Model-Generic Incrementally Verifiable Computation from Updatable BARGs

Eden Aldema Tshuva
Tel Aviv University, Israel

Rotem Oshman
Tel Aviv University, Israel

Abstract

Incrementally verifiable computation (IVC) is a computationally sound proof system that allows a prover to certify the correctness of a long or ongoing computation in an incremental manner, by repeatedly updating a proof certifying the computation so far. Updating the proof does not require access to the entire trace of the computation, which makes the IVC-prover memory efficient. Recently, such schemes were constructed for deterministic Turing machines from standard cryptographic assumptions (Paneth and Pass, FOCS 2022, and Devadas et al., FOCS 2022).

In this work we generalize and extend IVC to support incremental certification and verifiability of a large set of computation models, focusing on distributed and online computation. This allows distributed algorithms to efficiently certify their own execution using low memory and communication overhead. We construct IVC for a variety of computation models by proving one generic lifting theorem from a classical (non-incremental) delegation scheme (also known as SNARG) into full-fledged IVC, while preserving the delegation scheme's succinctness properties (up to an additive factor which is polynomial in the security parameter and independent of the size of the computation). Using this lifting theorem, we obtain IVC for the following computation models:

- *RAM and exclusive-read exclusive-write (EREW) PRAM algorithms*, using existing delegation schemes for these models.
- *Streaming algorithms*, using the natural memory-efficiency properties of the model.
- *Massively parallel computation (MPC)*. Notably, in this model, memory efficiency is a critical bottleneck: the machines participating in an MPC algorithm usually cannot store the entire trace of their computation. Thus, certifying MPC algorithms naturally benefits from IVC. Moreover, since prior to our work, no delegation scheme for this model was known, we also construct a delegation scheme for one-round massively parallel computations, and then apply our lifting theorem to it.
- *Distributed graph algorithms*, using existing distributed delegation schemes (also known as locally verifiable distributed SNARGs). Here, in order to use our lifting theorem we have to first make some observations about the verification procedure of these existing schemes.

At the heart of this work is a new abstraction, *updatable batch arguments for* NP (UpBARGs), which we define and construct. Standard BARGs allow one to prove a batch of k NP-statements using a proof whose length barely grows with k; however, the statements and their witnesses must all be known in advance. In contrast, UpBARGs support adding statements and witnesses on the fly, making them a flexible tool for constructing IVC across different computational models.

2012 ACM Subject Classification Theory of computation → Cryptographic protocols

Keywords and phrases incrementally verifiable computation, massively parallel computation, streaming, parallel RAM, batch arguments, SNARG

Digital Object Identifier 10.4230/LIPIcs.ITCS.2026.6

Funding *Eden Aldema Tshuva*: Research supported by the Israeli Science Foundation, Grant No. 2338/23, and by AFOSR award FA9550-24-1-0156.
Rotem Oshman: Research funded by the Israel Science Foundation, Grant No. 2801/20, and by AFOSR award FA9550-24-1-0156.

1 Introduction

Delegation schemes [19, 16, 17] enable a computationally weak entity to outsource a computation to a more powerful entity without having to trust it: the powerful entity returns the result alongside with a succinct proof of correct execution, which the weak entity can verify efficiently. Delegation schemes typically require the prover to access the entire trace of the computation, which can be impractical. *Incrementally verifiable computation (*IVC*)* [23] avoids this by maintaining the proof incrementally: given the state of the computation after t steps, and proof π_t showing that the first t steps have been executed correctly, the prover updates the proof to a new proof π_{t+1} that the first $t+1$ steps have been executed correctly, without having to store in memory the entire trace of the computation so far. This makes IVC a natural fit for scenarios where long computations must be executed reliably over time: for example, resuming a paused computation from an intermediate verified state, or continuously auditing the correctness of outsourced computation performed by an untrusted server or cloud. While [23] and follow-up work [5, 4] show how to construct IVC for NP from heuristic (non-falsifiable) assumptions, recently IVC for deterministic computations was constructed from LWE [20, 11],[1] and this is the setting we adopt in the current paper.

While [20, 11] develop IVC for sequential RAM, there are many other computation models that would benefit from IVC. In fact, a natural application of IVC is to computations that are *distributed* or *reactive* in nature, where the inputs arrive over time, or are jointly held by multiple parties that collaborate to carry out the computation, such as in streaming algorithms and in massively parallel computations. The incremental structure of IVC is particularly well-suited to such computation models, where it is usually impractical to store the full trace of the computation, which is required for non-incremental delegation schemes (i.e., succinct non-interactive arguments (SNARGs)). For example, when a computation is outsourced to a server farm, typically no single server has enough memory to store the entire trace of the full computation carried out across all the servers. Incremental verification avoids the need to store the full trace, and allows us instead to create a succinct proof of correctness and update it on-the-fly as the computation proceeds, with no asymptotic memory overhead other than the proof itself.

Despite this apparent fit, IVC for reactive and distributed computation models has received very little attention in the context of standard cryptographic assumptions.[2] To our knowledge, the only work in this direction is [1], which developed zero-knowledge IVC for streaming algorithms (using a different completeness notion and techniques than ours). In this paper we extend the scope of IVC to encompass such distributed and reactive models of computation, bringing the benefits of incremental verification to settings that arguably stand to gain from it the most. We do this in a generic and modular way, by giving a *lifting theorem* that applies to any deterministic computation model represented as a labelled transition system (LTS), and allows us to lift a succinct proof for the correctness of *one* computation step in the model into full multi-step IVC.

Constructing incrementally verifiable computation. Consider a computation starting from an initial configuration cf_0 and ending at configuration cf_t. Proving that the computation $\mathsf{cf}_0 \to \ldots \to \mathsf{cf}_t$ is correct boils down to proving the conjunction of t statements: "for

[1] While the constructions in [20, 11] are using specifically LWE, combining them with the works [24, 7] gives the same results also under bilinear maps assumption or under subexponential security of DDH.

[2] From heuristic assumptions, IVC schemes have been developed for streaming algorithms [13, 22], and a generalization of IVC, known as *proof-carrying data* (PCD), may be considered as a form of distributed IVC for some types of distributed computations [6, 5].

each $i = 0, \ldots, t-1$, the system transitions from configuration cf_i to configuration cf_{i+1} in one step". In Valiant's paper introducing IVC [23], this was done using a primitive he called *noninteractive computationally sound proof of knowledge*, and involved a tree-based construction where proofs for short computations are merged into proofs for longer ones: conceptually, at each point in time, the proof consists of a forest, where each tree proves correctness of a sub-computation of the form $\mathsf{cf}_i \to \ldots \to \mathsf{cf}_j$ for some $i < j$. There is at most one tree of any given height, and the proof stores only the roots of the trees, not the entire forest (making it succinct). To update the proof, the prover creates a new tree of height 1 for the last computation step $\mathsf{cf}_t \to \mathsf{cf}_{t+1}$, and then repeatedly *merges* any two consecutive proof-trees of the same height, until the invariant that there is at most one tree of any given height is restored. *Merging* two proof trees for sub-computations $\mathsf{cf}_i \to \ldots \to \mathsf{cf}_j$ and $\mathsf{cf}_j \to \ldots \to \mathsf{cf}_k$ yields a new tree proving the sub-computation $\mathsf{cf}_i \to \ldots \to \mathsf{cf}_k$, whose height is greater by 1 than the merged trees.

In [23], Valiant constructed noninteractive CS proofs of knowledge in the random oracle model; unfortunately, to date no construction of such proof system from standard assumptions is known. Follow-up work on (non-incremental) delegation [9, 15, 24, 7] and on IVC [11, 20] follows the conceptual idea of Valiant's construction but replaces noninteractive CS proofs of knowledge with *batch arguments for* NP *(*BARG*s)*, which are known to be constructible from standard assumptions [20, 11]. Informally, a BARG for an NP-language $\mathcal{L}$ allows us to prove the conjunction of k statements, each of the form "$x_i \in \mathcal{L}$", using a proof whose length grows linearly with the length of a *single* NP-witness for $\mathcal{L}$, and only polylogarithmically with the number of statements k.

The transition from noninteractive CS proofs of knowledge (used in [23]) to BARGs for NP (used in the IVC of [11, 20]) is not as smooth as one might hope: many technical issues arise, which are resolved in an ad-hoc manner specific to each construction. Trying to extend these constructions to more complex computation models becomes prohibitively complicated. One key difference is that in standard sequential computation (e.g., Turing machines or RAM), given the initial configuration, *the entire trace of the computation is determined*; even though the prover does not actually store the entire trace, this fact is crucial for the soundness proof of [11, 20]. In contrast, in more complex models such as distributed and streaming algorithms, the computation trace (of a single machine, in the distributed case) cannot be determined from the initial configuration alone, because it depends on inputs or messages that arrive during the computation. Fundamentally, while standard sequential IVC requires us to prove a *predictable* sequence of statements $\mathsf{cf}_0 \to \mathsf{cf}_1, \ldots, \mathsf{cf}_{t-1} \to \mathsf{cf}_t$, in more complex models we cannot predict in advance the statements that we will need to prove, as each statement has the form "$\mathsf{cf}_i \overset{z_{i+1}}{\to} \mathsf{cf}_{i+1}$", where z_{i+1} is the input (or message) that arrives in step $i+1$.

A generic framework for incrementally verifiable computation. To handle inputs that arrive over time in a simple and unified way across different computation models, we define and construct an updatable form of BARGs, which we call *updatable hash-and-*BARG *(*UpBARG*)*. An UpBARG proves the conjunction of k NP-statements, like a regular BARG; however, unlike a regular BARG, the UpBARG exposes a hash of the statements that it proves, and can be updated on-the-fly to add a new statement (which is not true for plain BARGs). Moreover, the UpBARG supports a *consistency check* specified by the user, which is applied to every pair of consecutive statements added to the UpBARG. This is especially useful for IVC, where we prove a batch of statements that each have the form "$\mathsf{cf}_i \overset{z_{i+1}}{\to} \mathsf{cf}'_i$"; in order for the conjunction of these statements to represent an actual trace of a computation, we must require that $\mathsf{cf}'_i = \mathsf{cf}_{i+1}$ for each i, and this is enabled by a consistency check applied to the statements "$\mathsf{cf}_i \overset{z_{i+1}}{\to} \mathsf{cf}'_i$" and "$\mathsf{cf}_{i+1} \overset{z_{i+2}}{\to} \mathsf{cf}'_{i+1}$".

With the UpBARG in hand, we are able to extend delegation schemes into IVC. To give a unified statement which applies to diverse computational models, we use the general notion of deterministic computation models represented as a labeled transition system. This allows us to generalize the concept of delegation, and refer to a delegation from a stronger model of computation (where the prover is implemented) to a weaker model of computation (where the verifier is implemented). For instance, in the case of RAM delegation, the prover is a RAM machine and the verifier is a low-space Turing machine. Note that since the prover and verifier are in different models of computation, and thus have different computational power, even delegating one step of computation on a stronger model to a weaker model may be non-trivial (and this is the exact case in some of the models we handle, such as the massively parallel computation model).

