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]UT Austin
{rachg96, kristin, bwaters, dwu4}@cs.utexas.edu
[2]NTT Research

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 size $|\pi|$ 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 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 for 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.[1] 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.

[1] 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.

Our results. 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 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 and somewhere statistically 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 a high-level 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 on (C, x) if $C(x, w) = 1$ and $\perp$ otherwise. The proof is simply the signature $\pi = \sigma$. The Verify program takes in the description of the circuit C, the statement x, and the proof $\pi = \sigma$, and checks whether σ is a valid signature on (C, x) or not. The signature in this case just corresponds to the evaluation of a pseudorandom function (PRF) on the input (C, 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 that $C(i, w_i) = 1$ and that the proof π 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],[2] 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 it with the check

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

  where G is a length-doubling pseudorandom generator (PRG). 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 the PRF F is $\{0,1\}^\lambda$ and suppose that $\mathsf{G}\colon \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$ is a length-doubling PRG.

- 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.

[2] 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.

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 CRS.

- 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:

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

– 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 $\bot$.
	– 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 $i\mathcal{O}$:

$\mathsf{Prove}(C, i, w_i, \pi)$:	$\mathsf{Verify}(C, i, \pi)$:
– 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 $\bot$.	– 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 $\bot$.
– If $C = C^*$ and $i^* \leq i \leq i^* + 1$, output $\bot$.
– 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 $\bot$.
– If $\mathsf{G}(\pi) = \mathsf{G}(\mathsf{F}(k', (C, i-1)))$, output $\mathsf{F}(k', (C, i))$.
– Output $\bot$.
	– 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 $\bot$ 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 $i^* \le i \le i^* + 2$, 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 $i^* \le i \le i^* + 1$, 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 $T = \mathrm{poly}(\lambda)$, 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 $i^* \le i \le T + 1$, 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 $i^* \le i \le T$, 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 $i^* \le i \le T$. 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 Appendix A. 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. Moreover, in the case of general NP languages, our adaptively-sound construction has an *expensive* verification procedure (i.e., which runs in time $\mathrm{poly}(\lambda, T, s)$, where T is the number of instances and s is the size of the underlying NP relation). 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].[3] 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:

- 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:[4]
  - 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 $(k, h', \sigma') \leftarrow (k + 1, H(h_i, h'), \sigma'')$.

[3] If the underlying BARG is not rate-1, then we can only compose a *bounded* number of times.
[4] In our formal construction (Section 5), we defer the "merging" step to the subsequent update.

– 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 efficient if it runs in probabilistic polynomial time (PPT) in the length of its input. Throughout this work, we consider security against *non-uniform* adversaries (indexed by λ) that run in *deterministic* polynomial time in the length of their input and takes in an advice string of $\mathsf{poly}(\lambda)$ size.[5]

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 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)[6] 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 that

$$\left|\Pr[\mathcal{A}(i\mathcal{O}(1^\lambda, C_0)) = 1] - \Pr[\mathcal{A}(i\mathcal{O}(1^\lambda, C_1)) = 1]\right| \leq \varepsilon(\lambda).$$

[5] Recall that in the non-uniform model, we can derandomize any adversary by fixing its random coins to the choice that maximizes the adversary's advantage; this fixed set of coins is in turn provided to the adversary as *advice*.

[6] In Remark 3.10, we clarify why we rely on hardness against non-uniform adversaries in our constructions.

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 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 pseudorandomness 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 that

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

2.1 Batch Arguments for NP

We now recall 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 $C\colon \{0,1\}^\ell \times \{0,1\}^m \to \{0,1\}$, a list of statements $x_1, \ldots, x_t \in \{0,1\}^\ell$, and a 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 $C\colon \{0,1\}^\ell \times \{0,1\}^m \to \{0,1\}$, a list of statements $x_1, \ldots, x_t \in \{0,1\}^\ell$, and a proof π, the verification algorithm outputs a bit $b \in \{0,1\}$.

Moreover, the BARG scheme should satisfy the following properties:

- **Completeness:** For all security parameters $\lambda \in \mathbb{N}$ and bounds $\ell \in \mathbb{N}$, $s \in \mathbb{N}$, $T \in \mathbb{N}$, $t \le T$, Boolean circuits $C\colon \{0,1\}^\ell \times \{0,1\}^m \to \{0,1\}$ 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

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

- **Succinctness:** We require Π_{BARG} satisfy two notions of succinctness:
  - **Succinct proof size:** There exists a universal polynomial $\mathsf{poly}(\cdot,\cdot,\cdot)$ such that for all $t \le T$, $|\pi| = \mathsf{poly}(\lambda, \log t, s)$ in the completeness experiment defined above. We say the proof is *fully succinct* if for all $t \le T$, we have that $|\pi| = \mathsf{poly}(\lambda, \log t, \log s)$.
  - **Succinct verification time:** There exists a universal polynomial $\mathsf{poly}(\cdot,\cdot,\cdot)$ such that for all $t \le T$, the running time of $\mathsf{V}(\mathsf{crs}, C, (x_1, \ldots, x_t), \pi)$ is bounded by $\mathsf{poly}(\lambda, t, \ell) + \mathsf{poly}(\lambda, \log t, s)$ in the completeness experiment defined above.
- **Soundness:** We consider two different notions of soundness:
  - **Non-adaptive soundness:** For a security parameter λ, we define the non-adaptive soundness experiment between a challenger and an adversary $\mathcal{A}$ as follows:
    - Algorithm $\mathcal{A}$ outputs a bound on the number of instances 1^T, the maximum circuit size 1^s, a Boolean circuit $C\colon \{0,1\}^\ell \times \{0,1\}^m \to \{0,1\}$ of size at most $s(\lambda)$ and statements $x_1, \ldots, x_t \in \{0,1\}^\ell$. Here, we require that $t \le T$.
    - 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 proof π.
    - 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 non-adaptive soundness if for every non-uniform polynomial time adversary $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that $\Pr[b = 1] = \mathsf{negl}(\lambda)$ in the non-adaptive soundness experiment.
  - **Adaptive soundness:** For a security parameter λ, we define the adaptive soundness experiment between a challenger and an adversary $\mathcal{A}$ as follows:
    - Algorithm $\mathcal{A}$ outputs a bound on the number of instances 1^T, the maximum circuit size 1^s, and the input size 1^ℓ.
    - 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 $t \le T$.

* 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 $\mathcal{A}$, 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 $\pi \leftarrow \mathsf{P}(\mathsf{crs}, C, (x_1, \ldots, x_t), (w_1, \ldots, w_t))$. Output (crs, π).
  - **Simulated distribution:** Output $(\mathsf{crs}^*, \pi^*) \leftarrow \mathcal{S}(1^\lambda, 1^T, 1^s, C, (x_1, \ldots, x_t))$.

Definition 2.7 (BARGs for Arbitrary Number of Statements)**.** We say that a BARG scheme $\Pi_{\mathsf{BARG}} = (\mathsf{Gen}, \mathsf{P}, \mathsf{V})$ supports an arbitrary polynomial number of statements if the algorithm Gen in Definition 2.6 runs in time $\mathsf{poly}(\lambda, \ell, s, \log T)$, and correspondingly, outputs 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.[7]

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 $(x_1, \ldots, x_t)$. Similarly, the generator algorithm Gen only needs the security parameter and the bound on the circuit size s; the bound on the instance size is simply λ (to support up to $T = 2^\lambda$ instances). Specifically, we modify the syntax as follows:

- $\mathsf{Gen}(1^\lambda, 1^s) \to \mathsf{crs}$: On input the security parameter λ and a bound on the circuit size s, the generator algorithm outputs a common reference string crs.
- $\mathsf{P}(\mathsf{crs}, C, t, (w_1, \ldots, w_t)) \to \pi$: The prove algorithm takes as input the common reference string crs, a Boolean circuit $C \colon \{0,1\}^\lambda \times \{0,1\}^m \to \{0,1\}$, 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.
- $\mathsf{V}(\mathsf{crs}, C, t, \pi) \to \{0,1\}$: The verification algorithm takes as input the common reference string crs, a Boolean circuit $C \colon \{0,1\}^\lambda \times \{0,1\}^m \to \{0,1\}$, 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 bound on the number of instances T in *binary* and the challenge number of instances t in *unary*. Thus, 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.

[7] 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.

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 $K \leftarrow \mathsf{PRF.KeyGen}(1^\lambda)$. 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 $\perp$.

  Figure 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.

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

  The setup algorithm constructs the obfuscated programs $\mathsf{ObfProve} \leftarrow i\mathcal{O}(1^\lambda, \mathsf{Prove}[K])$ 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 prove algorithm first sets $\pi_0 \leftarrow \bot$. Then, for $i \in [t]$, the prove algorithm computes $\pi_i \leftarrow \mathsf{ObfProve}(C, i, w_i, \pi_{i-1})$. Finally, the algorithm outputs 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 $\mathsf{ObfVerify}(C, t, \pi)$.

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

Proof. Take any security parameter $\lambda \in \mathbb{N}$, any s, any Boolean circuit $C\colon \{0,1\}^\lambda \times \{0,1\}^m \to \{0,1\}$ of size at most s and any instance number $t \in [2^\lambda]$. Let $w_1, \ldots, w_t$ be a collection of witnesses such that $C(i, w_i) = 1$ for all $i \in [t]$. Suppose $\mathsf{crs} = (\mathsf{ObfProve}, \mathsf{ObfVerify}) \leftarrow \mathsf{Gen}(1^\lambda, 1^s)$ and $\pi \leftarrow \mathsf{Prove}(\mathsf{crs}, C, (w_1, \ldots, w_t))$.

- Consider the sequence of proofs $\pi_1, \ldots, \pi_t = \pi$ computed by the Prove algorithm. By correctness of $i\mathcal{O}$, $\pi_1 = \mathsf{Prove}[K](C, 1, w_1, \bot) = \mathsf{PRF.Eval}(K, (C, 1))$. Then, for $i > 1$, by correctness of $i\mathcal{O}$, we have $\pi_i = \mathsf{Prove}[K](C, i, w_i, \pi_{i-1}) = \mathsf{PRF.Eval}(K, (C, i))$. Thus, $\pi_t = \mathsf{PRF.Eval}(K, (C, t))$.
- Consider the output of $\mathsf{Verify}(\mathsf{crs}, C, t, \pi)$. By correctness of $i\mathcal{O}$, the output of Verify is the output of the program $\mathsf{Verify}[K](C, t, \pi)$, which is 1 by construction.

Thus, the verification algorithm accepts and correctness holds. □

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

Proof. We start by defining a sequence of hybrid experiments:

- Hyb_0: This is the non-adaptive soundness experiment:
  - Adversary $\mathcal{A}$, on input 1^λ, starts by outputting the maximum circuit size $1^{s(\lambda)}$, a Boolean circuit C^*_λ of size at most $s(\lambda)$, and the number of instances 1^{t_λ} where $t_\lambda \le 2^\lambda$. The challenger checks that there exists an index $i^*_\lambda \in [t_\lambda]$ such that $C^*_\lambda(i^*_\lambda, w) = 0$ for all $w \in \{0,1\}^*$. If such an i^* does not exist, the challenger aborts with output 0. For ease of notation, we simply write $C^* = C^*_\lambda$, $t = t_\lambda$, and $i^* = i^*_\lambda$ in the following description.
  - The challenger samples $\mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda, 1^s)$ and gives $\mathsf{crs} = (\mathsf{ObfProve}, \mathsf{ObfVerify})$ to $\mathcal{A}$.
  - Adversary $\mathcal{A}$ outputs a proof π.
  - The output of the experiment is $\mathsf{Verify}(\mathsf{crs}, C^*, t, \pi)$, which by definition, is $\mathsf{ObfVerify}(C^*, t, \pi)$.
- Hyb_j for $j \in \{i^*, \ldots, t\}$: Same as Hyb_0, except the challenger changes the distribution of the CRS. Specifically, it defines the modified programs $\mathsf{Prove}'[K, i^*, i_{\mathsf{thresh}}, C^*]$ and $\mathsf{Verify}'[K, i^*, i_{\mathsf{thresh}}, C^*]$ as follows:

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

1. If $C = C^*$ and $i^* \le i \le i_{\mathsf{thresh}}$, output $\bot$.
2. Else if $i = 1$ and $C(1, w_1) = 1$, output $\mathsf{PRF.Eval}(K, (C, 1))$.
3. Else if $\mathsf{PRG}(\pi) = \mathsf{PRG}(\mathsf{PRF.Eval}(K, (C, i-1)))$ and $C(i, w_i) = 1$, output $\mathsf{PRF.Eval}(K, (C, i))$.
4. Otherwise, output $\bot$.

Figure 3: Program $\mathsf{Prove}'[K, i^*, i_{\mathsf{thresh}}, C^*]$

Constants: PRF key K, starting index i^*, threshold index i_{thresh}, Boolean circuit C^*
Input: Boolean circuit C of size at most s, instance number $t \in [2^\lambda]$, proof $\pi \in \{0,1\}^\lambda$

1. If $C = C^*$ and $i^* \le t < i_{\mathsf{thresh}}$, output 0.
2. Else if $\mathsf{PRG}(\pi) = \mathsf{PRG}(\mathsf{PRF.Eval}(K, (C, t)))$, output 1
3. Otherwise, output 0

Figure 4: Program $\mathsf{Verify}'[K, i^*, i_{\mathsf{thresh}}, C^*]$

To construct the CRS, the challenger computes $\mathsf{ObfProve} \leftarrow i\mathcal{O}(1^\lambda, \mathsf{Prove}'[K, i^*, j, C^*])$ and $\mathsf{ObfVerify} \leftarrow i\mathcal{O}(1^\lambda, \mathsf{Verify}'[K, i^*, j, C^*])$, where Prove' and Verify' are the programs in Fig. 3 and Fig. 4. As in the real scheme, the challenger pads the size of Prove' and Verify' to the maximum size of the circuits that appear in the proof of Theorem 3.3.

For an adversary $\mathcal{A}$, we write $\mathsf{Hyb}_i(\mathcal{A})$ to denote the output distribution of $\mathsf{Hyb}_i(\mathcal{A})$ with adversary $\mathcal{A}$. In the following analysis, we model $\mathcal{A}$ as a deterministic *non-uniform* algorithm that takes as input the security parameter 1^λ (and advice string ρ_λ), and outputs the maximum circuit size $1^{s(\lambda)}$, a Boolean circuit C^*_λ of size at most $s(\lambda)$, and the number of instances 1^{t_λ} where $t_\lambda \le 2^\lambda$. If the advantage of $\mathcal{A}$ is non-zero in the non-adaptive soundness game, it must be the case that there exists an index $i^*_\lambda \in [t_\lambda]$ such that $C^*_\lambda(i^*_\lambda, w) = 0$ for all $w \in \{0,1\}^*$. If there are multiple such indices, we define i^*_λ to be the first such index. In the following, we will consider deterministic non-uniform reduction algorithms that are provided $(\rho_\lambda, i^*_\lambda)$ as advice.[8] We now show that each pair of adjacent distributions defined above are indistinguishable.

Lemma 3.4. *Suppose $i\mathcal{O}$ is secure. Then, for all non-uniform polynomial time adversaries $\mathcal{A}$, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for all* $\lambda \in \mathbb{N}$, $|\Pr[\mathsf{Hyb}_{i^*}(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_0(\mathcal{A}) = 1]| = \mathsf{negl}(\lambda)$.

Proof. We claim that the programs $\mathsf{Prove}[K]$ and $\mathsf{Prove}'[K, i^*, i^*, C^*]$ have identical behavior, and similarly for programs $\mathsf{Verify}[K]$ and $\mathsf{Verify}'[K, i^*, i^*, C^*]$:

- By construction, $\mathsf{Prove}[K]$ and $\mathsf{Prove}'[K, i^*, i^*, C^*]$ have *identical* functionality except perhaps on inputs of the form (C^*, i^*, w, π) for some $w \in \{0,1\}^*$ and $\pi \in \{0,1\}^\lambda$. On such inputs, program $\mathsf{Prove}'[K, i^*, i^*, C^*]$ always outputs $\perp$. Next, by assumption, $C^*(i^*, w) = 0$ for all $w \in \{0,1\}^*$, so $\mathsf{Prove}[K](C, i^*, w, \pi) = \perp$ for all $w \in \{0,1\}^*$ and $\pi \in \{0,1\}^\lambda$. Thus, we conclude that $\mathsf{Prove}[K]$ and $\mathsf{Prove}'[K, i^*, i^*, C^*]$ have identical input/output behavior.
- We claim that $\mathsf{Verify}[K]$ and $\mathsf{Verify}'[K, i^*, i^*, C^*]$ compute the same functionality. The only difference between these two programs is the extra check that Verify' performs. By construction, $\mathsf{Verify}'[K, i^*, i^*, C^*]$ only differs from $\mathsf{Verify}[K]$ if the circuit C satisfies $C = C^*$ and the instance number t satisfies $i^* \le t < i^*$. This latter condition is always false, so the two programs have identical input/output behavior.

Since $\mathsf{Prove}[K]$ and $\mathsf{Verify}[K]$ compute identical functions as $\mathsf{Prove}'[K, i^*, i^*, C^*]$ and $\mathsf{Verify}'[K, i^*, i^*, C^*]$ respectively, indistinguishability now follows by $i\mathcal{O}$ security and a standard hybrid argument. Note that constructing the programs $\mathsf{Prove}'[K, i^*, i^*, C^*]$ and $\mathsf{Verify}'[K, i^*, i^*, C^*]$ requires knowledge of the index i^*, which would be provided as part of the advice string in our *non-uniform* reduction. □

Lemma 3.5. *If* PRF *is functionality-preserving and satisfies punctured pseudorandomness,* PRG *is a secure PRG, and $i\mathcal{O}$ is secure, then for all $j \in \{i^*, \ldots, t-1\}$ and all non-uniform polynomial time adversaries $\mathcal{A}$, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for all* $\lambda \in \mathbb{N}$, $|\Pr[\mathsf{Hyb}_j(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_{j+1}(\mathcal{A}) = 1]| = \mathsf{negl}(\lambda)$.

Proof. We begin by introducing a sequence of intermediate hybrids:

[8] We rely on non-uniformity here because the index i^* may not be efficiently-computable from the challenge circuit C^*. For this reason, we rely on a *non-uniform* reduction where the reduction algorithm is given the index i^* as *advice* (we are guaranteed that such an index i^* always exists if algorithm $\mathcal{A}$ successfully breaks non-adaptive soundness with non-negligible advantage). Correspondingly, security relies on *non-uniform* hardness of the underlying cryptographic primitives. We discuss this further in Remark 3.10.

- $\mathsf{Hyb}_j^{(1)}$: Same as Hyb_j except the challenger changes the distribution of the CRS. Specifically, it defines the modified programs $\mathsf{Prove}''[K\{(C^*, i_{\mathsf{thresh}})\}, i^*, i_{\mathsf{thresh}}, C^*, z]$ and $\mathsf{Verify}''[K\{(C^*, i_{\mathsf{thresh}})\}, i^*, i_{\mathsf{thresh}}, C^*, z]$ as follows:

Constants: Punctured PRF key K_p, starting index i^*, threshold index i_{thresh}, Boolean circuit C^*, value z
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 $C = C^*$ and $i^* \le i \le i_{\mathsf{thresh}}$, output $\bot$.
2. Else if $i = 1$ and $C(1, w_1) = 1$, output $\mathsf{PRF.Eval}(K_p, (C, 1))$.
3. Else if $(C, i-1) = (C^*, i_{\mathsf{thresh}})$ and $\mathsf{PRG}(\pi) = z$ and $C(i, w_i) = 1$, output $\mathsf{PRF.Eval}(K_p, (C, i))$.
4. Else if $(C, i-1) \neq (C^*, i_{\mathsf{thresh}})$ and $\mathsf{PRG}(\pi) = \mathsf{PRG}(\mathsf{PRF.Eval}(K_p, (C, i-1)))$ and $C(i, w_i) = 1$, output $\mathsf{PRF.Eval}(K_p, (C, i))$.
5. Otherwise, output $\bot$

Figure 5: Program $\mathsf{Prove}''[K_p, i^*, i_{\mathsf{thresh}}, C^*, z]$

Constants: Punctured PRF key K_p, starting index i^*, threshold index i_{thresh}, Boolean circuit C^*, value z
Input: Boolean circuit C of size at most s, instance number $t \in [2^\lambda]$, proof $\pi \in \{0,1\}^\lambda$

1. If $C = C^*$ and $i^* \le t < i_{\mathsf{thresh}}$, output 0.
2. Else if $(C, t) = (C^*, i_{\mathsf{thresh}})$ and $\mathsf{PRG}(\pi) = z$, output 1.
3. Else if $(C, t) \neq (C^*, i_{\mathsf{thresh}})$ and $\mathsf{PRG}(\pi) = \mathsf{PRG}(\mathsf{PRF.Eval}(K_p, (C, t)))$, output 1.
4. Otherwise, output 0.

Figure 6: Program $\mathsf{Verify}''[K_p, i^*, i_{\mathsf{thresh}}, C^*, z]$

Next, the challenger computes the punctured key $K\{(C^*, j)\} \leftarrow \mathsf{PRF.Puncture}(K, (C^*, j))$ and the evaluation $z^* \leftarrow \mathsf{PRG}(\mathsf{PRF.Eval}(K, (C^*, j)))$. It then constructs $\mathsf{ObfProve} \leftarrow i\mathcal{O}(1^\lambda, \mathsf{Prove}''[K\{(C^*, j)\}, i^*, j, C^*, z^*])$ and $\mathsf{ObfVerify} \leftarrow i\mathcal{O}(1^\lambda, \mathsf{Verify}''[K\{(C^*, j)\}, i^*, j, C^*, z^*])$. As in the real scheme, the challenger pads the size of Prove'' and Verify'' to the maximum size of the circuits that appear in the proof of Theorem 3.3. The CRS is still $\mathsf{crs} = (\mathsf{ObfProve}, \mathsf{ObfVerify})$.

- $\mathsf{Hyb}_j^{(2)}$: Same as $\mathsf{Hyb}_j^{(1)}$ but when constructing the CRS, the challenger sets $z^* \leftarrow \mathsf{PRG}(y^*)$ where $y^* \xleftarrow{R} \{0,1\}^\lambda$.
- $\mathsf{Hyb}_j^{(3)}$: Same as $\mathsf{Hyb}_j^{(2)}$ but when constructing the CRS, the challenger samples $z^* \xleftarrow{R} \{0,1\}^{2\lambda}$.

We now argue that each pair of adjacent hybrid experiments are indistinguishable for all $j \in \{i^*, \ldots, t\}$.

Claim 3.6. *Suppose* PRF *is functionality-preserving and $i\mathcal{O}$ is secure. Then, for all non-uniform polynomial time adversaries $\mathcal{A}$, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for all $\lambda \in \mathbb{N}$,*

$$|\Pr[\mathsf{Hyb}_j^{(1)}(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_j(\mathcal{A}) = 1]| = \mathsf{negl}(\lambda).$$

Proof. Similar to the proof of Lemma 3.4, it suffices to show that the prover and verifier programs in Hyb_j and $\mathsf{Hyb}_j^{(1)}$ have identical input/output behavior. First, since the PRF is functionality-preserving property, for all inputs $(C, i) \neq (C^*, j)$, we have that $\mathsf{PRF.Eval}(K, (C, i)) = \mathsf{PRF.Eval}(K\{(C^*, j)\}, (C, i))$. We first argue that $\mathsf{Prove}'[K, i^*, j, C^*]$ and $\mathsf{Prove}''[K\{(C^*, j)\}, i^*, j, C^*, z]$ have identical input/output behavior:

- Suppose $j > 1$. Since the PRF satisfies functionality-preserving, $\mathsf{PRF.Eval}(K, (C, i)) = \mathsf{PRF.Eval}(K\{(C^*, j)\}, (C, i))$ whenever $(C, i) \neq (C^*, j)$. Next, in $\mathsf{Hyb}_j^{(1)}$, the challenger sets $z \leftarrow \mathsf{PRG}(\mathsf{PRF.Eval}(K, (C^*, j)))$. As such, the checks in program Prove'' are identical to those in Prove'. Thus, the two programs have the same input/output behavior.
- Suppose $j = 1$. Recall that $j \ge i^* \ge 1$. In this case, on input $(C^*, 1)$, the first check in Prove' and Prove'' ensures that the output is $\bot$. This matches the behavior of Prove' in Hyb_j. On all other inputs (C, i) with $i \neq 1$, the behavior is identical by functionality-preserving of the underlying punctured PRF.

Now consider $\mathsf{Verify}'[K, i^*, j, C^*]$ and $\mathsf{Verify}''[K\{(C^*, j)\}, i^*, j, C^*, z^*]$. Once again, since the challenger sets $z = \mathsf{PRG}(\mathsf{PRF.Eval}(K, (C^*, j)))$, the verification checks in the two programs are identical. The claim now follows from $i\mathcal{O}$ security and a standard hybrid argument. Similar to the proof of Lemma 3.4, the formal reduction is non-uniform (with advice string i^*) and thus, security relies on non-uniform hardness of $i\mathcal{O}$. □

Claim 3.7. *If* PRF *satisfies punctured pseudorandomness, then for all non-uniform polynomial-time adversaries $\mathcal{A}$, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for all* $\lambda \in \mathbb{N}$, $|\Pr[\mathsf{Hyb}_j^{(2)}(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_j^{(1)}(\mathcal{A}) = 1]| = \mathsf{negl}(\lambda)$.

Proof. Suppose there exists a non-uniform polynomial time adversary $\mathcal{A}$ (with advice string ρ_λ) such that

$$|\Pr[\mathsf{Hyb}_j^{(2)}(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_j^{(1)}(\mathcal{A}) = 1]| \geq \varepsilon.$$

We use $\mathcal{A}$ to construct a non-uniform adversary $\mathcal{B}$ with advice string $(\rho, i^*) = (\rho_\lambda, i^*_\lambda)$ that breaks puncturing security of PRF. Recall that $i^* = i^*_\lambda$ is the index of the (first) false instance output by $\mathcal{A}$ (on input 1^λ and with advice ρ_λ).

1. Algorithm $\mathcal{B}$ runs adversary $\mathcal{A}$ on input 1^λ and with advice string ρ. Algorithm $\mathcal{A}$ outputs the maximum circuit size 1^s, a Boolean circuit C^* of size at most s, and the number of instances 1^t where $t \leq 2^\lambda$.
2. Algorithm $\mathcal{B}$ chooses (C^*, j) as it challenge point. It receives from the challenger a punctured key $K\{(C^*, j)\}$ and a challenge $y \in \{0,1\}^\lambda$.
3. Algorithm $\mathcal{B}$ computes $z^* \leftarrow \mathsf{PRG}(t)$, $\mathsf{ObfProve} \leftarrow i\mathcal{O}(1^\lambda, \mathsf{Prove}''[K\{(C^*, j)\}, i^*, j, C^*, z^*]$, and $\mathsf{ObfVerify} \leftarrow i\mathcal{O}(1^\lambda, \mathsf{Verify}''[K\{(C^*, j)\}, i^*, j, C^*, z^*)$. Finally, it sets the $\mathsf{crs} = (\mathsf{ObfProve}, \mathsf{ObfVerify})$ and gives crs to $\mathcal{A}$.
4. At the end of the game, algorithm $\mathcal{A}$ outputs a proof π and algorithm $\mathcal{B}$ outputs $\mathsf{ObfVerify}(C^*, t, \pi)$.

By construction, the challenger samples $K \leftarrow \mathsf{PRF.KeyGen}(1^\lambda)$ and constructs the punctured key as $K\{(C^*, j)\} \leftarrow \mathsf{PRF.Puncture}(K, (C^*, j))$. This coincides with the specification in $\mathsf{Hyb}_j^{(1)}$ and $\mathsf{Hyb}_j^{(2)}$. Consider now the distribution of the challenge t:

- Suppose $y = \mathsf{PRF.Eval}(K, (C^*, j))$. Then algorithm $\mathcal{A}$ perfectly simulates distribution $\mathsf{Hyb}_j^{(1)}$.
- Suppose $y \xleftarrow{R} \{0,1\}^\lambda$. Then algorithm $\mathcal{A}$ perfectly simulates distribution $\mathsf{Hyb}_j^{(2)}$.

Algorithm $\mathcal{B}$ breaks puncturing security of the PRF with advantage ε and the claim follows. □

Claim 3.8. *If* PRG *is secure, then for all non-uniform polynomial time adversaries $\mathcal{A}$, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for all* $\lambda \in \mathbb{N}$, $|\Pr[\mathsf{Hyb}_j^{(3)}(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_j^{(2)}(\mathcal{A}) = 1]| = \mathsf{negl}(\lambda)$.

