 1

[KVZ21] Yael Tauman Kalai, Vinod Vaikuntanathan, and Rachel Yun Zhang. Somewhere statistical soundness, post-quantum security, and snargs. In Kobbi Nissim and Brent Waters, editors, *Theory of Cryptography - 19th International Conference, TCC 2021, Raleigh, NC, USA, November 8-11, 2021, Proceedings, Part I*, volume 13042 of *Lecture Notes in Computer Science*, pages 330–368. Springer, 2021. 1

[Lip12] Helger Lipmaa. Progression-free sets and sublinear pairing-based non-interactive zero-knowledge arguments. In Ronald Cramer, editor, *TCC 2012*, volume 7194 of *LNCS*, pages 169–189. Springer, Heidelberg, March 2012. 1

[Mer88] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *CRYPTO'87*, volume 293 of *LNCS*, pages 369–378. Springer, Heidelberg, August 1988. 4

[Mic93] Silvio Micali. Fair public-key cryptosystems. In Ernest F. Brickell, editor, *CRYPTO'92*, volume 740 of *LNCS*, pages 113–138. Springer, Heidelberg, August 1993. 5

[Mic94] Silvio Micali. CS proofs (extended abstracts). In *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*, pages 436–453. IEEE Computer Society, 1994. 1, 8

[NT16] Assa Naveh and Eran Tromer. Photoproof: Cryptographic image authentication for any set of permissible transformations. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 255–271, 2016. 5

[OPWW15a] Tatsuaki Okamoto, Krzysztof Pietrzak, Brent Waters, and Daniel Wichs. New realizations of somewhere statistically binding hashing and positional accumulators. In Tetsu Iwata and Jung Hee Cheon, editors, *ASIACRYPT 2015*, volume 9452 of *Lecture Notes in Computer Science*, pages 121–145. Springer, 2015. 2, 4

[OPWW15b] Tatsuaki Okamoto, Krzysztof Pietrzak, Brent Waters, and Daniel Wichs. New realizations of somewhere statistically binding hashing and positional accumulators. In Tetsu Iwata and Jung Hee Cheon, editors, *ASIACRYPT 2015, Part I*, volume 9452 of *LNCS*, pages 121–145. Springer, Heidelberg, November / December 2015. 13

[RR20] Guy N. Rothblum and Ron D. Rothblum. Batch verification and proofs of proximity with polylog overhead. In Rafael Pass and Krzysztof Pietrzak, editors, *Theory of Cryptography - 18th International Conference, TCC 2020, Durham, NC, USA, November 16-19, 2020, Proceedings, Part II*, volume 12551 of *Lecture Notes in Computer Science*, pages 108–138. Springer, 2020. 1

[RRR18] Omer Reingold, Guy N. Rothblum, and Ron D. Rothblum. Efficient batch verification for UP. In Rocco A. Servedio, editor, *33rd Computational Complexity Conference, CCC 2018, June 22-24, 2018, San Diego, CA, USA*, volume 102 of *LIPIcs*, pages 22:1–22:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. 1

[Val08] Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In *Theory of Cryptography, Fifth Theory of Cryptography Conference, TCC 2008, New York, USA, March 19-21, 2008*, pages 1–18, 2008. 5, 8

[WW22] Brent Waters and David J. Wu. Batch arguments for NP and more from standard bilinear group assumptions. *IACR Cryptol. ePrint Arch.*, page 336, 2022. 1, 3, 5

A Homomorphic Encryption with Local Compression Compiler

In this section, we build a homomorphic encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval}, \mathsf{Dec})$ along with compression algorithms

$$(\mathsf{CompGen}, \mathsf{Compress}_1, \mathsf{LinEval}, \mathsf{Compress}_2, \mathsf{CompDec})$$

as defined in section 3.4. Our construction is based on the rate-1 homomorphic encryption by Brakerski et al. [BDGM19].

Construction. Our encryption scheme has a linear decrypt-and-multiply property (see [BDGM19, Definition 2.9]) for message space $\{0,1\}$ and circuit class $\mathcal{C}$, and the linearly homomorphic packed Regev encryption scheme as in [BDGM19, § 3.3]. Since our usage of the linearly homomorphic packed Regev encryption scheme is more specific than in [BDGM19], thus we use it directly below rather than abstracting out the required properties as in their rate-1 homomorphic encryption construction. Below we describe our homomorphic encryption with the compression compiler.