Our main contribution is the following "lifting"-type theorem from one-step delegation to multi-step IVC:

▶ **Theorem 1** (Informal). *Let $\mathbb{M}, \mathbb{V}$ be deterministic computation models represented as labeled transition systems, such that*

1. *There exists a delegation scheme succinctly proving the correctness of a* single computation step *of $\mathbb{M}$, where the prover runs in the model $\mathbb{M}$ and the verifier runs in $\mathbb{V}$; and*
2. *The prover and the verifier of the* UpBARG *can be implemented in $\mathbb{M}$ and $\mathbb{V}$, respectively.*

Then there is an IVC *scheme for $\mathbb{M}$, where the prover runs in $\mathbb{M}$ and the verifier runs in $\mathbb{V}$.*

The IVC construction described in Theorem 1 and its analysis are relatively straightforward compared to prior work on IVC: the interface of the UpBARG allows us to cleanly implement Valiant's construction [23], avoiding many of the complications that arise in prior work. The construction of the UpBARG itself is closely based on [20], but there are some subtleties. For example, as part of the UpBARG's interface, we must be able to provide an opening of the hash to any given statement, without holding all the statements in the clear, even though we do not know what statements will arrive in the future and go into the hash *after* the statement we are interested in. We show that we can compute the required openings *in hindsight*, using properties of tree-based hash families with local openings.

Applications. Using Theorem 1 we develop IVC for several reactive and distributed computation models: streaming algorithms,concurrent PRAM algorithms in the exclusive-read exclusive-write model (EREW), distributed algorithms in the CONGEST model, and distributed algorithms in the massively parallel computing (MPC) model. CONGEST is a model of network algorithms: it features a network of processors that communicate synchronously over some graph topology, with the number of bits that can be sent across any communication link bounded per synchronous round. The focus in this model is on communication as a bottleneck, not on space or even on local computation steps. In contrast, MPC is a model of server farms or cloud computing, where a large number of space-bounded machines collaborate to compute on an input partitioned between them. In each synchronous round, each machine can communicate with multiple other machines arbitrarily, subject only to a bound on the total number of bits sent and received.

While most of the models we consider already have *non-incremental* delegation schemes, for the MPC model this is not the case: proving the correctness of MPC computations almost *requires* IVC, because no single machine can even store the entire input, much less the entire trace of the distributed computation. Thus, while communication-efficient distributed provers have already been developed for other distributed models which are not space-bounded [12, 2, 3], prior to our work no such construction was known for MPC, even if we

do not insist on IVC and are willing to settle for a non-incremental construction. In fact, even proving the correctness of a *single* global computation step of the MPC model (i.e., a single synchronized round, where each machine receives messages, computes locally, and sends messages) is non-trivial, as it involves message exchanges between a large number of machines. To instantiate Theorem 1, we first use UpBARGs to construct a one-step delegation scheme for the MPC model, and then apply the theorem to obtain IVC for MPC.

We consider two use cases for our IVC for MPC. The first is reliable delegation to the cloud: here, the user holds an input x, and outsources a difficult computation on x to a server farm or a cloud provider, which returns both the output and a short correctness certificate. The user is a single time- and space-bounded machine; it hashes x once, never re-reads x (only its hash), and verifies the result of the computation in time polylogarithmic in $|x|$. The second use case that we consider is providing *internal* reliability to an MPC computation: instead of an external user, here it is the network itself that wants to verify that the computation so far has proceeded correctly (e.g., to protect against faults or issues with some of the machines). Our distributed IVC allows the network to maintain a succinct, continuously updatable correctness proof that any machine can individually store and verify locally against its own input, such that if *all* machines accept the proof, then the computation is correct (except with negligible probability). This use case resembles a distributed fault-tolerance mechanism for the CONGEST model called *proof labeling schemes* [18], where each machine stores a short certificate, and the machines communicate with each other to verify that the network is in a legal configuration. Our IVC for MPC scheme is more efficient, as our proof size grows polylogarithmically with the network and input size (in proof labeling schemes, the overall proof size is at least linear in the network size), and verifying the proof requires no communication between the machines. We note that the IVC construction that we give for the CONGEST model returns to the traditional proof labeling scheme model, and has the machines each store a short certificate and communicate with its neighbors to verify it (as is also done in the non-incremental delegation schemes of [12, 2, 3]).

1.1 Related Work

We briefly survey related work. Since our results and techniques lie in the regime of arguments and delegation schemes from *standard*, and more importantly, falsifiable, cryptographic assumptions, we focus here on constructions in this regime.

Incrementally verifiable computation from LWE. The recent work of Paneth and Pass [20] and Devadas et al. [11] constructs IVC from standard cryptographic assumptions. Our construction is similar in spirit to these constructions (specifically to [20]), but the provers of [20, 11] require access to the full current state of the system, and their verifiers must read the full state of the system to verify it. We allow for more limited access; for instance, in distributed graph algorithms our prover updates the proof without any single entity knowing the entire communication graph state. That said, these works are closely related to ours: both [20, 11] construct rate-1 batch arguments, which we use to construct our UpBARGs, and the framework of [20] for constructing IVC serves as a starting point for the construction of our UpBARGs.

As mentioned above, there is an independent construction of IVC for streaming algorithms [1], which is also zero-knowledge. In the current paper, IVC for streaming is almost trivially implied by Theorem 1 and our UpBARG construction. We do not attempt to provide zero-knowledge.

Works from indistinguishability obfuscation. In general, succinct arguments from $i\mathcal{O}$ have some major drawbacks; in order to instantiate them from falsifiable assumptions they usually require the CRS to be non-succinct and the underlying assumption to be exponentially secure. In contrast, all of our results are instantiable from either LWE, bilinear maps assumptions, or subexponential security of DDH. Nevertheless, the following recent works are related to ours. The first notion of updatable batch arguments was defined and constructed in [14] from $i\mathcal{O}$. As is the case with the earlier SNARG for NP from $i\mathcal{O}$ [21], their argument is non-adaptively sound. That is, the statements in the soundness game are chosen before the CRS, and the latter may depend on them. The somewhere argument of knowledge property is incomparable, as it is on one hand adaptive (the CRS is chosen before the statements) but on the other hand only provides guarantee somewhere (while non-adaptive soundness is a guarantee about all locations at the same time).

Very recently, [10] constructed IVC for NP from the combination of $i\mathcal{O}$ and LWE with proofs which grow polynomially with the size of one configuration and polylogarithmically with the number of steps. They also offered an IVC with proofs which do not grow with the size of one configuration which works for the special case of trapdoor languages.

2 Preliminaries

Algorithms, computation models and traces. We represent an algorithm by a labeled transition system (LTS) (C, Λ, T), where C is a set of configurations (i.e., states of the algorithm), Λ is a set of labels, and $T \subseteq C \times \Lambda \times C$ is a transition relation between configurations. The labels typically represent the input to the algorithm, but in some scenarios they may represent something else (e.g., a message received by a machine in the MPC model). We restrict to *deterministic* algorithms, where T is single-valued: for any configuration $\mathsf{cf} \in C$ and label $\lambda \in \Lambda$ there is a unique configuration $\mathsf{cf}' \in C$ such that $(\mathsf{cf}, \lambda, \mathsf{cf}') \in T$. We sometimes represent T is a function, $T(\mathsf{cf}, \lambda) = \mathsf{cf}'$.

A *computation model* is simply a set of algorithms (i.e., LTSs). A *trace* or *computation* of an algorithm is a sequence $\mathsf{cf}_0 \xrightarrow{\lambda_1} \mathsf{cf}_1 \xrightarrow{\lambda_2} \ldots \xrightarrow{\lambda_t} \mathsf{cf}_t$, such that $(\mathsf{cf}_i, \lambda_{i+1}, \mathsf{cf}_i) \in T$ for each $i = 0, \ldots, t-1$.

The common reference string model and computational hardness. Our work is set in the *common reference string* (CRS) model, where all parties have access to a string that is sampled randomly by a trusted setup process, denoted Gen, which takes a *security parameter* λ and returns the CRS. The security parameter governs the computational resources that must be invested to break security: we say that a task involving the CRS is *computationally hard* there is no adversary that takes $\mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda)$ and runs in time polynomial in λ, can succeed in the task, except with *negligible probability* in λ. The following primitives are all in the CRS model and all include CRS generation as part of their interface.

Delegation schemes. A delegation scheme involves a prover and a verifier; the prover tries to convince the verifier of the correctness of some computation. The goal here is succinctness: the proof short, and in this context, succinct means polylogarithmic in the size of the input to the computation. We adopt the notion from [16], where the proof is with respect to a *digest* (a hash) of the initial and final configuration of the computation. The digest is computed honestly by calling $\mathsf{Digest}(\mathsf{crs}, \mathsf{cf})$ with a configuration cf, and the algorithm Digest satisfies computational collision-resistance. Soundness requires that a dishonest prover cannot

convince the verifier of two "contradictory statements" x_0, x_1, where each x_i asserts that from an initial configuration represented by *the same* digest h, the computation reaches a final configuration represented by digest h_i, but $h_0 \neq h_1$.

In this work we use a generalized notion of delegation scheme we call $\mathbb{V}$-$\mathbb{P}$ delegation, where the digest and prover are $\mathbb{P}$ algorithms and the verifier is a $\mathbb{V}$ algorithm. We moreover adjust the syntax to fit label transition systems, and add a label digest algorithm. The syntax is given below.

- $\mathsf{Digest}(\mathsf{crs}, \mathsf{cf}) \to h$: A $\mathbb{P}$-algorithm that takes a reference string crs, and a configuration $\mathsf{cf} \in \mathsf{CF}[\mathbb{P}]$. It outputs digest h.
- $\mathsf{LblDigest}(\mathsf{crs}, (\mathsf{lbl}_i)_{i \in t}) \to H$: A $\mathbb{P}$-algorithm that takes a reference string crs and a series of labels $(\mathsf{lbl}_i)_{i \in [t]} \subseteq \mathsf{Lbl}[\mathbb{P}]$. It outputs digest H.
- $\mathcal{P}(\mathsf{crs}, \mathcal{M}, \mathsf{cf}, t, (\mathsf{lbl}_i)_{i \in [t]}) \to \pi$: A $\mathbb{P}$-algorithm that takes a reference string crs, a $\mathbb{P}$-algorithm $\mathcal{M}$ a configuration $\mathsf{cf} \in \mathsf{CF}[\mathbb{P}]$, a number of steps t, and a set of t labels $(\mathsf{lbl}_i)_{i \in [t]}$. It outputs a proof π.
- $\mathcal{V}(\mathsf{crs}, \mathcal{M}, h, H, h', t, \pi) \to b$: A $\mathbb{V}$-algorithm that takes a reference string crs, a $\mathbb{P}$-algorithm $\mathcal{M}$, an initial digest h, a final digest h', a number of steps t, and a proof π. It outputs an acceptance bit b.

Incrementally Verifiable Computation (IVC). An IVC scheme is similar to a delegation scheme, but the prover incrementally *updates* an existing proof, and is therefore represented by an update procedure $\mathcal{U}(\mathsf{crs}, \mathcal{M}, \mathsf{cf}, \pi)$ that takes the current configuration cf and proof π, and returns the next configuration cf' and proof π'. The verifier also takes the current number of steps t. Soundness is the same as for delegation. However, completeness is replaced by *incremental completeness*, which intuitively requires that any proof accepted by the verifier can be extended into a proof for the next step:

- Incremental completeness: for any crs generated by Gen, step count $t < 2^\lambda$, digest h_{start}, machine $\mathcal{M}$, configuration cf and proof π, if $h_{\mathsf{start}} = \mathsf{Digest}(\mathsf{crs}, \mathsf{cf})$ and $\mathcal{V}(\mathsf{crs}, \mathcal{M}, t, h_{\mathsf{start}}, h, \pi) = 1$, then for $(\mathsf{cf}', \pi') = \mathcal{U}(\mathsf{crs}, \mathcal{M}, \mathsf{cf}, \pi)$ and $h' = \mathsf{Digest}(\mathsf{crs}, \mathsf{cf}')$, we have $\mathcal{V}(\mathsf{crs}, \mathcal{M}, t+1, h_{\mathsf{start}}, h', \Pi') = 1$.

