Computationally Succinct Authentication from DCR
Attribute-Based Laconic Function Evaluation and More

Pierre Meyer[1], Claudio Orlandi[1], Lawrence Roy[1], and Peter Scholl[1]

Aarhus University, Aarhus, Denmark
{pierre.meyer,orlandi,peter.scholl}@cs.au.dk, ldr709@gmail.com

Abstract. We present the first construction of *attribute-based laconic function evaluation* (AB-LFE) from the decisional composite residuosity (DCR) assumption. This yields the first example of computationally succinct secure computation from a group-based assumption, avoiding reliance on noisy lattice assumptions such as LWE. Our construction builds on recent work in fully homomorphic MACs and homomorphic secret sharing by Ishai, Li and Lin (FOCS 2025) and Meyer, Orlandi, Roy, and Scholl (Crypto 2025), which we extend by constructing a fully homomorphic MAC for packed vector operations, where the evaluation time of one party is independent of the vector length. On the way, we also provide novel constructions of depth-preserving *bundled universal circuits*.

As applications, we obtain full-fledged *laconic function evaluation* (LFE) from the combination of DCR and standard LWE, avoiding the need for sub-exponential modulus-to-noise ratio LWE as in previous work. In addition, we obtain the first constrained pseudorandom function whose master evaluation key is succinct in the constraint.

These results highlight the unexplored power of group-based cryptography for succinct secure computation.

Table of Contents

1 Introduction **4**

2 Technical Overview **5**
2.1 Concurrent Work of [LLL25] 10
2.2 Related Work 10

3 Preliminaries **11**
3.1 Damgård-Jurik-ElGamal Cryptosystem 12
3.2 Distributed Discrete Logarithm 13
3.3 Laconic Function Evaluation 13
3.4 Constrained Pseudorandom Functions 14
3.5 Succinct, Bilinear HSS 15
3.6 Circuits 16
3.6.1 Circuit Classes. 17

4 A Bundled Universal Circuit **18**

5 Semi-private Bundled HSS (from DCR) **20**
5.1 Definition 20
5.2 Construction from DCR 22

6 Attributed-based laconic function evaluation from semi-private succinct HSS **26**

7 Computationally succinct constrained PRF from semi-private succinct HSS **28**

A Details on Circuit Definitions **34**

B Details on Universal Circuit Construction **35**
B.1 Proof of Theorem 13 35
B.2 Remaining Details 36
B.3 Proof of Theorem 15 39

C Details on HSS Definition **39**
C.1 Succinct HSS Definition 39
C.2 HSS Correctness Definition 40
C.3 HSS Privacy Definitions 41
C.4 Details on HSS Construction 42
C.5 Restricted Multiplication HSS Operations 42
C.6 Bundled HSS Operations 43

D Construction of Key Switching Graphs **44**

E Semi-Private HSS, Gate-by-Gate Correctness **45**
E.1 Correctness of component-wise multiplication 45
E.2 Correctness of bitflip-swaps 46
E.3 Correctness of key/level-switching 47

E.4 Proof of Theorem Theorem 18 48

F DCR Reductions **50**
F.1 DJE Encryption 50
F.2 Privacy of Key Switching Gadget Ciphertexts 51

G Details on LFE Definition **52**

H Proof of Theorem 19 **53**

I Proof of Theorem 21 **57**

1 Introduction

One of the central questions in *secure multiparty computation* (MPC) is whether security comes at an inherent cost. In other words, what is the overhead of turning an insecure computation into a secure one? This paper continues the long line of work on understanding the communication and computation overhead in the setting of secure two-party computation.

We consider the case where one party holds a (large) description of a function f, while the other holds a (short) input x. At the end of the protocol, Alice should learn only $f(x)$, and Bob should learn nothing. Thus, we will also refer to Alice as the *receiver* and to Bob as the *sender*.

Early solutions to this problem [Yao86, GMW87] required both the communication and the computation to scale with the size of the function f. A major breakthrough came with the introduction of *fully homomorphic encryption* (FHE) [Gen09, BV11, GHS12]. With FHE, Alice can encrypt her short input x and send it to Bob, who then homomorphically evaluates f on the ciphertext and returns an encryption of $f(x)$. This way, Alice's work is proportional only to the size of x and $f(x)$, rather than the entire description of f. Thus, FHE enables *receiver-succinct secure computation*.

Subsequently, the seminal work on *laconic function evaluation* (LFE) [QWW18] enabled the dual notion, namely *sender-succinct secure computation*. Here, Alice, holding the description of f, sends Bob a succinct digest of f. Bob, with input x, returns an encoding of x (dependent on Alice's digest of f) that enables Alice to recover $f(x)$ and nothing else. Crucially, Bob's work in this setting is sublinear in the size of f.

