Towards Optimally Efficient Secret-Key Authentication from PRG

Ivan Damgård[1] and Sunoo Park[2]

[1]Aarhus University
[2]MIT

Abstract

We propose a new approach to the construction of secret-key authentication protocols making black-box use of a pseudorandom generator (PRG). Our authentication protocols require only two messages, have perfect completeness, and achieve concurrent man-in-the-middle security. Finally, when based on a sufficiently efficient PRG, our protocol has (amortised) complexity $O(n)$ bit operations where n is the security parameter. To the best of our knowledge, this construction is the first to have all these properties simultaneously, in particular the first with linear complexity. We achieve this at the cost of having the prover (but not the verifier) keep a small amount of state. Very practical PRGs can be constructed, for instance, based on the Learning Parity with Noise (LPN) problem, and our protocol is in several respects an attractive alternative even to protocols derived directly from LPN. A variant of our construction is secure even if the adversary is able to reset the prover.

1 Introduction

Secret-key authentication is one of most basic cryptographic tasks: a prover and a verifier share a secret key K, and the aim is to design a protocol that will allow the prover (and the prover alone), to convince the verifier that he indeed knows the key K. We call this *one-sided authentication*, to distinguish from mutual authentication where both parties must be convinced of each other's identity.

The strongest security notion in the authentication literature for one-sided authentication [1] considers an adversary who may first interact (concurrently) as many times as he wants with the honest prover and the verifier; after this, he is on his own and must attempt to falsely convince the verifier that he knows the secret key. If no efficient adversary can win this game, we say the protocol is concurrent man-in-the-middle (MIM) secure. For stateful protocols one can consider an even stronger variant of this notion: *multi-instance* concurrent MIM security. Here the adversary may, in the first phase, have several concurrent sessions with different incarnations of the prover and the verifier, where each incarnation will start from a fresh initial state. Thus, this stronger notion captures also attacks where the adversary can reset the parties.

Let us consider what resources we need, in terms of communication and computation, to realise (one-sided) authentication. It is easy to see that we need at least two messages, where the verifier speaks first. Furthermore, both messages must be long enough to not be easily guessed. Informally, if the prover's message is short, the adversary can guess what to say in the final phase; on the other hand, if the verifier's message is short, the adversary can guess what the verifier will say in the final phase and query the prover for the right answer in the first phase. Finally, we assume that both parties access n key bits, where n is the security parameter, so it follows that the computational complexity must be $\Omega(n)$ bit-operations.

[1]Bellare and Rogaway [BR93] proposed a definition for *mutual* authentication using an attack model that is essentially the same as the concurrent MIM attack we consider. We do not consider mutual authentication here, as we focus on two-message protocols, in which of course only one party can be convinced of the other's identity.

It is therefore a natural theoretical goal to build two-message authentication protocols of complexity $O(n)$, where n is the bit length of the shared key. Also, from a practical point of view, efficient "lightweight" authentication protocols are becoming ever more relevant as application scenarios emerge where, for instance, the prover may be a low-cost RFID tag or smartcard.

1.1 Motivation and related work

The first solution to the authentication problem is from [GGM86], as follows. Suppose we are given a pseudorandom function family (PRF) $\mathcal{F}$ whose functions f_K have a key K and input/output size n bits. The crucial property of a PRF is that even an adversary who chooses the input (but does not know the key) cannot distinguish the output of f_K from random. Given a PRF, we can simply let the verifier send a random input x, and have the prover respond with $f_K(x)$. This simple two-round protocol already achieves concurrent MIM security[2].

However, from the perspective of efficiency, this simple solution is problematic in general: the standard construction of a PRF (from [GGM86]), is based on a pseudorandom generator (PRG), which is a much simpler primitive that expands a short key into a random-looking longer output. However, the GGM construction does not yield a particularly efficient PRF: even assuming a very efficient PRG that only requires a constant number of operations per output bit, the PRF will have complexity $\Omega(n^2)$. In [BPR12], a construction of a PRF directly from the well-known LWE problem was suggested. Whereas this is indeed more efficient than applying the [GGM86] construction on the PRG that follows naturally from LWE, the complexity of the resulting PRF is still $\Omega(n \log n)$.

There are in fact several very efficient and low-depth constructions of PRGs from specific problems, such as Learning Parity with Noise (LPN); and furthermore, even linear-time computable PRGs with linear stretch are known to follow from general and plausible assumptions[3]. Hence, it is compelling to try to design authentication directly based on weaker pseudorandom primitives such as PRGs, in such a way that the protocol inherits the efficiency.

Authentication from weak pseudorandom primitives. This direction has been explored in [DKPW12], which constructs a three-round protocol based on any weak PRF. A weak PRF is a relaxed notion of PRF in which function outputs are only required to look random for uniformly random (rather than adversarially chosen) inputs. The protocol of [DKPW12] achieves active security (a weaker notion than MIM security). Subsequently, [LM13] proposed a three-round protocol based on any weak PRF, which is secure against sequential MIM attacks[4]. In addition, they give a variant three-round protocol that can be built from any *randomized* weak PRF, a yet slightly weaker primitive. However, these protocols are not concurrent MIM secure (in fact, [LM13] outlines an attack), and all require three rounds. Moreover, even a randomized weak PRF seems to be a significantly stronger primitive than a PRG.

Authentication from concrete hardness assumptions. An obvious alternative approach to authentication is to build protocols directly from a concrete problem. Much work in this direction has been based on the LPN problem, which can be briefly stated as follows: given polynomially

[2]For authentication, it is actually sufficient to have an unpredictable function, a computationally secure "MAC", rather than a PRF; however, we do not know of more efficient constructions of MACs either. In [IKOS08], constant-overhead PRFs and MACs are constructed, but here the output size is much smaller than the input, thus in our context this would make the prover's answer much easier to guess than the verifier's challenge.

[3]A number of different sufficient conditions for such PRG's are known. In [IKOS08] it is observed that such PRGs follow from Alekhnovich's variant of the Learning Parity with Noise assumption. Applebaum [App13] shows that such PRGs can be obtained from the assumption that a natural variant of Goldreich's candidate for a one-way function in NC0 is indeed one-way. The improved HILL-style result of Vadhan and Zheng [VZ12] implies that such PRGs can be obtained from any exponentially strong one-way function that can be computed by a linear-size circuit.

[4]Sequential MIM security is stronger than active security, but weaker than concurrent MIM security. In a sequential MIM attack, the adversary can interact polynomially many times with an honest prover and verifier, but the interactions cannot involve multiple concurrent sessions with the prover (or the verifier). In contrast, an active adversary can interact only with the prover.

many samples of the form $(\mathbf{a}, \langle \mathbf{a}, \mathbf{s} \rangle + e)$, where $\mathbf{a}$ is a random n-bit vector, $\mathbf{s}$ is a secret n-bit vector fixed across all samples, and e is a bit which is 1 with probability $\tau < 1/2$, can we discover the secret $\mathbf{s}$ (or at least, distinguish the samples from random)? Authentication from LPN has been studied in a series of papers, including [HB01; JW05; KSS10; Hey+12]. The latest in this line of work [Kil+11; DKPW12] have proposed two-message MIM secure protocols. Even based on a compact version of LPN (namely, the ring-LPN problem [Hey+12]), these protocols have computational complexity $\Omega(n \log n)$; and moreover, they are not perfectly complete: the verifier may reject the honest prover with non-zero probability. In order to keep this error probability low, the protocol's communication (and hence also computational) complexity must depend on the LPN noise parameter τ, and in fact the complexity grows quite dramatically as τ approaches $1/2$. Thus, these protocols incur a rather high price if we want to use a weaker variant of the LPN assumption where τ is close to $1/2$.

1.2 Our contribution

We propose a new two-message authentication protocol that can be based on black-box access to any PRG. The protocol has perfect completeness and is concurrent MIM secure. When based on a linear-time PRG, the amortised complexity[5] of the protocol is $O(n)$. To the best of our knowledge, our protocol is the first to have all these properties simultaneously. We achieve this by having the prover (but not the verifier) keep a small amount of state that grows logarithmically with the number of executions of the protocol.

A variant of the protocol is even multi-instance concurrent MIM secure for any polynomial number of instances, as long as this polynomial is fixed when the system is set up. If the number of instances is constant, we can maintain the linear complexity of the protocol, using a new construction we propose of linear-time computable ℓ-wise independent hash functions for any constant ℓ.

If we make a stronger assumption on the security of the PRG – namely, that it produces pseudorandom output even when used with seeds that are chosen from an ℓ-wise independent distribution rather than at random – then the second protocol achieves multi-instance concurrent MIM security (for *any* polynomial number of instances, which need not be known or bounded in advance).

A concrete instantiation of our protocol could be based on PRGs that follow naturally from LPN, and since we have perfect completeness (and thus avoid the bad dependence on τ mentioned in the previous section), this offers an attractive alternative to known protocols based directly on LPN. In particular, unlike previous protocols, our protocol has the property that as we move τ towards $1/2$ to weaken the underlying LPN assumption, we pay only in terms of computational complexity: our communication complexity stays unchanged.

Ideas and techniques. The basic idea of our protocol is that the verifier sends an n-bit random challenge a to the prover, who responds with an unconditionally secure MAC on a computed from a secret key that he shares with the verifier.

The good news is that unconditionally secure MACs (in contrast to computationally secure ones) can give us the efficiency we are after. In particular we will use a MAC of the form $C(a) * s \oplus e$ where (s, e) is the key, and where C is a linear-time encodable linear code that is good in the sense that both its length and minimum distance are $\Theta(n)$. Then $C(a)$ is the encoding of a and the product $C(a) * s$ is the component-wise (Schur) product. This one-time MAC is linear-time computable and was proven secure in [DZ13].

However, the bad news is that while s can be reused over several executions, e cannot: e must be a fresh random value every time, or the MAC is not secure. An obvious solution is to choose

[5] It should be noted that this refers to the time spent in normal operation between the honest prover and verifier. An adversary can force the verifier (but not the prover) to spend more time, but we do not view this as a significant problem: in practice, an adversary could always waste the verifier's time by doing a standard denial of service attack.

e pseudorandomly using another shared key K. Computing it as $e = f_K(a)$ where f is PRF may seem natural, but this would be pointless, because then the simpler PRF-based protocol mentioned above might as well be used, and we will again face the efficiency issues associated with using a PRF in this way. To get around this, we propose to have the prover keep a counter i that is incremented for each execution, and compute e as $e = f_K(i)$. The point is that now the prover does not need to compute the PRF on arbitrary unpredictable inputs, but only on values that arrive in a particular order $i = 1, 2, \ldots$. To see why this is an advantage, consider that the GGM construction basically forms a tree with exponentially many leaves, one for each possible input. Computing an output requires computing the path to the relevant leaf, calling the PRG once for each vertex. Then, to achieve linear time, the idea is to save the last path we computed and only recompute the part that changes for the next input. If the inputs indeed arrive in order, it turns out that on average we will only need to call the PRG twice per call to the PRF.

Using the original GGM tree in this way requires that we build a tree large enough to accommodate the maximal number of times we expect that the protocol will be executed. The storage requirement will depend on the bound that we decide. This is not desirable: to make sure we stay out of trouble, we may have to use more storage than it turns out we really need. We therefore propose a variant of the GGM construction where every node delivers an output value. This allows us to grow the tree as we go, so that the storage requirement only depends (logarithmically) on how many times *the protocol is actually executed*[6].

