Fully Succinct Batch Arguments for NP from Indistinguishability Obfuscation

Rachit Garg[1], Kristin Sheridan[1], Brent Waters[1,2], and David J. Wu[1]

[1] University of Texas at Austin, Austin, TX, USA
[2] NTT Research, Sunnyvale, CA, USA

Abstract. Non-interactive batch arguments for NP provide a way to amortize the cost of NP verification across multiple instances. In particular, they allow a prover to convince a verifier of multiple NP statements with communication that scales sublinearly in the number of instances.

In this work, we study *fully succinct* batch arguments for NP in the common reference string (CRS) model where the length of the proof scales not only sublinearly in the number of instances T, but also sublinearly with the size of the NP relation. Batch arguments with these properties are special cases of succinct non-interactive arguments (SNARGs); however, existing constructions of SNARGs either rely on idealized models or strong non-falsifiable assumptions. The one exception is the Sahai-Waters SNARG based on indistinguishability obfuscation. However, when applied to the setting of batch arguments, we must impose an *a priori* bound on the number of instances. Moreover, the size of the common reference string scales linearly with the number of instances.

In this work, we give a *direct* construction of a fully succinct batch argument for NP that supports an unbounded number of statements from indistinguishability obfuscation and one-way functions. Then, by additionally relying on a somewhere statistically-binding (SSB) hash function, we show how to extend our construction to obtain a fully succinct and *updatable* batch argument. In the updatable setting, a prover can take a proof π on T statements $(x_1, \ldots, x_T)$ and "update" it to obtain a proof π' on $(x_1, \ldots, x_T, x_{T+1})$. Notably, the update procedure only requires knowledge of a (short) proof for $(x_1, \ldots, x_T)$ along with a *single* witness w_{T+1} for the new instance x_{T+1}. Importantly, the update does *not* require knowledge of witnesses for $x_1, \ldots, x_T$.

1 Introduction

Non-interactive batch arguments (BARGs) provide a way to amortize the cost of NP verification across multiple instances. Specifically, in a batch argument, the prover has a collection of NP statements $x_1, \ldots, x_T$ and their goal is to convince the verifier that $x_i \in \mathcal{L}$ for all i, where $\mathcal{L}$ is the associated NP language. The trivial solution is to have the prover send over the associated NP witnesses $w_1, \ldots, w_T$ and have the verifier check each one individually. The goal in a batch argument is to obtain *shorter* proofs—namely, proofs whose size scales *sublinearly* in T.

In this work, we operate in the common reference string (CRS) model where we assume that there is a one-time (trusted) sampling of a structured reference string. Within this model, we focus on the setting where where the proof is non-interactive (i.e., the proof consists of a single message from the prover to the verifier) and publicly-verifiable (i.e., verifying the proof only requires knowledge of the associated statements and the CRS). Finally, we require soundness to hold against computationally-bounded provers; namely, our goal is to construct batch *argument* systems. Recently, there has been a flurry of work constructing batch arguments for NP satisfying these requirements from standard lattice assumptions [CJJ21b, DGKV22], assumptions on groups with bilinear maps [WW22], and from a combination of subexponential hardness of the DDH assumption together with the QR assumption [CJJ21a].

This work: fully succinct batch arguments. The size of the proof in the aforementioned BARG constructions all scale linearly with the size of the NP relation. In other words, to check T statements for an NP relation that is computable by a circuit of size s, the proof sizes scale with $\mathsf{poly}(\lambda, s) \cdot o(T)$, where λ is the security parameter. In this work, we study the setting where the proof π scales *sublinearly* in *both* the number of instances T and the size s of the NP relation. More precisely, we require that $|\pi| = \mathsf{poly}(\lambda, \log s, \log T)$, and we refer to batch arguments satisfying this property to be "fully succinct." Our primary goal in this work is to *minimize* the communication cost (and in conjunction, the verifier cost) of batch NP verification.

We note that this level of succinctness is typically characteristic of succinct non-interactive arguments (SNARGs), and indeed any SNARG directly implies a fully succinct batch argument. However, existing constructions of SNARGs either rely on random oracles [Mic95, BBHR18, COS20, CHM+20, Set20], the generic group model [Gro16], or strong non-falsifiable assumptions [Gro10, BCCT12, DFH12, Lip13, PHGR13, GGPR13, BCI+13, BCPR14, BISW17, BCC+17, BISW18, ACL+22]. Indeed, Gentry and Wichs [GW11] showed that no construction of an (adaptively-sound) SNARG for NP can be proven secure via a black-box reduction to a falsifiable assumption [Nao03].

The only construction of (non-adaptively sound) SNARGs from falsifiable assumptions is the construction by Sahai and Waters based on indistinguishability obfuscation ($i\mathcal{O}$) [SW14] in conjunction with the recent breakthrough works of Jain et al. [JLS21, JLS22] that base indistinguishability obfuscation on falsifiable assumptions. However, the Sahai-Waters SNARG from $i\mathcal{O}$ imposes an *a priori* bound on the number of statements that can be proven, and in particular, the size of the CRS grows with the total length of the statement and witness (i.e., the CRS consists of an obfuscated program that reads in the statement and the witness and outputs a signature on the statements if the input is well-formed). When applied to the setting of batch verification, this limitation means that we need to impose an *a priori* bound of the number of instances that can be proved, and the size of the CRS necessarily scales with this bound. Our goal in this work is to construct a fully succinct batch argument for NP that supports an arbitrary

number of instances from indistinguishability obfuscation and one-way functions (i.e., the same assumption as the construction of Sahai and Waters).

An approach using recursive composition. A natural approach to constructing a fully succinct batch argument that supports an arbitrary polynomial number of statements is to compose a SNARG with polylogarithmic verification cost (for a single statement) with a batch argument that supports an unbounded number of statements. Namely, to prove that $(x_1, \ldots, x_T)$ are true, the prover would proceed as follows:

1. First, for each statement $x_i \in \{0,1\}^\ell$, the prover constructs a SNARG proof π_i. If the SNARG has a polylogarithmic verification procedure, then the size of the SNARG verification circuit for checking (x_i, π_i) is bounded by $\mathsf{poly}(\lambda, \ell, \log s)$, where s is the size of the circuit for checking the underlying NP relation.
2. Next, the prover uses a batch argument to demonstrate that it knows $(\pi_1, \ldots, \pi_T)$ where π_i is an accepting SNARG proof on instance $x_i \in \{0,1\}^\ell$. This is a batch argument for checking T instances of the SNARG verification circuit, which has size $\mathsf{poly}(\lambda, \ell, \log s)$. If the size of the batch argument scales polylogarithmically with the number of instances, then the overall proof has size $\mathsf{poly}(\lambda, \ell, \log s, \log T)$.

Moreover, using a somewhere extractable commitment scheme [HW15, CJJ21b], it is possible to remove the dependence on the instance size ℓ.[3] This yields a fully succinct batch argument with proof size $\mathsf{poly}(\lambda, \log s, \log T)$. To argue (non-adaptive) soundness of this approach, we rely on soundness of the underlying SNARG and somewhere extractability of the underlying batch argument (i.e., a BARG where the CRS can be programmed to a specific (hidden) index i^* such that there exists an efficient extractor that takes any accepting proof π for a tuple $(x_1, \ldots, x_T)$ and outputs a valid witness w_{i^*} for instance x_{i^*}). We can now instantiate the SNARG with polylogarithmic verification cost using the Sahai-Waters construction based on $i\mathcal{O}$ and one-way functions, and the somewhere extractable BARG for an unbounded number of instances with the recent lattice-based scheme of Choudhuri et al. [CJJ21b]. This result provides a basic feasibility result for the existence of fully succinct batch arguments for NP. However, instantiating this compiler requires two sets of assumptions: $i\mathcal{O}$ and one-way functions for the underlying SNARG, and lattice-based assumptions for the BARG.

This work. In this work, we provide a direct route for constructing fully succinct BARGs that support an unbounded number of statements from $i\mathcal{O}$ and one-way

[3] One way to do this is to observe that the above approach already gives a fully succinct batch argument for index languages (i.e., a batch language where the $T \leq 2^\lambda$ instances are defined to be $(x_1, x_2, \ldots, x_T) = (1, 2, \ldots, T)$). Then, we can apply the index BARG to BARG transformation from Choudhuri et al. [CJJ21b], which relies on somewhere extractable commitments.

functions. Notably, combined with the breakthrough work of Jain, Lin, and Sahai [JLS22], this provides an instantiation of fully succinct BARGs *without* lattice assumptions (in contrast to the generic approach above). Using our construction, proving T statements for an NP relation of size s requires a proof of length $\mathsf{poly}(\lambda)$. This is *independent* of both the number of statements T and the size s of the associated NP relation. Like the scheme of Sahai and Waters, our construction satisfies *non-adaptive* soundness (and perfect zero-knowledge). We summarize this instantiation in the informal theorem below:

Theorem 1.1 (Fully Succinct BARG (Informal)). *Assuming the existence of indistinguishability obfuscation and one-way functions, there exists a fully succinct, non-adaptively sound batch argument for* NP*. The batch argument satisfies perfect zero knowledge.*

Updatable batch arguments. We also show how to extend our construction to obtain an *updatable* BARG through the use of somewhere statistically binding (SSB) hash functions [HW15, OPWW15]. In an updatable BARG, a prover is able to take an existing proof π_T on statements $(x_1, \ldots, x_T)$ along with a new statement x_{T+1} with associated NP witness w_{T+1} and update π to a new proof π' on instances $(x_1, \ldots, x_T, x_{T+1})$. Notably, the update algorithm does *not* require the prover to have a witness for any statement other than x_{T+1}. This is useful in settings where the full set of statements/witnesses are not fixed in advance (e.g., in a streaming setting). For example, a prover might want to compute a summary of all transactions that occur in a given day and then provide a proof that the summary reflects the complete set of transactions from the day. An updatable BARG would allow the prover to maintain just a single proof that authenticates all of the summary reports from different days, and moreover, the prover does *not* have to maintain the full list of transactions from earlier days to perform the update. We show how to obtain a fully succinct updatable BARG in Section 5, and we summarize this instantiation in the following theorem.

Theorem 1.2 (Updatable BARG (Informal)). *Assuming the existence of indistinguishability obfuscation scheme and somewhere statistical binding hash functions, there exists a fully succinct, non-adaptively sound updatable batch argument for* NP*. The batch argument satisfies perfect zero knowledge.*

1.1 Technical Overview

In this section, we provide an informal overview of the techniques that we use to construct fully succinct BARGs. Throughout this section, we consider the batch NP language of Boolean circuit satisfiability. Namely, the prover has a Boolean circuit C and a collection of instances $x_1, \ldots, x_T$, and its goal is to convince the verifier that there exist witnesses $w_1, \ldots, w_T$ such that $C(x_i, w_i) = 1$ for all $i \in [T]$.

The Sahai-Waters SNARG. As a warmup, we recall the Sahai-Waters [SW14] construction of SNARGs from $i\mathcal{O}$ for a single instance (i.e., the case where $T = 1$). In this construction, the common reference string (CRS) consists of two obfuscated programs: Prove and Verify. The Prove program takes in the circuit C, the statement x, and the witness w, and outputs a signature σ_x on x if $C(x, w) = 1$ and $\perp$ otherwise. The proof is simply the signature $\pi = \sigma_x$. The Verify program takes in the description of the circuit C, the statement x, and the proof $\pi = \sigma_x$ and checks whether σ_x is a valid signature on x or not. The signature in this case just corresponds to the evaluation of a pseudorandom function (PRF) on the input x. The key to the PRF is hard-coded in the obfuscated proving and verification programs. Security in turn, relies on the Sahai-Waters "punctured programming" technique.

Batch arguments for index languages. To construct fully succinct batch arguments, we start by considering the special case of an *index language* (similar to the starting point in the lattice-based construction of Choudhuri et al. [CJJ21b]). In a BARG for an index language, the statements are simply the indices $(1, 2, \ldots, T)$. The prover's goal is to convince the verifier that there exists w_i such that $C(i, w_i) = 1$ for all $i \in [T]$. We start by showing how to construct a fully succinct BARG for index languages with an unbounded number of instances (i.e., an index language for arbitrary polynomial T). Our construction proceeds *iteratively* as follows. Like the Sahai-Waters construction, the CRS in our scheme consists of the obfuscation of the following two programs:

- The proving program takes in a circuit C, an index i, a witness w_i for instance i and a proof π for the first $i - 1$ statements. The program checks if $C(i, w_i) = 1$ and that the proof on the first $i - 1$ statements is valid. When $i = 1$, then we ignore the latter check. If both conditions are satisfied, the program outputs a signature on statement (C, i). Notably, the size of the prover program only scales with the size of the circuit and the bit-length of the number of instances (instead of linearly with the number of instances).

  Similar to the construction of Sahai and Waters, we define the "signature" on the statement (C, i) to be $\pi = \mathsf{F}(k, (C, i))$, where F is a puncturable PRF [BW13, KPTZ13, BGI14],[4] and k is a PRF key that is hard-coded in the proving program.
- To verify a proof on T statements (i.e., the instances $1, \ldots, T$), the verification program simply checks that the proof π is a valid signature on the pair (C, T). Based on how we defined the proving program above, this corresponds to checking that $\pi = \mathsf{F}(k, (C, T))$. Now, to argue soundness using the Sahai-Waters punctured programming paradigm, we modify this check and replace

[4] A puncturable PRF is a PRF where the holder of the master secret key can "puncture" the key on an input x^*. The resulting punctured key k' can be used to evaluate the PRF on all inputs except x^*. The value of the PRF at x^* remains pseudorandom (i.e., computationally indistinguishable from random) even given the punctured key k'. We provide the formal definition in Definition 2.2.

it with the check

$$\mathsf{G}(\pi) \stackrel{?}{=} \mathsf{G}(\mathsf{F}(k, (C, T))),$$

where G is a length-doubling pseudorandom generator. This will be critical for arguing soundness.

