Incrementally Verifiable Computation via Rate-1 Batch Arguments

Omer Paneth
Tel Aviv University

Tel Aviv, Israel
omerpa@tauex.tau.ac.il

Rafael Pass
Cornell Tech and Tel Aviv University

New York, USA and Tel Aviv, Israel
rafael@cs.cornell.edu

***Abstract*—Non-interactive delegation schemes enable producing succinct proofs (that can be efficiently verified) that a machine M transitions from c_1 to c_2 in a certain number of deterministic steps. We here consider the problem of efficiently *merging* such proofs: given a proof Π_1 that M transitions from c_1 to c_2, and a proof Π_2 that M transitions from c_2 to c_3, can these proofs be efficiently merged into a single short proof (of roughly the same size as the original proofs) that M transitions from c_1 to c_3? To date, the only known constructions of such a mergeable delegation scheme rely on strong non-falsifiable "knowledge extraction" assumptions. In this work, we present a provably secure construction based on the standard LWE assumption.**

As an application of mergeable delegation, we obtain a construction of incrementally verifiable computation (IVC) (with polylogarithmic length proofs) for any (unbounded) polynomial number of steps based on LWE; as far as we know, this is the first such construction based on any falsifiable (as opposed to knowledge-extraction) assumption. The central building block that we rely on, and construct based on LWE, is a *rate-1 batch argument* (BARG): this is a non-interactive argument for NP that enables proving k NP statements $x_1, ..., x_k$ with communication/verifier complexity $m + o(m)$, where m is the length of one witness. Rate-1 BARGs are particularly useful as they can be recursively composed a super-constant number of times.

I. Introduction

Consider some very long computation spanning over multiple generations of humans. Is there a way for the current generation to ensure that the current state of the computation is correct? The notion of incrementally verifiable computation (IVC), proposed by Valiant [1], addresses exactly this problem.

An IVC scheme for a machine $\mathcal{M}$ and time bound T consists of a key generation algorithm, and algorithms for updating and verifying proofs. The key-generation algorithm takes a security parameter λ and outputs a public key that is used to prove and verify statements of the form $(\mathsf{cf}, \mathsf{cf}', t)$ claiming that $\mathcal{M}$ starting from configuration cf reaches configuration cf' after t steps. To prove a statement $(\mathsf{cf}_0, \mathsf{cf}_t, t)$ we start from from the trivial statement $(\mathsf{cf}_0, \mathsf{cf}_0, 0)$ and the empty proof $\Pi_0 = \mathcal{E}$ which is always accepted. We then apply the proof updating algorithm t times. The update algorithm takes an accepting proof Π_i for the statement $(\mathsf{cf}_0, \mathsf{cf}_i, i)$ were $i < T$ and produces an accepting proof Π_{i+1} for the next statement $(\mathsf{cf}_0, \mathsf{cf}_{i+1}, i+1)$.[1]

The computational soundness requirement states that no polynomial-size attacker can find an accepting proof of a false statement with $t \leq T$. For the IVC to be useful, we additionally need the scheme to satisfy an efficiency requirement: the proof for a statement $x = (\mathsf{cf}, \mathsf{cf}', t)$ is of length $\mathrm{poly}(\lambda, \log T)$ (that is, essentially independent of T) and the time to update and verify proofs is $|x| \cdot \mathrm{poly}(\lambda, \log T)$. One may also consider *weakly-efficient* IVC where the proof length, as well as update and verification times are only required to be sublinear (as opposed to poly-logarithmic) in T.

The notion of IVC can be seen as a strengthening of non-interactive delegation of computation (also known as SNARGs for P). In delegation we require the same proof length and verification time requirements as IVC. A delegation proof for a statement $x = (\mathsf{cf}, \mathsf{cf}', t)$ can be generated in time $|x| \cdot \mathrm{poly}(\lambda, T)$. However, in contrast to IVC, delegation proofs may not be updated.

IVC, on top of being a fascinating notion in its own right, has several applications to delegation:

- **Delegating computation with transferable intermediate proofs:** Consider a client that outsourcing a long computation to an untrusted server. Using delegation, the client can make sure that final result is correct. However, if the the server does not perform the computation correctly, the client may only realize this once the computation is over. Instead, using IVC, the client can ask the server to provide the current state of the computation together with a proof of correctness at any point during the computation. If at some point the client detects cheating, or if the cloud is unable to continue the

The first author is supported by an Azrieli Faculty Fellowship, Len Blavatnik and the Blavatnik Foundation, the Blavatnik Interdisciplinary Cyber Research Center at Tel Aviv University, and ISF grant 1789/19. The second author is supported in part by NSF CNS-2149305, NSF Award SATC-1704788, NSF Award RI-1703846, CNS-2128519, AFOSR Award FA9550-18-1-0267, and a JP Morgan Faculty Award. This material is based upon work supported by DARPA under Agreement No. HR00110C0086. 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. Work done primarily while second author was a distinguished scholar at the Institute for Advanced Study at Tel-Aviv University.

[1] We typically require the update algorithm to work for *any* accepting proof, even adversarially generated ones. This means that given any accepting proof for some partial computation, we are not only guaranteed that the current state is correct, but also that we can continue to update the state and its proof.

978-1-6654-5519-0/22/$31.00 ©2022 IEEE
DOI 10.1109/FOCS54457.2022.00102

computation, the client can hire another server to continue the verifiable computation from its most recent verified state

- **Time and Space-preserving delegation:** While delegation schemes enable fast verification, the prover typically has significant computational overhead. In particular, in existing solutions based on standard assumption [2]–[4] both the time and the space of the prover grow polynomially with running time of the original computation. This is the case even if the original computation requires small space. Ideally we would like a delegation scheme that preserves both the time and space complexity of the original computation up to quasi-linear overhead. While such efficient arguments have been the focus of recent work, existing solutions either have heuristic security [5]–[7], or are restricted to the designated-verifier setting [8].[2] An IVC directly enables proving the correctness of a time-T space-S computation in time $T \cdot \mathrm{poly}(\lambda, \log T)$ and space $\mathrm{poly}(\lambda, S, \log T)$.

As we shall discuss below, however, known construction of IVC are either based on heuristics, non-falsifiable assumptions (so-called knowledge-extraction assumptions) or only achieve weak efficiency.

IVC from proof merging. Valiant's original work proposed an approach for constructing IVC based on *proof merging*: Assume that we are given a proof Π_1 for the statement $(\mathsf{cf}, \mathsf{cf}', t_1)$ and another proof Π_2 for the statement $(\mathsf{cf}', \mathsf{cf}'', t_2)$. Can we efficiently (in time that is independent of t_1, t_2) "merge" them into a single proof for the statement $(\mathsf{cf}, \mathsf{cf}'', t_1 + t_2)$ that is roughly the same length as each of the original proofs? As observed by Valiant, proofs of length $\mathrm{poly}(\lambda, \log T)$ that can be recursively merged $\log T$ times imply IVC. The idea is to incrementally construct a proof for each step of the computation as it is executed and merge the proofs into a single proof in the form of a binary tree. (Or, if the computation time t is not a power of two, into at most $\log t$ proofs.)

IVC under non-falsifiable assumptions. Valiant showed how to instantiate this approach and construct IVC and mergeable proofs, albeit, under a highly non-standard assumption: the existence succinct non-interactive arguments of knowledge for NP (SNARKs) such that for any attacker A of size $\leq \lambda^{\log T}$ there exists a knowledge extractor of *linear size* $|A| \cdot \mathrm{poly}(\lambda)$. All known SNARK constructions are based on non-falsifiable so-called extractability/knowledge assumptions or heuristic assumptions such as Fiat-Shamir and are subject to strong limitations [9], [10]. Currently, we do not know of any candidate SNARK construction with an explicit extractor, let alone a linear one.

To relax the knowledge extraction assumption, Bitansky et al. [6] showed that by merging proofs according to a tree of fan-out λ instead of a binary tree, we can reduce the depth of the "merge tree" to $\log_\lambda T$ instead of $\log T$. This gives IVC for any bound $T = \mathrm{poly}(\lambda)$ based on SNARKs with a polynomial-size (as opposed to linear-size) knowledge extractor. Assuming SNARKs with sub-exponential security, their construction gives IVC for $T = \lambda^{o(\log \lambda)}$. A recent line of work [11]–[13] obtained IVC without relying directly on SNARKs, however, the assumptions underlying their constructions are known to imply SNARKS. Accordingly, they suffer from the same drawbacks and are subject to the same limitations as previous IVC constructions.

Weakly-efficient IVC under falsifiable assumptions. The work of Kalai, Paneth and Yang (KPY) [2] provides the first non-trivial IVC scheme from a falsifiable assumptions on bilinear groups. Their scheme, however, only satisfies weak efficiency with proofs of length $2^{\sqrt{O(\log T \cdot \log \lambda)}}$. Their IVC is based on a weak form a proof merging where many proofs are merged in one shot, and the merged proof is longer than the original proof by a factor of $\mathrm{poly}(\lambda)$. To achieve sub-linear proof size, proofs must be merged in a tree of fan-out $> \lambda$.

Summarizing, to date, known IVC constructions require either non-falsifiable knowledge-extraction assumptions or heuristics, while weakly-efficient IVC is known under falsifiable assumptions. The state of the art thus leaves open the following question:

> *Is IVC possible under standard/falsifiable hardness assumptions?*

In this work, we resolve this question by presenting an IVC scheme assuming polynomial hardness of the LWE assumption.

A. Our results

Our main result is a new delegation scheme that enables merging $\mathrm{poly}(\lambda)$ many proofs into a single proof, by recursively merging pairs of proofs in a binary tree of depth $O(\log(\lambda))$. Our delegation scheme is for RAM computations (which implies delegation also for Turing machines).

Theorem 1.1 (Mergeable Delegation from LWE): Assuming the LWE assumption holds, there exists a mergeable delegation scheme for RAM computations.

Previously, proof systems that can be merged with such efficiently were only known based on non-falsifiable extractability/knowledge assumptions or heuristics. Following [1], any such a mergeable delegation scheme can be turned into an IVC scheme. This yields the first IVC scheme under any falsifiable assumption.

Theorem 1.2 (IVC from LWE): Assuming the LWE assumption holds, there exists an IVC for any time bound $T = \mathrm{poly}(\lambda)$.

More generally, for any $T = T(\lambda)$, we get IVC for bound $T(\lambda)$ under the T-hardness of LWE. This gives the first IVC scheme with for quasi-polynomial (or larger) time bound T under any assumption other than Valiant's SNARKs with linear-size knowledge extractor. In particular, by setting $\lambda = \mathrm{polylog}(T)$ and relying on the sub-exponential hardness of LWE we can get IVC with proofs of length $\mathrm{polylog}\, T$.

[2] In designated-verifier delegation, the public key is generated together with a secret key that is required to verify proofs.

Finally, as observed above, any IVC scheme directly implies a time- and space-preserving delegation scheme, and as such we also get the first space-preserving delegation scheme under falsifiable assumptions.

Theorem 1.3 (Time and Space-Preserving Delegation from LWE): For any $T = T(\lambda)$, as assuming the LWE problem is hard for algorithms of size $\mathrm{poly}(T)$, there exists an non-interactive delegation scheme where proving a time-T space-S computation, the prover requires time $T \cdot \mathrm{poly}(\lambda)$, and space $\mathrm{poly}(\lambda, S, \log T)$.

Rate-1 batch arguments for NP. Our main tool in constructing mergeable delegation scheme is a new non-interactive argument for conjunctions of NP statements. Given k NP statements with witnesses of length m we can give a computationally sound proof of length $m + O(m/\lambda) + \mathrm{poly}(\lambda, \log k)$ for their conjunction. This is a strengthening of the notion of batch arguments for NP [14] where the proof is of length $\mathrm{poly}(\lambda, m, \log k)$. Therefore, our arguments can be viewed as batch arguments with rate (the ratio between the witness length and the proof length) approaching 1. In contrast to plain batch arguments, where we must batch together many NP statements to achieve non-trivial efficiency, rate-1 batch arguments are useful even for two statements. Another important feature of rate-1 batch arguments is that they can be composed together recursively for a super-constant number of times without blowing up the proof size.