This formulation does not involve input that arrives during the computation. In reactive models, the prover takes the input in addition to the current configuration, and produces a digest of the inputs in addition to the new configuration and proof. The verifier is given the digest of the inputs in addition to the parameters it is given above.

Somewhere extractable batch arguments for NP. A batch argument (BARG) [8] allows a prover to succinctly prove the conjunction of k NP statements, with a proof whose size is roughly the same as the size of one witness and barely grows with k. The interface is as follows:

- $\mathcal{P}(\mathsf{crs}, \mathcal{M}, x_1, \ldots, x_k, w_1, \ldots, w_k) \to (\pi)$: takes a reference string crs, an NP-machine $\mathcal{M}$, instances $x_1, \ldots, x_k$ and witnesses $w_1, \ldots, w_k$, and outputs a proof π.
- $\mathcal{V}(\mathsf{crs}, \mathcal{M}, \pi) \to b$: takes a crs, an NP-machine $\mathcal{M}$, and proof π and outputs an acceptance bit b.

Note that the prover $\mathcal{P}$ requires *all k statements and corresponding witnesses* to construct the BARG proof; this is one key difference between plain BARGs and the UpBARGs that we introduce later.

We say a BARG is *somewhere extractable* if it allows the extraction of *one witness* from a convincing proof π: the crs generation procedure Gen can be called in *trapdoor mode*, where it takes as additional input an index $i \in [k]$, called the *binding index*. In this mode, Gen

outputs a pair $(\mathsf{crs}, \mathsf{td}_i)$, where td is a *trapdoor* that can later be used to recover the i-th witness. In trapdoor mode, the Gen procedure has a property called *index hiding*: given crs (without td_i), it is computationally hard to *find* the binding index i. We require *somewhere argument of knowledge*: there exists an efficient *extractor* $\mathcal{E}(\mathsf{td}_i, \mathcal{M}, \pi) \to w_i$, which satisfies the following. Suppose we call Gen in trapdoor mode with a binding index i and obtain $(\mathsf{crs}, \mathsf{td}_i)$. Given only crs, it is computationally hard to find a proof π that is accepted by the verifier, such that when we extract a witness w_i using $\mathcal{E}(\mathsf{td}_i, \mathcal{M}, \pi)$, we have $\mathcal{M}(x_i, w) \neq 1$.

3 Technical Overview

In this section we give an overview of our IVC construction, focusing on the definition and construction of the UpBARG, the proof of Theorem 1, which lifts from one-step delegation to IVC, and the application to massively parallel computation (MPC).

3.1 Defining Updatable Hash-and-BARGs

An updatable hash-and-BARG is defined with respect to some hash family with local opening; for simplicity, in this overview we use a Merkle tree. The UpBARG consists of three algorithms, $(\mathsf{Gen}, \mathcal{U}, \mathcal{V})$, where Gen is a CRS generation algorithm, and $\mathcal{U}, \mathcal{V}$ are update and verification algorithms, respectively, with the following syntax.

- $\mathcal{U}(\mathsf{hk}, \mathsf{crs}, \mathcal{M}, H, \Pi, x, w) \to (H^*, \Pi^*)$: the UpBARG's update procedure takes as input a hash key hk, common reference string crs, NP-machine $\mathcal{M}$, hash value H, proof Π, and new statement x and witness w, and outputs a new hash value H^* and proof Π^*.
- $\mathcal{V}(\mathsf{hk}, \mathsf{crs}, \mathcal{M}, H, \Pi) \to b$: the UpBARG verifier takes a hash key hk, common reference string crs, NP-machine $\mathcal{M}$, hash value H and proof Π, and outputs an acceptance bit $b \in \{0, 1\}$.

Note the difference between the UpBARG's interface and the interface of a plain BARG: first, the UpBARG's update procedure does not require access to all the statements and witness, only to the current UpBARG (H, Π) and the new statement and witness (x, w). In addition, the UpBARG (H, Π) directly exposes a hash H of the statements, in addition to the proof Π. This differs from plain BARGs, where only the proof is exposed (implicitly, there is still a hash of the statements, but it is part of the proof and is not accessible to the user of the BARG). For our applications it is crucial to expose the hash, because in a reactive computation model it is meaningless to say that a computation starting in configuration cf_0 and ending in cf_t is "correct"; we can only say that it is correct *with respect to the input* that arrived as the computation was ongoing. Thus, the IVC verifier requires access to a hash of the statements (which include the input), not just the proof of their conjunction.

Our UpBARG satisfies the usual succinctness and index hiding properties of a BARG, but completeness and somewhere argument of knowledge are strengthened as follows:

- *Incremental completeness:* if $\mathcal{V}(\mathsf{hk}, \mathsf{crs}, \mathcal{M}, H, \Pi) = 1$ and $\mathcal{M}(x, w) = 1$, then we also have $\mathcal{V}(\mathsf{hk}, \mathsf{crs}, \mathcal{M}, H^*, \Pi^*) = 1$, where $(H^*, \Pi^*) = \mathcal{U}(\mathsf{hk}, \mathsf{crs}, \mathcal{M}, H, \Pi, x, w)$.
- *Somewhere argument of knowledge:* upon using the trapdoor version of Gen, programmed to location i, the extractor $\mathcal{E}$ extracts not only a witness w_i, but also a statement x_i and an opening ρ proving that H indeed opens to x in the index i.

The opening ρ extracted by $\mathcal{E}$ is crucial to the soundness of our IVC (discussed below), but it is non-trivial to obtain: at the time the i-th statement and witness are added to the UpBARG, we do not yet *know* what the opening to index i will be, because future statements added to the UpBARG will change it. We show that we can use the structure of Merkle trees and auxiliary information computed at update time to compute this opening in hindsight.

Supporting consistency checks. Our IVC requires the ability to claim that two extracted instances are *consistent* with each other, for a definition of "consistent" that is supplied by the user. To that end, we extend the syntax above, and give the update and verification procedures two additional inputs: a Turing machine $\mathcal{C}(\cdot,\cdot)$ specifying the consistency check, and a "last instance" x^-, which is intended to be the last statement added to the UpBARG. The incremental completeness guarantee now applies only when $\mathcal{C}(x^-, x) = 1$, that is, only when the new statement x is "consistent" with the preceding statement x^-. Also, the extractor $\mathcal{E}$ of the somewhere argument of knowledge guarantee now additionally extracts a *previous instance* x^- and a respective opening ρ^-, and we require in addition to the previous requirements that (1) H opens to x^- in location $i-1$ and this is proved by the opening ρ^-, and (2) $\mathcal{C}(x^-, x) = 1$. Finally, in the version which supports consistency checks we also require a useful property that we call *last-statement collision-resistance:* it is computationally hard to find a tuple $(\mathcal{M}, H, \Pi, \mathcal{C})$, two different last statements $x_0^- \neq x_1^-$, and an opening ρ, such that $\mathcal{V}$ accepts $(\mathsf{hk}, \mathsf{crs}, \mathcal{M}, H, \Pi, \mathcal{C}, x_0^-)$, but ρ proves that H opens to x_1^- in the location of the final instance.

3.2 Lifting One-Step Delegation into IVC using UpBARGs

We now outline our generic IVC construction using UpBARGs (Theorem 1). Let $\mathbb{P}$ be the computation model whose executions we would like to incrementally prove correct; we assume that the IVC prover also runs in the model $\mathbb{P}$. However, the verifier may be computationally weaker, and runs in some other model $\mathbb{V}$. We require that the UpBARG's prover and verifier run in the models $\mathbb{P}$ and $\mathbb{V}$, respectively; this is not a significant limitation, because in our construction both are space-efficient Turing machines.

Suppose we have a (non-incremental) delegation scheme $(\mathsf{Gen}, \mathcal{P}, \mathcal{V})$ that can succinctly prove *one-step* computations in $\mathbb{P}$, and let $\mathcal{A}(C, \Lambda, T) \in \mathbb{P}$ be an algorithm (represented as an LTS). Let $\mathcal{M}_{\mathcal{V}}$ be an algorithm in $\mathbb{V}$ that implements the verifier $\mathcal{V}$, with the parameters of the delegation scheme (e.g., the crs, the algorithm and the security parameter) hardwired: $\mathcal{M}_{\mathcal{V}}$ takes as input digests h, h', a label $\lambda \in \Lambda$ and a proof π, interprets h, h' as digests $h = \mathsf{Digest}(\mathsf{cf}), h' = \mathsf{Digest}(\mathsf{cf}')$, and computes the output of the verifier $\mathcal{V}$ on (h, λ, h, π).

Our IVC prover is quite simple: it maintains an UpBARG for the language $\mathcal{M}_{\mathcal{V}}$, viewed as an NP-language where the statement is the triplet (h, λ, h') and the witness is the one-step delegation proof π (so that $\mathcal{M}_{\mathcal{V}}((h, \lambda, h'), \pi) = 1$ iff the verifier $\mathcal{V}$ of the one-step delegation scheme accepts). We use a simple consistency check which requires that for any two consecutive statements (h, λ, h') and (g, σ, g') we have $h' = g$. The IVC prover runs alongside the algorithm $\mathcal{A}$; whenever $\mathcal{A}$ transitions from cf to cf' upon receiving input λ, the IVC prover computes $h = \mathsf{Digest}(\mathsf{cf}), h' = \mathsf{Digest}(\mathsf{cf}')$, and uses $\mathcal{P}$ to obtain a proof π for the (digested) transition $\mathsf{cf} \xrightarrow{\lambda} \mathsf{cf}'$. Then it updates the UpBARG by adding the instance-witness pair $((h, \lambda, h'), \pi)$.

The prove soundness, we show that if an adversary $\mathcal{A}$ convinces the verifier to accept two contradictory proofs π_0, π_1 for final digests $h^0 \neq h^1$ with respect to the same initial digest h_{start}, input hash h_{in} and step number T, then for each $i = 0, \ldots, T-1$, a similar adversary $\mathcal{A}_i$ can do the same when we use the CRS generator of the UpBARG in trapdoor mode to bind location i. This allows us to extract instances $x_i^0 = (h_i^0, \lambda_i^0, h_{i+1}^0), x_i^1 = (h_i^1, \lambda_i^1, h_{i+1}^1)$ from the two respective proofs, alongside with their respective witnesses, which are proofs for the state transition in the delegation scheme: $w_i^0 = \pi_i^0$ and $w_i^1 = \pi_i^1$. We then prove by induction on i that in the ith experiment described above, we have $h_{i+1}^0 = h_{i+1}^1$ with overwhelming probability . Then, using the last-statement collision resistance property for experiment T, we argue that it must be that $h_t^0 = h^0$ and $h_t^1 = h^1$, which implies that final digests were the same, $h^0 = h^1$, a contradiction.

To prove the ith induction step, we use the index hiding property to move from the induction hypothesis experiment into a hybrid experiment where the crs is binding in both indices $i-1, i$, and then use the consistent argument of knowledge property of the UpBARG, the soundness property of the underlying delegation scheme, and the collision resistance property of the hash family to prove that $h_i^0 = h_i^1$ in this hybrid experiment. We finally use the index hiding property again to move into the next experiment of the induction, where the crs is only binding on i.

One way in which this construction and its analysis differ from previous IVC constructions (e.g., [20, 11]) is the explicit use of consistency checks to enforce equality between the last and first configurations (respectively) of every two successive transitions; prior work, which did not have consistency checks, used somewhat ad-hoc solutions to ensure this equality.