Soundness of the index BARG. To argue non-adaptive soundness of the above approach (i.e., the setting where the statement is chosen independently of the CRS), we apply the punctured programming techniques of Sahai and Waters [SW14]. Take any circuit C^* and suppose there is an index i^* where for all witnesses w, we have that $C^*(i^*, w) = 0$. Our soundness analysis proceeds in two steps:

- We first show that no efficient prover can compute an accepting proof π on instances $(1, \ldots, i^*)$ for circuit C^*.
- Then, we show how to "propagate" the inability to construct a valid proof on index i^* to all indices $i \geq i^*$. This in turn suffices to argue non-adaptive soundness for an arbitrary polynomial number of statements.

We now sketch the argument for the first step. In the following overview, suppose the output space of F is $\{0,1\}^\lambda$ and suppose that $\mathsf{G}\colon \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$ is length-doubling.

- The real CRS consists of obfuscations of the following proving and verification programs:

$\mathsf{Prove}(C, i, w_i, \pi)$:	$\mathsf{Verify}(C, i, \pi)$:
• If $C(i, w_i) = 0$, output $\perp$. • If $i = 1$, output $\mathsf{F}(k, (C, i))$. • If $\mathsf{G}(\pi) = \mathsf{G}(\mathsf{F}(k, (C, i-1)))$, output $\mathsf{F}(k, (C, i))$. • Output $\perp$.	• If $\mathsf{G}(\pi) = \mathsf{G}(\mathsf{F}(k, (C, i)))$, output 1 • Output 0.

- First, instead of embedding the real PRF key k in the proving and verification programs, we embed a punctured PRF key k' that is punctured on the input (C^*, i^*). Whenever the proving and verification program needs to evaluate F on the punctured point (C^*, i^*), we hard-code the value $z = \mathsf{F}(k, (C^*, i^*))$:

$\mathsf{Prove}(C, i, w_i, \pi)$:	$\mathsf{Verify}(C, i, \pi)$:

• If $C(i, w_i) = 0$, output $\perp$.
• If $C = C^*$ and $i = i^*$, output $\perp$.
• If $i = 1$, output $\mathsf{F}(k', (C, i))$.
• If $C = C^*$ and $i - 1 = i^*$:
  ∗ If $\mathsf{G}(\pi) = \mathsf{G}(z)$, output $\mathsf{F}(k', (C, i))$.
  ∗ Otherwise, output $\perp$.
• If $\mathsf{G}(\pi) = \mathsf{G}(\mathsf{F}(k', (C, i-1)))$, output $\mathsf{F}(k', (C, i))$.
• Output $\perp$.
	• If $C = C^*$ and $i = i^*$, output 1 if $\mathsf{G}(\pi) = \mathsf{G}(z)$ and 0 otherwise.
	• If $\mathsf{G}(\pi) = \mathsf{G}(\mathsf{F}(k', (C, i)))$, output 1.
	• Output 0.

Since the punctured PRF is functionality-preserving, on all inputs $(C, i) \neq (C^*, i^*)$, we have that $\mathsf{F}(k, (C, i)) = \mathsf{F}(k', (C, i))$. Since $z = \mathsf{F}(k, (C^*, i^*))$, the input/output behavior of the verification program is unchanged. Next, $C(i^*, w) = 0$ for all w, so the input/output behavior of the proving program is also unchanged. Security of $i\mathcal{O}$ then ensures that the obfuscated proving and verification programs are computationally indistinguishable from those in the real scheme.

– Observe that both the proving and verification programs can be constructed given just the value of $\mathsf{G}(z)$ *without* necessarily knowing z itself. We now replace the target value $\mathsf{G}(z)$ with a uniform random string $t \xleftarrow{R} \{0,1\}^{2\lambda}$. This follows by (1) puncturing security of F which says that the value of $z = \mathsf{F}(k, (C^*, i^*))$ is computationally indistinguishable from a uniform string $z \xleftarrow{R} \{0,1\}^{\lambda}$; and (2) by PRG security since the distribution of $\mathsf{G}(z)$ where $z \xleftarrow{R} \{0,1\}^{\lambda}$ is computationally indistinguishable from sampling a uniform random string $t \xleftarrow{R} \{0,1\}^{2\lambda}$. With these modifications, the proving and verification programs behave as follows:

$\mathsf{Prove}(C, i, w_i, \pi)$:

- If $C(i, w_i) = 0$, output $\bot$.
- If $C = C^*$ and $i = i^*$, output $\bot$.
- If $i = 1$, output $\mathsf{F}(k', (C, i))$.
- If $C = C^*$ and $i - 1 = i^*$:
  - If $\mathsf{G}(\pi) = t$, output $\mathsf{F}(k', (C, i))$.
  - Otherwise, output $\bot$.
- If $\mathsf{G}(\pi) = \mathsf{G}(\mathsf{F}(k', (C, i-1)))$, output $\mathsf{F}(k', (C, i))$.
- Output $\bot$.

$\mathsf{Verify}(C, i, \pi)$:

- If $C = C^*$ and $i = i^*$, output 1 if $\mathsf{G}(\pi) = t$ and 0 otherwise.
- If $\mathsf{G}(\pi) = \mathsf{G}(\mathsf{F}(k', (C, i)))$, output 1.
- Output 0.

– Since t is uniform in $\{0,1\}^{2\lambda}$, the probability that t is even in the image of G is at most $2^{-\lambda}$. Thus, in this experiment, with probability $1 - 2^{-\lambda}$, there does not exist any accepting proof π for input (C^*, i^*). This means that we can now revert to using the PRF key k in both the proving and verification programs and simply reject all proofs on instance (C^*, i^*). In other words, we can replace the proving and verification programs with obfuscations of the following programs by appealing to the security of $i\mathcal{O}$:

$\mathsf{Prove}(C, i, w_i, \pi)$:

- If $C(i, w_i) = 0$, output $\bot$.
- If $C = C^*$ and $i = i^*$, output $\bot$.
- If $i = 1$, output $\mathsf{F}(k, (C, i))$.
- If $C = C^*$ and $i - 1 = i^*$, output $\bot$.
- If $\mathsf{G}(\pi) = \mathsf{G}(\mathsf{F}(k, (C, i-1)))$, output $\mathsf{F}(k', (C, i))$.
- Output $\bot$.

$\mathsf{Verify}(C, i, \pi)$:

- If $C = C^*$ and $i = i^*$, output 0.
- If $\mathsf{G}(\pi) = \mathsf{G}(\mathsf{F}(k, (C, i)))$, output 1.
- Output 0.

In this final experiment, there no longer exists an accepting proof π on instances $(1, \ldots, i^*)$ for circuit C^*. Next, we show how to extend this argument to *additionally* remove accepting proofs on the batch of instances $(1, \ldots, i^*, i^* + 1)$. We leverage a similar strategy as before:

– We replace the PRF key k with a punctured key k' that is punctured at $(C^*, i^* + 1)$ in both the proving and verification programs. Again, whenever the programs need to compute $\mathsf{F}(k, (C^*, i^* + 1))$, we substitute a hard-coded value $z = \mathsf{F}(k, (C^*, i^* + 1))$:

$\mathsf{Prove}(C, i, w_i, \pi)$:	$\mathsf{Verify}(C, i, \pi)$:

• If $C(i, w_i) = 0$, output $\perp$.
• If $C = C^*$ and $i^* \leq i \leq i^* + 1$, output $\perp$.
• If $i = 1$, output $\mathsf{F}(k', (C, i))$.
• If $C = C^*$ and $i - 1 = i^* + 1$:
  ∗ If $\mathsf{G}(\pi) = \mathsf{G}(z)$, output $\mathsf{F}(k', (C, i))$.
  ∗ Otherwise, output $\perp$.
• If $\mathsf{G}(\pi) = \mathsf{G}(\mathsf{F}(k', (C, i-1)))$, output $\mathsf{F}(k', (C, i))$.
• Output $\perp$.
	• If $C = C^*$, and $i = i^*$, output 0.
	• If $C = C^*$, $i = i^* + 1$, output 1 if $\mathsf{G}(\pi) = \mathsf{G}(z)$ and 0 otherwise.
	• If $\mathsf{G}(\pi) = \mathsf{G}(\mathsf{F}(k', (C, i)))$, output 1.
	• Output 0.

Note that to simplify the notation, we merged the individual checks ($C = C^*$ and $i = i^*$) and ($C = C^*$ and $i - 1 = i^*$) in the proving program into a single check that outputs $\perp$ if satisfied.

– Observe once again that the description of the proving and verification programs only depends on $\mathsf{G}(z)$ (and *not* z itself). By the same sequence of steps as above, we can appeal to puncturing security of F, pseudorandomness of G, and security of $i\mathcal{O}$ to show that the obfuscated proving and verification programs are computationally indistinguishable from the following programs:

$\mathsf{Prove}(C, i, w_i, \pi)$:	$\mathsf{Verify}(C, i, \pi)$:
• If $C(i, w_i) = 0$, output $\perp$. • If $C = C^*$ and $i^* \leq i \leq i^* + 2$, output $\perp$. • If $i = 1$, output $\mathsf{F}(k, (C, i))$. • If $\mathsf{G}(\pi) = \mathsf{G}(\mathsf{F}(k, (C, i-1)))$, output $\mathsf{F}(k, (C, i))$. • Output $\perp$.	• If $C = C^*$ and $i^* \leq i \leq i^* + 1$, output 0. • If $\mathsf{G}(\pi) = \mathsf{G}(\mathsf{F}(k, (C, i)))$, output 1. • Output 0.

We can repeat the above strategy any polynomial number of times. In particular, for any $T = \mathsf{poly}(\lambda)$, we can replace the obfuscated programs in the CRS with the following programs:

$\mathsf{Prove}(C, i, w_i, \pi)$:	$\mathsf{Verify}(C, i, \pi)$:
• If $C(i, w_i) = 0$, output $\perp$. • If $C = C^*$ and $i^* \leq i \leq T + 1$, output $\perp$. • If $i = 1$, output $\mathsf{F}(k, (C, i))$. • If $\mathsf{G}(\pi) = \mathsf{G}(\mathsf{F}(k, (C, i-1)))$, output $\mathsf{F}(k, (C, i))$. • Output $\perp$.	• If $C = C^*$ and $i^* \leq i \leq T$, output 0. • If $\mathsf{G}(\pi) = \mathsf{G}(\mathsf{F}(k, (C, i)))$, output 1. • Output 0.

By security of $i\mathcal{O}$, the puncturable PRF, and the PRG, this modified CRS is computationally indistinguishable from the real CRS. However, when the verification program is implemented as above, there are no accepting proofs on input (C^*, i) for any $i^* \leq i \leq T$. Moreover, the size of the obfuscated programs only depends on $\log T$ (and not T). As such, the scheme supports an arbitrary polynomial number of statements. We give the full analysis in Section 3.

Adaptive soundness and zero knowledge. Using standard complexity leveraging techniques, we show how to extend our BARG for index languages with non-adaptive soundness into one with adaptive soundness in the full version of this paper. We note that due to the reliance on complexity leveraging, the resulting BARGs we obtain are no longer fully succinct; the proof size now scales with the *size* of the NP relation, but critically, still sublinearly in the number of instances. We also note that much like the construction of Sahai and Waters,

both our fully succinct non-adaptive BARG and our adaptive BARG satisfy perfect zero-knowledge.

From index languages to general NP *languages.* Next, we show how to bootstrap our fully succinct BARG for index languages to obtain a fully succinct BARG for NP that supports an arbitrary polynomial number of statements. In this setting, the prover has a Boolean circuit C and *arbitrary* instances $x_1, \ldots, x_T$; the prover's goal is to convince the verifier that for all $i \in [T]$, there exists w_i such that $C(x_i, w_i) = 1$.

The key difference between general NP languages and index languages is that the tuple of statements $(x_1, \ldots, x_T)$ no longer has a succinct description. This property was critical in our soundness analysis above. The soundness argument we described above works by embedding the instances $x_{i^*}, x_{i^*+1}, \ldots, x_T$ into the proving and verification programs (where x_{i^*} denotes a false instance) and have the programs always reject proofs on these statements (with respect to the target circuit C^*). For index languages, these instances just correspond to the interval $[i^* + 1, T]$, which can be described succinctly with $O(\log T)$ bits. When $x_{i^*}, x_{i^*+1}, \ldots, x_T$ are *arbitrary* instances, they do not have a short description, and we cannot embed these instances into the proving and verification programs without imposing an *a priori* bound on the number of instances.

Instead of modifying the above construction, we instead adopt the approach of Choudhuri et al. [CJJ21b] who previously showed how to generically upgrade any BARG for index languages to a BARG for NP by relying on somewhere extractable commitment schemes. If the underlying BARG for index languages supports an unbounded number of instances, then the transformed scheme also does. In our setting, we observe that if we only require (non-adaptive) soundness (as opposed to "somewhere extraction"), we can use a positional accumulator [KLW15] in place of the somewhere extractable commitment scheme. The advantage of basing the transformation on positional accumulators is that we can construct positional accumulators directly from indistinguishability obfuscation and one-way functions. Applied to the above index BARG construction (see also Section 3), we obtain a fully succinct batch argument for NP from the *same* set of assumptions. In contrast, if we invoke the compiler of Choudhuri et al., we would need to *additionally* assume the existence of a somewhere extractable commitment scheme which *cannot* be based solely on indistinguishability obfuscation together with one-way functions in a fully black-box way [AS15].