We note that in adaptive setting where the NP statements can be chosen by the adversary as a function of the public key, batch arguments (with any non-trivial rate) satisfying the standard notion of knowledge soundness are subject to strong limitations [14]. Therefore, our arguments satisfy the relaxed notion of somewhere argument of knowledge for adaptively chosen statements introduced in [3]. This notion is sufficient for our application to mergeable arguments. (See the technical overview for more details.)

We show how to leverage the rate-1 FHE construction of Brakerski et al. [15] together with the recent construction of (plain) batch arguments of Choudhuri, Jain and Jin [3] (both based on LWE) to construct rate-1 batch arguments.

Theorem 1.4: Assuming the LWE assumption holds, there exist rate-1 batch arguments for NP.

B. Technical Overview

We first elaborate on the construction of rate-1 batch arguments which is our main technical contribution. In Section I-B2 we describe the construction of mergeable proofs and IVC from from rate-1 batch batch arguments.

1) Rate-1 batch arguments: Our construction is based on standard batch arguments with arbitrary rate. Recall that a batch argument proves the conjunction of k NP statements $x^1, \ldots x^k$ with a proof of length $\mathrm{poly}(\lambda, m, \log k)$ where m is the size of one witness. In fact, we rely on a stronger notion known as batch arguments for the index language [3] were each statements x^i is simply the index i. In more detail, in a batch argument for the index language the prover and verifier are given the NP verification machine $\mathcal{M}$ that takes the statement $i \in [k]$ and a witness w^i. The prover is also given witnesses $w^1, \ldots, w^k$ such that $\mathcal{M}(i, w^i)$ accepts for every $i \in [k]$. The batch argument verifier runs in time $|\mathcal{M}| \cdot \mathrm{poly}(\lambda, m, \log k)$.[3]

The notion of soundness we require is somewhere argument of knowledge for adaptively chosen statements [3]. Roughly speaking, the requirement is that we can generate a programmed public key for the statement i together with a corresponding secret key such that the programmed public key is indistinguishable from an honestly generated public key (in particular, it hides i). Moreover, for every efficient adversary, if, given the programmed public key, the adversary produces an NP verification machine $\mathcal{M}$ together with an accepting proof Π, then, given the secret key, we can extract a witness w from Π such that $\mathcal{M}(i, w) = 1$ with overwhelming probability. Batch arguments for the index language satisfing somewhere argument of knowledge for adaptively chosen statements were constructed in [3] based on the hardness of LWE.

Our approach. To construct rate-1 batch arguments, our high-level idea is as follows. Given an NP verification machine $\mathcal{M}$, and given witnesses $w^1, \ldots, w^k \in \{0,1\}^m$, we splits each statement $i \in [k]$ into $n \leq m$ statements, each with a short witness. We then prove all $k \cdot n$ statements together with a single batch argument. In more detail, we define a new NP verification machine $\tilde{\mathcal{M}}$ for $k \cdot n$ statements indexed by pairs $(i, j) \in [k] \times [n]$ such that the witness $\tilde{w}^i_j$ for the statement (i, j) is of length $\mathrm{poly}(\lambda)$ (independent of m). Therefore, a batch argument for $\tilde{\mathcal{M}}$ has proof of length $\mathrm{poly}(\lambda, \log k)$.

To split the statement $i \in [k]$, we first split the witness w^i into n blocks $w^i = (w^i_1, \ldots, w^i_n)$ each of length $\ell = m/n$. Then, we emulate the execution of $\mathcal{M}(i, w^i)$ in a sequence of n intervals implemented by RAM machines $\mathcal{R}^i_1, \ldots, \mathcal{R}^i_n$ executed sequentially, with each machine starting from the final configuration of the previously executed machine. For every $j \in [n]$, the machine $\mathcal{R}^i_j$ has the witness block w^i_j hard-coded. For $j < n$ the machine $\mathcal{R}^i_j$ writes w^i_j to memory and terminates. The final machine $\mathcal{R}^i_n$ writes the final witness block w^i_n to memory and then, once the entire witness w^i is in memory, it emulates $\mathcal{M}(i, w^i)$ and accepts if and only if $\mathcal{M}$ accepts. Let cf^i_0 be the starting configuration of $\mathcal{R}^i_1$ and for $j \in [n]$, let cf^i_j be the final configuration of $\mathcal{R}^i_j$.

Shrinking the sub-statements with RAM delegation. Note that we cannot simply include the configuration cf^i_j as part of the witness $\tilde{w}^i_j$ since it is too long (recall that $\tilde{w}^i_j$ should be of length $\mathrm{poly}(\lambda)$ independent of m). Therefore, we instead verify the computation of each interval using a delegation scheme for RAM machines. Such delegation scheme allows us to verify that the machine $\mathcal{R}^i_j$ transitions from configuration cf^i_{j-1} to cf^i_j without knowing the (potentially long) configurations. Instead, to verify the delegation proof we only need to know the short digests $\mathsf{h}^i_{j-1}, \mathsf{h}^i_j$ of $\mathsf{cf}^i_{j-1}, \mathsf{cf}^i_j$ respectively. The length of the delegation proof is $\mathrm{poly}(\lambda)$ and the verification

[3] Batch arguments for the index language imply standard batch arguments by considering a verification machine $\mathcal{M}$ that has the instances $x^1, \ldots x^k$ hard-coded in it.

time is $|\mathcal{R}^i_j| \cdot \mathrm{poly}(\lambda)$. The soundness of the delegation scheme states that an efficient adversary cannot produce configurations $\mathsf{cf}^i_{j-1}, \mathsf{cf}^i_j$ together with digests $\mathsf{h}^i_{j-1}, \tilde{\mathsf{h}}^i_j$ and an accepting proof Π^i_j such that $\mathcal{R}^i_j$ transitions from cf^i_{j-1} to cf^i_{j-1} and h^i_{j-1} is the digest of cf^i_{j-1}, but $\tilde{\mathsf{h}}^i_j$ is not the digest of cf^i_j.

We proceed to describe a simplified version of the NP verification machine $\tilde{\mathcal{M}}$ and witness $\tilde{w}^i_j$: For every $(i,j) \in [k] \times [0,n]$ we compute the digest h^i_j of the configuration cf^i_j as above, and generate a delegation proof Π^i_j that the machine $\mathcal{R}^i_j$ transitions from cf^i_{j-1} to cf^i_j. We then compute a hash tree over the $k \cdot (n+1)$ pairs $\left\{y^i_j = (w^i_j, \mathsf{h}^i_j)\right\}$ (we set $w^i_0 = \bot$) and hard-code the root h of the tree into $\tilde{\mathcal{M}}$. The witness $\tilde{w}^i_j$ contains the pairs y^i_{j-i}, y^i_j, their authentication paths to h, and the delegation proof Π^i_j. To verify $\tilde{w}^i_j$, the machine $\tilde{\mathcal{M}}$ checks that the authentication paths are valid and that the delegation proof for the machine $\mathcal{R}^i_j$ (with w^i_j hard-coded) and the digests $\mathsf{h}^i_{j-1}, \mathsf{h}^i_j$ is accepting. If $j = 1$ (resp. $j = n$), $\tilde{\mathcal{M}}$ also checks that h^i_0 (resp. h^i_n) is the digest of the starting (resp. accepting) configuration.[4]

Towards somewhere argument of knowledge. If we simply use the underlying batch argument for $\tilde{\mathcal{M}}$ as a batch argument for $\mathcal{M}$ we cannot prove that it satisfies the somewhere argument of knowledge property. Recall that to prove somewhere argument of knowledge we must be able to program the public key and extract a complete witness w^i for $\mathcal{M}(i)$ from any accepting proof. However, the underlying batch argument for $\tilde{\mathcal{M}}$ only lets us extract one short witness $\tilde{w}^i_j$. We may try to execute the adversary with different public keys and extract $\tilde{w}^i_j$ for each $j \in [n]$. However, since the adversary may choose $\tilde{\mathcal{M}}$ adaptively, it may contain a different root h in each execution and, therefore, the extracted witness blocks may not be consistent.

To resolve this, we add to the proof a separate mechanism for extracting w^i based on the notion of somewhere extractable hashing [3], [16]. A somewhere extractable hash is a collision resistant hash that can shrinks k blocks $y^1, \ldots, y^k$ into a single root h and supports local opening. Additionally, in a somewhere extractable hash we can generate a programmed hash key for any index i (that is indistinguishable from an honestly generated key) together with a corresponding secret key that can be used to efficiently extract the block y^i from the root h. In fact, even if the root h is generated adversarially, as long as there exists a valid authentication path from y^i to h, we can extract y^i from h.

We modify the construction of $\tilde{\mathcal{M}}$ as follows: Instead of computing a hash tree directly over the $k \cdot n$ pairs $\left\{y^i_j = (w^i_j, \mathsf{h}^i_j)\right\}$, we compute the hash in two stages. First, for every $j \in [n]$, we use a somewhere extractable hash to compute a hash tree over the k pairs $y^1_j, \ldots, y^k_j$ and obtain a root h_j. Then we compute a (standard) hash tree over the n roots $\mathsf{h}_1, \ldots, \mathsf{h}_n$ and hard-code the root h into $\tilde{\mathcal{M}}$. The witness $\tilde{w}^i_j$ is unchanged except that we replace the direct authentication path from y^i_j to h with two paths: one from y^i_j to h_j and one from h_j to h (and similarly for y^i_{j-i}).

The final batch argument for $\mathcal{M}$ consist of the underlying batch argument for $\tilde{\mathcal{M}}$ together with the the n roots $\{\mathsf{h}_j\}$. To verify, the batch argument for $\mathcal{M}$, we check that the underlying batch argument for $\tilde{\mathcal{M}}$ is accepting and that the root h hard-codded in $\tilde{\mathcal{M}}$ is indeed the hash of $\mathsf{h}_1, \ldots, \mathsf{h}_n$.

To get a rate-1 proof, the somewhere extractable hash we use must also have good rate. We observe that instantiating the hash construction of [16] based on FHE, with the rate-1 FHE of [15] gives a somewhere extractable hash where each root h_j is of length $\ell + O(\ell/\lambda)$ (assuming that the block length ℓ is a sufficiently large polynomial in λ) and therefore the length of all n roots together is $m + O(m/\lambda)$.

Ensuring consistency. Our final step is to modify the NP verification machine $\tilde{\mathcal{M}}$ to check that the witness encoded in the machines $\mathcal{R}^i_1, \ldots, \mathcal{R}^i_n$ is the same witness extracted from the roots $\mathsf{h}_1, \ldots, \mathsf{h}_n$. Otherwise, there is no guarantee that the extracted witness is a valid. The idea is to change the way we compute the root h hard-codded in $\tilde{\mathcal{M}}$. Instead of computing a hash tree directly over the $k \cdot n$ pairs $\left\{y^i_j = (w^i_j, \mathsf{h}^i_j)\right\}$, we compute a hash tree over the n roots $\mathsf{h}_1, \ldots, \mathsf{h}_n$ (recall that each h_j is itself the root of a hash tree over the the k pairs $\left\{y^i_j\right\}$ using a somewhere extractable hash) and set h to be its root. Moreover, instead of a direct authentication path from y^i_j to h, we give two paths: one from y^i_j to h_j and one from h_j to h. Finally, to verify the batch argument for $\mathcal{M}$, we check that the underlying batch argument for $\tilde{\mathcal{M}}$ is accepting and that the root h hard-codded in $\tilde{\mathcal{M}}$ is indeed the hash of $\mathsf{h}_1, \ldots, \mathsf{h}_n$.

