Constrained Pseudorandom Functions from Homomorphic Secret Sharing

Geoffroy Couteau[1], Pierre Meyer[1,2], Alain Passelègue[3,4], and Mahshid Riahinia[4]

[1] Université Paris Cité, CNRS, IRIF, Paris, France.
couteau@irif.fr
[2] Reichman University, Herzliya, Israel.
pierre.meyer@irif.fr
[3] Inria, France.
alain.passelegue@inria.fr
[4] ENS de Lyon, Laboratoire LIP (U. Lyon, CNRS, ENSL, Inria, UCBL), France.
mahshid.riahinia@ens-lyon.fr

Abstract. We propose and analyze a simple strategy for constructing 1-key constrained pseudorandom functions (CPRFs) from homomorphic secret sharing. In the process, we obtain the following contributions. First, we identify desirable properties for the underlying HSS scheme for our strategy to work. Second, we show that (most of) recent existing HSS schemes satisfy these properties, leading to instantiations of CPRFs for various constraints and from various assumptions. Notably, we obtain the first (1-key selectively secure, private) CPRFs for inner-product and (1-key selectively secure) CPRFs for NC^1 from the DCR assumption, and more. Lastly, we revisit two applications of HSS, equipped with these additional properties, to secure computation: we obtain secure computation in the silent preprocessing model with one party being able to precompute its whole preprocessing material before even knowing the other party, and we construct one-sided statistically secure computation with sublinear communication for restricted forms of computation.

1 Introduction

Since their introduction in [29], pseudorandom functions (PRFs) have played a central role in modern cryptography and numerous extensions have been proposed. Of particular interest is the notion of constrained pseudorandom functions (CPRFs), introduced concurrently in [6,34,11]. Recall that a PRF is a family of keyed functions $\{F_k\}_{k\in\mathcal{K}} : \mathcal{X} \to \mathcal{Y}$ such that the input-output behavior of any randomly selected F_k should be computationally indistinguishable from that of a truly random function with same domain and range (without any knowledge of k). Constrained pseudorandom functions for a class of constraints $\mathcal{C}$ extend PRFs by allowing to delegate partial evaluation keys ck_C for any $C : \mathcal{X} \to \{0,1\} \in \mathcal{C}$, termed constrained keys, generated from the master secret key k as $\mathsf{ck}_C \leftarrow \mathsf{Constrain}(k, C)$. A partial key allows to compute $F_k(x)$ for any input x such that $C(x) = 0$, by running a constrained evaluation algorithm $\mathsf{CEval}(\mathsf{ck}_C, x)$, while preserving pseudorandomness of evaluations on inputs x satisfying $C(x) = 1$[5]. A constrained PRF can further be private, or constraint-hiding, if a constrained key hides the constraint C. Significant efforts have been made to obtain CPRFs for broad classes of constraints from various assumptions in the recent years [4,15,14,18,3,21,39,31,26]. As of today, CPRFs for simple class of constraints (e.g., point functions or constant-degree CNFs) are known from minimal assumptions (e.g., from one-way functions [29,26]). Yet, constructing CPRFs for broader classes of constraints such as NC^1 has proven notoriously hard. While (private) CPRFs for NC^1 and even $\mathsf{P/poly}$ exist based on the learning with errors assumption (with subexponential modulus-to-noise ratio) [15,14], other families of standard assumptions have so far failed to provide advanced constructions, except for one construction for NC^1 based on an exotic Q-type variant of DDH over the group of quadratic residues modulo a safe prime $q = 2p + 1$, and the DDH assumption [3].

This serious lack of constructions remains when considering simpler classes of constraints such as inner products ($C(x) = 0$ iff $\langle x, y\rangle = 0$ for some fixed vector y), despite the large amount of work on inner-product-based encryption in other contexts (e.g., for attribute-based encryption or functional encryption) and the recent lattice-based CPRF for inner-product [26].

[5] The inverse condition is often used (pseudorandomness if $C(x) = 0$ and partial evaluation if $C(x) = 1$). Our choice slightly simplifies our constructions.

In this work, we draw connections between constrained pseudorandom functions and homomorphic secret sharing (HSS), a notion introduced by Boyle et al. in [10]. One of our main contributions is to construct CPRFs for inner-product as well as for NC^1 via HSS, leading to instantiations from a wide variety of assumptions thanks to the recent developments in HSS [40,37,1]. Before describing in more details our contributions, we briefly remind the definition of HSS. An HSS scheme for a class of functions $\mathcal{F}$ allows to generate a public key pk and two evaluations keys $\mathsf{ek}_0, \mathsf{ek}_1$, such that one can securely share an input x into two shares $(\mathsf{I}_0, \mathsf{I}_1) \leftarrow \mathsf{Input}(\mathsf{pk}, x)$ such that, given one of the two evaluation keys: each share computationally hides x, and it is possible to *homomorphically evaluate* any function $f \in \mathcal{F}$ on the shares of x as $y_b = \mathsf{Eval}(\mathsf{ek}_b, \mathsf{I}_b, f)$, for $b \in \{0, 1\}$. Moreover, the resulting shares satisfy $y_1 - y_0 = f(x)$. Since its introduction, HSS has found numerous applications in cryptography and beyond, and notably for (1) low-communication secure computation [10], and for (2) secure computation with *silent* preprocessing [9,37]. In this work, we also revisit the latter applications. Again, we briefly remind them before diving into the details of our contributions. A long-standing problem in secure computation had been to achieve communication smaller than the circuit size (for rich classes of functions). It was first solved via fully-homomorphic encryption (FHE) [27]. To securely compute a function f on their respective private inputs x and y, Alice and Bob can use the following protocol: Alice sends to Bob an FHE encryption of x, and Bob homomorphically computes an encryption of $f(x, y)$ by evaluating $f(\cdot, y)$. He then sends back the result to Alice who can recover $f(x, y)$ by decrypting. Homomorphic secret sharing leads to another solution to this problem, by first having Alice and Bob compute shares of x and y (which is independent of circuit size) and then locally compute shares of $f(x, y)$.

Regarding secure computation in the preprocessing model, a protocol is split in two phases: a first *preprocessing phase* run ahead-of-time (independently of inputs and function to compute) in which Alice and Bob *jointly* generate long, correlated random strings, and a second *online phase* where the actual secure computation takes place. In the latter phase, the former correlated random strings are consumed by a fast, non-cryptographic, information-theoretic secure computation protocol. Homomorphic secret sharing enables secure computation with *silent* preprocessing: a short one-time interaction allows Alice and Bob to generate short keys, from which they can later *locally* (*i.e.*, without any interaction) stretch arbitrarily long correlated (pseudo-)random strings, which are later used in the online phase. Effectively, this pushes almost all the computational overhead of the preprocessing phase to a purely local computation.

1.1 Our Contributions

In this work, we show how to use homomorphic secret sharing schemes towards constructing constrained pseudorandom functions for rich classes of constraints and from new assumptions. Our main contributions are threefold.

Extending HSS Properties. We identify two natural extensions of homomorphic secret sharing, which we term respectively *homomorphic secret sharing with simulatable memory shares* and *staged homomorphic secret sharing.* At a high level, both notions capture the ability to perform some limited form of *programming* of HSS shares, *i.e.*, to construct one of the two HSS shares of an input x before knowing x. It turns out that most of known HSS constructions already achieve these extensions, leading to constructions based on a wide variety of assumptions.

New Constructions of CPRFs. Combining our extensions of HSS with any standard PRF with evaluation in NC^1 (which is known from every assumption implying HSS), we construct: (1) private CPRFs for inner-product, starting with any HSS with simulatable memory shares with statistical correctness, and (2) CPRFs for NC^1 starting with any staged HSS with statistical correctness. This leads to the following statement.

Theorem 1 (informal). *Assuming any of the following assumptions:*

- *the DCR assumption,*
- *the hardness of the Joye-Libert encryption scheme,*
- *the DDH and DXDH assumptions over class groups,*
- *the Hard Subgroup Membership assumption over class groups,*

- *the LWE assumption with super-polynomial modulus-to-noise ratio,*

there exist (1-key, selectively secure) private CPRFs for inner product, and (1-key, selectively secure) CPRFs for NC^1*.*

Our results significantly expand the set of assumptions known to imply CPRFs for rich classes of constraints. In particular, our CPRF for NC^1 from DCR yields the first construction of a CPRF for a rich class of constraints from a well-founded standard assumption beyond LWE-based constructions.

Revisiting Applications of HSS to Secure Computation. Equipped with our additional properties for HSS, we revisit two standard applications, namely secure computation with silent preprocessing, and secure computation with sublinear communication, and obtain the following results.

Precomputable secure computation with silent preprocessing. As described above, secure computation with silent preprocessing requires a short initial interaction before being able to run the heavy local preprocessing. In particular, the parties need to have decided *who* they will execute a secure computation protocol with. In contrast, we show that using staged HSS allows to build a silent preprocessing protocol where one of the parties (say, Alice) can entirely run the heavy offline computation *before she even knows the identity of Bob* (and in particular, before she interacts with Bob). This means that Alice can, at any point, locally generate (her share of) long pseudorandom correlated strings and store them for later use. Then, when she meets someone she wants to securely compute a function with in the future, she can execute the short, one-time interactive protocol (with little communication and computation), and be done with the preprocessing phase. Of course, the other party still needs to execute the heavy offline computation after their interaction[6]. We call this model secure computation with *precomputable* silent preprocessing; it is especially well suited to a client-server setting, where a weak client (Alice) wants to start the bulk of the computation a long time in advance, whereas the powerful server can run the heavy computation after its interaction with the client.

One-sided statistically secure computation with sublinear communication. A core feature of FHE-based sublinear secure computation is that it achieves *one-sided statistical security* when using an FHE scheme with statistical circuit-privacy, since homomorphic evaluation of $f(\cdot, y)$ leaks *statistically* no information about y beyond $f(x, y)$. In other words, Bob's security in the aforementioned protocol holds unconditionally. One-sided statistical security is a desirable security notion and can be achieved quite easily if we do not require sublinear communication, e.g., by using the seminal GMW protocol [30] with a one-sided statistically secure oblivious transfer [35] (to our knowledge, this was first observed in [20]). Yet, as of today, one-sided statistically secure computation with sublinear communication is *only known from FHE*: all HSS-based constructions inherently achieve only computational security for both parties.

Using staged HSS, we obtain the first non-FHE-based constructions of one-sided statistically secure protocols with sublinear communication. Concretely, we obtain secure computation for any $\log\log$-depth circuits with optimal communication, where x remains statistically hidden, provided that $|x| < |y|/\mathsf{poly}(\lambda)$ (where $\mathsf{poly}(\lambda)$ denotes some fixed polynomial), via a black-box use of staged HSS. We also get secure computation of any layered arithmetic circuit C of size s over a sufficiently large ring $\mathbb{Z}_n$, with sublinear communication $O(s/\log\log s)$ and one-sided statistical security (without any restriction on the statistically protected input size), assuming the Paillier encryption scheme is circular-secure. The latter construction is non-black box and exploits the specific structure of a concrete Paillier-based staged HSS scheme from [37].

2 Technical Overview

2.1 General Strategy

Let us first explain a (partly wrong but insightful) strategy for constructing CPRFs from HSS. Let F denote a pseudorandom function with keyspace $\mathcal{K}$ and domain $\mathcal{X}$, and let $\mathcal{C} : \mathcal{X} \mapsto \{0, 1\}$ be a class of

[6] It is not too hard to see that having *both* parties execute the bulk of the computation prior to interacting (while keeping a non-cryptographic online phase) is impossible.

constraints. Consider an HSS scheme $\mathsf{HSS} = (\mathsf{Setup}, \mathsf{Input}, \mathsf{Eval})$ for a class of programs $\mathcal{P}$ such that it contains all functions $f_x : (k, C) \mapsto C(x) \cdot F_k(x)$, for all $x \in \mathcal{X}$. Then, we consider the following construction.

- $\mathsf{KeyGen}(1^\lambda, C)$: sample a PRF key $K \xleftarrow{\$} \mathcal{K}$. Run $(\mathsf{pk}, \mathsf{ek}_0, \mathsf{ek}_1) \leftarrow \mathsf{Setup}(1^\lambda)$, $(\mathsf{I}_0^k, \mathsf{I}_1^k) \leftarrow \mathsf{Input}(\mathsf{pk}, k)$, and $(\mathsf{I}_0^C, \mathsf{I}_1^C) \leftarrow \mathsf{Input}(\mathsf{pk}, C)$. Set $\mathsf{pp} \leftarrow \mathsf{pk}$ and $\mathsf{msk} \leftarrow (\mathsf{ek}_0, \mathsf{ek}_1, \mathsf{I}_0^k, \mathsf{I}_1^k, \mathsf{I}_0^C, \mathsf{I}_1^C)$.
- $\mathsf{Constrain}(\mathsf{msk}, C)$: parse msk as $(\mathsf{ek}_0, \mathsf{ek}_1, \mathsf{I}_0^k, \mathsf{I}_1^k, \mathsf{I}_0^C, \mathsf{I}_1^C)$ and output $\mathsf{ck}_C \leftarrow (\mathsf{ek}_1, \mathsf{I}_1^k, \mathsf{I}_1^C)$.
- $\mathsf{Eval}(\mathsf{pp}, \mathsf{msk}, x)$: run $y_0 \leftarrow \mathsf{Eval}(0, \mathsf{ek}_0, \mathsf{I}_0^k, \mathsf{I}_0^C, f_x)$ and output y_0.
- $\mathsf{CEval}(\mathsf{pp}, \mathsf{ck}_C, x)$: run $y_1 \leftarrow \mathsf{Eval}(1, \mathsf{ek}_1, \mathsf{I}_1^k, \mathsf{I}_1^C, f_x)$ and output y_1.

By correctness of the HSS scheme, for any input x, we have $y_1 - y_0 = C(x) \cdot F_k(x)$. Therefore, if $C(x) = 0$, $y_1 = y_0$ i.e. the CEval algorithm outputs the same value as the evaluation with msk. Yet, if $C(x) = 1$, $y_1 = y_0 + F_k(x)$ and y_0 is pseudorandom, even given y_1 (and ck_C).

The problem with the above construction is that the master secret key does depend on the constraint C while it should be independent of it[7]. A way around this issue would be to use an HSS scheme with *programmable input shares*, *i.e.*, a scheme where I_0^C can be generated before knowing C, and the second share I_1^C can be constructed afterwards from I_0^C and C, when the constraint is chosen. Unfortunately, the only known constructions of HSS with such a strong programmability feature rely on powerful primitives such as threshold FHE. As FHE-style constructions of CPRFs for all circuits are already known, this would defeat the purpose of obtaining constructions based on new assumptions. In this work, we identify weaker properties which still suffice to instantiate the above template, yet are achieved by most of known HSS constructions.

2.2 CPRF from HSS with Simulatable Memory Shares

As a start, we propose a first simple solution to circumvent the lack of programmability. This first property already allows to handle simple forms of constraints such as inner-product, and follows from the common design of HSS constructions. We start by providing a high-level description of HSS schemes, which applies to essentially all known HSS constructions (beside FHE-based constructions).

HSS schemes rely on an additively homomorphic encryption scheme with some form of linear decryption. The public key of the HSS scheme is the public key pk of the underlying encryption scheme, and evaluation keys $\mathsf{ek}_0, \mathsf{ek}_1$ are additive shares of the underlying secret key s. A scheme uses two types of data: (1) **Input shares** $(\mathsf{I}_0, \mathsf{I}_1)$ which are generated by running $\mathsf{Input}(\mathsf{pk}, x)$ on some input x and consist in an encryption of $(x, x \cdot s)$, and (2) **Memory shares** (M_0, M_1) which are typically additive shares of $(x, x \cdot s)$ over $\mathbb{Z}$. Two types of operations are handled: **Additions of memory shares** (simply add the shares as $(x, x \cdot s) + (y, y \cdot s) = (x + y, (x + y) \cdot s)$), and a restricted form of **Multiplication**. Specifically, multiplication can only be performed between an *input share* of some value x and a *memory share* of some value y, and returns a *memory share* of their product $x \cdot y$. Typically, multiplication uses the memory share $(y, y \cdot s)$ to "linearly multiply-and-decrypt" the encryption of $(x, x \cdot s)$, getting some encoding of $(xy, xy \cdot s)$. Then, the encoding is converted into a valid memory share using a specific procedure, which depends on the concrete scheme and is often a form of *distributed discrete logarithm.* We provide more details about multiplication later. Note that one can transform any input share into a memory share of the same value by multiplying it with a memory share of 1. At the end of a computation, each party recovers a memory value consisting in an additive share of $(z, z \cdot s)$, and therefore a share of the result z by dropping the second part. One can evaluate any polynomial-size program following the above restrictions, which precisely corresponds to *restricted multiplication straight-line* (RMS) programs, and encompasses branching programs, NC^1, and more.

HSS with simulatable memory shares. Our starting point is the result of two observations. First, we observe that any HSS following the above structure does in fact allow for a limited form of programming regarding memory values. Indeed, while input shares include a homomorphic encryption of the input (which cannot be generated without knowing the input), *memory shares* are simply

[7] If the key could depend on C, one could just generate two independent PRF keys k_0, k_1 and define the evaluation as $F_{k_{C(x)}}(x)$. Revealing k_0 then allows to compute the evaluation on any x such that $C(x) = 0$ and reveals nothing about the key k_1 used when $C(x) = 1$.

additive shares. Thus, we can always *simulate* a memory share of one party before knowing the value to share, by generating a first random share u. The other share is later set to $x - u$ when the actual value x to share is known.

Second, we remark that two parties sharing input shares of some values $(x_1, \ldots, x_n)$ as well as memory shares of a value z can compute memory shares of $z \cdot P(x_1, \ldots, x_n)$ for any RMS program P. The trick is to evaluate all the operations of P "with z in front", i.e. by maintaining as an invariant that any memory share for any value y that should be used in the computation is replaced by a memory share for the value $z \cdot y$. This invariant being preserved by the two RMS operations (addition and multiplication), it is sufficient to guarantee that every memory value satisfies it when created. This is simply done by transforming an input x into a memory value by multiplying it with the memory share of z in order to get a memory share for $z \cdot x$ rather than for x.

CPRF for Linear Constraints. Combining these two observations leads to constructions of constrained PRFs for linear constraints (and in particular for inner-product). Looking back to the construction aforementioned, we just would like to be able to generate I_0^C, the share of C used for evaluation with the master secret key, without knowing the constraint C in advance. We do it by replacing I_0^C by a simulated *memory share* M_0 of the (yet unknown) constraint C. The constrained key for C is then computed from M_0 and C to generate the appropriate memory share M_1 (i.e. setting M_1 such that $\mathsf{M}_0 + \mathsf{M}_1 = C$).

While this prevents the need for knowing the constraint ahead of time, this comes with a price: we now get a memory share of C rather than an input share, which reduces the set of functions one can evaluate. Still, thanks to our second observation, having a memory share of C and an input share of k allows to compute shares of $C \cdot P(k)$ for any RMS program P. Moreover, given memory shares of multiple C_i's, one can then compute any linear combination of shares $C_i \cdot P(k)$, by summing the latter additive shares. Notably, this allows computing shares of $\langle C, x\rangle \cdot F_k(x)$ as long as the function $k \mapsto F_k(x)$ is an RMS program (assuming F is in NC^1 is sufficient for that purpose).

We just constructed constrained pseudorandom functions for inner-product from any assumption that suffices to construct an HSS scheme for RMS programs satisfying the above conditions. For example, using the recent HSS scheme of [37] yields a CPRF for inner products over $\mathbb{Z}$ (or any integer ring) under the DCR assumption (which also implies PRFs in NC^1). The construction extends immediately to any constant-degree polynomial constraints (by memory-sharing all the coefficients of C). It achieves 1-key selective security, as well as *constraint privacy*. To the best of our knowledge, this is the first construction of (1-key, selective, private) CPRF for inner products that does not rely on LWE.

Security analysis proceeds through a sequence of hybrid games. Recall that the adversary is given a constrained key ck_C of its choice, and access to an evaluation oracle $\mathsf{Eval}(\mathsf{pp}, \mathsf{msk}, \cdot)$. We first modify the evaluation oracle to return $C(x) \cdot F_K(x) + \mathsf{CEval}(\mathsf{pp}, k_C, x)$ on query x. By correctness of the HSS, the adversary's view remains identical to its view in the previous game though the game no longer relies in msk (and in particular now only relies on the evaluation key ek_1 from ck_C). This let us replace the input share I_1 of k in ck_C by an input share of a dummy value, thanks to HSS security. Then, the adversary does no longer have any information about k except in the evaluations, and we can use PRF security to replace evaluations of $F_K(\cdot)$ by truly random values, therefore proving pseudorandomness. Constraint privacy is proven in a similar fashion.

2.3 Handling more Constraints via Staged HSS

While the above already offers enough flexibility to evaluate linear functions (and extensions thereof, such as low-degree polynomials), we still cannot handle general computations like NC^1 circuits. To overcome this limitation, we show by a deeper analysis of known HSS schemes that most of them also achieve some specific, limited form of programmability, which turns out to be sufficient to construct CPRFs for all RMS programs (hence in particular for NC^1).

Concretely, for a vector $\mathbf{u} = (u_1, \ldots, u_\ell)$, our core observation is that it is possible to share $\mathbf{u}$ between parties P_0 and P_1 with two alternate sharing algorithms $(\overline{\mathsf{Input}}_0, \overline{\mathsf{Input}}_1)$ such that: (1) P_0's share of $\mathbf{u}$, obtained from $\overline{\mathsf{Input}}_0$, is *independent* of $\mathbf{u}$ (and can be generated without $\mathbf{u}$), (2) P_0 and P_1 can use specific $\overline{\mathsf{Eval}}_0, \overline{\mathsf{Eval}}_1$ evaluation algorithms to produce memory shares of $P(\mathbf{u})$ for any RMS program P, *provided that* P_1 *knows* $\mathbf{u}$ *in the clear*. We call staged-HSS an HSS scheme satisfying the latter properties, as it intuitively allows to split share generation and evaluation in

2 stages: a first *input-independent* stage, corresponding to P_0's view, and a second *input-dependent* stage corresponding to P_1's view.

At first sight, staged-HSS might not seem particularly useful: if P_1 knows $\mathbf{u}$ in the clear, then P_1 can already compute $P(\mathbf{u})$ for any RMS program P. The key observation is that P_0 and P_1 get *memory shares* of $P(\mathbf{u})$, and not just $P(\mathbf{u})$. This memory share can then be combined with the prior observations to let P_0, P_1 compute additive shares of $P(\mathbf{u}) \cdot Q(\mathbf{v})$, for any other RMS program P, Q, given *input shares* of $\mathbf{v}$. Setting $\mathbf{u}$ to be the description of the constraint C, P to be a universal circuit (with input x hardwired) which on input C returns $C(x)$, $\mathbf{v}$ to be a PRF key k, and Q to be the RMS program (with x hardwired) which on input k returns $F_k(x)$, parties P_0 and P_1 can then compute shares of $C(x) \cdot F_k(x)$, with shares of P_0 being independent of C. We can then instantiate our simple aforementioned strategy for constructing CPRFs while circumventing the need for C during KeyGen. As a result, we obtain (1-key selective) CPRFs for RMS programs (and therefore for NC^1) from any staged-HSS, i.e. from a wide variety of assumptions (including DCR [37,40], class groups assumptions, or variants of QR [1,19], and more.). The security analysis is similar to our construction for inner-product, though this new construction is no longer constraint-hiding, since the CEval algorithm now relies on knowing C (i.e. $\mathbf{u}$ above) in clear.

It remains to explain why known HSS schemes are also staged-HSS schemes. To illustrate this, we use the simple ElGamal-based HSS scheme from [10][8]. We assume basic knowledge of ElGamal encryption in what follows. This scheme follows the general structure detailed above by instantiating the additively homomorphic encryption scheme with ElGamal encryption. That is, an *input share* for x is an ElGamal encryption of the pair $(x, x \cdot s)$[9], i.e. a tuple $(c_0, c'_0, c_1, c'_1) = (g^{r_0}, h^{r_0} \cdot g^x, g^{r_1}, h^{r_1} \cdot g^{x \cdot s})$ with $s \in \mathbb{Z}_p$ being the secret key, $h = g^s$ being the public key, and $r_0, r_1 \stackrel{\$}{\leftarrow} \mathbb{Z}_p$ encryption randomness[10].

Multiplication between an input share (c_0, c'_0, c_1, c'_1) of x and a memory share $(\alpha_\sigma, \beta_\sigma)$ of y (which is just an additive share of $(y, y \cdot s)$ over $\mathbb{Z}_p$ owned by party P_σ) is done as follows. First, party P_σ computes $g_\sigma \leftarrow (c'_0)^{\alpha_\sigma} / c_0^{\beta_\sigma}$. Observe that $g_0 \cdot g_1 = (c'_0)^{\alpha_0 + \alpha_1} / c_0^{\beta_0 + \beta_1} = (g^{sr} \cdot g^x)^y / (g^r)^{sy} = g^{xy}$. Hence, parties get *multiplicative shares* g_0, g_1 of g^{xy}. Doing the same with c_1, c'_1 allows to get multiplicative shares of $g^{xy \cdot s}$. Then, an operation termed *distributed discrete logarithm* allows to transform these multiplicative shares of $(g^{xy}, g^{xy \cdot s})$ into additive shares of $(xy, xy \cdot s)$, i.e. memory shares for the value xy, as desired. Despite being at the core of HSS constructions, the details of the distributed discrete logarithm procedure do not matter here. The only important observation is that the $c_i = g^{r_i}$ components of input shares are independent of the input x; only the c'_i components actually depend on x. Furthermore, in the multiplication above, the only place where c'_i is involved is in the computation of $g_\sigma \leftarrow (c'_i)^{\alpha_\sigma} / c_i^{\beta_\sigma}$. Now, assume that one of the parties, say, P_1, already knows y in the clear: in this case, one can simply define $\alpha_1 \leftarrow y$ and $\alpha_0 \leftarrow 0$, which form valid additive shares of y. But now, P_0 does no longer need to know c'_i components either, since we now have $g_0 = 1/(c_i)^{\beta_0}$.

2.4 Applications of Staged HSS to Secure Computation

From a different angle, staged HSS allows Alice and Bob, respectively owning private inputs x and y, to securely retrieve, given shares of their joint input (x, y), additive shares of $f(x) \cdot g(y)$ for any RMS programs f, g, and even of any $P(x, y) = \sum_{i=1}^{m} f_i(x) \cdot g_i(y)$, where the (f_i, g_i) are RMS programs since additive shares can be added.

Secure computation with precomputable silent preprocessing. In this setting, the goal of the preprocessing phase is to securely distribute *correlated randomness* of a particular form (e.g., random oblivious transfers, vector-OLE, batch-OLE, Beaver triples, authenticated Beaver triples, etc.) which can be seen as special cases of the following general additive correlation: Alice receives random vectors $(\mathbf{r}^A, \mathbf{s}^A)$ and Bob receives random vectors $(\mathbf{r}^B, \mathbf{s}^B)$, such that $\mathbf{s}^A$ and $\mathbf{s}^B$ form additive shares of the tuple $\mathbf{s} = (Q_1(\mathbf{r}^A, \mathbf{r}^B), \cdots, Q_m(\mathbf{r}^A, \mathbf{r}^B))$, where $Q_1, \ldots, Q_m$ are public low-degree polynomials. To *silently* distribute such (pseudorandom) correlations, Alice and Bob can use a generic secure computation protocol to distribute HSS shares of two PRF keys (k_A, k_B) sampled by Alice and Bob

[8] This scheme does not yield CPRFs as it does not achieve statistical correctness, but staged-HSS is easily illustrated with it.

[9] Actually of x and $x \cdot s_i$'s for each bit s_i of s.