This construction achieves concurrent MIM security: the strongest security notion in the case where the adversary cannot clone or reset the prover. If the adversary can clone or reset, then we need multi-instance concurrent MIM security. We achieve this by modifying our protocol using an additional technique based on universal hashing. To this end, we propose an extension of a result by Ishai et al.[IKOS08]: they give the first construction of pairwise-independent hash functions that can be computed in linear time, but this construction fails already for 3-wise independence. We show a simple way to extend the construction to get ℓ-wise independence for any constant ℓ while maintaining the linear time complexity. This seems to be optimal in the sense that if the function outputs $\Omega(n)$ bits and is ℓ-wise independent, the randomness used to select the function must be of size $\Omega(\ell n)$ bits – and if the evaluation algorithm must look at all these random bits, linear time is not possible if ℓ is super-constant in n.

Usage of our protocols in practice. Since we have presented our results in terms of asymptotic complexity, one may wonder if they are of any use in practice. After all, in a practical scenario, it may seem much easier to just use your favourite block cipher as a PRF in the simple protocol we mentioned at the start.

However, one should consider the fact that (synchronous) stream ciphers can be used as PRGs and are often much faster than block ciphers. Seen from this angle, our protocol may offer an advantage because it provides a way to do authentication using, instead, your favourite stream cipher, where the prover on average calls it twice per execution, and in addition only needs to compute an efficient unconditionally secure MAC.

Note that in software, using a more standard MAC of form $a \cdot s + e$, where the product is in the field with 2^n elements, may be an advantage (even if it is not preferable asymptotically). For instance, for $n = 64$, we can multiply by a fixed element s using 8 table look-ups in 16 KB of precomputed tables. However, if the prover is a small hardware device, whereas the verifier has more power, then using the MAC $C(a) * s + e$ that we suggest may be even better, since we can ask the verifier to send the encoding $C(a)$ rather than a. Our proof of security still applies if the prover checks that he receives a codeword, and this can be extremely fast in hardware if we use an

[6] Goldreich [Gol01] suggested a somewhat related idea where the last leaf of the tree is used as the root of a new one. Using this technique instead could give us the similar time complexity for the prover, but an adversary could force the verifier to spend much more time than in normal operation, potentially $\Omega(nt)$, where t is the number of times the protocol is executed. The worst case for our approach is $O(n \log t)$.

LDPC (low-density parity check) code. This reduces the prover's computation essentially to the cost of running the stream cipher.

2 Preliminaries

Notation. For a finite set B, we will write $b \leftarrow B$ to denote that b is drawn uniformly randomly from B. For $n \in \mathbb{N}$, $[n]$ denotes the set $\{1, 2, \ldots, n\}$. The relation $\overset{s}{\approx}$ between distributions denotes statistical indistinguishability, and $\overset{c}{\approx}$ denotes computational indistinguishability. $\mathsf{negl}(n)$ denotes a negligible function in parameter n, and $\mathsf{poly}(n)$ denotes a polynomial function. An *efficient* algorithm is one which runs in probabilistic polynomial time (PPT).

The definitions of pseudorandom generators (PRGs) and pseudorandom function families (PRFs) are given in Appendix A.

2.1 Authentication protocols

A *authentication protocol* is an interactive two-party protocol $(\mathcal{P}, \mathcal{V})$ between a prover $\mathcal{P}$ and a verifier $\mathcal{V}$: these may be respectively thought of as a (lightweight) *tag*, and a *reader* to which the tag is identifying itself. Both parties are PPT, and hold a shared secret s generated according to some generation algorithm $\mathsf{Gen}(1^\kappa)$ (where κ denotes the security parameter) in an initial phase. After an execution of the protocol, the verifier $\mathcal{V}$ outputs either accept or reject – this is also called the *output* of the protocol execution.

In this work we consider *prover-stateful protocols* where the prover also maintains some (small amount of) state between protocol executions.

Definition 2.1 (Completeness)**.** *The* completeness error *of a protocol is:*

$$\Pr_{s \leftarrow \mathsf{Gen}(1^\kappa)}[(\mathcal{P}(s), \mathcal{V}(s)) = \text{reject}].$$

A protocol is complete *if its completeness error is negligible in a security parameter. It is* perfectly complete *if its completeness error is zero.*

Common definitions of security for authentication protocols are given below. Note that these definitions are adapted for the prover-stateful setting; they have natural analogues in the stateless setting which is more common in the literature.

Definition 2.2 (Active security)**.** *An authentication protocol $(\mathcal{P}, \mathcal{V})$ is* secure against active attacks *if for any secret $s \leftarrow \mathsf{Gen}(1^\kappa)$, for any PPT adversary $\mathcal{A}$ which first can interact arbitrarily polynomially many times with an honest prover $\mathcal{P}$ (but cannot reset the prover's state), and then afterward (now, without access to $\mathcal{P}$) interacts once with an honest verifier $\mathcal{V}$, it holds that*

$$\Pr[(\mathcal{A}, \mathcal{V}(s)) = \text{accept}] \leq \mathsf{negl}(\kappa).$$

Definition 2.3 (Concurrent man-in-the-middle (MIM) security)**.** *An authentication protocol $(\mathcal{P}, \mathcal{V})$ is* secure against concurrent man-in-the-middle attacks *if for any PPT adversary $\mathcal{A}$ which first can interact arbitrarily polynomially many times with an honest prover $\mathcal{P}$ and/or an honest verifier $\mathcal{V}$ (the interactions may be concurrent, but the adversary cannot reset the prover's state), and then afterward (now, without access to $\mathcal{P}, \mathcal{V}$) interacts once with an honest verifier $\mathcal{V}'$, it holds that*

$$\Pr[(\mathcal{A}, \mathcal{V}'(s)) = \text{accept}] \leq \mathsf{negl}(\kappa).$$

In this setting, the adversary learns the accept/reject *decisions of the verifier.*

Definition 2.4 (Multi-instance concurrent MIM security)**.** *An authentication protocol* $(\mathcal{P}, \mathcal{V})$ *is* secure against multi-instance concurrent man-in-the-middle attacks *if for any* PPT *adversary* $\mathcal{A}$ *which first can interact arbitrarily polynomially many times with polynomially many honest provers* $\mathcal{P}_1, \ldots, \mathcal{P}_k$ *and/or honest verifiers* $\mathcal{V}_1, \ldots, \mathcal{V}_k$*, and then afterward (now, without access to the* $\mathcal{P}_i, \mathcal{V}_i$*) interacts once with an honest verifier* $\mathcal{V}'$*, it holds that*

$$\Pr[(\mathcal{A}, \mathcal{V}'(s)) = \text{accept}] \leq \mathsf{negl}(\kappa).$$

As above, the adversary learns the accept/reject *decisions of the verifier(s).*

A natural relaxation of multi-instance concurrent MIM security addresses the case when the adversary has access to only a ℓ provers and verifiers, where $\ell = \mathsf{poly}(n)$ is bounded in advance. We write *ℓ-instance concurrent MIM security* to denote security against an adversary who has concurrent access to up to ℓ honest provers (and verifiers), but no more. This definition allows a construction to take advantage of the fact that ℓ is chosen and fixed initially. Furthermore, we define *ℓ-instance active security* to be the corresponding notion for active security: that is, where the adversary has access to up to ℓ honest provers.

Note that for stateful protocols, the multi-instance definition covers the case of an adversary who can reset, e.g., the prover's state: clearly, an adversary who can do this k times can be emulated by an adversary having access to k copies of the prover.

3 Authentication via "one-time" MACs

In this section, our approach is to build authentication protocols from very efficient MACs. It was observed in earlier sections that known MAC constructions are not very efficient as they are based on PRFs – however, this is only true of *computationally* secure MACs. In contrast, there are *unconditionally* secure MACs that are very efficient – but these are only secure for one-time use, so have not thus far been considered suitable for authentication protocols.

Notation. For vectors $v, w \in \{0,1\}^n$, $v * w$ denotes the component-wise (Schur) product, and $v \cdot w$ is the field product (in $\mathbb{F}_{2^n}$). For an error-correcting code C, $C.\mathsf{Enc}$ and $C.\mathsf{Dec}$ denote the encoding and decoding functions, respectively.

Consider the following simple and unconditionally secure MAC: for a message $a \in \{0,1\}^n$, the MAC on the message is $a \cdot s + e$, where $(s, e) \in \{0,1\}^n \times \{0,1\}^n$ is the secret key (which is chosen uniformly at random). MACs of this form are well known, and it is also known that although a key for an unconditionally secure MAC can usually be used only once, in this case the multiplier (s) can be reused provided that e is freshly chosen for each message (see e.g. [BDOZ11]).

In this work, we focus on a slightly different MAC, which might be considered a variant of the above. For a message $a \in \{0,1\}^n$, the MAC on the message is $C.\mathsf{Enc}(a) * s + e \in \{0,1\}^{cn}$ where C is an error-correcting code (with constant-fraction distance) with expansion c, and $(s, e) \in \{0,1\}^{cn} \times \{0,1\}^{cn}$ is the secret key. The security of this variant MAC is shown in [DZ13].

In our protocols, we consider s to be the secret key, and generate e pseudorandomly per execution. The man-in-the-middle security of our protocol does *not* follow from MAC security, however: the standard security notion for MACs simply requires that an adversary who observes a message and a valid MAC cannot produce a different message and valid MAC. We consider a more complicated game where the adversary interacts with prover and verifier concurrently.

Finally, we remark that although our protocols are presented in terms of the variant MAC, the proofs of correctness and security all go through (with almost no changes) also when using the "$a \cdot s + e$" MAC. Which version is better in a concrete application would be determined by the encoding efficiency of the error-correcting code for the relevant parameters.

3.1 Pseudorandom look-up function

As a building block for our authentication protocols, we construct a logarithmic-depth "look-up function" for efficient retrieval of pseudorandom values using the PRG, and show that the look-up function is a PRF. Note that this technique may be of independent interest towards generic constructions of low-depth PRFs.

Notation. Since a PRG G can be used to build a PRG of any stretch, we write $G_{n\to m}$ to denote the PRG based on G which maps n bits to m bits. We write $G_{n\to m}(r)^{[i,j]}$ to denote the substring of the PRG output $G_{n\to m}(r) \in \{0,1\}^m$ ranging from the i^{th} bit to the j^{th} bit, inclusive.

Given a PRG G taking an n-bit input, our goal is to generate a series of polynomially many pseudorandom values $r_1, r_2, r_3, \ldots$, such that each r_i can be looked up in time (poly-)logarithmic in i. This is achieved using the tree structure below.

Note that while the PRF construction of [GGM86] also performs lookups in logarithmic time (and can be easily adapted to support an unlimited input domain $\{0,1\}^*$, as pointed out in [Gol01]), their construction does not satisfy our required property: in their case, the lookup time for *any* output r_i is poly-logarithmic in the *total* number of queries that will ever be made to the PRF.