Proof. Suppose there exists a non-uniform polynomial time adversary $\mathcal{A}$ (with advice string ρ_λ) where

$$|\Pr[\mathsf{Hyb}_j^{(3)}(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_j^{(2)}(\mathcal{A}) = 1]| \geq \varepsilon$$

for some non-negligible ε. We use $\mathcal{A}$ to construct an adversary $\mathcal{B}$ with advice string $(\rho, i^*) = (\rho_\lambda, i^*_\lambda)$ that breaks PRG security:

1. Algorithm $\mathcal{B}$ runs adversary $\mathcal{A}$ on input 1^λ and advice string ρ_λ. Algorithm $\mathcal{A}$ outputs the maximum circuit size 1^s, a Boolean circuit C^* of size at most s, and the number of instances 1^t where $t \leq 2^\lambda$.
2. Algorithm $\mathcal{B}$ receives a challenge $z^* \in \{0,1\}^{2\lambda}$ from the PRG challenger.
3. Algorithm $\mathcal{B}$ samples $K \leftarrow \mathsf{PRF.KeyGen}(1^\lambda)$ and computes the punctured key

   $$K\{(C^*, j)\} \leftarrow \mathsf{PRF.Puncture}(K, (C^*, j)).$$

   Next, it computes the obfuscated programs $\mathsf{ObfProve} \leftarrow i\mathcal{O}(1^\lambda, \mathsf{Prove}''[K\{(C^*, j)\}, i^*, j, C^*, z^*])$ and $\mathsf{ObfVerify} \leftarrow i\mathcal{O}(1^\lambda, \mathsf{Verify}''[K\{(C^*, j)\}, i^*, j, C^*, z^*])$. Algorithm $\mathcal{B}$ gives $\mathsf{crs} = (\mathsf{ObfProve}, \mathsf{ObfVerify})$ to $\mathcal{A}$.

4. Algorithm $\mathcal{A}$ outputs a proof π and algorithm $\mathcal{B}$ outputs $\mathsf{ObfVerify}(C^*, t, \pi)$.

If $z^* \leftarrow \mathsf{PRG}(y^*)$ where $y^* \xleftarrow{R} \{0,1\}^\lambda$, then algorithm $\mathcal{B}$ perfectly simulates $\mathsf{Hyb}_j^{(2)}$ for $\mathcal{A}$. Alternatively, if $z^* \xleftarrow{R} \{0,1\}^{2\lambda}$, then algorithm $\mathcal{B}$ perfectly simulates $\mathsf{Hyb}_j^{(3)}$ for $\mathcal{A}$. The claim follows. □

Claim 3.9. *If* PRF *is functionality-preserving and $i\mathcal{O}$ is secure, then for all non-uniform polynomial time adversaries $\mathcal{A}$, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for all $\lambda \in \mathbb{N}$, $|\Pr[\mathsf{Hyb}_{j+1}(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_j^{(3)}(\mathcal{A}) = 1]| = \mathsf{negl}(\lambda)$.*

Proof. We start by showing that with overwhelming probability over the choice of $z^* \xleftarrow{R} \{0,1\}^{2\lambda}$, the programs $\mathsf{Prove}''[K\{(C^*, j)\}, i^*, j, C^*, z^*]$ and $\mathsf{Verify}''[K\{(C^*, j)\}, i^*, j, C^*, z^*]$ in $\mathsf{Hyb}_j^{(3)}$ compute identical functionality as programs $\mathsf{Prove}'[K, i^*, j+1, C^*]$ and $\mathsf{Verify}'[K, i^*, j+1, C^*]$ in Hyb_{j+1}. Since $z^* \xleftarrow{R} \{0,1\}^{2\lambda}$,

$$\Pr[\exists y \in \{0,1\}^\lambda : \mathsf{PRG}(y) = z^*] \leq 2^{-\lambda}.$$

Thus, with overwhelming probability, the value z^* in $\mathsf{Hyb}_j^{(3)}$ is not in the range of PRG. Next, since PRF is functionality-preserving, $\mathsf{PRF.Eval}(K, (C, i)) = \mathsf{PRF.Eval}(K\{(C^*, j)\}, (C, i))$ whenever $(C, i) \neq (C^*, j)$. Now, consider the programs $\mathsf{Prove}''[K\{(C^*, j)\}, i^*, j, C^*, z^*]$ and $\mathsf{Prove}'[K, i^*, j+1, C^*]$:

- Suppose $j > 1$. By construction of Prove' and Prove'', this means that the *only* inputs on which the programs can differ are inputs of the form $(C^*, j+1, w, \pi)$ for some choice of $w \in \{0,1\}^*$ and $\pi \in \{0,1\}^\lambda$. Consider the behavior of the two programs on inputs of this form:
  - $\mathsf{Prove}''[K\{(C^*, j)\}, i^*, j, C^*, z^*](C^*, j+1, w, \pi)$ outputs $\perp$ if z^* is not in the image of PRG (in which case $\mathsf{PRG}(\pi) \neq z^*$).
  - $\mathsf{Prove}'[K, i^*, j+1, C^*](C^*, j+1, w, \pi)$ always outputs $\perp$ since $i^* \leq j+1 \leq j+1$.

  We conclude that on all inputs, the output of Prove' and Prove'' is identical with overwhelming probability over the choice of z^*.
- Suppose $j = 1$. In this case, the two programs' logic also differ on inputs of the form $(C^*, 1, w, \pi)$ for some $w \in \{0,1\}^*$ and $\pi \in \{0,1\}^\lambda$ (since the punctured key is used to evaluate at $(C^*, 1)$ in Prove'' while the real key is used in Prove'). However, since $j \geq i^* \geq 1$, both programs output $\perp$ on input $(C^*, 1, w, \pi)$. The behavior on all other inputs is identical by the analysis from the previous case.

Consider now the verification programs $\mathsf{Verify}'[K, i^*, j+1, C^*]$ and $\mathsf{Verify}''[K\{(C^*, j)\}, i^*, j, C^*, z^*]$. Similar to the case with the proving circuits, the *only* inputs on which the programs can differ are inputs of the form $(C^*, j+1, w, \pi)$ for some choice of $w \in \{0,1\}^*$ and $\pi \in \{0,1\}^\lambda$.

- $\mathsf{Verify}''[K\{(C^*, j)\}, i^*, j, C^*, z^*]$ outputs 0 if z^* is not in the image of PRG (in which case $\mathsf{PRG}(\pi) \neq z^*$).
- $\mathsf{Verify}'[K, i^*, j+1, C^*](C^*, j+1, w, \pi)$ always outputs 0.

From the above analysis, we see that as long as z^* is not in the image of PRG, Prove' and Prove'' as well as Verify' and Verify'' in the two experiments have identical input/output behavior. Since this event happens with overwhelming probability, the claim now follows by $i\mathcal{O}$ security. □

Combining Claims 3.6 to 3.8, we have that for all $j \in \{i^*, \ldots, t-1\}$, hybrids Hyb_j and Hyb_{j+1} are computationally indistinguishable and Lemma 3.5 follows. □

Combining Lemmas 3.4 and 3.5, we have that hybrids Hyb_0 and Hyb_t are computationally indistinguishable. It is easy to show that for all adversaries $\mathcal{A}$ in Hyb_t, $\Pr[\mathsf{Hyb}_t(\mathcal{A}) = 1] = 0$. This follows by construction: namely, in Hyb_t, ObfVerify is an obfuscation of the verification program $\mathsf{Verify}'[K, i^*, t, C^*]$ which outputs 0 on all inputs of the form (C^*, t, π) for any $\pi \in \{0,1\}^\lambda$. Correspondingly, for all efficient adversaries $\mathcal{A}$, $\Pr[\mathsf{Hyb}_0(\mathcal{A}) = 1] = \mathsf{negl}(\lambda)$ and non-adaptive soundness holds. □

Remark 3.10 (Non-Uniform Hardness)**.** The proof of non-adaptive soundness (Theorem 3.3) leverages *non-uniform* security reductions where the reduction algorithms are also given the particular index i^* of the false instance as non-uniform advice (i.e., the instance i^* where $C^*(i^*, w) = 0$ for all $w \in \{0,1\}^*$). We rely on non-uniform advice since in general, computing i^* from the adversary's chosen circuit C^* may not be efficient (note that such an index is guaranteed to exist given an adversary that breaks non-adaptive soundness with non-negligible probability). Because our security reductions are non-uniform, we correspondingly rely on non-uniform hardness of each of the underlying primitives. Note that we could alternatively rely on sub-exponential hardness (and compute i^* from C^* in a brute-force way) to obtain a uniform security reduction, but this would jeopardize the full succinctness of our construction. We also note that for settings where the index i^*_λ can be efficiently computed from C^*_λ by a *uniform* family of circuits, then our security reductions would also be uniform (and correspondingly, we can base security on hardness of the underlying primitives against uniform adversaries).

Theorem 3.11 (Succinctness)**.** *Construction 3.1 is fully succinct.*

Proof. We show that Construction 3.1 satisfies the two succinctness requirements from Definition 2.9:

- **Succinct proof size:** The size of the proof is the output of PRF which has length λ. Thus, the proof is fully succinct.
- **Succinct verification time:** Verification consists of evaluating the ObfVerify program on input (C, t, π). By construction, ObfVerify is an obfuscation of the verification algorithm Verify$[K]$. Again by construction, the running time of Verify$[K]$ is $\mathsf{poly}(\lambda, s)$. Since the obfuscator is efficient, the running time of ObfVerify is also $\mathsf{poly}(\lambda, s)$, as required. □

Theorem 3.12 (Zero Knowledge)**.** *Construction 3.1 satisfies perfect zero knowledge.*

Proof. To show zero knowledge, we construct an efficient simulator $\mathcal{S}$ as follows. On input the security parameter λ, the bound s on the circuit size, a Boolean circuit $C\colon \{0,1\}^\ell \times \{0,1\}^m \to \{0,1\}$ of size at most s, and the instance number t, the simulator algorithm proceeds as follows:

1. Compute $\mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda, 1^s)$. Let K be the PRF key sampled in the construction of crs and compute the simulated proof $\pi \leftarrow \mathsf{PRF.Eval}(K, (C, t))$.
2. Output (crs, π).

By construction, the simulator samples crs exactly as in the real scheme. It suffices to consider the proofs. By construction and correctness of $i\mathcal{O}$, a proof on $(C, (1, \ldots, t))$ is always $\pi = \mathsf{PRF.Eval}(K, (C, t))$. This is the simulated proof. □

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 an 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 and a tuple of inputs $x_1, \ldots, x_t \in \{0,1\}^\ell$, 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 $x_1, \ldots, x_t \in \{0,1\}^\ell$ 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 $b \in \{0,1\}$.

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

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

- **Succinctness:** There exists a polynomial $\mathsf{poly}(\cdot,\cdot)$ such that the length of the hash value y output by Hash and the length of the proof π output by Open in the correctness 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 collection of 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 (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).[9]

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:

  > **Constants:** Public parameters pp for Π_{PA}, a hash value h for Π_{PA}, 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.

  Figure 7: 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 $\mathsf{crs} = (\mathsf{pp}, \mathsf{IndexBARG.crs})$.
- $\mathsf{P}(\mathsf{crs}, C, (x_1, \ldots, x_t), (w_1, \ldots, w_t))$: 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. 7.

[9] 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.

- $\mathsf{V}(\mathsf{crs}, C, (x_1, \ldots, x_t), \pi)$: 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. 7.

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

Proof. Take any security parameter $\lambda \in \mathbb{N}$, circuit size bound $s \in \mathbb{N}$, input length $\ell \in \mathbb{N}$, any Boolean circuit $C\colon \{0,1\}^\ell \times \{0,1\}^m \to \{0,1\}$ of size at most s, and any instance number $t \in [2^\lambda]$. Let $x_1, \ldots, x_t \in \{0,1\}^\ell$ be a collection of statements and $w_1, \ldots, w_t$ be a collection of corresponding witnesses such that $C(x_i, w_i) = 1$ for all $i \in [t]$. Suppose $\mathsf{crs} = (\mathsf{pp}, \mathsf{IndexBARG.crs}) \leftarrow \mathsf{Gen}(1^\lambda, 1^\ell, 1^s)$ and $\pi \leftarrow \mathsf{Prove}(\mathsf{crs}, C, (x_1, \ldots, x_t), (w_1, \ldots, w_t))$. Let $\sigma_i \leftarrow \mathsf{PA.Open}(\mathsf{pp}, (x_1, \ldots, x_t), i)$ be the openings computed by the prove algorithm. Consider $\mathsf{V}(\mathsf{crs}, C, (x_1, \ldots, x_t), \pi)$. Since Π_{PA} is correct, for every $i \in [t]$, $\mathsf{PA.Verify}(\mathsf{pp}, h, x_i, i, \sigma_i) = 1$. Thus, for every $i \in [t]$, $C'(i, (x_i, \sigma_i, w_i)) = 1$, where $C' = C'[\mathsf{pp}, h, C]$ is the circuit from Fig. 7. Completeness now follows from completeness of the underlying BARG for index languages. □

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.*

Proof. We start by defining a sequence of hybrid experiments:

- Hyb_0 : This is the non-adaptive soundness experiment:
  - Adversary $\mathcal{A}$ starts by outputting the maximum circuit size $1^{s(\lambda)}$, a Boolean circuit C^*_λ of size at most $s(\lambda)$, and statements $x^*_1, \ldots, x^*_{t_\lambda}$ where $t_\lambda \le 2^\lambda$. The challenger checks that there exists an index $i^*_\lambda \in [t_\lambda]$ such that $C^*_\lambda(x^*_{i^*_\lambda}, w) = 0$ for all $w \in \{0,1\}^*$. If such an i^* does not exist, the challenger aborts with output 0. For ease of notation, we simply write $C^* = C^*_\lambda$, $t = t_\lambda$, and $i^* = i^*_\lambda$ in the following description.
  - The challenger samples $\mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda, 1^\ell, 1^s)$ and gives it to $\mathcal{A}$. Here, $\mathsf{crs} = (\mathsf{pp}, \mathsf{IndexBARG.crs})$ where $\mathsf{pp} \leftarrow \mathsf{PA.Setup}(1^\lambda, 1^\ell)$ and $\mathsf{IndexBARG.crs} \leftarrow \mathsf{IndexBARG.Gen}(1^\lambda, 1^{s'})$.
  - Adversary $\mathcal{A}$ outputs a proof π.
  - The output of the experiment is 1 if $\mathsf{V}(\mathsf{crs}, C^*, (x^*_1, \ldots, x^*_t), \pi)$ and 0 otherwise.
- Hyb_1: Same as the previous experiment, but the challenger samples the public parameters pp using SetupEnforce: $\mathsf{pp} \leftarrow \mathsf{PA.SetupEnforce}(1^\lambda, 1^\ell, (x^*_1, \ldots, x^*_t), i^*)$.

For an adversary $\mathcal{A}$, we write $\mathsf{Hyb}_i(\mathcal{A})$ to denote the output distribution of $\mathsf{Hyb}_i(\mathcal{A})$ with adversary $\mathcal{A}$. We now show that each pair of adjacent distributions defined above are indistinguishable. As in the proof of Theorem 3.3, we model the adversary $\mathcal{A}$ as a deterministic *non-uniform* algorithm that takes as input the security parameter 1^λ (and advice string ρ_λ) and outputs the maximum circuit size $1^{s(\lambda)}$, a Boolean circuit C^*_λ of size at most $s(\lambda)$, and statements $x^*_1, x^*_2, \ldots, x^*_{t_\lambda}$ where $t_\lambda \le 2^\lambda$. If the advantage of $\mathcal{A}$ is non-zero in the non-adaptive soundness game, it must be the case that there exists an index $i^*_\lambda \in [t_\lambda]$ such that $C^*_\lambda(x^*_{i^*_\lambda}, w) = 0$ for all $w \in \{0,1\}^*$. If there are multiple such indices, we define i^*_λ to be the first such index. In the following, we will consider deterministic non-uniform reduction algorithms that are provided $(\rho_\lambda, i^*_\lambda)$ as advice (similar to the proof of Theorem 3.3, we rely on non-uniformity because the index $i^* = i^*_\lambda$ may not be efficiently-computable; see also Remark 3.10). We now show that each pair of adjacent distributions defined above are indistinguishable.

Lemma 4.7. *Suppose Π_{PA} satisfies setup indistinguishability. Then for every non-uniform polynomial time adversary $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that for all $\lambda \in \mathbb{N}$, $|\Pr[\mathsf{Hyb}_1(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_0(\mathcal{A}) = 1]| = \mathsf{negl}(\lambda)$.*

Proof. Let $\mathcal{A}$ be a (deterministic) non-uniform polynomial time adversary where

$$|\Pr[\mathsf{Hyb}_1(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_0(\mathcal{A}) = 1]| \ge \varepsilon$$

for some non-negligible ε. We construct a non-uniform adversary $\mathcal{B}$ with advice string $(\rho, i^*) = (\rho_\lambda, i^*_\lambda)$ that breaks setup indistinguishability of the positional accumulator.

1. Algorithm $\mathcal{B}$ runs adversary $\mathcal{A}$ on input 1^λ and advice string ρ. Algorithm $\mathcal{A}$ outputs the maximum circuit size 1^s, a Boolean circuit C^* of size at most s, and statements $x_1, \ldots, x_t$ where $t \leq 2^\lambda$.

2. Algorithm $\mathcal{B}$ gives $(x_1^*, \ldots, x_t^*)$ along with the index i^* to the challenger.

3. Algorithm $\mathcal{B}$ receives a set of public parameters pp from the challenger. If $b = 0$, these are sampled as $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, 1^\ell)$ and if $b = 1$, they are sampled as $\mathsf{pp} \leftarrow \mathsf{SetupEnforce}(1^\lambda, 1^\ell, (x_1^*, \ldots, x_t^*), i^*)$.

4. Algorithm $\mathcal{B}$ computes $\mathsf{IndexBARG.crs} \leftarrow \mathsf{IndexBARG.Gen}(1^\lambda, 1^{s'})$ and gives $\mathsf{crs} = (\mathsf{pp}, \mathsf{IndexBARG.crs})$ to $\mathcal{A}$. Algorithm $\mathcal{A}$ then outputs a proof π.

5. Algorithm $\mathcal{B}$ outputs $\mathsf{V}(\mathsf{crs}, C^*, (x_1^*, \ldots, x_t^*), \pi)$.

Observe that if $b = 0$, algorithm $\mathcal{B}$ perfectly simulates distribution Hyb_0 and if $b = 1$, algorithm $\mathcal{B}$ perfectly simulates distribution Hyb_1. Thus, $\mathcal{B}$'s advantage in breaking setup indistinguishability is ε, which is non-negligible. □

Lemma 4.8. *Suppose* $\Pi_{\mathsf{IndexBARG}}$ *satisfies non-adaptive soundness and* Π_{PA} *is enforcing. Then for every non-uniform polynomial time adversary* $\mathcal{A}$*, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for all* $\lambda \in \mathbb{N}$, $\Pr[\mathsf{Hyb}_1(\mathcal{A}) = 1] = \mathsf{negl}(\lambda)$.

Proof. Suppose there exists a (deterministic) non-uniform polynomial time adversary $\mathcal{A}$ where $\Pr[\mathsf{Hyb}_1(\mathcal{A}) = 1] \geq \varepsilon$ and ε is non-negligible. We construct a non-uniform adversary $\mathcal{B}$ with advice string $(\rho, i^*) = (\rho_\lambda, i^*_\lambda)$ that breaks non-adaptive soundness of $\Pi_{\mathsf{IndexBARG}}$:

1. Algorithm $\mathcal{B}$ runs adversary $\mathcal{A}$ on input 1^λ and advice ρ. Algorithm $\mathcal{A}$ outputs the maximum circuit size 1^s, a Boolean circuit C^* of size at most s, and statements $x_1^*, \ldots, x_t^*$ where $t \leq 2^\lambda$.

2. Algorithm $\mathcal{B}$ computes $\mathsf{pp} \leftarrow \mathsf{PA.SetupEnforce}(1^\lambda, 1^\ell, x_1^*, \ldots, x_t^*, i^*)$ and $h^* \leftarrow \mathsf{PA.Hash}(\mathsf{pp}, x_1^*, \ldots, x_t^*)$. It also computes the circuit $C'[\mathsf{pp}, h^*, C^*](\cdot)$ according to Fig. 7.

3. Algorithm $\mathcal{B}$ outputs the maximum circuit size $1^{s'}$, where s' is the bound on the size of the circuit from Fig. 7, the Boolean circuit C' of size at most s', and the number of instances 1^t where $t \leq 2^\lambda$.

4. Algorithm $\mathcal{B}$ receives a common reference string $\mathsf{IndexBARG.crs} \leftarrow \mathsf{Gen}(1^\lambda, 1^{s'})$. Using IndexBARG.crs, algorithm $\mathcal{B}$ sets $\mathsf{crs} \leftarrow (\mathsf{pp}, \mathsf{IndexBARG.crs})$ and gives crs to $\mathcal{A}$.

5. Algorithm $\mathcal{A}$ outputs a proof π, which algorithm $\mathcal{B}$ also outputs.

By construction, algorithm $\mathcal{B}$ perfectly simulates an execution of Hyb_1 for $\mathcal{A}$. Thus, with probability at least ε, algorithm $\mathcal{A}$ outputs a proof π such that $\mathsf{V}(\mathsf{crs}, C^*, (x_1^*, \ldots, x_t^*), \pi) = 1$. This means that

$$\mathsf{IndexBARG.V}(\mathsf{IndexBARG.crs}, C', t, \pi) = 1, \tag{4.1}$$

where $C'[\mathsf{pp}, h^*, C^*](\cdot)$ is the circuit for the index relation according to Fig. 7 and $h^* \leftarrow \mathsf{PA.Hash}(\mathsf{pp}, x_1^*, \ldots, x_t^*)$. We now argue that $C'(i^*, w'_{i^*}) = 0$ for all w'_{i^*}. First write $w'_{i^*} = (x', \sigma', w')$. We consider two possibilities:

- Suppose $x' = x^*_{i^*}$. By definition of i^*, $x^*_{i^*}$ is a false instance so $C^*\left(x^*_{i^*}, w'\right) = 0$ irrespective of the value of w'. Thus, $C'(i^*, w'_{i^*}) = 0$.

- Suppose $x' \neq x^*_{i^*}$. Since the parameters (pp, h) are sampled in enforcing mode to bind on statement $x^*_{i^*}$ at index i^*, with all but negligible probability over the choice of pp, the *only* value of x' for which there exists σ' such that $\mathsf{PA.Verify}(\mathsf{pp}, h, x', i^*, \sigma') = 1$ is $x' = x^*_{i^*}$. Thus, with overwhelming probability over the choice of pp, $\mathsf{PA.Verify}(\mathsf{pp}, h, x', i^*, \sigma') = 0$ in this case. This again means $C'(i^*, w'_{i^*}) = 0$.

In both cases, we see that $C'(i^*, w'_{i^*}) = 0$. This holds for all $w'_{i^*} \in \{0,1\}^*$, so $(C', t) \notin \mathcal{L}_{\mathsf{BatchCSATindex},t}$, and yet Eq. (4.1) holds. Thus, algorithm $\mathcal{B}$ breaks non-adaptive soundness of the underlying index BARG with advantage $\geq \varepsilon - \mathsf{negl}(\lambda)$. □

Combining Lemmas 4.7 and 4.8, we conclude that for all polynomial-time (non-uniform) adversaries $\mathcal{A}$, there exists a negligible function such that $\Pr[\mathsf{Hyb}_0(\mathcal{A}) = 1] = \mathsf{negl}(\lambda)$. □

Theorem 4.9 (Succinctness)**.** *If $\Pi_{\mathsf{IndexBARG}}$ is succinct (resp., fully succinct) and Π_{PA} is efficient, then Construction 4.4 is succinct (resp., fully succinct).*

Proof. We show the two necessary succinctness properties.

- **Succinct proof size:** The proof π on t instances output by P in Construction 4.4 consists of a proof for $\Pi_{\mathsf{IndexBARG}}$ on the circuit $C' = C'[\mathsf{pp}, h, C]$ and the same number of instances t. By construction, $\mathsf{pp} \leftarrow \mathsf{PA.Setup}(1^\lambda, 1^\ell)$ and $h \leftarrow \mathsf{PA.Hash}(\mathsf{pp}, (x_1, \ldots, x_t))$. The efficiency and succinctness requirements of Π_{PA} imply that $|\mathsf{pp}| = \mathsf{poly}(\lambda, \ell)$ and $|h| = \mathsf{poly}(\lambda, \ell)$. Correspondingly, this means that $|C'| = \mathsf{poly}(\lambda, \ell, s) = \mathsf{poly}(\lambda, s)$ since $s \geq \ell$. Succinctness (resp., full succinctness) now follows from succicntness (resp., full succinctness) of $\Pi_{\mathsf{IndexBARG}}$.
- **Succinct verification time**: By construction, the verification algorithm needs to compute PA.Hash followed by IndexBARG.Verify. Since PA.Hash runs in polynomial time, computing $h \leftarrow \mathsf{PA.Hash}(\mathsf{pp}, (x_1, \ldots, x_t))$ takes $\mathsf{poly}(\lambda, t, \ell)$ time. By the same analysis as above, constructing the circuit $C'[\mathsf{pp}, h, C]$ can takes $\mathsf{poly}(\lambda, s)$ time. Finally, succinctness of $\Pi_{\mathsf{IndexBARG}}$ requires $\mathsf{poly}(\lambda, |C'|, |\pi|)$ time, so the overall verification time is bounded by $\mathsf{poly}(\lambda, t, \ell) + \mathsf{poly}(\lambda, s)$ time, as required. □

Theorem 4.10 (Zero Knowledge)**.** *If $\Pi_{\mathsf{IndexBARG}}$ is perfect zero-knowledge, then Construction 4.4 is perfect zero-knowledge.*

Proof. Let $\mathsf{IndexBARG}.\mathcal{S}$ be the simulator for $\Pi_{\mathsf{IndexBARG}}$. We construct a simulator for Π_{BARG} as follows. On input the security parameter λ, a bound ℓ on the instance size, a bound s on the circuit size, a Boolean circuit $C\colon \{0,1\}^\ell \times \{0,1\}^m \to \{0,1\}$, and instances $x_1, \ldots, x_t \in \{0,1\}^\ell$, the simulator proceeds as follows:

1. Sample $\mathsf{pp} \leftarrow \mathsf{PA.Setup}(1^\lambda, 1^\ell)$ and compute $h \leftarrow \mathsf{PA.Hash}(\mathsf{pp}, (x_1, \ldots, x_t))$.
2. Let $C' = C'[\mathsf{pp}, h, s]$ be the circuit from Fig. 7 and let s' be a bound on the size of C', and compute the simulated CRS and proof $(\mathsf{IndexBARG.crs}, \pi) \leftarrow \mathsf{IndexBARG}.\mathcal{S}(1^\lambda, 1^{s'}, C', t)$.
3. Output the simulated CRS $\mathsf{crs} = (\mathsf{pp}, \mathsf{IndexBARG.crs})$ and the simulated proof π.

By construction, the positional accumulator parameters pp and the circuit $C' = C'[\mathsf{pp}, h, s]$ are constructed exactly as in the real scheme. Perfect zero-knowledge now follows from perfect zero-knowledge of $\Pi_{\mathsf{IndexBARG}}$. □

Remark 4.11 (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 UpdateP algorithm. The UpdateP algorithm takes in a hash h_t (representing a short representation for some 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 BARG)**.** An updatable batch argument (BARG) for the language of Boolean circuit satisfiability consists of a tuple of efficient algorithms $\Pi_{\mathsf{BARG}} = (\mathsf{Gen}, \mathsf{Hash}, \mathsf{UpdateP}, \mathsf{V})$ 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.

- $\mathsf{Hash}(\mathsf{crs}, (x_1, \ldots, x_t)) \to h_t$: On input the common reference string crs, a sequence of statements $x_1, \ldots, x_t \in \{0,1\}^\ell$, the hash algorithm outputs a hash h_t.
- $\mathsf{UpdateP}(\mathsf{crs}, C, h_t, \pi_t, x_{t+1}, w_{t+1}) \to (h_{t+1}, \pi_{t+1})$: On input the common reference string crs, a Boolean circuit $C\colon \{0,1\}^\ell \times \{0,1\}^m \to \{0,1\}$, a hash h_t, a proof π_t, a new statement $x_{t+1} \in \{0,1\}^\ell$, and a witness $w_{t+1} \in \{0,1\}^m$, the update proof algorithm outputs an updated hash h_{t+1} and an updated proof π_{t+1}. Note that h_t, π_t are allowed to be empty. We write $\perp$ to denote an empty hash (representing an empty list of statements) and an empty proof.
- $\mathsf{V}(\mathsf{crs}, C, (x_1, \ldots, x_t), \pi) \to b$: 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 $\lambda \in \mathbb{N}$, any $t \le 2^\lambda$, bounds $\ell \in \mathbb{N}$ 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 $\forall i \in [t], C(x_i, w_i) = 1$, we have that,

$$\Pr\left[\forall i \in [t], \mathsf{V}(\mathsf{crs}, C, (x_1, \ldots, x_i), \pi_i) = 1 : \begin{array}{c} \mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda, 1^\ell, 1^s) \\ h_0 \leftarrow \perp, \pi_0 \leftarrow \perp, \\ \forall i \in [t], (h_i, \pi_i) \leftarrow \mathsf{UpdateP}(\mathsf{crs}, C, h_{i-1}, \pi_{i-1}, x_i, w_i) \end{array}\right] = 1.$$

- **Succinctness:** Similar to Definition 2.6, we require two succinctness properties:
  - **Succinct proof size:** There exists a universal polynomial $\mathsf{poly}(\cdot,\cdot)$ such that for every $\lambda \in \mathbb{N}$, $t \le 2^\lambda$, $i \in [t]$, $s \in \mathbb{N}$, we have, $|\pi_i| = \mathsf{poly}(\lambda, s)$ in the completeness experiment above. Moreover, we say the proof is *fully succinct* if $|\pi| = \mathsf{poly}(\lambda, \log s)$.
  - **Succinct verification time:** There exists a universal polynomial $\mathsf{poly}(\cdot,\cdot,\cdot)$ such that for $\lambda \in \mathbb{N}$, $t \le 2^\lambda$, $i \in [t]$, $s \in \mathbb{N}$, $\ell \in \mathbb{N}$, 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 except the adversary outputs the bound on the number of instances T in *binary* (since we implicitly set $T = 2^\lambda$).
- **Perfect zero knowledge:** The scheme satisfies perfect zero knowledge if there exists an efficient simulator $\mathcal{S}$ such that for all $\lambda \in \mathbb{N}$, all bounds $\ell \in \mathbb{N}$, $s \in \mathbb{N}$, all $t \le 2^\lambda$, 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:** Set $h_0, \pi_0 = \perp$. Sample $\mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda, 1^\ell, 1^s)$ and for $i = 1$ to t, $(h_i, \pi_i) \leftarrow \mathsf{P}(\mathsf{crs}, C, h_{i-1}, \pi_{i-1}, x_i, w_i)$. Output (crs, π_t).
  - **Simulated distribution:** Output $(\mathsf{crs}^*, \pi_t^*) \leftarrow \mathcal{S}(1^\lambda, 1^\ell, 1^s, C, (x_1, \ldots, x_t))$.