[10] s is encrypted bit-by-bit in the actual construction.

respectively. Then, Alice locally defines $\mathbf{r}^A \leftarrow (F_{k_A}(1), \cdots, F_{k_A}(n))$, and Bob does the same with F_{k_B}. Both of them also compute their share $\mathbf{s}^A$ and $\mathbf{s}^B$ by homomorphically evaluating the program P_i for $i \leq m$ with their share of (k_A, k_B), where P_i is defined as:

$$P_i : (k_A, k_B) \to Q_i((F_{k_A}(1), \cdots, F_{k_A}(n)), (F_{k_B}(1), \cdots, F_{k_B}(n))) \ .$$

Note that, as long as F is in NC^1 and Q_i is a constant-degree polynomial, P_i remains in NC^1. We now observe that when Q_i is a constant-degree polynomial, the program P_i can always be (publicly) rewritten as

$$P_i(k_A, k_B) = \sum_{j=1}^{M} \alpha_j \cdot \prod_{i \in S_A^j} F_{k_A}(i) \cdot \prod_{i \in S_B^j} F_{k_B}(i) = \sum_{j=1}^{M} f_j(k_A) \cdot g_j(k_B) \ ,$$

where S_A^i, S_B^i are public subsets of $[n]$, by writing Q_i in algebraic normal form and separating the component of each monomial depending on whether they are computed using k_A or k_B. Above, each of the f_j, g_j functions belong to NC^1. Therefore, P_i belongs to the class of programs supported by our staged HSS construction. Furthermore, Bob always knows his input k_B in the clear. Therefore, using staged HSS, Alice can generate the HSS shares of k_A together with the *input-independent* share of k_B, and she can locally compute $(\mathbf{r}^A, \mathbf{s}^A)$ entirely from these shares, using the staged evaluation algorithm, and later execute a short interactive update protocol with Bob (with communication and computation *independent* of n and m) to let Bob (with input k_B) obtain the full HSS shares of (k_A, k_B). Therefore, Alice can entirely compute all of her preprocessing material *before she even interacts with Bob* (or knows his identity).

Sublinear secure computation with one-sided statistical security. Our last application follows the exact same line as above, further noting that evaluation of $F(x, y) = \sum_i f_i(x) \cdot g_i(y)$ can be performed while statistically protecting one of the two inputs (e.g., x). Moreover, the class of such functions $F(x, y)$ contains in particular all arithmetic circuits (with fan-in 2) of size s and depth $\log \log s$, as in such circuits, every output bit depends on at most $\log s$ inputs, and can therefore be written as a multivariate polynomial in the inputs, with at most s monomials. As a consequence, if there is a secure computation protocol for generating staged HSS shares of inputs x and y with communication $c(|x|, |y|)$, then there exists a protocol for securely computing all circuits of size s and depth $\log \log s$ with $|x| + |y|$ inputs and m outputs with communication $c(|x|, |y|) + 2m$, which is asymptotically optimal. It only remains to find a protocol to securely distribute staged HSS shares with linear communication.

This is not easily done in general, as the standard technique to generate HSS shares with low communication uses *hybrid encryption*: to share an input x, one generate HSS shares of some seed seed (using a generic secure computation protocol), and publishes $x \oplus \mathsf{PRG}(\mathsf{seed})$. Then, the homomorphic evaluation first computes $\mathsf{PRG}(\mathsf{seed})$, unmaskes x, and then applies the function. The issue is that this is inherently incompatible with having (one-sided) statistical security. We describe two cases where we can get around this issue:

1. The first way is to use hybrid encryption only on y, for which we just aim to computational security, and to share x using the standard staged HSS sharing algorithm. This yields a one-sided statistically secure protocol for all $\log\log$-depth circuits with communication $|y| + |x| \cdot \mathsf{poly}(\lambda) + O(m)$, which is optimal as soon as $|x| < |y|/\mathsf{poly}(\lambda)$. In other terms, if the input to be statistically protected is polynomially smaller than the other input, we achieve optimal communication.
2. Our second solution relies on a specific construction of staged HSS scheme that relies on the circular security of the Paillier-ElGamal encryption scheme. Here, we manage to leverage the inherent compactness of this specific scheme to get a protocol with optimal communication $|y| + |x| + O(m)$ for arithmetic circuits over a sufficiently large ring (since Paillier encryption is compact only when the values are from a large ring), by designing a tailored low-communication HSS share distribution protocol. By breaking the circuit into $\log\log$-depth blocks, this generalizes naturally to a one-sided statistically secure protocol with *sublinear* communication $O(s/\log\log s)$ for any layered arithmetic circuits[11] over a sufficiently large field.

[11] An arithmetic circuit is layered if its nodes can be partitioned into layers, such that any wire connects adjacent layers.

3 Preliminaries

We use λ to denote the security parameter. For a natural integer $n \in \mathbb{N}$, the set $\{0, 1, \ldots, n-1\}$ is denoted by $[n]$. We mostly use bold lowercase letters (e.g., $\mathbf{r}$) to denote vectors. For a vector $\mathbf{r} = (r_1, \ldots, r_n)$, the vector $(g^{r_1}, \ldots, g^{r_n})$ is sometimes denoted by $g^{\mathbf{r}}$. We write $\mathsf{poly}(\lambda)$ to denote an arbitrary polynomial function. We denote by $\mathsf{negl}(\lambda)$ a negligible function in λ, and PPT stands for probabilistic polynomial-time. For a finite set S, we write $x \stackrel{\$}{\leftarrow} S$ to denote that x is sampled uniformly at random from S. For an algorithm $\mathcal{A}$, we denote by $y \leftarrow \mathcal{A}(x)$ the output y after running $\mathcal{A}$ on input x.

We recall the notion of constrained pseudorandom functions. For simplicity, we focus on selective, 1-key secure, constraint-hiding, constrained pseudorandom functions, which are the main focus of our work, and refer the reader to [6,34,11,5] for the general definitions. Additional definitions related to our assumptions or applications to multi-party computation (MPC), and in particular definition of pseudorandom correlation functions, can be found in the supplementary material, Section A.

Definition 1 (Constrained Pseudorandom Functions). Denote by λ a security parameter. A Constrained Pseudorandom Function (CPRF) with domain $\mathcal{X} = \{\mathcal{X}_\lambda\}_{\lambda \in \mathbb{N}}$, key space $\mathcal{K} = \{\mathcal{K}_\lambda\}_{\lambda \in \mathbb{N}}$, and range $\mathcal{Y} = \{\mathcal{Y}_\lambda\}_{\lambda \in \mathbb{N}}$, that supports a class of circuits $\mathcal{C} = \{\mathcal{C}_\lambda\}_{\lambda \in \mathbb{N}}$, where each C_λ has domain $\mathcal{X}_\lambda$ and range $\{0, 1\}$, consists of the following four algorithms:[12]

- $\mathsf{KeyGen}(1^\lambda) \to (\mathsf{pp}, \mathsf{msk})$: On input the security parameter λ, the master key generation algorithm outputs a public parameter pp and a master secret key $\mathsf{msk} \in \mathcal{K}$.
- $\mathsf{Eval}(\mathsf{pp}, \mathsf{msk}, x) \to y$: On input the public parameter pp, the master secret key msk, and an input $x \in \mathcal{X}$, the evaluation algorithm outputs a value $y \in \mathcal{Y}$.
- $\mathsf{Constrain}(\mathsf{msk}, C) \to \mathsf{ck}_C$: On input the master secret key msk, and a circuit $C \in \mathcal{C}$, the constrained key generation algorithm outputs a constrained key ck_C.
- $\mathsf{CEval}(\mathsf{pp}, \mathsf{ck}_C, x) \to y$: On input the public parameter pp, a constrained key ck_C, and an input $x \in \mathcal{X}$, the constrained evaluation algorithm outputs a value $y \in \mathcal{Y}$.

Correctness. For any security parameter λ, any constrain $C \in \mathcal{C}$, and any input $x \in \mathcal{X}$ such that $C(x) = 0$, we have:

$$\Pr\left[\mathsf{Eval}(\mathsf{pp}, \mathsf{msk}, x) \neq \mathsf{CEval}(\mathsf{pp}, \mathsf{ck}_C, x) : \begin{array}{l} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda) \\ \mathsf{msk} \leftarrow \mathsf{KeyGen}(\mathsf{pp}) \\ \mathsf{ck}_C \leftarrow \mathsf{Constrain}(\mathsf{msk}, C) \end{array}\right] \leq \mathsf{negl}(\lambda).$$

1-Key Selective Security. We say that a CPRF is 1-key selectively secure if the advantage of any PPT adversary $\mathcal{A}$ in the following game is negligible:

- **Selective Choice of Constraint:** The adversary chooses a (single) circuit $C \in \mathcal{C}$ and sends it to the challenger.
- **Setup:** The challenger runs $(\mathsf{pp}, \mathsf{msk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$, initializes a set $S_{\mathsf{eval}} = \varnothing$, and computes $\mathsf{ck}_C \leftarrow \mathsf{Constrain}(\mathsf{msk}, C)$. The challenger also chooses a random bit $b \stackrel{\$}{\leftarrow} \{0, 1\}$. It sends $\mathsf{pp}, \mathsf{ck}_C$ to $\mathcal{A}$.
- **Pre-Challenge Evaluation Queries:** $\mathcal{A}$ can adaptively send arbitrary input values $x \in \mathcal{X}$ to chall. The challenger computes $y \leftarrow \mathsf{Eval}(\mathsf{pp}, \mathsf{msk}, x)$ and returns y to $\mathcal{A}$. It also updates $S_{\mathsf{eval}} \leftarrow S_{\mathsf{eval}} \cup \{x\}$.
- **Challenge Phase:** $\mathcal{A}$ sends an input $x^* \in \mathcal{X}$ as its challenge query to chall with the restriction that $x^* \notin S_{\mathsf{eval}}$, and $C(x^*) \neq 0$. If $b = 0$, then chall computes $y^* \leftarrow \mathsf{Eval}(\mathsf{pp}, \mathsf{msk}, x^*)$. If $b = 1$, it picks a random value $y^* \stackrel{\$}{\leftarrow} \mathcal{Y}$. Finally, chall returns y^* to $\mathcal{A}$.
- **Post-Challenge Evaluation Queries:** $\mathcal{A}$ continues the queries as before, with the restriction that it cannot query x^* as an evaluation query.
- **Guess:** $\mathcal{A}$ outputs a bit $b' \in \{0, 1\}$.

1-Key Selective Constraint-Hiding. We say that a CPRF is selectively 1-key constraint-hiding if the advantage of any PPT adversary $\mathcal{A}$ in the following game is negligible:

[12] In the remaining of the paper, we drop the λ subscript when it is clear from context.

- **Selective Choice of Constraint:** The adversary chooses a (single) pair of circuits $(C_0, C_1) \in \mathcal{C}$ and sends it to the challenger.
- **Setup:** The challenger runs $(\mathsf{pp}, \mathsf{msk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$, chooses a random bit $b \xleftarrow{\$} \{0,1\}$, and computes $\mathsf{ck}^* \leftarrow \mathsf{Constrain}(\mathsf{msk}, C_b)$. It sends $\mathsf{pp}, \mathsf{ck}^*$ to $\mathcal{A}$.
- **Evaluation Queries:** $\mathcal{A}$ can query evaluations for arbitrary inputs $x \in \mathcal{X}$ to chall, with the restriction that $C_0(x) = C_1(x)$ must hold. The challenger returns $y \leftarrow \mathsf{Eval}(\mathsf{pp}, \mathsf{msk}, x)$ to $\mathcal{A}$.
- **Guess:** $\mathcal{A}$ outputs a bit $b' \in \{0,1\}$.

In both games, $\mathcal{A}$ wins if $b' = b$ and its advantage is defined as $|2 \cdot \Pr[\mathcal{A} \text{ wins}] - 1|$ where the probability is over the internal coins of $\mathcal{A}$ and of Setup.

4 Homomorphic Secret Sharing and Extensions

The core notion underlying our constructions is homomorphic secret sharing (HSS), introduced by Boyle et al. in [10]. In this section, we remind the standard definition of HSS as well as propose several extensions, in particular defining some special properties that play an important role in our constructions. We further remark that these extensions are easily instantiated using the DCR-based HSS construction from [37].

4.1 Homomorphic Secret Sharing

We start by recalling the standard definition of homomorphic secret sharing, as well as of Restricted Multiplication Straight-line (RMS) programs which is the common model of computation in the context of HSS.

Definition 2 (Homomorphic Secret Sharing). Denote by λ a security parameter. A *Homomorphic Secret Sharing* (HSS) scheme for a class of programs $\mathcal{P}$ which is defined over a ring $\mathcal{R}$ and has input space $\mathcal{I} \subseteq \mathcal{R}$ consists of three PPT algorithms $(\mathsf{Setup}, \mathsf{Input}, \mathsf{Eval})$ such that:

- $\mathsf{Setup}(1^\lambda) \to (\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1))$: On input the security parameter λ, the setup algorithm outputs a public key pk and a pair of evaluation keys $(\mathsf{ek}_0, \mathsf{ek}_1)$.
- $\mathsf{Input}(\mathsf{pk}, x) \to (\mathsf{I}_0, \mathsf{I}_1)$: On input the public key pk and an input $x \in \mathcal{I}$, the input algorithm outputs a pair of input information $(\mathsf{I}_0, \mathsf{I}_1)$.
- $\mathsf{Eval}(\sigma, \mathsf{ek}_\sigma, \mathsf{I}_\sigma = (\mathsf{I}_\sigma^{(1)}, \dots, \mathsf{I}_\sigma^{(\rho)}), P) \to y_\sigma$: On input a party index $\sigma \in \{0,1\}$, an evaluation key ek_σ, a vector of ρ input values $(\mathsf{I}_\sigma^{(1)}, \dots, \mathsf{I}_\sigma^{(\rho)})$, and a program $P \in \mathcal{P}$, the evaluation algorithm outputs the party σ's corresponding share of the output y_σ.

We require an HSS scheme to satisfy the following two properties:

- **Correctness.** For any security parameter $\lambda \in \mathbb{N}$, and any program $P \in \mathcal{P}$ with input space $\mathcal{I} \subseteq \mathcal{R}$, we have:
$$\Pr\left[y_0 - y_1 = P(x^{(1)}, \dots, x^{(\rho)})\right] \geq 1 - \mathsf{negl}(\lambda) \ ,$$
where the probability is taken over $(\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \leftarrow \mathsf{Setup}(1^\lambda)$, $(\mathsf{I}_0^{(i)}, \mathsf{I}_1^{(i)}) \leftarrow \mathsf{Input}(\mathsf{pk}, x^{(i)})$ for $i \in [\rho]$, and $y_\sigma \leftarrow \mathsf{Eval}(\sigma, \mathsf{ek}_\sigma, (\mathsf{I}_\sigma^{(1)}, \dots, \mathsf{I}_\sigma^{(\rho)}), P)$, for $\sigma \in \{0,1\}$.
- **Security.** For any PPT adversaries $\mathcal{A}, \mathcal{A}'$, and any bit $\sigma \in \{0,1\}$ the following value should be negligible in λ:
$$\left| \Pr\left[b' = b : \begin{array}{l} (x_0, x_1, \mathsf{state}) \leftarrow \mathcal{A}(1^\lambda) \\ (\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \leftarrow \mathsf{Setup}(1^\lambda) \\ b \xleftarrow{\$} \{0,1\} \\ (\mathsf{I}_0, \mathsf{I}_1) \leftarrow \mathsf{Input}(x_b) \\ b' \leftarrow \mathcal{A}'(\mathsf{state}, \mathsf{pk}, \mathsf{ek}_\sigma, \mathsf{I}_\sigma) \end{array} \right] - \frac{1}{2} \right|$$

We now remind the definition of Restricted Multiplication Straight-line (RMS) programs. RMS programs form a class of programs which encompasses branching programs of polynomial-size and therefore NC^1 circuits. In an RMS program, the multiplication is restricted to happen between an input value and an intermediate value of the computation (so-called "memory" value).

Definition 3 (RMS Programs). *An RMS program with magnitude bound B is defined as a sequence of the instructions as follows:*

- $\mathsf{ConvertInput}(\mathsf{I}^x) \to \mathsf{M}^x$*: Loads an input x into memory.*
- $\mathsf{Add}(\mathsf{M}^x, \mathsf{M}^y) \to \mathsf{M}^{x+y}$*: Adds two memory values.*
- $\mathsf{Mul}(\mathsf{I}^x, \mathsf{M}^y) \to \mathsf{M}^{x \cdot y}$*: Multiplies an input value and a memory value to produce a memory value of their product.*
- $\mathsf{Output}(\mathsf{M}^x, n) \to x \bmod n$*: Outputs a memory value* w.r.t. *a modulus $n < B$.*

4.2 HSS following the RMS Template

Similarly to [9], we first propose a more specific definition for HSS with additional algorithms that are relevant in the context of RMS programs.

Definition 4 (HSS Following the RMS Template). A homomorphic secret sharing scheme $\mathsf{HSS} = (\mathsf{Setup}, \mathsf{Input}, \mathsf{MemGen}, \mathsf{Eval})$ following the RMS template is an HSS scheme as defined in Definition 2 with an additional algorithm MemGen which serves to produce memory values as follows:

- $\mathsf{MemGen}(\sigma, \mathsf{ek}_\sigma, x) \to \mathsf{M}_\sigma$: On input a party index $\sigma \in \{0, 1\}$, an evaluation key ek_σ, and an input $x \in \mathcal{I}$, the memory generator algorithm outputs a memory value M_σ.

Moreover, the Eval algorithm proceeds with sub-routines following the RMS operations $\mathsf{ConvertInput}$, Add, Mul, Output as follows:

- $\mathsf{Eval}(\sigma, \mathsf{ek}_\sigma, (\mathsf{I}_\sigma^{(1)}, \ldots, \mathsf{I}_\sigma^{(\rho)}), P) \to y_\sigma$: On input a party index $\sigma \in \{0, 1\}$, an evaluation key ek_σ, a vector of ρ input values $(\mathsf{I}_\sigma^{(1)}, \ldots, \mathsf{I}_\sigma^{(\rho)})$, and an RMS program P, this algorithm follows the instructions of P and processes them as follows:
  - $\mathsf{ConvertInput}(\sigma, \mathsf{ek}_\sigma, \mathsf{I}_\sigma^x) \to \mathsf{M}_\sigma^x$: This algorithm simply uses the MemGen and Mult algorithms as follows:
    - Run $\mathsf{MemGen}(\sigma, \mathsf{ek}_\sigma, 1) \to \mathsf{M}_\sigma^1$.
    - Run $\mathsf{Mult}(\sigma, \mathsf{ek}_\sigma, \mathsf{I}_\sigma^x, \mathsf{M}_\sigma^1) \to \mathsf{M}_\sigma^x$.
  - $\mathsf{Add}(\sigma, \mathsf{ek}_\sigma, \mathsf{M}^x, \mathsf{M}^y) \to \mathsf{M}^{x+y}$: This algorithm directly adds the given memory values of x and y. Namely, $\mathsf{M}_\sigma^{x+y} = \mathsf{M}_\sigma^x + \mathsf{M}_\sigma^y$.
  - $\mathsf{Mul}(\sigma, \mathsf{ek}_\sigma, \mathsf{I}^x, \mathsf{M}^y) \to \mathsf{M}^{x \cdot y}$: It multiplies an input value I_x and a memory value M_y and outputs a memory value of $x \cdot y$. The template does not impose any non-black box requirement on this algorithm.
  - $\mathsf{Output}(\sigma, \mathsf{M}^x, n) \to x \bmod n$: It uses M^x to output $x_\sigma \bmod n$.

Correctness and security properties are defined as in Definition 2, and we further require the following property:
Additively Homomorphic Memory. The memory values generated in HSS should be additively homomorphic. Meaning that for any two $x, y \in \mathcal{I}$ and any party index $\sigma \in \{0, 1\}$, it holds that

$$\mathsf{M}_\sigma^x + \mathsf{M}_\sigma^y = \mathsf{M}_\sigma^{x+y} \ ,$$

where $\mathsf{M}_\sigma^z \leftarrow \mathsf{MemGen}(\sigma, \mathsf{ek}_\sigma, z)$, for $z \in \{x, y\}$, and $(\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \leftarrow \mathsf{Setup}(1^\lambda)$. Throughout this work, we may refer to memory values satisfying this property as "valid" memory values.

4.3 Extended Evaluation and Simulatable Memory Values

Any HSS following the RMS template as defined above satisfies the following lemma, which states that one can evaluate share of $z \cdot P(x^{(1)}, \ldots, x^{(\rho)})$ using only a memory value of z (instead of an input value) together with the input values of the rest of variables $(x^{(1)}, \ldots, x^{(\rho)})$. This lemma plays a central role in our CPRF constructions.

Lemma 1. *Let* $\mathsf{HSS} = (\mathsf{Setup}, \mathsf{Input}, \mathsf{MemGen}, \mathsf{Eval})$ *be an HSS scheme following the RMS template. There exists an extended evaluation algorithm* $\mathsf{ExtEval}$*:*

- $\mathsf{ExtEval}(\sigma, \mathsf{ek}_\sigma, \mathsf{M}_\sigma, (\mathsf{I}_\sigma^{(1)}, \ldots, \mathsf{I}_\sigma^{(\rho)}), P) \to y_\sigma$*: On input a party index $\sigma \in \{0, 1\}$, an evaluation key ek_σ, a single memory value M_σ, a vector of ρ input values $(\mathsf{I}_\sigma^{(1)}, \ldots, \mathsf{I}_\sigma^{(\rho)})$, and an RMS program P, return a value y_σ such that the following holds.*

For any security parameter $\lambda \in \mathbb{N}$ *and any RMS program* P*, we have:*

$$\Pr\left[y_0 - y_1 = z \cdot P(x^{(1)}, \ldots, x^{(\rho)})\right] \geq 1 - \mathsf{negl}(\lambda) \ , \tag{1}$$

where the probability is taken of the choice of $(\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \leftarrow \mathsf{Setup}(1^\lambda)$*,* $(\mathsf{I}_0^{(i)}, \mathsf{I}_1^{(i)}) \leftarrow \mathsf{Input}(\mathsf{pk}, x^{(i)})$*,* $\mathsf{M}_\sigma \leftarrow \mathsf{MemGen}(\sigma, \mathsf{ek}_\sigma, z)$*, and* $y_\sigma \leftarrow \mathsf{ExtEval}(\sigma, \mathsf{ek}_\sigma, \mathsf{M}_\sigma, (\mathsf{I}_\sigma^{(1)}, \ldots, \mathsf{I}_\sigma^{(\rho)}), P)$*, for* $\sigma \in \{0,1\}, i \in [\rho]$*.*

The proof of the above lemma is detailed in the supplementary material in Section B. It essentially consists in recursively incorporating the memory value M_σ using the standard Eval algorithm by first multiplying inputs with it.

We now introduce an additional property termed *simulatable memory values*. Here, we require that for an input $x \in \mathcal{I}$, the memory value of one of the two parties can be generated ahead of time and without the knowledge of x using a simulation algorithm, while the other memory value can be generated given the pre-computed first memory value and the exact value of x. This simulation should not affect the correctness of $\mathsf{ExtEval}$.

Definition 5 (HSS with Simulatable Memory Values). Let $\mathsf{HSS} = (\mathsf{Setup}, \mathsf{Input}, \mathsf{MemGen}, \mathsf{Eval})$ be an HSS following the RMS template as per Definition 4, with input space $\mathcal{I}$ over the ring $\mathcal{R}$. We say that HSS is simulatable with respect to its memory values if there exist algorithms Sim_0 and Sim_1 such that

- $\mathsf{Sim}_0(1^\lambda) \to \mathsf{M}_0$: on input the security parameter λ outputs a memory value M_0.
- $\mathsf{Sim}_1(\mathsf{M}_0, z, (\mathsf{ek}_0, \mathsf{ek}_1)) \to \mathsf{M}_1$: on input a memory value M_0, an element $z \in \mathcal{I}$, and two encoding keys $(\mathsf{ek}_0, \mathsf{ek}_1)$ outputs a memory value M_1.

We also require the two following properties:
Simulation Correctness. For any $\lambda \in \mathbb{N}$ and any $z \in \mathcal{I}$, the above correctness condition (equation 1) still holds when the memory value is simulated, *i.e.* when we first sample $\mathsf{M}_0 \leftarrow \mathsf{Sim}_0(1^\lambda)$ and then $\mathsf{M}_1 \leftarrow \mathsf{Sim}_1(\mathsf{M}_0, z, (\mathsf{ek}_0, \mathsf{ek}_1))$.
Simulation Security. It should be computationally hard to distinguish the two memory values obtained via the simulation algorithms. That is, for any $\lambda \in \mathbb{N}$ and any $z \in \mathcal{I}$, we have $(z, \mathsf{M}_0) \approx_c (z, \mathsf{M}_1)$ for any $(\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \leftarrow \mathsf{Setup}(1^\lambda)$, $\mathsf{M}_0 \leftarrow \mathsf{Sim}_0(1^\lambda)$, and $\mathsf{M}_1 \leftarrow \mathsf{Sim}_1(\mathsf{M}, z, (\mathsf{ek}_0, \mathsf{ek}_1))$.

4.4 Staged Homomorphic Secret Sharing

Finally, we define a new notion termed staged-HSS which is merely extending the idea of HSS with simulatable memory values to the case where we require the possibility of input values to be simulatable as well.

Definition 6 (staged-HSS). Let $\mathsf{HSS} = (\mathsf{Setup}, \mathsf{MemGen}, \mathsf{Input}, \mathsf{Eval})$ be an HSS scheme following the RMS template, with input space $\mathcal{I}$ over the ring $\mathcal{R}$. We say it is a staged-HSS if there exist additional algorithms $(\overline{\mathsf{Input}}_0, \overline{\mathsf{Input}}_1)$, and $(\overline{\mathsf{Eval}}_0, \overline{\mathsf{Eval}}_1)$ such that:

- $\overline{\mathsf{Input}}_0(\mathsf{pk}) \to (\bar{\mathsf{I}}_0, \mathsf{aux})$: On input a public key pk, return a value $\bar{\mathsf{I}}_0$ and an auxiliary output aux.
- $\overline{\mathsf{Input}}_1(\mathsf{pk}, x, \mathsf{aux}, (\mathsf{ek}_0, \mathsf{ek}_1)) \to \bar{\mathsf{I}}_1$: On input a public key pk, an input $x \in \mathcal{I}$, an auxiliary input aux, and two encoding keys $(\mathsf{ek}_0, \mathsf{ek}_1)$, return a value $\bar{\mathsf{I}}_1$.
- $\overline{\mathsf{Eval}}_0(\mathsf{ek}_0, (\bar{\mathsf{I}}_0^{(1)}, \ldots, \bar{\mathsf{I}}_0^{(\rho)}), P) \to \mathsf{M}_0$: On input an evaluation key ek_0, a vector of ρ input values $(\bar{\mathsf{I}}_0^{(1)}, \ldots, \bar{\mathsf{I}}_0^{(\rho)})$, and a program P, return a memory value M_0.
- $\overline{\mathsf{Eval}}_1(\mathsf{ek}_1, (\bar{\mathsf{I}}_1^{(1)}, \ldots, \bar{\mathsf{I}}_1^{(\rho)}), (x^{(1)}, \ldots, x^{(\rho)}), P) \to \mathsf{M}_1$: On input an evaluation key ek_1, a vector of ρ input values $(x^{(1)}, \ldots, x^{(\rho)})$ as well as $(\bar{\mathsf{I}}_1^{(1)}, \ldots, \bar{\mathsf{I}}_1^{(\rho)})$, and a program P, return a memory value M_1.