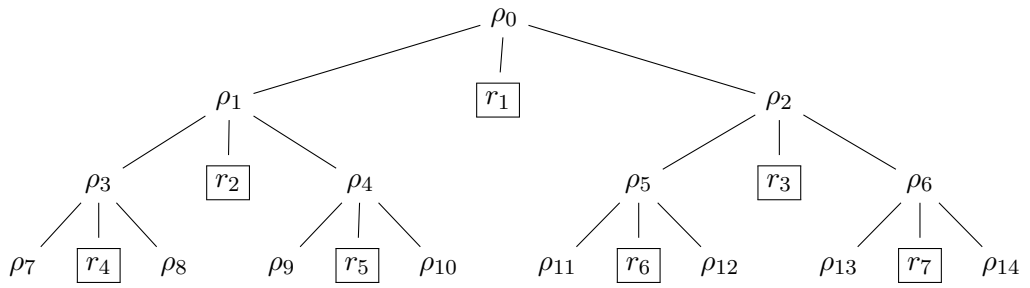

Figure 1: Tree illustrating efficient look-up of pseudorandom values (first 4 levels)

In Figure 1, $\rho_0 \in \{0,1\}^n$ is the original (random) input to the PRG $G_{n\to m+2n}$. The $\rho_i \in \{0,1\}^n$ are values which are subsequently pseudorandomly generated, which are used again as input to the PRG to produce more pseudorandom values: in particular, if ρ_i is a child of ρ_j in the tree, then $\rho_i = G_{n\to m+2n}(\rho_j)^{[m+1,m+n]}$ if i is even, and $\rho_i = G_{n\to m+2n}(\rho_j)^{[m+n+1,m+2n]}$ if i is odd. The boxed nodes $r_i \in \{0,1\}^m$ are leaves that represent the output pseudorandom values which we want to look up, and they are generated by $r_i = G_n(\rho_j)^{[1,m]}$ where ρ_j is the parent node of r_i.

Let $\mathsf{lookup}^G_{n,m}(\rho_0, i) \in \{0,1\}^m$ denote the i^{th} output value, $r_i \in \{0,1\}^m$, obtained using the above tree method. It is clear that for any i of polynomial size, the number of PRG evaluations required to look up r_i is logarithmic. This gives rise to a PRF family with logarithmic-depth evaluations, as proven in Theorem 3.2 below. Before proving that the look-up function is a PRF, we give a simple supporting lemma.

Lemma 3.1. *Let $G : \{0,1\}^n \to \{0,1\}^m$ be a PRG. Then for any polynomial $q = q(n)$, it holds that there is no efficient distinguisher D for which it holds that*

$$|\Pr[\mathsf{D}((r_1,\ldots,r_q)) = 1] - \Pr[\mathsf{D}((G(s_1),\ldots,G(s_q))) = 1]| \geq \mathsf{negl}(n)$$

for all negligible functions negl, where $r_1,\ldots,r_q \leftarrow \{0,1\}^{m(n)}$ and $s_1,\ldots,s_q \leftarrow \{0,1\}^n$.

Proof. Suppose, for contradiction, that there is a distinguisher $\widehat{\mathsf{D}}$ for which

$$\left|\Pr\left[\widehat{\mathsf{D}}((r_1,\ldots,r_q)) = 1\right] - \Pr\left[\widehat{\mathsf{D}}((G(s_1),\ldots,G(s_q))) = 1\right]\right| \geq 1/P(n)$$

where P is a polynomial. For $i \in [q]$, define tup_i to be the distribution of tuples whose first i elements are uniformly random in $\{0,1\}^m$ and whose remaining elements are sampled as $G(s_{i+1}), \ldots, G(s_q)$ for $s_{i+1}, \ldots, s_q \leftarrow \{0,1\}^n$. Let $p_i = \Pr[\widehat{\mathsf{D}}(\mathsf{tup}_i) = 1]$ denote the probability that $\widehat{\mathsf{D}}$ outputs 1 on input from tup_i.

By our supposition, we know $|p_0 - p_q| \geq P(n)$. Then, since $p_0 - p_q = \sum_{i \in [q]}(p_{i-1} - p_i)$, there must exist $i^* \in [q]$ such that $|p_{i^*-1} - p_{i^*}| \geq \frac{1}{q \cdot P(n)}$, which is non-negligible. Then there exists a distinguisher $\widehat{\mathsf{D}}'$ which can distinguish a *single* output of the PRG from random, as follows: on input $r \in \{0,1\}^m$, $\widehat{\mathsf{D}}'$ generates a tuple t whose first $i^* - 1$ elements are random in $\{0,1\}^m$, whose $(i^*)^{th}$ element is r, and whose remaining elements are generated as $G(s_{i^*+1}), \ldots, G(s_q)$ for $s_{i+1}, \ldots, s_q \leftarrow \{0,1\}^n$. If r is truly random then $t \leftarrow \mathsf{tup}_{i^*}$; otherwise, $t \leftarrow \mathsf{tup}_{i^*-1}$. Hence, running $\widehat{\mathsf{D}}$ on input t will distinguish with non-negligible probability between these cases. This contradicts that G is a PRG. □

Theorem 3.2. *Let G be a PRG and $n, m \in \mathbb{N}$ be positive integers with $m = \mathsf{poly}(n)$. Then the family of functions $\mathcal{F}^{(n,m)} \stackrel{\text{def}}{=} \{\mathsf{lookup}^G_{n,m}(\rho, \cdot)\}_{\rho \in \{0,1\}^n}$ is a PRF with input size n' bits and output size n bits, for any $n' = \mathsf{poly}(n)$.*

Proof. The statement to prove is that for any PRG G and random $\rho_0 \leftarrow \{0,1\}^n$, there is no efficient distinguisher D that satisfies

$$\left|\Pr\left[\mathsf{D}^{\mathsf{lookup}^G_{n,m}(\rho_0,\cdot)}() = 1\right] - \Pr\left[\mathsf{D}^{\mathcal{R}(\cdot)}() = 1\right]\right| > \mathsf{negl}(\kappa)$$

for all negligible functions negl and all sufficiently large values of the security parameter κ. In the above, $\mathcal{R}$ is a random oracle, and $\mathsf{D}^{\mathcal{O}}$ may make any polynomial number of "polynomial-depth"[7] queries to the oracle $\mathcal{O}$.

Suppose, for contradiction, that there exists such a distinguisher $\widehat{\mathsf{D}}$, for which the above inequality does not hold: that is, there is a polynomial q for which

$$\left|\Pr\left[\mathsf{D}^{\mathsf{lookup}^G_{n,m}(\rho_0,\cdot)}() = 1\right] - \Pr\left[\mathsf{D}^{\mathcal{R}(\cdot)}() = 1\right]\right| > 1/q(\kappa).$$

Let $T = T(\kappa)$ be the run-time of distinguisher $\widehat{\mathsf{D}}$, and let H_j be a stateful algorithm which is defined as follows. (Note that when referring to tree structure, the root node is considered to be at depth 0.) On input i, the algorithm H_j does the following:

- if i has already been queried previously, look up the stored tuple $(\mathsf{leaf}, i, \rho_i)$, and output ρ_i;
- else if $i < 2^{j+1}$ (that is, the i^{th} output node is at depth less than or equal to j in the tree), then choose some $\rho_i \leftarrow \mathbb{Z}_n^2$ uniformly at random, store the tuple $(\mathsf{leaf}, i, \rho_i)$ in memory, and output ρ_i;
- otherwise (that is, the i^{th} output node is at depth greater than j in the tree):
  - if there is no stored tuple of the form $(\mathsf{root}, \alpha, \rho)$, then choose some $\rho \leftarrow \mathbb{Z}_n^2$ uniformly at random, store the tuple $(\mathsf{root}, \alpha, \rho)$ in memory, and output $\mathsf{lookup}^G_{n,m}(\rho, \gamma)$;
  - otherwise, look up the stored tuple $(\mathsf{root}, \alpha, \rho)$ and output $\mathsf{lookup}^G_{n,m}(\rho, \gamma)$;

  where $\alpha = \alpha_{(i,j)}, \gamma = \gamma_{(i,j)}$ are defined by the following:

$$\begin{aligned}
\alpha_{(i,j)} &= \lfloor (i - 2^{\lfloor \log_2(i) \rfloor})/2^j \rfloor \\
\beta_{(i,j)} &= i - 2^{\lfloor \log_2(i) \rfloor} \mod 2^{\lfloor \log_2(i) \rfloor - j} \\
\gamma_{(i,j)} &= 2^{(2^{\lfloor \log_2(i) \rfloor - j})} + \beta_{(i,j)}.
\end{aligned}$$

[7] More precisely, the distinguisher cannot make queries that would require a super-polynomial depth look-up in the tree structure of $\mathsf{lookup}^G_{n,m}$. This is because in this case, $\mathsf{lookup}^G_{n,m}$ would not run in polynomial time.

When considering the tree representation of $\mathsf{lookup}^G_{n,m}$, the algorithms H_j can be explained in more intuitive terms as follows. For each j, the outputs of H_j behave as a random oracle up to and including depth j of the tree. Below depth j, the outputs are obtained deterministically by the $\mathsf{lookup}^G_{n,m}(\rho,\cdot)$ function, with the appropriate depth-j value ρ (which is randomly chosen) acting as the "root node" of the subtree in which the lookup is performed.

Observe that H_0 behaves exactly as $\mathsf{lookup}^G_{n,m}(\rho,\cdot)$ for random $\rho \leftarrow \{0,1\}^n$. Moreover, H_k behaves exactly like a random oracle in the distinguishing experiment, provided that all of $\widehat{\mathsf{D}}$'s queries can be retrieved from depth at most k. Since the input size is $n' = \mathsf{poly}(n)$ bits, there exists such a maximum depth k from which queries can be retrieved, with $k = \mathsf{poly}(n)$.

Let $p_i = \Pr[\widehat{\mathsf{D}}^{H_i}() = 1]$ denote the probability that the distinguisher outputs 1 given H_i as an oracle. By our earlier supposition, $|p_0 - p_k| > 1/q(\kappa)$. It follows that for some $k^* \in [k]$, we have $|p_{k^*-1} - p_{k^*}| > \frac{1}{k \cdot q(\kappa)}$. Such a k^* can be found in polynomial time with non-negligible probability[8].

We now construct a new distinguisher $\widehat{\mathsf{D}}_{\mathsf{PRG}}$ attacking the PRG $G_{n\to m+2n}$: specifically, we will show that the following expression is non-negligible:

$$\left|\Pr\left[\widehat{\mathsf{D}}_{\mathsf{PRG}}((r_1,\ldots,r_T)) = 1\right] - \Pr\left[\widehat{\mathsf{D}}_{\mathsf{PRG}}((G_{n\to m+2n}(s_1),\ldots,G_{n\to m+2n}(s_T))) = 1\right]\right|,$$

where $r_1,\ldots,r_T \leftarrow \{0,1\}^{m+2n}$ and $s_1,\ldots,s_T \leftarrow \{0,1\}^n$.

$\widehat{\mathsf{D}}_{\mathsf{PRG}}$ operates as follows. Given input $(\widetilde{r}_1,\ldots,\widetilde{r}_T)$, $\widehat{\mathsf{D}}_{\mathsf{PRG}}$ first determines a $k^* \in [k]$ as described above, then runs $\widehat{\mathsf{D}}$ and responds to the oracle queries of $\widehat{\mathsf{D}}$ in the following way: when $\widehat{\mathsf{D}}$ makes query i, $\widehat{\mathsf{D}}_{\mathsf{PRG}}$ responds with $\widetilde{H}_{k^*}(\widetilde{r}_1,\ldots,\widetilde{r}_T)$, where $\widetilde{H}_{k^*}$ is a variant algorithm based on H_{k^*}. $\widetilde{H}_{k^*}$ takes as input $(\widetilde{r}_1,\ldots,\widetilde{r}_T)$, and then behaves exactly like H_{k^*}, except that the values associated with nodes at depth k^* of the tree are obtained as substrings of the input values $\widetilde{r}_1,\ldots,\widetilde{r}_T$. (For a detailed formal description of $\widetilde{H}_{k^*}$, refer to Appendix B.)