Strong completeness. For updatable batch arguments, we can define an even stronger notion of completeness which says that *any* valid proof (i.e., not just one output by the honest UpdateP algorithm) on statements $(x_1, \ldots, x_t)$ can be extended to a proof on $(x_1, \ldots, x_{t+1})$. To ensure non-triviality, we require that the empty proof (denoted $\perp$) be a valid proof for the empty tuple of statements (also denoted $\perp$). We state the formal definition below. It is easy to see that strong completeness implies the vanilla version of completeness.

Definition 5.2 (Strong Completeness)**.** We say that an updatable BARG $\Pi_{\mathsf{BARG}} = (\mathsf{Gen}, \mathsf{Hash}, \mathsf{UpdateP}, \mathsf{V})$ satisfies *strong completeness* if for every security parameter $\lambda \in \mathbb{N}$, any $t \le 2^\lambda$, bounds $\ell \in \mathbb{N}$ and $s \in \mathbb{N}$, Boolean circuits $C\colon \{0,1\}^\ell \times \{0,1\}^m \to \{0,1\}$ of size at most s, the following conditions hold:

- $\Pr[\mathsf{V}(\mathsf{crs}, C, \perp, \perp) = 1 : \mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda, 1^\ell, 1^s)] = 1$.

- For all statements $x_1, \ldots, x_t \in \{0,1\}^\ell$, any proof π_{t-1}, any witness $w_t \in \{0,1\}^m$, and any crs in the support of $\mathsf{Gen}(1^\lambda, 1^\ell, 1^s)$ where $\mathsf{V}(\mathsf{crs}, C, (x_1, \ldots, x_{t-1}), \pi_{t-1}) = 1$ and $C(x_t, w_t) = 1$, we have,

$$\Pr\left[\mathsf{V}(\mathsf{crs}, C, (x_1, \ldots, x_t), \pi_t) = 1 : \begin{array}{c} h_{t-1} \leftarrow \mathsf{Hash}(\mathsf{crs}, (x_1, \ldots, x_{t-1})), \\ \pi_t \leftarrow \mathsf{UpdateP}(\mathsf{crs}, C, h_{t-1}, \pi_{t-1}, x_t, w_t) \end{array}\right] = 1.$$

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].

Two-to-one somewhere statistically binding hash functions. Our construction relies 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 5.3 (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 $1^{\ell_{\mathsf{blk}}}$, 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 5.4 (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. We say $\mathsf{ind} \leq \mathsf{ind}'$ if the string ind precedes the string ind' lexicographically (if the strings have uneven length, the shorter one is first padded with 0s on the right to the length of the longer string before comparing them 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 $y = x\|z$. For a length parameter ℓ, we write $\{0,1\}^{\leq \ell}$ to denote the set of bit-strings with length at most ℓ.

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

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

  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

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

Claim 5.5 (Size of Frontier)**.** *Let Γ be a binary tree of height d. Then, for any leaf node $\mathsf{ind} \in \{0,1\}^d$, $|\mathsf{frontier}(\mathsf{ind})| \leq d+1$ (i.e., the frontier of a leaf nodes ind contains at most $d+1$ nodes).*

Proof. The path from the root to $\mathsf{ind} \in \{0,1\}^d$ contains $d+1$ nodes. The frontier of ind includes ind itself along with the left sibling of each node along the path (if one exists). Since the root node has no sibling node, the maximum number of nodes in $\mathsf{frontier}(\mathsf{ind})$ is at most $d+1$. □

Construction 5.6 (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 $\ell_j = \ell_{\mathsf{out}}(\lambda, \ell_{j-1})$.[10] Let $\ell_{\max} = \max(\ell_0, \ldots, \ell_j)$.

[10] 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.

- Let $\Pi_{\mathsf{PRF}} = (\mathsf{PRF.KeyGen}, \mathsf{PRF.Puncture}, \mathsf{PRF.Eval})$ 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 $\Pi_{\mathsf{BARG}} = (\mathsf{Gen}, \mathsf{Hash}, \mathsf{UpdateP}, \mathsf{Verify})$ 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 $K \leftarrow \mathsf{PRF.KeyGen}(1^\lambda)$. 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:

Constants: PRF key K, hash key $\mathsf{hk} = (\mathsf{hk}_1, \ldots, \mathsf{hk}_d)$
Input: Boolean circuit $C\colon \{0,1\}^\ell \times \{0,1\}^m \to \{0,1\}$ of size at most s, node values $h_0, h_1 \in \{0,1\}^{\leq \ell_{\max}}$, index $\mathsf{ind} \in \{0,1\}^{\leq d}$, and proofs $\pi_0, \pi_1 \in \{0,1\}^{\leq \max(m,\lambda)}$

1. If $\mathsf{ind} \in \{0,1\}^d$ (i.e., a leaf in the binary tree),
   (a) Parse h_0 as a statement $x_1 \in \{0,1\}^\ell$ and π_0 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 the hash $h \leftarrow \mathsf{SSB.LocalHash}(\mathsf{hk}_{d'}, h_0, h_1)$.
   (b) Check the following conditions:
   - $\mathsf{PRG}(\pi_0) = \mathsf{PRG}(\mathsf{PRF.Eval}(K, (C, h_0, \mathsf{ind}\|0)))$;
   - $\mathsf{PRG}(\pi_1) = \mathsf{PRG}(\mathsf{PRF.Eval}(K, (C, h_1, \mathsf{ind}\|1)))$.

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

Figure 8: 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.

Figure 9: 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 5.8. Finally, it outputs the common reference string $\mathsf{crs} = (\mathsf{ObfProve}, \mathsf{ObfVerify}, \mathsf{hk})$.

- $\mathsf{Hash}(\mathsf{crs}, (x_1, \ldots, x_t))$: On input a common reference string $\mathsf{crs} = (\mathsf{ObfProve}, \mathsf{ObfVerify}, \mathsf{hk} = (\mathsf{hk}_1, \ldots, \mathsf{hk}_d))$ and sequence of statements $x_1, \ldots, x_t \in \{0,1\}^\ell$, the hash algorithm proceeds as follows:
  1. If $t = 0$ (i.e., the sequence of statements is *empty*), then the hash algorithm outputs $(0, \varnothing)$.
  2. Otherwise, if $t \neq 0$, the algorithm constructs a binary tree $(\Gamma_{\mathsf{hash}}, \mathsf{val}_{\mathsf{hash}}) \leftarrow \mathsf{HashProg}[\mathsf{hk}](x_1, \ldots, x_t)$ of depth d whose values correspond to the statements $(x_1, \ldots, x_t)$ and their hashes. Specifically, we define the $\mathsf{HashProg}[\mathsf{hk}]$ 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 $i \in [1, 2^d]$ be its associated value (when viewed as an integer). For instance, we associate the string 0^d with the integer 1, and more generally, if $\mathsf{ind} = b_1, \ldots, b_d$, then we associated ind with the integer $1 + \sum_{j \in [d]} 2^{d-j} b_j$). Then, we associate a value $\mathsf{val}_{\mathsf{hash}}(\mathsf{ind})$ as follows:
$$\mathsf{val}_{\mathsf{hash}}(\mathsf{ind}) = \begin{cases} x_i & i \le t \\ \bot & \text{otherwise.} \end{cases}$$
- **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 $\mathsf{val}_{\mathsf{hash}}(\mathsf{ind}) \leftarrow \bot$. Recall that $\mathsf{ind} > \mathsf{bin}_d(t-1)$ if the binary string ind follows the binary string $\mathsf{bin}_d(t-1)$ lexicographically.
  * 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,
$$\mathsf{val}_{\mathsf{hash}}(\mathsf{ind}) \leftarrow \mathsf{SSB.LocalHash}\big(\mathsf{hk}_{d'}, \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}\|0), \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}\|1)\big).$$

Output $(\Gamma_{\mathsf{hash}}, \mathsf{val}_{\mathsf{hash}})$. By construction, the value of $\mathsf{val}_{\mathsf{hash}}(\mathsf{ind})$ is $\bot$ whenever $\mathsf{ind} > \mathsf{bin}_d(t-1)$. The number of such indices is at most $2t$, so the value function $\mathsf{val}_{\mathsf{hash}}(\mathsf{ind})$ is defined (i.e., not $\bot$) on at most $2t$ indices.

Figure 10: The function $\mathsf{HashProg}[\mathsf{hk}](x_1, \ldots, x_t)$

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

3. Finally, output the hash $h_t = \Big(t, \{(\mathsf{ind}, \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}))\}_{\mathsf{ind} \in \mathsf{frontier}(\mathsf{ind}^{(t)})}\Big)$.

- $\mathsf{UpdateP}(\mathsf{crs}, C, h_t, \pi_t, x_{t+1}, w_{t+1})$: On input a common reference string $\mathsf{crs} = (\mathsf{ObfProve}, \mathsf{ObfVerify}, \mathsf{hk})$, where $\mathsf{hk} = (\mathsf{hk}_1, \ldots, \mathsf{hk}_d)$, a Boolean circuit $C\colon \{0,1\}^\ell \times \{0,1\}^m \to \{0,1\}$, a hash of t statements denoted by $h_t = (t, \{(\mathsf{ind}, h_{\mathsf{ind}})\}_{\mathsf{ind} \in \mathcal{I}_1})$, a proof $\pi_t = \{(\mathsf{ind}, \pi_{\mathsf{ind}})\}_{\mathsf{ind} \in \mathcal{I}_2}$ on the first t statements where $t \le 2^\lambda$ and $\mathcal{I}_1, \mathcal{I}_2 \subseteq \{0,1\}^{\le d}$, and a witness $w_{t+1} \in \{0,1\}^m$, the update algorithm proceeds as follows:[11]

  1. If $t = 0$, let $\mathsf{ind}^{(1)} = \mathsf{bin}_d(0) = 0^d$. Let $\pi \leftarrow \mathsf{ObfProve}(C, x_1, \bot, \mathsf{ind}^{(1)}, w_1, \bot)$ and output $\{(\mathsf{ind}^{(1)}, \pi)\}$.
  2. If $\mathcal{I}_1 \ne \mathcal{I}_2$, output $\bot$. Otherwise let $\mathcal{I} = \mathcal{I}_1 = \mathcal{I}_2$. Then, 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 $\bot$.
  3. The hash algorithm then defines a binary tree Γ_{hash} of depth d with the following value function $\mathsf{val}_{\mathsf{hash}}$:
     - For each index $\mathsf{ind} \in \mathcal{I}$, let $\mathsf{val}_{\mathsf{hash}}(\mathsf{ind}) = h_{\mathsf{ind}}$.
     - Let $\mathsf{ind}^{(t+1)} = \mathsf{bin}_d(t)$. Let $\mathsf{val}_{\mathsf{hash}}(\mathsf{ind}^{(t+1)}) = x_{t+1}$.
     - For all other nodes $\mathsf{ind} \notin \mathcal{I}$ or $\mathsf{ind} \ne \mathsf{ind}^{(t+1)}$, let $\mathsf{val}_{\mathsf{hash}}(\mathsf{ind}) = \bot$.

     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 hash h_{ind}.
  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}.

[11] Note that if $\pi_t = \bot$, then we interpret $\mathcal{I}_2$ as $\mathcal{I}_2 = \emptyset$.

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
   $$\mathsf{val}_{\mathsf{hash}}(\mathsf{ind}) \leftarrow \mathsf{SSB.LocalHash}\left(\mathsf{hk}_{d-k}, \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}\|0), \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}\|1)\right),$$
   $$\mathsf{val}_{\mathsf{proof}}(\mathsf{ind}) \leftarrow \mathsf{ObfProve}\left(C, h_0, h_1, \mathsf{ind}, \mathsf{val}_{\mathsf{proof}}(\mathsf{ind}\|0), \mathsf{val}_{\mathsf{proof}}(\mathsf{ind}\|1)\right),$$
   where $h_0 \leftarrow \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}\|0)$ and $h_1 \leftarrow \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}\|1)$.
6. Output the updated hash $h_{t+1} = \left(t+1, \{(\mathsf{ind}, \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}))\}_{\mathsf{ind}\in\mathsf{frontier}(\mathsf{ind}^{(t+1)})}\right)$ and the updated proof $\pi_{t+1} = \left\{(\mathsf{ind}, \mathsf{val}_{\mathsf{proof}}(\mathsf{ind}))\right\}_{\mathsf{ind}\in\mathsf{frontier}(\mathsf{ind}^{(t+1)})}$.

- $\mathsf{V}(\mathsf{crs}, C, (x_1, \ldots, x_t), \pi)$: On input a common reference string $\mathsf{crs} = (\mathsf{ObfProve}, \mathsf{ObfVerify}, \mathsf{hk})$, a Boolean circuit $C\colon \{0,1\}^\ell \times \{0,1\}^m \to \{0,1\}$, 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. If $t = 0$ and $\pi = \bot$, then the verification algorithm outputs 1. If $t = 0$ and $\pi \neq \bot$, then the verification algorithm outputs 0.
  2. Otherwise, let $\mathsf{ind}^{(t)} = \mathsf{bin}_d(t-1)$. If $\mathcal{I} \neq \mathsf{frontier}(\mathsf{ind}^{(t)})$, output $\bot$.
  3. The algorithm runs $h_t \leftarrow \mathsf{Hash}(\mathsf{crs}, (x_1, \ldots, x_t))$. Parse $h_t = \left(t, \{(\mathsf{ind}, \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}))\}_{\mathsf{ind}\in\mathsf{frontier}(\mathsf{ind}^{(t)})}\right)$.
  4. 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.

Theorem 5.7 (Strong Completeness)**.** *If $i\mathcal{O}$ is correct, then Construction 5.6 satisfies strong completeness.*

Proof. Take any security parameter $\lambda \in \mathbb{N}$, any $t \leq 2^\lambda$, any bounds $\ell, s \in \mathbb{N}$, and any Boolean circuit $C\colon \{0,1\}^\ell \times \{0,1\}^m \to \{0,1\}$ of size at most s. Let $\mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda, 1^\ell, 1^s)$. First, $\mathsf{V}(\mathsf{crs}, C, \bot, \bot)$ always outputs 1 by construction. For the main requirement, take any sequence of statements $x_1, \ldots, x_t \in \{0,1\}^\ell$, any proof π_{t-1}, and any witness $w_t \in \{0,1\}^m$ where $\mathsf{V}(\mathsf{crs}, C, (x_1, \ldots, x_{t-1}), \pi_{t-1}) = 1$ and $C(x_t, w_t) = 1$. For each $j \in [t]$, let $\mathsf{ind}^{(j)} = \mathsf{bin}_d(j-1)$. We consider two cases depending on the value of t.

Case 1. Suppose $t = 1$. Let $h_0 \leftarrow \mathsf{Hash}(\mathsf{crs}, \bot)$. By construction, $h_0 = (0, \varnothing)$. Since $\mathsf{V}(\mathsf{crs}, C, \bot, \pi_0) = 1$, it must be the case that $\pi_0 = \bot$. Consider the proof π_1 output by $\mathsf{UpdateP}(\mathsf{crs}, C, h_0, \pi_0, x_1, w_1)$. Since $t = 0$, UpdateP algorithm computes a proof π by evaluating $\pi \leftarrow \mathsf{ObfProve}(C, x_1, \bot, \mathsf{ind}^{(1)}, w_1, \bot)$. By correctness of $i\mathcal{O}$, and the fact that $C(x_1, w_1) = 1$,

$$\pi = \mathsf{PRF.Eval}\left(K, \left(C, x_1, \mathsf{ind}^{(1)}\right)\right).$$

By construction, UpdateP outputs the proof

$$\pi_1 = \left\{\left(\mathsf{ind}^{(1)}, \mathsf{PRF.Eval}(K, (C, x_1, \mathsf{ind}^{(1)}))\right)\right\}.$$

Now consider the output of $\mathsf{V}(\mathsf{crs}, C, (x_1), \pi_1)$. The verification algorithm first computes $\mathsf{ind}^{(1)} = \mathsf{bin}_d(0)$. Since $\mathsf{ind}^{(1)}$ is the leftmost node, we have $\mathsf{frontier}(\mathsf{ind}^{(1)}) = \mathsf{ind}^{(1)}$, and the first check succeeds. Next, the verification algorithm computes $h_1 \leftarrow \mathsf{Hash}(\mathsf{crs}, (x_1))$. By construction, this means

$$h_1 = \left(1, \left\{\left(\mathsf{ind}^{(1)}, \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}^{(1)})\right)\right\}\right).$$

By construction of Hash, we have that $\mathsf{val}_{\mathsf{hash}}(\mathsf{ind}^{(1)}) = x_1$. Then, the verification algorithm runs ObfVerify on the input $\left(C, x_1, \mathsf{ind}^{(1)}, \mathsf{PRF.Eval}(K, (C, x_1, \mathsf{ind}^{(1)}))\right)$. By correctness of $i\mathcal{O}$, the program check succeeds. Since all the checks pass, the verification algorithm outputs 1.

Case 2. Suppose $t > 1$. First, write $\pi_{t-1} = \{(\mathsf{ind}, \pi_{\mathsf{ind}})\}_{\mathsf{ind}\in\mathcal{I}}$. Similarly, let $h_{t-1} \leftarrow \mathsf{Hash}(\mathsf{crs}, (x_1, \ldots, x_{t-1}))$. By construction of Hash, we can write

$$h_{t-1} = \left(t-1, \{(\mathsf{ind}, \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}))\}_{\mathsf{ind}\in\mathsf{frontier}(\mathsf{ind}^{(t-1)})}\right). \tag{5.1}$$

Let $\mathsf{ind}^{(t-1)} = \mathsf{bin}_d(t-2)$. Since π_{t-1} is a valid proof on $(x_1, \ldots, x_t)$, the following properties hold by definition of V:

- The set of indices $\mathcal{I}$ associated with π_{t-1} satisfies $\mathcal{I} = \mathsf{frontier}(\mathsf{ind}^{(t-1)})$.
- Since verification succeeds, for all $\mathsf{ind} \in \mathsf{frontier}(\mathsf{ind}^{(t-1)})$, $\mathsf{ObfVerify}(C, \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}), \mathsf{ind}, \pi_{\mathsf{ind}}) = 1$. By correctness of $i\mathcal{O}$, this means that,

$$\mathsf{PRG}(\pi_{\mathsf{ind}}) = \mathsf{PRG}(\mathsf{PRF.Eval}(K, (C, \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}), \mathsf{ind}))). \tag{5.2}$$

Consider the proof π_t computed by $\mathsf{UpdateP}(\mathsf{crs}, C, h_{t-1}, \pi_{t-1}, x_t, w_t)$ where h_{t-1} and π_{t-1} are defined above.

- Since $t - 1 > 0$, and from the checks above, h_{t-1} and π_{t-1} are both computed on the set $\mathcal{I} = \mathsf{frontier}(\mathsf{ind}^{(t-1)})$.
- The update algorithm defines a binary tree Γ_{hash} and defines $\mathsf{val}_{\mathsf{hash}}$ on $\mathcal{I} = \mathsf{frontier}(\mathsf{ind}^{(t-1)})$ using the values taken from h_{t-1} (i.e., this defines $\mathsf{val}_{\mathsf{hash}}$ on all $\mathsf{ind} \in \mathcal{I}$). In addition, it sets $\mathsf{val}_{\mathsf{hash}}(\mathsf{ind}^{(t)}) = x_t$. For all other nodes $\mathsf{ind} \notin \mathcal{I} \cup \{\mathsf{ind}^{(t)}\}$, it sets $\mathsf{val}_{\mathsf{hash}}(\mathsf{ind}) = \bot$.
- Similarly, the update algorithm defines a binary tree Γ_{proof} and sets $\mathsf{val}_{\mathsf{proof}}$ on $\mathcal{I} = \mathsf{frontier}(\mathsf{ind}^{(t-1)})$ using the values taken from π_{t-1}. It sets $\mathsf{val}_{\mathsf{proof}}(\mathsf{ind}^{(t)}) = \mathsf{ObfProve}(\mathsf{crs}, C, x_t, \bot, \mathsf{ind}^{(t)}, w_t, \bot)$. For all other nodes $\mathsf{ind} \notin \mathcal{I} \cup \{\mathsf{ind}^{(t)}\}$, it sets $\mathsf{val}_{\mathsf{hash}}(\mathsf{ind}) = \bot$.
- Since $C(x_t, w_t) = 1$, by correctness of $i\mathcal{O}$, we have that,

$$\pi_{\mathsf{ind}^{(t)}} = \mathsf{val}_{\mathsf{proof}}(\mathsf{ind}^{(t)}) = \mathsf{PRF.Eval}(K, (C, x_t, \mathsf{ind}^{(t)})). \tag{5.3}$$

Let ind' be the longest common prefix to $\mathsf{ind}^{(t-1)}$ and $\mathsf{ind}^{(t)}$ and let $\rho = |\mathsf{ind}'|$. Let $\mathsf{ind}^{(t-1)} = b_1 \cdots b_d$ and $\mathsf{ind}' = b_1 \cdots b_\rho$. Let $\mathcal{I}^* = \{\mathsf{ind}^{(t)}\} \cup \mathsf{frontier}(\mathsf{ind}^{(t-1)})$. Observe first that $\mathsf{ind}^{(t-1)} \neq \mathsf{ind}^{(t)}$, so the length of their longest prefix is less than d (i.e., $0 \leq \rho < d$). We now consider two possibilities:

- **Case (i):** Suppose $\rho = d - 1$. Then $\mathsf{ind}^{(t-1)} = \mathsf{ind}'\|0$ and $\mathsf{ind}^{(t)} = \mathsf{ind}'\|1$. This corresponds to the case where $\mathsf{ind}^{(t-1)}$ and $\mathsf{ind}^{(t)}$ are siblings in the binary tree. Thus, in this case,

$$\mathsf{frontier}(\mathsf{ind}^{(t)}) = \mathsf{frontier}(\mathsf{ind}^{(t-1)}) \cup \{\mathsf{ind}^{(t)}\} = \mathcal{I}^*.$$

  In this case, the hash value h_t and the proof π_t include the same set of components as in h_{t-1} and π_{t-1} along with additional values corresponding to the hash value and the proof associated with node $\mathsf{ind}^{(t)}$. We now show that $\mathsf{V}(\mathsf{crs}, C, (x_1, \ldots, x_t), \pi_t)$ outputs 1.

  - The verification algorithm starts by computing $h_t \leftarrow \mathsf{Hash}(\mathsf{crs}, (x_1, \ldots, x_t))$. By definition, this means that

$$h_t = \left(t, \{(\mathsf{ind}, \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}))\}_{\mathsf{ind}\in\mathsf{frontier}(\mathsf{ind}^{(t-1)})} \cup \left\{(\mathsf{ind}^{(t)}, x_t)\right\}\right),$$

    where $\{(\mathsf{ind}, \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}))\}_{\mathsf{ind}\in\mathsf{frontier}(\mathsf{ind}^{(t-1)})}$ are the same values as in h_{t-1}.
  - Consider now the verification relation. As argued above, validity of π_{t-1} for statements $(x_1, \ldots, x_{t-1})$ means that that for all $\mathsf{ind} \in \mathsf{frontier}^{(t-1)}$,

$$\mathsf{PRG}(\pi_{\mathsf{ind}}) = \mathsf{PRG}(\mathsf{PRF.Eval}(K, (C, \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}), \mathsf{ind}))).$$

    Moreover, by Eq. (5.3), the verification relation also holds for $\mathsf{ind} = \mathsf{ind}^{(t)}$, and the verification algorithm outputs 1.

- **Case (ii):** Suppose $\rho < d-1$. Then the update algorithm constructs the proof iteratively. Let k be the loop counter (i.e., k ranges from $d-1$ to $\rho+1$). Since $\mathsf{ind}^{(t-1)}$ and $\mathsf{ind}^{(t)}$ are adjacent leaves in the binary tree, we can write them as $\mathsf{ind}^{(t-1)} = b_1 \cdots b_\rho 01 \cdots 1$ and $\mathsf{ind}^{(t)} = b_1 \cdots b_\rho 10 \cdots 0$. Let $(\Gamma'_{\mathsf{hash}}, \mathsf{val}'_{\mathsf{hash}}) \leftarrow \mathsf{HashProg}[\mathsf{hk}](x_1, \ldots, x_t)$ (using the algorithm from Fig. 10). Let $\mathcal{I}^*_d = \{\mathsf{ind}^{(t)}\} \cup \mathsf{frontier}(\mathsf{ind}^{(t-1)})$, and for each $k \in \{\rho+1, \ldots, d-1\}$, define $\mathcal{I}^*_k = \mathcal{I}^*_{k+1} \cup \{\mathsf{ind}_k\}$, where $\mathsf{ind}_k = b_1 b_2 \cdots b_k$ is the index that the update algorithm processes in iteration k. We now show that the following invariant holds for all $k \in \{\rho+1, \ldots, d\}$:

  for all indices $\mathsf{ind} \in \mathcal{I}^*_k$*, it holds that*

  $$\mathsf{PRG}(\mathsf{val}_{\mathsf{proof}}(\mathsf{ind})) = \mathsf{PRG}(\mathsf{PRF.Eval}(K, (C, \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}), \mathsf{ind}))); \tag{5.4}$$

  $$\mathsf{val}_{\mathsf{hash}}(\mathsf{ind}) = \mathsf{val}'_{\mathsf{hash}}(\mathsf{ind}). \tag{5.5}$$

  First, we use the fact that $\mathsf{V}(\mathsf{crs}, C, (x_1, \ldots, x_{t-1}), \pi) = 1$ to argue that the invariant holds at the beginning of the update process (for the initial $\mathcal{I}^*_d$):

  - Consider an index $\mathsf{ind} \in \mathsf{frontier}(\mathsf{ind}^{(t-1)})$. Then the above analysis (see Eq. (5.2)) shows that the first property (Eq. (5.4)) holds. It suffices to show that the second property (Eq. (5.5)) also holds for such indices. For all $\mathsf{ind} \in \mathsf{frontier}(\mathsf{ind}^{(t-1)})$, since $\mathsf{ind}^{(t-1)} < \mathsf{ind}^{(t)}$, ind cannot be a prefix of $\mathsf{ind}^{(t)}$ (recall that ind is a *left* sibling of a node on the path to $\mathsf{ind}^{(t-1)}$). By construction of HashProg, the value of any node only depends on the values of its descendants. Correspondingly, this means that $\mathsf{val}'_{\mathsf{hash}}(\mathsf{ind})$ only depends on the values of $\mathsf{val}_{\mathsf{hash}}(\mathsf{ind}^{(1)}) = x_1, \ldots, \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}^{(t-1)}) = x_{t-1}$, or in other words, the first $t-1$ nodes of the tree. This precisely coincides with the values obtained by computing $\mathsf{HashProg}[\mathsf{hk}](x_1, \ldots, x_{t-1})$. Recall that $\mathsf{ind}^{(t-1)} = b_1 b_2 \cdots b_d$. By Eq. (5.1), we conclude that for all $\mathsf{ind} \in \mathsf{frontier}(\mathsf{ind}^{(t-1)})$, $\mathsf{val}'_{\mathsf{hash}}(\mathsf{ind}) = \mathsf{val}_{\mathsf{hash}}(\mathsf{ind})$.
  - For $\mathsf{ind} = \mathsf{ind}^{(t)}$, the update algorithm sets $\mathsf{val}_{\mathsf{proof}}(\mathsf{ind}) = \mathsf{PRF.Eval}(K, (C, x_t, \mathsf{ind}^{(t)}))$ (see Eq. (5.3)) so the first requirement holds. Moreover, by construction of HashProg, $\mathsf{val}'_{\mathsf{hash}}(\mathsf{ind}^{(t)}) = x_t = \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}^{(t)})$, so Eq. (5.5) of the invariant also holds.