A common template for building LFE [QWW18, DHM+24] is to combine FHE with a weaker primitive called *attribute-based LFE* (AB-LFE). In AB-LFE, Bob's input is a pair (x, m), and Alice should learn $(x, f(x) \cdot m)$. In other words, Alice obtains the payload m only if the function f evaluates to 1 on the (public) attribute x.

Previous constructions of secure protocols for succinct computation of expressive classes of functions (such as FHE and LFE) rely on lattice-based assumptions such as *learning-with-errors* (LWE). This raises a natural question:

Can we achieve computational succinctness in secure computation using only group-based assumptions?

In this work, we answer this question in the affirmative. We show that a significant subclass of LFE, namely AB-LFE, can be instantiated under the decisional composite residuosity (DCR) assumption alone. To the best of our knowledge, this is the first construction of a computationally succinct secure computation protocol that avoids noisy lattice assumptions, showing the unexplored power of group-based assumptions.

Technically, our work builds on the recent and rapidly evolving line of research on *homomorphic secret-sharing* (HSS) [BGI16]. In a nutshell, an HSS scheme for a class of functions $\mathcal{F}$ allows two parties, holding shares of some value x, to non-interactively derive shares of the output $f(x)$ for any $f \in \mathcal{F}$. Recent progress, in particular techniques for computing *distributed discrete logarithms* (DDLog), has extended HSS to very expressive function classes, which in turn has enabled a wide range of applications such as trapdoor hash functions [DGI+19, RS21], pseudorandom correlation functions [OSY21], constrained PRFs [CMPR23, MORS24, ILL25a] and compact garbled circuits [ILL25b, BMO+25, MORS25].

Our work relies on a perspective of HSS as performing *authenticated* homomorphic computation. At first sight, the connection between homomorphic authentication and succinct computation may not be obvious. A useful way to view AB-LFE is as a primitive that allows Alice to learn m if and only if $f(x) = 1$. Since both f and x are known to her, Alice can compute $f(x)$ on her own; the point is to ensure that this computation is *authenticated*. If

Alice and Bob can derive an HSS-based authentication tag that certifies $f(x) = 1$ (relative to Alice's digest of f), then Bob can encrypt the payload m under this tag. Consequently, when $f(x) = 1$ Alice can recover m using the tag, whereas when $f(x) = 0$ the authentication prevents her from learning anything about m.

The main difficulty is that Bob must produce the correct authentication tag for the case $f(x) = 1$ (needed to encrypt m) without doing work proportional to the size of f. As we describe in Section 2, we overcome this obstacle by introducing new tools in the HSS toolbox.

Our Contributions. The contributions of this work are as follows:

1. We introduce a new tool for succinct packed fully homomorphic MACs based on DCR, extending the ideas of [ILL25a].
2. We develop a new framework for depth-preserving, "bundled" universal circuits, where the universal circuit has a packed structure that is compatible with our MACs.
3. Using these tools, we provide the first construction of a succinct secure two-party computation protocol from group-based assumptions, namely, a protocol for AB-LFE based on the DCR assumption. For a circuit C, the computational cost for Bob is $O(|C|^{2/3}\mathsf{depth}(C)^{1/3})$, which is sublinear in C for sufficiently wide circuits.
4. As an application, we achieve full-fledged LFE via the known FHE-based bootstrapping from AB-LFE to LFE. In contrast to the original construction of [QWW18], which built AB-LFE from sub-exponential modulus-to-noise ratio LWE (with bootstrapping requiring only standard LWE), our approach replaces their AB-LFE with one from DCR. This yields LFE from the combination of DCR and standard LWE, thus achieving LFE from a qualitatively different set of assumptions than previously known.
5. We further show how our techniques yield the first constrained pseudorandom function with a master evaluation key that is succinct in the constraint function, by instantiating the HSS-to-CPRF template of [CMPR23] with our succinct HSS construction.

2 Technical Overview

Our technical contribution is two-fold:

1. we present a mechanism to achieve computationally succinct authentication from DCR,
2. we show how to combine it with existing tools in order to get attribute-based laconic function evaluation (and computationally succinct constrained PRFs) from DCR.

The first step could be formalised either as a form of succinct Homomorphic MAC [ILL25a] or (essentially equivalently) as a form of offline-online semi-private Homomorphic Secret Sharing [MORS25]. However, our applications require this new object to integrate with existing notions of HSS, so the body presents our construction in the definitional framework of HSS. For ease of reading however, we start by describing our contribution as a form of HMAC (whose formalism should be more accessible to most readers), and then briefly explain how this fits to an HSS framework.