By construction, it holds that if the inputs $\widetilde{r}_i$ are truly random, then $\widetilde{H}_{k^*}$ and H_{k^*} behave identically; on the other hand, if the inputs $\widetilde{r}_i$ are generated by $G_{n\to m+2n}$, then $\widetilde{H}_{k^*}$ and H_{k^*-1} behave identically. By the choice of k^*, we know that $\widehat{\mathsf{D}}$ distinguishes H_{k^*} and H_{k^*-1} with non-negligible probability. Hence, $\widehat{\mathsf{D}}_{\mathsf{PRG}}$ distinguishes with (the same) non-negligible probability between the case where the $\widetilde{r}_1,\ldots,\widetilde{r}_T$ are random and the case where they are generated by $G_{n\to m+2n}$. By Lemma 3.1, this contradicts that $G_{n\to m+2n}$ is a PRG. Therefore, our initial supposition was false: that is, there cannot exist a $\widehat{\mathsf{D}}$ which distinguishes between $\mathsf{lookup}^G_{n,m}(\rho_0,\cdot)$ and $\mathcal{R}$ with non-negligible probability. The result follows. □

3.1.1 Looking up random values in order.

We would like to look up the random values $\mathsf{lookup}^G_{n,m}(\rho_0,\cdot)$ *in order*, that is, first r_1, then r_2, and so on. This can be done more efficiently than by traversing the tree starting at the root for each new value, essentially by storing the path to the most recently retrieved leaf, and implementing a "next leaf" function which takes the stored path as an input. Naturally, this incurs additional (logarithmic) storage cost, compared to looking up each leaf starting afresh from the root.

An algorithm to find the next leaf in a binary tree given the path to the "current" leaf is given in Algorithm 1. The description given in Algorithm 1 is recursive, for clarity of exposition. Note that in practice, there is a more efficient implementation that avoids recursion. The method returns the entire path to the next leaf, rather than just the leaf node, because the path must be passed into the next invocation of the method to obtain the following leaf.

Lemma 3.3. *For any given depth d, when Algorithm 1 is used to compute all the leaves of a complete binary tree (of depth d) in order, the* `leftChild()` *and* `rightChild()` *methods are called exactly once for each non-leaf node in the tree. More precisely, the method employed is the following: in order to obtain the first leaf, the standard method traversing the tree downwards from the root is*

[8] This can be done by running $\widehat{\mathsf{D}}$ polynomially many times on the oracles $H_0,\ldots,H_k$, and taking the adjacent pair (k^*-1,k^*) for which there were the most differences in output.

```
Path pathToNextLeaf(int depth,
                    Path currentPath, int currentLeafNum) {
  if (depth = 1) {
    Leaf nextLeaf = currentPath.root.rightChild();
    return currentPath.removeEndNode().append(nextLeaf);
  } else if (depth > 1) {
    if (currentLeafNum is even) {
      Leaf nextLeaf = currentPath.endNode.rightChild();
      return currentPath.removeEndNode().append(nextLeaf);
    } else {
      Path pathToParent = currentPath.removeEndNode();
      int parentLeafNum = floor(currentLeafNum/2);
      Path pathToNextParent =
        pathToNextLeaf(depth-1, pathToParent, parentLeafNum);
      return pathToNextParent.append(
        pathToNextParent.endNode.leftChild());
    }
  }
}
```

Algorithm 1: Recursive method to find next leaf from path to current leaf

used; and then subsequent leaves are obtained in order by calling `pathToLeaf1 = nextLeafPath(d, pathToLeaf0, 0)`*, then* `pathToLeaf2 = nextLeafPath(d, pathToLeaf1, 1)`*, etc.*

Proof. Given in Appendix C. □

In order to apply this method to our lookup tree, we observe that the non-leaf nodes of the lookup tree constitute a binary tree (shown in blue in Figure 2). We show that applying Algorithm 1 to this binary tree allows in-order retrieval of the first k leaf values in the look-up tree in time $O(k)$, for any $k \in \mathbb{N}$. The storage requirement is $\log(k) \cdot \log(n)$ where n is the number of input bits to G.

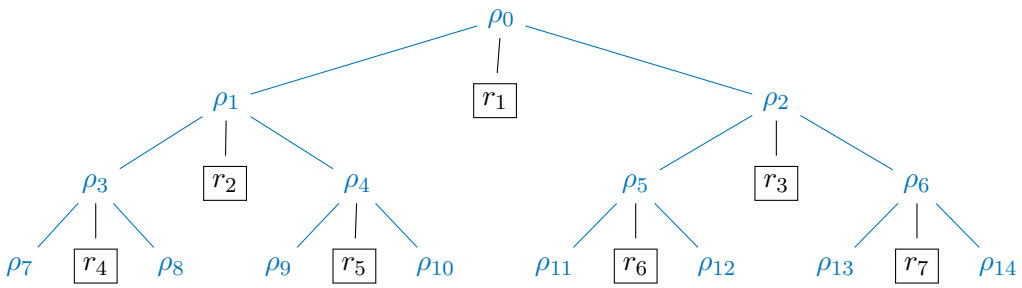

Figure 2: A binary tree within the lookup tree

Lemma 3.4. *For any given depth $d \geq 1$, when all the output values (boxed nodes) at depth d of the lookup tree are computed in order (from left to right) by:*

- *first, computing the leftmost output value by traversing the tree downwards from the root,*
- *then, computing each subsequent output value by applying Algorithm 1 to the binary tree of non-leaf nodes up to and including depth $d-1$, and calling the underlying PRG to obtain each actual (leaf) output value,*

the total number C of calls to the underlying PRG G that is required to compute all *the output values at depth d is exactly $2^d + 2^{d-1} - 2$. (Note that in order to use Algorithm 1, the path to the most recently retrieved output node must be stored at all times: this incurs logarithmic storage space.)*

Proof. By the construction of the lookup tree, `leftChild()` and `rightChild()` can each be implemented by a single invocation of the PRG G. By Lemma 3.3, when Algorithm 1 is used to look up all nodes at a particular depth $d-1$ in a binary tree (in order), the `leftChild()` and `rightChild()` methods will each be called exactly once for every node at depth less than $d-1$ in the tree. Thus, G will be called twice for every node at depth less than $d-1$ in the binary tree. There are $2^{d-1}-1$ such nodes, so G will be called a total of $2^d - 2$ times while traversing the tree. In addition, there is one invocation of G per output value, which we have not counted in the above analysis, because it is not within the *binary* tree. There are 2^{d-1} output values at depth d of the lookup tree, so this adds 2^{d-1} invocations to our total. Hence, the total number of calls of G required to compute all the output values at depth d (in order) is $2^d + 2^{d-1} - 2$. □

Corollary 3.5. *When computing the values $\mathsf{lookup}^G_{n,m}(\rho_0, i)$ for $i = 1, 2, \ldots$ (in order) by the method described in Lemma 3.4, the amortized number of calls to G per output value looked up is constant. To be precise, it is less than* 1.5*.*

Proof. For any given depth $d > 1$, there are 2^{d-1} output values at that depth. By Lemma 3.4, all of these values can be looked up with a total of $2^d + 2^{d-1} - 2$ calls to G. Hence, the number of calls to G per output value (at depth d) is $\frac{2^d + 2^{d-1} - 2}{2^d} < 1.5$. □

3.2 Concurrent man-in-the-middle secure protocol

The protocol construction which follows can be realized with black-box access to any PRG, and achieves concurrent man-in-the-middle security.

Public parameters. PRG G, security parameter $n \in \mathbb{Z}$, error-correcting code C with constant-fraction distance.
Key generation. $\mathsf{Gen}(1^n)$ samples $s, s' \leftarrow \{0,1\}^n$ and outputs secret key (s, s').
Initial state. The prover's state consists of $i \in \mathbb{N}$ initialised to 1.

$\mathcal{P}(s, s'; i)$		$\mathcal{V}(s, s')$
	$\xleftarrow{a}$	$a \leftarrow R$
$e := \mathsf{lookup}^G_{n,n}(s', i)$ $z := C.\mathsf{Enc}(a) * s + e$ $i := i + 1$	$\xrightarrow{z,\ i}$	accept iff $z + C.\mathsf{Enc}(a) * s = \mathsf{lookup}^G_{n,n}(s', i)$

Protocol 1: Concurrent MIM secure protocol

The blue color in the protocol indicates (updating of) the prover's state.

Lemma 3.6. *Protocol 1 is perfectly complete.*

Proof. This is clear since $\mathsf{lookup}^G_{n,n}$ is deterministic. □

We first prove that Protocol 1 is actively secure, which serves as a stepping-stone to the proof of concurrent MIM security.

Lemma 3.7. *Protocol 1 is secure against active attacks.*

Proof. Let e_j denote the noise string for index j. Consider the following games:

Game 1. $\mathcal{P}, \mathcal{V}$ and the adversary $\mathcal{A}$ play the active security game.

Game 2. $\mathcal{P}, \mathcal{V}$ and $\mathcal{A}$ play the active security game as before, except that $\mathcal{P}, \mathcal{V}$ no longer know s', but instead have oracle access to $\mathsf{lookup}^G_{n,n}(s', \cdot)$.

Game 3. Like Game 2, but $\mathsf{lookup}^G_{n,n}(s', \cdot)$ is replaced by a random oracle.

Games 1 and 2 are perfectly indistinguishable for the adversary, since the messages sent by are distributed identically in the two games. Suppose, for contradiction, that there exists an adversary $\mathcal{A}$ which can efficiently distinguish between Games 2 and 3. Then, this adversary could be used to efficiently distinguish between (oracle access to) $\mathsf{lookup}^G_{n,n}(s', \cdot)$ and a random oracle – this contradicts Theorem 3.2. Therefore, Games 1, 2, and 3 are computationally indistinguishable, and so the e_j are indistinguishable from uniformly random noise.

We have established that the prover's message $z = C.\mathsf{Enc}(a) * s + e_j$ is indistinguishable from $C.\mathsf{Enc}(a) * s + r$ for random r. Hence, z is indistinguishable from random to any active adversary, regardless of the choice of a. It remains only to consider the interaction of $\mathcal{A}$ with the jonest verifier $\mathcal{V}$. Given a challenge a from $\mathcal{V}$, $\mathcal{A}$ can have at most negligible advantage at guessing the (unique) value of z that $\mathcal{V}$ will accept, as shown by considering the following two cases:

1. $\mathcal{A}$ sends an index i that was not used when talking to the honest prover. In this case, we could give the adversary the e values for this i for free (as it is independent of the what happens for the other indices). Now the adversary's task is equivalent to guessing $C.\mathsf{Enc}(a) * s$, which he cannot do since he has no information about s.

2. $\mathcal{A}$ sends an index i that was previously used in a query to the prover. Let z, i be the response (to a) from the honest prover. Say the honest verifier sends a' and let z', i be the adversary's response. If there is a non-negligible probability that z' is accepted, then it follows that $z - z' = (C.\mathsf{Enc}(a) - C.\mathsf{Enc}(a')) * s$. This happens with negligible probability since all of a, a', z, z' were chosen independently of s.

□

Theorem 3.8. *Protocol 1 is secure against concurrent MIM attacks.*

Proof. We show that if there is an adversary $\mathcal{A}$ which achieves a certain advantage when conducting a concurrent MIM attack, then there is another adversary $\mathcal{A}'$ that *only talks to the prover* in Protocol 1 and achieves essentially the same advantage. First, we replace the honest verifier by a fake verifier $\mathcal{V}'$ who has no access to s or the e_j but still gives essentially the same answers as $\mathcal{V}$. Then we argue that for any concurrent MIM attack, there is an equally successful *active* attack, and finally refer to Lemma 3.7 for the active security of the protocol.