The proof of security. We argue that our final construction satisfies the somewhere argument knowledge property: To extract a witness for the statement $i \in [k]$, we program the somewhere extractable hash key and for every $j \in [n]$, we extract $\bar{y}^i_j = (\bar{w}^i_j, \bar{\mathsf{h}}^i_j)$ from the root h_j. We condition on the event that the batch argument proof for $\mathcal{M}$ is accepting and we show that the extracted witness $\bar{w}^i = (\bar{w}^i_1, \ldots, \bar{w}^i_n)$ must be valid with overwhelming probability. To this end, we emulate the machines $\mathcal{R}^i_1, \ldots, \mathcal{R}^i_n$ with the hard-coded witness blocks $\bar{w}^i_1, \ldots, \bar{w}^i_n$ and obtain the configurations $\bar{\mathsf{cf}}^i_0, \ldots, \bar{\mathsf{cf}}^i_n$. We say that the digest $\bar{\mathsf{h}}^i_j$ extracted from h_j is "good" if it is indeed the digest of $\bar{\mathsf{cf}}^i_j$. Below we argue that for every $j \in [n]$, $\bar{\mathsf{h}}^i_j$ is good with overwhelming probability. However, before proving that, we show that assuming all digests $\bar{\mathsf{h}}^i_j$ are good, the witness $\bar{w}^i$ must be valid.

To show that $\bar{w}^i$ is valid, consider an experiment where we program the public key of the underlying batch argument and extract the witness $\tilde{w}^i_n$. Since the programmed public key is indistinguishable from the real key (even when extracting from the somewhere extractable hash) we still have that $\bar{\mathsf{h}}^i_n$ is good with overwhelming probability. Recall the witness $\tilde{w}^i_n$ contains the pair $y^i_n = (w^i_n, \mathsf{h}^i_n)$ and its authentication path to h. We have that with overwhelming probability $\tilde{\mathcal{M}}$ accepts $\tilde{w}^i_j$ and, therefore, the authentication path is valid and h^i_n is the digest of the accepting configuration of $\mathcal{R}^i_n$. By collision resistance and the extraction guarantee of the somewhere extractable

[4] Assume WLOG that $\mathcal{R}^i_n$ clears its memory before accepting and, therefore, it has a unique accepting configuration.

hash, we have that $\bar{\mathsf{h}}^i_n = \mathsf{h}^i_n$. Since $\bar{\mathsf{h}}^i_n$ is good and, again, using collision resistance, we have that $\bar{\mathsf{cf}}^i_n$ must be the accepting configuration of $\mathcal{R}^i_n$ and, thus, $\bar{w}^i$ is valid.

It remains to argue that for every $j \in [n]$, $\bar{\mathsf{h}}^i_j$ is good with overwhelming probability. For $j = 0$, $\bar{\mathsf{h}}^i_0$ is good by definition. If for some $j > 1$, $\bar{\mathsf{h}}^i_{j-1}$ is good but $\bar{\mathsf{h}}^i_j$ is not with noticeable probability, then we break the soundness of the delegation scheme as follows. First, we program the public key of the underlying batch argument and extract the witness $\tilde{w}^i_j$. Since the programmed public key is indistinguishable from the real key (even when extracting from the somewhere extractable hash) we still have that $\bar{\mathsf{h}}^i_{j-1}$ is good but $\bar{\mathsf{h}}^i_j$ is not with noticeable probability. Recall the witness $\tilde{w}^i_j$ contains the pairs $y^i_{j-i} = (w^i_{j-1}, \mathsf{h}^i_{j-1})$ and $y^i_j = (w^i_j, \mathsf{h}^i_j)$, their authentication paths to h, and the delegation proof Π^i_j. We have that $\tilde{\mathcal{M}}$ accepts $\tilde{w}^i_j$ with overwhelming probability and, therefore, the authentication paths and the delegation proof are all valid. By collision resistance and the extraction guarantee of the somewhere extractable hash we have that $\bar{y}^i_{j-1} = y^i_{j-1}$ and $\bar{y}^i_j = y^i_j$, thus to break the soundness of the delegation scheme for the machine $\mathcal{R}^i_j$ with $\bar{w}^i_j$ hard-coded, we produce the configurations $\bar{\mathsf{cf}}^i_{j-1}, \bar{\mathsf{cf}}^i_j$, the digests $\bar{\mathsf{h}}^i_{j-1}, \bar{\mathsf{h}}^i_j$ and the proof Π^i_j.

2) Mergeable proofs and IVC: We overview our construction of mergeable delegation for RAM machines based on rate-1 batch arguments. Recall that in delegation for a RAM machines, we can verify that a RAM machine $\mathcal{R}$ transitions from configuration cf to cf' in t steps without knowing the (potentially long) configurations. Instead, the verifier is only given access to the short digests h, h' of $\mathsf{cf}, \mathsf{cf}'$ respectively. Soundness states that an efficient adversary cannot produce $(\mathsf{cf}, \mathsf{cf}', t)$ such that $\mathcal{R}$ transitions from cf to cf' in t steps, together with a "hashed statement" $(\mathsf{h}, \mathsf{h}', t)$ and an accepting proof Π such that h is indeed the digest of cf, but h' is not the digest of cf'. We note that delegation for RAM implies delegation for Turing machines since the verifier can compute the digests of the configurations in the statement itself.

The high-level idea behind out construction is as follows. Given two accepting proofs Π_1, Π_2 under some public key pk for the hashed statements $x_1 = (\mathsf{h}, \mathsf{h}', t_1)$ and $x_2 = (\mathsf{h}', \mathsf{h}'', t_2)$ respectively, we merge them into a single proof Π for $(\mathsf{h}, \mathsf{h}'', t_1 + t_2)$. The proof Π contains both statements x_1, x_2 as well as a batch argument for the following NP verification machine $\mathcal{M}$ with $k = 2$. The machine $\mathcal{M}$ has the hashed statements x_1, x_2, the public key pk and the code of the RAM machine $\mathcal{R}$ hard-coded. Given an index $i \in [2]$ and a witness Π_i, the machine $\mathcal{M}$ checks that the original delegation verifier given the public key pk, the RAM machine $\mathcal{R}$ and the hashed statement x_i accepts the proof Π_i. We verify a merged proof Π for a hashed statement $(\mathsf{h}, \mathsf{h}', t)$ as follows. The proof Π contains a pair of instances x_1, x_2 where $x_i = (\mathsf{h}_i, \mathsf{h}'_i, t_i)$ and a batch argument proof. We first verify that the hashed statements are indeed of the correct form. That is, $(\mathsf{h}_1, \mathsf{h}'_2, t_1 + t_2) = (\mathsf{h}, \mathsf{h}', t)$ and $\mathsf{h}'_1 = \mathsf{h}_2$. Then, we accept if and only if the batch argument proof for the machine $\mathcal{M}$ hard-coded with the hashed statements x_1, x_2 (as well as pk and $\mathcal{R}$) is accepting.

The proof of security. To argue the soundness of the merged proof, consider an adversary that cheats with noticeable probability ϵ. That is, the adversary outputs a true statement $(\mathsf{cf}, \mathsf{cf}'', t)$, a hashed statement $(\mathsf{h}, \mathsf{h}'', t)$ and an accepting merged proof Π such that h is the digest of cf, but h'' is not the digest of cf''. Let x_1, x_2 be the hashed statements contained in the proof. If the proof is accepting these statements must be of the form $x_1 = (\mathsf{h}, \mathsf{h}', t_1)$ and $x_2 = (\mathsf{h}', \mathsf{h}'', t_2)$ where $t_1 + t_2 = t$. Let cf' be the configuration that follows t_1 steps after cf and let $\tilde{\mathsf{h}}'$ be its digest. Conditioned on the adversary cheating, we have that either $\mathsf{h}' = \tilde{\mathsf{h}}'$ or $\mathsf{h}' \neq \tilde{\mathsf{h}}'$ must hold with probability at least $1/2$. Say that the adversary cheats and $\mathsf{h}' \neq \tilde{\mathsf{h}}'$ with probability $\geq \epsilon/2$ (the proof in the other case is similar). To break the underlying delegation scheme we run the adversary with a programmed public key and extract the witness Π_1 for $i = 1$. Since the programmed public key is indistinguishable from an honestly generated one, the event that the adversary cheats and $\tilde{\mathsf{h}}' \neq \tilde{\mathsf{h}}'$ still occurs with roughly the same probability $\geq \epsilon/2$. By the somewhere argument of knowledge property, if the batch argument proof in Π is accepting, then $\mathcal{M}(1, \Pi_1)$ accepts with all but negligible probability and, therefore, Π_1 is an accepting proof for the hashed statement x_1. To break the the underlying delegation scheme we output the true statement $(\mathsf{cf}, \mathsf{cf}', t_1)$, the hashed statement $x_1 = (\mathsf{h}, \mathsf{h}', t_1)$ where $\mathsf{h}' \neq \tilde{\mathsf{h}}'$ and the accepting proof Π_1. Similarly, if $\tilde{\mathsf{h}}' = \tilde{\mathsf{h}}'$ with probability at least $\epsilon/2$, we program the batch argument public key, extract the witness Π_2. To break the underlying delegation scheme we output the true statement $(\mathsf{cf}', \mathsf{cf}'', t_2)$, the hashed statement $x_2 = (\mathsf{h}', \mathsf{h}'', t_1)$ where $\mathsf{h}' = \tilde{\mathsf{h}}'$ and the accepting proof Π_2.

Recursive merging. Since we are using rate-1 batch arguments, we can marge proofs recursively with the proof length the proof growing from m to $m + O(m/\lambda) + \mathrm{poly}(\lambda)$ in each level. Therefore, we can recursively merge proofs $O(\lambda)$ times while keeping the length of the merged proofs bounded by $\mathrm{poly}(\lambda)$. However, to prove soundness of recursively merged proofs we apply the argument above inductively, halving the adversary's advantage with each level. Therefore, to ensure that the advantage remains noticeable we must bound the depth of nested proofs to be logarithmic or, more generally, $\log T$ assuming T-hardness. This allows us to marge T proofs and, therefore, it is sufficient for constructing IVC for bound T.

IVC from mergeable delegation. The construction of IVC from mergeable delegation follows the outline of [1]. The high-level idea is as follows. Given a statement $(\mathsf{cf}, \mathsf{cf}', t)$ we split it into $d \leq \log t$ consecutive statements $\left\{(\mathsf{cf}_i, \mathsf{cf}_{i+1}, t_i = 2^{\ell_i}) \in \mathcal{T}_{\mathcal{M}}\right\}_{i \in [d-1]}$ such that $(\mathsf{cf}_1, \mathsf{cf}_d) = (\mathsf{cf}, \mathsf{cf}')$ and for every i, $\ell_i > \ell_{i+1}$. The proof Π for $(\mathsf{cf}, \mathsf{cf}', t)$ consist of one mergeable delegation proof for each of the d segments, where the proof for the i-th segment is of level ℓ_i. To increment the proof Π by one step we create a new level-0 proof for the statement $(\mathsf{cf}', \mathsf{cf}'', 1)$ and append it to Π. If Π contains two proofs of the same level ℓ we merge them into a

single proof of level $\ell+1$ (note that such proofs will always be for two consecutive statements) . We repeat this merging operation until Π no longer contains two proof of the same level and it is a valid proof for the statement $(\mathsf{cf}, \mathsf{cf}'', t+1)$.

II. Preliminaries

For a deterministic Turing machine $\mathcal{M}$, let $\mathcal{U}_{\mathcal{M}}$ be the language that contains (x,t) if and only if the machine $\mathcal{M}$ accepts x in t steps. A deterministic RAM machine $\mathcal{R}$ with a word size of λ has random access to memory of size 2^λ bits and a local state of size $O(\lambda)$. At every step, the machine reads or writes a single memory bit and updates its state. For a Turing or RAM machine $\mathcal{M}$, we refer to the machine's state and the content of its memory/tapes at a given timestep as its configuration. Let $\mathcal{T}_{\mathcal{M}}$ be the language that contains $(\mathsf{cf}, \mathsf{cf}', t)$ if and only if the machine $\mathcal{M}$ starting from configuration cf transitions to configuration cf' in t steps.