Prior Homomorphic MACs. A standard way to build a statistically secure, *linearly* homomorphic MAC is the construction

$$M = \mathsf{k}x + r$$

where M is the MAC on message x under key k, and randomness r. This construction is most often used over a finite field, but can also work over (bounded) integers, if k and r are carefully sampled from suitably sized ranges.

In secure computation scenarios, typically, one party will hold (x, M) while the other party holds (k, r). Letting $X_0 := r$ and $X_1 := \mathsf{k}x + r$, this can now be seen as a subtractive sharing $X_1 - X_0 = \mathsf{k}x$.

Recently, Ishai, Li and Lin [ILL25a] showed how to make this construction *fully homomorphic*, using computational assumptions. Concretely, they rely on groups that admit a special form of *distributed discrete logarithm* computation, which were first introduced to build homomorphic secret sharing schemes [BGI16, OSY21, RS21, ADOS22].

Taking the group from the Damgård-Jurik encryption scheme [DJ01] as an example, their construction works as follows. Let N be an RSA modulus, $g \in \mathbb{Z}_{N^\zeta}$ a random, public base element and $f = (1 + N)$ a generator of the easy discrete log subgroup (of order N) in $\mathbb{Z}^*_{N^\zeta}$. Then the public key consists of the elements:

$$(g, g^{\mathsf{k}}, g^{\mathsf{k}^2} \cdot f^{\mathsf{k}}) \tag{1}$$

Now, suppose the parties have authentications of two values x and y, and the respective randomness of $\mathcal{P}_B$ (who also holds k) is denoted $X_0 = X_1 - x\mathsf{k}$ and $Y_0 = Y_1 - y\mathsf{k}$. Consider the identity:

$$X_0 Y_0 = X_1 Y_1 + (-Y_1 x - X_1 y)\mathsf{k} + xy\mathsf{k}^2$$

Notice that using (1), the parties can compute shares of this *in the exponent*: $\mathcal{P}_B$ computes $z'_0 = g^{X_0 Y_0}$, and $\mathcal{P}_A$ computes

$$z'_1 = g^{X_1 Y_1} (g^{\mathsf{k}})^{(-xY_1 - yX_1)} (g^{\mathsf{k}^2} f^{\mathsf{k}})^{xy}$$

Now, it holds that $z'_1 / z'_0 = f^{xy\mathsf{k}}$. To convert these multiplicative shares into linear shares, the parties locally run a procedure DDLog that distributively computes the discrete logarithm (in the subgroup generated by f), giving shares Z_0, Z_1 such that $Z_1 = Z_0 + xy\mathsf{k}$. This multiplication procedure can be iterated, giving a fully homomorphic MAC, which can be instantiated under a variety of group-based or lattice-based assumptions [ILL25a].

Packed Fully Homomorphic MAC. Instead of authenticating a single bit x with the linear MAC $\mathsf{k}x + r$, we build on the construction of [ILL25a] and extend it to a packed setting, where both the message and the MAC key are now vectors of length ℓ. A MAC on $\vec{x} \in \{0,1\}^\ell$ under key $\vec{\mathsf{k}} \in \mathbb{Z}^\ell$ is then defined as:

$$\mathsf{Auth}(\vec{\mathsf{k}}, \vec{x}; r) = \vec{\mathsf{k}}^{\mathsf{T}} \vec{x} + r$$

where $r \in \mathbb{Z}$ is a random mask chosen from a suitably large range to ensure statistical hiding.

Letting $X_0 := r$ and $X_1 := \mathsf{Auth}(\vec{\mathsf{k}}, \vec{x}; r)$, this can again be seen as a subtractive sharing $X_1 - X_0 = \vec{\mathsf{k}}^{\mathsf{T}} \vec{x}$, where $\mathcal{P}_A$ holds $(\vec{x}, X_1)$, while $\mathcal{P}_B$ holds $(\vec{\mathsf{k}}, X_0)$.

The packed MACs remain linearly homomorphic, but the straightforward extension of the original construction breaks the crucial multiplicative homomorphism. To overcome this obstacle, our key idea is to introduce an extended structured public key, in which each position is associated with the relevant cross products of independent position keys k_i. This new key structure enables us to generalise the homomorphic multiplication procedure of [ILL25a] to the packed setting, resulting in a non-interactive way to obtain a succinct MAC on the component-wise product $\vec{x} \circ \vec{y}$.