$\mathcal{V}'$ works as follows: when queried by $\mathcal{A}$, it chooses a random challenge a (just like $\mathcal{V}$ does). When $\mathcal{A}$ returns an answer z, j, there are two cases to consider:

1. $\mathcal{A}$ previously received answer z', j from $\mathcal{P}$, where $z' = C.\mathsf{Enc}(a') * s + e_j$ and a' is $\mathcal{A}$'s query to $\mathcal{P}$. Here we have two possibilities:

   (a) $a = a'$ (which could be the case if $\mathcal{A}$ queried $\mathcal{P}$ during the current protocol execution): in this case, if $z = z'$, $\mathcal{V}'$ accepts; else, it rejects.

   (b) $a \neq a'$: $\mathcal{V}'$ always rejects.

2. $\mathcal{A}$ never previously received an answer of the form z', j from $\mathcal{P}$. In this case $\mathcal{V}'$ always rejects.

Consider the first time $\mathcal{A}$ queries the verifier. $\mathcal{V}$'s challenge is distributed identically to that of $\mathcal{V}'$. In case 1a, $\mathcal{V}$ will accept if and only if z has the correct value $C.\mathsf{Enc}(a) * s + e_j$: so $\mathcal{V}'$ always makes the same decision as $\mathcal{V}$. In case 1b, V only accepts if $z = C.\mathsf{Enc}(a) * s + e_j$, but since

$z' = C.\mathsf{Enc}(a') * s + e_j$ it must be that $(z - z') = (C.\mathsf{Enc}(a) - C.\mathsf{Enc}(a')) * s$. This happens with negligible probability because s is random and z, z', a, a' are all independent of s: $\mathcal{P}$'s responses, including z', are independent of s; and since this is the first query, $\mathcal{V}$ has not seen s yet, so a, a' and z must be independent of s too. Thus, $\mathcal{V}$ rejects with overwhelming probability, so $\mathcal{V}'$ is statistically close to the right behavior. Finally, in case 2, no one sees e_j before $\mathcal{A}$ produces z, j. If $\mathcal{V}$ accepts, we have $z = C.\mathsf{Enc}(a) * s + e_j$, so $e_j = z - C.\mathsf{Enc}(a) * s$, which happens with negligible probability since z, a and s are independent of e_j.

Therefore, we can replace $\mathcal{V}$ with $\mathcal{V}'$ for the first query, and $\mathcal{A}$'s advantage changes at most negligibly as a result. Repeating this argument for all the queries, we reach the game where $\mathcal{V}$ is entirely replaced by $\mathcal{V}'$, and $\mathcal{A}$'s advantage is still at most negligibly different from in the original game. Since $\mathcal{V}'$ does not possess any secret information, an adversary can run $\mathcal{V}'$ "in his head". So for any adversary $\mathcal{A}$ which has non-negligible advantage in a man-in-the-middle attack, we can construct an adversary $\mathcal{A}'$ that emulates both $\mathcal{A}$ and $\mathcal{V}'$ "in his head" and achieves the same advantage, but conducting an *active* attack (since he need not interact with the real verifier $\mathcal{V}$). The result then follows from Lemma 3.7. □

3.2.1 Linear-time implementation.

The prover in Protocol 1 can run in time $O(n)$ and space $O(\log(n) \cdot \log(k))$, where k is the number of protocol executions run so far[9]: this is possible by using Algorithm 1 to compute $\mathsf{lookup}^G_{n,n}(\cdot,\cdot)$ as described in Lemma 3.4. This follows from Corollary 3.5, and the fact that there exist linear-time, linear-stretch PRGs and linear-time encodable codes with constant-factor expansion and large constant-fraction distance (such as [GI05]).

The verifier can also be implemented to run in linear time *for honest executions*, by using the same method as the prover to compute $\mathsf{lookup}^G_{n,n}(\cdot,\cdot)$. Clearly, if the prover is honest, the verifier will run in linear time. If the prover cheats and breaks the sequence, then the verifier can retrieve the required $\mathsf{lookup}^G_{n,n}(\cdot,\cdot)$ value by the "backup method" of traversing the lookup tree downwards from the root, which takes $O(n \cdot \log(k))$ time instead. This implementation requires $N \cdot O(\log(n) \cdot \log(k))$ space, where N is the number of different provers with which the verifier interacts. Note that since the multiplication can be done in depth $O(\log(n))$, if the PRG G is of poly-logarithmic depth, then the verifier does only poly-logarithmic depth computation (even when the prover cheats).

3.3 ℓ-instance concurrent MIM secure protocol

Building upon the ideas of Protocol 1, our next protocol achieves ℓ-instance concurrent MIM security for any polynomial ℓ. Moreover, if ℓ is constant, we can still get (amortised) linear time. The next protocol makes use of ℓ-wise independent hashing, which is defined below.

Definition 3.9 (ℓ-wise independent hash function family)**.** *A function family $\mathcal{H}$ of functions that map n bits to m bits is a* ℓ-wise independent hash function family *if for all $y_1, \ldots, y_\ell \in \{0,1\}^m$ and for all distinct $x_1, \ldots, x_\ell \in \{0,1\}^n$, it holds that $\Pr_{h \leftarrow \mathcal{H}}[h(x_1) = y_1 \wedge h(x_2) = y_2 \wedge \cdots \wedge h(x_\ell) = y_\ell] = 2^{-\ell m}$.*

Lemma 3.10. *Protocol 2 is perfectly complete.*

Proof. This is clear since $\mathsf{lookup}^G_{n,(\ell+1)\cdot n+\beta}$ is deterministic. □

Our security proofs follow a similar structure to those of Protocol 1: we first prove ℓ-instance active security, then use this to prove ℓ-instance concurrent MIM security. The proofs of Lemma

[9] In other words, k is the number of leaf values in the lookup tree that have been retrieved so far. Note that if desired, the value of k can be upper-bounded by some polynomial-size K, by "starting a new tree" after K values have been retrieved from the initial tree: the $(K+1)^{th}$ leaf value in the first tree serves as the root of a new tree in which subsequent lookups are done. This technique was suggested in [Gol01].

Public parameters. $\ell = \mathsf{poly}(n)$, PRG G, security parameter $n \in \mathbb{Z}$, error-correcting code C with constant-fraction distance, function family $\mathcal{H} = \{h_r\}_{r\in\{0,1\}^\beta}$ of 2ℓ-wise independent hash functions mapping $(\ell+1)\cdot n$ bits to n bits.

Key generation. $\mathsf{Gen}(1^n)$ samples $s \leftarrow R, s' \leftarrow \{0,1\}^n$ and outputs (s, s').

Initial state. The prover's state consists of $i \in \mathbb{N}$ initialised to 1.

$\mathcal{P}(s, s'; i)$		$\mathcal{V}(s, s')$
	$\xleftarrow{\quad a \quad}$	$a \leftarrow R$
$r_i := \mathsf{lookup}^G_{n,(\ell+1)\cdot n+\beta}(s', i)$		$r_i := \mathsf{lookup}^G_{n,(\ell+1)\cdot n+\beta}(s', i)$
$e := h_{r_i^{[1,\beta]}}\left(a + r_i^{[\beta+1,(\ell+1)\cdot n+\beta]}\right)$		accept iff $z + C.\mathsf{Enc}(a) * s =$
$z := C.\mathsf{Enc}(a) * s + e$	$\xrightarrow{\quad z,\ i \quad}$	$h_{r_i^{[1,\beta]}}\left(a + r_i^{[\beta+1,(\ell+1)\cdot n+\beta]}\right)$
$i := i + 1$		

Protocol 2: ℓ-instance concurrent MIM secure protocol

3.11 and Theorem 3.12 are given in Appendix D, due to space constraints. We remark that the proof of Theorem 3.12 is very similar to that of Theorem 3.8

Lemma 3.11. *Protocol 2 is secure against ℓ-instance active attacks.*

Theorem 3.12. *Protocol 2 is secure against ℓ-instance concurrent MIM attacks.*

3.3.1 Linear-time implementation.

If ℓ is constant, then the prover in Protocol 2 can run in (amortised) time $O(n)$ and space $O(\log(n)\cdot \log(k))$, where k is the number of protocol executions run so far: as with Protocol 1, this requires the use of Algorithm 1 to compute $\mathsf{lookup}^G_{n,n}(\cdot,\cdot)$, and the use of a linear-time PRG and linear-time encodable code. In addition, we require an ℓ-wise independent hash function family whose functions can be sampled and computed in linear time. A construction of a hash function family satisfying these properties for constant ℓ is given in Section E. As in the case of Protocol 1, the verifier in Protocol 2 can also be implemented to run in linear time when the prover is honest.

3.3.2 On achieving (unbounded) Multi-instance MIM security.

Consider the PRG G as a mapping from n-bit seeds to n-bit outputs. Then clearly the string $G(s_1), ..., G(s_t)$ is pseudorandom for any polynomial k, if the s_i's are independent and random. We say that G is *strongly $\mathcal{H}, \ell$-wise secure* if this still holds, even if the s_i are not independent, but are chosen from an ℓ-wise independent distribution generated by a hash function from ℓ-wise independent family $\mathcal{H}$ -= more precisely, any subset of ℓ seeds are uniformly random and independent, and $s_i = h(x_i)$ for h chosen at random from $\mathcal{H}$ and distinct x_i. We can now define a small change to Protocol 2: namely, in Step 2 of the Prover's computation, e is computed as

$$e := G\left(h_{r_i^{[1,\beta]}}\left(a + r_i^{[\beta+1,(\ell+1)\cdot n+\beta]}\right)\right).$$

(That is, we do the same as before, but apply G at the end.) Now, assuming G is strongly $\mathcal{H}, \ell$-wise secure, where $\mathcal{H}$ is the hash function family used in the protocol, we will get pseudorandom output after applying G, no matter how (polynomially) many outputs we generate, As a result this variant of the protocol is Multi-instance MIM secure (with an unbounded number of instances), if G is

strongly $\mathcal{H}, \ell$-wise secure. If G has this property even for a constant ℓ, the protocol can run in linear time, by using the ℓ-wise independent hash functions described in Appendix E.

We emphasise that this security notion for a PRG is non-standard and its plausibility depends very much on how the concrete PRG and hash function family relate to each other. It should therefore not be used without a careful analysis of the building blocks.

References

[App13] Benny Applebaum. "Pseudorandom generators with long stretch and low locality from random local one-way functions". In: *SIAM Journal on Computing* 42.5 (2013), pp. 2008–2037.

[BPR12] Abhishek Banerjee, Chris Peikert, and Alon Rosen. "Pseudorandom Functions and Lattices". In: *EUROCRYPT*. Ed. by David Pointcheval and Thomas Johansson. Vol. 7237. Lecture Notes in Computer Science. Springer, 2012, pp. 719–737. ISBN: 978-3-642-29010-7. DOI: 10.1007/978-3-642-29011-4. URL: http://dx.doi.org/10.1007/978-3-642-29011-4.

[BR93] Mihir Bellare and Phillip Rogaway. "Entity Authentication and Key Distribution". In: *Advances in Cryptology - CRYPTO '93, 13th Annual International Cryptology Conference, Santa Barbara, California, USA, August 22-26, 1993, Proceedings*. Ed. by Douglas R. Stinson. Vol. 773. Lecture Notes in Computer Science. Springer, 1993, pp. 232–249. ISBN: 3-540-57766-1. DOI: 10.1007/3-540-48329-2_21. URL: http://dx.doi.org/10.1007/3-540-48329-2_21.