A. Incrementally Verifiable Computation

In this section we define incrementally verifiable computation scheme for deterministic Turing machines. The definition is adapted from [2].

An incrementally verifiable computation scheme for deterministic Turing machines consists of algorithms $(\mathsf{G}, \mathsf{U}, \mathsf{V})$ with the following syntax:

- G: The randomized setup algorithm takes as input a security parameter $\lambda \in \mathbb{N}$. It outputs a public key pk.
- U: The deterministic update algorithm takes as input the public key pk, a machine $\mathcal{M}$, a statement $(\mathsf{cf}, \mathsf{cf}', t)$ and a proof Π. It outputs a proof Π'.
- V: The deterministic verifier algorithm takes as input the public key pk, a machine $\mathcal{M}$, a statement $(\mathsf{cf}, \mathsf{cf}', t)$ and a proof Π. It outputs an acceptance bit.

Definition 2.1: A $T(\cdot)$-secure incrementally verifiable computation scheme for deterministic Turing machines satisfies the following requirements:

Incremental Completeness. For every $\lambda \in \mathbb{N}$ and machine $\mathcal{M}$:

- For every configuration cf:

$$\Pr\left[\begin{array}{l}\mathsf{pk} \leftarrow \mathsf{G}(\lambda) \\ : \quad \mathsf{V}(\mathsf{pk}, \mathcal{M}, (\mathsf{cf}, \mathsf{cf}, 0), \mathcal{E}) = 1\end{array}\right] = 1 \ .$$

- For every $t < 2^\lambda$, pair of statements $x, x' \in \mathcal{T}_{\mathcal{M}}$ of the form $x = (\mathsf{cf}, \mathsf{cf}', t)$ and $x' = (\mathsf{cf}, \mathsf{cf}'', t+1)$ and a proof Π:

$$\Pr\left[\begin{array}{l}\mathsf{pk} \leftarrow \mathsf{G}(\lambda) \\ \Pi' \leftarrow \mathsf{U}(\mathsf{pk}, \mathcal{M}, x, \Pi) \\ : \quad \begin{array}{l}\mathsf{V}(\mathsf{pk}, \mathcal{M}, x, \Pi) = 1 \\ \mathsf{V}(\mathsf{pk}, \mathcal{M}, x', \Pi') = 0\end{array}\end{array}\right] = 0 \ .$$

Efficiency. In the incremental completeness experiments above:

- The generation algorithm runs in time $\mathrm{poly}(\lambda)$.
- The verifier and update algorithms run in time $(|\mathcal{M}| + |x|) \cdot \mathrm{poly}(\lambda)$.

$T(\cdot)$-Soundness. For every $\mathrm{poly}(T(\lambda))$-size adversary Adv, there exists a negligible function μ such that for every $\lambda \in \mathbb{N}$:

$$\Pr\left[\begin{array}{l}\mathsf{pk} \leftarrow \mathsf{G}(\lambda) \\ (\mathcal{M}, x = (\mathsf{cf}, \mathsf{cf}', t), \Pi) \leftarrow \mathsf{Adv}(\mathsf{pk}) \\ : \quad \begin{array}{l}t \le T(\lambda) \\ \mathsf{V}(\mathsf{pk}, \mathcal{M}, x, \Pi) = 1 \\ x \notin \mathcal{T}_{\mathcal{M}}\end{array}\end{array}\right] \le \mu(T(\lambda)) \ .$$

We say that the scheme is polynomially secure if it is $T(\cdot)$-secure for every polynomial T.

B. RAM delegation

In this section, we define non-interactive delegation for RAM. The definition is a strengthening of the definition in [2] (see discussion following Theorem 2.3). In a RAM delegation scheme, the prover convinces the verifier of a statement of the form $(\mathsf{cf}, \mathsf{cf}', t) \in \mathcal{T}_{\mathcal{R}}$. Since the configurations might be long, the verifier only needs to know a hashed down version of the statement $(\mathsf{h}, \mathsf{h}', t)$ where h, h' are short digests of $\mathsf{cf}, \mathsf{cf}'$ respectively.

A non-interactive delegation scheme for RAM consists of algorithms $(\mathsf{G}, \mathsf{H}, \mathsf{P}, \mathsf{V})$ with the following syntax:

- G: The randomized setup algorithm takes as input a security parameter $\lambda \in \mathbb{N}$. It outputs a public key pk.
- H: The deterministic digest algorithm takes as input the public key pk and a configuration $\mathsf{cf} \in \{0,1\}^*$. It outputs a digest h.
- P: The deterministic prover algorithm takes as input the public key pk, a RAM machine $\mathcal{R}$ and a statement $(\mathsf{cf}, \mathsf{cf}', t)$. It outputs a proof Π.
- V: The deterministic verifier algorithm takes as input the public key pk, a RAM machine $\mathcal{R}$, a hashed statement $(\mathsf{h}, \mathsf{h}', t)$ and a proof Π. It outputs an acceptance bit.

Definition 2.2: A $T(\cdot)$-secure non-interactive delegation scheme for RAM satisfies the following requirements.

Completeness. For every $\lambda \in \mathbb{N}$, RAM machine $\mathcal{R}$, $t \in [2^\lambda]$ and statement $(\mathsf{cf}, \mathsf{cf}', t) \in \mathcal{T}_{\mathcal{R}}$ we have that:

$$\Pr\left[\begin{array}{l}\mathsf{pk} \leftarrow \mathsf{G}(\lambda) \\ \mathsf{h} \leftarrow \mathsf{H}(\mathsf{pk}, \mathsf{cf}) \\ \mathsf{h}' \leftarrow \mathsf{H}(\mathsf{pk}, \mathsf{cf}') \\ \Pi \leftarrow \mathsf{P}(\mathsf{pk}, \mathcal{R}, (\mathsf{cf}, \mathsf{cf}', t)) \\ : \ \mathsf{V}(\mathsf{pk}, \mathcal{R}, (\mathsf{h}, \mathsf{h}', t), \Pi) = 1\end{array}\right] = 1 \ .$$

Efficiency. In the completeness experiment above:

- The generation algorithm runs in time $\mathrm{poly}(\lambda)$.
- The digest algorithm on cf runs in time $|\mathsf{cf}| \cdot \mathrm{poly}(\lambda)$ and outputs a digest of length $\mathrm{poly}(\lambda)$.
- The prover algorithm runs run in time $|\mathcal{R}| \cdot \mathrm{poly}(\lambda, t)$ and outputs a proof of length $\mathrm{poly}(\lambda)$.
- The verifier algorithm runs in time $|\mathcal{R}| \cdot \mathrm{poly}(\lambda)$.[5]

[5]The definition only bounds the verification time of honestly generated proofs. We can assume WLOG that the same bound holds also for maliciously generated proofs by capping the verifier execution time.

$T(\cdot)$**-Collision Resistance.** For every $\mathrm{poly}(T(\lambda))$-size adversary Adv, there exists a negligible function μ such that for every $\lambda \in \mathbb{N}$:

$$\Pr\left[\begin{array}{l}\mathsf{pk} \leftarrow \mathsf{G}(\lambda) \\ (\mathsf{cf}, \mathsf{cf}') \leftarrow \mathsf{Adv}(\mathsf{pk}) \\ \quad : \quad \begin{array}{l}\mathsf{cf} \neq \mathsf{cf}' \\ \mathsf{H}(\mathsf{pk}, \mathsf{cf}) = \mathsf{H}(\mathsf{pk}, \mathsf{cf}')\end{array}\end{array}\right] \leq \mu(T(\lambda)) \ .$$

$T(\cdot)$**-Soundness.** For every $\mathrm{poly}(T(\lambda))$-size adversary Adv, there exists a negligible function μ such that for every $\lambda \in \mathbb{N}$:

$$\Pr\left[\begin{array}{l}\mathsf{pk} \leftarrow \mathsf{G}(\lambda) \\ (\mathcal{R}, \mathsf{cf}, \mathsf{cf}', \mathsf{h}, \mathsf{h}', t, \Pi) \leftarrow \mathsf{Adv}(\mathsf{pk}) \\ \quad : \quad \begin{array}{l}t \leq T(\lambda) \\ \mathsf{V}(\mathsf{pk}, \mathcal{R}, (\mathsf{h}, \mathsf{h}', t), \Pi) = 1 \\ (\mathsf{cf}, \mathsf{cf}', t) \in \mathcal{T}_{\mathcal{R}} \\ \mathsf{h} = \mathsf{H}(\mathsf{pk}, \mathsf{cf}) \\ \mathsf{h}' \neq \mathsf{H}(\mathsf{pk}, \mathsf{cf}')\end{array}\end{array}\right] \leq \mu(T(\lambda)) \ .$$

We say that the scheme is polynomially secure if it is $T(\cdot)$-secure for every polynomial T.

Theorem 2.3 ([3]): Assuming the LWE problem is $T(\cdot)$-hard, there exists a $T(\cdot)$-secure non-interactive delegation scheme for RAM.

We remark that the notion of RAM delegation considered in [3] is slightly weaker than the notion in Definition 2.2. We explain how to modify the [3] construction to obtain our notion:

- In [3], the RAM machine $\mathcal{R}$ is fixed while in our notion the prover and verifier are given the description of $\mathcal{R}$. We can realize this by using the scheme of [3] for the universal machine and encoding the description of $\mathcal{R}$ as part of the machine's configuration.
- In [3], the setup algorithm is given the time t for the statement proven while in our notion the prover and verifier are given t as part of the statement. We can realize this by using λ copies the scheme of [3]: for every $i \in [\lambda]$ we generate a public key for time 2^i. To prove a statement with time t, we divide the t execution steps into at most λ segments, each of length that is a power of two. The proof contains each of the segments and its proof under one of the public keys.
- While [3] only argue polynomial security of their scheme, it is straightforward to extend their proof to show $T(\cdot)$-security under the $T(\cdot)$-hardness of LWE.

C. Tree Hash

A tree hash consist of algorithms $(\mathsf{G}, \mathsf{H}, \mathsf{V})$ with the following syntax:

G**:** The randomized generation algorithm takes as input the security parameter λ. It outputs a public key pk.

H**:** The deterministic hashing algorithm takes as input the public key pk and messages $x_1, \ldots, x_k \in \{0,1\}^\ell$. It outputs a hash h and opening $\Pi_1, \ldots, \Pi_k$.

V**:** The deterministic verification algorithm takes as input the public key pk, a hash h, a message $x \in \{0,1\}^\ell$, an index $i \in [k]$ and a proof Π. It outputs an acceptance bit.

Definition 2.4: A $T(\cdot)$-secure tree hash satisfies the following requirements:

Completeness. For every $\lambda \in \mathbb{N}$, $k \leq 2^\lambda$, messages $x_1, \ldots, x_k \in \{0,1\}^\ell$ and index $i \in [k]$:

$$\Pr\left[\begin{array}{l}\mathsf{pk} \leftarrow \mathsf{G}(\lambda) \\ (\mathsf{h}, (\Pi_1, \ldots, \Pi_k)) \leftarrow \mathsf{H}(\mathsf{pk}, (x_1, \ldots, x_k)) \\ \quad : \ \mathsf{V}(\mathsf{pk}, \mathsf{h}, x_i, i, \Pi_i) = 1\end{array}\right] = 1 \ .$$

Efficiency. In the completeness experiment above:

- The generation algorithm runs in time $\mathrm{poly}(\lambda)$.
- The hashing algorithm runs in polynomial time in its input length and outputs a hash of length $\mathrm{poly}(\lambda)$.
- The verification algorithm runs in time $\ell \cdot \mathrm{poly}(\lambda)$.