Concretely, the resulting extended public key contains the elements $h_i = g^{\mathsf{k}_i}$, together with the matrix[1]:

[1] Since $\boldsymbol{H}$ is symmetric across its leading diagonal, only $\ell(\ell+1)/2$ entries need to be sent.

$$\boldsymbol{H} = \begin{pmatrix} g^{\mathsf{k}_1^2} f^{\mathsf{k}_1} & g^{\mathsf{k}_1\mathsf{k}_2} & \cdots & g^{\mathsf{k}_1\mathsf{k}_\ell} \\ g^{\mathsf{k}_2\mathsf{k}_1} & g^{\mathsf{k}_2^2} f^{\mathsf{k}_2} & \cdots & g^{\mathsf{k}_2\mathsf{k}_\ell} \\ \vdots & & \ddots & \\ g^{\mathsf{k}_\ell\mathsf{k}_1} & \cdots & & g^{\mathsf{k}_\ell^2} f^{\mathsf{k}_\ell} \end{pmatrix} \tag{2}$$

We now describe how to authenticate an homomorphic component-wise product of vectors: Suppose the parties hold authentications of two vectors $\vec{x}$ and $\vec{y}$, that is, $\mathcal{P}_A$ has $\vec{x}, \vec{y}, X_1, Y_1$ and $\mathcal{P}_B$ holds $\vec{\mathsf{k}}, X_0, Y_0$ such that $X_1 - X_0 = \vec{\mathsf{k}}^\mathsf{T}\vec{x}$ and $Y_1 - Y_0 = \vec{\mathsf{k}}^\mathsf{T}\vec{y}$. Notice that $\mathcal{P}_B$ can compute:

$$\begin{aligned} g^{X_0Y_0} &= g^{(X_1 - \vec{\mathsf{k}}^\mathsf{T}\vec{x})(Y_1 - \vec{\mathsf{k}}^\mathsf{T}\vec{y})} \\ &= g^{X_1Y_1 - X_1\vec{\mathsf{k}}^\mathsf{T}\vec{y} - Y_1\vec{\mathsf{k}}^\mathsf{T}\vec{x} + \sum_{i,j}\mathsf{k}_i\mathsf{k}_j x_i y_j} \\ &= g^{X_1Y_1}(\prod_i h_i^{y_i})^{-X_1}(\prod_i h_i^{x_i})^{-Y_1} g^{\sum_{i,j}\mathsf{k}_i\mathsf{k}_j x_i y_j} \\ &= \underbrace{g^{X_1Y_1}\prod_i h_i^{-X_1y_i - Y_1x_i}\prod_{i,j}\boldsymbol{H}_{i,j}^{x_iy_j}}_{\text{known to } \mathcal{P}_A}\prod_i f^{-\mathsf{k}_i x_i y_i} \end{aligned}$$

Using our notation from the technical sections, we can also write this more compactly as

$$= g^{X_1Y_1}\left(\left(\vec{\boldsymbol{h}}^\mathsf{T}\right)^{-X_1\vec{y} - Y_1\vec{x}}\right)\left(\vec{x}^\mathsf{T}\boldsymbol{H}^{\vec{y}}\right)\mathsf{diag}\left(f^{\vec{x}^\top(\vec{\mathsf{k}}\circ\vec{y})}\right)$$

Thus, the parties can obtain subtractive shares satisfying $Z_1 - Z_0 = \vec{\mathsf{k}}^\mathsf{T}\vec{z}$, where $\vec{z} = \vec{x}\circ\vec{y}$, as follows:

- $Z_0 = \mathsf{DDLog}(g^{X_0Y_0})$
- $Z_1 = \mathsf{DDLog}(g^{X_1Y_1}\prod_i h_i^{-X_1y_i - Y_1x_i}\prod_{i,j}\boldsymbol{H}_{i,j}^{x_iy_j})$

At this point, it holds that $Z_1 - Z_0 = \vec{\mathsf{k}}^\mathsf{T}\vec{z} \bmod N$. To be able to continue performing homomorphic operations, however, we need this equation to hold over the integers. Fortunately, a standard trick suffices to ensure this: as long as $\vec{\mathsf{k}}^\mathsf{T}\vec{z}$ is superpolynomially smaller than the modulus N, the shares can be lifted to correct shares over the integers with overwhelming probability.

Notice that, to carry out the above packed multiplication, $\mathcal{P}_B$ only has to perform $O(1)$ group operations to get his MAC key for the vector product, so we achieve homomorphism with *succinct computation* for the verifier. In particular, after a one-time, $O(\ell^2)$ cost of generating and sending M to $\mathcal{P}_A$, the parties can non-interactively perform a batched, homomorphic evaluation (or, SIMD computation) on MACs, where the parties evaluate the same circuit C on a batch of ℓ different, authenticated inputs. The computational cost of homomorphic evaluation is $O(\ell|C|)$ for $\mathcal{P}_A$, and just $O(|C|)$ for $\mathcal{P}_B$ (or, accounting for the setup cost of generating M, a total of $O(|C| + \ell^2)$).