3.3 Constructing an Updatable Hash-and-BARG

Our UpBARG construction is similar to [20], with the main differences being the extractor's ability to extract an opening in hindsight, and the support for consistency checks, which also requires extraction of the opening to the next-to-last statement. We briefly review the parts of the construction that are similar to [20], and then explain how the unique features of the UpBARG (which are not present in [20]) are implemented.

In [20] (as in prior work on delegation and IVC), the proof of a conjunction of statements $x_1, \ldots, x_k$ with corresponding witnesses $w_1, \ldots, w_k$ is represented as a forest of complete binary trees. The leaves of the trees are individual statements and witnesses, and each inner node intuitively represents a proof for the conjunction of its two children, so that the root of each tree (informally) proves all the statements represented by its leaves. We do not store the entire forest, but rather only the root of each tree; in addition, we make sure that there is always at most one tree of any given height, so that the total number of trees is $O(\log k)$.

We refer to the height of a trees as its *level*, and we define a different verifier for each level: at level 0 (a leaf), the verifier is simply the NP-verifier of the language, which takes (x_i, w_i) and accepts or rejects. At levels $\ell > 0$, each node stores a BARG proof asserting that the level-$(\ell-1)$ verifier accepts both of its children. The witness for each child is the level-$(\ell-1)$ BARG proof that it represents, if the child is an inner node, or if the child is a leaf, the witness w_i for the statement represented by the leaf. In turn, the level-ℓ verifier is the verifier of the BARG constructed for level-ℓ. To add a new statement and witness (x_{k+1}, w_{k+1}), the prover creates a new leaf for (x_{k+1}, w_{k+1}). It then repeatedly merges any two consecutive trees of the same level ℓ, by creating a new level-$(\ell+1)$ node with a BARG asserting that the roots of both trees are accepted by the level-ℓ verifier, and then discarding the two level-ℓ roots.

To extract the i-th witness, we find the tree root containing statement i (this is determined by the index i and the levels of the trees, as all trees are complete). From the root, we descend down the tree to the leaf representing the i-th statement and witness, at each step using the extractor of the BARG at the current node to obtain the BARG proof of the child to which we descend, until we arrive at a leaf; in [20], at the leaf we extract the i-th witness w_i, which we then return.

Maintaining an explicit hash of statements, and tying it to the proof. In the UpBARG we expose to the user a hash with local openings of the statements that went into the UpBARG (which is not done in prior work). This is straightforward: we maintain a separate hash forest, with the same structure as the proof forest described above – at the leaves we hash the statements themselves, and each inner node is the hash of its two children. To tie the proof

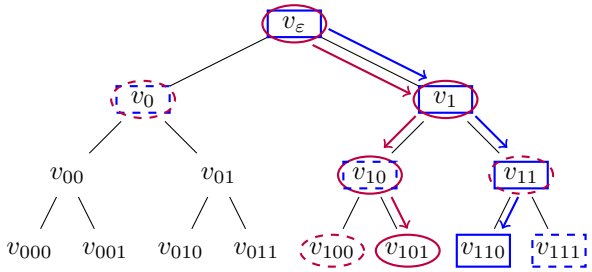

Figure 1 The Merkle tree opening to location i consists of all of the siblings of the nodes on the path from the root to the ith leaf. In this example, the paths from the root v_ε to the leaves v_{101}, v_{110} are indicated with red arrows and ellipses and with blue arrows and rectangles, respectively; the openings to nodes v_{101}, v_{110} are indicated with dashed red ellipses and blue rectangles, respectively.

forest to the hash forest so that it "refers" to the same statements and supports extraction, we modify the level-ℓ verifier: at leaf nodes ($\ell = 0$), the verifier takes the statement stored in the leaf of the hash forest and the witness stored in the leaf of the proof forest, and checks that the NP-verifier accepts them. At inner nodes ($\ell > 0$), the verifier checks that the current node h in the hash forest indeed stores the hash of its two children h_0, h_1 (in addition to verifying the level-ℓ BARG as before). The *witness* for this claim is the pair (h_0, h_1), i.e., the hashes represented by the children.

Extracting openings in hindsight. The extractor of the UpBARG must return the i-th statement x_i and a corresponding opening in the hash forest, in addition to the witness w_i whose extraction we described above. Extracting the statement x_i is easy: as we descend towards the i-th leaf, we hold at each point a BARG proof for some node in the proof forest, and the hash of the corresponding node in the hash forest. We extract from the BARG proof both the BARG proof of the child to which we descend, and also the hash at that child – recall that the hashes (h_0, h_1) of both children are part of the witness. This allows us to continue our descent, until we arrive at a leaf in the proof forest, which stores the witness w_i, and simultaneously a leaf in the hash forest, which stores the statement x_i.

To extract an opening ρ_i to index i in the hash forest, we rely on the structure of a Merkle tree opening – see Figure 1. The opening is assembled as we recurse down the tree: at each node h in the hash tree, we descend to some child h_b (for $b \in \{0, 1\}$), but the witness that we extract provides both h_0 and h_1, that is, it provides the sibling of the node to which we descend. We append this sibling to the opening, and continue our descent.

Supporting consistency checks. Fix a consistency-checking machine $\mathcal{C}(\cdot, \cdot)$. In order to support consistency checks, we store additional information in the proof forest, which allows us to locate and open the last statement added. We also modify the verifiers of the proof forest, to check that this auxiliary information correctly reflects the structure of the forest.

We observe that the opening to index $i-1$ consists of three parts, ordered top-down each of which might be empty (see Figure 1 for an illustration):

1. A prefix consisting of the opening to the lowest common ancestor of $i-1$ and i: this part of the opening to index $i-1$ is shared with the opening to index i (in Figure 1, the lowest common ancestor is v_1, and the opening to it is v_0).
2. A middle part consisting of a single node – the right child of the lowest common ancestor of $i-1$ and i, where the paths to $i-1$ branch left and right, respectively (in Figure 1, this is v_{11}).

3. A suffix consisting of the opening to from the left child of the common ancestor down to index $i-1$, which must be the rightmost child in the subtree. Importantly, this suffix is the same as the opening from the *root* of a Merkle tree containing only the first $i-1$ statements (in Figure 1, this is v_{100}), which we can compute at the time statement i is added (i.e., it does not require us to know what future statements will be added). While the trees containing $i-1$ and/or i may be merged with other trees in the future, the opening from the lowest common ancestor of $i-1, i$ to $i-1$ does not change.

The description above assumes that $i-1$ and i are in the same tree in the forest. If they are not, then $i-1$ is the rightmost leaf in its tree, and i is the leftmost leaf in its tree; the first two parts described above are empty, and the third part is simply the opening from the root to the rightmost leaf in the tree containing $i-1$.

We already saw how to extract an opening to index i, and during this process we obtain the first two parts of the opening to index $i-1$: the opening from the root to the lowest common ancestor v of $i-1$ and i, as well as the right child v_1 of v (which is on the path down to i). It remains to obtain the third part: the opening to the leaf that precedes the leftmost leaf in v_1's subtree.

Obtaining this information at extraction time requires us to store additional information when we construct the hash tree: when a node v is constructed, we store (ℓ^-, h^-, ρ^-) at node v, where ℓ^- and h^- are the level and hash root (respectively) of the tree containing the leaf immediately preceding the leftmost leaf in v's subtree, and ρ^- is the opening to this rightmost leaf (from the root h^-). We emphasize that this information is computed and stored *at construction time*, and may be rendered inaccurate by future updates: for example, h^- may no longer be the *root* of the tree containing the desired leaf, as that tree can be merged with other trees. Nevertheless, our observation above shows that the information we store at construction time is exactly what we need to compute the opening to the preceding statement: as we descend to index i, we identify the lowest common ancestor v of $i-1$ and i, where the path to index $i-1$ diverges from the path to i; we descend to the right child v_1 of v, and take ρ^- from v_1 to complete the opening to index $i-1$.

We must also modify the BARG proofs and verifiers at each level, to ensure that the additional information is correct. For nodes at level $\ell \geq 1$, the BARG now asserts an *asymmetric* conjunction, which states as before that the level-$(\ell-1)$ verifier accepts each child node (with an appropriate witness), but also that each child of the current node "points correctly" towards the tree containing the leaf that immediately preceded its subtree at construction time: the left child of a node v must point towards the same leaf that node v itself points to, whereas the right child of v must point towards the left child of v (hence the asymmetry).

Finally, at each leaf node i we additionally store the preceding statement x_{i-1}, which allows us to extract x_{i-1} together with x_i. The level-0 verifier checks that $\mathcal{C}(x_{i-1}, x_i)$ accepts (i.e., statement x_{i-1} is "consistent" with x_i), and also that if (ℓ^-, h^-, ρ^-) is the extra information stored at leaf i, then h^- indeed opens to x_{i-1} at index $2^{\ell^-}-1$ using the opening ρ^-.

3.4 Delegating One-Round Massively Parallel Computations

In the MPC model, n space-bounded machines communicate in synchronized rounds. Each machine has s bits of memory, and in each round it can send and receive messages from any other machine, provided the total number of bits sent or received does not exceed s. In this section we outline how we use UpBARGs to construct a one-round delegation scheme for MPC, which we then lift into IVC using Theorem 1.

One MPC round as a RAM program. We observe that one round of MPC on n machines can be described as a RAM program, which takes a vector of the machines' initial states $(\mathsf{st}(1), \ldots, \mathsf{st}(n))$, and outputs the machines' states in the end of the round $((\mathsf{st}'(1), \ldots, \mathsf{st}'(n))$. Each state $\mathsf{st}(i)$ or $\mathsf{st}'(i)$ is an s-bit vector, where s is the space bound of an individual machine. To construct a delegation scheme for one-round-MPC, we imitate a delegation scheme for the program above. However, while a RAM program is executed on a single machine that can access any place in its random-access memory, in the MPC model, the "memory" is partitioned across the n machines. Fortunately, existing RAM delegation schemes have the property that constructing the proof does not require access to the entire memory configuration: it is enough to have a *hash with local openings* of the memory, and openings to the locations that are accessed by the algorithm in each step. Thus, to construct our desired RAM-delegation-like proof for the MPC model, we need to go through the following three stages: (1) Obtain a hash of the state vector $(\mathsf{st}(1), \ldots \mathsf{st}(n))$; (2) For each machine $i \in [n]$, obtain openings of the above hash value to the messages $m_1^i, \ldots, m_k^i$ (where $k \leq s$) that machine i receives during the round[3]; and (3) Using the openings, prove the transition $(\mathsf{st}(1), \ldots, \mathsf{st}(n)) \to (\mathsf{st}'(1), \ldots, \mathsf{st}'(n))$. The main challenge in this scheme is that each stage must be implemented in the MPC network: we do not *truly* have random access to the state assignments, because these are actual states of individual machines. For all three stages, our solution is based on having the n machines simulate some process on a tree-structured network. This network is an s-ary tree with n leaves which represent the states of the n actual machines, and the rest of the nodes are virtual nodes, simulated by the n real machines.