  We now show that the invariant continues to hold at the end of each iteration:

  - **Base case:** When $k = d-1$, the update algorithm sets $\mathsf{ind} = \mathsf{ind}_{d-1} = b_1 \cdots b_{d-1}$ (i.e., the first $d-1$ bits of $\mathsf{ind}^{(t-1)}$).
    * Since $\mathsf{ind}\|1 = \mathsf{ind}^{(t-1)}$, this means that $\mathsf{ind}\|0$ is a left sibling of $\mathsf{ind}^{(t-1)}$. By definition, both $\mathsf{ind}\|0$ and $\mathsf{ind}\|1$ are contained in the set $\mathsf{frontier}(\mathsf{ind}^{(t-1)}) \subseteq \mathcal{I}^*$.
    * Algorithm UpdateP runs $\mathsf{val}_{\mathsf{hash}}(\mathsf{ind}) \leftarrow \mathsf{SSB.LocalHash}(\mathsf{hk}_{d-k}, \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}\|0), \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}\|1))$. Additionally, it runs $\mathsf{val}_{\mathsf{proof}}(\mathsf{ind}) \leftarrow \mathsf{ObfProve}\left(C, h_0, h_1, \mathsf{ind}, \mathsf{val}_{\mathsf{proof}}(\mathsf{ind}\|0), \mathsf{val}_{\mathsf{proof}}(\mathsf{ind}\|1)\right)$, where $h_0 \leftarrow \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}\|0)$ and $h_1 \leftarrow \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}\|1)$. By correctness of $i\mathcal{O}$, this means that

      $$\mathsf{val}_{\mathsf{proof}}(\mathsf{ind}) = \mathsf{Prove}[K, \mathsf{hk}]\left(C, h_0, h_1, \mathsf{ind}, \mathsf{val}_{\mathsf{proof}}(\mathsf{ind}\|0), \mathsf{val}_{\mathsf{proof}}(\mathsf{ind}\|1)\right).$$

    * By construction, the Prove program checks that Eq. (5.4) holds on $\mathsf{ind}\|0, \mathsf{ind}\|1$. Since $\mathsf{ind}\|0$ and $\mathsf{ind}\|1$ are both in $\mathsf{frontier}(\mathsf{ind}^{(t-1)})$, the checks pass by the above analysis (see Eq. (5.2)). In this case, the Prove program outputs $\mathsf{PRF.Eval}(K, (C, \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}), \mathsf{ind}))$, which is the value of $\mathsf{val}_{\mathsf{proof}}(\mathsf{ind})$. Clearly, Eq. (5.4) holds for index ind.
    * By construction, ind is a prefix of $\mathsf{ind}^{(t-1)}$ but *not* a prefix of $\mathsf{ind}^{(t)}$. Since $\mathsf{ind}^{(t-1)} < \mathsf{ind}^{(t)}$, this means that the descendants of ind *cannot* include $\mathsf{ind}^{(t)}$. By the same argument as above, we can appeal to the construction of HashProg to conclude that $\mathsf{val}'_{\mathsf{hash}}(\mathsf{ind})$ is a function only of $(x_1, \ldots, x_{t-1})$ and so $\mathsf{val}_{\mathsf{hash}}(\mathsf{ind}) = \mathsf{val}'_{\mathsf{hash}}(\mathsf{ind})$. Thus, ind satisfies Eq. (5.5).

    At the end of this step, we see that the invariant holds for index $\mathsf{ind} = \mathsf{ind}_{d-1}$. Since the invariant holds for $\mathcal{I}^*_d$, it now holds for $\mathcal{I}^*_d \cup \{\mathsf{ind}_{d-1}\} = \mathcal{I}^*_{d-1}$, as required.

– **Iterative case:** When $\rho + 1 \le k < d - 1$, the update algorithm sets $\mathsf{ind}_k = \mathsf{ind} = b_1 \cdots b_k$. Since $\mathsf{ind}^{(t-1)} = b_1 \cdots b_\rho 01 \cdots 1$ and $\mathsf{ind}^{(t)} = b_1 \cdots b_\rho 10 \cdots 0$, it must be the case that $b_{k+1} = 1$. First, we observe that $\mathsf{ind}\|0, \mathsf{ind}\|1 \in \mathcal{I}^*_{t+1}$:

  * Since $\mathsf{ind}\|0$ is a left sibling of $\mathsf{ind}\|1$ and $\mathsf{ind}\|1$ is a prefix of $\mathsf{ind}^{(t-1)}$, we conclude that $\mathsf{ind}\|0 \in \mathsf{frontier}(\mathsf{ind}^{(t-1)}) \subseteq \mathcal{I}^*_{k+1}$.
  * Since $b_{k+1} = 1$, $\mathsf{ind}\|1 = b_1 \cdots b_{k+1} = \mathsf{ind}_{k+1} \in \mathcal{I}^*_{k+1}$.

  Since the invariant holds for $\mathcal{I}^*_{k+1}$, Eq. (5.4) holds for both $\mathsf{ind}\|0$ and $\mathsf{ind}\|1$. Now, by the same analysis as in the base case, we conclude that both Eq. (5.4) and Eq. (5.5) holds for index ind.

To complete the proof, we consider the behavior of the verification algorithm. Since $\mathsf{ind}^{(t)} = b_1 \cdots b_\rho 10 \cdots 0$, every $\mathsf{ind} \in \mathsf{frontier}(\mathsf{ind}^{(t)})$ satisfies one of the following three conditions:

– $\mathsf{ind} = \mathsf{ind}^{(t)}$ and thus, $\mathsf{ind} \in \mathcal{I}^*_d \subset \mathcal{I}^*_{\rho+1}$;
– ind is a left sibling of a prefix of $\mathsf{ind}' = b_1 \cdots b_\rho$, and thus, $\mathsf{ind} \in \mathsf{frontier}(\mathsf{ind}^{(t-1)}) \subset \mathcal{I}^*_d \subset \mathcal{I}^*_{\rho+1}$; or
– $\mathsf{ind} = b_1 \cdots b_\rho 0 = \mathsf{ind}_{\rho+1} \in \mathcal{I}^*_{\rho+1}$.

Thus, we conclude that $\mathsf{frontier}(\mathsf{ind}^{(t)}) \subseteq \mathcal{I}^*_{\rho+1}$. We now show that $\mathsf{V}(\mathsf{crs}, C, (x_1, \ldots, x_t), \pi_t)$ outputs 1.

– The verification algorithm computes the hash h_t using Hash

$$h_t = \left(t, \left\{(\mathsf{ind}, \mathsf{val}'_{\mathsf{hash}}(\mathsf{ind}))\right\}_{\mathsf{ind} \in \mathsf{frontier}(\mathsf{ind}^{(t)})}\right),$$

  where $(\Gamma'_{\mathsf{hash}}, \mathsf{val}'_{\mathsf{hash}}) \leftarrow \mathsf{HashProg}[\mathsf{hk}](x_1, \ldots, x_t)$. By the invariant, $\mathsf{val}'_{\mathsf{hash}}(\mathsf{ind}) = \mathsf{val}_{\mathsf{hash}}(\mathsf{ind})$ for all $\mathsf{ind} \in \mathsf{frontier}(\mathsf{ind}^{(t)})$. Namely, the hash tree computed by the verification algorithm is the same as that computed by the update algorithm.
– The verification algorithm checks that $\mathsf{ObfVerify}(C, \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}), \mathsf{ind}, \pi_{\mathsf{ind}}) = 1$ for all $\mathsf{ind} \in \mathsf{frontier}(\mathsf{ind}^{(t)})$. By correctness of $i\mathcal{O}$, this corresponds to checking that Eq. (5.4) holds for all $\mathsf{ind} \in \mathsf{frontier}(\mathsf{ind}^{(t)})$, which is precisely our invariant condition. Since $\mathsf{frontier}(\mathsf{ind}^{(t)}) \subseteq \mathcal{I}^*_{\rho+1}$, the claim holds.

We conclude that strong completeness holds in this case. □

Theorem 5.8 (Soundness). *If Π_{PRF} is correct and satisfies punctured pseudorandomness,* PRG *is a secure PRG, Π_{SSB} is a two-to-one somewhere statistically binding hash function, and $i\mathcal{O}$ is secure, then Construction 5.6 satisfies non-adaptive soundness.*

Proof. We begin by defining a sequence of hybrid experiments:

- Hyb_0: This is the non-adaptive soundness experiment:
  – Adversary $\mathcal{A}$, on input 1^λ, outputs the maximum circuit size $1^{s(\lambda)}$, a Boolean circuit C^*_λ of size at most $s(\lambda)$, and statements $x^*_1, \ldots, x^*_{t_\lambda}$ where $t_\lambda \le 2^\lambda$. The challenger checks that there exists $i^*_\lambda \in [t_\lambda]$ such that $C^*_\lambda(x^*_{i^*_\lambda}, w) = 0$ for all $w \in \{0,1\}^*$. If such an index i^* does not exist, the experiment aborts and the challenger outputs 0. For ease of notation, we simply write $C^* = C^*_\lambda$, $t = t_\lambda$, and $i^* = i^*_\lambda$ in the following description.
  – The challenger samples $\mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda, 1^\ell, 1^s)$ and gives $\mathsf{crs} = (\mathsf{ObfProve}, \mathsf{ObfVerify}, \mathsf{hk})$ to $\mathcal{A}$. By construction, $\mathsf{hk} = (\mathsf{hk}_1, \ldots, \mathsf{hk}_d)$.
  – Adversary $\mathcal{A}$ outputs a proof π.
  – The output of the experiment is $\mathsf{Verify}(\mathsf{crs}, C^*, (x^*_1, \ldots, x^*_t), \pi)$.
- Hyb_1: Same as Hyb_0, except the challenger samples the hash keys $\mathsf{hk}_1, \ldots, \mathsf{hk}_d$ to bind on the bits of i^*. Specifically, let $\mathsf{ind}^{(i^*)} = \mathsf{bin}_d(i^* - 1) = b_1 \cdots b_d \in \{0,1\}^d$. For each $j \in [d]$, the challenger samples $\mathsf{hk}_j \leftarrow \mathsf{SSB.GenTD}(1^\lambda, 1^{\ell_{j-1}}, b_{d+1-j})$.

- Hyb_2: Same as Hyb_1, except when constructing the CRS, the challenger changes how it constructs the obfuscated programs in the CRS:

  1. First, the challenger samples the hash keys $\mathsf{hk} = (\mathsf{hk}_1, \ldots, \mathsf{hk}_d)$ exactly as in Hyb_1.
  2. Next, the challenger constructs a binary tree $(\Gamma_{\mathsf{hash}}, \mathsf{val}_{\mathsf{hash}}) \leftarrow \mathsf{HashProg}[\mathsf{hk}](x_1^*, \ldots, x_t^*)$ using the algorithm from Fig. 10.
  3. Let $\mathsf{ind}^{(i^*)} = \mathsf{bin}_d(i^* - 1) = b_1 \cdots b_d$. For all $i \in [0, j]$, let $\mathsf{prefix}_{i^*}^{(i)} = b_1 \cdots b_{d-i}$ be the prefix of $\mathsf{ind}^{(i^*)}$ of length $d - i$. Let $\mathcal{X} = \{(\mathsf{prefix}_{i^*}^{(i)}, \mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}_{i^*}^{(i)}))\}_{i \in \{0,\ldots,d\}}$.
  4. The challenger now defines the modified prover program $\mathsf{Prove}'[K, \mathsf{hk}, C^*, \mathcal{X}, 0, \mathsf{ind}^{(i^*)}]$ and verifier program $\mathsf{Verify}'[K, C^*, \mathcal{X}, 0, \mathsf{ind}^{(i^*)}]$ as follows:

     Constants: PRF key K, hash key $\mathsf{hk} = (\mathsf{hk}_1, \ldots, \mathsf{hk}_d)$, Boolean circuit C^*, set $\mathcal{X}$, level d_{thresh}, index $\mathsf{ind}^{(i^*)}$
     Input: Boolean circuit $C\colon \{0,1\}^\ell \times \{0,1\}^m \to \{0,1\}$ of size at most s, node values $h_0, h_1 \in \{0,1\}^{\leq \ell_{\max}}$, index $\mathsf{ind} \in \{0,1\}^{\leq d}$, proofs $\pi_0, \pi_1 \in \{0,1\}^{\max(m,\lambda)}$

     (a) If $\mathsf{ind} \in \{0,1\}^d$ (i.e., a leaf in the binary tree),
        i. If $C = C^*$, ind is a prefix of $\mathsf{ind}^{(i^*)}$, and $(\mathsf{ind}, h_0) \in \mathcal{X}$, then output $\perp$.
        ii. Parse h_0 as a statement $x_1 \in \{0,1\}^\ell$ and π_0 as a witness $w_1 \in \{0,1\}^m$.
        iii. If $C(x_1, w_1) \neq 1$, output $\perp$. Otherwise, output $\mathsf{PRF.Eval}(K, (C, x_1, \mathsf{ind}))$.

     (b) Otherwise, if $\mathsf{ind} \in \{0,1\}^{<d}$ (i.e., an internal node in the binary tree),
        i. Let $d' = \mathsf{level}(\mathsf{ind})$ and compute the hash $h \leftarrow \mathsf{SSB.LocalHash}(\mathsf{hk}_{d'}, h_0, h_1)$.
        ii. If $d' \leq d_{\mathsf{thresh}}$, $C = C^*$, ind is a prefix of $\mathsf{ind}^{(i^*)}$, and $(\mathsf{ind}, h) \in \mathcal{X}$, then output $\perp$.
        iii. Check the following conditions:
           - $\mathsf{PRG}(\pi_0) = \mathsf{PRG}(\mathsf{PRF.Eval}(K, (C, h_0, \mathsf{ind}\|0)))$;
           - $\mathsf{PRG}(\pi_1) = \mathsf{PRG}(\mathsf{PRF.Eval}(K, (C, h_1, \mathsf{ind}\|1)))$.

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

Figure 11: Program $\mathsf{Prove}'[K, \mathsf{hk}, C^*, \mathcal{X}, d_{\mathsf{thresh}}, \mathsf{ind}^{(i^*)}]$

Constants: PRF key K, circuit C^*, set $\mathcal{X}$, level d_{thresh}, index $\mathsf{ind}^{(i^*)}$
Input: Boolean circuit C of size at most s, node value $h \in \{0,1\}^{\leq \ell_{\max}}$, index $\mathsf{ind} \in \{0,1\}^{\leq d}$, proof $\pi \in \{0,1\}^\lambda$

(a) Let $d' = \mathsf{level}(\mathsf{ind})$. If $d' < d_{\mathsf{thresh}}$, $C = C^*$, ind is a prefix of $\mathsf{ind}^{(i^*)}$, and $(\mathsf{ind}, h) \in \mathcal{X}$, then output 0.
(b) Output 1 if $\mathsf{PRG}(\pi) = \mathsf{PRG}(\mathsf{PRF.Eval}(K, (C, h, \mathsf{ind})))$ and 0 otherwise.

Figure 12: Program $\mathsf{Verify}'[K, C^*, \mathcal{X}, d_{\mathsf{thresh}}, \mathsf{ind}^{(i^*)}]$

  5. When constructing the CRS, the challenger computes $\mathsf{ObfProve} \leftarrow i\mathcal{O}(1^\lambda, \mathsf{Prove}'[K, \mathsf{hk}, C^*, \mathcal{X}, 0, \mathsf{ind}^{(i^*)}])$ and $\mathsf{ObfVerify} \leftarrow i\mathcal{O}(1^\lambda, \mathsf{Verify}'[K, C^*, \mathcal{X}, d_{\mathsf{thresh}}, \mathsf{ind}^{(i^*)}])$. The challenger pads the size of the proving circuit $\mathsf{Prove}'[K, \mathsf{hk}, C^*, \mathcal{X}, 0, \mathsf{ind}^{(i^*)}]$ and verification circuit $\mathsf{Verify}'[K, C^*, \mathcal{X}, d_{\mathsf{thresh}}, \mathsf{ind}^{(i^*)}]$ to the maximum size of any circuit that appear in the proof of Theorem 5.8. The challenger gives $\mathsf{crs} = (\mathsf{ObfProve}, \mathsf{ObfVerify}, \mathsf{hk})$ to $\mathcal{A}$.
  6. The remainder of the experiment proceeds identically to Hyb_1.

- Hyb_{j+2} for $j \in [d]$: Same as Hyb_{j+1}, except when constructing the CRS, the challenger computes $\mathsf{ObfProve} \leftarrow i\mathcal{O}(1^\lambda, \mathsf{Prove}'[K, \mathsf{hk}, C^*, \mathcal{X}, j, \mathsf{ind}^{(i^*)}])$ and $\mathsf{ObfVerify} \leftarrow i\mathcal{O}(1^\lambda, \mathsf{Verify}'[K, C^*, \mathcal{X}, j, \mathsf{ind}^{(i^*)}])$, where Prove' and Verify' are the programs in Fig. 11 and Fig. 12.

For an adversary $\mathcal{A}$, we write $\mathsf{Hyb}_i(\mathcal{A})$ to denote the output distribution of $\mathsf{Hyb}_i(\mathcal{A})$ with adversary $\mathcal{A}$. As in the proof of Theorem 3.3, we model the adversary $\mathcal{A}$ as a deterministic *non-uniform* algorithm that takes as input the

security parameter 1^λ (and advice string ρ_λ), and outputs the maximum circuit size $1^{s(\lambda)}$, a Boolean circuit C^*_λ of size at most $s(\lambda)$, and statements $x^*_1, x^*_2, \ldots, x^*_{t_\lambda}$ where $t_\lambda \le 2^\lambda$. If the advantage of $\mathcal{A}$ is non-zero in the non-adaptive soundness game, it must be the case that there exists an index $i^*_\lambda \in [t_\lambda]$ such that $C^*_\lambda(x^*_{i^*_\lambda}, w) = 0$ for all $w \in \{0,1\}^*$. If there are multiple such indices, we define i^*_λ to be the first such index. In the following, we will consider deterministic non-uniform reduction algorithms that are provided $(\rho_\lambda, i^*_\lambda)$ as advice. We now show that each pair of adjacent distributions defined above are indistinguishable.

Lemma 5.9. *Suppose Π_{SSB} satisfies index hiding. Then for every non-uniform polynomial time adversary $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that for all $\lambda \in \mathbb{N}$, $|\Pr[\mathsf{Hyb}_0(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_1(\mathcal{A}) = 1]| = \mathsf{negl}(\lambda)$.*

Proof. We begin by introducing a sequence of intermediate hybrids. First, we set $\mathsf{Hyb}^{(0)}_0 \equiv \mathsf{Hyb}_0$. Then for $j \in [d]$, we define experiment $\mathsf{Hyb}^{(j)}_0$ as follows:

- $\mathsf{Hyb}^{(j)}_0$: Same as $\mathsf{Hyb}^{(j-1)}_0$ except when constructing the CRS, the challenger uses i^* to sample hk_j. Specifically, let $\mathsf{ind}^{(i^*)} = \mathsf{bin}_d(i^* - 1) = b_1 \cdots b_d$. The challenger samples $\mathsf{hk}_j \leftarrow \mathsf{SSB.GenTD}(1^\lambda, 1^{\ell_{j-1}}, b_{d+1-j})$.

We now analyze each adjacent pair of intermediate hybrid experiments:

Claim 5.10. *Suppose Π_{SSB} satisfies index hiding. Then for all $j \in [d]$ and every non-uniform polynomial time adversary $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\cdot)$, such that for all $\lambda \in \mathbb{N}$, $|\Pr[\mathsf{Hyb}^{(j-1)}_0(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}^{(j)}_0(\mathcal{A}) = 1]| = \mathsf{negl}(\lambda)$.*

Proof. Let $\mathcal{A}$ be an efficient non-uniform adversary (with advice string ρ_λ) where

$$|\Pr[\mathsf{Hyb}^{(j-1)}_0(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}^{(j)}_0(\mathcal{A}) = 1]| \ge \varepsilon,$$

and ε is non-negligible. We use $\mathcal{A}$ to construct a non-uniform adversary $\mathcal{B}$ with advice string $(\rho, i^*) = (\rho_\lambda, i^*_\lambda)$ that breaks index hiding:

1. Algorithm $\mathcal{B}$ runs algorithm $\mathcal{A}$ on input 1^λ and advice ρ. Algorithm $\mathcal{A}$ outputs the maximum circuit size 1^s, a Boolean circuit C^* of size at most s, and statements $x_1, \ldots, x_t$ where $t \le 2^\lambda$.
2. Next, algorithm $\mathcal{B}$ sets $\mathsf{ind}^{(i^*)} = \mathsf{bin}_d(i^* - 1) = b_1 \cdots b_d \in \{0,1\}^d$ and sends index $b_{d+1-j} \in \{0,1\}$ to the challenger.
3. Algorithm $\mathcal{B}$ receives hk_j from the challenger. If $b = 0$, the challenger sampled $\mathsf{hk}_j \leftarrow \mathsf{SSB.Gen}(1^\lambda, 1^{\ell_{j-1}})$ and if $b = 1$, the challenger sampled $\mathsf{hk}_j \leftarrow \mathsf{SSB.GenTD}(1^\lambda, 1^{\ell_{j-1}}, b_{d+1-j})$.
4. Then, for each $k \in [d]$, algorithm $\mathcal{B}$ computes the hk_k as follows:
   - If $k < j$, set $\mathsf{hk}_k \leftarrow \mathsf{SSB.GenTD}(1^\lambda, 1^{\ell_{k-1}}, b_{d+1-k})$.
   - If $k > j$, set $\mathsf{hk}_k \leftarrow \mathsf{SSB.Gen}(1^\lambda, 1^{\ell_{k-1}})$.
5. Algorithm $\mathcal{B}$ constructs ObfProve, ObfVerify according to Hyb_0 and sets $\mathsf{crs} = (\mathsf{ObfProve}, \mathsf{ObfVerify}, \mathsf{hk})$ where $\mathsf{hk} = (hk_1, \ldots, \mathsf{hk}_d)$. Algorithm $\mathcal{B}$ gives crs to $\mathcal{A}$.
6. After $\mathcal{A}$ outputs the proof π, algorithm $\mathcal{B}$ outputs $\mathsf{Verify}(\mathsf{crs}, C^*, (x^*_1, \ldots, x^*_t), \pi)$.

By construction, if $b = 0$, algorithm $\mathcal{B}$ perfectly simulates distribution $\mathsf{Hyb}^{(j-1)}_0$ and if $b = 1$, algorithm $\mathcal{B}$ perfectly simulates distribution $\mathsf{Hyb}^{(j)}_0$. Thus, algorithm $\mathcal{B}$ breaks index hiding with advantage at least ε, and the claim holds. □

By construction, for all adversaries $\mathcal{A}$, $\mathsf{Hyb}^{(d)}_0(\mathcal{A}) \equiv \mathsf{Hyb}_1(\mathcal{A})$. Appealing to Claim 5.10 and the fact that $d = \mathsf{poly}(\lambda)$, the lemma follows by a hybrid argument. □

Lemma 5.11. *Suppose $i\mathcal{O}$ is secure. Then, for all non-uniform polynomial time adversaries $\mathcal{A}$, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for all* $\lambda \in \mathbb{N}$, $|\Pr[\mathsf{Hyb}_1(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_2(\mathcal{A}) = 1]| = \mathsf{negl}(\lambda)$.

Proof. We show that the programs $\mathsf{Prove}[K, \mathsf{hk}]$ and $\mathsf{Prove}'[K, \mathsf{hk}, C^*, \mathcal{X}, d_{\mathsf{thresh}}, \mathsf{ind}^{(i^*)}]$ have identical behavior and similarly for programs $\mathsf{Verify}[K]$ and $\mathsf{Verify}'[K, C^*, \mathcal{X}, d_{\mathsf{thresh}}, \mathsf{ind}^{(i^*)}]$:

- Consider the programs $\mathsf{Prove}[K, \mathsf{hk}]$ and $\mathsf{Prove}'[K, \mathsf{hk}, C^*, \mathcal{X}, d_{\mathsf{thresh}}, \mathsf{ind}^{(i^*)}]$. Consider an input of the form $(C, h_0, h_1, \mathsf{ind}, \pi_0, \pi_1)$:
  - Suppose $\mathsf{ind} \in \{0,1\}^d$ is a leaf node. By construction, the only inputs on which Prove and Prove′ can differ in this case are those where $\mathsf{ind} = \mathsf{ind}^{(i^*)}$. By definition of $\mathsf{HashProg}[\mathsf{hk}]$ (see Fig. 10), we have that $h_{\mathsf{ind}} = \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}^{(i^*)}) = x^*_{i^*}$. Then, this means Prove and Prove′ agree on all inputs unless $C = C^*$ and $h_0 = x^*_{i^*}$. On these inputs, Prove′ always outputs $\perp$. Consider the output of Prove. By assumption, there does not exist any input $\pi_0 \in \{0,1\}^*$ where $C^*(x^*_{i^*}, \pi_0) = 0$, so Prove on these inputs also outputs $\perp$.
  - Suppose $\mathsf{ind} \in \{0,1\}^{<d}$ is an internal node. In this case, $\mathsf{level}(\mathsf{ind}) = d - |\mathsf{ind}| > 0$. Since $d_{\mathsf{thresh}} = 0$ in Hyb_2, the condition $\mathsf{level}(\mathsf{ind}) < d$ is never satisfied, so the extra check introduced in Hyb_2 never triggers.

  We conclude that $\mathsf{Prove}[K, \mathsf{hk}]$ and $\mathsf{Prove}'[K, \mathsf{hk}, C^*, \mathcal{X}, d_{\mathsf{thresh}}, \mathsf{ind}^{(i^*)}]$ have identical input/output behavior.
- The programs $\mathsf{Verify}[K]$ and $\mathsf{Verify}'[K, C^*, \mathcal{X}, d_{\mathsf{thresh}}, \mathsf{ind}^{(i^*)}]$ have identical functionality. The only difference between these two programs is the extra check that Verify′ performs. By definition $\mathsf{level}(\mathsf{ind}) \geq 0 = d^*$, so the additional condition $\mathsf{level}(\mathsf{ind}) < d^*$ in Verify′ never triggers. Consequently, Verify and Verify′ has identical input/output behavior.

Since $\mathsf{Prove}[K, \mathsf{hk}]$ and $\mathsf{Prove}'[K, \mathsf{hk}, C^*, \mathcal{X}, d_{\mathsf{thresh}}, \mathsf{ind}^{(i^*)}]$ compute identical functions and likewise for $\mathsf{Verify}[K]$ and $\mathsf{Verify}'[K, C^*, \mathcal{X}, d_{\mathsf{thresh}}, \mathsf{ind}^{(i^*)}]$, indistinguishability now follows by $i\mathcal{O}$ security and a standard hybrid argument. □

Lemma 5.12. *Suppose* Π_{PRF} *is functionality-preserving and satisfies punctured pseudorandomness,* PRG *is a secure PRG,* Π_{SSB} *is somewhere statistically binding, and $i\mathcal{O}$ is secure. Then, for all $j \in [d]$ and all non-uniform polynomial time adversaries $\mathcal{A}$, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for all* $\lambda \in \mathbb{N}$,

$$|\Pr[\mathsf{Hyb}_{j+1}(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_{j+2}(\mathcal{A}) = 1]| = \mathsf{negl}(\lambda).$$

Proof. We begin by introducing a sequence of intermediate hybrids:

- $\mathsf{Hyb}^{(1)}_{j+1}$: Same as Hyb_{j+1} except the challenger changes the distribution of the CRS. Specifically, it starts by defining the programs $\mathsf{Prove}''[K_p, \mathsf{hk}, C^*, \mathcal{X}, d_{\mathsf{thresh}}, \mathsf{ind}^{(i^*)}, z]$ and $\mathsf{Verify}''[K_p, C^*, \mathcal{X}, d_{\mathsf{thresh}}, \mathsf{ind}^{(i^*)}, z]$ as follows:

Constants: Punctured PRF key K_p, hash key $\mathsf{hk} = (\mathsf{hk}_1, \ldots, \mathsf{hk}_d)$, Boolean circuit C^*, set $\mathcal{X}$, level d_{thresh}, index $\mathsf{ind}^{(i^*)}$, hard-coded value z
Input: Boolean circuit $C\colon \{0,1\}^\ell \times \{0,1\}^m \to \{0,1\}$ of size at most s, node values $h_0, h_1 \in \{0,1\}^{\leq \ell_{\max}}$, index $\mathsf{ind} \in \{0,1\}^{\leq d}$, proofs $\pi_0, \pi_1 \in \{0,1\}^{\max(m,\lambda)}$

1. If $\mathsf{ind} \in \{0,1\}^d$ (i.e., a leaf in the binary tree),
   (a) If $C = C^*$, ind is a prefix of $\mathsf{ind}^{(i^*)}$, and $(\mathsf{ind}, h_0) \in \mathcal{X}$, then output $\perp$.
   (b) Parse h_0 as a statement $x_1 \in \{0,1\}^\ell$. Parse π_0 as witness $w_1 \in \{0,1\}^m$.
   (c) If $C(x_1, w_1) \neq 1$, output $\perp$. Otherwise, output $\mathsf{PRF.Eval}(K_p, (C, x_1, \mathsf{ind}))$.
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 the hash $h \leftarrow \mathsf{SSB.LocalHash}(\mathsf{hk}_{d'}, h_0, h_1)$.
   (b) If $d' \leq d_{\mathsf{thresh}}$, $C = C^*$, ind is a prefix of $\mathsf{ind}^{(i^*)}$, and $(\mathsf{ind}, h) \in \mathcal{X}$, then output $\perp$.
   (c) Check the following conditions:
   - If $d' = d_{\mathsf{thresh}} + 1$, $C = C^*$, $\mathsf{ind}\|0$ is a prefix of $\mathsf{ind}^{(i^*)}$ and $(\mathsf{ind}\|0, h_0) \in \mathcal{X}$, check if $\mathsf{PRG}(\pi_0) = z$. Otherwise, check if $\mathsf{PRG}(\pi_0) = \mathsf{PRG}(\mathsf{PRF.Eval}(K_p, (C, h_0, \mathsf{ind}\|0)))$.
   - If $d' = d_{\mathsf{thresh}} + 1$, $C = C^*$, $\mathsf{ind}\|1$ is a prefix of $\mathsf{ind}^{(i^*)}$ and $(\mathsf{ind}\|1, h_1) \in \mathcal{X}$, check if $\mathsf{PRG}(\pi_1) = z$. Otherwise, check if $\mathsf{PRG}(\pi_1) = \mathsf{PRG}(\mathsf{PRF.Eval}(K_p, (C, h_1, \mathsf{ind}\|1)))$.