Very briefly, in the Choudhuri et al. approach, to construct a batch argument on the tuple $(C, x_1, \ldots, x_T)$, the prover first computes a succinct hash y of the statements $(x_1, \ldots, x_T)$. Using y, they define an index relation where instance i is satisfied if there exists an opening (x_i, π_i) to y at index i, and moreover, there exists a satisfying witness w_i where $C(x_i, w_i) = 1$. The proof then consists of the hash y and a proof for the index relation. In this work, we show that using a positional accumulator to instantiate the hash function suffices to obtain a BARG with non-adaptive soundness. We provide the full details in Section 4.

Updatable BARGs for NP. Our techniques also readily generalize to obtain an updatable batch argument (for general NP) from the same underlying set of assumptions. Recall that in an updatable BARG, a prover can take an existing proof π on a tuple $(C, x_1, \ldots, x_T)$ together with a new statement x_{T+1} and witness w_{T+1} and extend π to a new proof π' on the tuple $(C, x_1, \ldots, x_T, x_{T+1})$. One way to construct an updatable BARG is to recursive compose a succinct non-interactive argument of knowledge [BCCT13] or a rate-1 batch argument [DGKV22].[5] Here, we opt for a more direct approach based on the above techniques, which does not rely on recursive composition.

First, our index BARG construction described above is already updatable. However, if we apply the Choudhuri et al. [CJJ21b] transformation to obtain a BARG for NP, the resulting scheme is no longer updatable. This is because the transformation requires the prover to commit to the complete set of statements and then argue that the statement associated with each index is true (which in turn requires knowledge of all of the associated witnesses).

Instead, we take a different and more direct *tree-based* approach. For ease of exposition, suppose first that $T = 2^k$ for some integer k. Our construction will rely on a hash function H. Given a tuple of T statements $(x_1, \ldots, x_T)$, we construct a binary Merkle hash tree [Mer87] of depth k as follows: the leaves of the tree are labeled $x_1, \ldots, x_T$, and the value of each internal node v is the hash $H(v_1, v_2)$ of its two children v_1 and v_2. The output h of the hash tree is the value at the root node, and we denote this by writing $h = H_{\mathsf{Merkle}}(x_1, \ldots, x_T)$. A proof on the tuple of instances $(x_1, \ldots, x_T)$ is simply a signature on the root node $H_{\mathsf{Merkle}}(x_1, \ldots, x_T)$. Now, instead of providing an obfuscated program that takes a proof on index i and extends it into a proof on index $i + 1$, we define our obfuscated proving program to take in two signatures on hash values $h_1 = H_{\mathsf{Merkle}}(x_1, \ldots, x_T)$ and $h_2 = H_{\mathsf{Merkle}}(y_1, \ldots, y_T)$ and output a signature on the hash value $h = H(h_1, h_2) = H_{\mathsf{Merkle}}(x_1, \ldots, x_T, y_1, \ldots, y_T)$. This new "two-to-one" obfuscated program allows us to merge two proofs on T instances into a single proof on $2T$ instances. More generally, the (obfuscated) proving program in the CRS now supports the following operations:

- **Signing a single instance:** Given a circuit C, a statement x, and a witness w, output a signature on $(C, x, 1)$ if $C(x, w) = 1$ and $\perp$ otherwise. This can be viewed as a signature on a hash tree of depth 1.
- **Merge trees:** Given a circuit C, hashes h_1, h_2 associated with two trees of depth k, along with signatures σ_1, σ_2, check that σ_1 is a valid signature on (C, h_1, k), and σ_2 is a valid signature on (C, h_2, k). If both checks pass, output a signature on $(C, H(h_1, h_2), k+1)$. This is a signature on a hash tree of depth $k + 1$.

To construct a proof on instances $(x_1, \ldots, x_T)$ using witnesses $(w_1, \ldots, w_T)$ for *arbitrary* T, we now proceed as follows:

[5] If the underlying BARG is not rate-1, then we can only compose a *bounded* number of times.

- Run the (obfuscated) proving algorithm on (C, x_1, w_1) to obtain a signature σ on $(C, x_1, 1)$. The initial proof π is simply the set $\{(1, x_1, \sigma)\}$.
- Suppose $\pi = \{(i, h_i, \sigma_i)\}$ is a proof on the first $T-1$ statements. To update the proof π to a proof on the first T statements, first run the proving algorithm on (C, x_T, w_T) to obtain a signature σ on $(C, x_T, 1)$. Now, we apply the following merging procedure:
  - Initialize $(k, h', \sigma') \leftarrow (1, x_T, \sigma)$ and $\pi' \leftarrow \pi$.
  - While there exists $(i, h_i, \sigma_i) \in \pi'$ where $i = k$, run the (obfuscated) merge program on $(C, h_i, h', k, \sigma_i, \sigma')$ to obtain a signature σ'' on $(C, H(h_i, h'), k+1)$. Remove (i, h_i, σ_i) from π' and update $(k, h', \sigma') \leftarrow (k+1, H(h_i, h'), \sigma'')$.
  - Add the tuple (k, h', σ'') to π' at the conclusion of the merging process.

  Observe that the update procedure only requires knowledge of the new statement x_T, its witness w_T, and the proof on the previous statements π; it does *not* require knowledge of the witnesses to the previous statements. Moreover, observe that the number of hash-signature tuples in π is always bounded by $\log T$.

To verify a proof $\pi = \{(i, h_i, \sigma_i)\}$ with respect to a Boolean circuit C, the verifier checks that σ_i is a valid signature on (C, h_i, i) for all tuples in π, and moreover, that each of the intermediate hash values h_i are correctly computed from $(x_1, \ldots, x_T)$. Non-adaptive soundness of the above construction follows by a similar argument as that for our index BARG. Notably, we show that if an instance x_{i^*} is false, then the proving program will never output a signature on input $(C, x_{i^*}, 1)$. Using the same punctured programming technique sketched above, we can again "propagate" the inability to compute a signature on the leaf node i^* to argue that any efficient prover cannot compute a signature on any node that is an ancestor of x_{i^*} in the hash tree. Here, we will need to rely on the underlying hash function being somewhere statistically binding [HW15, OPWW15]. By a hybrid argument, we can eventually move to an experiment where there are no accepting proofs on tuples that contain x_{i^*}, and soundness follows. We provide the formal description in Section 5.

2 Preliminaries

Throughout this work, we write λ to denote the security parameter. We say a function f is negligible in the security parameter λ if $f = o(\lambda^{-c})$ for all $c \in \mathbb{N}$. We denote this by writing $f(\lambda) = \mathsf{negl}(\lambda)$. We write $\mathsf{poly}(\lambda)$ to denote a function that is bounded by a fixed polynomial in λ. We say an algorithm is PPT if it runs in probabilistic polynomial time in the length of its input. By default, we consider *non-uniform* adversaries (indexed by λ) where the algorithm may additionally take in an advice string (of $\mathsf{poly}(\lambda)$ length).

For a positive integer $n \in \mathbb{N}$, we write $[n]$ to denote the set $\{1, \ldots, n\}$ and $[0, n]$ to denote the set $\{0, \ldots, n\}$. For a finite set S, we write $x \xleftarrow{R} S$ to denote that x is sampled uniformly at random from S. For a distribution $\mathcal{D}$, we write $x \leftarrow D$ to denote that x is sampled from D. We say an event E occurs with overwhelming probability if its complement occurs with negligible probability.

Some of our constructions in this work will rely on hardness against adversaries running in sub-exponential time or achieving sub-exponential advantage (i.e., success probability). To make this explicit, we formulate our security definitions in the language of (τ, ε)-security, where $\tau = \tau(\lambda)$ and $\varepsilon = \varepsilon(\lambda)$. Here, we say a primitive is (τ, ε)-secure if for all (non-uniform) polynomial time adversaries running in time $\tau(\lambda)$ and all sufficiently large λ, the adversary's advantage is bounded by $\varepsilon(\lambda)$. For ease of exposition, we will also write that a primitive is "secure" (without an explicit (τ, ε) characterization) if for *every* polynomial $\tau = \mathsf{poly}(\lambda)$, there exists a negligible function $\varepsilon(\lambda) = \mathsf{negl}(\lambda)$ such that the primitive is (τ, ε)-secure. We now review the main cryptographic primitives we use in this work.

Definition 2.1 (Indistinguishability Obfuscation [BGI+01]). *An indistinguishability obfuscator for a circuit class $\mathcal{C} = \{\mathcal{C}_\lambda\}_{\lambda \in \mathbb{N}}$ is a PPT algorithm $i\mathcal{O}(\cdot, \cdot)$ with the following properties:*

- **Correctness:** *For all security parameters $\lambda \in \mathbb{N}$, all circuits $C \in \mathcal{C}_\lambda$, and all inputs x,*
$$\Pr[C'(x) = C(x) : C' \leftarrow i\mathcal{O}(1^\lambda, C)] = 1.$$
- **Security:** *We say that $i\mathcal{O}$ is (τ, ε)-secure if for all adversaries $\mathcal{A}$ running in time at most $\tau(\lambda)$, there exists $\lambda_{\mathcal{A}} \in \mathbb{N}$, such that for all security parameters $\lambda > \lambda_{\mathcal{A}}$, all pairs of circuits $C_0, C_1 \in \mathcal{C}_\lambda$ where $C_0(x) = C_1(x)$ for all inputs x, we have,*
$$\mathsf{Adv}^{i\mathcal{O}}_{\mathcal{A}} := \left|\Pr[\mathcal{A}(i\mathcal{O}(1^\lambda, C_0)) = 1] - \Pr[\mathcal{A}(i\mathcal{O}(1^\lambda, C_1)) = 1]\right| \leq \varepsilon(\lambda).$$

Definition 2.2 (Puncturable PRF [BW13, KPTZ13, BGI14]). *A puncturable pseudorandom function family on key space $\mathcal{K} = \{\mathcal{K}_\lambda\}_{\lambda \in \mathbb{N}}$, domain $\mathcal{X} = \{\mathcal{X}_\lambda\}_{\lambda \in \mathbb{N}}$ and range $\mathcal{Y} = \{\mathcal{Y}_\lambda\}_{\lambda \in \mathbb{N}}$ consists of a tuple of PPT algorithms $\Pi_{\mathsf{PPRF}} = (\mathsf{KeyGen}, \mathsf{Eval}, \mathsf{Puncture})$ with the following properties:*

- $\mathsf{KeyGen}(1^\lambda) \to K$*: On input the security parameter λ, the key-generation algorithm outputs a key $K \in \mathcal{K}_\lambda$.*
- $\mathsf{Puncture}(K, S) \to K\{S\}$*: On input the PRF key $K \in \mathcal{K}_\lambda$ and a set $S \subseteq \mathcal{X}_\lambda$, the puncturing algorithm outputs a punctured key $K\{S\} \in \mathcal{K}_\lambda$.*
- $\mathsf{Eval}(K, x) \to y$*: On input a key $K \in \mathcal{K}_\lambda$ and an input $x \in \mathcal{X}_\lambda$, the evaluation algorithm outputs a value $y \in \mathcal{Y}_\lambda$.*

In addition, Π_{PPRF} should satisfy the following properties:

- **Functionality-preserving:** *For every polynomial $s = s(\lambda)$, every security parameter $\lambda \in \mathbb{N}$, every subset $S \subseteq \mathcal{X}_\lambda$ of size at most s, and every $x \in \mathcal{X}_\lambda \backslash S$,*
$$\Pr[\mathsf{Eval}(K, x) = \mathsf{Eval}(K\{S\}, x) : K \leftarrow \mathsf{KeyGen}(1^\lambda), K\{S\} \leftarrow \mathsf{Puncture}(K, S)] = 1.$$
- **Punctured pseudorandomness:** *For a bit $b \in \{0, 1\}$ and a security parameter λ, we define the (selective) punctured pseudorandomness game Π_{PPRF}, between an adversary $\mathcal{A}$ and a challenger as follows:*

- *At the beginning of the game, the adversary commits to a set* $S \subseteq \mathcal{X}_\lambda$.
- *The challenger then samples a key* $K \leftarrow \mathsf{KeyGen}(1^\lambda)$*, constructs the punctured key* $K\{S\} \leftarrow \mathsf{Puncture}(K, S)$*, and gives* $K\{S\}$ *to* $\mathcal{A}$.
- *If* $b = 0$*, the challenger gives the set* $\{(x_i, \mathsf{Eval}(K, x_i))\}_{x_i \in S}$ *to* $\mathcal{A}$*. If* $b = 1$*, the challenger gives the set* $\{(x_i, y_i)\}_{x_i \in S}$ *where each* $y_i \xleftarrow{R} \mathcal{Y}_\lambda$.
- *At the end of the game, the adversary outputs a bit* $b' \in \{0,1\}$*, which is the output of the experiment.*

We say that Π_{PPRF} *satisfies* (τ, ε)*-punctured security if for all adversaries* $\mathcal{A}$ *running in time at most* $\tau(\lambda)$*, there exists* $\lambda_{\mathcal{A}}$ *such that for all security parameters* $\lambda > \lambda_{\mathcal{A}}$*,*

$$|\Pr[b' = 1 : b = 0] - \Pr[b' = 1 : b = 1]| \leq \varepsilon(\lambda)$$

in the punctured pseudorandomness security game.

For ease of notation, we will often write $F(K, x)$ *to represent* $\mathsf{Eval}(K, x)$.