[BDOZ11] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. "Semi-homomorphic Encryption and Multiparty Computation". In: *EUROCRYPT*. Ed. by Kenneth G. Paterson. Vol. 6632. Lecture Notes in Computer Science. Springer, 2011, pp. 169–188. ISBN: 978-3-642-20464-7.

[Can+00] Ran Canetti, Yevgeniy Dodis, Shai Halevi, Eyal Kushilevitz, and Amit Sahai. "Exposure-Resilient Functions and All-or-Nothing Transforms". In: *Advances in Cryptology - EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 14-18, 2000, Proceeding*. Ed. by Bart Preneel. Vol. 1807. Lecture Notes in Computer Science. Springer, 2000, pp. 453–469. ISBN: 3-540-67517-5. DOI: 10.1007/3-540-45539-6_33. URL: http://dx.doi.org/10.1007/3-540-45539-6_33.

[Cho+85] Benny Chor, Oded Goldreich, Johan Håstad, Joel Friedman, Steven Rudich, and Roman Smolensky. "The Bit Extraction Problem of t-Resilient Functions (Preliminary Version)". In: *26th Annual Symposium on Foundations of Computer Science, Portland, Oregon, USA, 21-23 October 1985*. IEEE Computer Society, 1985, pp. 396–407. ISBN: 0-8186-0644-4. DOI: 10.1109/SFCS.1985.55. URL: http://dx.doi.org/10.1109/SFCS.1985.55.

[DFMV13] Ivan Damgård, Sebastian Faust, Pratyay Mukherjee, and Daniele Venturi. "Tamper Resilient Cryptography Without Self-Destruct". In: *IACR Cryptology ePrint Archive* 2013 (2013), p. 124.

[DZ13] Ivan Damgrd and Sarah Zakarias. "Constant-Overhead Secure Computation of Boolean Circuits using Preprocessing". English. In: *Theory of Cryptography*. Ed. by Amit Sahai. Vol. 7785. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 621–641. ISBN: 978-3-642-36593-5. DOI: 10.1007/978-3-642-36594-2_35. URL: http://dx.doi.org/10.1007/978-3-642-36594-2_35.

[DKPW12] Yevgeniy Dodis, Eike Kiltz, Krzysztof Pietrzak, and Daniel Wichs. "Message Authentication, Revisited". In: *EUROCRYPT*. Ed. by David Pointcheval and Thomas Johansson. Vol. 7237. Lecture Notes in Computer Science. Springer, 2012, pp. 355–374. ISBN: 978-3-642-29010-7. DOI: 10.1007/978-3-642-29011-4. URL: http://dx.doi.org/10.1007/978-3-642-29011-4.

[Gol01] Oded Goldreich. *Foundations of Cryptography: Basic Techniques*. Cambridge University Press, 2001. ISBN: 0-521-79172-3.

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. "How to construct random functions". In: *J. ACM* 33.4 (1986), pp. 792–807.

[GI05] Venkatesan Guruswami and Piotr Indyk. "Linear-time encodable/decodable codes with near-optimal rate". In: *IEEE Transactions on Information Theory* 51.10 (2005), pp. 3393–3400. DOI: 10.1109/TIT.2005.855587. URL: http://dx.doi.org/10.1109/TIT.2005.855587.

[Hey+12] Stefan Heyse, Eike Kiltz, Vadim Lyubashevsky, Christof Paar, and Krzysztof Pietrzak. "Lapin: An Efficient Authentication Protocol Based on Ring-LPN". In: *FSE*. Ed. by Anne Canteaut. Vol. 7549. Lecture Notes in Computer Science. Springer, 2012, pp. 346–365. ISBN: 978-3-642-34046-8.

[HB01] NicholasJ. Hopper and Manuel Blum. "Secure Human Identification Protocols". English. In: *Advances in Cryptology ASIACRYPT 2001*. Ed. by Colin Boyd. Vol. 2248. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, pp. 52–66. ISBN: 978-3-540-42987-6. DOI: 10.1007/3-540-45682-1_4. URL: http://dx.doi.org/10.1007/3-540-45682-1_4.

[IKOS08] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. "Cryptography with constant computational overhead". In: *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17-20, 2008*. Ed. by Cynthia Dwork. ACM, 2008, pp. 433–442. ISBN: 978-1-60558-047-0. DOI: 10.1145/1374376.1374438. URL: http://doi.acm.org/10.1145/1374376.1374438.

[JW05] Ari Juels and Stephen A. Weis. "Authenticating Pervasive Devices with Human Protocols". In: *CRYPTO*. Ed. by Victor Shoup. Vol. 3621. Lecture Notes in Computer Science. Springer, 2005, pp. 293–308. ISBN: 3-540-28114-2.

[KSS10] Jonathan Katz, Ji Sun Shin, and Adam Smith. "Parallel and Concurrent Security of the HB and HB$^+$ Protocols". In: *J. Cryptology* 23.3 (2010), pp. 402–421.

[Kil+11] Eike Kiltz, Krzysztof Pietrzak, David Cash, Abhishek Jain, and Daniele Venturi. "Efficient Authentication from Hard Learning Problems". In: *EUROCRYPT*. Ed. by Kenneth G. Paterson. Vol. 6632. Lecture Notes in Computer Science. Springer, 2011, pp. 7–26. ISBN: 978-3-642-20464-7.

[LM13] Vadim Lyubashevsky and Daniel Masny. "Man-in-the-Middle Secure Authentication Schemes from LPN and Weak PRFs". English. In: *Advances in Cryptology CRYPTO 2013*. Ed. by Ran Canetti and JuanA. Garay. Vol. 8043. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 308–325. ISBN: 978-3-642-40083-4. DOI: 10.1007/978-3-642-40084-1_18. URL: http://dx.doi.org/10.1007/978-3-642-40084-1_18.

[MNT90] Y. Mansour, N. Nisan, and P. Tiwari. "The Computational Complexity of Universal Hashing". In: *Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing*. STOC '90. Baltimore, Maryland, USA: ACM, 1990, pp. 235–243. ISBN: 0-89791-361-2. DOI: 10.1145/100216.100246. URL: http://doi.acm.org/10.1145/100216.100246.

[Pat11] Kenneth G. Paterson, ed. *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings.* Vol. 6632. Lecture Notes in Computer Science. Springer, 2011. ISBN: 978-3-642-20464-7.

[PJ12] David Pointcheval and Thomas Johansson, eds. *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings.* Vol. 7237. Lecture Notes in Computer Science. Springer, 2012. ISBN: 978-3-642-29010-7. DOI: 10.1007/978-3-642-29011-4. URL: http://dx.doi.org/10.1007/978-3-642-29011-4.

[VZ12] Salil Vadhan and Colin Jia Zheng. "Characterizing pseudoentropy and simplifying pseudorandom generator constructions". In: *Proceedings of the 44th symposium on Theory of Computing.* ACM. 2012, pp. 817–836.

A Pseudorandom primitives

We give the standard definitions of pseudorandom generators (PRGs) and pseudorandom function families (PRFs).

Definition A.1 (Pseudorandom generator)**.** *Let $G : \{0,1\}^n \to \{0,1\}^{m(n)}$ be a deterministic polynomial-time algorithm. G is a* pseudorandom generator *(PRG) if $m(n) > n$ and for any efficient distinguisher D that outputs a single bit, it holds that $|\Pr[\mathsf{D}(r) = 1] - \Pr[\mathsf{D}(G(s)) = 1]| \leq \mathsf{negl}(n)$, where $r \leftarrow \{0,1\}^{m(n)}, s \leftarrow \{0,1\}^n$ are chosen uniformly at random, and the probabilities are taken over r, s, and the random coins of D.*

It is well known that any pseudorandom generator implies pseudorandom generation with any polynomial expansion factor $m(n)$, by applying the PRG to its own output repeatedly.

Definition A.2 (Pseudorandom function family (PRF))**.** *Let $\mathcal{F} = \{F_K\}$ be family of deterministic polynomial-time keyed algorithms mapping n bits to m bits. $\mathcal{F}$ is a* pseudorandom function family (PRF)*, if for any efficient distinguisher D that outputs a single bit, it holds that $|Pr[\mathsf{D}^{F_K} = 1] - \Pr[\mathsf{D}^{\mathcal{R}_{n\to m}} = 1]| \leq \mathsf{negl}(n)$, where $\mathcal{R}_{n\to m}$ is a random oracle mapping n bits to m bits, and the probabilities are taken over the random coins of D and the key K which is randomly chosen.*

B Formal specification of hybrid $\widetilde{H}_{k^*}$

In this section we give a formal description of the algorithm $\widetilde{H}_{k^*}$ used in the proof of Theorem 3.2. $\widetilde{H}_{k^*}$ takes as input $(\widetilde{r}_1, \ldots, \widetilde{r}_T)$, and then behaves exactly like H_{k^*}, except in the following aspects:

- when $\widetilde{H}_{k^*}$ is initialised, it sets a variable $\mathsf{next} := 0$; and
- if $\lfloor \log_2(i) \rfloor + 1 \geq k^*$ (that is, the depth of the output node for query i is at least k^*) then $\widetilde{H}_{k^*}$ first stores the three tuples

$$(\mathsf{leaf}, \ell, (\widetilde{r}_{\mathsf{next}})^{[1,m]}),$$
$$(\mathsf{root}, \alpha_0, (\widetilde{r}_{\mathsf{next}})^{[m+1,m+n]}),$$
$$(\mathsf{root}, \alpha_0 + 1, (\widetilde{r}_{\mathsf{next}})^{[m+n+1,m+2n]}),$$

  where α_0, ℓ are defined by

$$\alpha_0 = \begin{cases} 2 \cdot (i - 2^{\lfloor \log_2(i) \rfloor} - 1) & \text{if } \lfloor \log_2(i) \rfloor + 1 = k^* \\ 2 \cdot \lfloor \alpha_{(i,k^*)}/2 \rfloor & \text{otherwise} \end{cases},$$

$$\ell = \begin{cases} i & \text{if } \lfloor \log_2(i) \rfloor + 1 = k^* \\ 2^{(k^*-1)} + (\alpha_0/2) & \text{otherwise} \end{cases};$$

then $\widetilde{H}_{k^*}$ increments next by 1, and outputs ρ_i, defined by:

$$\rho_i = \begin{cases} (\widetilde{r}_{\mathsf{next}})^{[1,m]} & \text{if } \lfloor \log_2(i) \rfloor + 1 = k^* \\ \mathsf{lookup}^G_{n,m}\left((\widetilde{r}_{\mathsf{next}})^{[m+1,m+n]}, \gamma_{(i,j^*)}\right) & \text{if } \alpha_{(i,k^*)} = \alpha_0 \\ \mathsf{lookup}^G_{n,m}\left((\widetilde{r}_{\mathsf{next}})^{[m+n+1,m+2n]}, \gamma_{(i,j^*)}\right) & \text{if } \alpha_{(i,k^*)} = \alpha_0 + 1 \end{cases} .$$

In terms of the tree representation of the pseudorandom look-up function: $\widetilde{H}_{k^*}$ behaves exactly like H_{k^*}, except that the values associated with nodes at depth k^* are taken from the input values $\widetilde{r}_1, \ldots, \widetilde{r}_T$. Note that although the number of nodes at depth k^* may be greater than T, next can never become greater than T during an execution of $\widehat{\mathsf{D}}_{\mathsf{PRG}}$, because $\widehat{\mathsf{D}}$ cannot make more than T queries: therefore, $\widetilde{r}_{\mathsf{next}}$ is always well-defined.