We further require the two following properties:

Correctness. We would like the outputs of $\overline{\mathsf{Eval}}_0$ and $\overline{\mathsf{Eval}}_1$ to be usable within the extended evaluation algorithm $\mathsf{ExtEval}$ (Lemma 1). Formally, for any $\lambda \in \mathbb{N}$ and any two RMS programs $P, Q \in \mathcal{P}$, it should hold that

$$\Pr[y_0 - y_1 = P(z^{(1)}, \ldots, z^{(\ell)}) \cdot Q(x^{(1)}, \ldots, x^{(\rho)})] \geq 1 - \mathsf{negl}(\lambda) \ ,$$

where

- $(\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \leftarrow \mathsf{Setup}(1^\lambda)$,
- $(\mathsf{I}_0^{x^{(i)}}, \mathsf{I}_1^{x^{(i)}}) \leftarrow \mathsf{Input}(\mathsf{pk}, x^{(i)})$ for all $i \in [\rho]$,
- $(\bar{\mathsf{I}}_0^{z^{(i)}}, \mathsf{aux}^{(i)}) \leftarrow \overline{\mathsf{Input}}_0(\mathsf{pk})$ and $\bar{\mathsf{I}}_1^{z^{(i)}} \leftarrow \overline{\mathsf{Input}}_1(\mathsf{pk}, z^{(i)}, \mathsf{aux}^{(i)}, (\mathsf{ek}_0, \mathsf{ek}_1))$ for all $i \in [\ell]$,
- $\mathsf{M}_0 \leftarrow \overline{\mathsf{Eval}}_0(\mathsf{ek}_0, (\bar{\mathsf{I}}_0^{z^{(1)}}, \ldots, \bar{\mathsf{I}}_0^{z^{(\ell)}}), P)$,
- $\mathsf{M}_1 \leftarrow \overline{\mathsf{Eval}}_1(\mathsf{ek}_1, (\bar{\mathsf{I}}_1^{z^{(1)}}, \ldots, \bar{\mathsf{I}}_1^{z^{(\ell)}}), (z^{(1)}, \ldots, z^{(\ell)}), P)$,
- $y_\sigma \leftarrow \mathsf{ExtEval}(\sigma, \mathsf{ek}_\sigma, (\mathsf{M}_\sigma, \mathsf{I}_\sigma^{x^{(1)}}, \ldots, \mathsf{I}_\sigma^{x^{(\rho)}}), Q)$, for $\sigma \in \{0, 1\}$.

Security. The output of $\overline{\mathsf{Input}}_1$ and Input should be computationally indistinguishable. Formally, for any $\lambda \in \mathbb{N}$, and any $x \in \mathcal{I}$, the two following distributions should be computationally indistinguishable:

$$\left\{ \bar{\mathsf{I}}_1 : \begin{array}{c} (\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \leftarrow \mathsf{Setup}(1^\lambda) \\ (\bar{\mathsf{I}}_0, \mathsf{aux}) \leftarrow \overline{\mathsf{Input}}_0(\mathsf{pk}) \\ \bar{\mathsf{I}}_1 \leftarrow \overline{\mathsf{Input}}_1(\mathsf{pk}, x, \mathsf{aux}, (\mathsf{ek}_0, \mathsf{ek}_1)) \end{array} \right\} \overset{c}{\approx} \left\{ \mathsf{I}_1 : \begin{array}{c} (\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \leftarrow \mathsf{Setup}(1^\lambda), \\ (\mathsf{I}_0, \mathsf{I}_1) \leftarrow \mathsf{Input}(\mathsf{pk}, x) \end{array} \right\} .$$

Theorem 2. *Assuming the hardness of* DCR*, there exists HSS scheme following the RMS template which generates simulatable memory values, as well as staged-HSS scheme for the class of RMS programs.*

The above theorem follows from the HSS scheme introduced by Orlandi, Scholl, and Yakoubov in [37] that supports the class of RMS programs and works under the DCR assumption. In Appendix C,we show that it satisfies the properties of all the three introduced variants.

5 Constrained Pseudorandom Functions

We now present our two transformations from homomorphic secret sharing to constrained pseudorandom functions.

5.1 CPRF for Inner-Product from HSS

Our first construction is a 1-key selectively secure constrained pseudorandom function for inner-product. The space input is $\mathcal{R}^n$ for some ring $\mathcal{R}$ and $n > 0$, and a constraint is defined by a vector $\mathbf{z} \in \mathcal{R}^n$. A constrained key for a vector $\mathbf{z}$ allows to compute the PRF evaluation on input $\mathbf{x} \in \mathcal{R}^n$ if and only if $\langle \mathbf{z}, \mathbf{x} \rangle = 0$. Specifically, the class of constraints is $\{C_\mathbf{z} \mid \mathbf{z} \in \mathcal{R}^n\}$ where the circuit $C_\mathbf{z} : \mathcal{R}^n \rightarrow \{0, 1\}$ is defined as $C_\mathbf{z}(\mathbf{x}) = 0$ if $\langle \mathbf{z}, \mathbf{x} \rangle = 0$, else 1.

The intuition behind our construction is that the master secret key and the constrained key (for a vector $\mathbf{z}$) are used to compute, via HSS, a share of $\langle \mathbf{x}, \mathbf{z} \rangle \cdot F_k(\mathbf{x})$, where k is a PRF key encoded via the HSS scheme. Then, if $\langle \mathbf{x}, \mathbf{z} \rangle = 0$, the two evaluations produce substractive shares of 0, i.e. equal shares, while if $\langle \mathbf{x}, \mathbf{z} \rangle \neq 0$, the shares differ by (a non-zero multiple of) $F_k(\mathbf{x})$. By the security of HSS, the PRF key k remains hidden to the constrained key owner, hence the actual PRF evaluation (the value of the share computed from the master secret key) is pseudorandom even given the value of the second share (which can be computed from the constrained key).

Before diving into our construction, we generalize Lemma 1, stating that not only one can produce shares of any evaluation of the form $z \cdot P(\mathbf{x})$ given a memory value for z and encoding of $\mathbf{x}$, but of any linear combination $\sum_i \alpha^{(i)} z^{(i)} \cdot P(\mathbf{x})$ with known coefficients given memory values for multiple $z^{(i)}$'s, i.e. for $\langle \mathbf{z}, \boldsymbol{\alpha} \rangle$ for a known vector $\boldsymbol{\alpha} = (\alpha^{(1)}, \ldots, \alpha^{(\ell)})$.

Corollary 1. *Let* $\mathsf{HSS} = (\mathsf{Setup}, \mathsf{Input}, \mathsf{MemGen}, \mathsf{Eval})$ *be an HSS scheme following the RMS template. There exists an extended evaluation algorithm* $\mathsf{LinExtEval}$*:*

- $\mathsf{LinExtEval}(\sigma, \mathsf{ek}_\sigma, (\mathsf{M}_\sigma^{(1)}, \dots, \mathsf{M}_\sigma^{(\ell)}), (\mathsf{I}_\sigma^{(1)}, \dots, \mathsf{I}_\sigma^{(\rho)}), (\alpha^{(1)}, \dots, \alpha^{(\ell)}), P) \to y_\sigma$:
  On input a party index $\sigma \in \{0,1\}$, an evaluation key ek_σ, a vector of ℓ memory values $(\mathsf{M}_\sigma^{(1)}, \dots, \mathsf{M}_\sigma^{(\ell)})$, a vector of ρ input values $(\mathsf{I}_\sigma^{(1)}, \dots, \mathsf{I}_\sigma^{(\rho)})$, a vector of ℓ ring elements $\alpha^{(1)}, \dots, \alpha^{(\ell)}$, and an RMS program P, this algorithm outputs a value y_σ such that the following holds.

For any security parameter $\lambda \in \mathbb{N}$, any $\alpha^{(i)} \in \mathcal{R}$ for $i \in [\ell]$, and any RMS program P, we have:

$$\Pr\left[y_0 - y_1 = \left(\sum_{i=1}^{\ell} \alpha^{(i)} \cdot z^{(i)} \right) \cdot P(x^{(1)}, \dots, x^{(\rho)}) \right] \geq 1 - \mathsf{negl}(\lambda) \ ,$$

where the probability is taken over sampling $(\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \leftarrow \mathsf{Setup}(1^\lambda)$, $(\mathsf{I}_0^{(i)}, \mathsf{I}_1^{(i)}) \leftarrow \mathsf{Input}(\mathsf{pk}, x^{(i)})$, $\mathsf{M}_\sigma^{(j)} \leftarrow \mathsf{MemGen}(\sigma, \mathsf{ek}_\sigma, z^{(j)})$, and over the shares $y_\sigma \leftarrow \mathsf{LinExtEval}(\sigma, \mathsf{ek}_\sigma, (\mathsf{M}_\sigma^{(1)}, \dots, \mathsf{M}_\sigma^{(\ell)}), (\mathsf{I}_\sigma^{(1)}, \dots, \mathsf{I}_\sigma^{(\rho)}), (\alpha^{(1)}, \dots, \alpha^{(\ell)}), P)$, with $\sigma \in \{0,1\}, j \in [\ell], i \in [\rho]$.

The proof of the above statement follows from Lemma 1 by linearly combining the substractive shares obtained by applying $\mathsf{ExtEval}$ with each memory value.

For a PRF $F : \mathcal{K} \times \mathcal{R}^n \to \mathcal{Y}$ with domain $\mathcal{R}^n$ and for $\mathbf{x} \in \mathcal{R}^n$, we denote by $F_\bullet(\mathbf{x}) : \mathcal{K} \to \mathcal{Y}$ the function that maps $k \in \mathcal{K}$ to $F_k(\mathbf{x})$.

We now have all the ingredients for our first construction.

Construction 1 (CPRF for IP from HSS). Let $F : \mathcal{K} \times \mathcal{R}^n \to \mathcal{Y}$ be a PRF with evaluation in NC^1. Let $\mathsf{HSS} = (\mathsf{Setup}, \mathsf{Input}, \mathsf{MemGen}, \mathsf{Eval})$ be a homomorphic secret sharing following the RMS template with simulatable memory values. We design $(\mathsf{KeyGen}, \mathsf{Eval}, \mathsf{Constrain}, \mathsf{CEval})$ as follows:

<u>$\mathsf{KeyGen}(1^\lambda)$:</u>

1. $(\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \xleftarrow{\$} \mathsf{Setup}(1^\lambda)$.
2. Sample $k \xleftarrow{\$} \mathcal{K}$ for F
3. Run $(\mathsf{I}_0, \mathsf{I}_1) \leftarrow \mathsf{Input}(\mathsf{pk}, k)$.
4. For $i \in \{1, \dots, n\}$:
   $\mathsf{M}_0^i \leftarrow \mathsf{Sim}_0(1^\lambda)$.
5. $\mathsf{msk} \leftarrow ((\mathsf{ek}_0, \mathsf{I}_0, (\mathsf{M}_0^i)_{i\in[n]}), (\mathsf{ek}_1, \mathsf{I}_1))$
6. Output $\mathsf{pp} = \mathsf{pk}$ and msk.

<u>$\mathsf{Eval}(\mathsf{pp}, \mathsf{msk}, \mathbf{x})$:</u>

1. Parse msk as $((\mathsf{ek}_0, \mathsf{I}_0, (\mathsf{M}_0^i)_{i\in[n]}), (\mathsf{ek}_1, \mathsf{I}_1))$.
2. Compute $y_0 \leftarrow \mathsf{LinExtEval}(0, \mathsf{ek}_0, (\mathsf{M}_0^i)_{i\in[n]}, \mathsf{I}_0, \mathbf{x}, F_\bullet(\mathbf{x}))$.
3. Output y_0.

<u>$\mathsf{Constrain}(\mathsf{msk}, \mathbf{z})$:</u>

1. Parse msk as $((\mathsf{ek}_0, \mathsf{I}_0, (\mathsf{M}_0^i)_{i\in[n]}), (\mathsf{ek}_1, \mathsf{I}_1))$
2. Parse $\mathbf{z} = (z_1, \dots, z_n)$.
3. For $i \in \{1, \dots, n\}$:
   $\mathsf{M}_1^i \leftarrow \mathsf{Sim}_1(\mathsf{M}_0^i, z_i, (\mathsf{ek}_0, \mathsf{ek}_1))$
4. Return $\mathsf{ck}_\mathbf{z} = (\mathsf{ek}_1, \mathsf{I}_1, (\mathsf{M}_1^i)_{i\in[n]})$.

<u>$\mathsf{CEval}(\mathsf{pp}, \mathsf{ck}_\mathbf{z}, \mathbf{x})$:</u>

1. Parse $\mathsf{ck}_\mathbf{z} = (\mathsf{ek}_1, \mathsf{I}_1, (\mathsf{M}_1^i)_{i\in[n]})$.
2. Compute $y_1 \leftarrow \mathsf{LinExtEval}(1, \mathsf{ek}_1, (\mathsf{M}_1^i)_{i\in[n]}, \mathsf{I}_1, \mathbf{x}, F_\bullet(\mathbf{x}))$.
3. Output y_1.

Theorem 3. *Assuming F is a secure PRF with evaluation in NC^1 and HSS is a secure HSS scheme following the RMS template with simulatable memory values, then Construction 1 is a selective 1-key, constraint-hiding, secure CPRF for inner-product.*

The proof of Theorem 3 is detailed in Appendix D.1.

Remark 1. In the above construction, we require the PRF range $\mathcal{Y}$ to be such that F is pseudorandom on $\mathbb{Z}_n$, for a fixed $n < B$, where B is the magnitude bound of the RMS programs that the HSS scheme used in the construction supports. We need to then reduce the outputs of the HSS evaluation algorithm modulo n by inputting n as the modulus to algorithm Output (See Definition 4). This is used in the security proof to ensure that masking with a pseudorandom value over $\mathcal{Y}$ causes the output to be pseudorandom.

Corollary 2 (Private CPRF for Inner-Product from DCR). *There exist 1-key selectively-secure, constraint-hiding constrained pseudorandom functions for inner-product assuming the hardness of* DCR.

5.2 CPRF for NC^1 from HSS

We now describe CPRF for the class of NC^1 constraints. We consider the representation of an NC^1 circuit C with input size $n = \mathsf{poly}(\lambda)$ and depth $d = \mathcal{O}(\log n)$ to be a bit string $(C_1, \ldots, C_z) \in \{0,1\}^z$, where $z = \mathsf{poly}(n)$ is the description size. Also, we denote the universal circuit by $U(\cdot,\cdot)$ that on input a circuit $C \in \{0,1\}^z$ and $x = (x_1, \ldots, x_n) \in \{0,1\}^n$, outputs $U(C,x) = C(x)$. Due to the work of Cooks and Hoover [22], we know that there exists a universal circuit that correctly computes any NC^1 circuit and is itself an NC^1 circuit.

The strategy for our construction is similar as for inner-product. We aim to obtain substractive shares $U(C,x) \cdot F_k(x)$ via the (standard and constrained) evaluation algorithms, where F is a pseudo-random function with evaluation in NC^1, C denotes the constraint, and U denotes the above universal circuit.

A crucial point is that the master secret key should allow to compute such a share for any input x independently of the constraint C. Hence, we have to find a way to replace the encoding of C that is given to the evaluator by oblivious values that guarantee the correctness. In the inner-product case, where we want shares of $\langle \mathbf{x}, \mathbf{z} \rangle \cdot F_k(\mathbf{x})$, we used simulated memory values as the independent share of the undetermined constraint $\mathbf{z}$, and programmed the constrained key to guarantee correctness according to the constraint vector $\mathbf{z}$. However, this technique cannot be applied to the case of NC^1 constraints as we are dealing with non-linear evaluations.

The idea is again to use staged-HSS. We first compute a memory for $U(C,x)$ using $\overline{\mathsf{Eval}}_0$ and $\overline{\mathsf{Eval}}_1$. Then, this memory value is used in the $\mathsf{ExtEval}$ algorithm from Lemma 1 to compute a share of $U(C,x) \cdot F_k(x)$ additionally using an encoding of k.

The important point here, is that inputs of $\overline{\mathsf{Eval}}_0$ can be sampled obliviously using $(\bar{\mathsf{I}}_0, \mathsf{aux}) \leftarrow \overline{\mathsf{Input}}_0(\mathsf{pk})$, and therefore can be sampled during Setup without the knowledge of the constraint C. Yet, when computing the constrained key for C, the master key owner can use the full knowledge of C as well as auxiliary information generated during Setup to appropriately compute memory values for the i-th bit C_i of the description of C, using $\bar{\mathsf{I}}_1 \leftarrow \overline{\mathsf{Input}}_1(\mathsf{pk}, C_i, \mathsf{aux}, (\mathsf{ek}_0, \mathsf{ek}_1))$. The correctness of staged-HSS then guarantees the correctness of evaluations, while its security plays a role in the security proof to remove the need for both evaluation keys when computing $\bar{\mathsf{I}}_1$, therefore allowing to rely on HSS security to remove the information about the underlying PRF key k.

We now detail our construction. For any $x \in \{0,1\}^n$, we denote by $U(\cdot, x)$ the circuit that maps $C \in \{0,1\}^z$ to $U(C,x) = C(x) \in \{0,1\}$.

Construction 2 (CPRF for NC^1 from HSS). Let $F : \mathcal{K} \times \{0,1\}^n \to \mathcal{Y}$ be a pseudorandom function with evaluation in NC^1, where $\mathcal{Y}$ is a finite cyclic group. Let $\mathsf{HSS} = (\mathsf{Setup}, \mathsf{MemGen}, \mathsf{Input}, \mathsf{Eval})$ be a staged homomorphic secret sharing scheme and denote by $(\overline{\mathsf{Input}}_0, \overline{\mathsf{Input}}_1)$, and $(\overline{\mathsf{Eval}}_0, \overline{\mathsf{Eval}}_1)$ the additional algorithms defined in Definition 6. Let $\mathsf{ExtEval}$ be the modified evaluation algorithm as in Lemma 1. We construct a constrained pseudorandom function that supports NC^1 constraints as follows:

- $\mathsf{KeyGen}(1^\lambda)$:
  - Run $(\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \leftarrow \mathsf{Setup}(1^\lambda)$.
  - Choose a random key $k \xleftarrow{\$} \mathcal{K}$ for F and compute $(\mathsf{I}_0, \mathsf{I}_1) \leftarrow \mathsf{Input}(\mathsf{pk}, k)$.
  - For $i \in \{1, \ldots, z\}$, compute $(\bar{\mathsf{I}}_0^{(i)}, \mathsf{aux}^{(i)}) \leftarrow \overline{\mathsf{Input}}_0(\mathsf{pk})$.
  - Output $\mathsf{pp} = \mathsf{pk}$, and $\mathsf{msk} = ((\mathsf{ek}_0, \mathsf{ek}_1, \mathsf{I}_0, \mathsf{I}_1), (\bar{\mathsf{I}}_0^{(1)}, \mathsf{aux}^{(1)}, \ldots, \bar{\mathsf{I}}_0^{(z)}, \mathsf{aux}^{(z)}))$.
- $\mathsf{Eval}(\mathsf{pp}, \mathsf{msk}, x)$:
  - Parse $\mathsf{pp} = \mathsf{pk}$, and $\mathsf{msk} = ((\mathsf{ek}_0, \mathsf{ek}_1, \mathsf{I}_0, \mathsf{I}_1), (\bar{\mathsf{I}}_0^{(1)}, \mathsf{aux}^{(1)}, \ldots, \bar{\mathsf{I}}_0^{(z)}, \mathsf{aux}^{(z)}))$.
  - Run $\mathsf{M}_0 \leftarrow \overline{\mathsf{Eval}}_0(\mathsf{ek}_0, (\bar{\mathsf{I}}_0^{(1)}, \ldots, \bar{\mathsf{I}}_0^{(z)}), U(\cdot, x))$. Here, $\bar{\mathsf{I}}_0^{(i)}$ represents the input value of C_i for $i \in \{1, \ldots, z\}$.
  - Run $y_0 \leftarrow \mathsf{ExtEval}(0, \mathsf{ek}_0, \mathsf{M}_0, \mathsf{I}_0, F_\bullet(x))$. Here, M_0 denotes the memory value of $U(C,x)$, and I_0 denotes the input value of k.
  - Output y_0.
- $\mathsf{Constrain}(\mathsf{msk}, C)$:
  - Parse $\mathsf{msk} = ((\mathsf{ek}_0, \mathsf{ek}_1, \mathsf{I}_0, \mathsf{I}_1), (\bar{\mathsf{I}}_0^{(1)}, \mathsf{aux}^{(1)}, \ldots, \bar{\mathsf{I}}_0^{(z)}, \mathsf{aux}^{(z)}))$, and $C = (C_1, \ldots, C_z) \in \{0,1\}^z$.

- For $i \in \{1, \ldots, z\}$, run $\bar{\mathsf{I}}_1^{(i)} \leftarrow \overline{\mathsf{Input}}_1(\mathsf{pk}, C_i, \mathsf{aux}^{(i)}, (\mathsf{ek}_0, \mathsf{ek}_1))$.
- Output $\mathsf{ck}_C = (\mathsf{ek}_1, \mathsf{I}_1, (\bar{\mathsf{I}}_1^{(1)}, \ldots, \bar{\mathsf{I}}_1^{(z)}), C)$.

- $\mathsf{CEval}(\mathsf{pp}, \mathsf{ck}_C, x)$:
  - Parse $\mathsf{ck}_C = (\mathsf{ek}_1, \mathsf{I}_1, (\bar{\mathsf{I}}_1^{(1)}, \ldots, \bar{\mathsf{I}}_1^{(z)}), C)$.
  - Run $\mathsf{M}_1 \leftarrow \overline{\mathsf{Eval}}_1(\mathsf{ek}_1, (\bar{\mathsf{I}}_1^{(1)}, \ldots, \bar{\mathsf{I}}_1^{(z)}), (C^{(1)}, \ldots, C^{(z)}), U(\cdot, x))$.
  - Run $y_1 \leftarrow \mathsf{ExtEval}(1, \mathsf{ek}_1, \mathsf{M}_1, \mathsf{I}_1, F_\bullet(x))$.
  - Output y_1.

Theorem 4 (CPRF for NC^1 from Staged HSS). *Assuming F is a secure pseudorandom function with evaluation in NC^1 and HSS is a secure staged-HSS scheme, Construction 2 is a selective 1-key secure constrained pseudorandom function for NC^1.*

The proof of Theorem 4 is detailed in Appendix D.2.

Remark 2. We note that the above construction is not constraint-hiding, since the constrained evaluation algorithm relies on the knowledge of the constraint.

Corollary 3 (CPRF for NC^1 from DCR). *Assuming the DCR assumption holds, there exist 1-key selectively-secure constrained pseudorandom functions for NC^1 constraints.*

Remark 3 (Other Instantiations). Although not explicitly detailed in this work, our transformations from HSS to CPRF works using either of the schemes from [12] based on the Learning With Errors (LWE) assumption with super-polynomial modulus, from [1] based on the hardness of Joye-Libert encryption scheme, from [1] based on the Decisional Diffie-Hellman (DDH) and Decisional Cross-Group Diffie-Hellman (DXDH) assumptions over class groups, or from [19] based on the Hard Subgroup Membership (HSM) assumption over class groups. All of the above HSS schemes follow the same outline as the DCR-based scheme of [37] when generating input and memory values. More precisely, input values are ciphertexts computed using a PKE scheme, and in all of the mentioned schemes, the used encryption tool generates ciphertexts that contain a separate part as a commitment to the encryption randomness which is independent of the underlying plaintext. This feature makes it feasible to generalize these schemes into staged-HSS schemes and then use it to construct CPRF for NC^1 constraints. These schemes also allow simulation of memory values which enables using the scheme to construct CPRF for inner-product constraints. This holds since a valid memory value of these schemes is a subtractive share of a secret vector dependent on the secret key of the used PKE, thus one share can be sampled obliviously and the other one can be correctly computed given the secret vector.

Also, using HSS with only polynomial correctness (e.g., the DDH-based scheme of [10]) still yields CPRFs for *polynomial-size* domain. This leads to constructions of *poly-size domain* private CPRFs for inner-products and CPRFs for NC^1 from DDH, and from LWE with polynomial modulus-to-noise ratio.

6 Applications to Secure Multiparty Computation

In this section, we explore the applications of staged-HSS (defined in Section 4) to secure computation. We first show how using staged-HSS allows constructing a secure two-party computation protocol with *precomputable* silent preprocessing. In this model, one party can perform all of the heavy preprocessing, not only before the inputs are selected (which can be already achieved by "non-staged" HSS for RMS programs) but also before knowing the identity of the other party. Next, we show that the DCR-based construction of staged-HSS (provided in Section C) can be used to obtain sublinear-communication secure two-party computation with *one-sided statistical security*. Our proposal follows the same outline as [10] where the authors showed how HSS for RMS programs yields secure computation with sublinear communication. Definitions and proofs for this section can be found in Sections A and E.

We start by introducing the notion of *precomputability* for pseudorandom correlation functions. Informally, precomputability enables the first party to generate its key locally before knowing anything about the second party. The second party's key is then (securely) computed as a function of the first key.

In the absence of some form of trusted setup, dishonest-majority secure computation requires computational assumptions. A popular paradigm (used for instance in [33,25]) is to first have the parties jointly execute a precomputation phase which is independent of their inputs or the function they want to compute, in order to distribute correlated randomness, and afterwards, use the computed correlated randomness in an information-theoretic online phase to perform the secure computation. Heuristically, this online phase, which is free of any expensive cryptographic operations, can be made highly efficient. The generation of the correlated randomness in the precomputation phase can be done via a *pseudorandom correlation generator* (PCG) [7] or a *pseudorandom correlation function* (PCF), whose seeds (in the case of PCG), or keys (in the case of PCF) are generated using generic (computationally secure) MPC protocol.

Using a precomputable PCF allows the parties to perform the following three-phase MPC protocol: (1) Alice samples her PCF key, and can perform the expensive PCF evaluation with her key offline to recover her share of the correlated randomness; (2) Alice and Bob use generic secure computation to generate Bob's key, which then allows Bob to evaluate the PCF with his key and recover his share of the correlated randomness; (3) Alice and Bob perform the information-theoretic phase online, using their correlated randomness. This allows Alice to perform the brunt of her computation offline, before any interaction with Bob. This offline phase can be viewed as "party-independent" precomputation, which is more general than input-independence.

Definition 7 (Precomputable Pseudorandom Correlation Function). Let $\mathcal{Y}$ be a reverse-sampleable correlation with output lengths $\ell_0(\lambda), \ell_1(\lambda)$ and let $\lambda \leq n(\lambda) \leq \mathsf{poly}(\lambda)$ be its input length. We say that a pseudorandom correlation function $(\mathsf{PCF.Gen}, \mathsf{PCF.Eval})$ is *precomputable* if the description of $\mathsf{PCF.Gen}$ contains the descriptions of two algorithms $(\mathsf{PCF.Gen}_0, \mathsf{PCF.Gen}_1)$ such that

- $\mathsf{PCF.Gen}_0(1^\lambda)$: On input the security parameter λ, returns a key k_0 and auxiliary output aux.
- $\mathsf{PCF.Gen}_1(1^\lambda, k_0, \mathsf{aux})$: On input the security parameter λ, a key k_0, and an auxiliary input aux, outputs a key k_1.

We also require the following property to hold:
Precomputability. For any security parameter $\lambda \in \mathbb{N}$, the two following distributions are computationally indistinguishable:

$$\left\{(k_0, k_1)\colon\ (k_0, k_1) \leftarrow \mathsf{PCF.Gen}(1^\lambda)\right\} \overset{c}{\approx} \left\{(k_0, k_1)\colon \begin{array}{l}(k_0, \mathsf{aux}) \leftarrow \mathsf{PCF.Gen}_0(1^\lambda) \\ k_1 \leftarrow \mathsf{PCF.Gen}_1(1^\lambda, k_0, \mathsf{aux})\end{array}\right\}.$$

Remark 4 (Design Choices for Definition 7).