Using a virtual tree-structured network. To obtain a hash of the state assignment $(\mathsf{st}(1), \ldots, \mathsf{st}(n))$, we aggregate the hash up the tree: the leaves send their state to their parents, and each inner node, upon receiving values from its s children, hashes the values it received and sends them up to its parent. The root of the tree then obtains a hash of the entire state assignment. In order for each machine $i \in [n]$ to obtain an opening to its state $\mathsf{st}(i)$, we proceed this time down the tree, with each inner node receiving from its parent a partial opening down to its index, extending it to an opening for each of its children's indices, and sending each extended opening to the corresponding child. Each inner node is able to extend the opening because it is the one that hashed all of its children's values together. Eventually, each machine $i \in [n]$ receives from its parent a complete opening down to index i. Finally, we construct a proof for the alleged RAM computation $(\mathsf{st}(1), \ldots, \mathsf{st}(n)) \to (\mathsf{st}'(1), \ldots, \mathsf{st}'(n))$. To do so, we first observe that this transition is defined by n separate computations, and for each $i \in [n]$, we prove that when the network state assignment is $(\mathsf{st}(1), \ldots, \mathsf{st}(n))$, the next state of machine i is $\mathsf{st}'(i)$. Each such statement can be proved at machine i, as follows. Let j be a machine that sends message m_j^i to machine i during the round. Recall that after the previous stage, machine j has an opening from the global hash to its state $\mathsf{st}(j)$. Machine j now computes an opening from $\mathsf{st}(j)$ to m_j^i and sends this opening to machine i. Each machine, upon receiving these openings, uses a RAM-delegation prover to prove that its transition $\mathsf{st}(i) \to \mathsf{st}(i+1)$ is legal.

The remaining challenge is to aggregate the individual proofs of the machines into one global succinct proof while maintaining soundness. To do this we use the virtual network again: we proceed up the tree, with each node sending its current proof up to its parent. Upon receiving the s proofs from its children, each inner node uses an UpBARG to obtain a

[3] We assume for simplicity that the messages each machine sends are stored as part of its state at the beginning of the round.

proof of the *conjunction* of its children's statements. Finally, at the root of the tree, we obtain our one succinct proof for the entire transition $(\mathsf{st}(1),\ldots,\mathsf{st}(n)) \to (\mathsf{st}^+(1),\ldots,\mathsf{st}^+(n))$. We remark that the soundness of this scheme is non-trivial, and to argue soundness we have to use special properties of the hash family (shortly, that an opening to x_i always contains the hash of $(x_1,\ldots,x_{i-1})$).

3.5 Incrementally Verifying Distributed Graph Algorithms

We briefly discuss how we obtain incrementally verifiable computation for CONGEST algorithms. Importantly, in the delegation schemes which exist for this model [2, 3], the verification takes one communication round between the nodes, and the proof is considered accepted if and only if all nodes accept. For this reason, our UpBARG does not trivially apply to this model, so we cannot simply apply our lifting theorem to the existing constructions. To resolve this, instead of using the existing delegation schemes to construct IVC directly, we first show that the existing delegation schemes can be transformed into delegation schemes for a different model which verification does not include communication. Instead, the nodes have access to labels which are assigned to their edges. In this model, we can apply our theorem as UpBARGs are easily implemented there. Finally, we make some observations about the verifier communication in the distributed delegation scheme of [3], which allow us to use only one round of communication to verify many computation rounds.

4 Updatable Hash-and-BARGs (UpBARGs)

In this section we define and construct an updatable hash-and-BARG scheme, with and without consistency checks. Since the syntax and definition of the two versions mostly overlap, we define the notion once, and give one construction, where all of the parts which are only relevant to the version with consistency checks appear in gray. Specifically, in the syntax some inputs and outputs are obligatory and relevant only for the consistency checks version and in the definition, some properties are extended for the consistency checks version.

Syntax. An UpBARG is associated with a recursive hash family with local opening

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

and consists of the following algorithms.

- $\mathsf{Gen}(1^\lambda)$: A randomized setup algorithm that takes as input a security parameter 1^λ, It outputs a common reference string crs.
- $\mathsf{TGen}(1^\lambda, i) \to (\mathsf{crs}, \mathsf{td})$: A randomized setup trapdoor algorithm that takes as input a security parameter 1^λ, and an index i. It outputs a common reference string crs, and a trapdoor td.
- $\mathcal{U}(\mathsf{hk}, \mathsf{crs}, \mathcal{M}, H, \Pi, x, w, \mathcal{C}, x^-) \to (H^*, \Pi^*)$: A deterministic, polynomial-time update algorithm that takes as input a hash key hk, a common reference string crs, a machine $\mathcal{M}$, a hash value H, a proof Π, and a new statement x and witness w. Optionally, for the consistency checks version, it also takes a consistency checking machine $\mathcal{C}$, and a previous statement x^-. It outputs a hash value H and a new proof Π^*.
- $\mathcal{V}(\mathsf{hk}, \mathsf{crs}, \mathcal{M}, H, \Pi, \mathcal{C}, x^-) \to b \in \{0,1\}$: A deterministic, polynomial-time verifier algorithm that takes a hash key hk, a common reference string crs, a machine $\mathcal{M}$, a hash value H, and proof Π. Optionally, for the consistency checking version, it also takes a consistency check machine $\mathcal{C}$ and a last statement x^-. It outputs an acceptance bit b.

- $\mathcal{E}(\mathsf{hk}, \mathsf{td}, \mathcal{M}, H, \Pi, \mathcal{C}) \to (x, \rho, w, x^-, \rho^-)$: A deterministic, polynomial-time extraction algorithm that takes as input a hash key hk, a trapdoor td, a machine $\mathcal{M}$, a hash value H, and a proof Π. Optionally, for the consistency checks version, it also takes a consistency check machine $\mathcal{C}$. It outputs a statement x, an opening ρ, and a proof π. Optionally, in the consistency checks version, it also outputs a previous statement x^- and opening ρ^-.

▶ **Definition 2** (UpBARG). *An updatable batch argument satisfies the following properties.*

Incremental Completeness. *For any $\lambda \in \mathbb{N}$, machines $\mathcal{M}$, and $\mathcal{C}$, hash key hk in the support of $\mathsf{MT.Gen}(1^\lambda)$ and crs in the support of $\mathsf{Gen}(1^\lambda)$:*

- *The verifier always accepts an empty proof: $\mathcal{V}(\mathsf{hk}, \mathsf{crs}, \mathcal{M}, \varepsilon, \varepsilon, \mathcal{C}, \varepsilon) = 1$.*
- *For every $k < 2^\lambda$, a hash set H such that $\mathsf{MT.size}(H) = k$, proof Π, new statement x and witness w, and for the consistency checks version, a consistency checks machine $\mathcal{C}$ and last statement x^-, if $\mathcal{V}(\mathsf{hk}, \mathsf{crs}, \mathcal{M}, H, \Pi, \mathcal{C}, x^-) = 1$, $\mathcal{C}(x^-, x) = 1$, and $\mathcal{M}(x, w) = 1$, then $\mathcal{V}(\mathsf{hk}, \mathsf{crs}, \mathcal{M}, H^*, \Pi^*, \mathcal{C}, x) = 1$.*

Efficiency and Succinctness. *In the completeness experiment above, after applying $\mathcal{U}$ for $k < 2^\lambda$ times with instance-witness pairs $(x_1, w_1), \ldots, (x_k, w_k)$:*

- *The size of the reference string crs and the size of the hash value H^* are both $\mathrm{poly}(\lambda)$, and the size of the proof Π^* is $O(m) + \mathrm{poly}(\lambda)$, where m is the maximal size of an instance x_i or a witness w_i for $i \in [k]$.*
- *The verification algorithm runs in time $(|\mathcal{M}| + |\mathcal{C}|) \cdot \mathrm{poly}(\lambda, |x|, |H^*|, |\Pi^*|)$.*

Index hiding. *For every poly-size adversary $\mathcal{A}$, there exists a negligible function $\mathrm{negl}(\cdot)$ such that for every $\lambda \in \mathbb{N}$ and index $i \le 2^\lambda$,*

$$\left| \Pr\left[\mathcal{A}(\mathsf{hk}, \mathsf{crs}) = 1 \;\middle|\; \begin{array}{l} \mathsf{hk} \leftarrow \mathsf{MT.Gen}(1^\lambda) \\ \mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda) \end{array} \right] - \Pr\left[\mathcal{A}(\mathsf{hk}, \mathsf{crs}) = 1 \;\middle|\; \begin{array}{l} \mathsf{hk} \leftarrow \mathsf{MT.Gen}(1^\lambda) \\ \mathsf{crs}, \mathsf{td} \leftarrow \mathsf{TGen}(1^\lambda, i) \end{array} \right] \right| \le \mathrm{negl}(\lambda).$$

Somewhere Consistent Argument of Knowledge. *For any poly-size adversary $\mathcal{A}$, and polynomials $t_1 = t_1(\lambda)$, $t_2 = t_2(\lambda)$, $k = k(\lambda)$, for the experiments $\mathsf{Exp}_0, \ldots, \mathsf{Exp}_{k-1}$, defined as follows:*

$$\mathsf{Exp}_i = \left\{ \begin{array}{l} (\mathsf{hk}, \mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{TGen}(1^\lambda, i) \\ (\mathcal{M}, H, \Pi, C, x^-) \leftarrow \mathcal{A}(\mathsf{hk}, \mathsf{crs}) \\ (x, \rho, w, x', \rho^-) \leftarrow \mathcal{E}(\mathsf{hk}, \mathsf{td}, \mathcal{M}, H, \Pi, \mathcal{C}) \end{array} \right\},$$

there exists a negligible function $\mathrm{negl}(\cdot)$ such that

$$\Pr_{\mathsf{Exp}_0}\left[\begin{array}{l} H \neq \varepsilon \;\wedge\; \mathsf{MT.size}(H) \le k(\lambda) \\ \mathcal{V}(\mathsf{hk}, \mathsf{crs}, \mathcal{M}, H, \Pi, \mathcal{C}, x^-) = 1 \\ \wedge \left(\begin{array}{l} \mathcal{M}_{t_1}(x, w) = 0 \\ \vee\; \mathcal{C}_{t_2}(\varepsilon, x) = 0 \\ \vee\; \mathsf{MT.Verify}(\mathsf{hk}, H, 0, x, \rho) = 0 \end{array} \right) \end{array} \right] \le \mathrm{negl}(\lambda),$$

and for every $\lambda \in \mathbb{N}$ and index $i^* = i^*(\lambda) \in [k-1]$,

$$\Pr_{\mathsf{Exp}_{i^*}}\left[\begin{array}{l} H \neq \varepsilon \;\wedge\; \mathsf{MT.size}(H) \le k(\lambda) \\ \mathcal{V}(\mathsf{hk}, \mathsf{crs}, \mathcal{M}, H, \Pi, \mathcal{C}, x^-) = 1 \\ \wedge \left(\begin{array}{l} \mathcal{M}_{t_1}(x, w) = 0 \\ \vee\; \mathcal{C}_{t_2}(x', x) = 0 \\ \vee\; \mathsf{MT.Verify}(\mathsf{hk}, H, i^*, x, \rho) = 0 \\ \vee\; \mathsf{MT.Verify}(\mathsf{hk}, H, i^* - 1, x', \rho^-) = 0 \end{array} \right) \end{array} \right] \le \mathrm{negl}(\lambda).$$

where for a machine M, input $x \in \{0,1\}^*$, and $t \in \mathbb{N}$, $M_t(x) = 1$ if and only if M accepts x within t steps.