$\mathsf{Gen}(1^\lambda) \to (\mathsf{pk}, \mathsf{sk})$. The setup algorithm runs the FHE setup algorithm to sample the key pair as $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{FHE.Setup}(1^\lambda, 1^d)$.

$\mathsf{Enc}(\mathsf{pk}, \mu) \to \mathsf{ct}$. The encryption algorithm is the FHE encryption algorithm and computes the ciphertext as $\mathsf{ct} \leftarrow \mathsf{FHE.Enc}(\mathsf{pk}, \mu)$.

$\mathsf{Eval}(\mathsf{pk}, C, \mathbf{ct}) \to \mathbf{ct}'$. The evaluation algorithm simply runs the FHE evaluation algorithm as $\mathbf{ct}' \leftarrow \mathsf{FHE.Eval}(\mathsf{pk}, C, \mathbf{ct})$.

$\mathsf{Dec}(\mathsf{sk}, \mathsf{ct}) \to \mu$. The decryption algorithm is also the FHE decryption algorithm and computes the output as $\mu = \mathsf{FHE.Dec}(\mathsf{sk}, \mathsf{ct})$.

$\mathsf{CompGen}(\mathsf{pk}, \mathsf{sk}, 1^\ell) \to (\mathsf{pk}_\mathbf{c}, \mathsf{sk}_\mathbf{c})$. Let q be the LWE modulus chosen for the FHE scheme. Also, let χ be a B-bounded error distribution defined on $\mathbb{Z}$, and n, m be the LWE parameters. We discuss later how the parameters are selected. And, let $\ell_\mathsf{sk} = |\mathsf{sk}|$ denote the length of FHE secret key interpreted as a vector over $\mathbb{Z}_q$. That is, $\mathsf{sk} \in \mathbb{Z}_q^{\ell_\mathsf{sk}}$.

The compression parameter generator samples a random matrix $\mathbf{A} \leftarrow \mathbb{Z}_q^{n \times m}$, a secret matrix $\mathbf{S} \leftarrow \mathbb{Z}_q^{\ell \times n}$, and an error matrix $\mathbf{E} \leftarrow \chi^{\ell \times m}$. It computes matrix $\mathbf{B}$ as $\mathbf{B} = \mathbf{SA} + \mathbf{E}$, and samples the following sequence of $\ell \times \ell_\mathsf{sk}$ Regev ciphertexts:

$$\forall\, i \in [\ell], j \in [\ell_\mathsf{sk}], \qquad \mathsf{ck}_{i,j} = (\mathbf{AR}_{i,j}, \mathbf{BR}_{i,j} + \mathsf{sk}_j \cdot \mathbf{g}^\top \otimes \mathbf{e}_i)$$

where $\mathbf{R}_{i,j} \leftarrow \{0,1\}^{m \times \lceil \log q \rceil}$, $\mathsf{sk}_j \in \mathbb{Z}_q$ is the j-th element of the secret key sk, $\mathbf{g}^\top = (1, 2, \ldots, 2^{\lceil \log q \rceil - 1})$, and $\mathbf{e}_i \in \mathbb{Z}_q^\ell$ is the i-th unit vector. It outputs compression key pair $(\mathsf{pk}_\mathbf{c}, \mathsf{sk}_\mathbf{c})$ as

$$\mathsf{pk}_\mathbf{c} = (\mathsf{pk}, \{\mathsf{ck}_{i,j}\}_{i,j}), \quad \mathsf{sk}_\mathbf{c} = \mathbf{S}.$$

$\mathsf{Compress}_1(\mathsf{pk}_\mathbf{c}, \mathsf{ct}, \ell, i \in [\ell]) \to \mathsf{ct}_{\mathbf{c}_1}$. It parses the compression public key $\mathsf{pk}_\mathbf{c}$ as above, and let $\mathsf{vectorize} : [\ell] \times \{0,1\} \to \{0,1\}^\ell$ denote the function that takes as input an index $i \in [\ell]$, and a bit $\mu \in \{0,1\}$, and outputs a length-ℓ vector which contains μ in the i-th position and zeros everywhere else. Thus, $\mathsf{vectorize}(i, \mu) = \mu \cdot \mathbf{e}_i$ (i.e., the i-th unit vector mutliplied with bit μ).