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

Figure 13: Program $\mathsf{Prove}''[K_p, \mathsf{hk}, C^*, \mathcal{X}, d_{\mathsf{thresh}}, \mathsf{ind}^{(i^*)}, z]$

Constants: Punctured PRF key K_p, circuit C^*, set $\mathcal{X}$, level d_{thresh}, index $\mathsf{ind}^{(i^*)}$, hard-coded value z
Input: Boolean circuit C of size at most s, node value $h \in \{0,1\}^{\leq \ell_{\max}}$, index $\mathsf{ind} \in \{0,1\}^{\leq d}$, proof $\pi \in \{0,1\}^\lambda$

1. Let $d' = \mathsf{level}(\mathsf{ind})$. If $d' < d_{\mathsf{thresh}}$, $C = C^*$, ind is a prefix of $\mathsf{ind}^{(i^*)}$, and $(\mathsf{ind}, h) \in \mathcal{X}$, then output 0.
2. If $d' = d_{\mathsf{thresh}}$, $C = C^*$, ind is a prefix of $\mathsf{ind}^{(i^*)}$ and $(\mathsf{ind}, h) \in \mathcal{X}$, then output 1 if $\mathsf{PRG}(\pi) = z$ and 0 otherwise.
3. Otherwise, output 1 if $\mathsf{PRG}(\pi) = \mathsf{PRG}(\mathsf{PRF.Eval}(K_p, (C, h, \mathsf{ind})))$ and 0 otherwise.

Figure 14: Program $\mathsf{Verify}''[K_p, C^*, \mathcal{X}, d_{\mathsf{thresh}}, \mathsf{ind}^{(i^*)}, z]$

The challenger constructs the CRS as follows:

- First, the challenger samples the hash keys $\mathsf{hk} = (\mathsf{hk}_1, \ldots, \mathsf{hk}_d)$ exactly as in Hyb_{j+1} (same as in Hyb_1 and Hyb_2). It also samples the PRF key $K \leftarrow \mathsf{PRF.KeyGen}(1^\lambda)$.
- Next, the challenger computes a binary tree $(\Gamma_{\mathsf{hash}}, \mathsf{val}_{\mathsf{hash}}) \leftarrow \mathsf{HashProg}[\mathsf{hk}](x_1^*, \ldots, x_t^*)$ using the algorithm from Fig. 10.
- Let $\mathsf{ind}^{(i^*)} = \mathsf{bin}_d(i^* - 1) = b_1 \cdots b_d$. For all $i \in [0, j]$, let $\mathsf{prefix}_{i^*}^{(i)} = b_1 \cdots b_{d-i}$ be the prefix of $\mathsf{ind}^{(i^*)}$ of length $d - i$. Let $\mathcal{X} = \{(\mathsf{prefix}_{i^*}^{(i)}, \mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}_{i^*}^{(i)}))\}_{i \in \{0,\ldots,d\}}$.
- Next, the challenger computes $K_p \leftarrow \mathsf{PRF.Puncture}(K, \{(C^*, \mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}_{i^*}^{(j-1)}), \mathsf{prefix}_{i^*}^{(j-1)})\})$ and the evaluation $z^* \leftarrow \mathsf{PRG}(\mathsf{PRF.Eval}(K, (C^*, \mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}_{i^*}^{(j-1)}), \mathsf{prefix}_{i^*}^{(j-1)})))$.
- It then constructs the obfuscated programs $\mathsf{ObfProve} \leftarrow i\mathcal{O}(1^\lambda, \mathsf{Prove}''[K_p, \mathsf{hk}, C^*, \mathcal{X}, j-1, \mathsf{ind}^{(i^*)}, z^*])$ and $\mathsf{ObfVerify} \leftarrow i\mathcal{O}(1^\lambda, \mathsf{Verify}''[K_p, C^*, \mathcal{X}, j-1, \mathsf{ind}^{(i^*)}, z^*])$. As in the real scheme, the challenger pads the size of the proving circuit $\mathsf{Prove}''[K_p, \mathsf{hk}, C^*, \mathcal{X}, j-1, \mathsf{ind}^{(i^*)}, z^*]$ and the verification circuit $\mathsf{Verify}''[K_p, C^*, \mathcal{X}, j-1, \mathsf{ind}^{(i^*)}, z^*]$ to the maximum size of any circuit that appear in the proof of Theorem 5.8.
- The challenger gives $\mathsf{crs} = (\mathsf{ObfProve}, \mathsf{ObfVerify}, \mathsf{hk})$ to $\mathcal{A}$.

The remainder of the experiment proceeds as in Hyb_{j+1}.

- $\mathsf{Hyb}^{(2)}_{j+1}$: Same as $\mathsf{Hyb}^{(1)}_{j+1}$ but when constructing the CRS, the challenger sets $z^* \leftarrow \mathsf{PRG}(y^*)$ where $y^* \xleftarrow{R} \{0,1\}^\lambda$.
- $\mathsf{Hyb}^{(3)}_{j+1}$: Same as $\mathsf{Hyb}^{(2)}_{j+1}$ but when constructing the CRS, the challenger samples $z^* \xleftarrow{R} \{0,1\}^{2\lambda}$.

We now argue that each pair of adjacent hybrid experiments are indistinguishable for all $j \in [d]$.

Claim 5.13. *Suppose Π_{PRF} is functionality-preserving and $i\mathcal{O}$ is secure. Then, for all $j \in [d]$ and all non-uniform polynomial time adversaries $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that for all $\lambda \in \mathbb{N}$,*

$$|\Pr[\mathsf{Hyb}^{(1)}_{j+1}(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_{j+1}(\mathcal{A}) = 1]| = \mathsf{negl}(\lambda).$$

Proof. Similar to the proof of Lemma 5.11, it suffices to show that the prover and verifier programs in Hyb_{j+1} and $\mathsf{Hyb}^{(1)}_{j+1}$ have identical input/output behavior. The main difference in $\mathsf{Hyb}^{(1)}_{j+1}$ is that we substitute a PRF key K_p punctured at the point $p = (C^*, \mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}^{(j-1)}_{i^*}), \mathsf{prefix}^{(j-1)}_{i^*})$ for the real PRF key K. Since the punctured PRF is functionality-preserving, on all inputs $(C, h, \mathsf{ind}) \neq p$,

$$\mathsf{PRF.Eval}(K, (C, h, \mathsf{ind})) = \mathsf{PRF.Eval}(K_p, (C, h, \mathsf{ind})).$$

In addition, in $\mathsf{Hyb}^{(1)}_{j+1}$, the challenger sets $z^* = \mathsf{PRG}(\mathsf{PRF.Eval}(K, p))$. We first argue that the proving programs $\mathsf{Prove}'[K, \mathsf{hk}, C^*, \mathcal{X}, j-1, \mathsf{ind}^{(i^*)}]$ and $\mathsf{Prove}''[K_p, \mathsf{hk}, C^*, \mathcal{X}, j-1, \mathsf{ind}^{(i^*)}, z^*]$ have identical input/output behavior, where $\mathcal{X} = \{(\mathsf{prefix}^{(i)}_{i^*}, \mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}^{(i)}_{i^*}))\}_{i \in \{0,\ldots,d\}}$ is the set defined in Hyb_{j+1} and $\mathsf{Hyb}^{(1)}_{j+1}$. Consider any input $(C, h_0, h_1, \mathsf{ind}, \pi_0, \pi_1)$ to the two programs. Let $d' = \mathsf{level}(\mathsf{ind})$.

- Suppose $d' = 0$ (i.e., ind is a leaf in the binary tree),
  - If $C = C^*$, $(\mathsf{ind}, h_0) \in \mathcal{X}$, then both programs output $\perp$.
  - Suppose that either $C \neq C^*$ or $h_0 \neq \mathsf{prefix}^{(0)}_{i^*}$. Then PRF is never evaluated at point p. Both Prove′ and Prove″ perform identical checks using keys K and K_p, respectively. The two programs' behavior are identical by the functionality-preserving property of Π_{PRF}.
- We analyze the cases when $d' > 0$, (ind is an internal node in the binary tree).
  - First, if $d' \neq j$ or $C \neq C^*$, then neither program needs to evaluate the PRF at the point
    $$p = (C^*, \mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}^{(j-1)}_{i^*}), \mathsf{prefix}^{(j-1)}_{i^*}).$$
  - Suppose $d' = j$, $C = C^*$, $\mathsf{ind}\|0$ is a prefix of $\mathsf{ind}^{(i^*)}$, and $(\mathsf{ind}\|0, h_0) \in \mathcal{X}$. Since $\mathsf{ind}\|0$ is a prefix of $\mathsf{ind}^{(i^*)}$ and $d' = j$, we have $\mathsf{ind}\|0 = \mathsf{prefix}^{(j-1)}_{i^*}$ and $(\mathsf{ind}\|0, \mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}^{(j-1)}_{i^*}))$ is the corresponding value stored in $\mathcal{X}$. In this case, Prove′ checks the condition $\mathsf{PRG}(\pi_0) = \mathsf{PRG}(\mathsf{PRF.Eval}(K, (C, h_0, \mathsf{ind}\|0)))$ while Prove″ checks the condition $\mathsf{PRG}(\pi_0) = z^*$, where as noted above,
    $$z^* = \mathsf{PRG}(\mathsf{PRF.Eval}(K, p)) = \mathsf{PRG}(\mathsf{PRF.Eval}(K, (C^*, \mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}^{(j-1)}_{i^*}), \mathsf{prefix}^{(j-1)}_{i^*}))).$$
    Thus, the two programs perform identical checks in this case.
  - Suppose $d' = j$, $C = C^*$, $\mathsf{ind}\|1$ is a prefix of $\mathsf{ind}^{(i^*)}$, and $(\mathsf{ind}\|1, h_1) \in \mathcal{X}$. Since $\mathsf{ind}\|1$ is a prefix of $\mathsf{ind}^{(i^*)}$ and $d' = j$, we have $\mathsf{ind}\|1 = \mathsf{prefix}^{(j-1)}_{i^*}$ and $(\mathsf{ind}\|1, \mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}^{(j-1)}_{i^*}))$ is the corresponding value stored in $\mathcal{X}$. In this case, Prove′ checks the condition $\mathsf{PRG}(\pi_1) = \mathsf{PRG}(\mathsf{PRF.Eval}(K, (C, h_1, \mathsf{ind}\|1)))$ while Prove″ checks the condition $\mathsf{PRG}(\pi_1) = z^*$. By the analogous logic as in the previous case, the behavior of these two checks is identical.
  - Suppose none of the above conditions hold. Then, we have the following:
    - Suppose $d' = j$, $C = C^*$, and $\mathsf{ind}\|0$ is not a prefix of $\mathsf{ind}^{(i^*)}$. Then $\mathsf{ind}\|0 \neq \mathsf{prefix}^{(j-1)}_{i^*}$ and PRF is never evaluated at point p.

* Suppose $d' = j$, $C = C^*$, and $\mathsf{ind}\|1$ is not a prefix of $\mathsf{ind}^{(i^*)}$. Then $\mathsf{ind}\|1 \neq \mathsf{prefix}_{i^*}^{(j-1)}$ and PRF is again never evaluated at point p.
* Suppose $d' = j$, $C = C^*$, $\mathsf{ind}\|0$ is prefix of $\mathsf{ind}^{(i^*)}$, and $(\mathsf{ind}\|0, h_0) \notin \mathcal{X}$. Then $\mathsf{ind}\|0 = \mathsf{prefix}_{i^*}^{(j-1)}$ and $h_0 \neq \mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}_{i^*}^{(j-1)})$. In this case, PRF is not evaluated at point p.
* Suppose $d' = j$, $C = C^*$, $\mathsf{ind}\|1$ is prefix of $\mathsf{ind}^{(i^*)}$, and $(\mathsf{ind}\|1, h_1) \notin \mathcal{X}$. Then $\mathsf{ind}\|1 = \mathsf{prefix}_{i^*}^{(j-1)}$ and $h_1 \neq \mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}_{i^*}^{(j-1)})$. As in the previous case, PRF is not evaluated at point p.

Since PRF is never evaluated at point p, both Prove′ and Prove″ perform identical checks using PRF keys K and K_p, respectively. Thus, the two programs' behavior are identical by the functionality-preserving property of Π_{PRF}.

Next consider the verification programs $\mathsf{Verify}'[K, C^*, \mathcal{X}, j-1, \mathsf{ind}^{(i^*)}]$ and $\mathsf{Verify}''[K_p, C^*, \mathcal{X}, j-1, \mathsf{ind}^{(i^*)}, z^*]$. Once again, the challenger sets $z^* = \mathsf{PRG}(\mathsf{PRF.Eval}(K, p))$.

- Suppose that $d' \neq j-1$ or $C \neq C^*$. Then, PRF is never evaluated at point p and Verify′ and Verify″ have identical behavior (since Π_{PRF} is functionality-preserving).
- Suppose $d' = j-1$, $C = C^*$, and ind is not a prefix of $\mathsf{ind}^{(i^*)}$. Then $\mathsf{ind} \neq \mathsf{prefix}_{i^*}^{(j-1)}$ and PRF is never evaluated at point p.
- Suppose $d' = j-1$, $C = C^*$, ind is prefix of $\mathsf{ind}^{(i^*)}$, and $(\mathsf{ind}, h) \notin \mathcal{X}$. By construction of $\mathcal{X}$, this means that $\mathsf{ind} = \mathsf{prefix}_{i^*}^{(j-1)}$ and $h \neq \mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}_{i^*}^{(j-1)})$. Once again, PRF is never evaluated at point p in this case.
- Suppose $d' = j-1$, $C = C^*$, ind is a prefix to $\mathsf{ind}^{(i^*)}$, and $(\mathsf{ind}, h) \in \mathcal{X}$. Since $d' = \mathsf{level}(\mathsf{ind}) = j-1$ and ind is a prefix to $\mathsf{ind}^{(i^*)}$, this means that $\mathsf{ind} = \mathsf{prefix}_{i^*}^{(j-1)}$. By construction of $\mathcal{X}$, this means that $h = \mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}_{i^*}^{(j-1)})$. In this case, Verify″ checks the condition $\mathsf{PRG}(\pi) = z^* = \mathsf{PRG}(\mathsf{PRF.Eval}(K, p))$, which is exactly the same check as in Verify′.

The claim now follows from $i\mathcal{O}$ security and a standard hybrid argument. □

Claim 5.14. *If Π_{PRF} satisfies punctured pseudorandomness, then for all $j \in [d]$ and all non-uniform polynomial time adversaries $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\lambda)$ such that for all $\lambda \in \mathbb{N}$,*

$$|\Pr[\mathsf{Hyb}_{j+1}^{(2)}(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_{j+1}^{(1)}(\mathcal{A}) = 1]| = \mathsf{negl}(\lambda).$$

Proof. Suppose there exists a non-uniform polynomial time adversary $\mathcal{A}$ (with advice string ρ_λ) where

$$|\Pr[\mathsf{Hyb}_{j+1}^{(2)}(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_{j+1}^{(1)}(\mathcal{A}) = 1]| \geq \varepsilon,$$

for some non-negligible ε. We use $\mathcal{A}$ to construct a non-uniform adversary $\mathcal{B}$ with advice string $(\rho, i^*) = (\rho_\lambda, i_\lambda^*)$ that breaks punctured pseudorandomness of Π_{PRF}:

1. Algorithm $\mathcal{B}$ runs adversary $\mathcal{A}$ on input 1^λ and with advice string ρ. Algorithm $\mathcal{A}$ outputs the maximum circuit size 1^s, a Boolean circuit C^* of size at most s, and statements $x_1, \ldots, x_t$ where $t \leq 2^\lambda$.
2. Algorithm $\mathcal{B}$ sets $\mathsf{ind}^{(i^*)} = \mathsf{bin}_d(i^* - 1) = b_1 \cdots b_d \in \{0,1\}^d$. For all $i \in [0, j]$, let $\mathsf{prefix}_{i^*}^{(i)} = b_1 \cdots b_{d-i}$. For each $j \in [d]$, algorithm $\mathcal{B}$ samples $\mathsf{hk}_j \leftarrow \mathsf{SSB.GenTD}(1^\lambda, 1^{\ell_{j-1}}, b_{d+1-j})$. Then, it computes a binary tree $(\Gamma_{\mathsf{hash}}, \mathsf{val}_{\mathsf{hash}}) \leftarrow \mathsf{HashProg}[\mathsf{hk}](x_1^*, \ldots, x_t^*)$ using the algorithm from Fig. 10.
3. Then, for all $i \in [0, j]$, let $\mathsf{prefix}_{i^*}^{(i)} = b_1 \cdots b_{d-i}$ be the prefix of $\mathsf{ind}^{(i^*)}$ of length $d - i$. Algorithm $\mathcal{B}$ defines the set $\mathcal{X} = \{(\mathsf{prefix}_{i^*}^{(i)}, \mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}_{i^*}^{(i)}))\}_{i \in \{0,\ldots,d\}}$.
4. Algorithm $\mathcal{B}$ chooses $p = (C^*, \mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}_{i^*}^{(j-1)}), \mathsf{prefix}_{i^*}^{(j-1)})$ as its challenge point. It receives from the challenger a punctured key K_p and a challenge $y \in \{0,1\}^\lambda$.

5. Algorithm $\mathcal{B}$ computes $z^* \leftarrow \mathsf{PRG}(y)$, $\mathsf{ObfProve} \leftarrow i\mathcal{O}(1^\lambda, \mathsf{Prove}''[K_p, \mathsf{hk}, C^*, \mathcal{X}, j-1, \mathsf{ind}^{(i^*)}, z^*])$, and $\mathsf{ObfVerify} \leftarrow i\mathcal{O}(1^\lambda, \mathsf{Verify}''[K_p, C^*, \mathcal{X}, j-1, \mathsf{ind}^{(i^*)}, z^*])$. Finally, it sets the $\mathsf{crs} = (\mathsf{ObfProve}, \mathsf{ObfVerify}, \mathsf{hk})$ and gives crs to $\mathcal{A}$.

6. At the end of the game, algorithm $\mathcal{A}$ outputs a proof π and algorithm $\mathcal{B}$ outputs $\mathsf{Verify}(\mathsf{crs}, C^*, (x_1^*, \ldots, x_t^*), \pi)$.

By construction, the challenger samples $K \leftarrow \mathsf{PRF.KeyGen}(1^\lambda)$ and constructs the punctured key as $K_p \leftarrow \mathsf{PRF.Puncture}(K, (C^*, \mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}_{i^*}^{(j-1)}), \mathsf{prefix}_{i^*}^{(j-1)}))$. This coincides with the specification in $\mathsf{Hyb}_{j+1}^{(1)}$ and $\mathsf{Hyb}_{j+1}^{(2)}$. Consider now the distribution of the challenge y:

- Suppose $y = \mathsf{PRF.Eval}(K, (C^*, \mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}_{i^*}^{(j-1)}), \mathsf{prefix}_{i^*}^{(j-1)}))$. Then algorithm $\mathcal{A}$ perfectly simulates distribution $\mathsf{Hyb}_{j+1}^{(1)}$.
- Suppose $y \xleftarrow{R} \{0,1\}^\lambda$. Then algorithm $\mathcal{A}$ perfectly simulates distribution $\mathsf{Hyb}_{j+1}^{(2)}$.

Algorithm $\mathcal{B}$ breaks punctured pseudorandomness with the same advantage ε and the claim follows. □

Claim 5.15. *If* PRG *is secure, then for all $j \in [d]$ and all non-uniform polynomial time adversaries $\mathcal{A}$, there exists a negligible function* $\mathsf{negl}(\lambda)$ *such that for all $\lambda \in \mathbb{N}$, $|\Pr[\mathsf{Hyb}_{j+1}^{(3)}(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_{j+1}^{(2)}(\mathcal{A}) = 1]| = \mathsf{negl}(\lambda)$.*

Proof. Suppose there exists a non-uniform polynomial time adversary $\mathcal{A}$ (with advice string ρ_λ) where

$$|\Pr[\mathsf{Hyb}_{j+1}^{(3)}(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_{j+1}^{(2)}(\mathcal{A}) = 1]| = \varepsilon(\lambda),$$

for some non-negligible ε. We use $\mathcal{A}$ to construct a non-uniform adversary $\mathcal{B}$ with advice string $(\rho, i^*) = (\rho_\lambda, i_\lambda^*)$ that breaks PRG security:

1. Algorithm $\mathcal{B}$ runs adversary $\mathcal{A}$ on input 1^λ and advice string ρ. Algorithm $\mathcal{A}$ outputs the maximum circuit size 1^s, a Boolean circuit C^* of size at most s, and statements $x_1, \ldots, x_t$ where $t \le 2^\lambda$.
2. Algorithm $\mathcal{B}$ sets $\mathsf{ind}^{(i^*)} = \mathsf{bin}_d(i^* - 1) = b_1 \cdots b_d \in \{0,1\}^d$. For all $i \in [0, j]$, let $\mathsf{prefix}_{i^*}^{(i)} = b_1 \cdots b_{d-i}$. For each $j \in [d]$, algorithm $\mathcal{B}$ samples $\mathsf{hk}_j \leftarrow \mathsf{SSB.GenTD}(1^\lambda, 1^{\ell_{j-1}}, b_{d+1-j})$. Then, it computes a binary tree $(\Gamma_{\mathsf{hash}}, \mathsf{val}_{\mathsf{hash}}) \leftarrow \mathsf{HashProg}[\mathsf{hk}](x_1^*, \ldots, x_t^*)$ using the algorithm from Fig. 10.
3. Then, for all $i \in [0, j]$, let $\mathsf{prefix}_{i^*}^{(i)} = b_1 \cdots b_{d-i}$ be the prefix of $\mathsf{ind}^{(i^*)}$ of length $d - i$. Algorithm $\mathcal{B}$ defines the set $\mathcal{X} = \{(\mathsf{prefix}_{i^*}^{(i)}, \mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}_{i^*}^{(i)}))\}_{i \in \{0,\ldots,d\}}$.
4. Algorithm $\mathcal{B}$ receives a challenge $z^* \in \{0,1\}^{2\lambda}$ from the PRG challenger.
5. Algorithm $\mathcal{B}$ samples $K \leftarrow \mathsf{PRF.KeyGen}(1^\lambda)$ and constructs the punctured key $K_p \leftarrow \mathsf{PRF.Puncture}(K, p)$, where $p = (C^*, \mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}_{i^*}^{(j-1)}), \mathsf{prefix}_{i^*}^{(j-1)})$.
6. Next, it computes the obfuscated programs $\mathsf{ObfProve} \leftarrow i\mathcal{O}(1^\lambda, \mathsf{Prove}''[K_p, \mathsf{hk}, C^*, \mathcal{X}, j-1, \mathsf{ind}^{(i^*)}, z^*])$ and $\mathsf{ObfVerify} \leftarrow i\mathcal{O}(1^\lambda, \mathsf{Verify}''[K_p, C^*, \mathcal{X}, j-1, \mathsf{ind}^{(i^*)}, z^*])$. Algorithm $\mathcal{B}$ gives $\mathsf{crs} = (\mathsf{ObfProve}, \mathsf{ObfVerify}, \mathsf{hk})$ to $\mathcal{A}$.
7. Algorithm $\mathcal{A}$ outputs a proof π and algorithm $\mathcal{B}$ outputs $\mathsf{Verify}(\mathsf{crs}, C^*, (x_1^*, \ldots, x_t^*), \pi)$.

If $z^* \leftarrow \mathsf{PRG}(y^*)$ where $y^* \xleftarrow{R} \{0,1\}^\lambda$, then algorithm $\mathcal{B}$ perfectly simulates $\mathsf{Hyb}_{j+1}^{(2)}$ for $\mathcal{A}$. Alternatively, if $z^* \xleftarrow{R} \{0,1\}^{2\lambda}$, then algorithm $\mathcal{B}$ perfectly simulates $\mathsf{Hyb}_{j+1}^{(3)}$ for $\mathcal{A}$. The claim follows. □

Claim 5.16. *If Π_{PRF} is functionality-preserving, Π_{SSB} is somewhere statistically binding, and $i\mathcal{O}$ is secure, then for all $j \in [d]$ and all non-uniform polynomial time adversaries $\mathcal{A}$, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for all $\lambda \in \mathbb{N}$, $|\Pr[\mathsf{Hyb}_{j+2}(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_{j+1}^{(3)}(\mathcal{A}) = 1]| = \mathsf{negl}(\lambda)$.*

Proof. We first show that with overwhelming probability over the choice of $\mathsf{hk}_j \leftarrow \mathsf{SSB.GenTD}(1^\lambda, 1^{\ell_{j-1}}, b_{d+1-j})$ and $z^* \xleftarrow{R} \{0,1\}^{2\lambda}$, the programs $\mathsf{Prove}''[K_p, \mathsf{hk}, C^*, \mathcal{X}, j-1, \mathsf{ind}^{(i^*)}, z^*]$ and $\mathsf{Verify}''[K_p, C^*, \mathcal{X}, j-1, \mathsf{ind}^{(i^*)}, z^*]$ in $\mathsf{Hyb}^{(3)}_{j+1}$ have the same input/output behavior as the programs $\mathsf{Prove}'[K, \mathsf{hk}, C^*, \mathcal{X}, j, \mathsf{ind}^{(i^*)}]$ and $\mathsf{Verify}'[K, C^*, \mathcal{X}, j, \mathsf{ind}^{(i^*)}]$ in Hyb_{j+2}, where the set $\mathcal{X}$ is defined according to the specification of $\mathsf{Hyb}^{(3)}_{j+1}$ and Hyb_{j+2}. To see this, we start by analyzing the main quantities used to construct these programs:

- First, $z^* \xleftarrow{R} \{0,1\}^{2\lambda}$. Thus, $\Pr[\exists y \in \{0,1\}^\lambda : \mathsf{PRG}(y) = z^*] = 2^{-\lambda}$, so with overwhelming probability, the value z^* in $\mathsf{Hyb}^{(3)}_{j+1}$ is not in the range of PRG.
- Next, we note that hk_j is sampled in trapdoor mode to be binding on index b_{d+1-j}. We consider two possibilities:
  - Suppose $b_{d+1-j} = 0$. By construction of Γ_{hash} (see Fig. 10),

    $$\mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}^{(j)}_{i^*}) = \mathsf{SSB.LocalHash}(\mathsf{hk}_j, \mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}^{(j)}_{i^*} \| 0), \mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}^{(j)}_{i^*} \| 1)).$$

    For any h_0, h_1 if $\mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}^{(j)}_{i^*}) = \mathsf{SSB.LocalHash}(\mathsf{hk}_j, h_0, h_1)$, since $\mathsf{hk}_j \leftarrow \mathsf{SSB.GenTD}(1^\lambda, 1^{\ell_{j-1}}, 0)$ and Π_{SSB} is somewhere statistically binding; with overwhelming probability over the choice of hk_j, we have that h_0 must be equal to $\mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}^{(j)}_{i^*} \| 0)$. Since, $b_{d+1-j} = 0$, we have, $\mathsf{prefix}^{(j)}_{i^*} \| 0 = \mathsf{prefix}^{(j-1)}_{i^*}$. Thus, h_0 must be equal to $\mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}^{(j-1)}_{i^*})$.
  - Suppose $b_{d+1-j} = 1$. By construction of Γ_{hash} (see Fig. 10),

    $$\mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}^{(j)}_{i^*}) = \mathsf{SSB.LocalHash}(\mathsf{hk}_j, \mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}^{(j)}_{i^*} \| 0), \mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}^{(j)}_{i^*} \| 1)).$$

    For any h_0, h_1 if $\mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}^{(j)}_{i^*}) = \mathsf{SSB.LocalHash}(\mathsf{hk}_j, h_0, h_1)$, since $\mathsf{hk}_j \leftarrow \mathsf{SSB.GenTD}(1^\lambda, 1^{\ell_{j-1}}, 1)$ and Π_{SSB} is somewhere statistically binding; with overwhelming probability over the choice of hk_j, we have that h_1 must be equal to $\mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}^{(j)}_{i^*} \| 1)$. Since, $b_{d+1-j} = 1$, we have, $\mathsf{prefix}^{(j)}_{i^*} \| 1 = \mathsf{prefix}^{(j-1)}_{i^*}$. Thus, h_1 must be equal to $\mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}^{(j-1)}_{i^*})$.
- Finally, since Π_{PRF} is functionality-preserving, we have that $\mathsf{PRF.Eval}(K, (C, h, \mathsf{ind})) = \mathsf{PRF.Eval}(K_p, (C, h, \mathsf{ind}))$ whenever $(C, h, \mathsf{ind}) \neq (C^*, \mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}^{(j-1)}_{i^*}), \mathsf{prefix}^{(j-1)}_{i^*})$.