– *Syntax.* Because we want the statement "a precomputable PCF is a PCF" to be formally true, we define a PCF as a pair $(\mathsf{PCF.Gen}, \mathsf{PCF.Eval})$, and not as a triple $(\mathsf{PCF.Gen}_0, \mathsf{PCF.Gen}_1, \mathsf{PCF.Eval})$. In practice however, a precomputable PCF is more convenient to describe as such a triple, with $\mathsf{PCF.Gen}$ implicitly understood to be defined by: On input 1^λ, compute $(k_0, \mathsf{aux}) \leftarrow \mathsf{Gen}_0(1^\lambda)$ then $k_1 \leftarrow \mathsf{Gen}_1(1^\lambda, k_0, \mathsf{aux})$, and output (k_0, k_1).
– *Computational Indistinguishability.* A potentially inconvenient downside of only requiring computational indistinguishability in "precomputability" is that a precomputable PCF might be (N, B, ϵ)-secure, but "replacing Gen with Gen_0 and Gen_1" might incur a security loss and yield a PCF which is only $(N, B, \epsilon + \epsilon')$-secure. However, since we will never consider such precise notions of security, this will not be an issue in this paper.
– *Privately Precomputable PCF.* To be precise, the definition of precomputability actually captures the stronger notion of *private precomputability*, in that it is computationally hard to determine whether keys are generated together using Gen, or one after the other, using Gen_0 and Gen_1.

Below, we provide a construction of precomputable PCF for OLE correlations from staged-HSS, and using a pseudorandom function. Given an input, first, each party samples a PRF key and sets the first half of the correlated pair to be the value of the PRF on the input. Next, to generate the additive shares of the product of these two values, they use staged-HSS. Here, we require the staged-HSS scheme to generate shares that are individually pseudorandom given the input, and in Lemma 2 we show that this can be assumed without loss of generality. This is because the property "pseudorandom $\mathcal{R}$-OLE-correlated outputs" for a PCF, which can be seen as a form of *correctness* property, essentially requires that the PCF outputs not only valid OLE tuples but also pseudorandom ones from the view of an external adversary.

Lemma 2 (HSS with Pseudorandom Outputs). *Denote by $\mathcal{P}$ a class of programs defined over a ring $\mathcal{R}$, with input space $\mathcal{I} \subseteq \mathcal{R}$. Assuming the existence of one-way functions, any HSS scheme for $\mathcal{P}$ can be modified in such a way that each output share is pseudorandom to an external adversary given only the input (but neither input share).*

Formally, assuming the existence of an HSS scheme $\mathsf{HSS} = (\mathsf{Setup}, \mathsf{Input}, \mathsf{Eval}, \mathsf{Rec})$ *for $\mathcal{P}$, there exists an HSS scheme* $\mathsf{HSS}' = (\mathsf{Setup}', \mathsf{Input}', \mathsf{Eval}', \mathsf{Rec}')$ *for $\mathcal{P}$ such that:*

$$\forall \sigma \in \{0,1\}, \forall (P : \mathcal{R} \to \mathcal{Y}) \in \mathcal{P}, \forall x \in \mathcal{R} :$$

$$\left\{ (x, y_\sigma) \colon \begin{array}{l} (\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \leftarrow \mathsf{Setup}'(1^\lambda) \\ (\mathsf{I}_0, \mathsf{I}_1) \leftarrow \mathsf{Input}'(x) \\ y_\sigma \leftarrow \mathsf{Eval}'(\sigma, \mathsf{ek}_\sigma, I_\sigma, P) \end{array} \right\} \overset{c}{\approx} \{(x, r) \colon r \overset{\$}{\leftarrow} \mathcal{Y}\} .$$

Moreover, if HSS *has additive reconstruction, then so does* HSS'*, and if* HSS *is a staged-HSS scheme, then* HSS' *is also a staged-HSS.*

Construction 3 (Precomputable & Programmable PCF for OLE). Let $F : \mathcal{K} \times \mathcal{I} \to \mathcal{Y}$ be a pseudorandom function with evaluation in NC^1, where $\mathcal{I}, \mathcal{Y}$ are finite rings. Let $\mathsf{HSS} = (\mathsf{Setup}, \mathsf{MemGen}, \mathsf{Input}, \mathsf{Eval})$ be a staged homomorphic secret sharing scheme and denote by $(\overline{\mathsf{Input}}_0, \overline{\mathsf{Input}}_1)$, and $(\overline{\mathsf{Eval}}_0, \overline{\mathsf{Eval}}_1)$ the additional algorithms defined in Definition 6. Let $\mathsf{ExtEval}$ be the modified evaluation algorithm as in Lemma 1. Our PCF works as follows:

- $\mathsf{PCF.Gen}(1^\lambda)$:
  - Run $(k_0, \mathsf{aux}) \leftarrow \mathsf{PCF.Gen}_0(1^\lambda)$.
  - Run $k_1 \leftarrow \mathsf{PCF.Gen}_1(1^\lambda, k_0, \mathsf{aux})$.
  - Output (k_0, k_1).
- $\mathsf{PCF.Gen}_0(1^\lambda)$:
  - Run $(\mathsf{pk}, \mathsf{ek}_0, \mathsf{ek}_1) \leftarrow \mathsf{HSS.Setup}(1^\lambda)$.
  - Sample $k_{\mathsf{prf}}^{(0)} \overset{\$}{\leftarrow} \mathcal{K}$, and compute $(\mathsf{I}_0, \mathsf{I}_1) \leftarrow \mathsf{HSS.Input}(\mathsf{pk}, k_{\mathsf{prf}}^{(0)})$.
  - Run $(\bar{\mathsf{I}}_0, \mathsf{aux}') \leftarrow \mathsf{HSS}.\overline{\mathsf{Input}}_0(\mathsf{pk})$.
  - Output $k_0 = (\mathsf{ek}_0, \mathsf{I}_0, \bar{\mathsf{I}}_0, k_{\mathsf{prf}}^{(0)})$, and $\mathsf{aux} = (\mathsf{aux}', \mathsf{ek}_1, \mathsf{I}_1)$.
- $\mathsf{PCF.Gen}_1(1^\lambda, k_0, \mathsf{aux})$:
  - Parse $k_0 = (\mathsf{ek}_0, \mathsf{I}_0, \bar{\mathsf{I}}_0, k_{\mathsf{prf}}^{(0)})$, and $\mathsf{aux} = (\mathsf{aux}', \mathsf{ek}_1, \mathsf{I}_1)$.
  - Sample $k_{\mathsf{prf}}^{(1)} \overset{\$}{\leftarrow} \mathcal{K}$, and compute $\bar{\mathsf{I}}_1 \leftarrow \mathsf{HSS}.\overline{\mathsf{Input}}_1(\mathsf{pk}, k_{\mathsf{prf}}^{(1)}, \mathsf{aux}')$.
  - Output $k_1 = (\mathsf{ek}_1, \bar{\mathsf{I}}_1, \mathsf{I}_1, k_{\mathsf{prf}}^{(1)})$.
- $\mathsf{PCF.Eval}(\sigma, k_\sigma, \mathbf{x})$:
  - Parse $k_\sigma = (\mathsf{ek}_\sigma, \mathsf{I}_\sigma, \bar{\mathsf{I}}_\sigma, k_{\mathsf{prf}}^{(\sigma)})$.
  - If $\sigma = 0$, then
    - Run $\mathsf{M}_\sigma \leftarrow \mathsf{HSS}.\overline{\mathsf{Eval}}_0(\mathsf{ek}_\sigma, \bar{\mathsf{I}}_\sigma, F_\bullet(\mathbf{x}))$.

    Else if $\sigma = 1$,
    - Run $\mathsf{M}_\sigma \leftarrow \mathsf{HSS}.\overline{\mathsf{Eval}}_1(\mathsf{ek}_\sigma, \bar{\mathsf{I}}_\sigma, k_{\mathsf{prf}}^{(\sigma)}, F_\bullet(\mathbf{x}))$.
  - Run $y_\sigma \leftarrow \mathsf{HSS.ExtEval}(\mathsf{ek}_\sigma, (\mathsf{M}_\sigma, \mathsf{I}_\sigma), f(\mathbf{x}))$, with $f(\mathbf{x})$ defined as $f(\mathbf{x})\colon (k^{(0)}, k^{(1)}) \mapsto F_{k^{(0)}}(\mathbf{x}) \cdot F_{k^{(1)}}(\mathbf{x})$.
  - Output $(F_{k_{\mathsf{prf}}^{(\sigma)}}(\mathbf{x}), y_\sigma)$.

Theorem 5. *Let $\mathcal{R}$ be a finite ring. Assuming F is a secure pseudorandom function with evaluation in* NC^1 *and* HSS *is a secure staged-HSS scheme, Construction 3 is a two-party precomputable PCF for OLE correlations over $\mathcal{R}$. Furthermore, this PCF is programmable.*

The proof of Theorem 5 is provided in Appendix E.1. By combining Theorems 2 and 5, we get Corollary 4.

Corollary 4 (Precomputable PCF for $\mathcal{R}$-OLE from DCR). *Assuming the* DCR *assumption holds, there exists a two-party precomputable pseudorandom correlation function (as per Definition 7) for the* $\mathcal{R}$-OLE *correlation.*

Corollary 5 (From OLE to Low-Degree Correlations). *Assuming the existence of (one-way functions and of) staged-HSS supporting the class of RMS programs, there exists a two-party precomputable PCF (Definition 7) for low-degree correlations (Definition 13). In particular, such a PCF exists under the* DCR *assumption.*

6.1 Sublinear Computation with One-Sided Statistical Security

Most constructions of two-party HSS for super-constant depth circuits can be used in a non black-box way to build two-party secure computation with an amount of communication which is sublinear in (or even independent of) the circuit-size: if the input share generation algorithm is simple enough to be securely distributed with low communication, the parties need to only run the evaluation algorithm locally, then reconstruct the output.

In the $\mathcal{F}^{\mathsf{HSS}}_{\mathsf{update}}$-Hybrid Model. The main component (apart from the HSS scheme itself) in building sublinear secure computation from HSS is the low-communication distributed share generation. When using staged-HSS, the first party can simply sample its share locally, so the hard part is updating the second party so they can receive their share too. We formalize this task in Figure 1 as the ideal functionality $\mathcal{F}^{\mathsf{HSS}}_{\mathsf{update}}$. We prove in Lemma 3 that there exists sublinear two-party secure computation, provided this step can be performed with one-sided statistical security and with low-enough communication.

Functionality $\mathcal{F}^{\mathsf{HSS}}_{\mathsf{update}}$

The functionality is parameterized with a staged-HSS scheme $\mathsf{staged\text{-}HSS} = (\mathsf{staged\text{-}HSS.Setup}, \mathsf{staged\text{-}HSS.Input}, \mathsf{staged\text{-}HSS.MemGen}, \mathsf{staged\text{-}HSS.Eval})$.

Input: Wait to receive $(\mathtt{share}, \mathsf{staged\text{-}HSS.pk}, \bar{\mathsf{I}}_0, \mathsf{aux})$ from P_0 and $(\mathtt{input}, x_1)$ from P_1.
Output: Compute $\bar{\mathsf{I}}_1 \leftarrow \mathsf{staged\text{-}HSS.}\overline{\mathsf{Input}}_1(\mathsf{staged\text{-}HSS.pk}, x_1, \mathsf{aux})$; output $(\mathsf{staged\text{-}HSS.pk}, \bar{\mathsf{I}}_1)$ to P_1.

Fig. 1. Ideal functionality $\mathcal{F}^{\mathsf{HSS}}_{\mathsf{update}}$, parameterized by a staged-HSS scheme, for generating the second input share given the first, precomputed, one.

Functionality $\mathcal{F}_{\mathsf{SFE}}(C)$

The functionality is parameterised with an arithmetic circuit $C\colon \mathcal{R}^{n_0} \times \mathcal{R}^{n_1} \to \mathcal{R}^m$ over a finite field $\mathbb{F}$.

Input: Wait to receive $(\mathtt{input}, \sigma, x_\sigma)$ from each party P_σ $(\sigma \in \{0,1\})$, where $x_\sigma \in \mathbb{F}^{n_\sigma}$, and set $\vec{x} \leftarrow (x_0, x_1)$.

Output: Compute $\vec{y} \leftarrow C(\vec{x})$; Sample $\vec{y}_0 \xleftarrow{\$} \mathcal{R}^m$; Set $\vec{y}_1 \leftarrow \vec{y} - \vec{y}_0$; Output $\vec{y}_0$ to P_0 and $\vec{y}_1$ to P_1.

Fig. 2. Ideal functionality $\mathcal{F}_{\mathsf{SFE}}(C)$ for the two-party secure evaluation of an arithmetic circuit C.

Protocol Π_C

Parties: Alice and Bob
Parameters: The protocol is parameterized with:

- $C\colon \mathbb{F}^{n_0} \times \mathbb{F}^{n_1} \to \mathbb{F}^m$ is an arithmetic circuit over finite field $\mathbb{F}$.
- $\mathsf{HSS} = (\mathsf{HSS.Setup}, \mathsf{HSS.Input}, \mathsf{HSS.MemGen}, \mathsf{HSS.Eval})$ is a staged-HSS scheme with pseudorandom shares supporting the class of RMS programs over $\mathbb{F}$ (seen as a ring). We denote the staged input and evaluation algorithms by $(\mathsf{HSS.}\overline{\mathsf{Input}}_0, \mathsf{HSS.}\overline{\mathsf{Input}}_1)$ and $(\mathsf{HSS.}\overline{\mathsf{Eval}}_0, \mathsf{HSS.}\overline{\mathsf{Eval}}_1)$. Let $\mathsf{HSS.ExtEval}$ be defined as in Lemma 1.
- $F(\cdot,\cdot)$ is a PRF in NC^1 with domain $\{0,1\}^\lambda$, key space $\{0,1\}^\lambda$, and range $\mathbb{F}^{n_1}$.

Hybrid Model: The protocol is defined in the $\mathcal{F}^{\mathsf{HSS}}_{\mathsf{update}}$-hybrid model.
Input: Alice holds $x_0 \in \mathbb{F}^{n_0}$ and Bob holds $x_1 \in \mathbb{F}^{n_1}$.
The Protocol:

Alice's precomputation phase. Alice does the following:

1. $K \xleftarrow{\$} \{0,1\}^\lambda$
2. $(\mathsf{HSS.pk}, \mathsf{ek}_0, \mathsf{ek}_1) \leftarrow \mathsf{HSS.Setup}(1^\lambda)$
3. $(\bar{\mathsf{I}}_0, \mathsf{aux}) \leftarrow \mathsf{HSS.}\overline{\mathsf{Input}}_0(\mathsf{HSS.pk})$
4. $(\mathsf{I}_0, \mathsf{I}_1) \leftarrow \mathsf{HSS.Input}(1^\lambda, K)$
5. $\alpha \xleftarrow{\$} \{0,1\}^\lambda$, $c_{\mathsf{in}} \leftarrow x_0 + F(K,\alpha)$, and $r_{\mathsf{out}} \xleftarrow{\$} \mathbb{F}^m$
6. $M_0 \leftarrow \mathsf{HSS.}\overline{\mathsf{Eval}}(\mathsf{ek}_0, \bar{\mathsf{I}}_0, F(\cdot,\alpha))$
7. $y_0 \leftarrow \mathsf{HSS.ExtEval}(\mathsf{ek}_0, (M_0, I_0), f_{\alpha,c_{\mathsf{in}}})$,
   where $f_{\alpha,c_{\mathsf{in}}}\colon (X,Y) \mapsto C(c_{\mathsf{in}} - F(X,\alpha), Y)$

Online phase.

8. Alice sends $(\mathsf{ek}_1, \mathsf{I}_1, c_{\mathsf{in}}, \alpha, r_{\mathsf{out}})$ to Bob, who waits to receive it.
9. Alice sends $(\mathtt{share}, \mathsf{HSS.pk}, \bar{\mathsf{I}}_0, \mathsf{aux})$ to $\mathcal{F}^{\mathsf{HSS}}_{\mathsf{update}}$;
   Bob sends $(\mathtt{input}, x_1)$ to $\mathcal{F}^{\mathsf{HSS}}_{\mathsf{update}}$, and waits to receive $(\mathsf{HSS.pk}, \bar{\mathsf{I}}_1)$ from $\mathcal{F}^{\mathsf{HSS}}_{\mathsf{update}}$.

Bob's computation phase. Bob does the following:

1. $M_1 \leftarrow \mathsf{HSS.}\overline{\mathsf{Eval}}(\mathsf{ek}_1, \bar{\mathsf{I}}_1, F(\cdot,\alpha))$
2. $y_1 \leftarrow \mathsf{HSS.ExtEval}(\mathsf{ek}_1, (M_1, I_1), f_{\alpha,c_{\mathsf{in}}})$,
   where $f_{\alpha,c_{\mathsf{in}}}\colon (X,Y) \mapsto C(c_{\mathsf{in}} - F(X,\alpha), Y)$

Output phase. Alice outputs $y_0' \leftarrow y_0 + r_{\mathsf{out}}$; Bob outputs $y_1' \leftarrow y_1 - r_{\mathsf{out}}$.

Fig. 3. (Sublinear) Secure Two-Party Computation with One-Sided Statistical Security from staged-HSS Supporting the Class of RMS Programs.

Lemma 3 (Secure Computation with One-Sided Statistical Security in the $\mathcal{F}^{\mathsf{HSS}}_{\mathsf{update}}$-hybrid model). *Let $C\colon \mathbb{F}^{n_0} \times \mathbb{F}^{n_1} \to \mathbb{F}^m$ be an arithmetic circuit over a finite field $\mathbb{F}$. Let* staged-HSS *be a staged-HSS scheme with pseudorandom shares supporting the class of RMS programs over $\mathbb{F}$ (seen as a ring).*

The protocol Π_C provided in Figure 3 UC-securely implements the two-party functionality $\mathcal{F}_{\mathsf{SFE}}(C)$ in the $\mathcal{F}^{\mathsf{HSS}}_{\mathsf{update}}$-hybrid model, against a passive adversary statically corrupting at most one of the parties, with perfect security against Alice, and computational security against Bob. The protocol uses $\lambda^{\mathcal{O}(1)} + (n_1 + m) \cdot \log|\mathbb{F}|$ bits of communication.

Instantiating $\mathcal{F}^{\mathsf{HSS}}_{\mathsf{update}}$ under DCR. We now show how to instantiate $\mathcal{F}^{\mathsf{HSS}}_{\mathsf{update}}$ for construction of staged-HSS from DCR (Construction 4). This instantiation is non black-box in the HSS scheme, and uses a combination of the Paillier-ElGamal encryption scheme, which is provably semantically secure under DCR, and oblivious linear evaluation (OLE) with one-sided statistical security, which is known from DCR.

Functionality $\mathcal{F}_{\mathsf{OLE}}$

The functionality $\mathcal{F}_{\mathsf{OLE}}$ for (batch) oblivious linear evaluation is parameterized by a finite field $\mathbb{F}$, and interacts with two parties P_0 and P_1.

Input: Wait to receive $(\mathtt{input}, 0, \mathbf{u} = (u_1, \ldots, u_s))$ (where $u_1, \ldots, u_s \in \mathbb{F}$) from P_0 and $(\mathtt{input}, 1, \mathbf{v} = (v_1, \ldots, v_t))$ (where $v_1, \ldots, v_t \in \mathbb{F}$) from P_1.
Output: Compute $\mathbf{z} \leftarrow (u_i \cdot v_j)_{i\in[s],j\in[t]}$, sample $\mathbf{z}_0 \stackrel{\$}{\leftarrow} \mathbb{F}^{s \cdot t}$, set $\mathbf{z}_1 \leftarrow \mathbf{z} - \mathbf{z}_0$; Output $\mathbf{z}_\sigma$ to P_σ for $\sigma \in \{0, 1\}$.

Fig. 4. Ideal functionality $\mathcal{F}_{\mathsf{OLE}}$ for (batch) oblivious linear evaluation.

Protocol $\Pi^{\mathsf{HSS}}_{\mathsf{update}}$

Parties: Alice and Bob.
Parameters: $\mathbb{F}_{2^\lambda}$ is an exponential-size finite field; n_1 is an input size. staged-HSS is the staged-HSS scheme of Construction 4, instantiated from Paillier-ElGamal. The Paillier-ElGamal cryptosystem itself is parameterized by GenPQ, an algorithm that on input 1^λ, generates $(N = p \cdot q, p, q)$, where p and q are $\ell(\lambda)$-bit primes where $\ell \colon \mathbb{N}^\star \to \mathbb{N}^\star$ is a function such that $\forall \kappa \in \mathbb{N}^\star, \ell(\kappa) \geq 1.5\kappa$. $B_{\mathsf{sk}} := 2^{2\ell(\lambda)-2\log|\mathbb{F}|}$ is the base for the decomposition of the secret key into digits; $s := 2\ell(\lambda) + 2\log|\mathbb{F}|$ is the number of cyphertexts needed to encrypt the secret key; $t := \lceil n_1 \frac{\log|\mathbb{F}|}{2\ell(\lambda)} \rceil$.
Hybrid Model: The protocol is defined in the $\mathcal{F}_{\mathsf{OLE}}$-hybrid model.
Input: Alice holds $(\mathsf{HSS.pk}, \bar{\mathsf{I}}_0, \mathsf{aux})$ and Bob holds $x_1 = (x_1^{(1)}, \ldots, x_1^{(t)}) \in \mathcal{R}^{n_1} \approx [N]^t$.
The Protocol:

1. Alice does the following:
   - Parse $\mathsf{HSS.pk} = (\mathsf{pk}_{\mathsf{PaillierEG}}, D^{(0)}, \ldots, D^{(s-1)})$
     // $D^{(j)}$ is a Paillier-ElGamal encryption under pk of the j^{th} digit of the secret key in base B_{sk}
   - Parse $\bar{\mathsf{I}}_0 = (\mathsf{ct}_{\mathsf{ind}}, (\mathsf{ct}^{(i,j)}_{\mathsf{ind}})_{(i,j)\in[t]\times[s+1]})$
     // $\mathsf{ct}_{\mathsf{ind}}$ is of the form g^r, and $\mathsf{ct}^{(i,j)}_{\mathsf{ind}}$ is of the form $g^{r_{i,j}}$
   - Parse $\mathsf{aux} = (g^r, \mathsf{pk}^r_{\mathsf{PaillierEG}}, (g^{r_{i,j}})_{(i,j)\in[t]\times[s+1]}, (\mathsf{pk}^{r_{i,j}}_{\mathsf{PaillierEG}})_{(i,j)\in[t]\times[s+1]})$
     // $\mathsf{pk}_{\mathsf{PaillierEG}} = g^{\mathsf{sk}_{\mathsf{PaillierEG}}} \bmod N^2$
2. Alice sends $(N, \mathsf{pk}_{\mathsf{PaillierEG}}, \mathsf{ct}_{\mathsf{ind}})$ to Bob
3. Alice sends $(\mathtt{input}, 0, (1\|d))$ to $\mathcal{F}_{\mathsf{OLE}}$ and waits to receive $\mathbf{y}^{(0)} = (y^{(0)}_{i,j})_{(i,j)\in[t]\times[s+1]}$;
   Bob sends $(\mathtt{input}, 1, x_1)$ to $\mathcal{F}_{\mathsf{OLE}}$ and waits to receive $\mathbf{y}^{(1)} = (y^{(1)}_{i,j})_{(i,j)\in[t]\times[s+1]}$.
   // Adding the digit 1 to the secret key d condenses the notations of the encryption of the input alone, and those of the input times each digit of the secret key, as $x \cdot (1, d_0, \ldots, d_{s-1}) = (x, x \cdot d_0, \ldots, x \cdot d_{s-1})$.
4. Alice does the following:
   For each $(i,j) \in [t] \times [s+1]$, $c_{i,j} \leftarrow (1+N)^{y^{(0)}_{i,j}} \cdot h^{r_{i,j}}$
5. Alice sends $\mathbf{c} = (c_{i,j})_{(i,j)\in[t]\times[s+1]}$ to Bob, who waits to receive it.
6. Bob sets $\mathsf{ct}_{\mathsf{dep}} \leftarrow (c_{i,j} \cdot (1+N)^{y^{(1)}_{i,j}})_{(i,j)\in[t]\times[s+1]}$ and outputs $\bar{\mathsf{I}}_1 \leftarrow (\mathsf{ct}_{\mathsf{ind}}, \mathsf{ct}_{\mathsf{dep}})$.

Fig. 5. Protocol for securely realizing $\mathcal{F}^{\mathsf{HSS}}_{\mathsf{update}}$ under the circular security of the Paillier-ElGamal cryptosystem.

Lemma 4 (Instantiating Lemma 3 under DCR). *Let* HSS *be the staged-HSS scheme of Construction 4. Assuming the* DCR *assumption holds, the protocol $\Pi^{\mathsf{HSS}}_{\mathsf{update}}$ provided in Figure 5 UC-securely implements the two-party functionality $\mathcal{F}^{\mathsf{HSS}}_{\mathsf{update}}$ in the $\mathcal{F}_{\mathsf{OLE}}$-hybrid model, against a passive adversary statically corrupting at most one of the parties, with perfect security against Alice and Bob. The protocol uses $\mathcal{O}(\lambda \cdot n_1)$ bits of communication.*

We then obtain our final claim.

Theorem 6 (Computation for NC^1 with Circuit-Independent-Communication and One-Sided Statistical Security from Circular Security of Paillier-ElGamal). *Let C be an RMS program with $n = n_0 + n_1$ inputs and m outputs over $\mathbb{F}_{2^\lambda}$. Assuming* DCR *and the circular security of the Paillier-ElGamal encryption scheme, there exists a protocol that UC-securely implements the two-party functionality $\mathcal{F}_{\mathsf{SFE}}(C)$, against a passive adversary that statically corrupts at most one of the parties, with perfect security against a corrupted Alice, and computational security against a corrupted Bob. The protocol uses $\lambda^{\mathcal{O}(1)} + \mathcal{O}((n+m) \cdot \log |\mathbb{F}|)$ bits of communication.*

Proof. The proof follows from a combination of Lemmas 3 and 4, as well as a linear-communication protocol for $\mathbb{F}_{2^\lambda}$-OLE (which we batch "naively" in order to instantiate $\mathcal{F}_{\mathsf{OLE}}$, which in fact corresponds to $t \cdot (s+1)$-batch OLE). Such a protocol is folklore, and can be achieved *e.g.* by using Gilboa's [28] information-theoretic reduction of OLE to string-OT and OT-extension [32,2].

Acknowledgments. We thank the anonymous reviewers of Eurocrypt 2023 for their comments. Geoffroy Couteau was supported by the French ANR SCENE (ANR-20-CE39-0001) and by the France 2030 ANR Project ANR-22-PECY-003 SecureCompute. Pierre Meyer was supported by ERC Project HSS (852952). Alain Passelègue and Mahshid Riahinia were supported by the French ANR RAGE project (ANR-20-CE48-0011) and the France 2030 ANR Project ANR-22-PECY-003 SecureCompute.