The initial compression algorithm homomorphically evaluates $\mathsf{vectorize}(i, \cdot)$ on the ciphertext ct to compute $\mathbf{ct}_\mathsf{vec} = \mathsf{FHE.Eval}(\mathsf{pk}, \mathsf{vectorize}(i, \cdot), \mathsf{ct})$, a sequence of ℓ evaluated ciphertexts $\mathbf{ct}_\mathsf{vec} = (\mathsf{ct}_{\mathsf{vec},1}, \ldots, \mathsf{ct}_{\mathsf{vec},\ell})$. Next, let $\mathsf{Dec\&Mult}$ be the linear function that takes as input an FHE secret key $\mathsf{sk} \in \mathbb{Z}_q^{\ell_\mathsf{sk}}$, an FHE ciphertext ct encrypting a single bit μ, and a scaling factor ω, and outputs a scaled approximation $\mu\omega$ in the clear. We use the fact that $\mathsf{Dec\&Mult}$ is a linear function in sk over $\mathbb{Z}_q$. For more details, we refer the reader to [BDGM19, Definition 2.9]. The algorithm now defines ℓ linear function $f_i = (f_{i,1}, \ldots, f_{i,\ell_\mathsf{sk}}) \in \mathbb{Z}_q^{\ell_\mathsf{sk}}$ for $i \in [\ell]$ such that

$$\sum_{j \in [\ell_\mathsf{sk}]} f_{i,j} \cdot \mathsf{sk}_j = \mathsf{Dec\&Mult}(\mathsf{sk}, \mathsf{ct}_{\mathsf{vec},i}, 2^{\lceil \log q \rceil - 1}).$$

It then homomorphically evaluates the linear functions on the Regev ciphertexts $\{\mathsf{ck}_{i,j}\}_{i,j}$ as

$$\mathbf{c}_0 = \sum_{i,j} \mathsf{ck}_{i,j,1} \cdot \mathbf{g}^{-1}(f_{i,j}), \quad \mathbf{c}_1 = \sum_{i,j} \mathsf{ck}_{i,j,2} \cdot \mathbf{g}^{-1}(f_{i,j})$$

where $\mathsf{ck}_{i,j} = (\mathsf{ck}_{i,j,1}, \mathsf{ck}_{i,j,2}) \in \mathbb{Z}_q^{n \times \lceil \log q \rceil} \times \mathbb{Z}_q^{\ell \times \lceil \log q \rceil}$, and $\mathbf{g}^{-1}(\cdot)$ is the standard binary expansion function. Note that we get that $\mathbf{c}_0 \in \mathbb{Z}_q^n$, and $\mathbf{c}_1 \in \mathbb{Z}_q^\ell$. The algorithm outputs the compressed ciphertext as $\mathsf{ct}_{\mathbf{c}_1} = (\mathbf{c}_0, \mathbf{c}_1)$. And, it is interpreted as the following sequence of $(\ell + 1)$ sub-ciphertexts

$$\mathsf{ct}_{\mathbf{c}_1} = (\mathsf{sub\text{-}ct}_0, \mathsf{sub\text{-}ct}_1, \ldots, \mathsf{sub\text{-}ct}_\ell), \qquad \mathsf{sub\text{-}ct}_0 = \mathbf{c}_0 \in \mathbb{Z}_q^n, \quad \mathsf{sub\text{-}ct}_i = \mathbf{c}_{1,i} \in \mathbb{Z}_q,$$

where $\mathbf{c}_{1,i}$ is the i-th element of the second ciphertext vector.