Now, consider the programs $\mathsf{Prove}''[K_p, \mathsf{hk}, C^*, \mathcal{X}, j-1, \mathsf{ind}^{(i^*)}, z^*]$ and $\mathsf{Prove}'[K, \mathsf{hk}, C^*, \mathcal{X}, j, \mathsf{ind}^{(i^*)}]$:

- Suppose $b_{d+1-j} = 0$. In this case, the behavior of the two programs only differs on inputs $(C, h_0, h_1, \mathsf{ind}, \pi_0, \pi_1)$ where $d' = \mathsf{level}(\mathsf{ind}) = j > 0$ (non-leaf node), $C = C^*$, $\mathsf{ind} = \mathsf{prefix}^{(j)}_{i^*}$, and $h = \mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}^{(j)}_{i^*})$, where the hash h is computed as $h \leftarrow \mathsf{SSB.LocalHash}(\mathsf{hk}_j, h_0, h_1, \mathsf{prefix}^{(j)}_{i^*})$.
  - On such an input, $\mathsf{Prove}'[K, \mathsf{hk}, C^*, \mathcal{X}, j, \mathsf{ind}^{(j)}]$ always outputs $\perp$.
  - Consider the output of $\mathsf{Prove}''[K, \mathsf{hk}, C^*, \mathcal{X}, j-1, \mathsf{ind}^{(i^*)}, z^*]$. Since $b_{d+1-j} = 0$, by the above analysis, with overwhelming probability over the choice of hk_j, if $\mathsf{SSB.LocalHash}(\mathsf{hk}_j, h_0, h_1, \mathsf{prefix}^{(j)}_{i^*}) = \mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}^{(j)}_{i^*})$, then $h_0 = \mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}^{(j-1)}_{i^*})$. In this case, Prove'' checks whether $\mathsf{PRG}(\pi_0) = z^*$ and outputs $\perp$ if not. As argued above, with overwhelming probability over the choice of z^*, there does not exist any π_0 such that $\mathsf{PRG}(\pi_0) = z^*$. Thus, with overwhelming probability over the choice of hk_j and z^*, the output of Prove'' on all such inputs is $\perp$.

  On all other inputs, the programs' behavior is identical as long as the PRF is functionality-preserving (since the punctured key in Prove'' is *never* used to evaluate on the punctured point).
- Suppose $b_{d+1-j} = 1$. In this case, the behavior of the two programs only differs on inputs $(C, h_0, h_1, \mathsf{ind}, \pi_0, \pi_1)$ where $d' = \mathsf{level}(\mathsf{ind}) = j > 0$ (non-leaf node), $C = C^*$, $\mathsf{ind} = \mathsf{prefix}^{(j)}_{i^*}$, and $h = \mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}^{(j)}_{i^*})$, where the hash h is computed as $h \leftarrow \mathsf{SSB.LocalHash}(\mathsf{hk}_j, h_0, h_1, \mathsf{prefix}^{(j)}_{i^*})$.

– On such an input, $\mathsf{Prove}'[K, \mathsf{hk}, C^*, \mathcal{X}, j, \mathsf{ind}^{(j)}]$ always outputs $\perp$.
– Consider the output of $\mathsf{Prove}''[K, \mathsf{hk}, C^*, \mathcal{X}, j-1, \mathsf{ind}^{(i^*)}, z^*]$. Since $b_{d+1-j} = 1$, by the above analysis, with overwhelming probability over the choice of hk_j, if $\mathsf{SSB.LocalHash}(\mathsf{hk}_j, h_0, h_1, \mathsf{prefix}_{i^*}^{(j)}) = \mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}_{i^*}^{(j)})$, then $h_1 = \mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}_{i^*}^{(j-1)})$. In this case, Prove'' checks whether $\mathsf{PRG}(\pi_1) = z^*$ and outputs $\perp$ if not. As argued above, with overwhelming probability over the choice of z^*, there does not exist any π_1 such that $\mathsf{PRG}(\pi_1) = z^*$. Thus, with overwhelming probability over the choice of hk_j and z^*, the output of Prove'' on all such inputs is $\perp$.

On all other inputs, the programs' behavior is identical as long as the PRF is functionality-preserving (since the punctured key in Prove'' is *never* used to evaluate on the punctured point).

Consider now the verification programs $\mathsf{Verify}'[K, C^*, \mathcal{X}, j, \mathsf{ind}^{(i^*)}]$ and $\mathsf{Verify}''[K_p, C^*, \mathcal{X}, j-1, \mathsf{ind}^{(i^*)}, z^*]$. By design, the *only* inputs on which the programs can differ are those of the form $(C, h, \mathsf{ind}, \pi)$ where $C = C^*$, $\mathsf{ind} = \mathsf{prefix}_{i^*}^{(j-1)}$, and $h = \mathsf{val}_{\mathsf{hash}}(\mathsf{prefix}_{i^*}^{(j-1)}) = \mathsf{val}_{\mathsf{hash}}(\mathsf{ind})$.

- On such an input, $\mathsf{Verify}'[K, \mathsf{hk}, C^*, \mathcal{X}, d_{\mathsf{thresh}}, \mathsf{ind}^{(i^*)}]$ always outputs 0 since $d' = \mathsf{level}(\mathsf{ind}) = j - 1 < j$.
- Consider the output of $\mathsf{Verify}''[K_p, C^*, \mathcal{X}, d_{\mathsf{thresh}}, \mathsf{ind}^{(i^*)}, z^*]$. By construction, the output is 1 if $\mathsf{PRG}(\pi) = z^*$ and 0 otherwise. As argued above, with overwhelming probability over the choice of z^*, there does not exist π such that $\mathsf{PRG}(\pi) = z^*$, and so the output of Verify'' on all such inputs is 0 with overwhelming probability.

By the above analysis, we see that with overwhelming probability over the choice of hk_j and z^*, the programs Prove' in Hyb_{j+2} and Prove'' in $\mathsf{Hyb}_{j+1}^{(3)}$ as well as the programs Verify' in Hyb_{j+2} and Verify'' in $\mathsf{Hyb}_{j+1}^{(3)}$ have identical input/output behavior. The claim now follows by $i\mathcal{O}$ security. □

Combining Claims 5.13 to 5.16, we have that for all $j \in [d]$, hybrids Hyb_{j+1} and Hyb_{j+2} are computationally indistinguishable and the lemma follows. □

To complete the proof of Theorem 5.8, we show that for all adversaries $\mathcal{A}$, $\Pr[\mathsf{Hyb}_{d+2}(\mathcal{A}) = 1] = 0$.

Lemma 5.17. *For all non-uniform polynomial time adversaries $\mathcal{A}$ and all $\lambda \in \mathbb{N}$, $\Pr[\mathsf{Hyb}_{d+2}(\mathcal{A}) = 1] = 0$.*

Proof. Take any adversary $\mathcal{A}$. Let crs be the common reference string sampled according to the specification of Hyb_{d+2} and let π be the proof that $\mathcal{A}$ outputs for the statement $(C^*, (x_1^*, \ldots, x_t^*))$ in Hyb_{d+2}. We consider the probability that the output of $\mathsf{Hyb}_{d+2}(\mathcal{A}) = 1$. By definition, the output in Hyb_{d+2} is 1 only if the adversary outputs a circuit C^* and instances $x_1^*, \ldots, x_t^*$ where $x_{i^*}^*$ is a false instance. Consider $\mathsf{V}(\mathsf{crs}, C^*, (x_1^*, \ldots, x_t^*), \pi)$:

- First, the verification algorithm constructs a binary tree $(\Gamma_{\mathsf{hash}}, \mathsf{val}_{\mathsf{hash}}) \leftarrow \mathsf{HashProg}[\mathsf{hk}](x_1^*, \ldots, x_t^*)$ using the algorithm from Fig. 10 (when running Hash).
- The verification algorithm parses the proof π as $\pi = \{(\mathsf{ind}, \pi_{\mathsf{ind}})\}_{\mathsf{ind} \in \mathsf{frontier}(\mathsf{ind}^{(t)})}$, and outputs 0 if the proof does not have this format. Then, for each $\mathsf{ind} \in \mathsf{frontier}(\mathsf{ind}^{(t)})$, the verification algorithm checks that $\mathsf{ObfVerify}(C^*, \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}), \mathsf{ind}, \pi_{\mathsf{ind}}) = 1$ and rejects with output 0 if any check fails.

Let ind' be the longest common prefix of $\mathsf{ind}^{(i^*)}$ and $\mathsf{ind}^{(t)}$. We now define an index $\mathsf{ind}'' \in \mathsf{frontier}(\mathsf{ind}^{(t)})$ where ind'' is a prefix of $\mathsf{ind}^{(i^*)}$ as follows:

- Suppose $\mathsf{ind}' = \mathsf{ind}^{(i^*)} = \mathsf{ind}^{(t)}$. Then, define $\mathsf{ind}'' = \mathsf{ind}'$. By definition, $\mathsf{ind}'' \in \mathsf{frontier}(\mathsf{ind}^{(t)})$ and is a prefix of $\mathsf{ind}^{(i^*)}$.
- Suppose $\mathsf{ind}' \neq \mathsf{ind}^{(i^*)}$. Let $|\mathsf{ind}'| = j$ where $0 \leq j \leq d-1$. Note that $j \neq d$ since $\mathsf{ind}^{(i^*)} \neq \mathsf{ind}^{(t)}$. Since ind' is defined to be the the longest common prefix, the $(j+1)^{\text{th}}$ bit of $\mathsf{ind}^{(i^*)}$ and $\mathsf{ind}^{(t)}$ must be different. Additionally, $\mathsf{ind}^{(t)} > \mathsf{ind}^{(i^*)}$ and moreover, the $(j+1)^{\text{th}}$ is the first differing bit between $\mathsf{ind}^{(t)}$ and $\mathsf{ind}^{(i^*)}$. This means that the $j+1^{\text{th}}$ bit of $\mathsf{ind}^{(i^*)}$ must be 0 and the $(j+1)^{th}$ bit of $\mathsf{ind}^{(t)}$ must be 1. In this case then, let $\mathsf{ind}'' = \mathsf{ind}' \| 0$. By construction, ind'' is a prefix of $\mathsf{ind}^{(i^*)}$, and moreover is a left sibling of a node on the path to $\mathsf{ind}^{(t)}$. This means that $\mathsf{ind}'' \in \mathsf{frontier}(\mathsf{ind}^{(t)})$, as required.

Consider now the output of $\mathsf{ObfVerify}(C^*, \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}''), \mathsf{ind}'', \pi_{\mathsf{ind}''})$. By correctness of $i\mathcal{O}$, this corresponds to the output of program $\mathsf{Verify}'[K, C^*, \mathcal{X}, d, \mathsf{ind}^{(i^*)}](C^*, \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}''), \mathsf{ind}'', \pi_{\mathsf{ind}''})$:

- By construction, ind'' is not the root node so $\mathsf{level}(\mathsf{ind}'') < d$.
- Again by construction, ind'' is a prefix of $\mathsf{ind}^{(i^*)}$, and moreover, $(\mathsf{ind}'', \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}'')) \in \mathcal{X}$. Next, the hash tree Γ_{hash} is computed in an identical fashion in both the verification algorithm V and in the construction of the obfuscated program ObfVerify. The output of Verify'' on the input $(C^*, \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}''), \mathsf{ind}'', \pi_{\mathsf{ind}''})$ is always 0, irrespective of the value of $\pi_{\mathsf{ind}''}$.

We conclude there always exists an index $\mathsf{ind}'' \in \mathsf{frontier}(\mathsf{ind}^{(t)})$ such that $\mathsf{ObfVerify}(C^*, \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}), \mathsf{ind}, \pi_{\mathsf{ind}}) = 0$. Thus, the output of the verification algorithm is always 0 in Hyb_{d+2} and the claim holds. □

Non-adaptive soundness of Construction 5.6 now follows by Lemmas 5.9, 5.11, 5.12 and 5.17. □

Theorem 5.18 (Succinctness). *If Π_{SSB} is succinct, then Construction 5.6 is fully succinct.*

Proof. Recall that ℓ is the length of the statement. We start by showing that $\ell_{\max} = \mathsf{poly}(\lambda, \ell)$. Since Π_{SSB} is succinct, there exists some polynomial $q = q(\lambda)$ such that the output length ℓ_{out} of the hash function satisfies $\ell_{\mathsf{out}}(\lambda, \ell_{\mathsf{blk}}) = \ell_{\mathsf{blk}} \cdot (1 + 1/\Omega(\lambda)) + q(\lambda)$. Next, by definition, $\ell_0 = \ell$ and for $j \in [d]$, we define $\ell_j = \ell_{\mathsf{out}}(\lambda, \ell_{j-1})$. Thus, $\ell_1 = \ell \cdot (1 + 1/\Omega(\lambda)) + q(\lambda)$, and more generally, we have that

$$\ell_d = \ell \cdot (1 + 1/\Omega(\lambda))^d + q(\lambda) \cdot \sum_{j \in [d]} (1 + 1/\Omega(\lambda))^{j-1}.$$

Since $d = \lambda$ and $(1 + 1/\Omega(\lambda))^\lambda = O(1)$, we have that $\ell_d = O(\ell) + q(\lambda) \cdot O(\lambda)$. Thus, $\ell_{\max} = \mathsf{poly}(\lambda, \ell)$.

- **Succinct proof size:** The value stored at a node in the tree Γ_{proof} is the output of the PRF, whose codomain consists of bitstrings of length λ. From Claim 5.5, for any polynomial t, $\mathsf{frontier}(\mathsf{ind}^{(t)})$ consists of at most $d + 1$ nodes so the total proof size is at most $(d + 1) \cdot (d + \lambda)$. Since $d = \lambda$, the size of the proof is $O(\lambda^2)$. Thus, the BARG is fully succinct.[12]
- **Succinct verification time:** For all $\lambda \in \mathbb{N}$, instance numbers $t \le 2^\lambda$, indices $i \in [t]$, size parameters $s \in \mathbb{N}$, and statement lengths $\ell \in \mathbb{N}$, the verification algorithm on input $(\mathsf{crs}, C, (x_1, \ldots, x_i), \pi_i)$ starts by computing the hash tree Γ_{hash} (using the algorithm from Fig. 10). This requires times $\mathsf{poly}(\lambda, i, \ell)$ time. Next, the verification algorithm parses the proof π_i as $\pi_i = \{(\mathsf{ind}, \pi_{\mathsf{ind}})\}_{\mathsf{ind} \in \mathsf{frontier}(\mathsf{ind}^{(i)})}$. Since $i\mathcal{O}$ is efficient, it takes $\mathsf{poly}(\lambda, s)$ time to run the program ObfVerify on each proof π_{ind}. From Claim 5.5, $|\mathsf{frontier}(\mathsf{ind}^{(i)})| \le d + 1 = \lambda + 1$. Thus, the overall verification cost is $\mathsf{poly}(\lambda, i, \ell) + \mathsf{poly}(\lambda, s)$, which is succinct, as required. □

Theorem 5.19 (Zero Knowledge). *Construction 5.6 satisfies perfect zero-knowledge.*

Proof. To show zero knowledge, we construct an efficient simulator $\mathcal{S}$ as follows. On input the security parameter λ, the bound s on the circuit size, a Boolean circuit $C\colon \{0,1\}^\ell \times \{0,1\}^m \to \{0,1\}$ of size at most s, the set of statements $x_1, \ldots, x_t$ such that $(C, (x_1, \ldots, x_t)) \in \mathcal{L}_{\mathsf{BatchCSAT},t}$, the simulator algorithm proceeds as follows:

1. Compute $\mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda, 1^\ell, 1^s)$. Let hk be the hash key and K be the PRF key sampled in the construction of CRS.
2. Compute the binary tree $(\Gamma_{\mathsf{hash}}, \mathsf{val}_{\mathsf{hash}}) \leftarrow \mathsf{HashProg}[\mathsf{hk}](x_1, \ldots, x_t)$ using the algorithm from Fig. 10.
3. Let $\mathsf{ind}^{(t)} = \mathsf{bin}_d(t - 1)$. Then, construct the proof

$$\pi_t = \{(\mathsf{ind}, \mathsf{PRF.Eval}\,(K, (C, \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}), \mathsf{ind})))\}_{\mathsf{ind} \in \mathsf{frontier}(\mathsf{ind}^{(t)})}.$$

[12] Note that we can apply a tighter analysis to show that the proof size on t instances is $O(\lambda \log t)$. The analysis described here implicitly is for an arbitrary $t \le 2^\lambda$.

4. Output (crs, π_t).

By construction, the simulator samples crs exactly as in the real scheme. If suffices to show that the simulated proofs are distributed exactly as in the real distribution. Let $x_1, \ldots, x_t$ be a sequence of statements and $w_1, \ldots, w_t$ be a corresponding set of witnesses (for all $i \in [t]$, $C(x_i, w_i) = 1$). Let $\pi_0 = \bot$, $h_0 = (0, \varnothing)$, and for every $i \in [t]$, $h_i \leftarrow \mathsf{Hash}(\mathsf{crs}, (x_1, \ldots, x_i))$. We compute the proofs iteratively (i.e., for $i \in [t]$, $\pi_i \leftarrow \mathsf{UpdateP}(\mathsf{crs}, C, h_{i-1}, \pi_{i-1}, x_i, w_i)$). Let $\mathsf{ind}^{(i)} = \mathsf{bin}_d(i-1)$, $h_i = \left(i, \{(\mathsf{ind}, \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}))\}_{\mathsf{ind} \in \mathsf{frontier}(\mathsf{ind}^{(i)})}\right)$ and $\pi_i = \{(\mathsf{ind}, \pi_{\mathsf{ind}})\}_{\mathsf{ind} \in \mathsf{frontier}(\mathsf{ind}^{(i)})}$. We show that the following invariant in our proof computation: for all indices $\mathsf{ind} \in \mathsf{frontier}(\mathsf{ind}^{(i)})$,

$$\pi_{\mathsf{ind}} = \mathsf{PRF.Eval}(K, (C, \mathsf{val}_{\mathsf{hash}}(\mathsf{ind}), \mathsf{ind})). \tag{5.6}$$

We now show that this invariant holds:

- In the base case ($i = 1$), UpdateP computes $\pi \leftarrow \mathsf{ObfProve}(C, x_1, \bot, \mathsf{ind}^{(1)}, w_1, \bot)$. By correctness of $i\mathcal{O}$ and the fact that $C(x_1, w_1) = 1$, we have $\pi = \mathsf{PRF.Eval}\left(K, (C, x_1, \mathsf{ind}^{(1)})\right)$ and the invariant holds.
- By the exact same analysis as in the proof of Theorem 5.7, we can show that Eq. (5.6) holds.

We conclude that the proofs output by the simulator are distributed identically to the proofs output in the real scheme, so the scheme satisfies perfect zero knowledge. □

Combining Theorems 5.7, 5.8, 5.18 and 5.19, we obtain the following corollary:

Corollary 5.20 (Non-Adaptive Updatable BARGs)**.** *Assuming the existence of a secure indistinguishability obfuscation scheme (for Boolean circuits) and somewhere statistically binding hash functions, there exists an updatable batch argument for* NP *satisfying non-adaptive soundness.*

Remark 5.21 (Non-Adaptive Updatable BARGs from $i\mathcal{O}$ and One-Way Functions)**.** For our non-adaptive construction, we can replace the two-to-one somewhere statistically binding hash functions with positional accumulators to obtain an updatable BARG scheme based only on $i\mathcal{O}$ and one-way functions.

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.

[BB04] Dan Boneh and Xavier Boyen. Efficient selective-id secure identity-based encryption without random oracles. In *EUROCRYPT*, pages 223–238, 2004.

[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.

A Adaptive BARGs for Index Languages

In this section, we show how to construct adaptively-sound batch arguments for index languages that support an unbounded number of statements. The construction closely follow our non-adaptively-sound BARG from Section 3, but now relies on complexity leveraging [BB04] to argue adaptive soundness. In particular, the reduction algorithm will guess the challenge circuit upfront; this incurs a loss of 2^{-s} in the reduction's success probability, where s is a bound on the circuit size (i.e., the description length of the Boolean circuit). Moreover, the reduction will also need to decide the underlying circuit-SAT relation to identify the index of the false instance across the $t = t(\lambda)$ instances, Thus, we need to additionally rely on hardness against super-polynomial time adversaries. We provide the formal details below.

Construction A.1 (Adaptively-Sound BARG for Index Languages)**.** Let λ be a security parameter. We construct a BARG scheme that supports index languages with up to $T = 2^\lambda$ instances (i.e., which suffices to support an arbitrary a priori unbounded 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 $\{0,1\}^\lambda$ interchangeably in the following description. Our construction relies on the following primitives, which will be instantiated with different security parameters $\lambda_1 = \lambda_1(\lambda, s)$, $\lambda_2 = \lambda_2(\lambda, s)$, and $\lambda_3 = \lambda_3(\lambda, s)$. We set these parameters to satisfy the requirements of Theorem A.3.

- Let PRF be a puncturable PRF with key space $\{0,1\}^{\lambda_1}$, domain $\{0,1\}^s \times \{0,1\}^\lambda$ and range $\{0,1\}^{\lambda_2}$.
- Let $i\mathcal{O}$ be an indistinguishability obfuscator.
- Let PRG be a pseudorandom generator with domain $\{0,1\}^{\lambda_2}$ and range $\{0,1\}^{2\lambda_2}$.

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

- $\mathsf{Gen}(1^\lambda, 1^s)$: This is essentially the same as in Construction 5.6, except instantiated with different security parameters $\lambda_1 = \lambda_1(\lambda, s)$, $\lambda_2 = \lambda_2(\lambda, s)$, and $\lambda_3 = \lambda_3(\lambda, s)$. Specifically, on input the security parameter λ, and a bound on the circuit size s, the setup algorithm proceeds as follows:

  1. First, it samples a PRF key $K \leftarrow \mathsf{PRF.KeyGen}(1^{\lambda_1})$.
  2. Next, the setup algorithm defines the proving program $\mathsf{Prove}[K]$ and the verification program $\mathsf{Verify}[K]$ exactly as in Construction 3.1 (see Figs. 1 and 2). The setup algorithm constructs $\mathsf{ObfProve} \leftarrow i\mathcal{O}(1^{\lambda_3}, \mathsf{Prove}[K])$ and $\mathsf{ObfVerify} \leftarrow i\mathcal{O}(1^{\lambda_3}, \mathsf{Verify}[K])$. 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 A.3.

  Finally, it outputs the common reference string $\mathsf{crs} = (\mathsf{ObfProve}, \mathsf{ObfVerify})$.
- $\mathsf{P}(\mathsf{crs}, C, (w_1, \ldots, w_t))$: Same as in Construction 3.1.
- $\mathsf{V}(\mathsf{crs}, C, t, \pi)$: Same as in Construction 3.1.

Theorem A.2 (Completeness). *If $i\mathcal{O}$ is correct, then Construction A.1 is complete.*

Proof. This proof is exactly the same as the proof for the non-adaptive case (Theorem 3.2). □

Theorem A.3 (Soundness). *Suppose there exists positive constants $\alpha, \beta, \gamma \in (0,1)$ such that*

- *The puncturable PRF Π_{PRF} is functionality-preserving and satisfies $(2^{\lambda^\alpha}, 2^{-\lambda^\alpha})$-punctured pseudorandomness;*
- *The pseudorandom generator PRG is $(2^{\lambda^\beta}, 2^{-\lambda^\beta})$-secure; and*
- *The indistinguishability obfuscator $i\mathcal{O}$ is $(2^{\lambda^\gamma}, 2^{-\lambda^\gamma})$-secure.*

Suppose moreover that we instantiate Construction A.1 with $\lambda_1 = (s + \log s + \omega(\log\lambda))^{1/\alpha}$, $\lambda_2 = (s + \log s + \omega(\log\lambda))^{1/\beta}$, and $\lambda_3 = (s + \log s + \omega(\log\lambda))^{1/\gamma}$, for a fixed polynomial $\mathsf{poly}(\cdot)$. Note that $\lambda_1, \lambda_2, \lambda_3 = \mathsf{poly}(\lambda, s)$. Then, Construction A.1 is adaptively sound.

Proof. We start by defining a sequence of hybrid experiments. In the following analysis, we say that C is a Boolean circuit of size s if it can be described by a binary string of length exactly s. Moreover, we will associate the set of all Boolean circuits with size at most s with a binary string of length $s+1$ (i.e., an element of the set $\{0,1\}^{s+1}$).

- Hyb_0: This is the adaptive soundness experiment:
  - At the beginning of the experiment, the adversary $\mathcal{A}$ outputs the maximum circuit size 1^s. The number of instances is implicitly taken to be $T = 2^\lambda$.
  - The challenger samples $\mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda, 1^s)$ and gives $\mathsf{crs} = (\mathsf{ObfProve}, \mathsf{ObfVerify})$ to adversary $\mathcal{A}$.
  - The adversary $\mathcal{A}$ outputs (C^*, t, π) where the size of C^* is at most s.
  - The output of the experiment is 1 if $\mathsf{V}(\mathsf{crs}, C^*, t, \pi) = 1$ (i.e., if $\mathsf{ObfVerify}(C^*, t, \pi) = 1$) and there exists an index $i \in [t]$ such that for all $w \in \{0,1\}^*$, $C^*(i, w) = 0$. Otherwise, the experiment outputs 0.
- Hyb'_0: Same as Hyb_0, except at the beginning of the security game, after the adversary outputs the bound 1^s on the maximum circuit size, the challenger guesses a circuit $C' \xleftarrow{R} \{0,1\}^{s+1}$. After the adversary outputs (C^*, t, π), the challenger outputs 0 if $C' \neq C^*$. Otherwise, the output is computed exactly as in Hyb_0.

  Additionally, the challenger exhaustively searches for a bad instance $i^* \in \{0,1\}^\lambda$. Namely, for each instance index $i \in \{0,1\}^\lambda$, it checks to see if for all $w \in \{0,1\}^m$, it holds that $C'(i, w) = 0$. If so, it sets $i^* = i$. If there are multiple such indices $i \in \{0,1\}^\lambda$, it sets i^* to be the smallest index (when interpreting i as the binary representation of a λ-bit integer).

- Hyb_j for $j \in \{i^*, \ldots, t\}$: Same as Hyb'_0 except the challenger changes the distribution of the CRS. Specifically, it defines the modified programs $\mathsf{Prove}'[K, i^*, i_{\mathsf{thresh}}, C']$ and $\mathsf{Verify}'[K, i^*, i_{\mathsf{thresh}}, C']$ exactly as in the proof of Theorem 3.3 (see Figs. 3 and 4). To construct the CRS, the challenger computes $\mathsf{ObfProve} \leftarrow i\mathcal{O}(1^\lambda, \mathsf{Prove}'[K, i^*, j, C'])$ and $\mathsf{ObfVerify} \leftarrow i\mathcal{O}(1^\lambda, \mathsf{Verify}'[K, i^*, j, C'])$ As in the real scheme, the challenger pads the size of Prove' and Verify' to the maximum size of the circuits that appear in the proof of Theorem A.3.

For an adversary $\mathcal{A}$, we write $\mathsf{Hyb}_i(\mathcal{A})$ to denote the output distribution of $\mathsf{Hyb}_i(\mathcal{A})$ with adversary $\mathcal{A}$. We now show that each pair of adjacent distributions defined above are indistinguishable. Unlike the proof of Theorem 3.3, our analysis here relies on complexity leveraging and the reduction algorithm will compute for itself the index i^* of the false instance. As a result, we no longer need to rely on non-uniform advice, and thus, the following reductions are all uniform.