References

1. Abram, D., Damgård, I., Orlandi, C., Scholl, P.: An algebraic framework for silent preprocessing with trustless setup and active security. In: Dodis, Y., Shrimpton, T. (eds.) CRYPTO 2022, Part IV. LNCS, vol. 13510, pp. 421–452. Springer, Heidelberg (Aug 2022). https://doi.org/10.1007/978-3-031-15985-5˙15
2. Asharov, G., Lindell, Y., Schneider, T., Zohner, M.: More efficient oblivious transfer extensions. Journal of Cryptology **30**(3), 805–858 (Jul 2017). https://doi.org/10.1007/s00145-016-9236-6
3. Attrapadung, N., Matsuda, T., Nishimaki, R., Yamada, S., Yamakawa, T.: Constrained PRFs for NC^1 in traditional groups. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part II. LNCS, vol. 10992, pp. 543–574. Springer, Heidelberg (Aug 2018). https://doi.org/10.1007/978-3-319-96881-0˙19
4. Banerjee, A., Fuchsbauer, G., Peikert, C., Pietrzak, K., Stevens, S.: Key-homomorphic constrained pseudorandom functions. In: Dodis, Y., Nielsen, J.B. (eds.) TCC 2015, Part II. LNCS, vol. 9015, pp. 31–60. Springer, Heidelberg (Mar 2015). https://doi.org/10.1007/978-3-662-46497-7˙2
5. Boneh, D., Lewi, K., Wu, D.J.: Constraining pseudorandom functions privately. In: Fehr, S. (ed.) PKC 2017, Part II. LNCS, vol. 10175, pp. 494–524. Springer, Heidelberg (Mar 2017). https://doi.org/10.1007/978-3-662-54388-7˙17
6. Boneh, D., Waters, B.: Constrained pseudorandom functions and their applications. In: Sako, K., Sarkar, P. (eds.) ASIACRYPT 2013, Part II. LNCS, vol. 8270, pp. 280–300. Springer, Heidelberg (Dec 2013). https://doi.org/10.1007/978-3-642-42045-0˙15
7. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Scholl, P.: Efficient pseudorandom correlation generators: Silent OT extension and more. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019, Part III. LNCS, vol. 11694, pp. 489–518. Springer, Heidelberg (Aug 2019). https://doi.org/10.1007/978-3-030-26954-8˙16
8. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Scholl, P.: Correlated pseudorandom functions from variable-density LPN. In: 61st FOCS. pp. 1069–1080. IEEE Computer Society Press (Nov 2020). https://doi.org/10.1109/FOCS46700.2020.00103
9. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Orrù, M.: Homomorphic secret sharing: Optimizations and applications. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) ACM CCS 2017. pp. 2105–2122. ACM Press (Oct / Nov 2017). https://doi.org/10.1145/3133956.3134107
10. Boyle, E., Gilboa, N., Ishai, Y.: Breaking the circuit size barrier for secure computation under DDH. In: Robshaw, M., Katz, J. (eds.) CRYPTO 2016, Part I. LNCS, vol. 9814, pp. 509–539. Springer, Heidelberg (Aug 2016). https://doi.org/10.1007/978-3-662-53018-4˙19
11. Boyle, E., Goldwasser, S., Ivan, I.: Functional signatures and pseudorandom functions. In: Krawczyk, H. (ed.) PKC 2014. LNCS, vol. 8383, pp. 501–519. Springer, Heidelberg (Mar 2014). https://doi.org/10.1007/978-3-642-54631-0˙29
12. Boyle, E., Kohl, L., Scholl, P.: Homomorphic secret sharing from lattices without FHE. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part II. LNCS, vol. 11477, pp. 3–33. Springer, Heidelberg (May 2019). https://doi.org/10.1007/978-3-030-17656-3˙1

13. Brakerski, Z., Goldwasser, S.: Circular and leakage resilient public-key encryption under subgroup indistinguishability - (or: Quadratic residuosity strikes back). In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 1–20. Springer, Heidelberg (Aug 2010). https://doi.org/10.1007/978-3-642-14623-7˙1
14. Brakerski, Z., Tsabary, R., Vaikuntanathan, V., Wee, H.: Private constrained PRFs (and more) from LWE. In: Kalai, Y., Reyzin, L. (eds.) TCC 2017, Part I. LNCS, vol. 10677, pp. 264–302. Springer, Heidelberg (Nov 2017). https://doi.org/10.1007/978-3-319-70500-2˙10
15. Brakerski, Z., Vaikuntanathan, V.: Constrained key-homomorphic PRFs from standard lattice assumptions - or: How to secretly embed a circuit in your PRF. In: Dodis, Y., Nielsen, J.B. (eds.) TCC 2015, Part II. LNCS, vol. 9015, pp. 1–30. Springer, Heidelberg (Mar 2015). https://doi.org/10.1007/978-3-662-46497-7˙1
16. Bresson, E., Catalano, D., Pointcheval, D.: A simple public-key cryptosystem with a double trapdoor decryption mechanism and its applications. In: Laih, C.S. (ed.) ASIACRYPT 2003. LNCS, vol. 2894, pp. 37–54. Springer, Heidelberg (Nov / Dec 2003). https://doi.org/10.1007/978-3-540-40061-5˙3
17. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: 42nd FOCS. pp. 136–145. IEEE Computer Society Press (Oct 2001). https://doi.org/10.1109/SFCS.2001.959888
18. Canetti, R., Chen, Y.: Constraint-hiding constrained PRFs for NC^1 from LWE. In: Coron, J.S., Nielsen, J.B. (eds.) EUROCRYPT 2017, Part I. LNCS, vol. 10210, pp. 446–476. Springer, Heidelberg (Apr / May 2017). https://doi.org/10.1007/978-3-319-56620-7˙16
19. Castagnos, G., Laguillaumie, F., Tucker, I.: Threshold linearly homomorphic encryption on $z/2^kz$. Cryptology ePrint Archive (2022)
20. Chaum, D.: The spymasters double-agent problem: Multiparty computations secure unconditionally from minorities and cryptographically from majorities. In: Brassard, G. (ed.) CRYPTO'89. LNCS, vol. 435, pp. 591–602. Springer, Heidelberg (Aug 1990). https://doi.org/10.1007/0-387-34805-0˙52
21. Chen, Y., Vaikuntanathan, V., Wee, H.: GGH15 beyond permutation branching programs: Proofs, attacks, and candidates. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part II. LNCS, vol. 10992, pp. 577–607. Springer, Heidelberg (Aug 2018). https://doi.org/10.1007/978-3-319-96881-0˙20
22. Cook, S.A., Hoover, H.J.: A depth-universal circuit. SIAM J. Comput. **14**(4), 833–839 (1985). https://doi.org/10.1137/0214058, `https://doi.org/10.1137/0214058`
23. Cramer, R., Shoup, V.: Universal hash proofs and a paradigm for adaptive chosen ciphertext secure public-key encryption. In: Knudsen, L.R. (ed.) EUROCRYPT 2002. LNCS, vol. 2332, pp. 45–64. Springer, Heidelberg (Apr / May 2002). https://doi.org/10.1007/3-540-46035-7˙4
24. Damgård, I., Groth, J., Salomonsen, G.: The Theory and Implementation of an Electronic Voting System, pp. 77–99. Springer US, Boston, MA (2003). https://doi.org/10.1007/978-1-4615-0239-5˙6, `https://doi.org/10.1007/978-1-4615-0239-5_6`
25. Damgård, I., Pastro, V., Smart, N.P., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 643–662. Springer, Heidelberg (Aug 2012). https://doi.org/10.1007/978-3-642-32009-5˙38
26. Davidson, A., Katsumata, S., Nishimaki, R., Yamada, S., Yamakawa, T.: Adaptively secure constrained pseudorandom functions in the standard model. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020, Part I. LNCS, vol. 12170, pp. 559–589. Springer, Heidelberg (Aug 2020). https://doi.org/10.1007/978-3-030-56784-2˙19
27. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Mitzenmacher, M. (ed.) 41st ACM STOC. pp. 169–178. ACM Press (May / Jun 2009). https://doi.org/10.1145/1536414.1536440
28. Gilboa, N.: Two party RSA key generation. In: Wiener, M.J. (ed.) CRYPTO'99. LNCS, vol. 1666, pp. 116–129. Springer, Heidelberg (Aug 1999). https://doi.org/10.1007/3-540-48405-1˙8
29. Goldreich, O., Goldwasser, S., Micali, S.: How to construct random functions (extended abstract). In: 25th FOCS. pp. 464–479. IEEE Computer Society Press (Oct 1984). https://doi.org/10.1109/SFCS.1984.715949
30. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or A completeness theorem for protocols with honest majority. In: Aho, A. (ed.) 19th ACM STOC. pp. 218–229. ACM Press (May 1987). https://doi.org/10.1145/28395.28420
31. Hofheinz, D., Kamath, A., Koppula, V., Waters, B.: Adaptively secure constrained pseudorandom functions. In: Goldberg, I., Moore, T. (eds.) FC 2019. LNCS, vol. 11598, pp. 357–376. Springer, Heidelberg (Feb 2019). https://doi.org/10.1007/978-3-030-32101-7˙22
32. Ishai, Y., Kilian, J., Nissim, K., Petrank, E.: Extending oblivious transfers efficiently. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 145–161. Springer, Heidelberg (Aug 2003). https://doi.org/10.1007/978-3-540-45146-4˙9
33. Ishai, Y., Prabhakaran, M., Sahai, A.: Founding cryptography on oblivious transfer - efficiently. In: Wagner, D. (ed.) CRYPTO 2008. LNCS, vol. 5157, pp. 572–591. Springer, Heidelberg (Aug 2008). https://doi.org/10.1007/978-3-540-85174-5˙32

34. Kiayias, A., Papadopoulos, S., Triandopoulos, N., Zacharias, T.: Delegatable pseudorandom functions and applications. In: Sadeghi, A.R., Gligor, V.D., Yung, M. (eds.) ACM CCS 2013. pp. 669–684. ACM Press (Nov 2013). https://doi.org/10.1145/2508859.2516668
35. Naor, M., Pinkas, B.: Efficient oblivious transfer protocols. In: Kosaraju, S.R. (ed.) 12th SODA. pp. 448–457. ACM-SIAM (Jan 2001)
36. Naor, M., Reingold, O., Rosen, A.: Pseudo-random functions and factoring (extended abstract). In: 32nd ACM STOC. pp. 11–20. ACM Press (May 2000). https://doi.org/10.1145/335305.335307
37. Orlandi, C., Scholl, P., Yakoubov, S.: The rise of paillier: Homomorphic secret sharing and public-key silent OT. In: Canteaut, A., Standaert, F.X. (eds.) EUROCRYPT 2021, Part I. LNCS, vol. 12696, pp. 678–708. Springer, Heidelberg (Oct 2021). https://doi.org/10.1007/978-3-030-77870-5˙24
38. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: Stern, J. (ed.) EUROCRYPT'99. LNCS, vol. 1592, pp. 223–238. Springer, Heidelberg (May 1999). https://doi.org/10.1007/3-540-48910-X˙16
39. Peikert, C., Shiehian, S.: Privately constraining and programming PRFs, the LWE way. In: Abdalla, M., Dahab, R. (eds.) PKC 2018, Part II. LNCS, vol. 10770, pp. 675–701. Springer, Heidelberg (Mar 2018). https://doi.org/10.1007/978-3-319-76581-5˙23
40. Roy, L., Singh, J.: Large message homomorphic secret sharing from DCR and applications. In: Malkin, T., Peikert, C. (eds.) CRYPTO 2021, Part III. LNCS, vol. 12827, pp. 687–717. Springer, Heidelberg, Virtual Event (Aug 2021). https://doi.org/10.1007/978-3-030-84252-9˙23

Supplementary Material

A Additional Definitions

A.1 Decision Composite Residuosity Assumption

The DCR Assumption. Let SampleModulus be a polynomial-time algorithm that on input the security parameter λ, outputs (N, p, q), where $N = pq$ for λ-bit primes p and q.

Definition 8 (Decision Composite Residuosity assumption, [38]). Let λ be the security parameter. We say that the Decision Composite Residuosity (DCR) problem is hard relative to SampleModulus if $(N, x) \approx_c (N, x^N)$ where $(N, p, q) \xleftarrow{\$} \mathsf{SampleModulus}(1^\lambda)$, $x \xleftarrow{\$} \mathbb{Z}^*_{N^2}$, and x^N is computed modulo N^2.

Note that $\mathbb{Z}^*_{N^2}$ can be written as a product of subgroups $\mathbb{H} \times \mathbb{NR}_N$, where $\mathbb{H} = \{(1+N)^i : i \in [N]\}$ is of order N, and $\mathbb{NR}_N = \{x^N : x \in \mathbb{Z}^*_{N^2}\}$ is the subgroup of N-th residues that has order $\phi(N)$.

Circular-Secure Paillier Cryptosystem. We also recall the circular-secure Paillier cryptosystem presented by Brakerski and Goldwasser in [13] which is introduced as a circular-secure version of Paillier encryption [38]. The security of the scheme follows from the DCR assumption. The scheme is parameterized by $\ell \in \mathbb{N}$ that is polynomial in the security parameter λ.

$\mathsf{BG.KeyGen}(1^\lambda)$:
1. Sample $(N, p, q) \leftarrow \mathsf{SampleModulus}(1^\lambda)$.
2. Sample $\mathbf{g} = (g_0, \dots, g_{\ell-1}) \xleftarrow{\$} \mathbb{NR}^\ell_N$.
3. Sample $\mathbf{d} = (d^{(0)}, \dots, d^{(\ell-1)}) \xleftarrow{\$} \{0,1\}^\ell$.
4. Compute $\hat{g} = \prod_{i=0}^{\ell-1} g_i^{d^{(i)}} (\mod N^2)$.
5. Output $\mathsf{pk} = (N, \mathbf{g}, \hat{g})$ and $\mathsf{sk} = \mathbf{d}$.

$\mathsf{BG.Enc}(\mathsf{pk}, x)$:
1. Sample $r \xleftarrow{\$} \mathbb{Z}_N$.
2. Compute and output $\mathsf{ct} = (g_0^r, \dots, g_{\ell-1}^r, \hat{g}^r \cdot (1+N)^x)$.

$\mathsf{BG.Dec}(\mathsf{sk}, \mathsf{ct})$
1. Parse $\mathsf{ct} = (c_0, \dots, c_{\ell-1}, \hat{c})$.
2. Compute $\bar{c} = (\prod_{i=0}^{\ell-1} c_i^{-d^{(i)}}) \cdot \hat{c} (\mod N^2)$.
3. Compute and output $x = (\bar{c} - 1)/N$.

Paillier-ElGamal Cryptosystem. The Paillier-ElGamal cryptosystem [23,24,16] is defined by following triple (PaillierEG.Gen, PaillierEG.Enc, PaillierEG.Dec), and boils down to using the ElGamal cryptosystem over the group $(\mathbb{Z}^\star_{N^2}, \times)$ where N is a Blum integer of the form $N = pq$, where p and q are primes:

$\mathsf{PaillierEG.Gen}(1^\lambda)$:
1. Sample $g' \xleftarrow{\$} [N^2]$
2. Set $g \leftarrow (g')^{2N} \bmod N^2$
3. Sample $d \xleftarrow{\$} [N^2]$
4. Output $(\mathsf{pk} = g^d \bmod N^2, \mathsf{sk} = d)$

$\mathsf{PaillierEG.Enc}(\mathsf{pk}, x)$:
1. Sample $r \xleftarrow{\$} N$
2. Output $\mathsf{ct} = (g^r, \mathsf{pk}^r \cdot (1+N)^x)$

$\mathsf{PaillierEG.Dec}(\mathsf{sk}, \mathsf{ct} = (\mathsf{ct}_0, \mathsf{ct}_1))$:
1. Set $\mathsf{ct}' \leftarrow \mathsf{ct}_1 \cdot (\mathsf{ct}_0)^{-d} \bmod N^2$
2. Output $x = \frac{\mathsf{ct}'-1}{N}$

Assuming the DCR assumption (Definition 8), the Paillier-ElGamal cryptosystem is semantically secure. Observe that Paillier-ElGamal is a special case of the circular-secure Paillier cryptosystem of [13], where $\ell = 1$ (where ℓ is defined as in the previous paragraph). If we wish to encrypt (digits of) the secret key under Paillier-ElGamal however, we will need to assume the *circular security of the Paillier-ElGamal encryption scheme.*

A.2 Pseudorandom Functions (PRFs)

We first recall the Real-or-Random security notion of a pseudorandom function.

Definition 9 (Real-or-Random Security). Let λ be a security parameter. A function $F : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$ is called a secure pseudorandom function if it is efficiently computable and for any PPT adversary $\mathcal{A}$, the following holds:

$$\left| \Pr\left[\mathcal{A}^{F(k,\cdot)}(1^\lambda) = 1 \middle| k \xleftarrow{\$} \mathcal{K}\right] - \Pr\left[\mathcal{A}^{RF(\cdot)}(1^\lambda) = 1 \middle| RF \xleftarrow{\$} \mathcal{F}\right] \right| = \mathsf{negl}(\lambda),$$

where $\mathcal{F}$ is the set of all functions with domain $\mathcal{X}$ and range $\mathcal{Y}$.

Next, we recall the Find-and-Guess security of a pseudorandom function which is equivalent to the Real-or-Random security up to a multiplicative gap of $\mathcal{O}(Q)$ between the advantage functions, where Q is the number of evaluation queries.

Definition 10 (Find-then-Guess Security). Let λ be a security parameter. A function $F : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$ is called a secure pseudorandom function if it is efficiently computable and the advantage of any PPT adversary $\mathcal{A}$ in the following game is negligible:

- **Setup.** The challenger chooses a random key $k \xleftarrow{\$} \mathcal{K}$ and a random bit $b \xleftarrow{\$} \{0,1\}$, and initializes a set $S = \varnothing$.
- **Pre-Challenge Evaluation Queries.** $\mathcal{A}$ adaptively sends arbitrary inputs $x \in \mathcal{X}$ to the challenger. The challenger computes and returns $F_k(x)$ to $\mathcal{A}$. It also updates $S \leftarrow S \cup \{x\}$.
- **Challenge Phase.** $\mathcal{A}$ sends an input $x^* \in \mathcal{X}$ as its challenge query to the challenger with the restriction that $x^* \notin S$. If $b = 0$, then the challenger computes $y^* \leftarrow F_k(x^*)$. If $b = 1$, then the challenger samples a random element $y^* \xleftarrow{\$} \mathcal{Y}$. It then returns y^* to $\mathcal{A}$.
- **Post-Challenge Evaluation Queries.** $\mathcal{A}$ continues sending arbitrary inputs $x \in \mathcal{X}$ to the challenger with the restriction that $x \neq x^*$, and receives $F_k(x)$.
- **Guess.** $\mathcal{A}$ outputs a bit $b' \in \{0,1\}$.

$\mathcal{A}$ wins if $b' = b$.

A.3 Pseudorandom Correlation Function

Definition 11 (Reverse-Sampleable Correlation). Let $1 \leq \ell_0(\lambda), \ell_1(\lambda) \leq \mathsf{poly}(\lambda)$ be output-length functions. Let $\mathcal{Y}$ be a probabilistic algorithm that, on input 1^λ, returns a pair of outputs $(y_0, y_1) \in \{0,1\}^{\ell_0(\lambda)} \times \{0,1\}^{\ell_1(\lambda)}$, defining a correlation on the outputs.

We say that $\mathcal{Y}$ defines a *reverse-sampleable* correlation if there exists a probabilistic polynomial time algorithm $\mathsf{RSample}$ which takes as input 1^λ, $\sigma \in \{0,1\}$, and $y_\sigma \in \{0,1\}^{\ell_\sigma(\lambda)}$, and outputs $y_{1-\lambda}^{\ell_{1-\sigma}(\lambda)}$, such that for all $\sigma \in \{0,1\}$ the following distributions are statistically close:

$$\{(y_0, y_1) \colon (y_0, y_1) \xleftarrow{\$} \mathcal{Y}(1^\lambda)\} \text{ and } \{(y_0, y_1) \colon (y_0', y_1') \xleftarrow{\$} \mathcal{Y}(1^\lambda), y_\sigma \leftarrow y_\sigma', y_{1-\sigma} \leftarrow \mathsf{RSample}(1^\lambda, \sigma, y_\sigma)\} .$$

Definition 12 (OLE Correlation). Let $\mathcal{R}$ be a finite ring. *An OLE correlation over* $\mathcal{R}$ can be defined as being sampled as a pair (a, c_0) and (b, c_1), where $a, b, c_0 \xleftarrow{\$} \mathcal{R}$ and $c_1 \leftarrow (ab - c_0)$. In other words, each half of the correlation consists of a random ring element, a and b respectively, paired with an additive secret share of the product ab.

Remark 5 (Remarks on OLE Correlations).

– *An OLE Correlation is reverse-sampleable.* Indeed, observe that the reverse-sampling can be performed as follows. $\mathsf{RSample}(1^\sigma, \sigma, y_\sigma)$: Sample a random $\alpha \stackrel{\$}{\leftarrow} \mathcal{R}$, set $\beta \leftarrow (y_\sigma.\mathsf{first} \cdot \alpha - y_\sigma.\mathsf{second})$ and output (α, β).
– *An OLE over* $\mathbb{F}_2$ *is a* 1*-out-of-*2 *bit-OT.* A 1-out-of-2 bit-OT correlation can be defined as being sampled as a pair (m_0, m_1) and (σ, m_σ), where (m_0, m_1) are the OT sender's random messages in $\{0, 1\}$, and σ is the random choice bit given to the receiver. This notion is essentially equivalent to an OLE correlation over $\mathbb{F}_2$ in that there exists a silent (*i.e.* "communication-free") procedure which allows two parties to convert their halves of a sample from one correlation to halves of a sample from the other.
  - **From** (m_0, m_1) **and** (σ, m_σ) **to** (a, c_0) **and** (b, c_1)**:** Assuming Alice and Bob have access to pairs (m_0, m_1) and (σ, m_σ), they can set $a \leftarrow (m_1 - m_0)$, $c_0 \leftarrow -m_0$, $b \leftarrow \sigma$, $c_1 \leftarrow m_\sigma$.
  - **From** (a, c_0) **and** (b, c_1) **to** (m_0, m_1) **and** (σ, m_σ)**:** Assuming Alice and Bob have access to pairs (a, c_0) and (b, c_1), they can set $m_0 \leftarrow -c_0$, $m_1 \leftarrow a - c_0$, $\sigma \leftarrow b$. Note that Bob can "set" $m_\sigma \leftarrow c_1$ and have this value correctly satisfy $m_\sigma = \sigma \cdot m_1 + (1 - \sigma) \cdot m_0$ (*i.e.* $m_\sigma = m_0$ if $\sigma = 0$ and $m_\sigma = m_1$ if $\sigma = 1$).

  In particular, this local conversion procedure implies that the existence of a PCG (resp. PCF) for the first form of OT correlations is equivalent to the existence of a PCG (resp. PCF) for the second.
– *Oblivious Linear Evaluation.* An OLE correlation can be seen as giving one side the coefficients $(a, -c_0)$ of a random linear function over $\mathcal{R}$ and the other side its evaluation on a random point b, as $c_1 = a \cdot b + (-c_0)$.

Definition 13 (Constant-Degree Additive Correlations). Let $\mathcal{R}$ be a finite ring. A *degree-d correlation over* $\mathcal{R}$, parameterized by some tuple $(Q_1, \ldots, Q_m)$ of $(n = n_A + n_B)$-variate degree-d polynomials can be defined as being sampled as a pair $(\vec{r}_A, \vec{s}_A)$ and $(\vec{r}_B, \vec{s}_B)$, where $\vec{r}_A \stackrel{\$}{\leftarrow} \mathcal{R}^{n_A}$, $\vec{r}_B \stackrel{\$}{\leftarrow} \mathcal{R}^{n_B}$, $\vec{s}_A \stackrel{\$}{\leftarrow} \mathcal{R}^m$ and $\vec{s}_B \leftarrow ((Q_1(\vec{r}_A, \vec{r}_B), \ldots, Q_m(\vec{r}_A, \vec{r}_B)) - \vec{s}_A) \in \mathcal{R}^m$. In other words, each half of the correlation consists of a random vector, $\vec{r}_A$ and $\vec{r}_B$ respectively, paired with an additive secret share of the evaluations of the $Q_j(\vec{r}_A, \vec{r}_B)$ for $j \in [m]$.

As noted in [7], the reverse-sampling of additive correlations over a finite abelian group, such as that of Definition 13, is well-defined and computationally efficient.

Definition 14 (Pseudorandom Correlation Function (PCF), [8]). Let $\mathcal{Y}$ be a reverse-sampleable correlation with output length functions $\ell_0(\lambda), \ell_1(\lambda)$ and let $\lambda \le n(\lambda) \le \mathsf{poly}(\lambda)$ be an input length function. Let $(\mathsf{PCF.Gen}, \mathsf{PCF.Eval})$ be a pair of algorithms with the following syntax:

– $\mathsf{PCF.Gen}(1^\lambda)$ is a probabilistic polynomial time algorithm that on input 1^λ, outputs a pair of keys (k_0, k_1); we assume that λ can be inferred from the keys.
– $\mathsf{PCF.Eval}(\sigma, k_\sigma, x)$ is a deterministic polynomial time algorithm that on input $\sigma \in \{0, 1\}$, key k_σ and input value $x \in \{0, 1\}^{n(\lambda)}$, outputs a value $y_\sigma \in \{0, 1\}^{\ell_\sigma(\lambda)}$.

We say that $(\mathsf{PCF.Gen}, \mathsf{PCF.Eval})$ is an (N, B, ϵ)-secure pseudorandom correlation function ($\mathsf{pre\text{-}PCF}$) for $\mathcal{Y}$, if the following conditions hold:

– **Pseudorandom $\mathcal{Y}$-correlated outputs.** For every non-uniform adversary $\mathcal{A}$ of size $B(\lambda)$, it holds that for all sufficiently large λ,

$$|\Pr[\mathsf{Exp}^{\mathsf{pr}}_{\mathcal{A},N,0}(\lambda) = 1] - \Pr[\mathsf{Exp}^{\mathsf{pr}}_{\mathcal{A},N,1}(\lambda) = 1]| \le \epsilon(\lambda)$$

  where $\mathsf{Exp}^{\mathsf{pr}}_{\mathcal{A},N,b}$ $(b \in \{0, 1\})$ is defined as in Figure 6. In particular, the adversary is given access to $N(\lambda)$ samples.

Experiment Pseudorandom Correlated Outputs

$\mathsf{Exp}^{\mathsf{pr}}_{\mathcal{A},N,0}(\lambda)$:

For $i = 1, \ldots, N(\lambda)$:
$\quad x^{(i)} \xleftarrow{\$} \{0,1\}^{n(\lambda)}$
$\quad (y_0^{(i)}, y_1^{(i)}) \hookleftarrow \mathcal{Y}(1^\lambda)$
$b \leftarrow \mathcal{A}(1^\lambda, (x^{(i)}, y_0^{(i)}, y_1^{(i)})_{i \in [N(\lambda)]})$
Output b

$\mathsf{Exp}^{\mathsf{pr}}_{\mathcal{A},N,1}(\lambda)$:

$(k_0, k_1) \leftarrow \mathsf{PCF.Gen}(1^\lambda)$
For $i = 1, \ldots, N(\lambda)$:
$\quad x^{(i)} \xleftarrow{\$} \{0,1\}^{n(\lambda)}$
$\quad$ **for** $\sigma \in \{0,1\}$, $y_\sigma^{(i)} \leftarrow \mathsf{PCF.Eval}(\sigma, k_\sigma, x^{(i)})$
$b \leftarrow \mathcal{A}(1^\lambda, (x^{(i)}, y_0^{(i)}, y_1^{(i)})_{i \in [N(\lambda)]})$
Output b

Fig. 6. Pseudorandom $\mathcal{Y}$-correlated outputs of a PCF.

- **Security.** For every $\sigma \in \{0,1\}$ and every non-uniform adversary $\mathcal{A}$ of size $B(\lambda)$, it holds that for all sufficiently large λ,

$$|\Pr[\mathsf{Exp}^{\mathsf{sec}}_{\mathcal{A},N,\sigma,0}(\lambda) = 1] - \Pr[\mathsf{Exp}^{\mathsf{sec}}_{\mathcal{A},N,\sigma,1}(\lambda) = 1]| \leq \epsilon(\lambda)$$

where $\mathsf{Exp}^{\mathsf{sec}}_{\mathcal{A},N,\sigma,b}$ ($b \in \{0,1\}$) is defined as in Figure 7. In particular, the adversary is given access to $N(\lambda)$ samples

Experiment PCF Security

$\mathsf{Exp}^{\mathsf{sec}}_{\mathcal{A},N,\sigma,0}(\lambda)$:

$(k_0, k_1) \leftarrow \mathsf{PCF.Gen}(1^\lambda)$
For $i = 1, \ldots, N(\lambda)$:
$\quad x^{(i)} \xleftarrow{\$} \{0,1\}^{n(\lambda)}$
$\quad y_{1-\sigma}^{(i)} \leftarrow \mathsf{PCF.Eval}(1-\sigma, k_{1-\sigma}, x^{(i)})$
$b \leftarrow \mathcal{A}(1^\lambda, \sigma, k_\sigma, (x^{(i)}, y_{1-\sigma}^{(i)})_{i \in [N(\lambda)]})$
Output b

$\mathsf{Exp}^{\mathsf{sec}}_{\mathcal{A},N,\sigma,1}(\lambda)$:

$(k_0, k_1) \leftarrow \mathsf{PCF.Gen}(1^\lambda)$
For $i = 1, \ldots, N(\lambda)$:
$\quad x^{(i)} \xleftarrow{\$} \{0,1\}^{n(\lambda)}$
$\quad y_\sigma^{(i)} \leftarrow \mathsf{PCF.Eval}(\sigma, k_\sigma, x^{(i)})$
$\quad y_{1-\sigma}^{(i)} \leftarrow \mathsf{RSample}(1^\lambda, \sigma, y_\sigma^{(i)})$
$b \leftarrow \mathcal{A}(1^\lambda, \sigma, k_\sigma, (x^{(i)}, y_{1-\sigma}^{(i)})_{i \in [N(\lambda)]})$
Output b

Fig. 7. Security of a PCF. Here, $\mathsf{RSample}$ is the algorithm for reverse sampling $\mathcal{Y}$ as per Definition 11.