Note that, as described here, our construction generalises the public information of [ILL25a] (recovered as the special case $\ell = 1$). As a result, it requires a circular security assumption, since $\boldsymbol{H}$ can be interpreted as an "encryption" of $\vec{\mathsf{k}}$ under $\vec{\mathsf{k}}$ itself. In the subsequent technical sections, we instead present an alternative that does not rely on circular security by assigning distinct key vectors to different circuit layers. In this variant, using similar ideas to [MORS25, ILL25a], we release multiple matrices $\boldsymbol{H}$, each containing "encryptions" of the key vectors at a given depth only under keys from lower depths, thereby ensuring that there is no cycle of encryptions of keys, thus avoiding the need for circularity security.

Generating Packed MACs. When using packed MACs in a protocol, it is sometimes useful to be able to set up a batch of MACs in a succinct way, that is, such that the work of $\mathcal{P}_B$ does not scale linearly with the number of $\mathcal{P}_A$'s messages being authenticated. One way to do this is using *succinct homomorphic secret sharing* for bilinear functions [ARS24]. This allows $\mathcal{P}_A$ and $\mathcal{P}_B$ to obtain additive shares of a function $\vec{y} = M\vec{x}$, where M is a matrix held by $\mathcal{P}_A$ and $\vec{x}$ a vector held by $\mathcal{P}_B$, by exchanging just one message each. Crucially, the size of the messages, and the running time of $\mathcal{P}_B$, only depend on the length of $\vec{x}$ and not on the number of rows in M. $\mathcal{P}_A$ can therefore pack into M all of the messages to be authenticated, while $\mathcal{P}_B$ chooses $\vec{x}$ to be the MAC key.

Beyond SIMD: Permuting Authenticated Vectors. So far, we have only achieved succinct computational costs (for $\mathcal{P}_B$) for a limited class of SIMD-style computations, which we refer to as operations on "bundled" wires containing ℓ bits each. To enable arbitrary computation, we additionally show how to perform permutations on bundled wires. Given some extra setup information, the parties can non-interactively apply "bit-flip swap" permutations of the form:

$$\boldsymbol{\mathfrak{C}}_j : (x_0, \ldots, x_{\ell-1}) \mapsto (x_{2^j}, x_{2^j \oplus 1}, \ldots, x_{2^j \oplus (\ell-1)}) \tag{3}$$

where $\oplus$ is bitwise XOR. In other words, each entry of $\vec{x}$ is swapped with the entry whose index differs only in position j.

This can be achieved by a careful modification of the multiplication procedure: we introduce a modified public key, analogous to $\boldsymbol{H}$ in (2), where the diagonal entries f^{k_i} are replaced by $f^{\boldsymbol{\mathfrak{C}}_j(\mathsf{k}_i)}$. Using this extended public key, $\mathcal{P}_B$ can apply the same DDLog operations as before to obtain shares satisfying $\boldsymbol{\mathfrak{C}}_j(\vec{\mathsf{k}}) \cdot \vec{x} = \vec{\mathsf{k}} \cdot \boldsymbol{\mathfrak{C}}_j(\vec{x})$, thereby authenticating the permuted vector. Using Beneš switching networks [Ben64], an arbitrary permutation on ℓ elements can then be realised by a circuit containing $O(\log \ell)$ bit-flip swap gates, giving only $O(\log \ell)$ computational cost for the verifier.

We can therefore perform homomorphic, authenticated computations for a class of circuits with ℓ-vector wire bundles, and the set of gates $\{+, *\} \cup \{\mathsf{perm}_\pi\}_\pi$, where π is any permutation on ℓ elements. Next, we leverage this to build a form of laconic function evalution for arbitrary Boolean circuits of sufficient width.

From Succinct Authentication to Attribute-Based LFE. In attribute-based LFE, a receiver $\mathcal{P}_A$ holds a large circuit C, while the sender $\mathcal{P}_B$ wants to encrypt a message μ together with a public attribute x such that $\mathcal{P}_A$ can recover μ if and only if $C(x) = 1$.[2] Equivalently, $\mathcal{P}_A$ should learn the output of:

$$(x, \mu) \mapsto (x, C(x)\mu)$$

Moreover, to fulfil the *laconic* requirement, the work of $\mathcal{P}_B$ should be much less than the size of C.

Our packed homomorphic MACs give a natural template for building AB-LFE, for highly parallelizable, bundled circuits, that is, containing ℓ-fold addition, multiplication and permutation gates. $\mathcal{P}_A$ can simply send to $\mathcal{P}_B$ a succinct *description* of the circuit (which can be up to $\approx \ell$ times smaller than the standard, Boolean circuit description). Then, $\mathcal{P}_B$ samples a packed homomorphic MAC key, and uses the evaluation algorithm to obtain a share of the dot product $(C(x), 0, \ldots, 0) \cdot \vec{\mathsf{k}} = C(x)\mathsf{k}_1$. Now, if $\mathcal{P}_B$ encrypts μ under the key k_1 in the Damgård-Jurik group, this share can be locally converted into a share of $C(x)\mu$, using

[2] While typically, LFE and ABE allow decryption iff $C(x) = 0$, we use $C(x) = 1$, to simplify the function to $C(x)\mu$.