C Ordered traversal of leaves of a binary tree

Lemma 3.3. For any given depth d, when Algorithm 1 is used to compute all the leaves of a complete binary tree (of depth d) in order, the `leftChild()` and `rightChild()` methods are called exactly once for each non-leaf node in the tree. More precisely, the method employed is the following: in order to obtain the first leaf, the standard method traversing the tree downwards from the root is used; and then subsequent leaves are obtained in order by calling `pathToLeaf1 = nextLeafPath(d, pathToLeaf0, 0)`, then `pathToLeaf2 = nextLeafPath(d, pathToLeaf1, 1)`, etc.

Proof. In order to obtain the the first leaf node (and the path thereto), we need to call `leftChild()` exactly once on all nodes along that path. When $d = 1$, this means that `leftChild()` is called on the root node when obtaining the first leaf node. Moreover, when $d = 1$, it is clear (from lines 2-4) that `rightChild()` is called on the root node exactly once (when obtaining the second leaf node). Hence, the lemma holds for $d = 1$.

For $d > 1$, we argue by induction. It is sufficient to prove that for any $d > 1$:

1. `pathToNextLeaf` is called[10] exactly once for every node at depth $d-1$, except the final node at depth $d-1$ (since for that node, there is no next leaf); and

2. `leftChild()` and `rightChild()` are called exactly once on each node at depth $d-1$.

We call `nextLeafPath()` once for each `currentLeafNum` $\in \{0, \ldots, 2^{\texttt{depth}} - 2\}$ (from the statement of the lemma). In particular, `nextLeafPath()` is called once for the left child of each node at depth `depth`-1 (these nodes are exactly those for which `currentLeafNum` is even). From lines 6-8, it follows that for every node at depth `depth`-1, the `rightChild()` method is called exactly once at line 7.

The recursive call to `pathToNextLeaf()` occurs on line 12, and is only executed when `currentLeafNum` is odd (by the if-clause on lines 6-14). To be precise, the odd values of `currentLeafNum` for which we run `pathToNextLeaf()` are $\{1, 3, \ldots, 2^{\texttt{depth}} - 3\}$. This corresponds exactly to the set of right-children of nodes at depth `depth`-1, except the last one. We see on line 12 that the path passed into the recursive call is `pathToParent`, the path to the parent of the current node. Thus, `pathToNextLeaf()` is recursively called exactly once for each node at depth `depth`-1, except the last one. This satisfies condition 1.

Finally, for each path (of depth `depth`-1) which is *returned* by a recursive call to `pathToNextLeaf()`, `leftChild()` is called on the end node of the path (on line 13). Since we have already established that `pathToNextLeaf()` is recursively called exactly once for each node at depth `depth`-1, except

[10] To be precise, when we write "`pathToNextLeaf` is called for a given node" we mean that `pathToNextLeaf(depth, path, leafNum)` is called, where `depth` is the depth of the node in the tree, `path` is the path from the root to that node, and `leafNum` is the number of the node when counting the nodes at depth `depth` from left to right.

the last node, and the functionality of `pathToNextLeaf()` is to return the next node at a given depth, it follows that the paths returned by such recursive calls to `pathToNextLeaf()` lead to each node at depth `depth`-1, except the *first* node. We conclude that `leftChild()` is called exactly once for each node at depth `depth`-1, except the first node. Moreover, `leftChild()` is called on the first node at depth `depth`-1 when we initially retrieve the first leaf. So, `leftChild()` is called exactly once on each node at depth `depth`-1. In conjunction with our earlier observatiosn about calls to `rightChild()`, this means that condition 2 is satisfied. The result follows. □

D Proof of ℓ-instance concurrent MIM security of Protocol 2

In this section, we show the ℓ-instance concurrent MIM security of Protocol 2. We require the following technical lemma, which can be seen as a generalization of the leftover hash lemma and has a similar proof.

Lemma D.1 ([DFMV13])**.** *Let $(X_1, X_2, \ldots, X_\ell) \in \mathcal{X}^\ell$ be ℓ (possibly dependent) random variables such that $H_\infty(X_i) \geq \gamma$ and $(X_1, \ldots, X_\ell)$ are pairwise different. Let $\mathcal{H} = \{h : \mathcal{X} \to \mathcal{Y}\}$ be a family of 2ℓ-wise independent hash functions, with $|\mathcal{Y}| = 2^k$. Then for random $h \leftarrow \mathcal{H}$ we have that the statistical distance satisfies*

$$\Delta((h, h(X_1), h(X_2), \ldots, h(X_\ell)); (h, U_{\mathcal{Y}}^1, \ldots, U_{\mathcal{Y}}^\ell)) \leq \frac{\ell}{2} \cdot 2^{(\ell \cdot k - \gamma)/2},$$

where $U_{\mathcal{Y}}^1, \ldots, U_{\mathcal{Y}}^\ell$ are ℓ independent and uniformly distributed variables.

We now prove two supporting lemmas, before the main theorem.

Lemma 3.11. Protocol 2 is secure against ℓ-instance active attacks.

Proof. Recall that an ℓ-instance active adversary may have concurrent access to up to ℓ honest provers, but as usual, he cannot reset the provers. The updating of the prover's state in Protocol 2 ensures that for any given prover, the value r_i is freshly pseudorandomly sampled for each execution (that is, each value of the counter i). In particular, the hash function seed $r_i^{[1,\beta]}$ is freshly pseudorandomly sampled for each value of i, and this means that for any polynomial number of executions, with overwhelming probablity, all the hash function seeds will be distinct. Therefore, an ℓ-instance active adversary can obtain at most ℓ outputs of the hash function h_s for any given seed s. For any $i \in \mathbb{N}$, let $s \stackrel{\text{def}}{=} r_i^{[1,\beta]}$ be the corresponding seed, and let $x \stackrel{\text{def}}{=} r_i^{[\beta+1,(\ell+1)\cdot n+\beta]}$ be the summand inside the hash function argument. Suppose that the adversary obtains samples $h_s(a_1 + x), \ldots, h_s(a_\ell + x)$ from the honest provers. If the adversary chooses some a_i, a_j to be equal, then the samples $h_s(a_i + x), h_s(a_j + x)$ will also be equal, so the adversary will not gain more information than if he made just one query a_i. Hence, we assume without loss of generality that the $a_1, \ldots, a_\ell$ are distinct.

Since G is a PRG, we can replace s and x with uniformly randomly chosen $s' \leftarrow \{0,1\}^\beta$ and $x' \leftarrow \{0,1\}^{(\ell+1)\cdot n}$, and the adversary's advantage will change at most negligibly. Let $U_{(\ell+1)\cdot n}$ be a random variable that is uniformly distributed over the set of $((\ell+1)\cdot n)$-bit strings. Consider the random variables $U_{(\ell+1)\cdot n} + a_1, U_{(\ell+1)\cdot n} + a_2, \ldots, U_{(\ell+1)\cdot n} + a_\ell$. Each variable $U_{(\ell+1)\cdot n} + a_i$ has entropy $H(U_{(\ell+1)\cdot n} + a_i) = H_\infty(U_{(\ell+1)\cdot n} + a_i) = (\ell+1)\cdot n$. They are not independent, but they are pairwise different because the a_i's are distinct. Therefore, by Lemma D.1,

$$\{h_s, h_s(U_{(\ell+1)\cdot n} + a_1), \ldots, h_S(U_{(\ell+1)\cdot n} + a_\ell)\} \stackrel{s}{\approx} \{h_s, U_n^1, \ldots U_n^\ell\}.$$

(In the notation of the lemma: we have $k = n$ and $\gamma = (\ell+1)\cdot n$, so the distance is $\frac{\ell}{2}2^{-n}$, which is negligible given that $\ell = \mathsf{poly}(n)$.)

Finally, since the seeds $s_i \stackrel{\text{def}}{=} r_i^{[1,\beta]}$ are freshly pseudorandomly sampled for each value of i, the hash functions $h_{s_1}, h_{s_2}, \ldots$ used in the protocol are indistinguishable from independent random

hash functions. Therefore, the output of any hash function h_{s_i} is indistinguishable from random even given the outputs of the other hash functions $h_{s_{i'}}, h_{s_{i''}}, \ldots$ from different executions of the protocol. □

Lemma D.2. *For any two-round authentication protocol in which the verifier sends the first message, and where the verifier's accept/reject decision is a deterministic function of the secret key, the initial message of the verifier, and the prover's response: it holds that any adversary $\mathcal{A}$ with access to multiple honest verifiers $\mathcal{V}_1, \ldots, \mathcal{V}_\ell$ can be perfectly simulated by another adversary $\mathcal{A}'$ with access to only one honest verifier $\mathcal{V}$*

Proof. The simulation works as follows: $\mathcal{A}'$ runs $\mathcal{A}$, and for every protocol session that $\mathcal{A}$ begins with honest verifier $\mathcal{V}_j$, $\mathcal{A}'$ starts a new session with verifier $\mathcal{V}$ and forwards the initial message a of $\mathcal{V}$ to $\mathcal{A}$. Then, when $\mathcal{A}$ returns to the open session with $\mathcal{V}_j$ and sends a response b, $\mathcal{A}'$ returns to the corresponding session with $\mathcal{V}$ and forwards b to $\mathcal{V}$; and finally, $\mathcal{A}'$ returns to $\mathcal{A}$ the accept/reject decision of $\mathcal{V}$. This is a perfect simulation since for any session, the verifier's decision is a deterministic function of the secret key, the initial message a and the (adversarial) prover's response b. □

Theorem 3.12. Protocol 2 is secure against ℓ-instance concurrent MIM attacks.

Proof. We show that given an adversary $\mathcal{A}$ which achieves a certain advantage when conducting an ℓ-instance concurrent MIM attack, it is possible to build a new adversary $\mathcal{A}''$ that *only talks to the ℓ provers* and achieves essentially the same advantage.

By Lemma D.2, $\mathcal{A}$ can be perfectly simulated with access to just one honest verifier, so we assume henceforth that there is only one honest verifier $\mathcal{V}$. Next, we replace the single honest verifier $\mathcal{V}$ by a fake verifier $\mathcal{V}'$ who has no access to s or the e values, but still gives essentially the same answers as $\mathcal{V}$. Then we argue that for any ℓ-instance concurrent man-in-the-middle attack, there is an equally successful ℓ-instance *active* attack, and finally refer to Lemma 3.11 for the ℓ-instance active security of the protocol.

$\mathcal{V}'$ works as follows: when queried by $\mathcal{A}$, it chooses a random challenge a (just like $\mathcal{V}$ does). When $\mathcal{A}$ returns an answer z, j (i.e. the second protocol message), there are two cases to consider:

1. $\mathcal{A}$ previously received answer z', j from $\mathcal{P}$, where $z' = a' \cdot s + e_j$ and a' is $\mathcal{A}$'s query to $\mathcal{P}$. Here we have two possibilities:
   (a) $a = a'$ (which could be the case if $\mathcal{A}$ queried $\mathcal{P}$ during the current protocol execution): in this case, if $z = z'$, $\mathcal{V}'$ accepts; else, it rejects.
   (b) $a \neq a'$: $\mathcal{V}'$ always rejects.
2. $\mathcal{A}$ never previously received an answer of the form z', j from $\mathcal{P}$. In this case $\mathcal{V}'$ always rejects.