Definition 15 (Programmability of a PCF for OLE, [8]). A pseudorandom correlation function $\mathsf{PCF} = (\mathsf{PCF.Gen}, \mathsf{PCF.Eval})$ for the OLE correlation supports *reusable inputs* if there exists an algorithm $\mathsf{PCF.Gen_p}$ that takes additional random inputs $\rho_0, \rho_1 \in \{0,1\}^\star$ such that:

- **Indistinguishability.** The following holds:

$$\{(k_0, k_1)\colon (k_0, k_1) \leftarrow \mathsf{PCF.Gen}(1^\lambda)\} \stackrel{c}{\approx} \{(k_0, k_1)\colon (\rho_0, \rho_1) \leftarrow \$, (k_0, k_1) \leftarrow \mathsf{PCF.Gen_p}(1^\lambda; \rho_0, \rho_1)\} \ .$$

- **Programmability.** There exist public efficiently computable functions f_0, f_1 for which

$$\Pr\left[\begin{cases} a = f_0(\rho_0, x) \\ b = f_1(\rho_1, x) \end{cases} : \begin{array}{l} \rho_0, \rho_1 \leftarrow \$, (k_0, k_1) \leftarrow \mathsf{PCF.Gen_p}(1^\lambda; \rho_0, \rho_1) \\ (a, c_0) \leftarrow \mathsf{PCF.Eval}(0, k_0, x) \\ (b, c_1) \leftarrow \mathsf{PCF.Eval}(1, k_1, x) \end{array}\right] \geq 1 - \mathsf{negl}(\lambda) \ .$$

- **Security.** for any $\sigma \in \{0,1\}$, the following distributions are computationally indistinguishable:

$$\{(k_\sigma, (\rho_0, \rho_1))\colon (\rho_0, \rho_1) \leftarrow \$, (k_0, k_1) \leftarrow \mathsf{PCF.Gen_p}(1^\lambda; \rho_0, \rho_1)\}, \text{ and}$$

$$\{(k_\sigma, \{\rho_\sigma, \tilde{\rho}\})\colon (\rho_0, \rho_1, \tilde{\rho}) \leftarrow \$, (k_0, k_1) \leftarrow \mathsf{PCF.Gen_p}(1^\lambda; \rho_0, \rho_1)\},$$

where the notation $\{\rho_\sigma, \tilde{\rho}\}$ means $(\rho_\sigma, \tilde{\rho})$ if $\sigma = 0$ and $(\tilde{\rho}, \rho_\sigma)$ if $\sigma = 1$.

A.4 Universal Composability

We refer the reader to [17] for details on the universal composability framework. The framework is based on the real/ideal paradigm for arguing about the security of a protocol. We say that a protocol π UC-realises (with computational security) an ideal functionality $\mathcal{F}$ in the presence of static semi-honest adversary corrupting at most t parties, if for any *p.p.t.* static semi-honest t-adversary $\mathcal{A}$ and any *p.p.t.* environment $\mathcal{Z}$, there exists a *p.p.t* ideal-model t-adversary Sim such that the output distribution of $\mathcal{Z}$ in the ideal-model computation of $\mathcal{F}$ with Sim is *computationally indistinguishable* from its output distribution in the real-model execution of π with $\mathcal{A}$. The composition theorem of [17] states the following.

Theorem 7 ([17], informal). *Let ρ be a protocol that UC-realizes $\mathcal{F}$ in the presence of adaptive semi-honest t-adversaries, and let π be a protocol that UC-realizes $\mathcal{G}$ in the $\mathcal{F}$-hybrid model in the presence of adaptive semi-honest t-adversaries. Then, for any* p.p.t. *adaptive semi-honest t-adversary $\mathcal{A}$ and any* p.p.t. *environment $\mathcal{Z}$, there exists a* p.p.t. *adaptive semi-honest t-adversary Sim in the $\mathcal{F}$-hybrid model such that the output distribution of $\mathcal{Z}$ when interacting with the protocol π and Sim is computationally indistinguishable from its output distribution when interacting with the protocol π^ρ (where every call to $\mathcal{F}$ is replaced by an execution of ρ) and $\mathcal{A}$ in the real model.*

B Proof of Lemma 1

We design the extended evaluation algorithm $\mathsf{ExtEval}(\sigma, \mathsf{ek}_\sigma, \mathsf{M}_\sigma, (\mathsf{I}_\sigma^{(1)}, \dots, \mathsf{I}_\sigma^{(\rho)}), P)$ using the original evaluation algorithm Eval. The idea is to recursively include the memory value in the computation. First, we define the modified input converting algorithm $\mathsf{ConvertInput}'(\sigma, \mathsf{ek}_\sigma, \mathsf{I}^x, \mathsf{M}^z)$ which converts input values into a memory values as follows

- $\mathsf{ConvertInput}'(\sigma, \mathsf{ek}_\sigma, \mathsf{I}^x, \mathsf{M}^z) \to \mathsf{M}^{x\cdot z}$:
  This algorithm runs $\mathsf{Mult}(\sigma, \mathsf{ek}_\sigma, \mathsf{I}^x, \mathsf{M}^z) \to \mathsf{M}^{x\cdot z}$ and returns $\mathsf{M}^{x\cdot z}$.

In other words, for any input $x \neq z$ for which an input value I^x is provided, we can compute a memory value $\mathsf{M}^{x\cdot z}$ that is a memory value of $z \cdot x$. In this way, we make sure that each memory value represents a memory value of a multiple of z. Regarding this new shape of memory values, the two other algorithms Add' which adds memory values and Mult' which multiplies input and memory values can be simply defined as the original algorithms of Eval. Namely,

- $\mathsf{Add}'(\mathsf{M}^{x\cdot z}, \mathsf{M}^{y\cdot z}) \to \mathsf{M}^{z\cdot(x+y)}$:
  This algorithm runs $\mathsf{Add}(\mathsf{M}^{x\cdot z}, \mathsf{M}^{y\cdot z}) \to \mathsf{M}^{(x+y)\cdot z}$.

- $\mathsf{Mult}'(\mathsf{I}^x, \mathsf{M}^{y\cdot z}) \to \mathsf{M}^{x\cdot y\cdot z}$:
  This algorithm runs $\mathsf{Mult}(\mathsf{I}^x, \mathsf{M}^{y\cdot z}) \to \mathsf{M}^{x\cdot y\cdot z}$.

Finally, the output algorithm $\mathsf{Output}'(\sigma, \mathsf{ek}_\sigma, \mathsf{M}^{x\cdot z}, n)$ also works as the original algorithm of Eval:

- $\mathsf{Output}'(\sigma, \mathsf{ek}_\sigma, \mathsf{M}^{x\cdot z}, n) \to (x \cdot z) \mod n$:
  This algorithm runs $\mathsf{Output}(\sigma, \mathsf{ek}_\sigma, \mathsf{M}^{x\cdot z}, n) \to (x \cdot z) \mod n$

Conditioned on Eval satisfying the correctness property of HSS (Definition 2), algorithm $\mathsf{ExtEval}$ also works correctly and on input $(\sigma, \mathsf{ek}_\sigma, \mathsf{M}_\sigma, (\mathsf{I}_\sigma^{(1)}, \dots, \mathsf{I}_\sigma^{(\rho)}), P)$ outputs $z \cdot P(x^{(1)}, \dots, x^{(\rho)})$.

□

C New Variants of HSS from DCR

In this section we provide instantiations for the three new variants of HSS introduced in Section 4 under the DCR assumption, therefore proving Theorem 2 and Theorem 2. In fact, our goal is to show that the HSS scheme introduced by Orlandi, Scholl, and Yakoubov in [37] that supports the class of RMS programs and works under the DCR assumption satisfies the properties of all our three definitions. First, we recall the following lemma due to [37], where they introduce a distributed discrete logarithm algorithm for a subset of $\mathbb{Z}^*_{N^2}$, where $N = pq$ for λ-bit primes p and q.

Lemma 5. *There exists an algorithm* $\mathsf{DDLog}_N(g)$ *for which the following holds: Let* $g_0, g_1 \in \mathbb{Z}^*_{N^2}$*, such that* $g_0 = g_1(1+N)^x (\text{mod } N^2)$*. If* $z_0 = \mathsf{DDLog}_N(g_0)$ *and* $z_1 = \mathsf{DDLog}_N(g_1)$*, then* $z_0 - z_1 = x (\text{mod } N)$*.*

More precisely, $\mathsf{DDLog}_N(g)$ works as follows:

- $\mathsf{DDLog}_N(g)$
  - Write $g = h + h'N$, where $h, h' < N$, using the division algorithm.
  - Output $z = h'h^{-1} \bmod N$.

We now recall the HSS construction of [37] based on circular-secure Paillier encryption (description A.1). The input space of the scheme is $\mathbb{Z}_N$ for a Blum integer $N = pq$.

Construction 4 (HSS from Paillier, [37]). Let $2^{-\kappa}$ be the correctness error. Let $N = pq$ be a Blum integer. Let $\mathcal{P}$ be the set of programs supported by the scheme, and B_{msg} be the magnitude bound of programs in $\mathcal{P}$. We require that $B_{\mathsf{msg}} = N/2^\kappa$. Let $\mathsf{BG} = (\mathsf{BG.KeyGen}, \mathsf{BG.Enc}, \mathsf{BG.Dec})$ be the circular-secure Paillier encryption as in Description A.1.

- $\mathsf{Setup}(1^\lambda)$:
  - Run $(\mathsf{BG.pk}, \mathsf{BG.sk}) \leftarrow \mathsf{BG.KeyGen}(1^\lambda)$, and parse them as $\mathsf{BG.pk} = (N, \mathbf{g}, \hat{g})$, and $\mathsf{BG.sk} = \mathbf{d} = (d^{(0)}, \ldots, d^{(\ell-1)})$.
  - Sample $\langle 1 \rangle_0$ as a random element of $[2^\kappa]$, and set $\langle 1 \rangle_1 := \langle 1 \rangle_0 - 1 \bmod N$.
  - For each $i \in [\ell]$, set $\langle d^{(i)} \rangle_0$ to be a random element of $[2^\kappa]$, and set $\langle d^{(i)} \rangle_1 := \langle d^{(i)} \rangle_0 - d^{(i)} \bmod N$.
  - For $i \in [\ell]$, compute $D^{(i)} \leftarrow \mathsf{BG.Enc}(\mathsf{BG.pk}, d^{(i)})$.
  - Sample a PRF key k_{prf} for a PRF F that outputs values in $\mathbb{Z}_N$.
  - Set and output $\mathsf{pk} = (\mathsf{BG.pk}, D^{(0)}, \ldots, D^{(\ell-1)})$, and $\mathsf{ek}_\sigma = (k_{\mathsf{prf}}, \langle 1 \rangle_\sigma, \langle d^{(0)} \rangle_\sigma, \ldots, \langle d^{(\ell-1)} \rangle_\sigma)$ for each $\sigma \in \{0,1\}$.
- $\mathsf{Input}(\mathsf{pk}, x)$
  - Parse $\mathsf{pk} = (\mathsf{BG.pk}, D^{(0)}, \ldots, D^{(\ell-1)})$, and $\mathsf{BG.pk} = (\mathbf{g}, \hat{g})$, and $D^{(i)} = (\mathbf{c}^{(i)}, \hat{c}^{(i)})$ for $i \in [\ell]$.
  - Compute $X \leftarrow \mathsf{BG.Enc}(\mathsf{BG.pk}, x)$.
  - For $i \in [\ell]$, compute $X^{(i)} \leftarrow (\mathbf{g}^{r'_i} \cdot (\mathbf{c}^{(i)})^x, \hat{g}^{r'_i} \cdot (\hat{c}^{(i)})^x)$, where $r'_i \xleftarrow{\$} \mathbb{Z}_N$.
  - Set $\mathsf{I} = (X, X^{(0)}, \ldots, X^{(\ell-1)})$, and output $(\mathsf{I}_0 = \mathsf{I}, \mathsf{I}_1 = \mathsf{I})$.
- $\mathsf{Eval}(\sigma, \mathsf{ek}_\sigma, (\mathsf{I}^{(0)}, \ldots, \mathsf{I}^{(n)}), P)$
  This function is divided into the following sub-modules:
  - $\mathsf{ConvertInput}(\sigma, \mathsf{ek}_\sigma, \mathsf{I}_x = (X, X^{(0)}, \ldots, X^{(\ell-1)}))$
    - Set $\mathsf{M}^1_\sigma = (\langle 1 \rangle_\sigma, \langle d^{(0)} \rangle_\sigma, \ldots, \langle d^{(\ell-1)} \rangle_\sigma)$ for $\sigma \in \{0,1\}$.
    - Compute $\mathsf{M}^x_\sigma \leftarrow \mathsf{Mult}(\sigma, \mathsf{ek}_\sigma, \mathsf{I}^x, \mathsf{M}^1_\sigma)$.
  - $\mathsf{Add}(\sigma, \mathsf{ek}_\sigma, \mathsf{M}^x_\sigma, \mathsf{M}^y_\sigma)$
    - Parse $\mathsf{M}^x_\sigma = (\langle x \rangle_\sigma, \langle xd^{(0)} \rangle_\sigma, \ldots, \langle xd^{(\ell-1)} \rangle_\sigma)$, and $\mathsf{M}^y_\sigma = (\langle y \rangle_\sigma, \langle yd^{(0)} \rangle_\sigma, \ldots, \langle yd^{(\ell-1)} \rangle_\sigma)$.
    - Compute $\langle z \rangle_\sigma = \langle x \rangle_\sigma + \langle y \rangle_\sigma$, and $\langle zd^{(i)} \rangle_\sigma = \langle xd^{(i)} \rangle_\sigma + \langle yd^{(i)} \rangle_\sigma$ for $i \in [\ell]$.
    - Output $\mathsf{M}^z_\sigma = (\langle z \rangle_\sigma, \langle zd^{(0)} \rangle_\sigma, \ldots, \langle zd^{(\ell-1)} \rangle_\sigma)$.
  - $\mathsf{Mult}(\sigma, \mathsf{ek}_\sigma, \mathsf{I}^x, \mathsf{M}^y_\sigma)$
    - Parse $\mathsf{I}^x = (X, X^{(0)}, \ldots, X^{(\ell-1)})$ and $\mathsf{M}^y_\sigma = (\langle y \rangle_\sigma, \langle yd^{(0)} \rangle_\sigma, \ldots, \langle yd^{(\ell-1)} \rangle_\sigma)$.
    - Parse $X = (c_0, \ldots, c_{\ell-1}, \hat{c})$, and $X^{(i)} = (c^{(i)}_0, \ldots, c^{(i)}_{\ell-1}, \hat{c}^{(i)})$ for $i \in [\ell]$.
    - Compute $\langle z \rangle_\sigma = \mathsf{DDLog}_N(\mathsf{ct}'_\sigma) (\text{mod } N) + F_{k_{\mathsf{prf}}}(\mathsf{id})$, where

$$\mathsf{ct}'_\sigma = (\hat{c})^{\langle y \rangle_\sigma} \cdot \left(\prod_{i=0}^{\ell-1} c_i^{-\langle yd^{(i)} \rangle_\sigma} \right) (\text{mod } N^2)$$

- For $j \in [\ell]$ compute $\left\langle zd^{(j)} \right\rangle_\sigma = \mathsf{DDLog}_N \left(\mathsf{ct}'_{\sigma,j}\right) \pmod N + F_{k_{\mathsf{prf}}}(\mathsf{id})$, where

$$\mathsf{ct}'_{\sigma,j} = (\hat{c}^{(j)})^{\langle y \rangle_\sigma} \cdot \left(\prod_{i=0}^{\ell-1} (c_i^{(j)})^{-\left\langle yd^{(i)} \right\rangle_\sigma} \right) \pmod{N^2}$$

- Output $\mathsf{M}_\sigma^z = (\langle z \rangle_\sigma, \left\langle zd^{(0)} \right\rangle_\sigma, \ldots, \left\langle zd^{(\ell-1)} \right\rangle_\sigma)$.

- $\mathsf{Output}(\sigma, \mathsf{ek}_\sigma, \mathsf{M}_\sigma^z, n_{\mathsf{out}})$
  - Parse $\mathsf{M}_\sigma^z = (\langle z \rangle_\sigma, \left\langle zd^{(0)} \right\rangle_\sigma, \ldots, \left\langle zd^{(\ell-1)} \right\rangle_\sigma)$.
  - Output $\langle z \rangle_\sigma \pmod{n_{\mathsf{out}}}$.

HSS Following the RMS Template from DCR. We show that construction 4 satisfies definition 4.

Proof. We show how the MemGen algorithm of the template work in this construction. One can see that the other algorithms of the HSS construction exactly follow the template. We define the memory generation algorithm as follows:

- $\mathsf{MemGen}(\sigma, \mathsf{ek}_\sigma, x) \to \mathsf{M}_\sigma^x$
  - If $x = 1$, do:
    - Parse $\mathsf{ek}_\sigma = (k_{\mathsf{prf}}, \langle 1 \rangle_\sigma, \left\langle d^{(0)} \right\rangle_\sigma, \ldots, \left\langle d^{(\ell-1)} \right\rangle_\sigma)$.
    - Output $\mathsf{M}_\sigma^1 = (\langle 1 \rangle_\sigma, \left\langle d^{(0)} \right\rangle_\sigma, \ldots, \left\langle d^{(\ell-1)} \right\rangle_\sigma)$.
  - Else, do:
    - Run $(\mathsf{I}_0^x, \mathsf{I}_1^x) \leftarrow \mathsf{Input}(\mathsf{pk}, x)$.
    - Run $\mathsf{M}_\sigma^x \leftarrow \mathsf{ConvertInput}(\sigma, \mathsf{ek}_\sigma, \mathsf{I}_\sigma^x)$.
    - Output M_σ^x.

It is easy to see that the outputs of this algorithm are additively homomorphic. This follows from the fact that for any $x \neq 1 \in \mathcal{I}$, this algorithm uses the Input and Eval.ConvertInput algorithms to generate the memory values. Thus, if the HSS scheme works correctly, the generated memory values are intrinsically homomorphic. More specifically, for an input $z \in \mathcal{I}$, the memory value M_σ^z is of the form $\mathsf{M}_\sigma^z = (\langle z \rangle_\sigma, \left\langle zd^{(0)} \right\rangle_\sigma, \ldots, \left\langle zd^{(\ell-1)} \right\rangle_\sigma)$. Furthermore, when $x = 1$, this algorithm outputs a valid share for the vector $(1, d^{(0)}, \ldots, d^{(\ell-1)})$.

□

HSS with Simulatable Memory Values from DCR. We show that Construction 4 satisfies Definition 5.

Proof. Regarding Definition 5, we need to show that there exist two algorithms Sim_0 and Sim_1 that simulate the output of MemGen. We define them as follows:

- $\mathsf{Sim}_0(1^\lambda) \to \mathsf{M}_0$
  - Sample a random vector $(t, t_0, \ldots, t_{\ell-1}) \xleftarrow{\$} [2^\kappa \cdot N]^{\ell+1}$.
  - Output $\mathsf{M}_0 = (t, t_0, \ldots, t_{\ell-1})$.
- $\mathsf{Sim}_1(\mathsf{M}, z, (\mathsf{ek}_0, \mathsf{ek}_1)) \to \mathsf{M}_1$
  - Parse $\mathsf{ek}_\sigma = (\langle 1 \rangle_\sigma, \left\langle d^{(0)} \right\rangle_\sigma, \ldots, \left\langle d^{(\ell-1)} \right\rangle_\sigma)$ for both $\sigma \in \{0, 1\}$.
  - For $i \in [\ell]$ reconstruct $d^{(i)} = \left\langle d^{(i)} \right\rangle_0 - \left\langle d^{(i)} \right\rangle_1 \bmod N$.
  - Compute and output $\mathsf{M}_1 = \mathsf{M}_0 - (z, zd^{(0)}, \ldots, zd^{(\ell-1)})$.

We prove the following two properties regarding the simulation algorithms:

Simulation Correctness. For any $z \in \mathbb{Z}_N$, it holds that

$$\mathsf{M}_0 - \mathsf{M}_1 = (z, zd^{(0)}, \ldots, zd^{(\ell-1)}),$$

where $\mathsf{M}_0 \leftarrow \mathsf{Sim}_0(1^\lambda)$, and $\mathsf{M}_1 \leftarrow \mathsf{Sim}_1(\mathsf{M}, z, (\mathsf{ek}_0, \mathsf{ek}_1))$. Therefore, the simulated memory values of z are correctly formed as subtractive shares of vector $(z, zd^{(0)}, \ldots, zd^{(\ell-1)})$. Thus, they are valid shares. This guarantees the correctness of multiplication between this values and real input values, and finally the correctness of $\mathsf{ExtEval}$ in Lemma 1 when M_σ is simulated.

Simulation Security. We need to prove that for any $x \in \mathcal{I}$, it holds that

$$(z, \mathsf{M}_0) \approx_s (z, \mathsf{M}_1),$$

where $\mathsf{M}_1 \leftarrow \mathsf{Sim}_1(\mathsf{M}_0, z, (\mathsf{ek}_0, \mathsf{ek}_1))$, and $\mathsf{M}_0 \leftarrow \mathsf{Sim}_0(1^\lambda)$.
Note that $\mathsf{M}_1 = \mathsf{M}_0 - (z, zd^{(0)}, \ldots, zd^{(\ell-1)})$, where each element of M_0 is chosen uniformly from $\mathbb{Z}_{2^\kappa N}$. Also, in a fixed vector $(z, zd^{(0)}, \ldots, zd^{(\ell-1)})$, x and each $zd^{(i)}$ for $i \in [\ell]$ are elements of $\mathbb{Z}_N$. Therefore, the distribution of each element of M_1 is within the statistical distance $2^{-\kappa}$ of the uniform distribution over $\mathbb{Z}_{2^\kappa N}$ which is the distribution of M_0.

□

Staged HSS from DCR. We prove that assuming the hardness of DCR, Construction 4 satisfies Definition 6.

Proof. We explicitly define four algorithms $(\overline{\mathsf{Input}}_0, \overline{\mathsf{Input}}_1)$ and $(\overline{\mathsf{Eval}}_0, \overline{\mathsf{Eval}}_1)$ according to Definition 6. We define the four algorithms as follows:

- $\overline{\mathsf{Input}}_0(\mathsf{pp}) \to (\bar{\mathsf{I}}_0, \mathsf{aux})$
  - Parse $\mathsf{pp} = (\mathsf{BG.pk}, D^{(0)}, \ldots, D^{(\ell-1)})$, and $\mathsf{BG.pk} = (N, \mathbf{g}, \hat{g})$.
  - Sample $r \xleftarrow{\$} \mathbb{Z}_N$ and compute $\mathsf{ct}_{\mathsf{ind}} = \mathbf{g}^r$.
  - For $i \in [\ell]$ do:
    - ∗ Sample $r_i \xleftarrow{\$} \mathbb{Z}_N$.
    - ∗ Compute $\mathsf{ct}_{\mathsf{ind}}^{(i)} = \mathbf{g}^{r_i}$.
  - Set $\bar{\mathsf{I}}_0 = (\mathsf{ct}_{\mathsf{ind}}, \mathsf{ct}_{\mathsf{ind}}^{(0)}, \ldots, \mathsf{ct}_{\mathsf{ind}}^{(\ell-1)})$.
  - Set $\mathsf{aux} = (\mathbf{g}^r, \hat{g}^r, \{\mathbf{g}^{r_i}\}_{i\in[\ell]}, \{\hat{g}^{r_i}\}_{i\in[\ell]})$.
  - Output $(\bar{\mathsf{I}}_0, \mathsf{aux})$.

- $\overline{\mathsf{Input}}_1(\mathsf{pp}, x, \mathsf{aux}, (\mathsf{ek}_0, \mathsf{ek}_1)) \to \bar{\mathsf{I}}_1$
  - Parse $\mathsf{pp} = (\mathsf{BG.pk}, D^{(0)}, \ldots, D^{(\ell-1)})$, $\mathsf{BG.pk} = (N, \mathbf{g}, \hat{g})$, $\mathsf{aux} = (\mathbf{g}^r, \hat{g}^r, \{\mathbf{g}^{r_i}\}_{i\in[\ell]}, \{\hat{g}^{r_i}\}_{i\in[\ell]})$, and $\mathsf{ek}_\sigma = (k_{\mathsf{prf}}, \langle d^{(0)}\rangle_\sigma, \ldots, \langle d^{(\ell-1)}\rangle_\sigma)$ for $\sigma \in \{0,1\}$.
  - Compute $\mathsf{ct} = (\mathbf{g}^r, \hat{g}^r \cdot (1+N)^x)$.
  - For $i \in [\ell]$ do
    - ∗ Reconstruct $d^{(i)} = \langle d^{(i)}\rangle_0 - \langle d^{(i)}\rangle_1 \mod N$.
    - ∗ Compute $\mathsf{ct}^{(i)} = (\mathbf{g}^{r_i}, \hat{g}^{r_i} \cdot (1+N)^{xd^{(i)}})$.
  - Output $\bar{\mathsf{I}}_1 = (\mathsf{ct}, \mathsf{ct}^{(0)}, \ldots, \mathsf{ct}^{(\ell-1)})$.

- $\overline{\mathsf{Eval}}_0(\mathsf{ek}_0, (\bar{\mathsf{I}}_0^{(1)}, \ldots, \bar{\mathsf{I}}_0^{(\rho)}), P) \to \mathsf{M}_0$
  - $\overline{\mathsf{ConvertInput}}_0(\mathsf{ek}_0, \bar{\mathsf{I}}_x)$ //same as in Eval
    - Parse $\mathsf{ek}_0 = (\langle 1\rangle_0, \langle d^{(0)}\rangle_0, \ldots, \langle d^{(\ell-1)}\rangle_0)$.
    - Set $\mathsf{M}_0^1 = (\langle 1\rangle_0, \langle d^{(0)}\rangle_0, \ldots, \langle d^{(\ell-1)}\rangle_0)$.
    - Compute $\mathsf{M}_0^x \leftarrow \overline{\mathsf{Mult}}_0(\mathsf{ek}_0, \bar{\mathsf{I}}_0, \mathsf{M}_0^1)$.
  - $\overline{\mathsf{Add}}_0(\mathsf{ek}_0, \mathsf{M}_0^x, \mathsf{M}_0^y)$ //same as in Eval
    - Parse $\mathsf{M}_0^x = (\langle x\rangle_0, \langle xd^{(0)}\rangle_0, \ldots, \langle xd^{(\ell-1)}\rangle_0)$, and $\mathsf{M}_0^y = (\langle y\rangle_0, \langle yd^{(0)}\rangle_0, \ldots, \langle yd^{(\ell-1)}\rangle_0)$.
    - Compute $\langle z\rangle_0 = \langle x\rangle_0 + \langle y\rangle_0$, and $\langle zd^{(i)}\rangle_0 = \langle xd^{(i)}\rangle_0 + \langle yd^{(i)}\rangle_0$ for $i \in [\ell]$.