Definition 2.3 (Pseudorandom Generator). *A pseudorandom generator (PRG) on domain* $\mathcal{X} = \{\mathcal{X}_\lambda\}_{\lambda \in \mathbb{N}}$ *and range* $\mathcal{Y} = \{\mathcal{Y}_\lambda\}_{\lambda \in \mathbb{N}}$ *is a deterministic polynomial-time algorithm* $\mathsf{PRG}\colon \mathcal{X} \to \mathcal{Y}$*. We say that the* PRG *is* (τ, ε)*-secure if for all adversaries* $\mathcal{A}$ *running in time at most* $\tau(\lambda)$*, there exists* $\lambda_{\mathcal{A}} \in \mathbb{N}$*, such that for all security parameters* $\lambda > \lambda_{\mathcal{A}}$*, we have,*

$$\mathsf{Adv}^{\mathsf{PRG}}_{\mathcal{A}} := |\Pr[\mathcal{A}(\mathsf{PRG}(x)) = 1 : x \leftarrow \mathcal{X}_\lambda] - \Pr[\mathcal{A}(y) = 1 : y \leftarrow \mathcal{Y}_\lambda]| \leq \varepsilon(\lambda).$$

2.1 Batch Arguments for NP

We now introduce the notion of a non-interactive batch argument (BARG) for NP. We focus specifically on the language of Boolean circuit satisfiability.

Definition 2.4 (Circuit Satisfiability). *For a Boolean circuit* $C\colon \{0,1\}^\ell \times \{0,1\}^m \to \{0,1\}$*, and a statement* $x \in \{0,1\}^n$*, we define the language of Boolean circuit satisfiability* $\mathcal{L}_{\mathsf{CSAT}}$ *as follows:*

$$\mathcal{L}_{\mathsf{CSAT}} = \{(C, x) \mid \exists w \in \{0,1\}^m : C(x, w) = 1\}.$$

Definition 2.5 (Batch Circuit Satisfiability). *For a Boolean circuit* $C\colon \{0,1\}^\ell \times \{0,1\}^m \to \{0,1\}$*, positive integer* $t \in \mathbb{N}$*, and statements* $x_1, \ldots, x_t \in \{0,1\}^n$*, we define the batch circuit satisfiability language as follows:*

$$\mathcal{L}_{\mathsf{BatchCSAT},t} = \{(C, x_1, \ldots, x_t) \mid \forall i \in [t], \exists w_i \in \{0,1\}^m : C(x_i, w_i) = 1\}.$$

Definition 2.6 (Batch Argument for NP). *A batch argument (BARG) for the language of Boolean circuit satisfiability consists of a tuple of PPT algorithms* $\Pi_{\mathsf{BARG}} = (\mathsf{Gen}, \mathsf{P}, \mathsf{V})$ *with the following properties:*

– $\mathsf{Gen}(1^\lambda, 1^\ell, 1^T, 1^s) \to \mathsf{crs}$*: On input the security parameter* λ*, a bound on the instance size* ℓ*, a bound on the number of statements* T*, and a bound on the circuit size* s*, the generator algorithm outputs a common reference string* crs.

- $\mathsf{P}(\mathsf{crs}, C, (x_1, \ldots, x_t), (w_1, \ldots, w_t)) \to \pi$*: On input the common reference string* crs*, a Boolean circuit* $C : \{0,1\}^\ell \times \{0,1\}^m \to \{0,1\}$*, a list of statements* $x_1, \ldots, x_t \in \{0,1\}^\ell$*, and a list of witnesses* $w_1, \ldots, w_t \in \{0,1\}^m$*, the prove algorithm outputs a proof* π*.*
- $\mathsf{V}(\mathsf{crs}, C, (x_1, \ldots, x_t), \pi) \to \{0,1\}$*: On input the common reference string* crs*, a Boolean circuit* $C : \{0,1\}^\ell \times \{0,1\}^m \to \{0,1\}$*, a list of statements* $x_1, \ldots, x_t \in \{0,1\}^\ell$*, and a proof* π*, the verification algorithm outputs a bit* $b \in \{0,1\}$*.*

Moreover, the BARG scheme should satisfy the following properties:

- **Completeness:** *For all security parameters* $\lambda \in \mathbb{N}$ *and bounds* $\ell \in \mathbb{N}$*,* $s \in \mathbb{N}$*,* $T \in \mathbb{N}$*,* $t \leq T$*, Boolean circuits* $C : \{0,1\}^\ell \times \{0,1\}^m \to \{0,1\}$ *of size at most* s*, all statements* $x_1, \ldots, x_t \in \{0,1\}^n$ *and all witnesses* $w_1, \ldots, w_t$ *where* $C(x_i, w_i) = 1$ *for all* $i \in [t]$*, it holds that*

$$\Pr\left[\mathsf{V}(\mathsf{crs}, C, (x_1, \ldots, x_t), \pi) = 1 : \begin{array}{c} \mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda, 1^\ell, 1^T, 1^s) \\ \pi \leftarrow \mathsf{P}(\mathsf{crs}, C, (x_1, \ldots, x_t), (w_1, \ldots, w_t)) \end{array}\right] = 1.$$

- **Succinctness:** *We require* Π_{BARG} *satisfy two notions of succinctness:*
  - ***Succinct proof size:*** *For all* $t \leq T$*, it holds that* $|\pi| = \mathsf{poly}(\lambda, \log t, s)$ *in the completeness experiment defined above. Moreover, we say the proof is* fully succinct *if* $|\pi| = \mathsf{poly}(\lambda, \log t, \log s)$*.*
  - ***Succinct verification time:*** *For all* $t \leq T$*, the running time of the verification algorithm* $\mathsf{V}(\mathsf{crs}, C, (x_1, \ldots, x_t), \pi)$ *is* $\mathsf{poly}(\lambda, t, \ell) + \mathsf{poly}(\lambda, \log t, s)$ *in the completeness experiment defined above.*
- **Soundness:** *We require two succinctness properties:*
  - **Non-adaptive soundness:** *For all polynomials* $T = T(\lambda), s = s(\lambda), \ell = \ell(\lambda), t = t(\lambda)$ *where* $t \leq T$*, and all PPT adversaries* $\mathcal{A}$*, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for all* $\lambda \in \mathbb{N}$*, all circuit families* $C = \{C_\lambda\}_{\lambda \in \mathbb{N}}$ *where* $C_\lambda \colon \{0,1\}^{\ell(\lambda)} \times \{0,1\}^{m(\lambda)} \to \{0,1\}$ *is a Boolean circuit of size at most* $s(\lambda)$*, and all statements* $x_1, \ldots, x_t \in \{0,1\}^{\ell(\lambda)}$ *where* $(C_\lambda, (x_1, \ldots, x_t)) \notin \mathcal{L}_{\mathsf{BatchCSAT},t}$*,*

$$\Pr\left[\mathsf{V}(\mathsf{crs}, C_\lambda, (x_1, \ldots, x_t), \pi) = 1 : \begin{array}{c} \mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda, 1^\ell, 1^T, 1^s); \\ \pi \leftarrow \mathcal{A}(1^\lambda, \mathsf{crs}, C_\lambda, (x_1, \ldots, x_t)) \end{array}\right] = \mathsf{negl}(\lambda).$$

  - **Adaptive soundness:** *For a security parameters* λ *and bounds* T, ℓ, s*, we define the adaptive soundness experiment between a challenger and an adversary* $\mathcal{A}$ *as follows:*
    - *The challenger samples* $\mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda, 1^\ell, 1^T, 1^s)$ *and sends* crs *to* $\mathcal{A}$*.*
    - *Algorithm* $\mathcal{A}$ *outputs a Boolean circuit* $C \colon \{0,1\}^\ell \times \{0,1\}^m \to \{0,1\}$ *of size at most* $s(\lambda)$*, statements* $x_1, \ldots, x_t \in \{0,1\}^{\ell(\lambda)}$*, and a proof* π*. Here, we require that* $t \leq T$*.*
    - *The experiment outputs* $b = 1$ *if* $\mathsf{V}(\mathsf{crs}, C, (x_1, \ldots, x_t), \pi) = 1$ *and* $(C, (x_1, \ldots, x_t)) \notin \mathcal{L}_{\mathsf{BatchCSAT},T}$*. Otherwise it outputs* $b = 0$*.*

The scheme satisfies adaptive soundness if for every non-uniform polynomial time adversary $\mathcal{A}$, every polynomial $T = T(\lambda)$,$\ell = \ell(\lambda)$, and $s = s(\lambda)$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that, $\Pr[b = 1] = \mathsf{negl}(\lambda)$ in the adaptive soundness experiment.

- **Perfect zero knowledge**: *The scheme satisfies perfect zero knowledge if there exists a PPT simulator $\mathcal{S}$ such that for all $\lambda \in \mathbb{N}$, all bounds $\ell \in \mathbb{N}$, $T \in \mathbb{N}$, $s \in \mathbb{N}$, all $t \leq T$, all tuples $(C, x_1, \ldots, x_t) \in \mathcal{L}_{\mathsf{BatchCSAT},t}$, and all witnesses $(w_1, \ldots, w_t)$ where $C(x_i, w_i) = 1$ for all $i \in [t]$, the following distributions are identically distributed:*
  - **Real distribution:** *Sample $\mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda, 1^\ell, 1^T, 1^s)$ and a proof $\pi \leftarrow \mathsf{P}(\mathsf{crs}, C, (x_1, \ldots, x_t), (w_1, \ldots, w_t))$. Output (crs, π).*
  - **Simulated distribution:** *Output $(\mathsf{crs}^*, \pi^*) \leftarrow \mathcal{S}(1^\lambda, 1^\ell, 1^T, 1^s, C, (x_1, \ldots, x_t))$.*

Definition 2.7 (BARGs for Unbounded Statements). *We say that a BARG scheme $\Pi_{\mathsf{BARG}} = (\mathsf{Gen}, \mathsf{P}, \mathsf{V})$ supports an unbounded polynomial of statements if the algorithm Gen in Definition 2.6 runs in time that is $\mathsf{poly}(\lambda, \ell, s, \log T)$, and correspondingly, output a CRS of size $\mathsf{poly}(\lambda, \ell, s, \log T)$. Notably, the dependence on the bound T is* polylogarithmic. *In this case, we implicitly set $T = 2^\lambda$ as the input to the Gen algorithm. Observe that in this case, the P and V algorithms can now take any* arbitrary *polynomial number $t = t(\lambda)$ of instances as input where $t \leq 2^\lambda$.*

Batch arguments for index languages. Similar to [CJJ21b], we also consider the special case of batch arguments for index languages. We recall the relevant definitions here.

Definition 2.8 (Batch Circuit Satisfiability for Index Languages). *For a positive integer $t \leq 2^\lambda$, we define the batch circuit satisfiability problem for index languages $\mathcal{L}_{\mathsf{BatchCSATindex},t} = \{(C, t) \mid \forall i \in [t], \exists w_i \in \{0,1\}^m : C(i, w_i) = 1\}$ where $C \colon \{0,1\}^\lambda \times \{0,1\}^m \to \{0,1\}$ is a Boolean circuit.*[6]

Definition 2.9 (Batch Arguments for Index Languages). *A BARG for index languages is a tuple of PPT algorithms $\Pi_{\mathsf{IndexBARG}} = (\mathsf{Gen}, \mathsf{P}, \mathsf{V})$ that satisfy Definition 2.7 for the index language $\mathcal{L}_{\mathsf{BatchCSATindex},t}$. Since we are considering index languages, the statements always consist of the indices $(1, \ldots, t)$. As such, we can modify the P and V algorithms in Definition 2.6 to take as input the single index t (of length λ bits) rather than the tuple of statements $(x_1, \ldots, x_t)$. Specifically, we modify the syntax as follows:*

- $\mathsf{P}(\mathsf{crs}, C, t, (w_1, \ldots, w_t)) \to \pi$: *The prove algorithm takes as input the common reference string crs, a Boolean circuit $C : \{0,1\}^\lambda \times \{0,1\}^m \to \{0,1\}$, the index $t \in \mathbb{N}$, and a list of witnesses $w_1, \ldots, w_t \in \{0,1\}^m$, and outputs a proof π.*

[6] Here, and throughout the exposition, we associate elements of the set $[2^\lambda]$ with their binary representation in $\{0,1\}^\lambda$, and the value 2^λ with the all-zeroes string 0^λ.

– $\mathsf{V}(\mathsf{crs}, C, t, \pi) \to \{0,1\}$*: The verification algorithm takes as input the common reference string* crs*, a Boolean circuit* $C : \{0,1\}^\lambda \times \{0,1\}^m \to \{0,1\}$*, the index* $t \in \mathbb{N}$*, and a proof* π*, and outputs a bit* $b \in \{0,1\}$*.*

The completeness and zero-knowledge properties are the same as those in Definition 2.6 (adapted to the unbounded case where $T = 2^\lambda$*). We define soundness analogously, but require that the adversary outputs the statement index* t *in* unary*. Namely, the adversary is still restricted to choosing a* polynomially-bounded *number of instances* $t = \mathsf{poly}(\lambda)$ *even if the upper bound on* t *is* $T = 2^\lambda$*. For succinctness, we require the following stronger property on the verification time:*

– ***Succinct verification time****: For all* $t \leq 2^\lambda$*, the verification algorithm* $\mathsf{V}(\mathsf{crs}, C, t, \pi)$ *runs in time* $\mathsf{poly}(\lambda, s)$ *in the completeness experiment.*

3 Non-Adaptive Batch Arguments for Index Languages

In this section, we show how to construct a batch argument for index languages that can support an arbitrary polynomial number of statements. We show how to obtain a construction with non-adaptive soundness. As described in Section 1.1, we include two obfuscated programs in the CRS to enable *sequential* proving and batch verification:

– The proving program takes as input a Boolean circuit $C\colon \{0,1\}^\lambda \times \{0,1\}^m \to \{0,1\}$, an instance number $i \in [2^\lambda]$, a witness $w \in \{0,1\}^m$ for instance i as well as a proof π for the first $i-1$ instances. The program validates the proof on the first $i-1$ instances and that $C(i,w) = 1$. If both checks pass, then the program outputs a proof for instance i. Otherwise, it outputs $\perp$.
– The verification program takes as input the circuit C, the *final* instance number $t \in [2^\lambda]$, and a proof π. It outputs a bit indicating whether the proof is valid or not. In this case, outputting 1 indicates that π is a valid proof on instances $(1, \ldots, t)$.