DDLog and a standard HSS multiplication procedure.

$\mathcal{P}_B$ then sends to $\mathcal{P}_A$ a MAC on the public input x, together with the MAC evaluation key, the encryption of $\mu \mathsf{k}_1$ and his final output share, denoted $\langle C(x)\mu \rangle_0$. This allows $\mathcal{P}_A$ to perform her side of the homomorphic evaluation to obtain $\langle C(x)\mu \rangle_1$, and hence recover $C(x)\mu$.

The main drawback of this approach is that it only achieves succinct computation when the circuit is parallelizable enough to have a compact description. A secondary concern is that the construction is not *function-hiding*; however, this could be addressed by running $\mathcal{P}_B$'s computation inside a garbled circuit [QWW18].

Designing Universal Circuits with Low Depth. We resolve the above issue — while also giving a simple way to hide the function — through a careful combination of universal circuits and succinct HSS. Instead of sending the description of the circuit C in the clear, $\mathcal{P}_A$ will *authenticate* it under the packed MAC scheme, using succinct HSS. This allows $\mathcal{P}_B$ to get shares of $\vec{c}_i^\top \vec{\mathsf{k}}$, for each length-$\ell$ chunk $\vec{c}_i$ of C, while only paying $O(|C|/\ell)$ work.

Now, to evaluate C, $\mathcal{P}_B$ can apply a universal circuit, $\mathcal{U}$, to the authenticated C and input x. Crucially, we observe that standard universal circuit constructions, which have size $O(|C| \log |C|)$, have a *compact description* that can be succinctly stored by $\mathcal{P}_B$. Furthermore, if the original circuit C is sufficiently wide, then a universal circuit $\mathcal{U}$ can be viewed as a *bundled circuit*, which allows $\mathcal{P}_B$ to perform his side of the *homomorphic* evaluation of $\mathcal{U}(C, x)$ with only $\tilde{O}(|C|/\ell)$ operations. One challenge in making this work, is that in our HSS construction based on DCR, the public evaluation key grows with the depth D of the circuit, since we need key-switching information for each layer. Standard universal circuits based on Valiant's construction unfortunately have a depth that grows with the worst-case depth of the circuit C, which would ruin our efficiency property. Through careful manipulation of packed bundles of different sizes, we show how to construct a lower depth version of the universal circuit that suffices for our construction.

Given this, our final construction is roughly as follows. We assume a succinct HSS scheme, biHSS, which allows the two parties to exchange messages $\mathsf{Hash}(\vec{x})$ and $\mathsf{Enc}(\vec{y})$, which can then be converted into shares of $\vec{x}^\top \vec{y}$ via $\mathsf{biHSS.Eval}$. Similarly, we use $\mathsf{MAC.Eval}_0/\mathsf{MAC.Eval}_1$ to denote the respective homomorphic MAC evaluation algorithms, and $\mathsf{HSS.Mul}$ a standard, HSS multiplication algorithm between a ciphertext and shared value.