- Output $\mathsf{M}_0^z = (\langle z \rangle_0, \langle zd^{(0)} \rangle_0, \ldots, \langle zd^{(\ell-1)} \rangle_0)$.

- $\overline{\mathsf{Mult}}_0(\mathsf{ek}_0, \bar{\mathsf{I}}_0^x, \mathsf{M}_0^y)$
  - Parse $\bar{\mathsf{I}}_0^x = (\mathsf{ct}_{\mathsf{ind}}, \mathsf{ct}_{\mathsf{ind}}^{(0)}, \ldots, \mathsf{ct}_{\mathsf{ind}}^{(\ell-1)})$, and $\mathsf{M}_0^y = (\langle y \rangle_0, \langle yd^{(0)} \rangle_0, \ldots, \langle yd^{(\ell-1)} \rangle_0)$.
  - Parse $\mathsf{ct}_{\mathsf{ind}} = (c_0, \ldots, c_{\ell-1})$, and $\mathsf{ct}_{\mathsf{ind}}^{(i)} = (c_0^{(i)}, \ldots, c_{\ell-1}^{(i)})$ for $i \in [\ell]$.
  - Compute $\langle z \rangle_0 = \mathsf{DDLog}_N(\mathsf{ct}')(\text{mod } N) + F_{k_{\mathsf{prf}}}(\mathsf{id})$, where

$$\mathsf{ct}' = \prod_{i=0}^{\ell-1} (c_i)^{-\langle yd^{(i)} \rangle_0} (\text{mod } N^2).$$

  - For $j \in [\ell]$, compute $\langle zd^{(j)} \rangle_0 = \mathsf{DDLog}_N(\mathsf{ct}_j')(\text{mod } N) + F_{k_{\mathsf{prf}}}(\mathsf{id})$, where

$$\mathsf{ct}_j' = \prod_{i=0}^{\ell-1} (c_i^{(j)})^{-\langle yd^{(i)} \rangle_0} (\text{mod } N^2).$$

  - Output $\mathsf{M}_0^z = (\langle z \rangle_0, \langle zd^{(0)} \rangle_0, \ldots, \langle zd^{(\ell-1)} \rangle_0)$.

- $\overline{\mathsf{Eval}}_1(\mathsf{ek}_1, (\bar{\mathsf{I}}_1^{(1)}, \ldots, \bar{\mathsf{I}}_1^{(\rho)}), (x^{(1)}, \ldots, x^{(\rho)}), P) \to \mathsf{M}_1$

  - $\overline{\mathsf{ConvertInput}}_1(\mathsf{ek}_1, \bar{\mathsf{I}}_x, x)$ //same as in Eval
    - Parse $\mathsf{ek}_1 = (\langle 1 \rangle_1, \langle d^{(0)} \rangle_1, \ldots, \langle d^{(\ell-1)} \rangle_1)$.
    - Set $\mathsf{M}_1^1 = (\langle 1 \rangle_1, \langle d^{(0)} \rangle_1, \ldots, \langle d^{(\ell-1)} \rangle_1)$.
    - Compute $\mathsf{M}_1^x \leftarrow \overline{\mathsf{Mult}}_1(\mathsf{ek}_1, \bar{\mathsf{I}}_1, \mathsf{M}_1^1, x)$.

  - $\overline{\mathsf{Add}}_1(\mathsf{ek}_1, \mathsf{M}_1^x, \mathsf{M}_1^y)$ //same as in Eval
    - Parse $\mathsf{M}_1^x = (\langle x \rangle_1, \langle xd^{(0)} \rangle_1, \ldots, \langle xd^{(\ell-1)} \rangle_1)$, and $\mathsf{M}_1^y = (\langle y \rangle_1, \langle yd^{(0)} \rangle_1, \ldots, \langle yd^{(\ell-1)} \rangle_1)$.
    - Compute $\langle z \rangle_1 = \langle x \rangle_1 + \langle y \rangle_1$, and $\langle zd^{(i)} \rangle_1 = \langle xd^{(i)} \rangle_1 + \langle yd^{(i)} \rangle_1$ for $i \in [\ell]$.
    - Output $\mathsf{M}_1^z = (\langle z \rangle_1, \langle zd^{(0)} \rangle_1, \ldots, \langle zd^{(\ell-1)} \rangle_1)$.

  - $\overline{\mathsf{Mult}}_1(\mathsf{ek}_0, \bar{\mathsf{I}}_0^x, \mathsf{M}_0^y, y)$
    - Parse $\bar{\mathsf{I}}_1^x = (\mathsf{ct}, \mathsf{ct}^{(0)}, \ldots, \mathsf{ct}^{(\ell-1)})$, and $\mathsf{M}_1^y = (\langle y \rangle_1, \langle yd^{(0)} \rangle_1, \ldots, \langle yd^{(\ell-1)} \rangle_1)$.
    - Parse $\mathsf{ct} = (c_0, \ldots, c_{\ell-1}, \hat{c})$, and $\mathsf{ct}^{(i)} = (c_0^{(i)}, \ldots, c_{\ell-1}^{(i)}, \hat{c}^{(i)})$ for $i \in [\ell]$.
    - Compute $\langle z \rangle_1 = \mathsf{DDLog}_N(\mathsf{ct}')(\text{mod } N) + F_{k_{\mathsf{prf}}}(\mathsf{id})$, where

$$\mathsf{ct}' = (\hat{c})^y \cdot \prod_{i=0}^{\ell-1} (c_i)^{-\langle yd^{(i)} \rangle_1} (\text{mod } N^2).$$

    - For $j \in [\ell]$, compute $\langle zd^{(j)} \rangle_1 = \mathsf{DDLog}_N(\mathsf{ct}_j')(\text{mod } N) + F_{k_{\mathsf{prf}}}(\mathsf{id})$, where

$$\mathsf{ct}_j' = (\hat{c}^{(j)})^y \cdot \prod_{i=0}^{\ell-1} (c_i^{(j)})^{-\langle yd^{(i)} \rangle_1} (\text{mod } N^2).$$

    - Output $\mathsf{M}_1^z = (\langle z \rangle_1, \langle zd^{(0)} \rangle_1, \ldots, \langle zd^{(\ell-1)} \rangle_1)$.

Correctness. We show that a memory value M_σ^y outputted by $\overline{\mathsf{Eval}}_\sigma$ is in fact party σ's subtractive share of vector $(y, yd^{(0)}, \ldots, yd^{(\ell-1)})$, thus it's valid. This guarantees the correctness of ExtEval algorithm when given as input a staged memory value and a vector of original input values.

Since the new evaluation algorithms $\overline{\mathsf{Eval}}_0$ and $\overline{\mathsf{Eval}}_1$ work the same as the original evaluation algorithm Eval except for the multiplication instruction, we briefly prove the correctness of multiplication in the following. Let $x, y \in \mathcal{I}$ be any two arbitrary input values. We show that

$$\Pr\left[z_0 - z_1 = xy\right] \geq 1 - \mathsf{negl}(\lambda),$$

and

$$\Pr\left[(zd^{(i)})_0 - (zd^{(i)})_1 = xyd^{(i)}\right] \geq 1 - \mathsf{negl}(\lambda),$$

for all $i \in [\ell]$, where
$\mathsf{M}_0^z = \left(z_0, (zd^{(0)})_0, \ldots, (zd^{(\ell-1)})_0\right) \leftarrow \overline{\mathsf{Mult}}_0(\mathsf{ek}_0, \bar{\mathsf{I}}_0^x, \mathsf{M}_0^y)$,
$\mathsf{M}_1^z = \left(z_1, (zd^{(0)})_1, \ldots, (zd^{(\ell-1)})_1\right) \leftarrow \overline{\mathsf{Mult}}_1(\mathsf{ek}_1, \bar{\mathsf{I}}_1^x, \mathsf{M}_1^y, y)$,
$(\bar{\mathsf{I}}_0^b, \mathsf{aux}^b) \leftarrow \overline{\mathsf{Input}}_0(\mathsf{pk})$ for $b \in \{x, y\}$,
$\bar{\mathsf{I}}_1^b \leftarrow \overline{\mathsf{Input}}_1(\mathsf{pk}, b, \mathsf{aux}^b)$, for $b \in \{x, y\}$,
$\mathsf{M}_0^y \leftarrow \overline{\mathsf{ConvertInput}}_0(\mathsf{ek}_0, \bar{\mathsf{I}}_0^y)$,
$\mathsf{M}_1^y \leftarrow \overline{\mathsf{ConvertInput}}_1(\mathsf{ek}_1, \bar{\mathsf{I}}_1^y, y)$, and
$(\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \leftarrow \mathsf{Setup}(1^\lambda)$.

Regarding how $\overline{\mathsf{Mult}}_0$ and $\overline{\mathsf{Mult}}_1$ works, it holds that $z_b = \mathsf{DDLog}_N(\mathsf{ct}'_b)$, for $b \in \{0, 1\}$. Thus, by Lemma 5, it's enough to prove that $\mathsf{ct}'_0 \cdot \mathsf{ct}'_1 = (1+N)^{xy}$. We have

$$\begin{aligned}
\mathsf{ct}'_0 \cdot \mathsf{ct}'_1 &= \prod_{i=0}^{\ell-1} (c_i)^{-\langle yd^{(i)} \rangle_0} \cdot (\hat{c})^y \cdot \prod_{i=0}^{\ell-1} (c_i)^{-\langle yd^{(i)} \rangle_1} \\
&= (\hat{c})^y \cdot \prod_{i=1}^{\ell} (c_i)^{-yd^{(i)}} \\
&= (1+N)^{xy} \cdot \prod_{i=1}^{\ell} (c_i)^{yd^{(i)}} \cdot \prod_{i=1}^{\ell} (c_i)^{-yd^{(i)}} \\
&= (1+N)^{xy} (\bmod\ N^2).
\end{aligned}$$

The equation $\mathsf{ct}_0'^{(j)} \cdot \mathsf{ct}_1'^{(j)} = (1+N)^{xyd^{(j)}}$ for all $j \in [\ell]$ is proved similarly.

Security. Outputs of the $\overline{\mathsf{Input}}_1$ algorithm are in fact in the same form as the Input algorithm. More precisely, they are both Paillier encryptions of the vector $(x, xd^{(0)}, \ldots, xd^{(\ell-1)})$, where d is the secret key of the encryption scheme. Therefore, they are computationally indistinguishable.

□

D Proofs of CPRF Generic Constructions

In this section we prove that our transformations from homomorphic secret sharing to constrained pseudorandom functions for inner-product and NC^1 constraints satisfy the correctness and security properties of a CPRF.

D.1 CPRF for Inner-Product from HSS (Proof of Theorem 3)

We now prove correctness and security of Construction 1, starting with correctness.

Correctness. The idea of the scheme is to choose a random key k for the PRF F, and use HSS to compute shares of $\langle \mathbf{x}, \mathbf{z} \rangle \cdot F_k(\mathbf{x})$, respectively owned by the master key owner and the constrained key owner. It is easy to verify that if HSS is correct (with simulated memory values) and if $\langle \mathbf{x}, \mathbf{z} \rangle$, the two shares form substractive shares of $\langle \mathbf{x}, \mathbf{z} \rangle \cdot F_k(\mathbf{x}) = 0$ and therefore both evaluations match.

Pseudorandomness. Let us now prove selective security of our construction. The proof relies on a sequence of hybrid games. Let $\mathcal{A}$ denote a 1-key selective adversary against the pseudorandomness of the above construction. Selective security plays an important role as one needs to rely on the knowledge of the constraint $\mathbf{z}$ to answer evaluation oracle queries appropriately in the early stage of the proof.

Hybrid $\mathcal{H}_0$: This is the standard CPRF security game where the challenge is answered with the real PRF evaluation.

Hybrid $\mathcal{H}_1$: In this first hybrid game, we only change the definition of the evaluation oracle. Since we are in the selective setting, the challenger knows the constraint $\mathbf{z}$ before answering any evaluation query. Therefore, it can compute the constrained key $\mathsf{ck}_{\mathbf{z}}$ from the start. When asked for an evaluation query on input x, the challenger replies to it by computing $y_1 \leftarrow \mathsf{CEval}(\mathsf{pp}, \mathsf{ck}_{\mathbf{z}}, \mathbf{x})$ and returning $y_1 + \langle \mathbf{x}, \mathbf{z} \rangle \cdot F_k(\mathbf{x})$. The adversary's view remains identical to its view in $\mathcal{H}_0$ by correctness of HSS, therefore these two hybrid games are perfectly indistinguishable.

Hybrid $\mathcal{H}_2$: In this second hybrid game, we simply switch the memory value sampled via Sim_1 in the constrained secret key to memory value sampled from Sim_0. It is immediate that $\mathcal{H}_1$ and $\mathcal{H}_2$ are indistinguishable thanks to simulation security of the HSS.

Hybrid $\mathcal{H}_3$: We now remove the information about k in the constrained evaluation key as follows: instead of defining $(\mathsf{I}_0, \mathsf{I}_1) \leftarrow \mathsf{Input}(\mathsf{pk}, k)$, we set $(\mathsf{I}_0, \mathsf{I}_1) \leftarrow \mathsf{Input}(\mathsf{pk}, 0)$[13]. A PRF key $k \xleftarrow{\$} \mathcal{K}$ is still sampled by the challenger, and evaluation (and challenge) queries are still answered on input $\mathbf{x}$ by having the challenger computing $y_1 \leftarrow \mathsf{CEval}(\mathsf{pp}, \mathsf{ck}_{\mathbf{z}}, \mathbf{x})$ and returning $y_1 + \langle \mathbf{x}, \mathbf{z} \rangle \cdot F_k(\mathbf{x})$.

By HSS security (for $\sigma = 1$), we claim that $\mathcal{H}_2$ and $\mathcal{H}_3$ are computationally indistinguishable. Suppose that an adversary $\mathcal{A}$ could distinguish between these two hybrids, we construct an adversary $\mathcal{B}$ against HSS security as follows: $\mathcal{B}$ first samples a key $k \xleftarrow{\$} \mathcal{K}$ and submits message $(k, 0)$ to its HSS challenger. It gets back $(\mathsf{pk}, \mathsf{ek}_1, \mathsf{I}_1)$ where I_1 is the second half of $\mathsf{Input}(k)$ or $\mathsf{Input}(0)$, depending on whether k or 0 was encoded by the challenger. Then, $\mathcal{B}$ computes the constrained key $\mathsf{ck}_{\mathbf{z}}$ as in the previous game, by sampling memory values using $\mathsf{Sim}_0(1^\lambda)$. It answers $\mathcal{A}$'s evaluation (and challenge) queries by computing $y_1 \leftarrow \mathsf{CEval}(\mathsf{pp}, \mathsf{ck}_{\mathbf{z}}, \mathbf{x})$ and returning $y_1 + \langle \mathbf{x}, \mathbf{z} \rangle \cdot F_k(\mathbf{x})$. When $\mathcal{A}$ halts with some output b', so does $\mathcal{B}$. It is clear that $\mathcal{B}$ simulates either $\mathcal{H}2$ or $\mathcal{H}3$, depending on whether it was given an encoding of k or of 0, which results in our claim.

Hybrid $\mathcal{H}_4$: In this hybrid, the challenger replies to the challenge query by returning a uniformly random value from $\mathbb{Z}_n$. Since the adversary's view does no longer contain any information about the PRF key k, the value of $F_k(\mathbf{x})$ is computationally indistinguishable from a random element of $\mathcal{Y}$ due to the security of the PRF. We also required $\mathcal{Y}$ to be such that F is pseudorandom on $\mathbb{Z}_n$. Therefore, hybrids $\mathcal{H}_3$ and $\mathcal{H}_4$ are computationally indistinguishable thanks to the security of the underlying PRF. Note that here that we only rely on Find-then-Guess security for the underlying PRF (See Definition 10).

The rest of the proof proceeds by reversing the sequence of hybrid games while leaving the challenge query answered by a uniformly random value.

Constraint-Hiding. We finally prove that our construction is also constraint-hiding. The proof essentially follows the same line as the proof of pseudorandomness except that one deviates at $\mathcal{H}_4$. Notice that in $\mathcal{H}_3$ already, the only place where $\mathbf{z}$ plays a role in the adversary's view is in the evaluations, since the constrained key is sampled using Sim_0 after $\mathcal{H}_2$.

Now, the hybrid game $\mathcal{H}_4$ for the constraint-hiding proof does the following: rather than using Find-then-Guess security and changing only the evaluation of the challenge (which no longer exists in the constraint-hiding security game), we use standard PRF security to replace answers to evaluation queries of the form $y_1 + \langle \mathbf{x}, \mathbf{z} \rangle \cdot F_k(\mathbf{x})$ by values $y_1 + \langle \mathbf{x}, \mathbf{z} \rangle \cdot f(\mathbf{x})$ where f is a truly random function (sampled lazily). This changes evaluation at points $\mathbf{x}$ such that $\langle \mathbf{x}, \mathbf{z} \rangle \neq 0$ to uniformly random (and independent) values, in particular these values are independent of the constraint $\mathbf{z}$. One can then switch the constraint $\mathbf{z}$ to $\mathbf{z}'$ easily, since the pair of constraints is required to satisfy $\langle \mathbf{x}, \mathbf{z} \rangle \neq 0$ if and only if $\langle \mathbf{x}, \mathbf{z}' \rangle \neq 0$ for all evaluation queries x.

This concludes the proof of Theorem 3. □

D.2 CPRF for NC^1 from HSS (Proof of Theorem 4)

We now prove correctness and pseudorandomness of the construction.

[13] By 0 we mean any fixed key, e.g. 0^λ if $\mathcal{K} = \{0,1\}^\lambda$.

Correctness. The proof of correctness is roughly the same as for Construction 1 and directly follows from correctness properties of the underlying staged HSS scheme. The output of $\mathsf{Eval}(\mathsf{pp}, \mathsf{msk}, x)$ and $\mathsf{CEval}(\mathsf{pp}, \mathsf{ck}_C, x)$ on an input x form substractive shares of $U(C, x) \cdot F_k(x) = C(x) \cdot F_k(x)$. When $C(x) = 0$, correctness of the staged HSS scheme guarantees that the outputs of evaluation and constrained evaluation algorithms form substractive shares of 0, thus they are equal.

Pseudorandomness. Here again, the proof follows a similar strategy as in the case of inner-product constraints. The goal is to remove the dependency to k the underlying PRF key in the constrained key such that the term $C(x^*) \cdot F_k(x^*)$ makes the challenge pseudorandom (since x^* is required to satisfy $C(x^*) = 1$). We proceed via a sequence of hybrid games.

Hybrid $\mathcal{H}_0$: This is the standard CPRF security game where the challenge is answered with the real PRF evaluation.

Hybrid $\mathcal{H}_1$: In this first hybrid game, we only change the definition of the evaluation oracle. Since we are in the selective setting, the challenger knows the constraint C before answering any evaluation query. Therefore, it can compute the constrained key ck_C from the start. When asked for an evaluation query on input x, the challenger now replies to it by computing $y_1 \leftarrow \mathsf{CEval}(\mathsf{pp}, \mathsf{ck}_C, x)$ and returning $y_1 + C(x) \cdot F_k(x)$. The adversary's view remains identical to its view in $\mathcal{H}_0$ by the correctness of HSS, therefore these two hybrid games are perfectly indistinguishable.

Hybrid $\mathcal{H}_2$: In this second hybrid game, instead of sampling the constrained key elements $\bar{\mathsf{I}}_1^{(i)}$ as $\bar{\mathsf{I}}_1^{(i)} \leftarrow \overline{\mathsf{Input}}_1(\mathsf{pk}, C_i, \mathsf{aux}^{(i)}, (\mathsf{ek}_0, \mathsf{ek}_1))$ for $i \in [z]$, we replace each of these values by $\mathsf{Input}(\mathsf{pk}, C_i)$. Computational indistinguishability between these two hybrid games follows from the staged-security of HSS.

Hybrid $\mathcal{H}_3$: We now remove the information about k in the constrained evaluation key as follows: instead of defining $(\mathsf{I}_0, \mathsf{I}_1) \leftarrow \mathsf{Input}(\mathsf{pk}, k)$, we set $(\mathsf{I}_0, \mathsf{I}_1) \leftarrow \mathsf{Input}(\mathsf{pk}, 0)$[14]. The PRF key $k \xleftarrow{\$} \mathcal{K}$ is still sampled by the challenger, and evaluation (and challenge) queries are still answered on input by having the challenger computing $y_1 \leftarrow \mathsf{CEval}(\mathsf{pp}, \mathsf{ck}_C, x)$ and returning $y_1 + C(x) \cdot F_k(x)$.

By HSS security (for $\sigma = 1$) and correctness of the HSS evaluation, we claim that $\mathcal{H}_2$ and $\mathcal{H}_3$ are computationally indistinguishable. Suppose that an adversary $\mathcal{A}$ could distinguish between these two hybrids, we construct an adversary $\mathcal{B}$ against HSS security as follows: $\mathcal{B}$ first samples a key $k \xleftarrow{\$} \mathcal{K}$ and submits message $(k, 0)$ to its HSS challenger. It gets back $(\mathsf{pk}, \mathsf{ek}_1, \mathsf{I}_1)$ where I_1 is the right-hand part of $\mathsf{Input}(k)$ or $\mathsf{Input}(0)$. Then, $\mathcal{B}$ computes the constrained key ck_C as described in $\mathcal{H}_2$. It answers $\mathcal{A}$'s evaluation (and challenge) queries x by computing $y_1 \leftarrow \mathsf{CEval}(\mathsf{pp}, \mathsf{ck}_C, x)$ and returning $y_1 + C(x) \cdot F_k(x)$. When $\mathcal{A}$ halts with some output b', so does $\mathcal{B}$. It is clear that $\mathcal{B}$ simulates either $\mathcal{H}_2$ or $\mathcal{H}_3$, depending on whether it was given an encoding of k or of 0, which results in our claim.

Hybrid $\mathcal{H}_4$: In this hybrid, the challenger now replies to the challenge query x^* by returning a uniformly random value of $\mathcal{Y}$. Since the adversary's view does no longer contain any information about the PRF key k, then $F_k(x^*)$ can be replaced by a random value of $\mathcal{Y}$. Also, since x^* must satisfy $C(x^*) = 1$, then $y_1 + C(x) \cdot F_k(x) = y_1 + F_k(x)$, which is computationally indistinguishable from a random element of $\mathcal{Y}$. Therefore, hybrids $\mathcal{H}_3$ and $\mathcal{H}_4$ are computationally indistinguishable thanks to the security of the underlying PRF. Note that here that we only rely on Find-then-Guess security for the underlying PRF.

The rest of the proof proceeds by reversing the sequence of hybrid games while leaving the challenge query answered by a uniformly random value. ◻

E Proofs of MPC Applications of Staged-HSS

E.1 Secure Computation with Precomputable Silent Preprocessing – Proofs

Proof of Lemma 2 (Additive HSS shares can be made pseudorandom).

Proof. Let $F \colon \mathcal{K} \times \mathcal{S} \to \mathcal{Y}$ be a pseudorandom function with an input space $\mathcal{S}$. Given a HSS scheme $\mathsf{HSS} = (\mathsf{Setup}, \mathsf{Input}, \mathsf{Eval})$ with additive reconstruction, we can define $\mathsf{HSS}' = (\mathsf{Setup}', \mathsf{Input}', \mathsf{Eval}')$ as follows:

[14] By 0 we mean any fixed key, e.g. 0^λ if $\mathcal{K} = \{0, 1\}^\lambda$.

– $\mathsf{HSS}'.\mathsf{Setup}(1^\lambda)$:
  1. $(\widetilde{\mathsf{ek}_0}, \widetilde{\mathsf{ek}_1}) \leftarrow \mathsf{HSS}.\mathsf{Setup}(1^\lambda)$.
  2. $k_{\mathsf{prf}} \xleftarrow{\$} \mathcal{K}$.
  3. For $\sigma \in \{0,1\}$,
     set $\mathsf{ek}_\sigma := (\widetilde{\mathsf{ek}_\sigma}, k_{\mathsf{prf}})$.
  4. Output $(\mathsf{ek}_0, \mathsf{ek}_1)$.

– $\mathsf{HSS}'.\mathsf{Input}(\mathsf{pk}, x) := \mathsf{HSS}.\mathsf{Input}(\mathsf{pk}, x)$.

– $\mathsf{HSS}'.\mathsf{Eval}(\sigma, \mathsf{ek}_\sigma, \mathsf{I}_\sigma, P, \mathsf{id})$:
  1. Parse ek_σ as $(\widetilde{\mathsf{ek}_\sigma}, k_{\mathsf{prf}})$.
  2. $y_\sigma \leftarrow \mathsf{HSS}.\mathsf{Eval}(\sigma, \widetilde{\mathsf{ek}_\sigma}, \mathsf{I}_\sigma, P)$
  3. Output $y_\sigma + F_{k_{\mathsf{prf}}}(\mathsf{id})$.

In other words, during the Setup, we additionally sample a PRF master key which is added to the evaluation keys of both parties. The evaluation algorithm Eval then uses these PRF keys to mask the output shares. We also assume that each evaluation instruction is assigned a unique identifier id. The PRF's input space $\mathcal{S}$ should contain the space of these identifiers. The modified scheme, HSS', inherits correctness (via additive reconstruction) from HSS as both parties use the same mask $F_{k_{\mathsf{prf}}}(\mathsf{id})$ for each instruction.

□

Proof of Theorem 5 (Precomputable PCF for OLE Correlations from HSS).

Proof. We show that (PCF.Gen, PCF.Eval) in Construction 3 is a precomputable and programmable PCF.

Pseudorandom OLE-Correlated Outputs. We prove that the joint distribution of outputs of the PCF.Eval algorithm is indistinguishable from the outputs of an OLE correlation (as required in Figure 6). Let $\mathcal{A}$ be an adversary in the experiment $\mathsf{Exp}^{\mathsf{pr}}_{\mathcal{A},N,\sigma,1}$, and consider the following sequence of hybrid games:

Hybrid $\mathcal{H}_0$: This is the Experiment $\mathsf{Exp}^{\mathsf{pr}}_{\mathcal{A},N,\sigma,1}$, where $\mathcal{A}$ receives outputs of the PCF.Eval algorithm. As a reminder, here, the view of $\mathcal{A}$ view consists of

$$\left(x^{(i)}, \left(F_{k^{(0)}_{\mathsf{prf}}}(x^{(i)}), y_0^{(i)}\right), \left(F_{k^{(1)}_{\mathsf{prf}}}(x^{(i)}), y_1^{(i)}\right)\right)_{i\in[N(\lambda)]}.$$

Hybrid $\mathcal{H}_1$: In this game, for each $x^{(i)} \in \{0,1\}^{n(\lambda)}$, we first compute $y_0^{(i)} \leftarrow \mathsf{PCF}.\mathsf{Eval}(0, k_0, x^{(i)})$, and compute $y_1^{(i)}$ as $y_1^{(i)} = y_0^{(i)} - F_{k^{(0)}_{\mathsf{prf}}}(x^{(i)}) \cdot F_{k^{(1)}_{\mathsf{prf}}}(x^{(i)})$. The adversary's view remains identical to its view in $\mathcal{H}_0$ by the correctness of the HSS scheme, therefore these two hybrid games are perfectly indistinguishable.

Hybrid $\mathcal{H}_2$: In this game, for each $x^{(i)} \in \{0,1\}^{n(\lambda)}$, we first sample a random value $y_0^{(i)} \xleftarrow{\$} \mathcal{Y}$, and then compute $y_1^{(i)}$ in the same way as in Hybrid $\mathcal{H}_1$. Assuming the HSS outputs are individually pseudorandom, $\mathcal{H}_1$ and $\mathcal{H}_2$ are computationally indistinguishable.