Last-Statement Collision-Resistance. *In the version with consistency chekcs, for any poly-size adversary $\mathcal{A}$, and polynomial $k = k(\lambda)$, for the following experiment:*

$$\mathsf{Exp} = \left\{ \begin{array}{l} \mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda) \\ \mathsf{hk} \leftarrow \mathsf{MT.Gen}(1^\lambda) \\ (\mathcal{M}, H, \Pi, \mathcal{C}, x_1^-, x_2^-, \rho) \leftarrow \mathcal{A}(\mathsf{hk}, \mathsf{crs}) \end{array} \right\}$$

there exists a negligible function $\mathrm{negl}(\cdot)$ *such that for every* $\lambda \in \mathbb{N}$*:*

$$\Pr_{\mathsf{Exp}} \left[\begin{array}{l} \mathsf{MT.size}(H) = k(\lambda) \\ \wedge\ \mathcal{V}(\mathsf{hk}, \mathsf{crs}, \mathcal{M}, H, \Pi, \mathcal{C}, x_1^-) = 1 \\ \wedge\ x_1^- \neq x_2^- \\ \wedge\ \mathsf{MT.Verify}(\mathsf{hk}, H, k-1, x_2^-, \rho) = 1 \end{array} \right] \leq \mathrm{negl}(\lambda).$$

▶ **Theorem 3.** *Updatable hash-and-*BARG *exist assuming either: (1) Polynomial hardness of LWE; (2) Subexponential hardness of DDH; (3) Polynomial hardness of DDH and k-Lin.*

4.1 Construction from Rate-1 BARGs and Recursive Hash Families

Let $\mathsf{BARG} = (\mathsf{BARG.Gen}, \mathsf{BARG}.\mathcal{P}, \mathsf{BARG}.\mathcal{V}, \mathsf{BARG}.\mathcal{E})$ be a rate-1 somewhere extractable batch argument for NP, and let $\mathcal{H} = \{\mathcal{H}_\lambda\}_{\lambda \in \mathbb{N}}$ a collision-resistant hash family ensemble. Let $\mathsf{MT} = (\mathsf{MT.Gen}, \mathsf{MT.Hash}, \mathsf{MT.Open}, \mathsf{MT.Verify})$ be the recursive hash family with local opening which is the result of constructing Merkle forests from the ensemble $\mathcal{H}$; that is:

- $\mathsf{MT.Gen}(1^\lambda)$ randomly chooses $\mathsf{hk} \leftarrow \mathcal{H}_\lambda$.
- $\mathsf{MT.Hash}(\mathsf{hk}, x_1, \ldots, x_n)$ uses hk to construct a Merkle forest over $x_1, \ldots, x_n$, that is, a set of Merkle trees of descending heights. It outputs the list of roots of trees, each coupled with its height.
- $\mathsf{MT.Open}(\mathsf{hk}, x_1, \ldots, x_n, i)$ returns the opening path from the respective tree in the forest.
- $\mathsf{MT.Verify}(\mathsf{hk}, \mathsf{Val}, x_i, i, \rho)$, finds the respective root val in Val using i and then checks there that r opens to x_i in the respective location using the opening path ρ.

In what follows we describe the construction of an updatable hash-and-BARG scheme $(\mathsf{Gen}, \mathsf{TGen}, \mathcal{U}, \mathcal{V}, \mathcal{E})$.

Moreover, we denote by $\mathsf{MT.L(Val)}$ the set of all heights of roots in the set Val, and by $\mathsf{MT.size(Val)}$ the size of the hashed content in the set Val: $\sum_{\ell \in \mathsf{L(Val)}} 2^\ell$.

$\mathsf{Gen}(1^\lambda)$.

1. For every $\ell \in [\lambda]$, set $\mathsf{crs}_\ell \leftarrow \mathsf{BARG.Gen}(1^\lambda)$. Set $\mathsf{crs} = (\mathsf{crs}_1, \ldots, \mathsf{crs}_\ell)$.

$\mathsf{TGen}(1^\lambda, i)$.

1. Let $i_1 ... i_\lambda$ be i's bit representation, from least to most significant bit. For every $\ell \in [\lambda]$, set $(\mathsf{crs}_\ell, \mathsf{td}_\ell) \leftarrow \mathsf{BARG.TGen}(1^\lambda, m, i_\ell)$. Set $\mathsf{crs} = (\mathsf{crs}_1, \ldots, \mathsf{crs}_\lambda)$ and $\mathsf{td}' = (\mathsf{td}_1, \ldots, sk_\lambda)$. Set $\mathsf{td} = (\mathsf{crs}, i, \mathsf{td}')$.

We now define a series of verifiers: $\{\mathcal{V}^\ell\}_\ell$, using a series of helper, parametrised machines $\{\mathcal{M}^\ell\}_\ell$, and a helper algorithm Merge, all of which will take part in the following UpBARG algorithms $\mathcal{U}, \mathcal{V}, \mathcal{E}$.

$\mathcal{V}^0(\mathsf{hk}, \mathcal{M}, h, w^*, \mathcal{C})$.

1. Parse: $w^* = (x, w)$. Parse $w^* = (x, \pi, h^-, \ell^-) = (x, (w, x^-, \rho^-), h^-, \ell^-)$.
2. Verify: $h = \mathcal{H}_\lambda(\mathsf{hk}, x)$.
3. Verify: $\mathcal{M}(x, w) = 1$.
4. Verify: $\mathcal{C}(x^-, x) = 1$.
5. If $\ell^- = \varepsilon$, verify $h^- = x^- = \rho^- = \varepsilon$.
6. If $\ell^- \neq \varepsilon$, verify $\mathsf{MT.Verify}(\mathsf{hk}, h^-, 2^{\ell^-} - 1, x^-, \rho^-, \ell^-) = 1$.
7. Accept if and only if all checks pass.

$\mathcal{M}^\ell[\mathsf{hk}, \mathsf{crs}, \mathcal{M}, x, \mathcal{C}, h^-, \ell^-](i, \widetilde{w})$ for $\ell > 0$.

1. Parse $x = (h_0, h_1)$.
2. If $\ell = 1$, verify $\mathcal{V}^0(\mathsf{hk}, \mathcal{M}, h_i, \widetilde{w}, \mathcal{C}) = 1$.
   Otherwise, verify $\mathcal{V}^{\ell-1}(\mathsf{hk}, \mathsf{crs}, \mathcal{M}, h_i, \widetilde{w}, \mathcal{C}) = 1$.
3. Parse $\widetilde{w} = (x', \pi', h', \ell')$.
4. If $i = 0$, verify $h' = h^-$ and $\ell' = \ell^-$.
5. If $i = 1$, Verify $h' = h_0$, and $\ell' = \ell - 1$.
6. Accept if and only if all checks pass.

$\mathcal{V}^\ell(\mathsf{hk}, \mathsf{crs}, \mathcal{M}, h, w^*, \mathcal{C})$ for $\ell > 0$.

1. Parse $\mathsf{crs} = (\mathsf{crs}_1, \ldots, \mathsf{crs}_\ell)$, and set $\mathsf{crs}' = (\mathsf{crs}_1, \ldots, \mathsf{crs}_{\ell-1})$ (for $\ell = 1$, set $\mathsf{crs}' = \varepsilon$).
2. Parse: $w^* = (x, \pi, h^-, \ell^-)$, and $x = (h_0, h_1)$.
3. Verify: $h = \mathcal{H}_\lambda(\mathsf{hk}, x)$.
4. Let $\mathcal{M}^\ell = \mathcal{M}^\ell[\mathsf{hk}, \mathsf{crs}', \mathcal{M}, x, \mathcal{C}, h^-, \ell^-]$. Verify $\mathsf{BARG}.\mathcal{V}(\mathsf{crs}_\ell, \mathcal{M}^\ell, \pi) = 1$.

$\mathsf{Merge}(\mathsf{hk}, \mathsf{crs}, \mathcal{M}, H, \Pi, \mathcal{C}, h^-, \ell^-, \rho^{\mathsf{start}}) \to (H, \Pi, \rho^{\mathsf{start}})$.

1. Parse $\mathsf{crs} = (\mathsf{crs}_1, \ldots, \mathsf{crs}_\lambda)$.
2. $\rho^+ = \varepsilon$.
3. While there exist $\{(h_i, \ell_i)\}_{i \in \{0,1\}} \subseteq H$ such that $\ell_0 = \ell_1$:[4]
   a. Set w_0, w_1 to be such that $(w_0, \ell_0) \in \Pi$ and $(w_1, \ell_1) \in \Pi$.
   b. Set $x = (h_0, h_1)$ and $h = \mathcal{H}(\mathsf{hk}, x)$.
   c. Set $\ell = \ell_0 + 1$.
   d. Set $\mathsf{crs}' = (\mathsf{crs}_1, \ldots, \mathsf{crs}_{\ell-1})$
   e. Let $\mathcal{M}^\ell = \mathcal{M}^\ell[\mathsf{hk}, \mathsf{crs}', \mathcal{M}, x, \mathcal{C}, h^-, \ell^-]$.
   f. Set $\pi = \mathsf{BARG}.\mathcal{P}(\mathsf{crs}_\ell, \mathcal{M}^\ell, (w_0, w_1))$.
   g. Parse $w_0 = (x_0, \pi_0, h_0^-, \ell_0^-)$.
   h. Set $w^* = (x, \pi, h_0^-, \ell_0^-)$.
   i. Set $\rho^+ = h_0 \parallel \rho^+$.
   j. If $\ell_0 = \max \mathsf{MT.L}(H)$, set $\rho^{\mathsf{start}} = h_1 \parallel \rho^{\mathsf{start}}$.
   k. Set $\Pi = (\Pi \cup \{(w^*, \ell)\}) \setminus \{(w_i, \ell_i)\}_{\ell \in \{0,1\}}$.
   l. Set $H = (H \cup \{(h, \ell)\}) \setminus \{(h_i, \ell_i)\}_{\ell \in \{0,1\}}$.
4. Output $(H, \Pi, \rho^{\mathsf{start}})$.

We continue to define the remaining algorithms $\mathcal{U}, \mathcal{V}, \mathcal{E}$.

[4] We assume here implicitly that the newer tuple is (h_1, ℓ_1) and the older one is (h_0, ℓ_0). This could be made formal with more notation, but we decided to omit it for the sake of simplicity and readability.

$\mathcal{U}(\mathsf{hk}, \mathsf{crs}, \mathcal{M}, \Pi, x, w, \mathcal{C}, x^-)$.

1. Parse $\mathsf{crs} = (\mathsf{crs}_1, \ldots, \mathsf{crs}_\lambda)$.
2. Set $h = \mathcal{H}(\mathsf{hk}, x)$.
3. If $\Pi \neq \varepsilon$, parse $\Pi = (\Pi', \mathsf{aux})$, $\mathsf{aux} = (h^-, \rho^-, \ell^-, \rho^{\mathsf{start}}, x^{\mathsf{start}})$, and set $\Pi = \Pi'$. Otherwise, set $h^- = \rho^- = \ell^- = \rho^{\mathsf{start}} = \varepsilon$, and $x^{\mathsf{start}} = x$.
4. Set $w^* = (x, w)$. Set $\pi = (w, x^-, \rho^-)$, and $w^* = (x, \pi, h^-, \ell^-)$.
5. Set $H = H \cup \{(h, \ell)\}$, and $\Pi = \Pi \cup \{(w^*, \ell)\}$.
6. Set $(H^*, \Pi^*, \rho^+, \rho^{\mathsf{start}*}) = \mathsf{Merge}(\mathsf{hk}, \mathsf{crs}, \mathcal{M}, H, \Pi, \mathcal{C}, h^-, \ell^-, \rho^{\mathsf{start}})$.
7. Set $\ell^+ = \min \mathsf{MT.L}(H)$, and set h^+ to be s.t $(h^+, \ell^+) \in H$.
8. Set $\mathsf{aux}^* = (h^+, \rho^+, \ell^+, x^{\mathsf{start}}, \rho^{\mathsf{start}*})$.
9. Set $\Pi^* = (\Pi^*, \mathsf{aux}^*)$
10. Output (H^*, Π^*).