Lemma A.4. *For all adversaries $\mathcal{A}$ and all security parameters $\lambda \in \mathbb{N}$, $\Pr[\mathsf{Hyb}_0(\mathcal{A}) = 1] = 2^{s+1} \cdot \Pr[\mathsf{Hyb}'_0(\mathcal{A}) = 1]$.*

Proof. If $\mathsf{Hyb}_0(\mathcal{A})$ outputs 1, then it must be the case that $\mathcal{A}$ outputs a circuit C^* of size at most s, which means $C^* \in \{0,1\}^{s+1}$. Since the challenger samples $C' \xleftarrow{R} \{0,1\}^{s+1}$ and moreover, C' is independent of the adversary's view, $\Pr[C' = C^*] = 1/2^{s+1}$. Thus, $\Pr[\mathsf{Hyb}'_0(\mathcal{A}) = 1] = 1/2^{s+1} \cdot \Pr[\mathsf{Hyb}_0(\mathcal{A}) = 1]$. □

Lemma A.5. *Suppose $i\mathcal{O}$ is $(2^{\lambda^\gamma}, 2^{-\lambda^\gamma})$-secure. Then for every efficient adversary $\mathcal{A}$, there exists a negligible function $\varepsilon(\lambda) = \mathsf{negl}(\lambda)$ such that for all $\lambda \in \mathbb{N}$, $|\Pr[\mathsf{Hyb}_{i^*}(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}'_0(\mathcal{A})]| \le \varepsilon/2^s$.*

Proof. By the same argument as in the proof of Lemma 3.4, the programs $\mathsf{Prove}[K]$ and $\mathsf{Prove}'[K, i^*, i^*, C']$ as well as the programs $\mathsf{Verify}[K]$ and $\mathsf{Verify}'[K, i^*, i^*, C']$ have identical input/output behavior. The claim now follows by $i\mathcal{O}$ security. Since we need to rely on complexity leveraging in the security reduction, we provide more details here.

First, let Hyb''_0 be an intermediate experiment where we change the ObfProve program from $\mathsf{ObfProve} \leftarrow i\mathcal{O}(1^{\lambda_3}, \mathsf{Prove}[K])$ (as in Hyb'_0) to $\mathsf{ObfProve} \leftarrow i\mathcal{O}(1^{\lambda_3}, \mathsf{Prove}'[K, i^*, i^*, C'])$ as in Hyb_{i^*}). In Hyb''_0, the verification program is still computed as $\mathsf{ObfVerify} \leftarrow i\mathcal{O}(1^{\lambda_3}, \mathsf{Verify}[K])$ as in Hyb'_0.

We now show that if there exists a $\mathsf{poly}(\lambda)$-time algorithm $\mathcal{A}$ where for for some non-negligible $\varepsilon = \varepsilon(\lambda)$, there exists an infinite set $\Lambda_{\mathcal{A}} \subseteq \mathbb{N}$, such that for all $\lambda \in \Lambda_{\mathcal{A}}$, $|\Pr[\mathsf{Hyb}''_0(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}'_0(\mathcal{A}) = 1]| \ge \varepsilon/2^s$. Then there exists a $2^{\lambda_3{}^\gamma}$-time algorithm $\mathcal{B}$ that breaks $i\mathcal{O}$ security. We note that while the adversary $\mathcal{A}$ runs on security parameter λ, the reduction will run on security parameter $\lambda_3 = \lambda_3(\lambda, s)$. The formal details are mentioned below.

Let s_λ be the deterministic value of the circuit output by $\mathcal{A}$ when run on security parameter $\lambda \in \mathbb{N}$. Let $\Lambda_{\mathcal{B}} = \left\{ \lceil (s_\lambda + \log(s_\lambda) + \omega(\log\lambda))^{1/\gamma} \rceil : \lambda \in \Lambda_{\mathcal{A}} \right\}$. Since Λ_A is an infinite set, and the function $\omega(\log\lambda)$ is monotone for sufficiently-large lambda, and s is non-negative, Λ_B is also infinite. Consider the reduction below, $\mathcal{B}(1^{\lambda_3})$, on input λ_3 we get a corresponding λ as advice such that $\lambda_3 = \lceil (s_\lambda + \log(s_\lambda) + \omega(\log\lambda))^{1/\gamma} \rceil$ (if collisions exist, i.e. two values of λ map to the value λ_3, we can break the tie arbitrarily).

1. Algorithm $\mathcal{B}$ starts running algorithm $\mathcal{A}$ on 1^λ, which outputs the bound on the circuit size 1^s.
2. Algorithm $\mathcal{B}$ randomly samples a Boolean circuit $C' \xleftarrow{R} \{0,1\}^{s+1}$ of size at most s. Algorithm $\mathcal{B}$ interprets $C' : \{0,1\}^\lambda \times \{0,1\}^m \to \{0,1\}$ as a circuit for an index relation. If C' cannot be interpreted as a Boolean circuit in this way, then algorithm $\mathcal{B}$ aborts with output 0.
3. Algorithm $\mathcal{B}$ exhaustively searches for a bad instance $i^* \in \{0,1\}^\lambda$. Namely, for each instance index $i \in \{0,1\}^\lambda$, algorithm $\mathcal{B}$ checks to see if for all $w \in \{0,1\}^m$, it holds that $C'(i, w) = 0$. If so, algorithm $\mathcal{B}$ sets $i^* = i$. If there are multiple such indices $i \in \{0,1\}^\lambda$, algorithm $\mathcal{B}$ sets i^* to be the smallest index (when interpreting i as the binary representation of a λ-bit integer).
4. Algorithm $\mathcal{B}$ samples a PRF key $K \leftarrow \mathsf{PRF.KeyGen}(1^{\lambda_1})$ and outputs $\mathsf{Prove}[K]$ and $\mathsf{Prove}'[K, i^*, i^*, C']$ as its challenge programs. Let ObfProve be the obfuscated program it receives from the challenger.
5. Algorithm $\mathcal{B}$ computes $\mathsf{ObfVerify} \leftarrow i\mathcal{O}(1^{\lambda_3}, \mathsf{Verify}[K])$ and gives $\mathsf{crs} = (\mathsf{ObfProve}, \mathsf{ObfVerify})$ to $\mathcal{A}$.
6. Finally, algorithm $\mathcal{A}$ outputs a tuple (C^*, t, π). If $C' \neq C^*$ or $i^* > t$, algorithm $\mathcal{B}$ outputs 0. Otherwise, algorithm $\mathcal{B}$ outputs 1 if $\mathsf{V}(\mathsf{crs}, C^*, t, \pi) = 1$ (i.e., if $\mathsf{ObfVerify}(C^*, t, \pi) = 1$) and 0 otherwise.

First, consider the running time of $\mathcal{B}$. Since C' is a circuit of size s, evaluating C' requires time s. Thus, computing the index of the bad instance i^* takes time at most $2^{\lambda+m} \cdot s \leq 2^{s+\log s}$ since $s \geq \lambda + m$ (i.e., the size of the circuit is at least as large as the input length to the circuit). Thus, the total running time of algorithm $\mathcal{B}$ is bounded by $2^s \cdot \mathsf{poly}(\lambda) \leq 2^{s+\log s+\omega(\log\lambda)} \leq 2^{\lambda_3{}^\gamma}$ (for sufficiently large $\lambda_3 \in \Lambda_{\mathcal{B}}$). Next, we consider the advantage of $\mathcal{B}$.

- Suppose the challenger obfuscates $\mathsf{Prove}[K]$. Then algorithm $\mathcal{B}$ perfectly simulates the distribution of Hyb_0' and algorithm $\mathcal{B}$ outputs 1 with probability $\Pr[\mathsf{Hyb}_0'(\mathcal{A}) = 1]$.
- Suppose the challenger obfuscates $\mathsf{Prove}'[K, i^*, i^*, C']$. Then $\mathcal{B}$ perfectly simulates Hyb_0'' and outputs 1 with probability $\Pr[\mathsf{Hyb}_0''(\mathcal{A}) = 1]$.

The distinguishing advantage of $\mathcal{B}$ is $|\Pr[\mathsf{Hyb}_0''(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_0'(\mathcal{A}) = 1]| \geq \varepsilon/2^s$. Since ε is non-negligible, we have that, $\varepsilon/2^s \geq 2^{-\lambda_3{}^\gamma}$ (for sufficiently large $\lambda_3 \in \Lambda_{\mathcal{B}}$). Thus we have a contradiction and we have broken $i\mathcal{O}$ security for sufficiently large values $\lambda \in \Lambda_{\mathcal{B}}$. Finally, we can conclude that $\varepsilon \leq 2^{-\omega(\log\lambda)} = \mathsf{negl}(\lambda)$, as required.

By an analogous argument, we can show that for all efficient adversaries $\mathcal{A}$, there exists a negligible function $\varepsilon'(\lambda) = \mathsf{negl}(\lambda)$ such that for all $\lambda \in \mathbb{N}$, $|\Pr[\mathsf{Hyb}_0''(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_{i^*}(\mathcal{A}) = 1]| = \varepsilon'/2^s$. The claim now follows by a hybrid argument. □

Lemma A.6. *Suppose $i\mathcal{O}$ is $(2^{\lambda^\gamma}, 2^{-\lambda^\gamma})$ secure, Π_{PRF} is functionality-preserving and satisfies $(2^{\lambda^\alpha}, 2^{-\lambda^\alpha})$-punctured pseudorandomness, and PRG is $(2^{\lambda^\beta}, 2^{-\lambda^\beta})$-secure. Then, for all $j \in \{i^*, \ldots, t-1\}$, and every efficient adversary $\mathcal{A}$, there exists a negligible function $\varepsilon(\lambda) = \mathsf{negl}(\lambda)$ such that for all $\lambda \in \mathbb{N}$*

$$|\Pr[\mathsf{Hyb}_{j+1}(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_j(\mathcal{A}) = 1]| \leq \varepsilon/2^s.$$

Proof. We begin by introducing a sequence of intermediate hybrids:

- $\mathsf{Hyb}_j^{(1)}$: Same as Hyb_j except the challenger changes the distribution of the CRS. Specifically, it defines the modified programs $\mathsf{Prove}''[K\{(C', i_{\mathsf{thresh}})\}, i^*, i_{\mathsf{thresh}}, C', z]$ and $\mathsf{Verify}''[K\{(C', i_{\mathsf{thresh}})\}, i^*, i_{\mathsf{thresh}}, C', z]$ exactly as in the proof of Lemma 3.5 (see Figs. 5 and 6). The challenger then computes the punctured key $K\{(C', j)\} \leftarrow \mathsf{PRF.Puncture}(K, (C', j))$ and the evaluation $z^* \leftarrow \mathsf{PRG}(\mathsf{PRF.Eval}(K, (C', j)))$. It then constructs $\mathsf{ObfProve} \leftarrow i\mathcal{O}(1^\lambda, \mathsf{Prove}''[K\{(C', j)\}, i^*, j, C', z^*])$ and $\mathsf{ObfVerify} \leftarrow i\mathcal{O}(1^\lambda, \mathsf{Verify}''[K\{(C', j)\}, i^*, j, C', z^*])$. As in the real scheme, the challenger pads the size of Prove'' and Verify'' to the maximum size of the circuits that appear in the proof of Theorem A.3. The CRS is still $\mathsf{crs} = (\mathsf{ObfProve}, \mathsf{ObfVerify})$.
- $\mathsf{Hyb}_j^{(2)}$: Same as $\mathsf{Hyb}_j^{(1)}$ but when constructing the CRS, the challenger sets $z^* \leftarrow \mathsf{PRG}(y^*)$ where $y^* \xleftarrow{R} \{0,1\}^\lambda$.
- $\mathsf{Hyb}_j^{(3)}$: Same as $\mathsf{Hyb}_j^{(2)}$ but when constructing the CRS, the challenger samples $z^* \xleftarrow{R} \{0,1\}^{2\lambda}$.

We now argue that each pair of adjacent hybrid experiments are indistinguishable for all $j \in \{i^*, \ldots, t\}$.

Claim A.7. *Suppose Π_{PRF} is functionality preserving and suppose $i\mathcal{O}$ is $(2^{\lambda^\gamma}, 2^{-\lambda^\gamma})$-secure. Then for all $j \in \{i^*, \ldots, t\}$ and every efficient adversary $\mathcal{A}$, there exists a negligible function $\varepsilon(\lambda) = \mathsf{negl}(\lambda)$ such that for all $\lambda \in \mathbb{N}$,*

$$|\Pr[\mathsf{Hyb}_j^{(1)}(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_j^{(0)}(\mathcal{A}) = 1]| \leq \varepsilon/2^s.$$

Proof. We appeal to the proof of Claim 3.6 to show that $\mathsf{Prove}'[K, i^*, j, C']$ and $\mathsf{Prove}''[K\{(C', j)\}, i^*, j, C', z]$ have the same functionality, as do $\mathsf{Verify}'[K, i^*, j, C']$ and $\mathsf{Verify}''[K\{(C', j)\}, i^*, j, C', z^*]$. From here, we can apply the same reduction as in the proof of Lemma A.5 to show the claim. □

Claim A.8. *Suppose Π_{PRF} satisfies $(2^{\lambda^\alpha}, 2^{-\lambda^\alpha})$-pseudorandomness. Then for all $j \in \{i^*, \ldots, t\}$, every efficient adversary $\mathcal{A}$, there exists a negligible function $\varepsilon(\lambda) = \mathsf{negl}(\lambda)$ such that for all $\lambda \in \mathbb{N}$,*

$$|\Pr[\mathsf{Hyb}_j^{(2)}(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_j^{(1)}(\mathcal{A}) = 1]| \leq \varepsilon/2^s.$$

Proof. We now show that if there exists a $\mathsf{poly}(\lambda)$-time algorithm $\mathcal{A}$ where for some non-negligible $\varepsilon = \varepsilon(\lambda)$, there exists an infinite set $\Lambda_{\mathcal{A}} \subseteq \mathbb{N}$, such that for all $\lambda \in \Lambda_{\mathcal{A}}$, $|\Pr[\mathsf{Hyb}_j^{(2)}(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_j^{(1)}(\mathcal{A}) = 1]| \leq \varepsilon/2^s$. Then there exists a $2^{\lambda_3{}^\alpha}$-time algorithm $\mathcal{B}$ that breaks punctured pseudorandomness security. We note that while the adversary $\mathcal{A}$ runs on security parameter λ, the reduction will run on security parameter $\lambda_1 = \lambda_1(\lambda, s)$. The formal details are mentioned below.

Let s_λ be the deterministic value of the circuit output by $\mathcal{A}$ when run on security parameter $\lambda \in \mathbb{N}$. Let $\Lambda_{\mathcal{B}} = \left\{\lceil (s_\lambda + \log(s_\lambda) + \omega(\log\lambda))^{1/\alpha} \rceil : \lambda \in \Lambda_{\mathcal{A}}\right\}$. Since Λ_A is an infinite set, and the function $\omega(\log\lambda)$ is monotone for sufficiently-large lambda, and s is non-negative, Λ_B is also infinite. Consider the reduction below, $\mathcal{B}(1^{\lambda_1})$, on input λ_1 we get a corresponding λ as advice such that $\lambda_1 = \lceil (s_\lambda + \log(s_\lambda) + \omega(\log\lambda))^{1/\alpha} \rceil$ (if collisions exist, i.e. two values of λ map to the value λ_1, we can break the tie arbitrarily).

1. Algorithm $\mathcal{B}$ starts running algorithm $\mathcal{A}$ on 1^λ, which outputs the bound on the circuit size 1^s.
2. Algorithm $\mathcal{B}$ randomly samples a Boolean circuit $C' \xleftarrow{R} \{0,1\}^{s+1}$ of size at most s. Algorithm $\mathcal{B}$ interprets $C' \colon \{0,1\}^\lambda \times \{0,1\}^m \to \{0,1\}$ as a circuit for an index relation. If C' cannot be interpreted as a Boolean circuit in this way, then algorithm $\mathcal{B}$ aborts with output 0.
3. Algorithm $\mathcal{B}$ exhaustively searches for a bad instance $i^* \in \{0,1\}^\lambda$. Namely, for each instance index $i \in \{0,1\}^\lambda$, algorithm $\mathcal{B}$ checks to see if for all $w \in \{0,1\}^m$, it holds that $C'(i,w) = 0$. If so, algorithm $\mathcal{B}$ sets $i^* = i$. If there are multiple such indices $i \in \{0,1\}^\lambda$, algorithm $\mathcal{B}$ sets i^* to be the smallest index (when interpreting i as the binary representation of a λ-bit integer).
4. Algorithm $\mathcal{B}$ outputs (C', j) as its challenge point. The challenger replies with a punctured key $K\{(C', j)\}$ and a challenge value y where either $y \leftarrow \mathsf{PRF.Eval}(K, (C', j))$ or $y \xleftarrow{R} \{0,1\}^{\lambda_2}$.
5. Algorithm $\mathcal{B}$ computes $z^* \leftarrow \mathsf{PRG}(y)$ and computes $\mathsf{ObfProve} \leftarrow i\mathcal{O}(1^{\lambda_3}, \mathsf{Prove}''[K\{(C', j)\}, i^*, j, C', z^*])$ and $\mathsf{ObfVerify} \leftarrow i\mathcal{O}(1^{\lambda_3}, \mathsf{Verify}''[K\{(C', j)\}, i^*, j, C', z^*])$. It gives $\mathsf{crs} = (\mathsf{ObfProve}, \mathsf{ObfVerify})$ to $\mathcal{A}$.
6. Adversary $\mathcal{A}$ outputs (C^*, t, π). If $C' \neq C^*$ or $i^* > t$, algorithm $\mathcal{B}$ outputs 0.
7. Otherwise, algorithm $\mathcal{B}$ outputs 1 if $\mathsf{ObfVerify}(C^*, t, \pi) = 1$.

If $\mathcal{B}$ received $\mathsf{PRF.Eval}(K, (C', j))$, then it perfectly simulates an execution of hybrid $\mathsf{Hyb}_j^{(1)}$ and outputs 1 with probability $\Pr[\mathsf{Hyb}_j^{(1)}(\mathcal{A}) = 1]$. Alternatively, if it receives a random challenge, then it perfectly simulates $\mathsf{Hyb}_j^{(2)}$ and outputs 1 with probability $\Pr[\mathsf{Hyb}_j^{(2)}(\mathcal{A}) = 1]$. Thus, the advantage of $\mathcal{B}$ is exactly

$$|\Pr[\mathsf{Hyb}_j^{(2)}(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_j^{(1)}(\mathcal{A}) = 1]| \leq \varepsilon/2^s.$$

Since algorithm $\mathcal{B}$ performs an exhaustive search to find the index i^*, the running time of $\mathcal{B}$ is $2^s \cdot s \cdot \mathsf{poly}(\lambda) \leq 2^{\lambda_1{}^\alpha}$ (for sufficiently large $\lambda_1 \in \Lambda_{\mathcal{B}}$). The claim now follows by $(2^{\lambda^\alpha}, 2^{-\lambda^\alpha})$ security: namely, we require that $\varepsilon/2^s \leq 2^{-\lambda_1{}^\alpha} = 2^{-s-\omega(\log\lambda)}$ (for sufficiently large $\lambda_1 \in \Lambda_{\mathcal{B}}$). □

Claim A.9. *Suppose* PRG *is* $(2^{\lambda^\beta}, 2^{-\lambda^\beta})$ *secure. Then for all* $j \in \{i^*, \ldots, t\}$ *and every efficient adversary* $\mathcal{A}$*, there exists a negligible function* $\varepsilon(\lambda) = \mathsf{negl}(\lambda)$ *such that for all* $\lambda \in \mathbb{N}$*,*

$$|\Pr[\mathsf{Hyb}_j^{(3)}(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_j^{(2)}(\mathcal{A}) = 1]| \leq \varepsilon/2^s.$$

Proof. We now show that if there exists a $\mathsf{poly}(\lambda)$-time algorithm $\mathcal{A}$ where for for some non-negligible $\varepsilon = \varepsilon(\lambda)$, there exists an infinite set $\Lambda_{\mathcal{A}} \subseteq \mathbb{N}$, such that for all $\lambda \in \Lambda_{\mathcal{A}}$, $|\Pr[\mathsf{Hyb}_j^{(3)}(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_j^{(2)}(\mathcal{A}) = 1]| \leq \varepsilon/2^s$. Then there exists a $2^{\lambda_2{}^\beta}$-time algorithm $\mathcal{B}$ that breaks PRG security. We note that while the adversary $\mathcal{A}$ runs on security parameter λ, the reduction will run on security parameter $\lambda_2 = \lambda_2(\lambda, s)$. The formal details are mentioned below.

Let s_λ be the deterministic value of the circuit output by $\mathcal{A}$ when run on security parameter $\lambda \in \mathbb{N}$. Let $\Lambda_{\mathcal{B}} = \left\{\lceil (s_\lambda + \log(s_\lambda) + \omega(\log\lambda))^{1/\beta} \rceil : \lambda \in \Lambda_{\mathcal{A}}\right\}$. Since Λ_A is an infinite set, and the function $\omega(\log\lambda)$ is monotone

for sufficiently-large lambda, and s is non-negative, Λ_B is also infinite. Consider the reduction below, $\mathcal{B}(1^{\lambda_2})$, on input λ_2 we get a corresponding λ as advice such that $\lambda_2 = \lceil (s_\lambda + \log(s_\lambda) + \omega(\log\lambda))^{1/\beta} \rceil$ (if collisions exist, i.e. two values of λ map to the value λ_2, we can break the tie arbitrarily).

1. Algorithm $\mathcal{B}$ starts running algorithm $\mathcal{A}$ on 1^λ, which outputs the bound on the circuit size 1^s.
2. Algorithm $\mathcal{B}$ randomly samples a Boolean circuit $C' \xleftarrow{R} \{0,1\}^{s+1}$ of size at most s. Algorithm $\mathcal{B}$ interprets $C' \colon \{0,1\}^\lambda \times \{0,1\}^m \to \{0,1\}$ as a circuit for an index relation. If C' cannot be interpreted as a Boolean circuit in this way, then algorithm $\mathcal{B}$ aborts with output 0.
3. Algorithm $\mathcal{B}$ exhaustively searches for a bad instance $i^* \in \{0,1\}^\lambda$. Namely, for each instance index $i \in \{0,1\}^\lambda$, algorithm $\mathcal{B}$ checks to see if for all $w \in \{0,1\}^m$, it holds that $C'(i,w) = 0$. If so, algorithm $\mathcal{B}$ sets $i^* = i$. If there are multiple such indices $i \in \{0,1\}^\lambda$, algorithm $\mathcal{B}$ sets i^* to be the smallest index (when interpreting i as the binary representation of a λ-bit integer).
4. Algorithm $\mathcal{B}$ samples a PRF key $K \leftarrow \mathsf{PRF.KeyGen}(1^{\lambda_1})$ and computes the punctured key $K\{(C',j)\} \leftarrow \mathsf{PRF.Puncture}(K,(C',j))$.
5. Algorithm $\mathcal{B}$ receives a challenge z^* from the challenger where either $z^* \leftarrow \mathsf{PRG}(y)$ for $y \leftarrow \{0,1\}^{\lambda_2}$ or $z^* \leftarrow \{0,1\}^{2\lambda_2}$.
6. Algorithm $\mathcal{B}$ computes the programs $\mathsf{ObfProve} \leftarrow i\mathcal{O}(1^{\lambda_3}, \mathsf{Prove}''[K\{(C',j)\}, i^*, j, C', z^*])$ and $\mathsf{ObfVerify} \leftarrow i\mathcal{O}(1^{\lambda_3}, \mathsf{Verify}''[K\{(C',j)\}, i^*, j, C', z^*])$. It gives $\mathsf{crs} = (\mathsf{ObfProve}, \mathsf{ObfVerify})$ to $\mathcal{A}$.
7. Adversary $\mathcal{A}$ outputs (C^*, t, π). If $C' \neq C^*$ or $i^* > t$, algorithm $\mathcal{B}$ outputs 0.
8. Algorithm $\mathcal{B}$ outputs 1 if $\mathsf{ObfVerify}(C^*, t, \pi) = 1$.

If $z^* \leftarrow \mathsf{PRG}(y)$, then algorithm $\mathcal{B}$ perfectly simulates an execution of hybrid $\mathsf{Hyb}_j^{(2)}$ and if $z^* \leftarrow \{0,1\}^{2\lambda_2}$, it perfectly simulates an execution of hybrid $\mathsf{Hyb}_j^{(3)}$.

Thus, the advantage of $\mathcal{B}$ is exactly

$$|\Pr[\mathsf{Hyb}_j^{(3)}(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_j^{(2)}(\mathcal{A}) = 1]| \leq \varepsilon/2^s.$$

Since algorithm $\mathcal{B}$ performs an exhaustive search to find the index i^*, the running time of $\mathcal{B}$ is $2^s \cdot s \cdot \mathsf{poly}(\lambda) \leq 2^{\lambda_2{}^\beta}$ (for sufficiently large $\lambda_2 \in \Lambda_{\mathcal{B}}$). The claim now follows by $(2^{\lambda^\beta}, 2^{-\lambda^\beta})$ security: namely, we require that $\varepsilon/2^s \leq 2^{-\lambda_2{}^\beta} = 2^{-s-\omega(\log\lambda)}$ (for sufficiently large $\lambda_2 \in \Lambda_{\mathcal{B}}$). □

Claim A.10. *Suppose $i\mathcal{O}$ is $(2^{\lambda^\gamma}, 2^{-\lambda^\gamma})$-secure. Then, for all $j \in \{i^*, \ldots, t\}$, and all efficient adversaries $\mathcal{A}$, there exists a negligible function $\varepsilon(\lambda) = \mathsf{negl}(\lambda)$ such that for all $\lambda \in \mathbb{N}$,*

$$|\Pr[\mathsf{Hyb}_{j+1}(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_{j^{(3)}}(\mathcal{A}) = 1]| \leq \varepsilon/2^s.$$

Proof. Using the same analysis as in Claim 3.9, with overwhelming probability $\mathsf{Prove}''[K\{(C',j)\}, i^*, j, C', z^*]$ and $\mathsf{Prove}'[K, i^*, j+1, C']$ have the same functionality, as do $\mathsf{Verify}''[K\{(C',j)\}, i^*, j, C', z^*]$ and $\mathsf{Verify}'[K, i^*, j+1, C']$ in Hyb_{j+1}. From here, we can apply the same reduction as in the proof of Lemma A.5 to show the claim. □

Combining Claims A.7 to A.10, we have that for all $j \in \{i^*, \ldots, t-1\}$, there exists a negligible function $\varepsilon(\lambda) = \mathsf{negl}(\lambda)$ such that for all $\lambda \in \mathbb{N}$, we have that $|\Pr[\mathsf{Hyb}_{j+1}(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_j(\mathcal{A}) = 1]| \leq \varepsilon/2^s$ and Lemma A.6 follows. □

By construction, in Hyb_t, the program $\mathsf{ObfVerify}$ is an obfuscation of the verification program $\mathsf{Verify}'[K, i^*, t, C']$ which outputs 0 on all inputs of the form (C', t, π) for any $\pi \in \{0,1\}^{\lambda_2}$. Correspondingly, for all efficient adversaries $\mathcal{A}$, it follows that $\Pr[\mathsf{Hyb}_t(\mathcal{A}) = 1] = 0$. Then combining Lemmas A.5 and A.6, we have that there exists a negligible function $\varepsilon(\lambda) = \mathsf{negl}(\lambda)$ such that $\Pr[\mathsf{Hyb}'_0(\mathcal{A}) = 1] \leq \varepsilon(\lambda)/2^s$. By Lemma A.4, this means that

$$\Pr[\mathsf{Hyb}_0(\mathcal{A}) = 1] \leq 2^{s+1} \cdot \varepsilon(\lambda)/2^s = 2 \cdot \varepsilon(\lambda) = \mathsf{negl}(\lambda),$$

and adaptive soundness holds. □

Theorem A.11 (Succinctness)**.** *Construction A.1 is succinct (but* not *fully succinct).*

Proof. We consider each property separately:

- **Succinct proof size:** The size of the proof is the output of PRF, which is a bit-string of length λ_2. For soundness (Theorem A.3), we require that $\lambda_2 = (s + \log s + \omega(\log \lambda))^{1/\beta}$, for some constant $\beta \in (0, 1)$. Thus, the proof size is $\mathsf{poly}(\lambda, s)$, which is independent of the number of instances. Thus, Construction A.1 is succinct (but not fully succinct since the proof size scales with the circuit size s).
- **Succinct verification time:** The verification algorithm consists of evaluating ObfVerify on a triple (C, t, π). By construction, ObfVerify is an obfuscation of the verification algorithm $\mathsf{Verify}[K]$. construction, the running time of $\mathsf{Verify}[K]$ is $\mathsf{poly}(\lambda, s)$. Since $i\mathcal{O}$ is efficiency-preserving, the running time of the obfuscated program ObfVerify is also $\mathsf{poly}(\lambda, s)$, as required. □

Theorem A.12 (Zero Knowledge)**.** *Construction A.1 satisfies perfect zero-knowledge.*

Proof. Our proof here is identical to the proof of Theorem 4.10 where the simulator invokes the underlying algorithms under their respective security parameters. □

B BARGs for NP from BARGs for Index Languages

In this section, we show how to upgrade an adaptively-secure index BARG (e.g., Construction A.1) to construct batch arguments for arbitrary NP languages. Our construction is the direct analog of Construction 4.4 except we need to rely on somewhere statistically-binding (SSB) hash functions [HW15] in place of the positional accumulators in order to argue adaptive security. Similar to Construction A.1, our construction critically relies on sub-exponential hardness of the underlying primitives. We start by recalling the formal definition of an SSB hash function (this is essentially a generalization of the two-to-one SSB hash functions from Section 5.1):

Definition B.1 (Somewhere Statistically Binding Hash Function [HW15, OPWW15, adapted])**.** A somewhere statistically binding (SSB) hash function consists of a tuple of efficient algorithms $\Pi_{\mathsf{SSB}} = (\mathsf{Gen}, \mathsf{GenTD}, \mathsf{Hash}, \mathsf{Open}, \mathsf{Verify})$ with the following properties:

- $\mathsf{Gen}(1^\lambda, 1^\ell) \to \mathsf{hk}$: On input the security parameter λ and the block size ℓ, the hash-key-generator algorithm outputs a hash key hk.
- $\mathsf{GenTD}(1^\lambda, 1^\ell, i^*) \to \mathsf{hk}$: On input the security parameter λ, the block size ℓ, and a target index $i^* \le 2^\lambda$, the trapdoor-generator algorithm outputs a hash key hk.
- $\mathsf{Hash}(\mathsf{hk}, (x_1, \ldots, x_t)) \to y$: On input a hash key hk and an ordered list of inputs $x_1, \ldots, x_t \in \{0,1\}^\ell$, the hash algorithm outputs a hash value y.
- $\mathsf{Open}(\mathsf{hk}, (x_1, \ldots, x_t), i) \to \pi$: On input a hash key hk, an ordered list of inputs $x_1, \ldots, x_t \in \{0,1\}^\ell$, and an index $i \in [t]$, the open algorithm outputs an opening π.
- $\mathsf{Verify}(\mathsf{hk}, y, x, i, \pi) \to b$: On input a hash key hk, 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 $b \in \{0,1\}$.