Hybrid $\mathcal{H}_3$: Finally, since the view of $\mathcal{A}$ is independent of the PRF keys $k^{(0)}_{\mathsf{prf}}$, and $k^{(1)}_{\mathsf{prf}}$, we can replace the value of these PRFs by random values from $\mathcal{Y}$. More specifically, in this game, for each $x^{(i)} \in \{0,1\}^{n(\lambda)}$, we first sample three random values $y_0^{(i)}, a^{(i)}, b^{(i)} \xleftarrow{\$} \mathcal{Y}$, and then compute $y_1^{(i)}$ as $y_1^{(i)} = y_0^{(i)} - a^{(i)} \cdot b^{(i)}$, and output $((a^{(i)}, y_0^{(i)}), (b^{(i)}, y_1^{(i)}))$. Hybrids $\mathcal{H}_2$ and $\mathcal{H}_3$ are computationally indistinguishable thanks to the security of the underlying PRF.

Note that Hybrid $\mathcal{H}_3$ is the same as Experiment $\mathsf{Exp}^{\mathsf{pr}}_{\mathcal{A},N,\sigma,0}$. Thus we proved that the two experiments are computationally indistinguishable.

Security. Let $\mathcal{A}$ be an adversary in the experiment $\mathsf{Exp}^{\mathsf{sec}}_{\mathcal{A},N,\sigma,0}$, and consider the following sequence of hybrid games:

Hybrid $\mathcal{H}_0$: This is the Experiment $\mathsf{Exp}^{\mathsf{sec}}_{\mathcal{A},N,\sigma,0}$, where $\mathcal{A}$ receives outputs of the PCF.Eval algorithm. As a reminder, here, the view of $\mathcal{A}$ view consists of

$$\left(1^\lambda, \sigma, k_\sigma, (x^{(i)}, y_{1-\sigma}^{(i)})_{i\in[N(\lambda)]}\right).$$

Hybrid $\mathcal{H}_1$: In this game, for each $x^{(i)} \in \{0,1\}^{n(\lambda)}$, we first compute $y_\sigma^{(i)} \leftarrow \mathsf{PCF.Eval}(\sigma, k_\sigma, x^{(i)})$, and compute $y_{1-\sigma}^{(i)}$ as

$$y_{1-\sigma}^{(i)} = \left(F_{k_{\mathsf{prf}}^{(1-\sigma)}}(x^{(i)}),\ y_\sigma^{(i)} + (-1)^{1-\sigma} \cdot F_{k_{\mathsf{prf}}^{(0)}}(x^{(i)}) \cdot F_{k_{\mathsf{prf}}^{(1)}}(x^{(i)})\right),$$

where the PRF keys $k_{\mathsf{prf}}^{(0)}$ and $k_{\mathsf{prf}}^{(1)}$ are parts of the generated keys k_0 and k_1. Note that the adversary's view remains identical to its view in $\mathcal{H}_0$ by the correctness of the HSS scheme, therefore $\mathcal{H}_0$ and $\mathcal{H}_1$ are perfectly indistinguishable.

Next, we move to the following hybrid when $\sigma = 0$.
Hybrid $\mathcal{H}_2^{(\sigma=0)}$: In this game, instead of running $(k_0, k_1) \leftarrow \mathsf{PCF.Gen}(1^\lambda)$, we only generate k_0 by running $k_0 \leftarrow \mathsf{PCF.Gen}_0(1^\lambda)$. Subsequently, for an input $x^{(i)}$, we compute $y_1^{(i)}$ as $y_1^{(i)} = (a^{(i)}, y_0^{(i)} - F_{k_{\mathsf{prf}}^{(0)}} \cdot a^{(i)})$, where $a^{(i)} \xleftarrow{\$} \mathcal{Y}$.
Hybrids $\mathcal{H}_2^{(\sigma=0)}$ and $\mathcal{H}_1$ are computationally indistinguishable thanks to the security of the underlying PRF. Observe that the view of the adversary in this hybrid is identical to Experiment $\mathsf{Exp}^{\mathsf{sec}}_{\mathcal{A},N,\sigma=0,1}$. Therefore, we showed that the two security experiments are computationally indistinguishable when $\sigma = 0$.

When $\sigma = 1$, we consider the following hybrid as the game after $\mathcal{H}_1$.
Hybrid $\mathcal{H}_2^{(\sigma=1)}$: In this game, we remove the information of $k_{\mathsf{prf}}^{(0)}$ from k_1. More specifically, we first parse $(k_0, k_1) \leftarrow \mathsf{PCF.Gen}$ as $(k_0, \mathsf{aux}) \leftarrow \mathsf{PCF.Gen}_0(1^\lambda)$, and $k_1 \leftarrow \mathsf{PCF.Gen}_1(1^\lambda, k_0, \mathsf{aux})$. This change does not affect the view of the adversary.

Next, when running $(k_0, \mathsf{aux}) \leftarrow \mathsf{PCF.Gen}_0(1^\lambda)$, instead of computing $(\mathsf{I}_0, \mathsf{I}_1) \leftarrow \mathsf{Input}(\mathsf{pk}, k_{\mathsf{prf}}^{(0)})$, we set $(\mathsf{I}_0, \mathsf{I}_1) \leftarrow \mathsf{Input}(\mathsf{pk}, 0)$[15]. As a result, the key k_1 does not contain any information about $k_{\mathsf{prf}}^{(0)}$. The PRF key $k_{\mathsf{prf}}^{(0)} \xleftarrow{\$} \mathcal{K}$ is still sampled, and y_0 is computed and returned in the same way as in Hybrid $\mathcal{H}_1$. By the security of HSS (for $\sigma = 1$), $\mathcal{H}_2^{(\sigma=1)}$ and $\mathcal{H}_1$ are computationally indistinguishable.

Hybrid $\mathcal{H}_3^{(\sigma=1)}$: In this game, for an input $x^{(i)}$, similarly to $\mathcal{H}_2^{(\sigma=1)}$, we first run $y_1^{(i)} \leftarrow \mathsf{PCF.Eval}(1, k_1, x^{(i)})$, but afterwards, we compute and output $y_0^{(i)}$ as $y_0^{(i)} = (a^{(i)}, y_1^{(i)} + F_{k_{\mathsf{prf}}^{(0)}} \cdot a^{(i)})$, where $a^{(i)} \xleftarrow{\$} \mathcal{Y}$.
Hybrids $\mathcal{H}_3^{(\sigma=1)}$ and $\mathcal{H}_3^{(\sigma=1)}$ are computationally indistinguishable thanks to the security of the underlying PRF. Observe that the view of the adversary in this hybrid is identical to Experiment $\mathsf{Exp}^{\mathsf{sec}}_{\mathcal{A},N,\sigma=1,1}$. Therefore, we showed that the two security experiments are computationally indistinguishable when $\sigma = 1$.

Programmability. We introduce an algorithm $\mathsf{Gen_p}$ that satisfies properties required in Definition 15, namely, indistinguishability, programmability, and security. Consider the algorithm $\mathsf{Gen_p}$ which, on input $(1^\lambda; \rho_0, \rho_1)$, runs exactly as Gen, but with the following difference:

- It uses its additional inputs (ρ_0, ρ_1) in the following manner: Instead of sampling "$k_{\mathsf{prf}}^{(0)} \xleftarrow{\$} \mathcal{K}$", it sets "$k_{\mathsf{prf}}^{(0)} \leftarrow \rho_0$", and instead of sampling "$k_{\mathsf{prf}}^{(1)} \xleftarrow{\$} \mathcal{K}$" it sets "$k_{\mathsf{prf}}^{(1)} \leftarrow \rho_1$".

It immediately follows that

$$\{(k_0, k_1)\colon (k_0, k_1) \leftarrow \mathsf{PCF.Gen}(1^\lambda)\} \approx_c \{(k_0, k_1)\colon (\rho_0, \rho_1) \leftarrow \$, (k_0, k_1) \leftarrow \mathsf{PCF.Gen_p}(1^\lambda; \rho_0, \rho_1)\}.$$

Thus, the indistinguishability holds.

Programmability follows from inspection if we define $f_0 := F$ and $f_1 := F$ (where F is the PRF used in Construction 3).

[15] By 0 we mean any fixed key, e.g. 0^λ if $\mathcal{K} = \{0,1\}^\lambda$.

Now we prove the security. Recall that we need to show for any $\sigma \in \{0,1\}$, the following two distributions are computationally indistinguishable:

$$\{(k_\sigma, (\rho_0, \rho_1))\colon (\rho_0, \rho_1) \leftarrow \$, (k_0, k_1) \leftarrow \mathsf{PCF.Gen_p}(1^\lambda; \rho_0, \rho_1)\} \approx_c \{(k_\sigma, \{\rho_\sigma, \tilde{\rho}\})\colon (\rho_0, \rho_1, \tilde{\rho}) \leftarrow \$, (k_0, k_1) \leftarrow \mathsf{PCF.Gen_p}(1^\lambda; \rho_0, \rho_1)\},$$

We differentiate between the following two cases:

- $\sigma = 0$: In this case, k_0 contains no information about $k_{\mathsf{prf}}^{(1)}$ when they are output by $\mathsf{PCF.Gen}$. Thus, k_0 contains no information about ρ_1 in $\mathsf{Gen_p}$, so it can be replaced by a random value in the view of an adversary. This implies the security for this case.

- $\sigma = 1$: In this case, we show that the two distributions are computationally indistinguishable by the security of the underlying HSS scheme. More specifically, recall that $k_1 = (\mathsf{ek}_1, \mathsf{I}_1, \bar{\mathsf{I}}_1, \rho_1)$, where $(\mathsf{I}_0, \mathsf{I}_1) \leftarrow \mathsf{HSS.Input}(\mathsf{HSS.pk}, \rho_0)$, and all the other elements of the tuple are independent of ρ_0. By the security of HSS, we can alternatively compute $(\mathsf{I}_0, \mathsf{I}_1) \leftarrow \mathsf{HSS.Input}(\mathsf{HSS.pk}, 0)$ and therefore, remove the information of ρ_0 from k_1, so it can be replaced by a random value in the view of an adversary. This implies the security for this case.

This concludes the proof of Theorem 5.

□

E.2 Sublinear Secure Computation with One-Sided Statistical Security – Proofs

Proof of Lemma 3.

Proof. First observe that if the PRF $F(\cdot,\cdot)$ and C have logarithmic depth then $f_{\alpha,c_{\mathsf{in}}}$ can indeed be expressed as an RMS program, and we cause the PRF from [36]. The required amount of communication follows from inspection of the protocol, and we are left to prove security. Let $\mathcal{A}$ be a semi-honest, static adversary that interacts with parties Alice and Bob running protocol Π_C in the $\mathcal{F}_{\mathsf{update}}^{\mathsf{HSS}}$-hybrid model. We need to construct a simulator Sim such that no environment $\mathcal{Z}$ can distinguish with non-negligible probability whether it is interacting with $\mathcal{A}$ and parties running Π_C in the $\mathcal{F}_{\mathsf{update}}^{\mathsf{HSS}}$-hybrid model, or with Sim and $\mathcal{F}_{\mathsf{SFE}}(C)$. Because the adversary is static, we can assume that the set of corrupted parties is fixed before the start of the protocol, and we can consider the cases of a corrupted Alice and a corrupted Bob separately.

– **Perfect security against a corrupted Alice.** Alice receives no messages (from Bob or $\mathcal{F}_{\mathsf{update}}^{\mathsf{HSS}}$) in the real execution, therefore a simulator can trivially perfectly simulate the joint view of $\mathcal{Z}$ and $\mathcal{A}$ in the execution of Π_C.
– **Computational security against a corrupted Bob.** Consider the simulator Sim which is given as input the corrupted Bob's input $x_1 \in \mathbb{F}^{n_1}$, runs an internal copy of $\mathcal{A}$, and acts as follows:
  - **Simulating the communication with $\mathcal{Z}$:** Every input value that Sim receives from $\mathcal{Z}$ is written on $\mathcal{A}$'s input tape (as if coming from $\mathcal{A}$'s environment), and every output value written by $\mathcal{A}$ on its output tape is copied to Sim's own output tape (to be read by $\mathcal{Z}$).
  - **Simulating the protocol's execution:**
    1. Sim sends $(\mathtt{input}, 1, x_1)$ to $\mathcal{F}_{\mathsf{SFE}}(C)$ and waits to receive y_1'.
    2. Sim samples $K \xleftarrow{\$} \{0,1\}^\lambda$, $(\mathsf{pk}, \mathsf{ek}_0, \mathsf{ek}_1) \leftarrow \mathsf{HSS.Setup}(1^\lambda)$, $(\bar{\mathsf{I}}_0, \mathsf{aux}) \leftarrow \mathsf{HSS.}\overline{\mathsf{Input}}_0(\mathsf{pk})$, $(\mathsf{I}_0, \mathsf{I}_1) \leftarrow \mathsf{HSS.Input}(1^\lambda, K)$, $\alpha \xleftarrow{\$} \{0,1\}^\lambda$.
    3. Sim sets $c_{\mathsf{in}} \leftarrow F(K, \alpha)$.
    4. Sim computes $\bar{\mathsf{I}}_1 \leftarrow \mathsf{staged\text{-}HSS.Share}(\mathsf{pk}, x_1, \mathsf{aux})$.
    5. Sim computes $M_1 \leftarrow \mathsf{HSS.}\overline{\mathsf{Eval}}(\mathsf{ek}_1, \bar{\mathsf{I}}_1, F(\cdot, \alpha))$.
    6. Sim sets $r_{\mathsf{out}} \leftarrow \mathsf{HSS.Eval}'(\mathsf{ek}_1, (M_1, I_1), f_{\alpha,c_{\mathsf{in}}}) - y_1'$.
    7. Sim writes $(\mathsf{ek}_1, \bar{\mathsf{I}}_1, c_{\mathsf{in}}, \alpha, r_{\mathsf{out}})$ on $\mathcal{A}$'s input tape (as if Alice sent it to Bob).
    8. Sim proceeds with the execution of $\mathcal{A}$ until the latter writes $(\mathtt{input}, x_1)$ on its output tape (the message from Bob to $\mathcal{F}_{\mathsf{update}}^{\mathsf{HSS}}$), at which point it writes $(\mathsf{HSS.pk}, \bar{\mathsf{I}}_1)$ on $\mathcal{A}$'s input tape (as if sent to Bob by $\mathcal{F}_{\mathsf{update}}^{\mathsf{HSS}}$).

Observe that the difference in the joint view of $\mathcal{Z}$ and $\mathcal{A}$ in the real and ideal worlds boils down to how c_{in} and r_{out} are defined. However, because in the ideal world y_1' is uniformly distributed and independent from the coins of Sim, r_{out} is uniformly distributed in the ideal world. Further, because there is no entropy in the single outgoing message of Bob (the message $(\mathtt{input}, 1, x_1)$ he sends to $\mathcal{F}_{\mathsf{OLE}}$), the internal coins of Bob are irrelevant in the joint view of $\mathcal{Z}$ and $\mathcal{A}$. Let $(x_0, x_1) \in \mathbb{F}^{n_0} \times \mathbb{F}^{n_1}$. From what precedes, it suffices to show that the distributions of outputs of the following probability experiments are computationally indistinguishable:

$\underline{\mathsf{RealView}(1^\lambda, x_0, x_1)}$:	$\underline{\mathsf{IdealView}(1^\lambda, x_1)}$:
$K \xleftarrow{\$} \{0,1\}^\lambda$	$K \xleftarrow{\$} \{0,1\}^\lambda$
$\alpha \xleftarrow{\$} \{0,1\}^\lambda$	$\alpha \xleftarrow{\$} \{0,1\}^\lambda$
$c_{\mathsf{in}} \leftarrow x_0 + F(K, \alpha)$	$c_{\mathsf{in}} \leftarrow F(K, \alpha)$
$(\mathsf{pk}, \mathsf{ek}_0, \mathsf{ek}_1) \leftarrow \mathsf{HSS.Setup}(1^\lambda)$	$(\mathsf{pk}, \mathsf{ek}_0, \mathsf{ek}_1) \leftarrow \mathsf{HSS.Setup}(1^\lambda)$
$(\bar{\mathsf{I}}_0, \mathsf{aux}) \leftarrow \mathsf{HSS}.\overline{\mathsf{Input}}_0(\mathsf{pk})$	$(\bar{\mathsf{I}}_0, \mathsf{aux}) \leftarrow \mathsf{HSS}.\overline{\mathsf{Input}}_0(\mathsf{pk})$
$(\mathsf{I}_0, \mathsf{I}_1) \leftarrow \mathsf{HSS.Input}(1^\lambda, K)$	$(\mathsf{I}_0, \mathsf{I}_1) \leftarrow \mathsf{HSS.Input}(1^\lambda, K)$
$\bar{\mathsf{I}}_1 \leftarrow \mathsf{staged\text{-}HSS.Share}(\mathsf{pk}, x_1, \mathsf{aux})$	$\bar{\mathsf{I}}_1 \leftarrow \mathsf{staged\text{-}HSS.Share}(\mathsf{pk}, x_1, \mathsf{aux})$
$r_{\mathsf{out}} \xleftarrow{\$} \mathbb{F}^m$	$r_{\mathsf{out}} \xleftarrow{\$} \mathbb{F}^m$
Output $(\mathsf{ek}_1, \mathsf{I}_1, c_{\mathsf{in}}, \alpha, r_{\mathsf{out}}, x_1, \mathsf{pk}, \bar{\mathsf{I}}_1)$	Output $(\mathsf{ek}_1, \mathsf{I}_1, c_{\mathsf{in}}, \alpha, r_{\mathsf{out}}, x_1, \mathsf{pk}, \bar{\mathsf{I}}_1)$

To that end, consider the following experiments (which differ with the above only in the law of $(\mathsf{I}_0, \mathsf{I}_1)$):

$\underline{\mathsf{HybridReal}(1^\lambda, x_0, x_1)}$:	$\underline{\mathsf{HybridIdeal}(1^\lambda, x_1)}$:
$K \xleftarrow{\$} \{0,1\}^\lambda$	$K \xleftarrow{\$} \{0,1\}^\lambda$
$\alpha \xleftarrow{\$} \{0,1\}^\lambda$	$\alpha \xleftarrow{\$} \{0,1\}^\lambda$
$c_{\mathsf{in}} \leftarrow x_0 + F(K, \alpha)$	$c_{\mathsf{in}} \leftarrow F(K, \alpha)$
$(\mathsf{pk}, \mathsf{ek}_0, \mathsf{ek}_1) \leftarrow \mathsf{HSS.Setup}(1^\lambda)$	$(\mathsf{pk}, \mathsf{ek}_0, \mathsf{ek}_1) \leftarrow \mathsf{HSS.Setup}(1^\lambda)$
$(\bar{\mathsf{I}}_0, \mathsf{aux}) \leftarrow \mathsf{HSS}.\overline{\mathsf{Input}}_0(\mathsf{pk})$	$(\bar{\mathsf{I}}_0, \mathsf{aux}) \leftarrow \mathsf{HSS}.\overline{\mathsf{Input}}_0(\mathsf{pk})$
$(\mathsf{I}_0, \mathsf{I}_1) \leftarrow \mathsf{HSS.Input}(1^\lambda, 0^\lambda)$	$(\mathsf{I}_0, \mathsf{I}_1) \leftarrow \mathsf{HSS.Input}(1^\lambda, 0^\lambda)$
$\bar{\mathsf{I}}_1 \leftarrow \mathsf{staged\text{-}HSS.Share}(\mathsf{pk}, x_1, \mathsf{aux})$	$\bar{\mathsf{I}}_1 \leftarrow \mathsf{staged\text{-}HSS.Share}(\mathsf{pk}, x_1, \mathsf{aux})$
$r_{\mathsf{out}} \xleftarrow{\$} \mathbb{F}^m$	$r_{\mathsf{out}} \xleftarrow{\$} \mathbb{F}^m$
Output $(\mathsf{ek}_1, \mathsf{I}_1, c_{\mathsf{in}}, \alpha, r_{\mathsf{out}}, x_1, \mathsf{pk}, \bar{\mathsf{I}}_1)$	Output $(\mathsf{ek}_1, \mathsf{I}_1, c_{\mathsf{in}}, \alpha, r_{\mathsf{out}}, x_1, \mathsf{pk}, \bar{\mathsf{I}}_1)$

Let $\mathcal{A}^{\mathsf{Real}}_{x_0,x_1}$ be a *PPT* adversary which distinguishes between the real view $\{\mathsf{RealView}(1^\lambda, x_0, x_1)\}$ and $\{\mathsf{HybridReal}(1^\lambda, x_0, x_1)\}$ with probability ϵ_{Real}. By the security of HSS, for every *PPT* adversaries $\mathcal{A}, \mathcal{A}'$, and any bit $\sigma \in \{0,1\}$:

$$\left| \Pr\left[b' = b : \begin{array}{l} (K_0, K_1, \mathsf{state}) \leftarrow \mathcal{A}(1^\lambda) \\ (\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \leftarrow \mathsf{Setup}(1^\lambda) \\ b \xleftarrow{\$} \{0,1\} \\ (\mathsf{I}_0, \mathsf{I}_1) \leftarrow \mathsf{Input}(K_b) \\ b' \leftarrow \mathcal{A}'(\mathsf{state}, \mathsf{pk}, \mathsf{ek}_\sigma, \mathsf{I}_\sigma) \end{array} \right] - \frac{1}{2} \right| \leq \mathsf{negl}(\lambda) \ .$$

Considering the following *PPT* algorithms $\mathcal{A}_{x_0}$ and $\mathcal{A}'_{x_1}$ proves that ϵ_{Real} must be negligible:

- $\mathcal{A}_{x_0}(1^\lambda)$: $K \xleftarrow{\$} \{0,1\}^\lambda$; $\alpha \xleftarrow{\$} \{0,1\}^\lambda$; $\mathsf{state} \leftarrow (x_0 + F(K, \alpha), \alpha)$; Output $(K, 0^\lambda, \mathsf{state})$.
- $\mathcal{A}'_{x_1}(\mathsf{state}, \mathsf{pk}, \mathsf{ek}_1, \mathsf{I}_1)$: $(\bar{\mathsf{I}}_0, \mathsf{aux}) \leftarrow \mathsf{HSS}.\overline{\mathsf{Input}}_0(\mathsf{pk})$;
  $\bar{\mathsf{I}}_1 \leftarrow \mathsf{staged\text{-}HSS.Share}(\mathsf{pk}, x_1, \mathsf{aux})$; $r_{\mathsf{out}} \xleftarrow{\$} \{0,1\}^\lambda$;
  Output $\mathcal{A}^{\mathsf{Real}}_{x_0,x_1}(\mathsf{ek}_1, \mathsf{I}_1, \mathsf{state.first}, \mathsf{state.second}, r_{\mathsf{out}}, x_1, \mathsf{pk}, \bar{\mathsf{I}}_1)$.

Therefore $\{\mathsf{RealView}(1^\lambda, x_0, x_1)\} \overset{c}{\approx} \{\mathsf{HybridReal}(1^\lambda, x_0, x_1)\}$, and similarly $\{\mathsf{IdealView}(1^\lambda, x_1)\} \overset{c}{\approx} \{\mathsf{HybridIdeal}(1^\lambda, x_1)\}$. Finally, because all other random variables than c_{in} are independent of K (both in $\mathsf{HybridReal}(1^\lambda, x_0, x_1)$ and in $\mathsf{HybridIdeal}(1^\lambda, x_1)$), we can conclude using a straightforward reduction to the security of the PRF $F(\cdot, \cdot)$ that the two distributions $\{\mathsf{HybridReal}(1^\lambda, x_0, x_1)\}$ and $\{\mathsf{HybridIdeal}(1^\lambda, x_1)\}$ are computationally indistinguishable. Wrapping up, this implies that the joint view of $\mathcal{Z}$ and $\mathcal{A}$ is indistinguishable in the real and ideal worlds.

□

Proof of Lemma 4.

Proof. Set $\ell(\lambda) = \Theta(\lambda)$ (and therefore $s = \mathcal{O}(1)$) such that $\ell(\lambda) \geq \frac{3}{2}\lambda$. Sending N requires $2\ell(\lambda)$ bits of communication, h requires $4\ell(\lambda)$, $\mathsf{ct}_{\mathsf{ind}}$ and $\vec{c}$ require $\mathcal{O}(\ell(\lambda) \cdot n_1)$. We now analyse security. Let $\mathcal{A}$ be a semi-honest, static adversary that interacts with parties Alice and Bob running protocol Π_C in the $\mathcal{F}^{\mathsf{HSS}}_{\mathsf{update}}$-hybrid model. We need to construct a simulator Sim such that no environment $\mathcal{Z}$ can distinguish with non-negligible probability whether it is interacting with $\mathcal{A}$ and Π_C in the $\mathcal{F}^{\mathsf{HSS}}_{\mathsf{update}}$-hybrid model, or with Sim and $\mathcal{F}_{\mathsf{SFE}}(C)$. Because the adversary is static, we can assume that the set of corrupted parties is fixed before the start of the protocol, and we can consider the cases of a corrupted Alice and a corrupted Bob separately.

– **Perfect security against a corrupted Alice.** Alice receives no messages from Bob in the real execution and the single message Alice receives from $\mathcal{F}_{\mathsf{OLE}}$ is a uniformly random value in $(\mathbb{F}_{2^\lambda})^{t\cdot(s+1)}$, so it is easy to see that joint view of $\mathcal{Z}$ and $\mathcal{A}$ in the execution of $\Pi^{\mathsf{HSS}}_{\mathsf{update}}$ can be simulated perfectly.
– **Perfect security against a corrupted Bob.** Consider the simulator Sim which is given as input the corrupted Bob's input $x_1 \in (\mathbb{F}_{2^\lambda})^{n_1}$, internally runs a copy of $\mathcal{A}$, and acts as follows:
  - **Simulating the communication with $\mathcal{Z}$:** Every input value that Sim receives from $\mathcal{Z}$ is written on $\mathcal{A}$'s input tape (as if coming from $\mathcal{A}$'s environment), and every output value written by $\mathcal{A}$ on its output tape is copied to Sim's own output tape (to be read by $\mathcal{Z}$).
  - **Simulating the protocol's execution:**
    1. Sim sends $(\mathtt{input}, x_1)$ to $\mathcal{F}_{\mathsf{SFE}}(C)$ and waits to receive $(\mathsf{HSS.pk}, \bar{\mathsf{I}}_1 = (\mathsf{ct}_{\mathsf{ind}}, \mathsf{ct}_{\mathsf{dep}}))$.
    2. Parse $\mathsf{HSS.pk} = (\mathsf{pk}_{\mathsf{PaillierEG}}, D^{(0)}, \ldots, D^{(s-1)})$ and recover N from $\mathsf{pk}_{\mathsf{PaillierEG}}$.
    3. Parse $\mathsf{ct}_{\mathsf{dep}} = (\mathsf{ct}^{(i,j)}_{\mathsf{dep}})_{(i,j)\in[t]\times[s+1]}$
    4. Sim writes $(N, \mathsf{pk}_{\mathsf{PaillierEG}}, \mathsf{ct}_{\mathsf{ind}})$ on $\mathcal{A}$'s input tape (as if sent by Alice to Bob).
    5. Sample $\vec{y}^{(1)} = (y^{(1)}_{(i,j)})_{(i,j)\in[t]\times[s+1]} \xleftarrow{\$} (\mathbb{F}_{2^\lambda})^{[t]\times[s+1]}$
    6. For $(i,j) \in [t] \times [s+1]$, set $c_{(i,j)} \leftarrow \mathsf{ct}^{(i,j)}_{\mathsf{dep}} \cdot (1+N)^{-y^{(1)}_{i,j}}$.
    7. Sim proceeds with the execution of $\mathcal{A}$ until the latter writes $(\mathtt{input}, 1, x_1)$ on its output tape (the message from Bob to $\mathcal{F}_{\mathsf{OLE}}$), at which point it writes $\vec{y}^{(1)}$ on $\mathcal{A}$'s input tape (as if sent to Bob by $\mathcal{F}_{\mathsf{OLE}}$).
    8. Sim writes $\vec{c}$ on $\mathcal{A}$'s input tape (as if sent by Alice to Bob).

  If follows from inspection that this simulation is perfect.

□