Construction 3.1 (Batch Argument for Index Languages). Let λ be a security parameter and $s = s(\lambda)$ be a bound on the size of the Boolean circuit. We construct a BARG scheme that supports index languages with up to $T = 2^\lambda$ instances (i.e., which suffices to support an arbitrary polynomial number of instances) and circuits of size at most s. The instance indices will be taken from the set $[2^\lambda]$. For ease of notation, we use the set $[2^\lambda]$ and the set $\{0,1\}^\lambda$ interchangably in the following description. Our construction relies on the following primitives:

– Let PRF be a puncturable PRF with key space $\{0,1\}^\lambda$, domain $\{0,1\}^s \times \{0,1\}^\lambda$ and range $\{0,1\}^\lambda$.
– Let $i\mathcal{O}$ be an indistinguishability obfuscator.
– Let PRG be a pseudorandom generator with domain $\{0,1\}^\lambda$ and range $\{0,1\}^{2\lambda}$.

We define our batch argument $\Pi_{\mathsf{BARG}} = (\mathsf{Gen}, \mathsf{P}, \mathsf{V})$ for index languages as follows:

– $\mathsf{Gen}(1^\lambda, 1^s)$: On input the security parameter λ, and a bound on the circuit size s, the setup algorithm starts by sampling a PRF key $K \leftarrow \mathsf{PRF.Setup}(1^\lambda)$. The setup algorithm then defines the proving program $\mathsf{Prove}[K]$ and the verification program $\mathsf{Verify}[K]$ as follows:

Constants: PRF key K
Input: Boolean circuit C of size at most s, instance number $i \in [2^\lambda]$, witness w_i, proof $\pi \in \{0,1\}^\lambda$

1. If $i = 1$ and $C(1, w_1) = 1$, output $\mathsf{PRF.Eval}(K, (C, 1))$.
2. Else if $\mathsf{PRG}(\pi) = \mathsf{PRG}(\mathsf{PRF.Eval}(K, (C, i-1)))$ and $C(i, w_i) = 1$, output $\mathsf{PRF.Eval}(K, (C, i))$.
3. Otherwise, output $\perp$.

Fig. 1: Program $\mathsf{Prove}[K]$

Constants: PRF key K
Input: Boolean circuit C of size at most s, instance count $t \in [2^\lambda]$, proof $\pi \in \{0,1\}^\lambda$

1. If $\mathsf{PRG}(\pi) = \mathsf{PRG}(\mathsf{PRF.Eval}(K, (C, t)))$, output 1.
2. Otherwise, output 0.

Fig. 2: Program $\mathsf{Verify}[K]$

The setup algorithm constructs $\mathsf{ObfProve} \leftarrow i\mathcal{O}(1^\lambda, \mathsf{Prove}[K])$ and $\mathsf{ObfVerify} \leftarrow i\mathcal{O}(1^\lambda, \mathsf{Verify}[K])$. Note that both the proving circuit $\mathsf{Prove}[K]$ and $\mathsf{Verify}[K]$ are padded to the maximum size of any circuit that appears in the proof of Theorem 3.3. Finally, it outputs the common reference string $\mathsf{crs} = (\mathsf{ObfProve}, \mathsf{ObfVerify})$.

– $\mathsf{P}(\mathsf{crs}, C, (w_1, \ldots, w_t))$: On input $\mathsf{crs} = (\mathsf{ObfProve}, \mathsf{ObfVerify})$, a Boolean circuit $C\colon \{0,1\}^\lambda \times \{0,1\}^m \to \{0,1\}$, and a collection of witnesses $w_1, \ldots, w_t \in \{0,1\}^m$, the prover algorithm does the following:
  - Compute $\pi_1 \leftarrow \mathsf{ObfProve}(C, 1, w_1, \perp)$.
  - For $i = 2, \ldots, t$, compute $\pi_i \leftarrow \mathsf{ObfProve}(C, i, w_i, \pi_{i-1})$.
  - Output π_t.
– $\mathsf{V}(\mathsf{crs}, C, t, \pi)$: On input $\mathsf{crs} = (\mathsf{ObfProve}, \mathsf{ObfVerify})$, a Boolean circuit $C\colon \{0,1\}^\lambda \times \{0,1\}^m \to \{0,1\}$, the instance count $t \in [2^\lambda]$, and a proof $\pi \in \{0,1\}^\lambda$, the verification algorithm outputs $\mathsf{ObfVerify}(C, t, \pi)$.

Completeness and security analysis. We now state the completeness and security properties of Construction 3.1, but defer their proofs to the full version of this paper.

Theorem 3.2 (Completeness). *If $i\mathcal{O}$ is correct, then Construction 3.1 is complete.*

Theorem 3.3 (Soundness). *If* PRF *is functionality preserving and a secure puncturable PRF,* PRG *is a secure PRG, and $i\mathcal{O}$ is secure, then Construction 3.1 satisfies non-adaptive soundness.*

Theorem 3.4 (Succinctness). *Construction 3.1 is fully succinct.*

Theorem 3.5 (Zero Knowledge). *Construction 3.1 satisfies perfect zero knowledge.*

4 Non-Adaptive BARGs for NP from BARGs for Index Languages

In this section, we describe an adaptation of the compiler of Choudhuri et al. [CJJ21b] for upgrading a batch argument for index language to a batch argument for NP. The transformation of Choudhuri et al. relied on somewhere extractable commitments, which can be based on standard lattice assumptions [HW15, CJJ21b] or pairing-based assumptions [WW22]. Here, we show that the same transformation is possible using the positional accumulators introduced by Koppula et al. [KLW15]. The advantage of basing the transformation on positional accumulators is that we can construct positional accumulators directly from indistinguishability obfuscation and one-way functions, so we can apply the transformation to Construction 3.1 from Section 3 to obtain a fully succinct batch argument for NP from the *same* set of assumptions. A drawback of using positional accumulators in place of somewhere extractable commitments is that our transformation can only provide *non-adaptive* soundness, whereas the Choudhuri et al. transformation satisfies the stronger notion of *semi-adaptive* somewhere extractability.

Positional accumulators. Like a somewhere statistically binding (SSB) hash function [HW15], a positional accumulator allows a user to compute a short "digest" or "hash" y of a long input $(x_1, \ldots, x_t)$. The scheme supports local openings where the user can open y to the value x_i at any index i with a *short* opening π_i. The security property is that the hash value y is statistically binding at a certain (hidden) index i^*. An important difference between positional accumulators and somewhere statistically binding hash functions is that positional accumulators are statistically binding for the hash y of a *specific* tuple of inputs $(x_1, \ldots, x_t)$ while SSB hash functions are binding for *all* hash values. We give the definition below. Our definition is a simplification of the corresponding definition of Koppula et al. [KLW15, §4] and we summarize the main differences in Remark 4.3.

Definition 4.1 (Positional Accumulators [KLW15, adapted]). *Let $\ell \in \mathbb{N}$ be an input length. A positional accumulator scheme for inputs of length ℓ is a tuple of PPT algorithms $\Pi_{\mathsf{PA}} = (\mathsf{Setup}, \mathsf{SetupEnforce}, \mathsf{Hash}, \mathsf{Open}, \mathsf{Verify})$ with the following properties:*

- $\mathsf{Setup}(1^\lambda, 1^\ell) \to \mathsf{pp}$*: On input the security parameter* λ *and the input length* ℓ*, the setup algorithm outputs a set of public parameters* pp*.*
- $\mathsf{SetupEnforce}(1^\lambda, 1^\ell, (x_1, \ldots, x_t), i^*) \to \mathsf{pp}$*: On input the security parameter* λ*, an input length* ℓ*, a tuple of inputs* $x_1, \ldots, x_t \in \{0,1\}^\ell$*, and an index* $i^* \in [t]$*, the enforcing setup algorithm outputs a set of public parameters* pp*.*
- $\mathsf{Hash}(\mathsf{pp}, (x_1, \ldots, x_t)) \to y$*: On input the public parameters* pp*, a tuple of inputs* $x_1 \in \{0,1\}^\ell, \ldots, x_t \in \{0,1\}^\ell$*, the hash algorithm outputs a value* y*. This algorithm is deterministic.*
- $\mathsf{Open}(\mathsf{pp}, (x_1, \ldots, x_t), i) \to \pi$*: On input the public parameters* pp*, a tuple of inputs* $x_1 \in \{0,1\}^\ell, \ldots, x_t \in \{0,1\}^\ell$ *and an index* $i \in [t]$*, the opening algorithm outputs an opening* π*.*
- $\mathsf{Verify}(\mathsf{pp}, y, x, i, \pi) \to \{0,1\}$*: On input the public parameters* pp*, a hash value* y*, an input* $x \in \{0,1\}^\ell$*, an index* $i \in \{0,1\}^\lambda$*, and an opening* π*, the verification algorithm outputs a bit* $\{0,1\}$*.*

Moreover, the positional accumulator Π_{PA} *should satisfy the following properties:*

- **Correctness:** *For all security parameters* $\lambda \in \mathbb{N}$ *and input lengths* $\ell \in \mathbb{N}$*, all polynomials* $t = t(\lambda)$*, indices* $i \in [t]$*, and inputs* $x_1, \ldots, x_t \in \{0,1\}^\ell$*, it holds that*

$$\Pr\left[\mathsf{Verify}(\mathsf{pp}, y, x_i, i, \pi) = 1 : \begin{array}{r} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, 1^\ell), \\ y \leftarrow \mathsf{Hash}(\mathsf{pp}, (x_1, \ldots, x_t)), \\ \pi \leftarrow \mathsf{Open}(\mathsf{pp}, (x_1, \ldots, x_t), i) \end{array}\right] = 1.$$

- **Succinctness:** *The length of the hash value* y *output by* Hash *and the length of the proof* π *output by* Open *in the completeness experiment satisfy* $|y| = \mathsf{poly}(\lambda, \ell)$ *and* $|\pi| = \mathsf{poly}(\lambda, \ell)$*.*
- **Setup indistinguishability:** *For a security parameter* λ*, a bit* $b \in \{0,1\}$*, and an adversary* $\mathcal{A}$*, we define the setup-indistinguishability experiment as follows:*
  - *Algorithm* $\mathcal{A}$ *starts by choosing inputs* $x_1, \ldots, x_t \in \{0,1\}^\ell$*, and an index* $i \in [t]$*.*
  - *If* $b = 0$*, the challenger samples* $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, 1^\ell)$*. Otherwise, if* $b = 1$*, the challenger samples* $\mathsf{pp} \leftarrow \mathsf{SetupEnforce}(1^\lambda, 1^\ell, (x_1, \ldots, x_t), i)$*. It gives* pp *to* $\mathcal{A}$*.*
  - *Algorithm* $\mathcal{A}$ *outputs a bit* $b' \in \{0,1\}$*, which is the output of the experiment.*

  We say that Π_{PA} *satisfies* (τ, ε)*-setup-indistinguishability if for all adversaries running in time* $\tau = \tau(\lambda)$*, there exists* $\lambda_{\mathcal{A}} \in \mathbb{N}$ *such that for all* $\lambda > \lambda_{\mathcal{A}}$

$$|\Pr[b' = 1 \mid b = 0] - \Pr[b' = 1 \mid b = 1]| \leq \varepsilon(\lambda).$$

  in the setup-indistinguishability experiment.
- **Enforcing:** *Fix a security parameter* $\lambda \in \mathbb{N}$*, block size* $\ell \in \mathbb{N}$*, a polynomial* $t = t(\lambda)$*, an index* $i^* \in [t]$*, and a set of inputs* $x_1, \ldots, x_t$*. We say that a set of public parameters* pp *are "enforcing" for a tuple* $(x_1, \ldots, x_t, i^*)$ *if there does not exist a pair* (x, π) *where* $x \neq x_{i^*}$*,* $\mathsf{Verify}(\mathsf{pp}, y, x, i^*, \pi) = 1$*,*

and $y \leftarrow \mathsf{Hash}(\mathsf{pp}, (x_1, \ldots, x_t))$. *We say that the positional accumulator is enforcing if for every polynomial* $\ell = \ell(\lambda)$, $t = t(\lambda)$, *index* $i^* \in [t]$ *and inputs* $x_1, \ldots, x_t \in \{0,1\}^\ell$, *there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for all* $\lambda \in \mathbb{N}$,

$$\Pr[\mathsf{pp} \text{ is "enforcing" for } (x_1, \ldots, x_T, i^*) : \mathsf{pp} \leftarrow \mathsf{SetupEnforce}(1^\lambda, 1^\ell, (x_1, \ldots, x_t), i^*)] \geq 1 - \mathsf{negl}(\lambda),$$

where the probability is taken over the random coins of SetupEnforce.

Theorem 4.2 (Positional Accumulators [KLW15]). *Assuming the existence of an indistinguishability obfuscation scheme and one-way functions, there exists a positional accumulator for arbitrary polynomial input lengths* $\ell = \ell(\lambda)$.