$\mathsf{LinEval}(\mathsf{pk}_\mathbf{c}, \mathsf{ct}_{\mathbf{c}_1}^{(1)}, \ldots, \mathsf{ct}_{\mathbf{c}_1}^{(\ell)}) \to \mathsf{ct}'_{\mathbf{c}_1}$. It parses the compressed ciphertexts as $(\ell + 1)$ sub-ciphertexts each (as describe above). That is, $\mathsf{ct}_{\mathbf{c}_1}^{(i)} = (\mathsf{sub\text{-}ct}_0^{(i)}, \mathsf{sub\text{-}ct}_1^{(i)}, \ldots, \mathsf{sub\text{-}ct}_\ell^{(i)})$ for $i \in [\ell]$, and

outputs the linearly evaluated ciphertext as

$$\mathsf{ct}'_{\mathbf{c}_1} = \left(\sum_i \mathsf{sub\text{-}ct}_0^{(i)}, \sum_i \mathsf{sub\text{-}ct}_1^{(i)}, \ldots, \sum_i \mathsf{sub\text{-}ct}_\ell^{(i)} \right).$$

That is, each resulting sub-ciphertext is computed by simply adding all the individual underlying sub-ciphertexts. Therefore, the linear evaluation algorithm for compressed ciphertexts satisfies both the 'locality' and 'low depth' properties as defined in section 3.4. Basically, for ensuring the low depth property, it is enough to perform the ℓ sub-ciphertext additions in a tree-like manner.

$\mathsf{Compress}_2(\mathsf{ct}_{\mathbf{c}_1}) \to \mathsf{ct}_{\mathbf{c}}$. The final compression algorithm parses the ciphertext into $\ell+1$ components as above. That is, $\mathsf{ct}_{\mathbf{c}_1} = (\mathsf{sub\text{-}ct}_0, \mathsf{sub\text{-}ct}_1, \ldots, \mathsf{sub\text{-}ct}_\ell)$. And, it compresses each component individually as follows:

$$\mathsf{sub\text{-}ct}_{\mathbf{c},j} = \mathsf{Compress}_2(\mathsf{sub\text{-}ct}_j) = \begin{cases} \mathsf{sub\text{-}ct}_j & \text{if } j = 0 \\ \lfloor \mathsf{sub\text{-}ct}_j \rceil_2 & \text{otherwise.} \end{cases}$$

Here $\lfloor \cdot \rceil_2 : \mathbb{Z}_q \to \{0,1\}$ outputs the most significant bit of the number. That is, $\lfloor x \rceil_2 = 1$ for $x \geq 2^{\lceil \log q \rceil - 1}$ and 0 otherwise. It outputs the fully compressed ciphertext as $\mathsf{ct}_{\mathbf{c}} = (\mathsf{sub\text{-}ct}_{\mathbf{c},0}, \mathsf{sub\text{-}ct}_{\mathbf{c},1}, \ldots, \mathsf{sub\text{-}ct}_{\mathbf{c},\ell})$.

$\mathsf{CompDec}(\mathsf{sk}_{\mathbf{c}}, \mathsf{ct}_{\mathbf{c}}) \to (\mu_1, \ldots, \mu_\ell)$. The decryption algorithm for fully compressed ciphertexts parses the compressed secret key $\mathsf{sk}_{\mathbf{c}} = \mathbf{S}$, and let $\mathbf{s}_i^\top$ be the i-th row of the matrix $\mathbf{S}$. It parses the ciphertext as $\mathsf{ct}_{\mathbf{c}} = (\mathsf{sub\text{-}ct}_{\mathbf{c},0}, \mathsf{sub\text{-}ct}_{\mathbf{c},1}, \ldots, \mathsf{sub\text{-}ct}_{\mathbf{c},\ell})$.

It outputs the message bits μ_i as $\mu_i = (\mathsf{sub\text{-}ct}_{\mathbf{c},i} - \lfloor \mathbf{s}_i^\top \cdot \mathsf{sub\text{-}ct}_{\mathbf{c},0} \rceil_2) \mod 2$ for $i \in [\ell]$.

Security and setting the LWE parameters. The LWE parameters are set exactly as in [BDGM19]. Also, the proof of correctness, compactness, and security are based on [BDGM19], except we do not have the randomized shift before rounding, thus at the time of proving correctness this leaves a negligible error over the choice of random coins of setup and encryption.