$\mathcal{V}(\mathsf{hk}, \mathsf{crs}, \mathcal{M}, H, \Pi, \mathcal{C}, x^-)$.

1. Parse $\mathsf{crs} = (\mathsf{crs}_1, \ldots, \mathsf{crs}_\lambda)$, and for every $\ell \in [\lambda]$, set $\mathsf{crs}^\ell = (\mathsf{hk}, \mathsf{crs}_1, \ldots, \mathsf{crs}_\ell)$.
2. If either of the following is true, then verify all of them are true, and if so accept and finish: (1) $H = \varepsilon$, (2) $\Pi = \varepsilon$ and (3) $x^- = \varepsilon$. In any other case, continue.
3. Parse $\Pi = (\Pi', \mathsf{aux})$, and $\mathsf{aux} = (h^-, \rho^-, \ell^-, x^{\mathsf{start}}, \rho^{\mathsf{start}})$, and set $\Pi = \Pi'$.
4. Verify that $\{\ell \mid \exists w : (w, \ell) \in \Pi\} = \mathsf{MT.L}(H)$, and that for every $(h_1, \ell_1), (h_2, \ell_2) \in H$, we have $\ell_1 \neq \ell_2$.
5. For every $\ell \in \mathsf{MT.L}(H)$, for h, w such that $(h, \ell) \in H$ and $(w, \ell) \in \Pi$, verify: $\mathcal{V}^\ell(\mathsf{hk}, \mathsf{crs}^\ell, \mathcal{M}, h, w, \mathcal{C}) = 1$ (For $\ell = 0$, omit crs from the input).
6. Verify $\mathsf{MT.Verify}(\mathsf{hk}, H, 0, x^{\mathsf{start}}, \rho^{\mathsf{start}}) = 1$.
7. Verify $\mathcal{C}(\varepsilon, x^{\mathsf{start}}) = 1$.
8. Verify that $\ell^- = \min \mathsf{MT.L}(H)$ and that $(h^-, \ell^-) \in \mathsf{MT.L}(H)$.
9. Let $k = \sum_{\ell \in \mathsf{MT.L}(H)} 2^\ell$. Verify $\mathsf{MT.Verify}(\mathsf{hk}, H, k-1, x^-, \rho^-) = 1$.
10. Accept if all checks pass.

$\mathcal{E}(\mathsf{td}, \mathsf{hk}, \mathcal{M}, H, \Pi, \mathcal{C})$.

1. Parse $\mathsf{td} = (\mathsf{crs}, i, \mathsf{td}')$, $\mathsf{crs} = (\mathsf{crs}_1, \ldots, \mathsf{crs}_\lambda)$, and $\mathsf{td}' = (\mathsf{td}_1, \ldots, \mathsf{td}_\lambda)$.
2. Let $i_0 \ldots i_{\lambda-1}$ be i's bit representation, from least significant bit to most significant bit.
3. Set $\ell^* \in \mathsf{MT.L}(H)$ to be such that

$$\sum_{\ell \in \mathsf{MT.L}(H) \ \wedge \ \ell > \ell^*} 2^\ell - 1 < i \leq \sum_{\ell \in \mathsf{MT.L}(H) \ \wedge \ \ell \geq \ell^*} 2^\ell - 1.$$

4. Parse $\Pi = (\Pi', \mathsf{aux})$, and set $\Pi = \Pi'$.
5. Set h, w^* to be such that $(h, \ell^*) \in H$ and $(w^*, \ell^*) \in \Pi$.
6. Set $\rho = \varepsilon$. Set $\rho^- = \varepsilon$.
7. Set $\ell = \ell^*$.
8. If $i > 0$, let ℓ^- be the index of the most significant bit where i and $i-1$ disagree.
9. While $\ell > 0$:
   a. Parse $w^* = (x, \pi, h^-, \ell')$, and $x = (h_0, h_1)$.
   b. Set $\mathsf{crs}' = (\mathsf{crs}_1, \ldots, \mathsf{crs}_{\ell-1})$, and let $\mathcal{M}^\ell = \mathcal{M}^\ell[\mathsf{hk}, \mathsf{crs}', \mathcal{M}, x, \mathcal{C}, h^-, \ell']$. Set $w^* = \mathsf{BARG}.\mathcal{E}(\mathsf{td}_\ell, \mathcal{M}^\ell, \pi)$.
   c. Set $\ell = \ell - 1$.
   d. Set $\rho = h_{1-i_\ell} \parallel \rho$.
   e. If $i > 0$, $\ell > \ell^-$, set $\rho^- = h_{1-i_\ell} \parallel \rho^-$. If $\ell = \ell^-$, set $\rho^- = h_{i_\ell} \parallel \rho^-$.

10. Parse $\pi = (w, x^-, \rho')$.
11. Set $\rho^- = \rho' \parallel \rho^-$.
12. Output (x, ρ, w) additionally, (x^-, ρ^-).

5 One-step Delegation to IVC via UpBARGs

In this section we show how to use UpBARG to lift a delegation scheme for one step into an IVC, using our UpBARG, in a generic manner: for deterministic computational models $\mathbb{P}$ and $\mathbb{V}$, we define and construct incrementally verifiable $\mathbb{V}$-$\mathbb{P}$ delegation scheme, in which computations in the model $\mathbb{P}$ are incrementally proven using a $\mathbb{P}$-algorithm to a verifier runs which is in a $\mathbb{V}$-algorithm.

Below we formally define our notion of $\mathbb{V}$-$\mathbb{P}$ delegation scheme. We then present our construction.

5.1 Incrementally Verifiable $\mathbb{V}$-$\mathbb{P}$ Delegation.

Syntax. Incrementally Verifiable $\mathbb{V}$-$\mathbb{P}$ Delegation consists of the following algorithms:

- $\mathsf{Gen}(1^\lambda) \to \mathsf{crs}$: A randomized setup algorithm that takes as input a security parameter 1^λ and outputs a common reference string crs.
- $\mathsf{Digest}(\mathsf{crs}, \mathsf{cf}) \to h$: A $\mathbb{P}$-algorithm that takes a reference string crs, and a configuration $\mathsf{cf} \in \mathsf{CF}[\mathbb{P}]$. It outputs digest h.
- $\mathsf{LblDigest}(\mathsf{crs}, \mathsf{lbl}) \to H$: A $\mathbb{P}$-algorithm that takes a reference string crs and a label $\mathsf{lbl} \in \mathsf{Lbl}[\mathbb{P}]$. It outputs digest hlbl.
- $\mathcal{U}(\mathsf{crs}, \mathcal{M}, \mathsf{cf}_0, \mathsf{cf}_t, \mathsf{lbl}_t, t, H_t, \Pi_t) \to \mathsf{cf}_{t+1}, H_{t+1}, \Pi_{t+1}$: A $\mathbb{P}$-algorithm that takes a reference string crs, a a $\mathbb{P}$-algorithm $\mathcal{M}$, an initial configuration cf_0, a current configuration cf_t, a label lbl_t, a number of steps t, a label digest H_t, and a proof Π_t. It outputs a new configuration cf_{t+1}, a new label digest H_{t+1}, and a new proof Π_{t+1}.
- $\mathcal{V}(\mathsf{crs}, \mathcal{M}, h, H_t, h', t, \Pi_t) \to b$: A $\mathbb{V}$ algorithm that takes a reference string crs, a $\mathbb{P}$-algorithm $\mathcal{M}$, an initial digest h, a possibly empty hash of labels H_t, a final digest h', a number of steps t, and a proof Π_t. It outputs an acceptance bit b.

▶ **Definition 4** ($\mathbb{V}$-$\mathbb{P}$ Incrementally verifiable Delegation). *A $\mathbb{V}$-$\mathbb{P}$ incrementally verifiable delegation scheme satisfies the same collision resistance succinctness, and soundness properties as a (non-incremental) $\mathbb{V}$-$\mathbb{P}$ delegation scheme. In addition, it satisfies the following incremental completeness property: For any $\lambda \in \mathbb{N}$, $\mathbb{P}_1$ algorithm $\mathcal{M}$ and crs in the support of $\mathsf{Gen}(1^\lambda)$:*

- *The verifier accepts the 0-steps statement with an empty proof. For every hash value h: $\mathcal{V}(\mathsf{crs}, \mathcal{M}, h, \varepsilon, h, 0, \varepsilon) = 1$.*
- *For every $t < 2^\lambda$, initial digest h_0, configuration cf_t, label lbl_t, hash of labels H_t and proof Π_t, we have that for $\mathsf{h}_t \leftarrow \mathsf{Digest}(\mathsf{crs}, \mathsf{cf}_t)$, if $\mathcal{V}(\mathsf{crs}, \mathcal{M}, h_0, H_t, h_t, t, \Pi_t) = 1$, then $\mathcal{V}(\mathsf{crs}, \mathcal{M}, h_0, H_{t+1}, h_{t+1}, t+1, \Pi_{t+1}) = 1$ where $h_{t+1} = \mathsf{Digest}(\mathsf{crs}, \mathsf{cf}_{t+1})$ and $(\mathsf{cf}_{t+1}, H_{t+1}, \Pi_{t+1}) = \mathcal{U}(\mathsf{crs}, \mathcal{M}, \mathsf{cf}_t, \mathsf{lbl}_t, H_t, \Pi_t)$.*

5.2 Construction

In our construction, we use a $\mathbb{V}$-$\mathbb{P}$ delegation scheme, and a $\mathbb{V}$-$\mathbb{P}$ UpBARG, defined below.

▶ **Definition 5** ($\mathbb{V}$-$\mathbb{P}$ UpBARG). *A $\mathbb{V}$-$\mathbb{P}$ UpBARG has the syntax of an updatable hash-and-BARG (see Definition 2) and satisfies the same properties, with the following generalizations:*

- *In the input to $\mathcal{U}, \mathcal{V}$, instead of being Turing machines, $\mathcal{M}, \mathcal{C}$ are $\mathbb{P}$-algorithms, which can also take labels in their input if $\mathbb{P}$ is a model of labeled algorithms.*

- $\mathcal{P}$ *is a* $\mathbb{P}$ *algorithm and* $\mathcal{V}$ *is a* $\mathbb{V}$ *algorithm.*
- *In the associated hash family* MT, $\mathsf{MT.Hash}$ *is a* $\mathbb{P}$*-algorithm and* $\mathsf{MT.Verify}$ *is a* $\mathbb{V}$*-algorithm.*

Moreover, we use an extension to the usual syntax of hash family with local opening which allows us to hash different vectors separately using the same invocation, that is: $\mathsf{MT.Hash}(\mathsf{hk}, (x_1, \ldots, x_k), (y_1, \ldots, y_k)) = (h_x, h_y)$. Note that using simply a pair of Merkle trees instead of one allows for such hash families with all the properties of hash families with local opening.

Let $\mathsf{UpBARG} = (\mathsf{UpBARG.Gen}, \mathsf{UpBARG.TGen}, \mathsf{UpBARG}.\mathcal{U}, \mathsf{UpBARG}.\mathcal{V}, \mathsf{UpBARG}.\mathcal{E})$ be an updatable hash-and-BARG scheme for NP , associated with the recursive hash family with local openings $\mathsf{MT} = (\mathsf{MT.Gen}, \mathsf{MT.Hash}, \mathsf{MT.Open}, \mathsf{MT.Verify})$, which in turn admits the extended syntax mentioned above. Let $\mathsf{Del} = (\mathsf{Del.Gen}, \mathsf{Del.Digest}, \mathsf{Del.LblDigest}, \mathsf{Del}.\mathcal{P}, \mathsf{Del}.\mathcal{V})$ be a one-step $\mathbb{V}$-$\mathbb{P}$ delegation scheme. For our construction we use the extended syntax for hash sets of the MT family, and the extended syntax for UpBARG that is extractable in 2 locations. As with regular BARGs, this could be achieved for UpBARGs by doubling the proof and CRS size.