- As setup, sample $\mathsf{crs} \leftarrow \mathsf{biHSS.crsGen}(1^\lambda)$
- $\mathcal{P}_A$ hashes (the description of) the circuit C, computing $\mathsf{digest}_C = \mathsf{biHSS.Hash}(\mathsf{pp}; C)$
- $\mathcal{P}_B$ encrypts a message $\mu \in \{0,1\}$, with respect to the attribute $\vec{x} \in \{0,1\}^\ell$:
  1. Sample a key $\vec{k}$, and evaluation key $\mathsf{evk} = (\{h_i\}_i, \vec{\boldsymbol{H}})$ as above
  2. Compute $\mathsf{ct}_\mathsf{k} = \mathsf{biHSS.Enc}(\vec{k})$ and $\mathsf{ct}_\mu = \mathsf{HSS.Enc}_{\vec{k}}(\mu)$
  3. Sample random shares satsifying $\langle \vec{x}^\top \vec{k} \rangle_1 - \langle \vec{x}^\top \vec{k} \rangle_0 = \vec{x}^\top \vec{k}$
  4. Run $\mathsf{biHSS.Eval}_0$ (using digest_C and $\vec{k}$) to get authentications of each chunk of C, denoted $\langle \vec{c}_i^\top \vec{k} \rangle_0$
  5. Run $\mathsf{MAC.Eval}_0(\mathcal{U}, (\langle \vec{x}^\top \vec{k} \rangle_0, \{\langle \vec{c}_i^\top \vec{k} \rangle_0\}_i))$ to get $\langle \mathcal{U}(C, \vec{x}) \cdot k_1 \rangle_0 = \langle C(\vec{x}) \cdot k_1 \rangle_0$.
  6. Run $\mathsf{HSS.Mul}(\mathsf{ct}_\mu, \langle C(\vec{x}) \cdot k_1 \rangle_0)$ to get $z_0 := \langle \mu \cdot C(\vec{x}) \rangle_0$
  7. Send $\vec{x}$ and the ciphertext $\mathsf{ct} = (\mathsf{evk}, \mathsf{ct}_\mathsf{k}, \mathsf{ct}_\mu, \langle \vec{x}^\top \vec{k} \rangle_1, \langle \mu \cdot C(\vec{x}) \rangle_0)$
- To decrypt, given $\vec{x}$ and ct:
  1. Compute shares $\langle \vec{c}_i^\top \vec{k} \rangle_1 = \mathsf{biHSS.Eval}_1(\vec{c}_i, \mathsf{ct}_\mathsf{k})$, for each i
  2. Run $\mathsf{MAC.Eval}_1(\mathsf{evk}; \mathcal{U}, (\vec{x}, C), (\langle \vec{x}^\top \vec{k} \rangle_1, \{\langle \vec{c}_i^\top \vec{k} \rangle_1\}_i))$ to obtain z_1'
  3. Run $\mathsf{HSS.Mul}(\mathsf{ct}_\mu, z_1')$ to get $z_1 = \langle \mu \cdot C(x) \rangle_1$.
  4. If $C(\vec{x}) = 1$, recover $\mu = z_1 - z_0$.

HSS Framework for Bundled, Homomorphic Computation. The above template uses HSS techniques in different ways, namely, in the "standard" HSS multiplication algorithm $\mathsf{HSS.Mul}$, for authenticating inputs with succinct HSS, and for the homomorphic MAC evaluation based on DDLog. To formalise our construction, we present a unified HSS framework that combines standard HSS with *semi-private* operations, also used in [MORS25], where one party knows the messages for the semi-private parts of the HSS computation. We additionally augment the HSS syntax to allow for bundled (vector) wire types and gate operations for both private and semi-private wires. This gives a general, composable framework that supports homomorphic evaluation of functions of the form $C_{\mathsf{spriv}}(x) \cdot C_{\mathsf{priv}}(y)$, where C_{priv} is a "restricted multiplication" circuit (e.g. any NC1 circuit) applied to a secret input y, and C_{pub} is any polynomially-sized circuit applied to a semi-private input x known to one party, which also supports packed evaluation with sublinear work for the other party. This allows for a simple specification of our AB-LFE construction, and is also easily reusable for other applications of our HSS techniques.

Application to Constrained PRFs. Couteau, Meyer, Paselegue and Rahinia [CMPR23] presented a general template for building constrained pseudorandom functions from HSS, which was further improved in [ILL25a,BMO+25]. A CPRF requires being able to derive from the master key a 'constrained key' that allows evaluating the PRF on any input where $C(x) = 0$, for some constraint function C. The high-level idea of [CMPR23] is to homomorphically evaluate $C(x)$ on an authenticated version of the public input x. Then, instead of simply multiplying the result with an encryption ct_μ (as above for AB-LFE), a secondary HSS evalution is performed on a private input k, to compute a PRF evaluation $F_k(x)$ that is multiplied with $C(x)$.

This type of computation is exactly what is supported by our HSS framework, so we can easily use it to build a constrained PRF for arbitrary circuits.[3] Unlike previous constructions [CMPR23, ILL25a, BMO+25], we additionally obtain a *succinctness* property, where, for sufficiently wide C, the size of the master secret key (and master evaluation algorithm) is sublinear in the size of C.

2.1 Concurrent Work of [LLL25]

Our work is based on the idea that a single integer can be used to authenticate a whole vector of values, by taking a dot product with a key vector, and that such a scheme is compatible with ElGamal-style encryption schemes. The group-based constructions in the concurrent work of [LLL25] are based on a similar idea, where authenticating a (secret) hash function evaluated on a vector is sufficient to authenticate the vector itself. It is likely, therefore, that our construction could be made to work in their context and vice versa. On the other hand, we develop this idea in a very different direction from them: we focus on computational succinctness for privacy-free authentication problems (AB-LFE and constrained PRFs), while they focus on reducing the depth complexity of the garbler in succinct garbling (both private and privacy-free).

2.2 Related Work

We discuss a few related techniques and concepts in prior literature.

[3] One technical detail is that the CPRF master key cannot depend on the circuit, C. Here, instead of relying on succinct HSS, the [CMPR23] framework uses a form of "offline-online" shares of C, where one party's share can be sampled before knowing C.