$T(\cdot)$**-Collision Resistance.** For every $\mathrm{poly}(T(\lambda))$-size adversary Adv there exists a negligible function μ such that for every $\lambda \in \mathbb{N}$:

$$\Pr\left[\begin{array}{l}\mathsf{pk} \leftarrow \mathsf{G}(\lambda) \\ (\mathsf{h}, x, x', i, \Pi, \Pi') \leftarrow \mathsf{Adv}(\mathsf{pk}) \\ \quad : \quad \begin{array}{l}x \neq x' \\ \mathsf{V}(\mathsf{pk}, \mathsf{h}, x, i, \Pi) = 1 \\ \mathsf{V}(\mathsf{pk}, \mathsf{h}, x', i, \Pi') = 1\end{array}\end{array}\right] \leq \mu(T(\lambda)) \ .$$

We say that the scheme is polynomially secure if it is $T(\cdot)$-secure for every polynomial T.

Theorem 2.5 ([17]–[19]): Assuming the LWE problem is $T(\cdot)$-hard, there exists a $T(\cdot)$-secure tree hash.

D. Somewhere Extractable Hash

In this section, we define somewhere extractable hash. The definition is a strengthening of the definition in [3] restricted to the case when the set of "binding coordinates" is of size 1. We explicitly define the hash to operate on long strings (rather than on bits as in [3]) to enable rate-1 constructions. A somewhere extractable hash consist of algorithms $(\mathsf{G}, \mathsf{T}, \mathsf{H}, \mathsf{V}, \mathsf{E})$ with the following syntax:

G**:** The randomized generation algorithm takes as input the security parameter λ. It outputs a public key pk.

T**:** The randomized trapdoor generation algorithm takes as input the security parameter λ and an index $i \in \mathbb{N}$. It outputs a public key pk and a secret key sk.

H**:** The deterministic hashing algorithm takes as input the public key pk and messages $x_1, \ldots, x_k \in \{0,1\}^\ell$. It outputs a hash h and openings $\Pi_1, \ldots, \Pi_k$.

V**:** The deterministic verification algorithm takes as input the public key pk, a hash h, a message $x \in \{0,1\}^\ell$, an index $i \in [k]$ and a proof Π. It outputs an acceptance bit.

E**:** The deterministic extraction algorithm takes as input the secret key sk and a hash h. It outputs a message $x \in \{0,1\}^\ell$.

Definition 2.6: A $T(\cdot)$-secure somewhere extractable hash satisfies the following requirements:

Completeness. For every $\lambda \in \mathbb{N}$, $k \leq 2^\lambda$, messages $x_1, \ldots, x_k \in \{0,1\}^\ell$ and index $i \in [k]$:

$$\Pr\left[\begin{array}{l}\mathsf{pk} \leftarrow \mathsf{G}(\lambda) \\ (\mathsf{h}, (\Pi_1, \ldots, \Pi_k)) \leftarrow \mathsf{H}(\mathsf{pk}, (x_1, \ldots, x_k)) \\ \quad : \ \mathsf{V}(\mathsf{pk}, \mathsf{h}, x_i, i, \Pi_i) = 1\end{array}\right] = 1 \ .$$

Efficiency. In the completeness experiment above:

- The generation, trapdoor generation and verification algorithms run in time $\ell \cdot \mathrm{poly}(\lambda)$.[6]
- The hashing and extraction algorithms run in polynomial time in their input length.
- We defined the additive overhead, $\alpha(\lambda, \ell)$, of the scheme as $|\mathsf{h}| - \ell$.

$T(\cdot)$-Key Indistinguishability. For every $\mathrm{poly}(T(\lambda))$-size adversary Adv there exists a negligible function μ such that for every $\lambda \in \mathbb{N}$ and index $i \leq 2^\lambda$:

$$\left| \begin{array}{l} \Pr\left[\mathsf{pk} \leftarrow \mathsf{G}(\lambda) \ : \ \mathsf{Adv}(\mathsf{pk}) = 1\right] \\ -\Pr\left[(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{T}(\lambda, i) \ : \ \mathsf{Adv}(\mathsf{pk}) = 1\right] \\ \quad \leq \mu(T(\lambda)) \ . \end{array} \right|$$

Extraction. For every $\lambda \in \mathbb{N}$, hash h, message $x \in \{0,1\}^\ell$, index $i \in [2^\lambda]$ and opening Π:

$$\Pr\left[\begin{array}{ll} & (\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{T}(\lambda, i) \\ & x^* \leftarrow \mathsf{E}(\mathsf{sk}, \mathsf{h}) \\ : & \mathsf{V}(\mathsf{pk}, \mathsf{h}, x, i, \Pi) = 1 \\ & x \neq x^* \end{array} \right] = 0 \ .$$

We say that the scheme is polynomially secure if it is $T(\cdot)$-secure for every polynomial T.

Under the hardness of the LWE problem, we can construct a rate-1 somewhere extractable hash by instantiating the construction of [16] based on rate-1 FHE [15].

Theorem 2.7: Assuming the LWE problem is $T(\cdot)$-hard, there exists a $T(\cdot)$-secure somewhere-extractable hash with additive overhead $\alpha(\lambda, \ell) = \frac{\ell}{\lambda} + \mathrm{poly}(\lambda)$.

Proof sketch: As shown in [15], assuming the LWE problem is $T(\cdot)$-hard, there exists an FHE scheme that enables encrypting a message of length $\ell = \mathrm{poly}(\lambda)$ (for sufficiently large ℓ) into a ciphertext of length $\ell + \frac{\ell}{\lambda}$. We next observe that the construction of somewhere extractable hash from [3], [16] the hash value is simply an FHE ciphertext encrypting one of the message hashed. If employing the rate-1 FHE from [15], the construction will satisfy the desired additive overhead requirement. ∎

E. Batch Arguments

In this section, we define non-interactive batch arguments for the index language. The definition is a strengthening of the definition in [3] (see discussion following Theorem 2.9).

A non-interactive batch argument for the index language consist of algorithms $(\mathsf{G}, \mathsf{T}, \mathsf{P}, \mathsf{V}, \mathsf{E})$ with the following syntax:

G: The randomized generation algorithm takes as input the security parameter λ and the witness length m. It outputs a public key pk.

T: The randomized trapdoor generation algorithm takes as input the security parameter λ, the witness length m and an index $i \in \mathbb{N}$. It outputs a public key pk and a secret key sk.

P: The deterministic prover algorithm takes as input the public key pk, a machine $\mathcal{M}$ and witnesses $w_1, \ldots, w_k \in \{0,1\}^m$. It outputs a proof Π.

V: The deterministic verifier algorithm takes as input the public key pk, a machine $\mathcal{M}$, and a proof Π. It outputs an acceptance bit.

E: The deterministic extraction algorithm takes as input the secret key sk and a proof Π. It outputs a witness $w \in \{0,1\}^m$.

If the generation and trapdoor generation algorithms do not require m, we say that the batch argument has *unbounded witness length* and omit m from the algorithms input.

Definition 2.8: A $T(\cdot)$-secure non-interactive batch argument for the index language satisfies the following requirements:

Completeness. For every $\lambda \in \mathbb{N}$, $k, m, t \leq 2^\lambda$, machine $\mathcal{M}$ and witnesses $w_1, \ldots, w_k \in \{0,1\}^m$ such that for every $i \in [k]$: $((i, w_i), t) \in \mathcal{U}_\mathcal{M}$ we have that:

$$\Pr\left[\begin{array}{l} \mathsf{pk} \leftarrow \mathsf{G}(\lambda, m) \\ \Pi \leftarrow \mathsf{P}(\mathsf{pk}, \mathcal{M}, (w_1, \ldots, w_k)) \\ : \ \mathsf{V}(\mathsf{pk}, \mathcal{M}, \Pi) = 1 \end{array} \right] = 1 \ .$$

Efficiency. In the completeness experiment above:

- The generation and trapdoor generation algorithms run in time $\mathrm{poly}(\lambda, m)$, or $\mathrm{poly}(\lambda)$ if the argument has unbounded witness length.
- The prover algorithm runs in time $|\mathcal{M}| \cdot \mathrm{poly}(\lambda, m, k, t)$ and outputs a proof of length $\mathrm{poly}(\lambda, m)$.
- The verification algorithm runs in time $|\mathcal{M}| \cdot \mathrm{poly}(\lambda, m)$.[7]
- The extraction algorithm runs in polynomial time in its input length.
- We defined the additive overhead, $\alpha(\lambda, \ell)$, of the scheme as $|\Pi| - m$.

$T(\cdot)$-Key Indistinguishability. For every $\mathrm{poly}(T(\lambda))$-size adversary Adv and polynomial $m(\lambda)$ there exists a negligible function μ such that for every $\lambda \in \mathbb{N}$ and index $i \leq 2^\lambda$:

$$\left| \begin{array}{l} \Pr\left[\mathsf{pk} \leftarrow \mathsf{G}(\lambda, m) \ : \ \mathsf{Adv}(\mathsf{pk}) = 1\right] \\ -\Pr\left[(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{T}(\lambda, m, i) \ : \ \mathsf{Adv}(\mathsf{pk}) = 1\right] \\ \quad \leq \mu(T(\lambda)) \ . \end{array} \right|$$

$T(\cdot)$-Somewhere Argument of Knowledge. For every $\mathrm{poly}(T(\lambda))$-size adversary Adv and polynomials $k(\lambda), m(\lambda), t(\lambda)$ there exists a negligible function μ such that for every $\lambda \in \mathbb{N}$ and index $i \in [k]$:

$$\Pr\left[\begin{array}{ll} & (\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{T}(\lambda, m, i) \\ & (\mathcal{M}, \Pi) \leftarrow \mathsf{Adv}(\mathsf{pk}) \\ & w \leftarrow \mathsf{E}(\mathsf{sk}, \Pi) \\ : & \mathsf{V}(\mathsf{pk}, \mathcal{M}, \Pi) = 1 \\ & ((i, w), t) \notin \mathcal{U}_\mathcal{M} \end{array} \right] \leq \mu(T(\lambda)) \ .$$

[6] The definition only bounds the verification time of honestly generated hash and opening. We can assume WLOG that the same bound holds also for maliciously generated hash and opening by capping the verifier execution time.

[7] The definition only bounds the verification time of honestly generated proofs. We can assume WLOG that the same bound holds also for maliciously generated proofs by capping the verifier execution time.

We say that the scheme is polynomially secure if it is $T(\cdot)$-secure for every polynomial T.

Theorem 2.9 ([3]): Assuming the LWE problem is $T(\cdot)$-hard, there exist a $T(\cdot)$-secure non-interactive batch argument for the index language.

We remark that the notion of batch arguments defined in [3] is slightly different than the notion in Definition 2.8. We explain how to modify the [3] construction to obtain our notion:

- In [3], the prover and verifier are given a circuit implementing the NP verification procedure and the setup algorithm is given the size of this circuit. In our notion, the NP verification procedure is given by a Turing machine and the setup algorithm is not given a bound on the size or running time of the machine but only the witness length m. As discussed in [3, Section 6], this can be realized by combining the notion of [3] with a RAM delegation scheme satisfying Definition 2.2. Such a RAM delegation scheme is also known under the LWE assumption (see Theorem 2.3).
- In [3], the setup algorithm is given the number of statements k while in our notion the the number of statements is not fixed ahead of time. We can realize this by using λ copies the scheme of [3]: for every $i \in [\lambda]$, we generate a public key for 2^i statements. To prove k statements, divide the statements into at most λ groups, each of size that is a power of two. The final proof contains, for each group, a proof under one of the public keys. Here we use the fact that in the batch arguments of [3], the setup time is poly-logarithmic in the number of statements.
- While [3] only argue polynomial security of their scheme, it is straightforward to extend their proof to show $T(\cdot)$-security under the $T(\cdot)$-hardness of LWE.

III. Rate-1 Batch Arguments

In this section, we prove Theorem 3.1 giving batch arguments with unbounded witness length and small additive overhead.

Theorem 3.1: Assume the existence of:

- A $T(\cdot)$-secure non-interactive batch argument for the index language.
- A $T(\cdot)$-secure somewhere extractable hash with additive overhead $\alpha(\lambda, \ell) = \frac{\ell}{\lambda} + \mathrm{poly}(\lambda)$.
- A $T(\cdot)$-secure non-interactive delegation scheme for RAM.