We first define for every $\mathbb{P}$-algorithm $\mathcal{M}$, reference string crs and hash value h two helper $\mathbb{P}$-algorithms, $\mathcal{C}[\mathcal{M}, \mathsf{crs}, h]$ and $\mathcal{M}'[\mathcal{M}, \mathsf{crs}]$, and then continue to define the scheme's algorithms.

Helper algorithms $\mathcal{C}[\mathcal{M}, \mathsf{crs}, h]$ and $\mathcal{M}'[\mathcal{M}, \mathsf{crs}]$.

- $\mathcal{C}[\mathcal{M}, \mathsf{crs}, h](x_1, x_2)$: If $x_1 = \varepsilon$, parse $x_2 = (h_2, h_2')$ and accept if and only if $h_2 = h$. Otherwise, Parse $x_1 = (h_1, h_1')$ and $x_2 = (h_2, h_2')$, and verify that $h_1' = h_2$.
- $\mathcal{M}'[\mathcal{M}, \mathsf{crs}](x, w)$: Parse $(x, x) = (((h_1, h_2), \mathsf{hlbl}), \pi)$, and verify $\mathsf{Del}.\mathcal{V}(\mathsf{crs}, \mathcal{M}, h_1, \mathsf{hlbl}, h_2, 1, \pi) = 1$.

$\mathsf{Gen}(1^\lambda)$.

1. $\mathsf{hk} = \mathsf{MT.Gen}(1^\lambda)$.
2. $\mathsf{crs}_{\mathsf{BA}} = \mathsf{UpBARG.Gen}^2(1^\lambda)$.
3. $\mathsf{crs}_{\mathsf{Del}} = \mathsf{Del.Gen}(1^\lambda)$.
4. Output $\mathsf{crs} = (\mathsf{hk}, \mathsf{crs}_{\mathsf{BA}}, \mathsf{crs}_{\mathsf{Del}})$.

$\mathcal{U}(\mathsf{crs}, \mathcal{M}, \mathsf{cf}_0, \mathsf{cf}_t, \mathsf{lbl}_t, t, H_t, \Pi_t) \to \mathsf{cf}_{t+1}, H_{t+1}, \Pi_{t+1}$.

1. Parse $\mathsf{crs} = (\mathsf{hk}, \mathsf{crs}_{\mathsf{BA}}, \mathsf{crs}_{\mathsf{Del}})$.
2. Parse $\Pi = (H_{\mathsf{BA}}, \Pi_{\mathsf{BA}}, x^-)$. (If $\Pi = \varepsilon$, set $\Pi_{\mathsf{BA}} = H_{\mathsf{BA}} = x^- = \varepsilon$.)
3. Set $\mathsf{cf}_{t+1} = \mathcal{M}_1(\mathsf{cf}_t, \mathsf{lbl}_t)$.
4. Set $h_{t+1} = \mathsf{Del.Digest}(\mathsf{crs}_{\mathsf{Del}}, \mathsf{cf}_{t+1})$.
5. Set $\mathsf{hlbl}_{t+1} = \mathsf{Del.LblDigest}(\mathsf{crs}_{\mathsf{Del}}, \mathsf{lbl}_t)$.
6. Set $x = ((h_t, h_{t+1}), \mathsf{hlbl}_t)$.
7. Set $\pi = \mathsf{Del}.\mathcal{P}(\mathsf{crs}_{\mathsf{Del}}, \mathcal{M}, \mathsf{cf}_t, 1)$.
8. Set $h_0 = \mathsf{Del.Digest}(\mathsf{crs}_{\mathsf{Del}}, \mathsf{cf}_0)$.
9. Set $(H'_{\mathsf{BA}}, \Pi'_{\mathsf{BA}}) = \mathsf{UpBARG}.\mathcal{U}^2(\mathsf{hk}, \mathsf{crs}_{\mathsf{BA}}, \mathcal{M}'[\mathcal{M}, \mathsf{crs}_{\mathsf{Del}}], H_{\mathsf{BA}}, \Pi_{\mathsf{BA}}, x, \pi, \mathcal{C}[\mathcal{M}, \mathsf{crs}_{\mathsf{Del}}, h_0], x^-)$.
10. Parse $H'_{\mathsf{BA}} = (H', H_{t+1})$.
11. Set $\Pi_{t+1} = (H'_{\mathsf{BA}}, \Pi'_{\mathsf{BA}}, x)$.
12. Output $(\mathsf{cf}_{t+1}, H_{t+1}, \Pi_{t+1})$.

$\mathcal{V}(\mathsf{crs}, \mathcal{M}, h_0, H_t, h_t, t, \Pi_t) \to b$.

1. Parse $\mathsf{crs} = (\mathsf{hk}, \mathsf{crs}_{\mathsf{BA}}, \mathsf{crs}_{\mathsf{Del}})$.
2. If $\Pi = \varepsilon$, verify that $h_0 = h_t$ and that $t = 0$, and finish. Otherwise, continue.
3. Parse $\Pi_t = (H_{\mathsf{BA}}, \Pi_{\mathsf{BA}}, x^-)$.
4. Parse $H_{\mathsf{BA}} = (H', H'_t)$, and verify that $H_t = H'_t$.
5. $x^- = ((\widetilde{h_{t-1}}, \widetilde{h_t}), \mathsf{hlbl}_t)$ and verify $\widetilde{h_t} = h_t$.
6. Verify $t = \mathsf{MT.size}(H_{\mathsf{BA}})$.
7. Verify $\mathsf{UpBARG}.\mathcal{V}^2(\mathsf{hk}, \mathsf{crs}_{\mathsf{BA}}, \mathcal{M}'[\mathcal{M}, \mathsf{crs}_{\mathsf{Del}}], H_{\mathsf{BA}}, \Pi_{\mathsf{BA}}, \mathcal{C}[\mathcal{M}, \mathsf{crs}_{\mathsf{Del}}, h_0], x^-) = 1$.
8. Accept if all checks pass.

References

1 Abtin Afshar and Rishab Goyal. Verifiable streaming computation and step-by-step zero-knowledge. Cryptology ePrint Archive, Paper 2025/251, 2025. URL: https://eprint.iacr.org/2025/251.

2 Eden Aldema Tshuva, Elette Boyle, Ran Cohen, Tal Moran, and Rotem Oshman. Locally verifiable distributed SNARGs. In *TCC*, pages 65–90. Springer, 2023. doi:10.1007/978-3-031-48615-9_3.

3 Eden Aldema Tshuva and Rotem Oshman. Fully Local Succinct Distributed Arguments. In *DISC*, pages 1:1–1:24. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.DISC.2024.1.

4 Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. *Algorithmica*, 79:1102–1160, 2017. doi:10.1007/S00453-016-0221-0.

5 Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for snarks and proof-carrying data. In *STOC*, pages 111–120, 2013. doi:10.1145/2488608.2488623.

6 Alessandro Chiesa and Eran Tromer. Proof-carrying data and hearsay arguments from signature cards. In *ICS (Vol. 10)*, pages 310–331, 2010. URL: http://conference.iiis.tsinghua.edu.cn/ICS2010/content/papers/25.html.

7 Arka Rai Choudhuri, Sanjam Garg, Abhishek Jain, Zhengzhong Jin, and Jiaheng Zhang. Correlation intractability and snargs from sub-exponential ddh. In *Crypto*, pages 635–668. Springer, 2023. doi:10.1007/978-3-031-38551-3_20.

8 Arka Rai Choudhuri, Abhishek Jain, and Zhengzhong Jin. Non-interactive batch arguments for NP from standard assumptions. In *Annual International Cryptology Conference*, pages 394–423. Springer, 2021. doi:10.1007/978-3-030-84259-8_14.

9 Arka Rai Choudhuri, Abhishek Jain, and Zhengzhong Jin. SNARGs for P from LWE. In *FOCS*, pages 68–79, 2021.

10 Pratish Datta, Abhishek Jain, Zhengzhong Jin, Alexis Korb, Surya Mathialagan, and Amit Sahai. Incrementally verifiable computation for np from standard assumptions. In *Annual International Cryptology Conference*, pages 611–642. Springer, 2025. doi:10.1007/978-3-032-01907-3_20.

11 Lalita Devadas, Rishab Goyal, Yael Kalai, and Vinod Vaikuntanathan. Rate-1 non-interactive arguments for batch-NP and applications. In *FOCS*, pages 1057–1068, 2022. doi:10.1109/FOCS54457.2022.00103.

12 Yuval Emek, Yuval Gil, and Shay Kutten. Locally Restricted Proof Labeling Schemes. In *36th International Symposium on Distributed Computing (DISC 2022)*, volume 246, pages 20:1–20:22, 2022. doi:10.4230/LIPIcs.DISC.2022.20.

13 Dario Fiore and Ida Tucker. Efficient zero-knowledge proofs on signed data with applications to verifiable computation on data streams. In *CCS*, pages 1067–1080, 2022. doi:10.1145/3548606.3560630.

14 Rachit Garg, Kristin Sheridan, Brent Waters, and David J Wu. Fully succinct batch arguments for NP from indistinguishability obfuscation. In *Theory of Cryptography Conference*, pages 526–555. Springer, 2022. doi:10.1007/978-3-031-22318-1_19.

15 Yael Kalai, Alex Lombardi, Vinod Vaikuntanathan, and Daniel Wichs. Boosting batch arguments and RAM delegation. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing (STOC)*, pages 1545–1552, 2023. doi:10.1145/3564246.3585200.

16 Yael Kalai and Omer Paneth. Delegating ram computations. In *TCC*, pages 91–118. Springer, 2016.

17 Yael Tauman Kalai, Omer Paneth, and Lisa Yang. How to delegate computations publicly. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, pages 1115–1124, 2019. doi:10.1145/3313276.3316411.

18 Amos Korman, Shay Kutten, and David Peleg. Proof labeling schemes. In *PODC*, pages 9–18, 2005. doi:10.1145/1073814.1073817.

19 Silvio Micali. Computationally sound proofs. *SIAM Journal on Computing*, 30(4):1253–1298, 2000. doi:10.1137/S0097539795284959.

20 Omer Paneth and Rafael Pass. Incrementally verifiable computation via rate-1 batch arguments. In *FOCS*, pages 1045–1056, 2022. doi:10.1109/FOCS54457.2022.00102.

21 Amit Sahai and Brent Waters. How to use indistinguishability obfuscation: deniable encryption, and more. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 475–484, 2014. doi:10.1145/2591796.2591825.

22 Janwillem Swalens, Lode Hoste, Emad Heydari Beni, and Lieven Trappeniers. zkstream: a framework for trustworthy stream processing. In *International Middleware Conference*, pages 252–265, 2024. doi:10.1145/3652892.3700763.

23 Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In *TCC*, pages 1–18. Springer, 2008. doi:10.1007/978-3-540-78524-8_1.

24 Brent Waters and David J Wu. Batch arguments for NP and more from standard bilinear group assumptions. In *Crypto*, pages 433–463. Springer, 2022. doi:10.1007/978-3-031-15979-4_15.