Consider the first time $\mathcal{A}$ queries the verifier. The challenge produced by $\mathcal{V}$ has exactly the same distribution as the one $\mathcal{V}'$ outputs. Now, in case 1a, notice that $\mathcal{V}$ will accept if and only if z has the correct value $a \cdot s + e_j$: so $\mathcal{V}'$ always makes the same decision as $\mathcal{V}$. In case 1b, $\mathcal{V}$ rejects except with negligible probability, so $\mathcal{V}'$ is statistically close to the right behavior. This is because accepting would imply that $z = a \cdot s + e_j$, but we also have $z' = a' \cdot s + e_j$ so then $s = (z - z')(a - a')^{-1}$. This happens with negligible probability because s is random and z, z', a, a' are all independent of s. This holds because all of $\mathcal{P}$'s responses (including z') are independent of s. Moreover, since this is the first query, $\mathcal{V}$ has not even looked at s yet, so a, a' and z must be independent of s too. Finally, in case 2, note that no one sees e_j before the adversary produces z, j. If $\mathcal{V}$ accepts, we have $z = a \cdot s + e_j$, so $e_j = z - a \cdot s$, which happens with negligible probability since z, a and s are independent of e_j.

Therefore, we can modify $\mathcal{V}$ so that it behaves like $\mathcal{V}'$ for the first query, and the adversary's advantage changes at most negligibly as a result. Repeating the same argument for all the queries,

we reach the game where $\mathcal{V}$ is entirely replaced by $\mathcal{V}'$, and the adversary's advantage is still at most negligibly different from in the original game.

Since $\mathcal{V}'$ does not possess any secret information, an adversary can run $\mathcal{V}'$ "in his head". So for any adversary $\mathcal{A}$ which has non-negligible advantage in a ℓ-instance concurrent MIM attack, we can construct an adversary $\mathcal{A}''$ that emulates both $\mathcal{A}$ and $\mathcal{V}'$ "in his head" and achieves the same advantage, but conducting an ℓ-instance *active* attack (since he need not interact with the real verifier $\mathcal{V}$).

Finally, the security of the protocol against ℓ-instance concurrent man-in-the-middle attacks follows from the security against ℓ-instance concurrent active attacks, which was shown in Lemma 3.11. □

E Linear-time ℓ-independent hashing

Mansour, Nisan, and Tiwari [MNT90] conjectured that pairwise-independent hash functions cannot be computed in linear time – in particular, that computing them requires $\Omega(n \log(n))$ time. Recently, Ishai et al. [IKOS08] disproved this conjecture, and gave a construction of a linear-time computable pairwise-independent hash function. In this section, we show that the [IKOS08] construction can be extended to achieve ℓ-wise independence for constant $\ell \in \mathbb{N}$.

Notation. For a vector $v \in \{0,1\}^n$ and a subset of indices $S \subseteq [n]$, we write $v_{[S]}$ to denote the $|S|$-bit vector obtained by restricting v to the coordinates in S. For a tuple of indices $t = (t_1, \ldots, t_d) \in [n]^d$, we write $v_{[t]}$ to denote the d-bit vector $(v_{t_1}, \ldots, v_{t_d})$. We write $||$ for vector concatenation.

Exposure resilient functions [Can+00] (also known as deterministic bit-fixing extractors [Cho+85]), are used as a building block in the construction. The definition is given below.

Definition E.1 (Exposure resilient function)**.** *A function $f : \{0,1\}^n \to \{0,1\}^m$ is an λ-*exposure resilient function *if for any $L \subset [n]$ of size $|L| = n - \lambda$, and for $r \leftarrow \{0,1\}^n$ and $R \leftarrow \{0,1\}^m$ chosen uniformly at random, the distributions $(r_{[L]}, f(r))$ and $(r_{[L]}, R)$ are identical.*

Our construction works as follows. For $n \in \mathbb{N}$, there is an ℓ-wise independent hash function family $\mathcal{H}_n^{C,G,E}$ of functions mapping n bits to n bits. Each family is parametrized by the following:

- $C : \{0,1\}^n \to \Sigma^m$ is the encoding algorithm of an error-correcting code with a constant-size alphabet Σ. The code has minimum distance $c = \Theta(n)$ and constant expansion factor (that is, $m = \Theta(n)$).
- G is a ℓ-wise independent hash function family mapping Σ to Σ.
- $E : \Sigma^m \to \{0,1\}^n$ is an λ-exposure resilient function where $\lambda = \Theta(n)$.

Each function in the family $\mathcal{H}_n^{C,G,E} = \{h_g\}_{g \in G^m}$ is indexed by a vector of hash functions $g = (g_1, \ldots, g_m)$, where each g_i is a member of the (smaller) ℓ-wise independent hash function family G. To sample a hash function in $h_g \leftarrow \mathcal{H}_n$, simply sample M small hash functions $g_1, \ldots, g_m \leftarrow G$.

Each hash function h_g is computed as follows.

1. Encode the input $x \in \{0,1\}^n$ to obtain codeword $y = C(x) \in \Sigma^m$.
2. For each $i \in [m]$, let $z_i = g_i(y_i) \in \Sigma$.
3. Let $z \in \Sigma^m$ denote the concatenation of all z_i, that is, $z = z_1||z_2||\ldots||z_m$. The output of the hash function is $E(z)$.

We remark that this procedure for computing the hash function is the same as in the [IKOS08] construction, except that in their work, G is a family of pairwise (rather than ℓ-wise) independent hash functions.

Theorem E.2 (ℓ-wise independence of $\mathcal{H}_n^{C,G,E}$)**.** *Let $\ell \geq 2$ be any constant. For any $m = \Theta(n)$, there exist $\lambda, c = \Theta(n)$ such that if C is an error-correcting code with minimum distance c and codeword length m, and E is a λ-exposure resilient function, then $\mathcal{H}_n^{C,G,E}$ is a family of ℓ-wise independent hash functions.*

Proof. Let $c = \Theta(n)$ be the following:

$$c = m - \frac{2m}{\ell(\ell-1)+1}. \tag{1}$$

Note that the right-hand side is $\Theta(n)$, because $m = \Theta(n)$ and $\ell = O(1)$. Moreover, since $\ell > 2$, inequality 1 implies that $0 < c < m$ as required.

Let $x_1, \ldots, x_\ell$ be any distinct vectors in $\{0,1\}^n$, and let $h_g \leftarrow \mathcal{H}_n$ be a hash function sampled from the family $\mathcal{H}_n^{C,G,E}$. Let the set of corresponding codewords be denoted by $Y = \{C(x_1), \ldots, C(x_\ell)\}$.

Define the *overlap* of two codewords $y, y' \in \Sigma^m$ as the set of positions $k \in [m]$ for which the k^{th} elements are equal: that is, where $y_k = y'_k$. Formally,

$$\mathsf{Overlap}(y, y') = \left\{k \in [m] \ : \ y_k = y'_k\right\}.$$

Building on this, we define the set $\overline{L}$ as follows:

$$\overline{L} = [m] \setminus \bigcup_{y,y' \in Y, y \neq y'} \mathsf{Overlap}(y, y') = \left\{k \in [m] \ : \ \forall y, y' \in Y,\ y_k \neq y'_k\right\}.$$

In other words, $\overline{L}$ is the set of positions k for which the k^{th} elements of of the codewords in Y are pairwise distinct.

Due to the minimum distance of the error-correcting code,

$$|\mathsf{Overlap}(C(x), C(x'))| \leq m - c$$

for all distinct $x, x' \in \{0,1\}^n$. Then, since $|Y| = \ell$:

$$\left|\bigcup_{y,y' \in Y, y \neq y'} \mathsf{Overlap}(y, y')\right| \leq (m-c) \cdot \frac{\ell \cdot (\ell-1)}{2} \tag{2}$$

$$\leq m. \tag{3}$$

Inequality 3 follows by substituting equation 1 into inequality 2. Moreover, the right-hand side of inequality 2 is clearly $\Theta(m)$ since $c = \Theta(m)$ and ℓ is constant. From this, it follows that

$$\left|\overline{L}\right| = m - \left|\bigcup_{y,y' \in Y, y \neq y'} \mathsf{Overlap}(y, y')\right| = \Theta(m) \quad \text{and} \quad \left|\overline{L}\right| > 0. \tag{4}$$

Recall that for each position $k \in \overline{L}$, it holds that the k^{th} elements of of the codewords in Y are pairwise distinct. Then, since g_k is an ℓ-wise independent hash function, the ℓ hash function outputs $g_k(C(x_1)), \ldots, g_k(C(x_\ell))$ will be independent and uniformly distributed. Moreover, since the hash function $g_1, \ldots, g_k$ are chosen independently, the following set consists of independent and uniformly distributed elements:

$$\left\{g_k(C(x_1)), \ldots, g_k(C(x_\ell)) \ : \ k \in \overline{L}\right\}. \tag{5}$$

Recall that when computing $h_g(x_i)$ (in step 3 of the description above), the input to E is the concatenation of the m "small hashes", $g_1(C(x_i))|| \ldots ||g_m(C(x_i))$. Let z_i denote $g_1(C(x_i))|| \ldots ||g_m(C(x_i))$. Define $L = [m] \setminus \overline{L}$ and $\lambda = |L| = \Theta(m)$. For each input $x_i \in \{0,1\}^n$ to the hash function h_g, E is evaluated on an input z_i for which $(z_i)_{[\overline{L}]}$ is distributed independently at random (this follows from 5). Hence, by the λ-exposure resilience of E, the outputs $E(z_1), \ldots, E(z_\ell)$ are distributed independently and randomly. The theorem follows. □

Finally, we show that the hash functions in $\mathcal{H}_n^{C,T,G,E}$ can be computed and sampled in linear time.

Theorem E.3. *If C and E are computable in linear time, then each hash function $h_g \in \mathcal{H}_n^{C,G,E}$ can be computed in linear time. Moreover, sampling a hash function $h_g \leftarrow \mathcal{H}_n^{C,G,E}$ can be done in linear time.*

Proof. Since C is linear-time computable, step 1 is computable in linear time, and has a linear-size output $y \in \{0,1\}^m$. In step 2, many small hashes of the form $g_i(y_i)$ are computed, where $g_i \leftarrow G$. Since the input to the hash function g_i is of constant size, each such evaluation of g_i will take constant time. The total number of small hashes computed is $m = \Theta(n)$ which is linear, so step 2 takes linear time. Finally, since E is computable in linear time, and is evaluated on a linear-size input $z \in \{0,1\}^M$, step 3 also takes linear time.

The sampling of a hash function $h_g \leftarrow \mathcal{H}_n^{C,G,E}$ consists of sampling m hash functions $g_1, \ldots, g_m \leftarrow G$. Since the family G is of constant size, each g_i can be sampled in constant time, and there are linearly many of them, so the whole sampling process takes linear time. □

There are known constructions of linear-time computable functions that satisfy the properties required by C and E (for any constant ℓ):

- Guruswami and Indyk [GI05] construct error correcting codes which have linear-time encoding (and decoding) algorithms, and for any positive constant $\varepsilon < 1$, their construction can achieve minimum distance c such that $c/m = \varepsilon$ with a constant-factor expansion. The encoding function of these codes would be suitable for use as C.
- Ishai et al. [IKOS08] give a construction of an infinite family of λ-exposure resilient functions mapping n bits to m bits, where $\lambda = \Theta(n)$ and $m = \Theta(n)$.