Remark 4.3 (Comparison with [KLW15]). Definition 4.1 describes a simplified variant of the positional accumulator from Koppula et al. [KLW15, §4]. Specifically, we instantiate their construction with an (implicit) bound of $T = 2^\lambda$ for the number of values that can be accumulated. The positional accumulators from Koppula et al. also supports insertions (i.e., "writes") to the accumulator structure, whereas in our setting, all of the inputs are provided upfront (as an input to Hash).

Construction 4.4 (Batch Argument for NP Languages). Let λ be a security parameter and $s = s(\lambda)$ be a bound on the size of the Boolean circuit. We construct a BARG scheme that supports arbitrary NP languages with up to $T = 2^\lambda$ instances (i.e., which suffices to support an arbitrary polynomial number of instances) and Boolean circuits of size at most s. For ease of notation, we use the set $[2^\lambda]$ and the set $\{0,1\}^\lambda$ interchangably in the following description. Our construction relies on the following primitives:

- Let $\Pi_{\mathsf{PA}} = (\mathsf{PA.Setup}, \mathsf{PA.SetupEnforce}, \mathsf{PA.Hash}, \mathsf{PA.Open}, \mathsf{PA.Verify})$ be a positional accumulator for inputs of length ℓ.
- Let $\Pi_{\mathsf{IndexBARG}} = (\mathsf{IndexBARG.Gen}, \mathsf{IndexBARG.P}, \mathsf{IndexBARG.V})$ be a BARG for index languages (that supports up to $T = 2^\lambda$ instances).[7]

We define our batch argument $\Pi_{\mathsf{BARG}} = (\mathsf{Gen}, \mathsf{P}, \mathsf{V})$ for batch circuit satisfiability languages as follows:

- $\mathsf{Gen}(1^\lambda, 1^\ell, 1^s)$: On input the security parameter λ, the statement length ℓ, and a bound on the circuit size s, sample $\mathsf{pp} \leftarrow \mathsf{PA.Setup}(1^\lambda, 1^\ell)$. Let s' be a bound on the size of the following circuit:

[7] Our transformation also applies in the setting where the number of instances is bounded and the transformed scheme inherits the same bound. For simplicity of exposition, we just describe the transformation for the unbounded case.

Constants: Public parameters pp for Π_{PA}, a hash value h (for Π_{PA}), and a Boolean circuit C of size at most s
Inputs: Index $i \in \{0,1\}^{\lambda}$, a tuple (x, σ, w) where $x \in \{0,1\}^{\ell}$

1. If $C(x, w) = 0$, output 0.
2. If $\mathsf{PA.Verify}(\mathsf{pp}, h, x, i, \sigma) = 0$, output 0.
3. Otherwise, output 1.

Fig. 3: The Boolean circuit $C'[\mathsf{pp}, h, C]$ for an index relation

Then, sample $\mathsf{IndexBARG.crs} \leftarrow \mathsf{IndexBARG.Gen}(1^{\lambda}, 1^{s'})$. Output the common reference string $\mathsf{crs} = (\mathsf{pp}, \mathsf{IndexBARG.crs})$.

– $\mathsf{P}(\mathsf{crs}, C, (x_1, \ldots, x_t), (w_1, \ldots, w_t))$: On input the common reference string $\mathsf{crs} = (\mathsf{pp}, \mathsf{IndexBARG.crs})$, a Boolean circuit $C\colon \{0,1\}^{\ell} \times \{0,1\}^{m} \to \{0,1\}$, statements $x_1, \ldots, x_t \in \{0,1\}^{\ell}$, and witnesses $w_1, \ldots, w_t \in \{0,1\}^{m}$, compute $h \leftarrow \mathsf{PA.Hash}(\mathsf{pp}, (x_1, \ldots, x_t))$. Then, for each $i \in [t]$, let $\sigma_i \leftarrow \mathsf{PA.Open}(\mathsf{pp}, (x_1, \ldots, x_t), i)$ and let $w'_i = (x_i, \sigma_i, w_i)$. Output $\pi \leftarrow \mathsf{IndexBARG.P}(\mathsf{IndexBARG.crs}, C'[\mathsf{pp}, h, C], t, (w'_1, \ldots, w'_t))$, where $C'[\mathsf{pp}, h, C]$ is the circuit for the index relation from Fig. 3.
– $\mathsf{V}(\mathsf{crs}, C, (x_1, \ldots, x_t), \pi)$: On input the common reference string $\mathsf{crs} = (\mathsf{pp}, \mathsf{IndexBARG.crs})$, the Boolean circuit $C\colon \{0,1\}^{\ell} \times \{0,1\}^{m} \to \{0,1\}$, instances $x_1, \ldots, x_t \in \{0,1\}^{\ell}$, and a proof π, the verification algorithm computes $h \leftarrow \mathsf{PA.Hash}(\mathsf{pp}, (x_1, \ldots, x_t))$ and outputs $\mathsf{IndexBARG.V}(\mathsf{IndexBARG.crs}, C'[\mathsf{pp}, h, C], t, \pi)$, where $C'[\mathsf{pp}, h, C]$ is the circuit for the index relation from Fig. 3.

Completeness and security analysis. We now state the completeness and security properties of Construction 4.4, but defer their formal analysis to the full version of this paper.

Theorem 4.5 (Completeness). *If $\Pi_{\mathsf{IndexBARG}}$ is complete and Π_{PA} is correct, then Construction 4.4 is complete.*

Theorem 4.6 (Soundness). *Suppose $\Pi_{\mathsf{IndexBARG}}$ satisfies non-adaptive soundness, Π_{PA} satisfies setup-indistinguishability and is enforcing. Then, Construction 4.4 satisfies non-adaptive soundness.*

Theorem 4.7 (Succinctness). *If $\Pi_{\mathsf{IndexBARG}}$ is succinct (resp., fully succinct), Π_{PA} is efficient, then Construction 4.4 is succinct (resp., fully succinct).*

Theorem 4.8 (Zero Knowledge). *If $\Pi_{\mathsf{IndexBARG}}$ is perfect zero-knowledge, then Construction 4.4 is perfect zero-knowledge.*

Remark 4.9 (Weaker Notions of Zero Knowledge). If $\Pi_{\mathsf{IndexBARG}}$ satisfies computational (resp., statistical) zero-knowledge, then Construction 4.4 satisfies computational (resp., statistical) zero-knowledge. In other words, Construction 4.4 preserves the zero-knowledge property on the underlying index BARG.

5 Updatable Batch Argument for NP

We say that a BARG scheme is *updatable* if it supports an *a priori* unbounded number of statements (see Definition 2.7) and the prover algorithm is updatable. Formally, we replace the prover algorithm P in the BARG with an $\mathsf{UpdateP}$ algorithm. The $\mathsf{UpdateP}$ algorithm takes in statements $(x_1, \ldots, x_t)$, a proof π_t on these t statements, a new statement x_{t+1}, along with an associated witness w_{t+1}, and outputs an "updated" proof π_{t+1} on the new set of statements $(x_1, \ldots, x_{t+1})$. The updated proof should continue to satisfy the same succinctness requirements as before. We give the formal definition below:

Definition 5.1 (Updatable BARGs). *An updatable batch argument (BARG) for the language of Boolean circuit satisfiability consists of a tuple of PPT algorithms* $\Pi_{\mathsf{BARG}} = (\mathsf{Gen}, \mathsf{UpdateP}, \mathsf{V})$ *with the following properties:*

- $\mathsf{Gen}(1^\lambda, 1^\ell, 1^s) \to \mathsf{crs}$*: On input the security parameter* $\lambda \in \mathbb{N}$*, a bound on the instance size* $\ell \in \mathbb{N}$*, and a bound on the maximum circuit size* $s \in \mathbb{N}$*, the generator algorithm outputs a common reference string* crs*.*
- $\mathsf{UpdateP}(\mathsf{crs}, C, (x_1, \ldots, x_t), \pi_t, x_{t+1}, w_{t+1}) \to \pi_{t+1}$*: On input the common reference string* crs*, a Boolean circuit* $C\colon \{0,1\}^\ell \times \{0,1\}^m \to \{0,1\}$*, a list of statements* $x_1, \ldots, x_t \in \{0,1\}^\ell$*, a proof* π_t*, a new statement* $x_{t+1} \in \{0,1\}^\ell$ *and witness* $w_{t+1} \in \{0,1\}^m$*, the update algorithm outputs an updated proof* π_{t+1}*. Note that the list of statements* $(x_1, \ldots, x_t)$ *is allowed to be empty. We will write* $\perp$ *to denote an empty list of statements.*
- $\mathsf{V}(\mathsf{crs}, C, (x_1, \ldots, x_t), \pi) \to b$*: On input the common reference string* crs*, a Boolean circuit* $C\colon \{0,1\}^\ell \times \{0,1\}^m \to \{0,1\}$*, a list of statements* $x_1, \ldots, x_t \in \{0,1\}^\ell$*, and a proof* π*, the verification algorithm outputs a bit* $b \in \{0,1\}$*.*

An updatable BARG scheme should satisfy the following properties:

- **Completeness:** *For every security parameter* $\lambda \in \mathbb{N}$ *and bounds* $t \in \mathbb{N}$*,* $\ell \in \mathbb{N}$*, and* $s \in \mathbb{N}$*, Boolean circuits* $C\colon \{0,1\}^\ell \times \{0,1\}^m \to \{0,1\}$ *of size at most* s*, any collection of statements* $x_1, \ldots, x_t \in \{0,1\}^\ell$ *and associated witnesses* $w_1, \ldots, w_t \in \{0,1\}^m$ *where* $C(x_i, w_i) = 1$ *for all* $i \in [t]$*, we have that*

$$\Pr\left[\forall i \in [t] : \mathsf{V}(\mathsf{crs}, C, (x_1, \ldots, x_i), \pi_i) = 1 : \begin{array}{r} \mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda, 1^\ell, 1^s), \pi_0 \leftarrow \perp, \\ \pi_i \leftarrow \mathsf{UpdateP}(\mathsf{crs}, C, (x_1, \ldots, x_{i-1}), \pi_{i-1}, x_i, w_i) \\ \text{for all } i \in [t] \end{array}\right] = 1.$$

- **Succinctness:** *Similar to Definition 2.6, we require two succinctness properties:*
  - ***Succinct proof size:*** *There exists a universal polynomial* $\mathsf{poly}(\cdot,\cdot,\cdot)$*, such that for every* $i \in [t]$*,* $|\pi_i| = \mathsf{poly}(\lambda, \log i, s)$ *in the completeness experiment above.*
  - ***Succinct verification time:*** *There exists a universal polynomial* $\mathsf{poly}(\cdot,\cdot,\cdot)$ *such that for all* $i \in [t]$*, the verification algorithm* $\mathsf{V}(\mathsf{crs}, C, (x_1, \ldots, x_i), \pi_i)$ *runs in time* $\mathsf{poly}(\lambda, i, \ell) + \mathsf{poly}(\lambda, \log i, s)$ *in the completeness experiment above.*

- **Soundness:** *The soundness definition is defined exactly as in Definition 2.6.*
- **Perfect zero knowledge:** *The zero-knowledge definition is defined exactly as in Definition 2.6.*

5.1 Updatable BARGs for NP from Indistinguishability Obfuscation

We now give a direct construction of an updatable batch argument for NP languages from indistinguishability obfuscation together with somewhere statistically binding (SSB) hash functions [HW15]. We start with a construction that provides non-adaptive soundness. We then show to use complexity leveraging to obtain a construction with adaptive soundness.

Two-to-one somewhere statistically binding hash functions. Our construction will rely on a two-to-one somewhere statistically binding (SSB) hash function [OPWW15]. Informally, a two-to-one SSB hash function hashes two input blocks to an output whose size is comparable to the size of a single block. We recall the definition below:

Definition 5.2 (Two-to-One Somewhere Statistically Binding Hash Function [OPWW15]). *Let λ be a security parameter. A two-to-one somewhere statistically binding (SSB) hash function with block size $\ell_{\mathsf{blk}} = \ell_{\mathsf{blk}}(\lambda)$ and output size $\ell_{\mathsf{out}} = \ell_{\mathsf{out}}(\lambda, \ell_{\mathsf{blk}})$ is a tuple of efficient algorithms $\Pi_{\mathsf{SSB}} = (\mathsf{Gen}, \mathsf{GenTD}, \mathsf{LocalHash})$ with the following properties:*

- $\mathsf{Gen}(1^\lambda, 1^{\ell_{\mathsf{blk}}}) \to \mathsf{hk}$*: On input the security parameter λ and the block size ℓ_{blk}, the generator algorithm outputs a hash key* hk*.*
- $\mathsf{GenTD}(1^\lambda, 1^{\ell_{\mathsf{blk}}}, i^*) \to \mathsf{hk}$*: On input a security parameter λ, a block size ℓ_{blk}, and an index $i^* \in \{0,1\}$, the trapdoor generator algorithm outputs a hash key* hk*.*
- $\mathsf{LocalHash}(\mathsf{hk}, x_0, x_1) \to y$*: On input a hash key* hk *and two inputs $x_0, x_1 \in \{0,1\}^{\ell_{\mathsf{blk}}}$, the hash algorithm outputs a hash $y \in \{0,1\}^{\ell_{\mathsf{out}}}$.*

Moreover, Π_{SSB} should satisfy the following requirements:

- **Succinctness:** *The output length ℓ_{out} satisfies $\ell_{\mathsf{out}}(\lambda, \ell_{\mathsf{blk}}) = \ell_{\mathsf{blk}} \cdot (1 + 1/\Omega(\lambda)) + \mathsf{poly}(\lambda)$.*
- **Index hiding:** *For a security parameter λ, a bit $b \in \{0,1\}$, and an adversary $\mathcal{A}$, we define the index-hiding experiment as follows:*
  - *Algorithm $\mathcal{A}$ starts by choosing a block size ℓ_{blk}, and an index $i \in \{0,1\}$.*
  - *If $b = 0$, the challenger samples $\mathsf{hk}_0 \leftarrow \mathsf{Gen}(1^\lambda, 1^{\ell_{\mathsf{blk}}})$. Otherwise, if $b = 1$, the challenger samples $\mathsf{hk}_1 \leftarrow \mathsf{GenTD}(1^\lambda, 1^{\ell_{\mathsf{blk}}}, i)$. It gives hk_b to $\mathcal{A}$.*
  - *Algorithm $\mathcal{A}$ outputs a bit $b' \in \{0,1\}$, which is the output of the experiment.*

  We say that Π_{SSB} satisfies (τ, ε)-index-hiding, if for all adversaries running in time $\tau = \tau(\lambda)$, there exists $\lambda_{\mathcal{A}} \in \mathbb{N}$ such that for all $\lambda > \lambda_{\mathcal{A}}$, $|\Pr[b' = 1 \mid b = 0] - \Pr[b' = 1 \mid b = 1]| \leq \varepsilon(\lambda)$ in the index-hiding experiment.