Then there exist a $T(\cdot)$-secure non-interactive batch argument for the index language with unbounded witness length and with additive overhead $\alpha(\lambda, m) = \frac{3m}{\lambda} + \mathrm{poly}(\lambda)$.

Construction. We construct a $T(\cdot)$-secure, non-interactive batch argument $(\mathsf{G}'_{\mathsf{BA}}, \mathsf{T}'_{\mathsf{BA}}, \mathsf{P}'_{\mathsf{BA}}, \mathsf{V}'_{\mathsf{BA}}, \mathsf{E}'_{\mathsf{BA}})$ with additive overhead $\frac{3m}{\lambda} + \mathrm{poly}(\lambda)$ using the following building blocks:

- A $T(\cdot)$-secure non-interactive batch argument for the index language $(\mathsf{G}_{\mathsf{BA}}, \mathsf{T}_{\mathsf{BA}}, \mathsf{P}_{\mathsf{BA}}, \mathsf{V}_{\mathsf{BA}}, \mathsf{E}_{\mathsf{BA}})$.
- A $T(\cdot)$-secure somewhere extractable hash $(\mathsf{G}_{\mathsf{EH}}, \mathsf{T}_{\mathsf{EH}}, \mathsf{H}_{\mathsf{EH}}, \mathsf{V}_{\mathsf{EH}}, \mathsf{E}_{\mathsf{EH}})$ with additive overhead $\frac{\ell}{\lambda} + Q_1(\lambda)$ for some polynomial Q_1.
- A $T(\cdot)$-secure non-interactive delegation scheme for RAM $(\mathsf{G}_{\mathsf{DL}}, \mathsf{H}_{\mathsf{DL}}, \mathsf{P}_{\mathsf{DL}}, \mathsf{V}_{\mathsf{DL}})$.
- A $T(\cdot)$-secure tree hash $(\mathsf{G}_{\mathsf{TH}}, \mathsf{H}_{\mathsf{TH}}, \mathsf{V}_{\mathsf{TH}})$ with hash of length $Q_2(\lambda)$ for some polynomial Q_2.

We are given an NP verification machine $\mathcal{M}$ and a sequence of witnesses $(w^1, \ldots, w^k)$, each of length m. Let t be the running time of $\mathcal{M}$ on each witness. We first split each witness w^i into n blocks $w^i = (w^i_1, \ldots, w^i_n)$ where each block is of length $\ell = m/n$. For every $i \in [k]$, we define a computation checking that $((i, w^i), t) \in \mathcal{U}_{\mathcal{M}}$. The computation is made of a sequence of n intervals implemented by RAM machines $\mathcal{R}^i_1, \ldots, \mathcal{R}^i_n$. We run the machines sequentially on the same memory, that is, each machine starts from the final configuration of the previous one. For every $j \in [n]$, the machine $\mathcal{R}^i_j$ has the witness block w^i_j hard-coded. For $j < n$ the machine $\mathcal{R}^i_j$ writes w^i_j to memory and terminates. The final machine $\mathcal{R}^i_n$ writes the final witness block w^i_n to memory and then, once the entire witness w^i is in memory, it emulates $\mathcal{M}(i, w^i)$ for t steps and accepts if and only if $\mathcal{M}$ accepts. Let cf^i_0 be the starting configuration of $\mathcal{R}^i_1$ and for $j \in [n]$, let cf^i_j be the final configuration of $\mathcal{R}^i_j$. Using a RAM delegation scheme we compute the digest h^i_j of the configuration cf^i_j and generate a proof Π^i_j that the machine $\mathcal{R}^i_j$ indeed transitions from cf^i_{j-1} to cf^i_j. Therefore, given the witness w^i and the digests $\{\mathsf{h}^i_j\}_j$ we can verify that $((i, w^i), t) \in \mathcal{U}_{\mathcal{M}}$ by checking that h^i_0 and h^i_n indeed are the digests of the correct starting and accepting configurations respectively, and that all the proofs $\{\Pi^i_j\}_{i,j}$ are accepting.

The next step is to prove that all proofs are accepting using a single batch argument. To this end, we use the witness blocks and the digests to define a collection of NP statements and hash them down. In more details, we first use a somewhere extractable hash to compute for every $j \in [n]$, the hash h_j of the k pairs $\{w^i_j, \mathsf{h}^i_j)\}_i$. Then, we further hash the n hash values $\{\mathsf{h}_j\}_j$ to a single hash h using a tree hash. We can now define the NP verification machine $\mathcal{M}'$ for which we provide the batch argument. The machine $\mathcal{M}'$ given index (i, j) accepts a witness that contains (a) valid openings of h to h_{j-1} and h_j, (b) valid openings of h_{j-1} and h_j to $(w^i_j, \mathsf{h}^i_j))$ and $(w^i_j, \mathsf{h}^i_j))$ respectively, and (c) an accepting proof Π^i_j for the RAM machine $\mathcal{R}^i_j$ (recall that the machine $\mathcal{R}^i_j$ can be computed from w^i_j) and the digests $(\mathsf{h}^i_{j-1}, \mathsf{h}^i_j)$. The final proof consist of the hash values $\{\mathsf{h}_j\}_j$ and the batch argument proof for $\mathcal{M}'$.

We proceed with a formal description of the construction. *The machine $\mathcal{R}^i_j$.* For $i \in [k]$, $j \in [n]$, a string $x \in \{0,1\}^\ell$ and a machine $\mathcal{M}$, let $\mathcal{R}^i_j[x, \mathcal{M}]$ be the following RAM machine:

- Writes x to memory starting at location $(j-1) \cdot \ell + 1$.
- If $j = n$:
  - Emulate $\mathcal{M}$ on input (i, w^i) where w^i is the given in the first $m = n \cdot \ell$ memory locations.
  - Empty the memory and accept if and only if $\mathcal{M}$ accepts.

Let $t^i_j = (\ell + |\mathcal{M}| + t) \cdot \mathrm{poly}(\lambda)$ be the running time of the

machine $\mathcal{R}_j^i[x, \mathcal{M}]$. Let $\mathsf{cf}_{\mathsf{start}}^i$ be the starting configuration of $\mathcal{R}_1^i[x, \mathcal{M}]$ and let $\mathsf{cf}_{\mathsf{accept}}^i$ be the accepting configuration of $\mathcal{R}_n^i$ with empty memory.

The machine $\mathcal{M}'$. Given a public key $\mathsf{pk} = (\mathsf{pk}_{\mathsf{TH}}, \mathsf{pk}_{\mathsf{EH}}, \mathsf{pk}_{\mathsf{DL}})$, a machine $\mathcal{M}$ and hash value h, let $\mathcal{M}'[\mathsf{pk}, \mathcal{M}, \mathsf{h}]$ be the machine on input an index i' and a witness w':

$$w' = ((x^b, \mathsf{h}_{\mathsf{DL}}^b, \Pi_{\mathsf{EH}}^b, \mathsf{h}_{\mathsf{EH}}^b, \Pi_{\mathsf{TH}}^b)_{b\in\{0,1\}} \ , \ \Pi_{\mathsf{DL}}) \ ,$$

proceeds as follows:

- Let $i \in [k]$, $j \in [n]$ be such that $i' = (i-1)\cdot n + j$.
- For $b \in \{0,1\}$, check that $\mathsf{V}_{\mathsf{TH}}(\mathsf{pk}_{\mathsf{TH}}, \mathsf{h}, \mathsf{h}_{\mathsf{EH}}^b, j - b, \Pi_{\mathsf{TH}}^b) = 1$.
- For $b \in \{0,1\}$, check that $\mathsf{V}_{\mathsf{EH}}(\mathsf{pk}_{\mathsf{EH}}, \mathsf{h}_{\mathsf{EH}}^b, (x^b, \mathsf{h}_{\mathsf{DL}}^b), i, \Pi_{\mathsf{EH}}^b) = 1$.
- Check that $\mathsf{V}_{\mathsf{DL}}(\mathsf{pk}_{\mathsf{DL}}, \mathcal{R}_j^i[x_0, \mathcal{M}], \mathsf{h}_{\mathsf{DL}}^1, \mathsf{h}_{\mathsf{DL}}^0, t_j^i, \Pi_{\mathsf{DL}}) = 1$.
- If $j = 1$ check that $\mathsf{h}_{\mathsf{DL}}^1 = \mathsf{H}_{\mathsf{DL}}(\mathsf{pk}_{\mathsf{DL}}, \mathsf{cf}_{\mathsf{start}}^i)$.
- If $j = n$ check that $\mathsf{h}_{\mathsf{DL}}^0 = \mathsf{H}_{\mathsf{DL}}(\mathsf{pk}_{\mathsf{DL}}, \mathsf{cf}_{\mathsf{accept}}^i)$.
- Accept if and only if all checks pass.

Let $m' = \ell \cdot \mathrm{poly}(\lambda)$ be the length of the witness w'.

The batch argument algorithms.

$\mathsf{G}'_{\mathsf{BA}}$: Given as input the security parameter λ the generation algorithm is as follows:
- Set $\ell = \lambda \cdot (Q_1(\lambda) + Q_2(\lambda))$.
- Set $\mathsf{pk}_{\mathsf{BA}} \leftarrow \mathsf{G}_{\mathsf{BA}}(\lambda, m')$.
- Set $\mathsf{pk}_{\mathsf{TH}} \leftarrow \mathsf{G}_{\mathsf{TH}}(\lambda)$.
- Set $\mathsf{pk}_{\mathsf{EH}} \leftarrow \mathsf{G}_{\mathsf{EH}}(\lambda)$.
- Set $\mathsf{pk}_{\mathsf{DL}} \leftarrow \mathsf{G}_{\mathsf{DL}}(\lambda)$.
- Output $\mathsf{pk} = (\mathsf{pk}_{\mathsf{BA}}, \mathsf{pk}_{\mathsf{TH}}, \mathsf{pk}_{\mathsf{EH}}, \mathsf{pk}_{\mathsf{DL}})$.

$\mathsf{T}'_{\mathsf{BA}}$: Given as input the security parameter λ and an index i, the trapdoor generation algorithm is as follows:
- Set $\ell = \lambda \cdot (Q_1(\lambda) + Q_2(\lambda))$.
- Set $\mathsf{pk}_{\mathsf{BA}} \leftarrow \mathsf{G}_{\mathsf{BA}}(\lambda, m')$.
- Set $\mathsf{pk}_{\mathsf{TH}} \leftarrow \mathsf{G}_{\mathsf{TH}}(\lambda)$.
- Set $(\mathsf{pk}_{\mathsf{EH}}, \mathsf{sk}) \leftarrow \mathsf{T}_{\mathsf{EH}}(\lambda, i)$.
- Set $\mathsf{pk}_{\mathsf{DL}} \leftarrow \mathsf{G}_{\mathsf{DL}}(\lambda)$.
- Output $(\mathsf{pk} = (\mathsf{pk}_{\mathsf{BA}}, \mathsf{pk}_{\mathsf{TH}}, \mathsf{pk}_{\mathsf{EH}}, \mathsf{pk}_{\mathsf{DL}}) \ , \ \mathsf{sk})$.