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

- **Correctness:** For all security parameters $\lambda \in \mathbb{N}$, block sizes $\ell \in \mathbb{N}$, all $t \le 2^\lambda$, all indices $i \in [t]$, and all tuples of inputs $x_1, \ldots, x_t \in \{0,1\}^\ell$,

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

- **Succinctness:** There exists a universal polynomial $\mathsf{poly}(\cdot,\cdot)$ such that the lengths of the hash values y output by Hash and the lengths of the proofs π output by Open in the completeness experiment satisfy $|y| = \mathsf{poly}(\lambda, \ell)$, $|\pi| = \mathsf{poly}(\lambda, \ell)$.
- **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 an input length ℓ and an index $i \leq 2^\lambda$.
  - If $b = 0$, the challenger samples $\mathsf{hk}_0 \leftarrow \mathsf{Gen}(1^\lambda, 1^\ell)$. Otherwise, if $b = 1$, the challenger samples $\mathsf{hk}_1 \leftarrow \mathsf{GenTD}(1^\lambda, 1^\ell, i)$. 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:** We say that a hash key hk is *statistically binding at index i* if for all y, π, π^*, there does not exist inputs $x, x^* \in \{0,1\}^\ell$ where $x \neq x^*$ and $\mathsf{Verify}(\mathsf{hk}, y, x, i, \pi) = 1 = \mathsf{Verify}(\mathsf{hk}, y, x^*, i, \pi^*)$. We say that the hash function is statistically binding if for all block sizes $\ell = \mathsf{poly}(\lambda)$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that for all $\lambda \in \mathbb{N}$
  $$\Pr[\mathsf{hk} \text{ is statistically binding at index } i : \mathsf{hk} \leftarrow \mathsf{GenTD}(1^\lambda, 1^\ell, i)] \geq 1 - \mathsf{negl}(\lambda).$$

Theorem B.2 (Somewhere Statistically Binding Hash Functions [HW15, OPWW15])**.** *Under standard number-theoretic assumptions (e.g., DDH, DCR, LWE, or ϕ-Hiding), there exists an SSB hash function for arbitrary polynomial input lengths $\ell = \ell(\lambda)$.*

Remark B.3 (Hashing Variable Number of Inputs)**.** In [HW15, OPWW15], the generator algorithms Gen, GenTD for the SSB hash function also take as input the number of inputs T (in binary); correspondingly, the hashing algorithm always takes T inputs $x_1, \ldots, x_T$ as input. In our setting, we allow the hash function to support an arbitrary number of inputs $t \leq 2^\lambda$.[13] We can construct an SSB hash function that supports a variable number of inputs (with a maximum of 2^λ inputs) with $\mathsf{poly}(\lambda)$ overhead using a standard "powers-of-two" construction:

- First, we define the hash key to be a tuple of λ hash keys $\mathsf{hk} = (\mathsf{hk}_1, \ldots, \mathsf{hk}_\lambda)$, where the i^{th} hash key hk_λ is for an SSB scheme on exactly 2^i inputs.
- To hash an input $(x_1, \ldots, x_t)$ where $t \leq 2^\lambda$, the hashing algorithm first pads $(x_1, \ldots, x_t, \bot, \ldots, \bot)$ to a tuple of length 2^i where i is the smallest integer where $t \leq 2^i$. It then hashes the padded input with hk_i to obtain the hash value y'. The overall hash value is the pair $y = (t, y')$.

Observe that this construction still supports efficient hashing (padding to the next power of two only incurs constant overhead). Including the input length as part of the hash output preserves the somewhere statistical binding property.

Construction B.4 (Adaptively-Sound 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 $\Pi_{\mathsf{SSB}} = (\mathsf{SSB.Gen}, \mathsf{SSB.GenTD}, \mathsf{SSB.Hash}, \mathsf{SSB.Open}, \mathsf{SSB.Verify})$ be an somewhere statistically binding hashing function.

[13] Note that padding the input to $T = 2^\lambda$ would render the hashing algorithm inefficient (when invoked on $\mathsf{poly}(\lambda)$-length inputs).

- Let $\Pi_{\mathsf{IndexBARG}} = (\mathsf{IndexBARG.Gen}, \mathsf{IndexBARG.P}, \mathsf{IndexBARG.V})$ be a BARG for Index languages that supports unbounded statements.[14]

We define our batch argument $\Pi_{\mathsf{BARG}} = (\mathsf{Gen}, \mathsf{P}, \mathsf{V})$ for batch circuit satisfiability 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 $\mathsf{hk} \leftarrow \mathsf{SSB.Gen}(1^{\lambda'}, 1^\ell)$ where λ' is set according to Theorem B.6. Let s' be the size of the following circuit:

  Constants: Hash key hk for Π_{SSB}, hash value h for Π_{SSB}, 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 $\mathsf{SSB.Verify}(\mathsf{hk}, h, x, i, \sigma) = 0$, output 0.
  3. Otherwise, output 1.

  Figure 15: The Boolean circuit $C'[\mathsf{hk}, h, C]$ for an index relation

  Then, sample $\mathsf{IndexBARG.crs} \leftarrow \mathsf{IndexBARG.Gen}(1^\lambda, 1^{s'})$. Output $\mathsf{crs} = (\mathsf{hk}, \mathsf{IndexBARG.crs})$.

- $\mathsf{P}(\mathsf{crs}, C, (x_1, \ldots, x_t), (w_1, \ldots, w_t))$: On input $\mathsf{crs} = (\mathsf{hk}, \mathsf{IndexBARG.crs})$, 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 $w_1, \ldots, w_t \in \{0,1\}^m$, the prove algorithm starts by computing a hash $h \leftarrow \mathsf{SSB.Hash}(\mathsf{hk}, (x_1, \ldots, x_t))$. Then, for each all $i \in [t]$, let $w'_i = (x_i, \sigma_i, w_i)$ where $\sigma_i \leftarrow \mathsf{SSB.Open}(\mathsf{hk}, (x_1, \ldots, x_t), i)$. Output the proof $\pi \leftarrow \mathsf{IndexBARG.P}(\mathsf{IndexBARG.crs}, C', t, (w'_1, \ldots, w'_t))$ where $C'[\mathsf{hk}, h, C]$ is the circuit for the index relation from Fig. 15.

- $\mathsf{V}(\mathsf{crs}, C, (x_1, \ldots, x_t), \pi)$: On input $\mathsf{crs} = (\mathsf{hk}, \mathsf{IndexBARG.crs})$, 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 a proof π, the verification algorithm starts by computing the hash $h \leftarrow \mathsf{SSB.Hash}(\mathsf{hk}, (x_1, \ldots, x_t))$. It then outputs $\mathsf{IndexBARG.V}(\mathsf{IndexBARG.crs}, C', t, \pi)$ where $C'[\mathsf{hk}, h, C]$ is the circuit for the index relation from Fig. 15.

Theorem B.5 (Completeness). *If $\Pi_{\mathsf{IndexBARG}}$ scheme is complete and Π_{SSB} scheme is correct, then Construction B.4 is complete.*

Proof. Take any security parameter $\lambda \in \mathbb{N}$, circuit size bound $s \in \mathbb{N}$, input length $\ell \in \mathbb{N}$, Boolean circuit $C\colon \{0,1\}^\ell \times \{0,1\}^m \to \{0,1\}$ with size at most s, and any instance number $t \le 2^\lambda$. Let $x_1, \ldots, x_t \in \{0,1\}^\ell$ be a collection of statements and $w_1, \ldots, w_t$ be a collection of corresponding witnesses such that $C(x_i, w_i) = 1$ for all $i \in [t]$. Suppose $\mathsf{crs} = (\mathsf{hk}, \mathsf{IndexBARG.crs}) \leftarrow \mathsf{Gen}(1^\lambda, 1^\ell, 1^s)$ and $\pi \leftarrow \mathsf{Prove}(\mathsf{crs}, C, (x_1, \ldots, x_t), (w_1, \ldots, w_t))$. Let $\sigma_i \leftarrow \mathsf{SSB.Open}(\mathsf{hk}, (x_1, \ldots, x_t), i)$ be the openings computed by the prove algorithm. Since SSB is correct, for every $i \in [t]$, $\mathsf{SSB.Verify}(\mathsf{hk}, h, x, i, \sigma_i) = 1$, and correspondingly, for every $i \in [t]$, $C'(i, (x_i, \sigma_i, w_i)) = 1$, where $C'(\cdot) = C'[\mathsf{hk}, h, C](\cdot)$ is the circuit from Fig. 15. Completeness now follows from completeness of the underlying BARG for index languages. □

Theorem B.6. *Suppose $\Pi_{\mathsf{IndexBARG}}$ satisfies adaptive soundness. Moreover, suppose there exists a constant $\delta \in (0,1)$ and a negligible function $\mathsf{negl}(\cdot)$ such that Π_{SSB} satisfies $(2^{\lambda^\delta}, \mathsf{negl}(\lambda))$-index hiding. Let $\lambda' = (s + \omega(\log \lambda))^{1/\delta}$, where s is a bound on the circuit size. Then Construction B.4 satisfies adaptive soundness.*

Proof. We start by defining a series of hybrid experiments. Let $q = q(\lambda)$ be a polynomial that upper bounds the number of instances an adversary outputs (or equivalently, the running time of the adversary).

- Hyb_0: This is the adaptive soundness experiment.
  - Adversary $\mathcal{A}$ starts by outputting the maximum circuit size $1^{s(\lambda)}$, and a statement length $1^{\ell(\lambda)}$.
  - The challenger responds with $\mathsf{crs} \leftarrow \mathsf{Gen}(1^\lambda, 1^\ell, 1^s)$.

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

- Adversary $\mathcal{A}$ outputs $(C^*, (x_1^*, \ldots, x_t^*), \pi^*)$ where C^* is a Boolean circuit of size at most $s(\lambda)$ and $x_i^* \in \{0,1\}^\ell$ for all $i \in [t]$.
- The output of the experiment is 1 if $\mathsf{V}(\mathsf{crs}, C^*, (x_1^*, \ldots, x_t^*), \pi^*) = 1$ and there exists $i \leq t$ where for all $w \in \{0,1\}^*$, $C^*(x_i^*, w) = 0$. Otherwise, the challenger outputs 0.

- Hyb_1: Same as Hyb_0, except at the beginning of the security experiment, the challenger samples a random index $i^* \in [q]$. After the adversary outputs $(C^*, (x_1^*, \ldots, x_t^*), \pi^*)$, the challenger outputs 0 if either $i^* > t$ or there exists a witness $w \in \{0,1\}^*$ such that $C^*(x_{i^*}^*, w) = 1$.

- Hyb_2: Same as Hyb_1 except the challenger samples $\mathsf{hk} \leftarrow \mathsf{SSB.GenTD}(1^\lambda, 1^\ell, i^*)$.

For an adversary $\mathcal{A}$, we write $\mathsf{Hyb}_i(\mathcal{A})$ to denote the output distribution of an execution of $\mathsf{Hyb}_i(\mathcal{A})$ with adversary $\mathcal{A}$. We now show that each pair of adjacent distributions are indistinguishable.

Lemma B.7. *For every adversary $\mathcal{A}$ and for all $\lambda \in \mathbb{N}$, $\Pr[\mathsf{Hyb}_0(\mathcal{A}) = 1] \leq q \cdot \Pr[\mathsf{Hyb}_1(\mathcal{A}) = 1]$.*

Proof. If Hyb_0 outputs 1, then there is at least one index $i \leq t$ such that for all witnesses $w \in \{0,1\}^*$, $C^*(x_i, w) = 0$. Since i^* is uniform and independent of the view of the adversary (and thus, of the index i) and $q \geq t$, with probability at least $1/q$, it will be the case that $i = i^*$. In this case, the output in Hyb_1 is identical to the output in Hyb_0. Thus $\Pr[\mathsf{Hyb}_1(\mathcal{A}) = 1] \geq 1/q \cdot \Pr[\mathsf{Hyb}_0(\mathcal{A}) = 1]$ and the claim holds. □

Lemma B.8. *Suppose there exist a constant $\delta \in (0,1)$ and a negligible function $\varepsilon(\lambda) = \mathsf{negl}(\lambda)$ such that Π_{SSB} satisfies $(2^{\lambda^\delta}, \varepsilon(\lambda))$-index hiding. Suppose also that $\lambda' = (s + \log s + \omega(\log \lambda))^{1/\delta}$, where s is a bound on the size of the Boolean circuit for the* NP *relation. Then for every efficient adversary $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that for all $\lambda \in \mathbb{N}$,*

$$|\Pr[\mathsf{Hyb}_2(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_1(\mathcal{A}) = 1]| \leq \mathsf{negl}(\lambda).$$

Proof. By construction, Hyb_1 only outputs 1 if for all witnesses $w \in \{0,1\}^*$, $C^*(x_{i^*}^*, w) = 0$ *and* the adversary outputs π^* such that $\mathsf{V}(\mathsf{crs}, C^*, (x_1^*, \ldots, x_t^*), \pi^*) = 1$. This is also the case in Hyb_2, except the hash key is now sampled to be binding on index i^*. Suppose if there exists a $\mathsf{poly}(\lambda)$-time algorithm $\mathcal{A}$ where for for some non-negligible $\varepsilon' = \varepsilon'(\lambda)$, there exists an infinite set $\Lambda_{\mathcal{A}} \subseteq \mathbb{N}$, such that for all $\lambda \in \Lambda_{\mathcal{A}}$, $|\Pr[\mathsf{Hyb}_2(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_1(\mathcal{A}) = 1]| = \varepsilon'$. Then there exists a $2^{\lambda'^\delta}$-time algorithm $\mathcal{B}$ that breaks index hiding. We note that while the adversary $\mathcal{A}$ runs on security parameter λ, the reduction will run on security parameter $\lambda' = \lambda'(\lambda, s)$. The formal details are mentioned below.

Let s_λ be the deterministic value of the circuit output by $\mathcal{A}$ when run on security parameter $\lambda \in \mathbb{N}$. Let $\Lambda_{\mathcal{B}} = \left\{ \lceil (s_\lambda + \log(s_\lambda) + \omega(\log \lambda))^{1/\delta} \rceil : \lambda \in \Lambda_{\mathcal{A}} \right\}$. Since Λ_A is an infinite set, and the function $\omega(\log \lambda)$ is monotone for sufficiently-large lambda, and s is non-negative, Λ_B is also infinite. Consider the reduction below, $\mathcal{B}(1^{\lambda'})$, on input λ' we get a corresponding λ as advice such that $\lambda' = \lceil (s_\lambda + \log(s_\lambda) + \omega(\log \lambda))^{1/\delta} \rceil$ (if collisions exist, i.e. two values of λ map to the value λ', we can break the tie arbitrarily).

1. Algorithm $\mathcal{B}$ starts running algorithm $\mathcal{A}$ on 1^λ who starts by outputting a bound on the circuit size 1^s and a bound on the statement size 1^ℓ.
2. Algorithm $\mathcal{B}$ samples a random index $i^* \in [q]$ and gives 1^ℓ and i^* to the index hiding challenger.
3. Algorithm $\mathcal{B}$ receives a hash key hk where either $\mathsf{hk} \leftarrow \mathsf{Gen}(1^{\lambda'}, 1^\ell)$ or $\mathsf{hk} \leftarrow \mathsf{GenTD}(1^{\lambda'}, 1^\ell, i^*)$.
4. Algorithm $\mathcal{B}$ samples $\mathsf{IndexBARG.crs} \leftarrow \mathsf{IndexBARG.Gen}(1^\lambda, 1^{s'})$ and gives $\mathsf{crs} = (\mathsf{hk}, \mathsf{IndexBARG.crs})$ to $\mathcal{A}$.
5. Adversary $\mathcal{A}$ outputs $(C^*, (x_1^*, \ldots, x_t^*), \pi^*)$ where C^* is of size at most $s(\lambda)$ and for every $i \in [t]$, $x_i^* \in \{0,1\}^\ell$.
6. Algorithm $\mathcal{B}$ outputs 1 if $\mathsf{V}(\mathsf{crs}, C^*, (x_1^*, \ldots, x_t^*), \pi^*) = 1$ and for all $w \in \{0,1\}^*$, $C^*(x_{i^*}^*, w) = 0$. Otherwise, it outputs 0.

If $\mathcal{A}$ runs in $\mathsf{poly}(\lambda)$ time, then $\mathcal{B}$ runs in time at most $2^s \cdot s \cdot \mathsf{poly}(\lambda) \leq 2^{s+\log s+\omega(\log\lambda)} = 2^{(\lambda')^\delta}$ (for sufficiently large $\lambda' \in \Lambda_{\mathcal{B}}$) since $\mathcal{B}$ needs to exhaustively check whether there exists a witness w where $C^*(x^*_{i^*}, w) = 1$, and there can be at most 2^s candidate values for w (checking each candidate requires time at most s). Next, if $\mathsf{hk} \leftarrow \mathsf{Gen}(1^{\lambda'}, 1^\ell)$, then $\mathcal{B}$ perfectly simulates Hyb_1. If $\mathsf{hk} \leftarrow \mathsf{GenTD}(1^{\lambda'}, 1^\ell, i^*)$, then $\mathcal{B}$ perfectly simulates Hyb_2. Thus, the distinguishing advantage of $\mathcal{B}$ is

$$|\Pr[\mathsf{Hyb}_2(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_1(\mathcal{A}) = 1]| = \varepsilon'.$$

Since algorithm $\mathcal{B}$ runs in time $2^{(\lambda')^\delta}$ and Π_{SSB} satisfies $(2^{\lambda^\delta}, \varepsilon(\lambda))$-index hiding, it must be the case that $\varepsilon'(\lambda) \leq \varepsilon(\lambda') = \mathsf{negl}(\lambda') = \mathsf{negl}(\lambda)$ (for sufficiently large $\lambda' \in \Lambda_{\mathcal{B}}$), and the claim holds. □

Lemma B.9. *Suppose $\Pi_{\mathsf{IndexBARG}}$ is adaptively sound and Π_{SSB} is statistically binding. Then for all efficient adversaries $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that for all $\lambda \in \mathbb{N}$, $\Pr[\mathsf{Hyb}_2(\mathcal{A}) = 1] = \mathsf{negl}(\lambda)$.*

Proof. Suppose there exists an efficient adversary $\mathcal{A}$ such that $\Pr[\mathsf{Hyb}_2(\mathcal{A}) = 1] = \varepsilon$ for some non-negligible ε. We use $\mathcal{A}$ to construct an adversary $\mathcal{B}$ to break adaptive soundness of $\Pi_{\mathsf{IndexBARG}}$.

1. Algorithm $\mathcal{B}$ starts running adversary $\mathcal{A}$. Algorithm $\mathcal{A}$ starts by outputting the maximum circuit size 1^s and the statement lengths 1^ℓ.
2. Algorithm $\mathcal{B}$ outputs the maximum circuit size $1^{s'}$ where s' is an upper bound on the circuit size for $C'[\mathsf{hk}, h, C]$, where C' is the circuit for the index relation from Fig. 15 and the circuit C has size at most s.
3. Algorithm $\mathcal{B}$ receives a common reference string IndexBARG.crs from the challenger. It randomly samples an index $i^* \xleftarrow{R} [q]$, computes $\mathsf{hk} \leftarrow \mathsf{SSB.GenTD}(1^{\lambda'}, 1^\ell, i^*)$, and gives $\mathsf{crs} = (\mathsf{hk}, \mathsf{IndexBARG.crs})$ to $\mathcal{A}$.
4. Adversary $\mathcal{A}$ outputs $(C^*, (x^*_1, \ldots, x^*_t), \pi^*)$. Algorithm $\mathcal{B}$ computes $h^* \leftarrow \mathsf{SSB.Hash}(\mathsf{hk}, (x^*_1, \ldots, x^*_t))$ and outputs the circuit $C'[\mathsf{hk}, h^*, C^*]$ as its challenge circuit and π^* as the proof.

By construction, algorithm $\mathcal{B}$ perfectly simulates an execution of Hyb_2 for $\mathcal{A}$. Thus, with probability at least ε, the output of Hyb_2 is 1. This means the following conditions hold with probability at least ε:

- The index i^* satisfies $i^* \leq t$.
- For all witnesses $w \in \{0,1\}^*$, it holds that $C^*(x^*_{i^*}, w) = 0$.
- The proof π^* is a valid proof for the index relation $C'[\mathsf{hk}, h^*, C^*]$. Specifically,

$$\mathsf{IndexBARG.V}(\mathsf{IndexBARG.crs}, C'[\mathsf{hk}, h^*, C^*], t, \pi^*) = 1.$$

We now argue that $C'[\mathsf{hk}, h^*, C^*](i^*, w'_{i^*}) = 0$ for all inputs w'_{i^*}. First write $w'_{i^*} = (x', \sigma', w')$. We consider two possibilities:

- Suppose $x' = x^*_{i^*}$. Since $C^*(x^*_{i^*}, w') = 0$ for every choice of w', we conclude that $C'[\mathsf{hk}, h^*, C^*](i^*, w'_{i^*}) = 0$.
- Suppose $x' \neq x^*_{i^*}$. Since the hash key hk is sampled to bind on index i^*, with all but negligible probability over the choice of hk, the *only* value of x' for which there exists σ' such that $\mathsf{SSB.Verify}(\mathsf{hk}, h^*, x', i, \sigma') = 1$ is $x' = x^*_{i^*}$. Thus, with overwhelming probability over the choice of hk, $\mathsf{SSB.Verify}(\mathsf{hk}, h^*, x', i^*, \sigma') = 0$ in this case. Once again, $C'[\mathsf{hk}, h^*, C^*](i^*, w'_{i^*}) = 0$.

Thus, we conclude that with overwhelming probability over the choice of hk, $C[\mathsf{hk}, h^*, C^*]'(i^*, w'_{i^*}) = 0$ for all $w'_{i^*} \in \{0,1\}^*$. Since $i^* \leq t$, if π^* is a valid proof on $(C'[\mathsf{hk}, h^*, C^*], t)$, then algorithm $\mathcal{B}$ breaks adaptive soundness of $\Pi_{\mathsf{IndexBARG}}$ with probability $\varepsilon - \mathsf{negl}(\lambda)$. □

Combining Lemmas B.7 to B.9, we have that for all efficient adversaries $\mathcal{A}$, there exists a negligible function $\varepsilon(\lambda) = \mathsf{negl}(\lambda)$ such that $\Pr[\mathsf{Hyb}_0(\mathcal{A}) = 1] \leq q(\lambda) \cdot \varepsilon(\lambda)$. Since $\mathcal{A}$ is efficient, $q(\lambda) = \mathsf{poly}(\lambda)$, and the claim holds. □

Theorem B.10 (Succinctness). *If $\Pi_{\mathsf{IndexBARG}}$ scheme is succinct and Π_{SSB} is succinct, then Construction B.4 has succinct proofs (but* not *succinct verification).*

Proof. Let ℓ be the statement length and s be the size of the Boolean circuit for the underlying NP relation. The proof π in Construction B.4 is a proof for $\Pi_{\mathsf{IndexBARG}}$ on the new circuit $C'[\mathsf{hk}, h, C]$. First, the size of the hash key hk satisfies $|\mathsf{hk}| = \mathsf{poly}(\lambda', \ell)$ and $\lambda' = \mathsf{poly}(\lambda, s)$. We can always bound the input length by the circuit size so overall, we can write $|\mathsf{hk}| = \mathsf{poly}(\lambda, s)$. Next, succinctness of Π_{SSB} requires that the length of a hash output h and of an opening σ to satisfy $|h|, |\sigma| = \mathsf{poly}(\lambda', \ell) = \mathsf{poly}(\lambda, s)$. As such, the size of the circuit $C'[\mathsf{hk}, h, C]$ is at most $\mathsf{poly}(\lambda, s)$. The claim now follows by succinctness of $\Pi_{\mathsf{IndexBARG}}$. □

Remark B.11 (Non-Succinct Verification). We note that due to complexity leveraging, Construction B.4 does not have succinct verification time. Namely, verifying t instances of a Boolean circuit of size s requires time $\mathsf{poly}(\lambda, s, t)$. The reason is that the size of the hash key $|\mathsf{hk}| = \mathsf{poly}(\lambda', \ell) = \mathsf{poly}(\lambda, s)$, where ℓ is the length of the statement and $\lambda' = \mathsf{poly}(\lambda, s)$. As such, computing $h \leftarrow \mathsf{SSB.Hash}(\mathsf{hk}, (x_1, \ldots, x_t))$ requires time $\mathsf{poly}(\lambda, s, t)$.

Note that if we have an a priori bound m on the length of the witness in the underlying NP relation, we could use a slightly tighter analysis in the proofs of Theorem B.6 and Lemma B.8 by considering reductions that run in time $2^{m+\log s+\omega(\log\lambda)}$. Specifically, the reduction algorithm in the proof of Lemma B.8 only needs to exhaustively search over all candidate witnesses, which requires time $2^m \cdot s$. In this case, we can set $\lambda' = m + \log s + \omega(\log\lambda)$ in Construction B.4. This would yield a BARG for NP where the verification time is $\mathsf{poly}(\lambda, m, t, \log s)$. This yields a modest saving over the naïve verification procedure in settings where the witness size is much smaller than the circuit size.

Theorem B.12 (Zero Knowledge). *If $\Pi_{\mathsf{IndexBARG}}$ satisfies perfect zero-knowledge, then Construction B.4 satisfies perfect zero-knowledge.*

Proof. Let $\mathsf{IndexBARG}.\mathcal{S}$ be a simulator for $\Pi_{\mathsf{IndexBARG}}$. We construct a simulator for Π_{BARG} as follows. On input the security parameter λ, a bound s on the circuit size, a Boolean circuit $C\colon \{0,1\}^\ell \times \{0,1\}^m \to \{0,1\}$, and instances $x_1, \ldots, x_t \in \{0,1\}^\ell$, the simulator proceeds as follows:

- Sample $\mathsf{hk} \leftarrow \mathsf{SSB.Gen}(1^{\lambda'}, 1^\ell)$, and compute $h \leftarrow \mathsf{SSB.Hash}(\mathsf{hk}, (x_1, \ldots, x_t))$.
- Let $C' = C'[\mathsf{hk}, h, C]$ be the circuit from Fig. 15 and let s' be a bound on the size of C'. Compute the simulated CRS and proof $(\mathsf{IndexBARG.crs}, \pi) \leftarrow \mathsf{IndexBARG}.\mathcal{S}(1^\lambda, 1^{s'}, C', t)$.
- Output the simulated CRS $\mathsf{crs} = (\mathsf{hk}, \mathsf{IndexBARG.crs})$ and the simulated proof π.

By construction, the hash function parameters hk and the circuit $C' = C'[\mathsf{hk}, h, s]$ are constructed exactly as in the real scheme. Perfect zero knowledge now follows from perfect zero knowledge of $\Pi_{\mathsf{IndexBARG}}$. □

Remark B.13 (Weaker Notions of Zero Knowledge). We note that Remark 4.11 also applies to Construction B.4. Namely, if $\Pi_{\mathsf{IndexBARG}}$ satisfies computational (resp., statistical) zero-knowledge, then Construction B.4 also satisfies computational (resp., statistical) zero-knowledge.