– **Somewhere statistically binding:** *Let $\lambda \in \mathbb{N}$ be a security parameter and $\ell \in \mathbb{N}$ be an input length. We say a hash key hk is "statistically binding" at index $i \in \{0,1\}$, if there does not exist two inputs (x_0, x_1) and (x_0^*, x_1^*) such that $x_i^* \neq x_i$ and $\mathsf{Hash}(\mathsf{hk}, (x_0, x_1)) = \mathsf{Hash}(\mathsf{hk}, (x_0^*, x_1^*))$. We then say that the hash function is somewhere statistically binding if for all polynomials $\ell_{\mathsf{blk}} = \ell_{\mathsf{blk}}(\lambda)$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that for all indices $i^* \in \{0,1\}$ and all $\lambda \in \mathbb{N}$,*

  $$\Pr[\mathsf{hk} \text{ is statistically binding at index } i : \mathsf{hk} \leftarrow \mathsf{GenTD}(1^\lambda, 1^{\ell_{\mathsf{blk}}}, i)] \geq 1 - \mathsf{negl}(\lambda).$$

Theorem 5.3 (Somewhere Statistically-Binding Hash Functions [OPWW15]). *Under standard number-theoretic assumptions (e.g., DDH, DCR, LWE, or ϕ-Hiding), there exists a two-to-one somewhere statistically binding hash function for arbitrary polynomial block size $\ell_{\mathsf{blk}} = \ell_{\mathsf{blk}}(\lambda)$.*

Notation. Our updatable BARG construction uses a tree-based construction. Before describing the construction, we introduce some notation. First, for an integer $t < 2^d$, we write $\mathsf{bin}_d(t) \in \{0,1\}^d$ to denote the d-bit binary representation of t. For two strings $\mathsf{ind} \in \{0,1\}^*$, $\mathsf{ind}' \in \{0,1\}^*$, let $\mathsf{pad}(\mathsf{ind})$ and $\mathsf{pad}(\mathsf{ind}')$ be the respective strings padded with zeros to the length $\max\{|\mathsf{ind}|, |\mathsf{ind}'|\}$. We say $\mathsf{ind} \leq \mathsf{ind}'$ if $\mathsf{pad}(\mathsf{ind})$ comes before $\mathsf{pad}(\mathsf{ind}')$ lexicographically. For strings $s_1, s_2 \in \{0,1\}^*$, we write $s_1 \| s_2$ to denote their concatenation. We say that a string $x \in \{0,1\}^*$ is a prefix of a string $y \in \{0,1\}^*$ if there exists a string $z \in \{0,1\}^*$ such that $y = x \| z$.

Binary trees. A binary tree Γ of height d consists of nodes where each node is indexed by a binary string of length at most d. We now define a recursive labeling scheme for the nodes of the tree; subsequently, we will refer to nodes by their labels.

– **Root node:** The root node is labeled with the empty string ε.
– **Child nodes:** The left child of node ind has label $\mathsf{ind} \| 0$ and the right child has label $\mathsf{ind} \| 1$. We also say that node $\mathsf{ind} \| 0$ is the "left sibling" of the node $\mathsf{ind} \| 1$.

We define the *level* of a node ind by $\mathsf{level}(\mathsf{ind}) = d - |\mathsf{ind}|$. In particular, the root node is at level d while the leaf nodes are at level 0. We write $\{0,1\}^{\leq d}$ to denote the set of node labels associated in the binary tree (i.e., the set of all binary strings of length at most d). Finally, we can also associate each node in the binary tree with a value; formally, for a binary tree Γ we write $\mathsf{val}(\mathsf{ind})$ to denote the value associated with the node ind. When we write $(\Gamma, \mathsf{val}(\cdot))$, we imply our binary tree has been initialized with the corresponding value function. Finally, we define the notion of a "path" and a "frontier" of a node in a binary tree Γ:

– **Path of a node:** We define the path associated with a node $\mathsf{ind} \in \{0,1\}^{\leq d}$ as

  $$\mathsf{path}(\mathsf{ind}) = \{\mathsf{ind}' \mid \mathsf{ind}' \in \{0,1\}^{\leq d} \text{ and } \mathsf{ind}' \text{ is a prefix of } \mathsf{ind}\}.$$

  Namely, $\mathsf{path}(\mathsf{ind})$ consists of the nodes along the path from the root to ind.

– **Frontier of a node:** For any $\mathsf{ind} \in \{0,1\}^{\leq d}$, we define

$$\mathsf{frontier}(\mathsf{ind}) = \{\mathsf{ind}\} \cup \{\mathsf{ind}' \in \{0,1\}^{\leq d} \mid \mathsf{ind}' \text{ is a left sibling of a node in } \mathsf{path}(\mathsf{ind})\}.$$

Construction 5.4 (Non-Adaptive Updatable Batch Argument for NP). Let λ be a security parameter, $\ell = \ell(\lambda)$ be the statement size, and $s = s(\lambda)$ be a bound on the size of the Boolean circuit. We construct an updatable BARG scheme that supports NP languages with up to $T = 2^\lambda$ instances of length ℓ and circuit size at most s. Note that setting $T = 2^\lambda$ means the construction support an arbitrary polynomial number of instances. Our construction relies on the following primitives:

– Let $\Pi_{\mathsf{SSB}} = (\mathsf{SSB.Gen}, \mathsf{SSB.GenTD}, \mathsf{SSB.LocalHash})$ be a two-to-one somewhere statistically binding hash function with output length $\ell_{\mathsf{out}} = \ell_{\mathsf{out}}(\lambda, \ell_{\mathsf{blk}})$, where ℓ_{blk} denotes the block length. Our construction will consider a binary tree of depth $d = \lambda$, and we define a sequence of block lengths $\ell_0, \ldots, \ell_d$ where $\ell_0 = \ell$ and for $j \in [d]$, let $\ell_j = \ell_{\mathsf{out}}(\lambda, \ell_{j-1})$.[8] Let $\ell_{\max} = \max(\ell_0, \ldots, \ell_j)$.

– Let $\Pi_{\mathsf{PRF}} = (\mathsf{PRF.Setup}, \mathsf{PRF.Puncture}, \mathsf{PRF.Eval})$ be a puncturable PRF with key space $\{0,1\}^\lambda$, domain $\{0,1\}^{\leq s} \times \{0,1\}^{\leq \ell_{\max}} \times \{0,1\}^d$ and range $\{0,1\}^\lambda$.

– Let $i\mathcal{O}$ be an indistinguishability obfuscator for general circuits.

– Let PRG be a pseudorandom generator with domain $\{0,1\}^\lambda$ and range $\{0,1\}^{2\lambda}$.

We define our updatable batch argument $\Pi_{\mathsf{BARG}} = (\mathsf{Gen}, \mathsf{UpdateP}, \mathsf{Verify})$ for NP languages as follows:

– $\mathsf{Gen}(1^\lambda, 1^\ell, 1^s)$: On input the security parameter λ, the statement size ℓ, and a bound on the circuit size s, the setup algorithm starts by sampling a PRF key $K \leftarrow \mathsf{PRF.Setup}(1^\lambda)$. For $j \in [d]$, sample $\mathsf{hk}_j \leftarrow \mathsf{SSB.Gen}(1^\lambda, 1^{\ell_{j-1}})$, Let $\mathsf{hk} \leftarrow (\mathsf{hk}_1, \ldots, \mathsf{hk}_d)$ and define the proving program $\mathsf{Prove}[K, \mathsf{hk}]$ and the verification program $\mathsf{Verify}[K]$ as follows:

[8] Formally, our hash function will take inputs in $\{0,1\}^{\ell_{j-1}} \cup \{\perp\}$. For ease of exposition, we drop the special input symbol $\perp$ in our block length description.

Constants: PRF key K, hash keys $\mathsf{hk} = (\mathsf{hk}_1, \ldots, \mathsf{hk}_d)$
Input: Boolean circuit $C\colon \{0,1\}^\ell \times \{0,1\}^m \to \{0,1\}$ of size at most s, node values $h_1, h_2 \in \{0,1\}^{\leq \ell_{\max}}$, index $\mathsf{ind} \in \{0,1\}^{\leq d}$, and proofs $\pi_1, \pi_2 \in \{0,1\}^{\leq \max(m,\lambda)}$

1. If $\mathsf{ind} \in \{0,1\}^d$ (i.e., a leaf in the binary tree),
   (a) Parse h_1 as a statement $x_1 \in \{0,1\}^\ell$ and π_1 as a witness $w_1 \in \{0,1\}^m$.
   (b) If $C(x_1, w_1) \neq 1$, output $\perp$. Otherwise, output $\mathsf{PRF.Eval}(K, (C, x_1, \mathsf{ind}))$.
2. Otherwise, if $\mathsf{ind} \in \{0,1\}^{<d}$ (i.e., an internal node in the binary tree),
   (a) Let $d' = \mathsf{level}(\mathsf{ind})$ and compute $h \leftarrow \mathsf{SSB.LocalHash}(\mathsf{hk}_{d'}, h_1, h_2)$.
   (b) Check the following conditions:
   - $\mathsf{PRG}(\pi_1) = \mathsf{PRG}(\mathsf{PRF.Eval}(K, (C, h_1, \mathsf{ind}\|0)))$;
   - $\mathsf{PRG}(\pi_2) = \mathsf{PRG}(\mathsf{PRF.Eval}(K, (C, h_2, \mathsf{ind}\|1)))$.

   If either check fails, output $\perp$. Otherwise, output $\mathsf{PRF.Eval}(K, (C, h, \mathsf{ind}))$.

Fig. 4: Program $\mathsf{Prove}[K, \mathsf{hk}]$

Constants: PRF key K
Input: Boolean circuit C of size at most s, node value $h \in \{0,1\}^{\leq \ell_{\max}}$, index $\mathsf{ind} \in \{0,1\}^{\leq d}$, a proof $\pi \in \{0,1\}^\lambda$

1. Output 1 if $\mathsf{PRG}(\pi) = \mathsf{PRG}(\mathsf{PRF.Eval}(K, (C, h, \mathsf{ind})))$ and 0 otherwise.

Fig. 5: Program $\mathsf{Verify}[K]$

The setup algorithm obfuscates the above programs to obtain $\mathsf{ObfProve} \leftarrow i\mathcal{O}(1^\lambda, \mathsf{Prove}[K, \mathsf{hk}])$ and $\mathsf{ObfVerify} \leftarrow i\mathcal{O}(1^\lambda, \mathsf{Verify}[K])$. Note that both the proving circuit $\mathsf{Prove}[K, \mathsf{hk}]$ and $\mathsf{Verify}[K]$ are padded to the maximum size of any circuit that appears in the proof of Theorem 5.6. Finally, it outputs the common reference string $\mathsf{crs} = (\mathsf{ObfProve}, \mathsf{ObfVerify}, \mathsf{hk})$.