$\mathsf{P}'_{\mathsf{BA}}$: Given as input the public key pk, a machine $\mathcal{M}$ and witnesses $w_1, \ldots, w_k \in \{0,1\}^m$, the prover algorithm is as follows:
- Let $\mathsf{pk} = (\mathsf{pk}_{\mathsf{BA}}, \mathsf{pk}_{\mathsf{TH}}, \mathsf{pk}_{\mathsf{EH}}, \mathsf{pk}_{\mathsf{DL}})$.
- Set $n = \frac{m}{\ell}$.
- For $i \in [k]$ let $w^i = (w_1^i, \ldots, w_n^i)$ where each w_j^i is of length ℓ.
- For $i \in [k]$ set $w_0^i = 0^\ell$.
- For $i \in [k]$ set $\mathsf{cf}_0^i = \mathsf{cf}_{\mathsf{start}}^i$.
- For $i \in [k]$ and $j \in [n]$ compute the final configuration cf_j^i of $\mathcal{R}_j^i[w_j^i, \mathcal{M}]$ when starting from configuration cf_{j-1}^i.
- For $i \in [k]$ and $j \in [n]$ set $\tilde{\Pi}_j^i \leftarrow \mathsf{P}(\mathsf{pk}_{\mathsf{DL}}, \mathcal{R}, \mathsf{cf}_{j-1}^i, \mathsf{cf}_j^i, t_j^i)$.
- For $i \in [k]$ and $j \in [0,n]$ set $\mathsf{h}_j^i \leftarrow \mathsf{H}_{\mathsf{DL}}(\mathsf{pk}_{\mathsf{DL}}, \mathsf{cf}_j^i)$.
- For $j \in [0,n]$ set $(\mathsf{h}_j, (\Pi_j^1, \ldots, \Pi_j^k)) \leftarrow \mathsf{H}_{\mathsf{EH}}(\mathsf{pk}_{\mathsf{EH}}, ((w_j^1, \mathsf{h}_j^1), \ldots, (w_j^k, \mathsf{h}_j^k)))$.
- Set $(\mathsf{h}, (\Pi_0, \ldots, \Pi_n)) \leftarrow \mathsf{H}_{\mathsf{TH}}(\mathsf{pk}_{\mathsf{EH}}, (\mathsf{h}_0, \ldots, \mathsf{h}_n))$.
- For $i \in [k]$ and $j \in [n]$ set $w'_{(i-1)\cdot n+j} = ((w_{j-b}, \mathsf{h}_{j-b}^i, \Pi_{j-b}^i, \mathsf{h}_{j-b}, \Pi_{j-b})_{b\in\{0,1\}} \ , \ \tilde{\Pi}_j^i)$.
- Set

$$\Pi_{\mathsf{BA}} \leftarrow \mathsf{P}_{\mathsf{BA}} \left(\begin{array}{l} \mathsf{pk}_{\mathsf{BA}}, \\ \mathcal{M}'[(\mathsf{pk}_{\mathsf{TH}}, \mathsf{pk}_{\mathsf{EH}}, \mathsf{pk}_{\mathsf{DL}}), \mathcal{M}, \mathsf{h}], \\ (w'_1, \ldots, w'_{k\cdot n}) \end{array} \right) .$$

- Output $\Pi = ((\mathsf{h}_0, \ldots, \mathsf{h}_n) \ , \ \Pi_{\mathsf{BA}})$.

$\mathsf{V}'_{\mathsf{BA}}$: Given as input the public key pk, a machine $\mathcal{M}$ and a proof Π, the verifier algorithm is as follows:
- Let $\mathsf{pk} = (\mathsf{pk}_{\mathsf{BA}}, \mathsf{pk}_{\mathsf{TH}}, \mathsf{pk}_{\mathsf{EH}}, \mathsf{pk}_{\mathsf{DL}})$.
- Let $\Pi = ((\mathsf{h}_0, \ldots, \mathsf{h}_n) \ , \ \Pi_{\mathsf{BA}})$.
- Set $(\mathsf{h}, (\Pi_0, \ldots, \Pi_n)) \leftarrow \mathsf{H}_{\mathsf{EH}}(\mathsf{pk}_{\mathsf{EH}}, (\mathsf{h}_0, \ldots, \mathsf{h}_n))$.
- Output the same as $\mathsf{V}_{\mathsf{BA}}(\mathsf{pk}_{\mathsf{BA}}, \mathcal{M}'[(\mathsf{pk}_{\mathsf{TH}}, \mathsf{pk}_{\mathsf{EH}}, \mathsf{pk}_{\mathsf{DL}}), \mathcal{M}, \mathsf{h}], \Pi_{\mathsf{BA}})$.

$\mathsf{E}'_{\mathsf{BA}}$: Given as input the secret key sk and a proof Π, the extraction algorithm is as follows:
- Let $\Pi = ((\mathsf{h}_0, \ldots, \mathsf{h}_n) \ , \ \Pi_{\mathsf{BA}})$.
- For $j \in [n]$, set $(w_j, \mathsf{h}'_j) \leftarrow \mathsf{E}_{\mathsf{EH}}(\mathsf{sk}, \mathsf{h}_j)$
- Output $w = (w_1, \ldots, w_n)$.

The analysis of the construction appears in the full version of this work.

IV. Mergeable Delegation

A mergeable delegation scheme for RAM is a non-interactive delegation scheme for RAM that is equipped with an additional merging algorithm M. Given the proofs for two statements $(\mathsf{cf}_1, \mathsf{cf}_2, t_1), (\mathsf{cf}_2, \mathsf{cf}_3, t_2) \in \mathcal{T}_{\mathcal{R}}$ the merging algorithm efficiently generates a proof for the combined statement $(\mathsf{cf}_1, \mathsf{cf}_3, t_1 + t_2) \in \mathcal{T}_{\mathcal{R}}$. What makes this notion non-trivial is the *compactness* requirement which states that the the merged proof is not much longer than (the longer one of) the original proofs. In essence, merging compresses two proofs into a single proof of roughly the same size.

Our notion of mergeable delegation allows proofs to be merged recursively. In more detail, for each proof we maintain a level, where newly generated proofs are of level 0 and merging two proofs of level i results in a proof of level $i+1$. Similarly to the notion of levelled-FHE, We restrict attention to schemes that support an a-priori bounded number of levels, say λ.

Formally, a mergeable non-interactive delegation scheme for RAM is a non-interactive delegation scheme for RAM $(\mathsf{G}, \mathsf{H}, \mathsf{P}, \mathsf{V})$ augmented with algorithms (M, L) with the following syntax:

M**:** The deterministic merging algorithm takes as input a public key pk, a RAM machine $\mathcal{R}$ and a pair of hashed statements and proofs $((\mathsf{h}_i, \mathsf{h}'_i, t_i), \Pi_i)_{i\in[2]}$. It outputs a new proof Π.

L**:** The deterministic level algorithm takes as input a proof Π, it outputs a level $\ell \in \mathbb{N}$.

Definition 4.1: A $T(\cdot)$-secure non-interactive delegation scheme for RAM is mergeable if it satisfies the following requirements:

Completeness with Level 0. For every $\lambda \in \mathbb{N}$, $t \leq 2^\lambda$, RAM machine $\mathcal{R}$ and statement $(\mathsf{cf}, \mathsf{cf}', t) \in \mathcal{T}_\mathcal{R}$ we have that:

$$\Pr\left[\begin{array}{l} \mathsf{pk} \leftarrow \mathsf{G}(\lambda) \\ \mathsf{h} \leftarrow \mathsf{H}(\mathsf{pk}, \mathsf{cf}) \\ \mathsf{h}' \leftarrow \mathsf{H}(\mathsf{pk}, \mathsf{cf}') \\ \Pi \leftarrow \mathsf{P}(\mathsf{pk}, \mathcal{R}, (\mathsf{cf}, \mathsf{cf}', t)) \\ : \quad \mathsf{V}(\mathsf{pk}, \mathcal{R}, (\mathsf{h}, \mathsf{h}', t), \Pi) = 1 \\ \quad\ \ \mathsf{L}(\Pi) = 0 \end{array}\right] = 1 \ .$$

Completeness of Merge. For every $\lambda \in \mathbb{N}$, public key pk, $\ell < \lambda$, RAM machine $\mathcal{R}$ and a pair of hashed statements and proofs $(x_i = (\mathsf{h}_i, \mathsf{h}'_i, t_i), \Pi_i)_{i\in[2]}$ such that $\mathsf{h}'_1 = \mathsf{h}_2$ and for every $i \in [2]$, $\mathsf{V}(\mathsf{pk}, \mathcal{R}, x_i, \Pi_i) = 1$ and $\mathsf{L}(\Pi_i) = \ell$ we have that:

$$\Pr\left[\begin{array}{l} \Pi \leftarrow \mathsf{M}(\mathsf{pk}, \mathcal{R}, (x_i, \Pi_i)_{i\in[2]}) \\ : \quad \mathsf{V}(\mathsf{pk}, \mathcal{R}, (\mathsf{h}_1, \mathsf{h}'_2, t_1 + t_2), \Pi) = 1 \\ \quad\ \ \mathsf{L}(\Pi) = \ell + 1 \end{array}\right] = 1 \ .$$

Efficiency. In the completeness experiments above:

- The merging algorithm runs in time $|\mathcal{R}| \cdot \mathrm{poly}(\lambda)$ and outputs a proof of length $\mathrm{poly}(\lambda)$.
- The level algorithm runs in time $\mathrm{poly}(\lambda)$.

$T(\cdot)$**-Soundness for Bounded Level Proofs.** For every $\mathrm{poly}(T(\lambda))$-size adversary Adv there exists a negligible function μ such that for every $\lambda \in \mathbb{N}$:

$$\Pr\left[\begin{array}{l} \mathsf{pk} \leftarrow \mathsf{G}(\lambda) \\ (\mathcal{R}, \mathsf{cf}, \mathsf{cf}', \mathsf{h}, \mathsf{h}', t, \Pi) \leftarrow \mathsf{Adv}(\mathsf{pk}) \\ \quad t \leq T(\lambda) \\ \quad \mathsf{L}(\Pi) \leq \log T(\lambda) \\ : \quad \mathsf{V}(\mathsf{pk}, \mathcal{R}, (\mathsf{h}, \mathsf{h}', t), \Pi) = 1 \\ \quad (\mathsf{cf}, \mathsf{cf}', t) \in \mathcal{T}_\mathcal{R} \\ \quad \mathsf{h} = \mathsf{H}(\mathsf{pk}, \mathsf{cf}) \\ \quad \mathsf{h}' \neq \mathsf{H}(\mathsf{pk}, \mathsf{cf}') \end{array}\right] \leq \mu(T(\lambda)) \ .$$

We say that the scheme is polynomially secure if it is $T(\cdot)$-secure for every polynomial T.

We make some remarks on Definition 4.1:

- The completeness with level 0 requirement is exactly the same as the plain completeness requirement of delegation except that we require that freshly created proofs are of level 0.
- The soundness for bounded levels requirement is exactly the same as the plain soundness requirement except that we restrict the level of the proof to be $O(\log T(\lambda))$.
- We only require completeness of merge for a pair of proofs at the same level and up to level λ. Therefore, constructing a proof of level ℓ requires merging 2^ℓ proofs in a full binary tree (some of these proof may be for 0 computation steps).

Theorem 4.2: Assume the existence of:

- A $T(\cdot)$-secure non-interactive batch arguments for the index language with unbounded witness length and additive overhead $\alpha(\lambda, m) = \frac{O(m)}{\lambda} + \mathrm{poly}(\lambda)$.
- A $T(\cdot)$-secure non-interactive delegation scheme for RAM.

Then there exists a $T(\cdot)$-secure mergeable non-interactive delegation scheme for RAM.

Construction. We construct a mergeable non-interactive delegation scheme $(\mathsf{G}'_{\mathsf{DL}}, \mathsf{H}'_{\mathsf{DL}}, \mathsf{P}'_{\mathsf{DL}}, \mathsf{V}'_{\mathsf{DL}}, \mathsf{M}'_{\mathsf{DL}}, \mathsf{L}'_{\mathsf{DL}})$ using the following building blocks:

- A $T(\cdot)$-secure non-interactive batch argument for the index language $(\mathsf{G}_{\mathsf{BA}}, \mathsf{T}_{\mathsf{BA}}, \mathsf{P}_{\mathsf{BA}}, \mathsf{V}_{\mathsf{BA}}, \mathsf{E}_{\mathsf{BA}})$ with unbounded witness length and additive overhead $\alpha(\lambda, m) = \frac{O(m)}{\lambda} + \mathrm{poly}(\lambda)$.
- A $T(\cdot)$-secure non-interactive delegation scheme for RAM $(\mathsf{G}_{\mathsf{DL}}, \mathsf{H}_{\mathsf{DL}}, \mathsf{P}_{\mathsf{DL}}, \mathsf{V}_{\mathsf{DL}})$.