– $\mathsf{UpdateP}(\mathsf{crs}, C, (x_1, \ldots, x_t), \pi_t, x_{t+1}, w_{t+1})$: On input a common reference string $\mathsf{crs} = (\mathsf{ObfProve}, \mathsf{ObfVerify}, \mathsf{hk})$, a Boolean circuit $C\colon \{0,1\}^\ell \times \{0,1\}^m \to \{0,1\}$, a set of statements $x_1, \ldots, x_t, x_{t+1} \in \{0,1\}^\ell$, a proof $\pi_t = \{(\mathsf{ind}, \pi_{\mathsf{ind}})\}_{\mathsf{ind} \in \mathcal{I}}$ on the first t statements where $\mathcal{I} \subset \{0,1\}^{\leq d}$, and a witness $w_{t+1} \in \{0,1\}^m$, the update algorithm proceeds as follows:
  1. If $t = 0$, let $\mathsf{ind}^{(1)} = \mathsf{bin}_d(0) = 0^d$. Let $\pi \leftarrow \mathsf{ObfProve}(C, x_1, \perp, \mathsf{ind}^{(1)}, w_1, \perp)$ and output $\{(\mathsf{ind}^{(1)}, \pi)\}$.
  2. Otherwise, if $t \neq 0$, the update algorithm computes $\mathsf{ind}^{(t)} = \mathsf{bin}_d(t-1)$ and checks that $\mathsf{frontier}(\mathsf{ind}^{(t)}) = \mathcal{I}$. If the check fails, then the update algorithm outputs $\perp$.
  3. Next, the update algorithm constructs a binary tree $(\Gamma_{\mathsf{hash}}, \mathsf{val}_{\mathsf{hash}}) \leftarrow \mathsf{Hash}[\mathsf{hk}](x_1, \ldots, x_t)$ of depth d whose values correspond to the statements $(x_1, \ldots, x_t)$ and their hashes. Specifically, we define the $\mathsf{Hash}[\mathsf{hk}]$ function as follows:

Constants: Hash key $\mathsf{hk} = (\mathsf{hk}_1, \ldots, \mathsf{hk}_d)$
Input: Statements $x_1, \ldots, x_t \in \{0,1\}^\ell$

On input a collection of statements $(x_1, \ldots, x_t)$, the hash algorithm constructs a binary tree Γ_{hash} of depth d with a value function $\mathsf{val}_{\mathsf{hash}}$ defined recursively as follows:

- **Leaf nodes:** For a leaf node $\mathsf{ind} \in \{0,1\}^d$, let $i \in [0, 2^d - 1]$ be its associated value (when viewed as an integer). Then, associate the value $\mathsf{val}_{\mathsf{hash}}(\mathsf{ind})$ as follows:

$$\mathsf{val}_{\mathsf{hash}}(\mathsf{ind}) = \begin{cases} x_{i+1} & i+1 \le t \\ \bot & \text{otherwise.} \end{cases}$$

- **Internal nodes:** For an internal node $\mathsf{ind} \in \{0,1\}^{<d}$, we define its value as follows:
  - For all indices where $\mathsf{ind} > \mathsf{bin}_d(t-1)$, define $\mathsf{val}_{\mathsf{hash}}(\mathsf{ind}) \leftarrow \bot$.
  - If $\mathsf{ind} \le \mathsf{bin}_d(t-1)$, define $\mathsf{val}_{\mathsf{hash}}(\mathsf{ind})$ to be the hash of its children $\mathsf{ind}\|0$ and $\mathsf{ind}\|1$ computed using $\mathsf{hk}_{d'}$, where $d' = \mathsf{level}(\mathsf{ind})$. Namely,

$$\mathsf{val}_{\mathsf{hash}}(\mathsf{ind}) \leftarrow \mathsf{SSB.LocalHash}\big(\mathsf{hk}_{d'}, \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}\|0), \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}\|1)\big).$$

We assume that $\mathsf{val}_{\mathsf{hash}}$ is efficiently encoded such that on any input $\mathsf{ind} \in \{0,1\}^{\le d}$, where $\mathsf{ind} > \mathsf{bin}_d(t-1)$, $\mathsf{val}_{\mathsf{hash}}$ outputs $\bot$ and initialized on nodes $\le \mathsf{bin}_d(t-1)$ and thus initialized on $\le 2t$ nodes.
Output $(\Gamma_{\mathsf{hash}}, \mathsf{val}_{\mathsf{hash}})$.

Fig. 6: The function $\mathsf{Hash}[\mathsf{hk}](x_1, \ldots, x_t)$

Essentially, $\mathsf{Hash}[\mathsf{hk}]$ computes a Merkle tree on the statements $(x_1, \ldots, x_t)$.

4. The update algorithm then defines a binary tree Γ_{proof} of depth d with the following value function $\mathsf{val}_{\mathsf{proof}}$:
   - For each index $\mathsf{ind} \in \mathcal{I}$, let $\mathsf{val}_{\mathsf{proof}}(\mathsf{ind}) = \pi_{\mathsf{ind}}$.
   - Let $\mathsf{ind}^{(t+1)} = \mathsf{bin}_d(t)$. Let $\mathsf{val}_{\mathsf{proof}}(\mathsf{ind}^{(t+1)}) = \mathsf{ObfProve}(C, x_{t+1}, \bot, \mathsf{ind}^{(t+1)}, w_{t+1}, \bot)$.
   - For all other nodes $\mathsf{ind} \notin \mathcal{I}$, let $\mathsf{val}_{\mathsf{proof}}(\mathsf{ind}) = \bot$.

   The invariant will be that the nodes ind associated with the frontier of leaf node t (with index $\mathsf{bin}_d(t-1)$) are associated with a proof π_{ind}.
5. Let ind' be the longest common prefix to $\mathsf{ind}^{(t)}$ and $\mathsf{ind}^{(t+1)}$. Write $\mathsf{ind}^{(t)} = b_1 \cdots b_d$ and $\mathsf{ind}' = b_1 \cdots b_\rho$, where $\rho = |\mathsf{ind}'|$ denotes the length of the common prefix. If $\rho < d-1$, then we apply the following procedure for $k = d-1, \ldots, \rho+1$ to merge proofs:
   - Let $\mathsf{ind} = b_1 \cdots b_k$ and compute

$$\mathsf{val}_{\mathsf{proof}}(\mathsf{ind}) \leftarrow \mathsf{ObfProve}\left(C, h_1, h_2, \mathsf{ind}, \mathsf{val}_{\mathsf{proof}}(\mathsf{ind}\|0), \mathsf{val}_{\mathsf{proof}}(\mathsf{ind}\|1)\right),$$

     where $h_1 \leftarrow \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}\|0)$ and $h_2 \leftarrow \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}\|1)$.
6. Output the updated proof, $\pi_{t+1} = \{(\mathsf{ind}, \mathsf{val}_{\mathsf{proof}}(\mathsf{ind}))\}_{\mathsf{ind} \in \mathsf{frontier}(\mathsf{ind}^{(t+1)})}$.

– $\mathsf{V}(\mathsf{crs}, C, (x_1, \ldots, x_t), \pi)$: On input $\mathsf{crs} = (\mathsf{ObfProve}, \mathsf{ObfVerify}, \mathsf{hk})$, a Boolean circuit $C\colon \{0,1\}^\ell \times \{0,1\}^m \to \{0,1\}$, statements $x_1, \ldots, x_t \in \{0,1\}^\ell$ and a proof $\pi = \{(\mathsf{ind}, \pi_{\mathsf{ind}})\}_{\mathsf{ind} \in \mathcal{I}}$, the verification algorithm proceeds as follows:
  1. The algorithm constructs a binary tree $(\Gamma_{\mathsf{hash}}, \mathsf{val}_{\mathsf{hash}}) \leftarrow \mathsf{Hash}[\mathsf{hk}](x_1, \ldots, x_t)$ (defined in Fig. 6) of depth d whose values correspond to the statements $(x_1, \ldots, x_t)$ and their hashes.
  2. Let $\mathsf{ind}^{(t)} = \mathsf{bin}_d(t-1)$. If $\mathcal{I} \neq \mathsf{frontier}(\mathsf{ind}^{(t)})$, output $\perp$.
  3. Finally, the verification algorithm checks that $\mathsf{ObfVerify}(C, \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}), \mathsf{ind}, \pi_{\mathsf{ind}}) = 1$ for all $\mathsf{ind} \in \mathsf{frontier}(\mathsf{ind}^{(t)})$. If any checks fail, output 0. Otherwise output 1.

Completeness and security analysis. We now state the completeness and security properties of Construction 5.4, but defer their proofs to the full version of this paper.

Theorem 5.5 (Completeness). *If $i\mathcal{O}$ is correct, then Construction 5.4 is complete.*

Theorem 5.6 (Soundness). *If Π_{PRF} is correct and a secure puncturable PRF, PRG is a secure PRG, Π_{SSB} is a secure statistically binding two-to-one SSB hash and $i\mathcal{O}$ is secure, then Construction 5.4 satisfies non-adaptive soundness.*

Theorem 5.7 (Succinctness). *If Π_{SSB} is succinct, then, Construction 5.4 is fully succinct.*

Theorem 5.8 (Zero-knowledge). *Construction 5.4 satisfies perfect zero-knowledge.*

Combining Theorems 5.5 to 5.8, we obtain the following corollary:

Corollary 5.9 (Non-Adaptive Updatable BARGs). *Assuming the existence of a secure indistinguishability obfuscation scheme and of somewhere extractable hash functions, there exists an updatable batch argument for NP.*

Acknowledgments

We thank the anonymous TCC reviewers for helpful feedback on this work. B. Waters is supported by NSF CNS-1908611, a Simons Investigator award, and the Packard Foundation Fellowship. D. J. Wu is supported by NSF CNS-2151131, CNS-2140975, a Microsoft Research Faculty Fellowship, and a Google Research Scholar award.

References

ACL+22. Martin R. Albrecht, Valerio Cini, Russell W. F. Lai, Giulio Malavolta, and Sri AravindaKrishnan Thyagarajan. Lattice-based SNARKs: Publicly verifiable, preprocessing, and recursively composable. In *CRYPTO*, 2022.

AS15. Gilad Asharov and Gil Segev. Limits on the power of indistinguishability obfuscation and functional encryption. In *FOCS*, pages 191–209, 2015.

BBHR18. Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *IACR Cryptol. ePrint Arch.*, 2018, 2018.

BCC+17. Nir Bitansky, Ran Canetti, Alessandro Chiesa, Shafi Goldwasser, Huijia Lin, Aviad Rubinstein, and Eran Tromer. The hunting of the SNARK. *J. Cryptol.*, 30(4), 2017.

BCCT12. Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *ITCS*, 2012.

BCCT13. 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.

BCI+13. Nir Bitansky, Alessandro Chiesa, Yuval Ishai, Rafail Ostrovsky, and Omer Paneth. Succinct non-interactive arguments via linear interactive proofs. In *TCC*, 2013.

BCPR14. Nir Bitansky, Ran Canetti, Omer Paneth, and Alon Rosen. On the existence of extractable one-way functions. In *STOC*, 2014.

BGI+01. Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *CRYPTO*, pages 1–18, 2001.

BGI14. Elette Boyle, Shafi Goldwasser, and Ioana Ivan. Functional signatures and pseudorandom functions. In *PKC*, pages 501–519, 2014.

BISW17. Dan Boneh, Yuval Ishai, Amit Sahai, and David J. Wu. Lattice-based SNARGs and their application to more efficient obfuscation. In *EUROCRYPT*, 2017.

BISW18. Dan Boneh, Yuval Ishai, Amit Sahai, and David J. Wu. Quasi-optimal snargs via linear multi-prover interactive proofs. In *EUROCRYPT*, pages 222–255, 2018.

BW13. Dan Boneh and Brent Waters. Constrained pseudorandom functions and their applications. In *ASIACRYPT*, pages 280–300, 2013.

CHM+20. Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas P. Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In *EUROCRYPT*, 2020.

CJJ21a. Arka Rai Choudhuri, Abhishek Jain, and Zhengzhong Jin. Non-interactive batch arguments for NP from standard assumptions. In *CRYPTO*, pages 394–423, 2021.

CJJ21b. Arka Rai Choudhuri, Abhishek Jain, and Zhengzhong Jin. Snargs for $\mathcal{P}$ from LWE. In *FOCS*, pages 68–79, 2021.

COS20. Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. In *EUROCRYPT*, 2020.

DFH12. Ivan Damgård, Sebastian Faust, and Carmit Hazay. Secure two-party computation with low communication. In *TCC*, 2012.

DGKV22. Lalita Devadas, Rishab Goyal, Yael Kalai, and Vinod Vaikuntanathan. Rate-1 non-interactive arguments for batch-NP and applications. *IACR Cryptol. ePrint Arch.*, 2022, 2022.

GGPR13. Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *EUROCRYPT*, 2013.

Gro10. Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In *ASIACRYPT*, 2010.

Gro16. Jens Groth. On the size of pairing-based non-interactive arguments. In *EUROCRYPT*, 2016.

GW11. Craig Gentry and Daniel Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In *STOC*, pages 99–108, 2011.

HW15. Pavel Hubácek and Daniel Wichs. On the communication complexity of secure function evaluation with long output. In *ITCS*, pages 163–172, 2015.

JLS21. Aayush Jain, Huijia Lin, and Amit Sahai. Indistinguishability obfuscation from well-founded assumptions. In *STOC*, pages 60–73, 2021.

JLS22. Aayush Jain, Huijia Lin, and Amit Sahai. Indistinguishability obfuscation from LPN over f_p, dlin, and prgs in nc^0. In *EUROCRYPT*, 2022.

KLW15. Venkata Koppula, Allison Bishop Lewko, and Brent Waters. Indistinguishability obfuscation for turing machines with unbounded memory. In *STOC*, pages 419–428, 2015.

KPTZ13. Aggelos Kiayias, Stavros Papadopoulos, Nikos Triandopoulos, and Thomas Zacharias. Delegatable pseudorandom functions and applications. In *ACM CCS*, pages 669–684, 2013.

Lip13. Helger Lipmaa. Succinct non-interactive zero knowledge arguments from span programs and linear error-correcting codes. In *ASIACRYPT*, 2013.

Mer87. Ralph C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO*, pages 369–378, 1987.

Mic95. Silvio Micali. Computationally-sound proofs. In *Proceedings of the Annual European Summer Meeting of the Association of Symbolic Logic*, 1995.

Nao03. Moni Naor. On cryptographic assumptions and challenges. In *CRYPTO*, 2003.

OPWW15. Tatsuaki Okamoto, Krzysztof Pietrzak, Brent Waters, and Daniel Wichs. New realizations of somewhere statistically binding hashing and positional accumulators. In *ASIACRYPT*, pages 121–145, 2015.

PHGR13. Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, 2013.

Set20. Srinath T. V. Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In *CRYPTO*, 2020.

SW14. Amit Sahai and Brent Waters. How to use indistinguishability obfuscation: deniable encryption, and more. In *STOC*, 2014.

WW22. Brent Waters and David J. Wu. Batch arguments for NP and more from standard bilinear group assumptions. In *CRYPTO*, 2022.