We proceed with a formal description of the construction. We start by defining a verification procedure for proofs of each level. The level-0 verification algorithm V^0 is simply the verifier V_{DL} of the underlying delegation scheme. For $\ell \in [\lambda]$, the level-ℓ verification algorithm V^ℓ is given as input the public key $\mathsf{pk} = (\mathsf{pk}_0, \ldots, \mathsf{pk}_\ell)$, a RAM machine $\mathcal{R}$, a hashed statement $x = (\mathsf{h}, \mathsf{h}', t)$ and a proof Π. It proceeds as follows:

- Let $\Pi = (x_1 = (\mathsf{h}_1, \mathsf{h}'_1, t_1), x_2 = (\mathsf{h}_2, \mathsf{h}'_2, t_2), \Pi')$.
- If $\mathsf{h}'_1 \neq \mathsf{h}_2$ or $(\mathsf{h}_1, \mathsf{h}'_2, t_1 + t_2) \neq (\mathsf{h}, \mathsf{h}', t)$ then reject.
- Let $\mathcal{M}^\ell = \mathcal{M}^\ell[\mathsf{pk}, \mathcal{R}, (x_1, x_2)]$ be the machine that on $(i, \tilde{\Pi})$ outputs $\mathsf{V}^{\ell-1}((\mathsf{pk}_0, \ldots, \mathsf{pk}_{\ell-1}), \mathcal{R}, x_i, \tilde{\Pi})$.
- Output $\mathsf{V}_{\mathsf{BA}}(\mathsf{pk}_\ell, \mathcal{M}^\ell, \Pi')$

The delegation scheme algorithms.

$\mathsf{G}'_{\mathsf{DL}}$: Given as input a security parameter $\lambda \in \mathbb{N}$, the setup algorithm is as follows:
- Set $\mathsf{pk}_0 \leftarrow \mathsf{G}_{\mathsf{DL}}(\lambda)$.
- For $\ell \in [\lambda]$, set $\mathsf{pk}_i \leftarrow \mathsf{G}_{\mathsf{BA}}(\lambda)$.
- Output $\mathsf{pk} = (\mathsf{pk}_0, \ldots, \mathsf{pk}_\lambda)$.

$\mathsf{H}'_{\mathsf{DL}}$: Given as input the public key $\mathsf{pk} = (\mathsf{pk}_0, \ldots, \mathsf{pk}_\lambda)$ and a configuration $\mathsf{cf} \in \{0,1\}^*$, the digest algorithm outputs $\mathsf{H}_{\mathsf{DL}}(\mathsf{pk}_0, \mathsf{cf})$.

$\mathsf{P}'_{\mathsf{DL}}$: Given as input the public key pk, a RAM machine $\mathcal{R}$, and a statement $(\mathsf{cf}, \mathsf{cf}', t)$ the prover algorithm is as follows:
- Let $\mathsf{pk} = (\mathsf{pk}_0, \ldots, \mathsf{pk}_\lambda)$.
- Set $\Pi \leftarrow \mathsf{P}_{\mathsf{DL}}(\mathsf{pk}_0, \mathcal{R}, (\mathsf{cf}, \mathsf{cf}', t))$.
- Output $(0, \Pi)$

$\mathsf{V}'_{\mathsf{DL}}$: Given as input the public key $\mathsf{pk} = (\mathsf{pk}_0, \ldots, \mathsf{pk}_\lambda)$, a RAM machine $\mathcal{R}$, a hashed statement x and a proof $\Pi = (\ell, \Pi')$ the verifier algorithm outputs $\mathsf{V}^\ell((\mathsf{pk}_0, \ldots, \mathsf{pk}_\ell), \mathcal{R}, x, \Pi')$.

$\mathsf{M}'_{\mathsf{DL}}$: Given as input a public key pk, a RAM machine $\mathcal{R}$ and a pair of hashed statements and proofs $(x_i = (\mathsf{h}_i, \mathsf{h}'_i, t_i), \Pi_i)_{i\in[2]}$, the merging algorithm is as follows:
- Let $\mathsf{pk} = (\mathsf{pk}_0, \ldots, \mathsf{pk}_\lambda)$.
- For $i \in [2]$, let $\Pi_i = (\ell_i, \Pi'_i)$.
- If $\ell_1 \neq \ell_2$ then reject.

- Set $\ell = \ell_1 + 1$
- Let $\mathcal{M}^{\ell} = \mathcal{M}^{\ell}[\mathsf{pk}, \mathcal{R}, (x_1, x_2)]$ be the machine that on $(i, \tilde{\Pi})$ outputs $\mathsf{V}^{\ell-1}((\mathsf{pk}_0, \ldots, \mathsf{pk}_{\ell-1}), \mathcal{R}, x_i, \tilde{\Pi})$.
- Set $\Pi \leftarrow \mathsf{P}_{\mathsf{BA}}(\mathsf{pk}_{\ell}, \mathcal{M}^{\ell}, (\Pi'_1, \Pi'_2))$
- Output $(\ell, (x_1, x_2, \Pi))$

$\mathsf{L}'_{\mathsf{DL}}$: Given as input a proof $\Pi = (\ell, \Pi')$, the level algorithm outputs ℓ.

The analysis of the construction appears in the full version of this work.

REFERENCES

[1] P. Valiant, "Incrementally verifiable computation or proofs of knowledge imply time/space efficiency," in *TCC*, 2008, pp. 1–18.

[2] Y. T. Kalai, O. Paneth, and L. Yang, "How to delegate computations publicly," in *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, M. Charikar and E. Cohen, Eds. ACM, 2019, pp. 1115–1124. [Online]. Available: https://doi.org/10.1145/3313276.3316411

[3] A. R. Choudhuri, A. Jain, and Z. Jin, "Snargs for *P* from LWE," *IACR Cryptol. ePrint Arch.*, p. 808, 2021. [Online]. Available: https://eprint.iacr.org/2021/808

[4] B. Waters and D. J. Wu, "Batch arguments for NP and more from standard bilinear group assumptions," *IACR Cryptol. ePrint Arch.*, p. 336, 2022. [Online]. Available: https://eprint.iacr.org/2022/336

[5] N. Bitansky and A. Chiesa, "Succinct arguments from multi-prover interactive proofs and their efficiency benefits," in *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, ser. Lecture Notes in Computer Science, R. Safavi-Naini and R. Canetti, Eds., vol. 7417. Springer, 2012, pp. 255–272. [Online]. Available: https://doi.org/10.1007/978-3-642-32009-5_16

[6] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer, "Recursive composition and bootstrapping for SNARKS and proof-carrying data," in *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, D. Boneh, T. Roughgarden, and J. Feigenbaum, Eds. ACM, 2013, pp. 111–120. [Online]. Available: https://doi.org/10.1145/2488608.2488623

[7] A. R. Block, J. Holmgren, A. Rosen, R. D. Rothblum, and P. Soni, "Time- and space-efficient arguments from groups of unknown order," in *Advances in Cryptology - CRYPTO 2021 - 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16-20, 2021, Proceedings, Part IV*, ser. Lecture Notes in Computer Science, T. Malkin and C. Peikert, Eds., vol. 12828. Springer, 2021, pp. 123–152. [Online]. Available: https://doi.org/10.1007/978-3-030-84259-8_5

[8] J. Holmgren and R. Rothblum, "Delegating computations with (almost) minimal time and space overhead," in *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018, Paris, France, October 7-9, 2018*, M. Thorup, Ed. IEEE Computer Society, 2018, pp. 124–135. [Online]. Available: https://doi.org/10.1109/FOCS.2018.00021

[9] N. Bitansky, R. Canetti, O. Paneth, and A. Rosen, "On the existence of extractable one-way functions," in *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, D. B. Shmoys, Ed. ACM, 2014, pp. 505–514. [Online]. Available: https://doi.org/10.1145/2591796.2591859

[10] E. Boyle and R. Pass, "Limits of extractability assumptions with distributional auxiliary input," in *Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part II*, ser. Lecture Notes in Computer Science, T. Iwata and J. H. Cheon, Eds., vol. 9453. Springer, 2015, pp. 236–261. [Online]. Available: https://doi.org/10.1007/978-3-662-48800-3_10

[11] B. Bünz, A. Chiesa, P. Mishra, and N. 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*, ser. Lecture Notes in Computer Science, R. Pass and K. Pietrzak, Eds., vol. 12551. Springer, 2020, pp. 1–18. [Online]. Available: https://doi.org/10.1007/978-3-030-64378-2_1

[12] B. Bünz, A. Chiesa, W. Lin, P. Mishra, and N. 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*, ser. Lecture Notes in Computer Science, T. Malkin and C. Peikert, Eds., vol. 12825. Springer, 2021, pp. 681–710. [Online]. Available: https://doi.org/10.1007/978-3-030-84242-0_24

[13] D. Boneh, J. Drake, B. Fisch, and A. Gabizon, "Halo infinite: Proof-carrying data from additive polynomial commitments," in *Advances in Cryptology - CRYPTO 2021 - 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16-20, 2021, Proceedings, Part I*, ser. Lecture Notes in Computer Science, T. Malkin and C. Peikert, Eds., vol. 12825. Springer, 2021, pp. 649–680. [Online]. Available: https://doi.org/10.1007/978-3-030-84242-0_23

[14] Z. Brakerski, J. Holmgren, and Y. T. Kalai, "Non-interactive delegation and batch NP verification from standard computational assumptions," in *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, H. Hatami, P. McKenzie, and V. King, Eds. ACM, 2017, pp. 474–482. [Online]. Available: https://doi.org/10.1145/3055399.3055497

[15] Z. Brakerski, N. Döttling, S. Garg, and G. Malavolta, "Leveraging linear decryption: Rate-1 fully-homomorphic encryption and time-lock puzzles," in *Theory of Cryptography - 17th International Conference, TCC 2019, Nuremberg, Germany, December 1-5, 2019, Proceedings, Part II*, ser. Lecture Notes in Computer Science, D. Hofheinz and A. Rosen, Eds., vol. 11892. Springer, 2019, pp. 407–437. [Online]. Available: https://doi.org/10.1007/978-3-030-36033-7_16

[16] P. Hubácek and D. Wichs, "On the communication complexity of secure function evaluation with long output," in *Proceedings of the 2015 Conference on Innovations in Theoretical Computer Science, ITCS 2015, Rehovot, Israel, January 11-13, 2015*, T. Roughgarden, Ed. ACM, 2015, pp. 163–172. [Online]. Available: https://doi.org/10.1145/2688073.2688105

[17] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings*, ser. Lecture Notes in Computer Science, C. Pomerance, Ed., vol. 293. Springer, 1987, pp. 369–378. [Online]. Available: https://doi.org/10.1007/3-540-48184-2_32

[18] M. Ajtai, "Generating hard instances of lattice problems (extended abstract)," in *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, G. L. Miller, Ed. ACM, 1996, pp. 99–108. [Online]. Available: https://doi.org/10.1145/237814.237838

[19] O. Goldreich, S. Goldwasser, and S. Halevi, "Collision-free hashing from lattice problems," in *Studies in Complexity and Cryptography. Miscellanea on the Interplay between Randomness and Computation - In Collaboration with Lidor Avigad, Mihir Bellare, Zvika Brakerski, Shafi Goldwasser, Shai Halevi, Tali Kaufman, Leonid Levin, Noam Nisan, Dana Ron, Madhu Sudan, Luca Trevisan, Salil Vadhan, Avi Wigderson, David Zuckerman*, ser. Lecture Notes in Computer Science, O. Goldreich, Ed. Springer, 2011, vol. 6650, pp. 30–39. [Online]. Available: https://doi.org/10.1007/978-3-642-22670-0_5