Homomorphic Encryption and LFE From Group-Based Assumptions. There are constructions of group-based HE and LFE for weak classes of functions, including well-known constructions of linear HE from DDH or DCR, degree-2 HE from pairings [BGN05], or a somewhat compact form of HE for branching programs [IP07]. The recent work of [CHKV25] constructed HE for low-degree functions based on linear HE and the sparse LPN assumption. LFE for linear functions and predicates can also be obtained from group-based assumptions, using the k-linear assumption in prime-order groups [QWW18].

Conditional Disclosure of Secrets and Partial Garbling. Partial garbling [IW14] is a variant of garbled circuits, where part of the input to the circuit is publicly known. Conditional disclosure of secrets can be viewed as partial garbling for the function $C(x) \cdot \mu$, where x is public and μ is a private input of the garbler.

[ILL25a] construct succinct, CDS schemes for arbitrary circuits C, where the size of the garbled circuit is $\mathsf{poly}(\lambda)$ or $\mathsf{poly}(\lambda \log |C|)$. Our AB-LFE construction implicitly builds a succinct partial garbling scheme (or CDS) for the function $C(x) \cdot \mu$, with greatly improved efficiency: the garbler's *runtime* is sublinear in the size of C (assuming the garbler knows a compact description, or hash, of C). As in [ILL25a], this can in turn be composed with a standard garbling scheme to obtain partial garbling for functions of the form $C_{\mathsf{priv}}(C_{\mathsf{pub}}(x), y)$. Now, the private part of the computation requires linear communication and computation, while the public part retains the succinct garbling phase.

Attribute-Based Encryption. While AB-LFE and attribute-based encryption have similarities, neither is known to imply the other. ABE requires a collusion-resistant property, where one can give out multiple secret keys for different circuits C, which we do now know how to achieve. On the other hand, standard ABE does not require any form of computational succinctness.

Information-Theoretic Packed MACs. [WYY+22] proposed a form of packed, information-theoretic MAC for use in zero-knowledge proofs of SIMD circuits, which were later improved in [BCC+23]. They use a polynomial-based MAC, where a message is encoded into the roots of a polynomial, p, over a finite field $\mathbb{F}$, and then authenticated under a key $\mathsf{k} = (\Delta, k) \in \mathbb{F}^2$ via $\mathsf{Auth}(\mathsf{k}, p) = p(\Delta) \cdot k + r$. This MAC was combined with somewhat homomorphic encryption to build interactive ZK proofs with sublinear communication complexity.

Bundled Circuits. We follow previous work in achieving computational efficiency / succinctness by evaluating a circuit that works on bundles of wires at a time, and using techniques for routing wires within and between bundles. Previous work [DIK10, GHS12] constructed compilers for converting sufficiently-wide boolean circuits into bundled circuits, with a polylog multiplicative overhead in circuit size and circuit depth (logarithmic and log-squared respectively for [DIK10]). The main underlying tool is the Beneš switching network [Ben64], which implements an arbitrary permutation using a highly regular network of conditional swap gates. Since we need the bundled circuit to be universal (so that original circuit's description can be hashed), we combine these techniques with Valiant's construction of universal circuits [Val76]. We achieve a qualitatively and quantitatively stronger result. Quantitatively, we achieve logarithmic multiplicative overhead for both circuit depth and circuit size, and qualitatively we construct a universal circuit instead of a circuit compiler.

3 Preliminaries

Notation. The security parameter is λ. We write $[n] = \{1, \ldots, n\}$, $[a, b] = \{a, \ldots, b\}$ and $[a, b) = \{a, \ldots, b-1\}$. We sometimes use $\langle x \rangle$ to denote that x is split into subtractive

shares $\langle x \rangle_1 - \langle x \rangle_0 = x$, over the integers. We use the mod$N$ operation both as a function that reduces the input modulo N and returns the corresponding integer in $[0, N)$, and also occasionally as a congruence, $a \equiv b \bmod N$.

3.1 Damgård-Jurik-ElGamal Cryptosystem

Theorem 1. *Assuming DCR, Damgård–Jurik–ElGamal encryption Fig. 1 is a public key encryption scheme satisfying correctness and KDM security for affine functions of the key. Specifically, the following properties hold:*

Correctness: $\mathsf{DJE.Dec}_{\mathsf{sk}}(\mathsf{DJE.Enc}_{\mathsf{pk}}(x)) = x$ *and* $\mathsf{DJE.Dec}_{\mathsf{sk}}(\mathsf{DJE.EncK}_{\mathsf{pk}}(x)) = k \cdot x$*, for any* $(\mathsf{sk} = (k, N), \mathsf{pk})$ *in the support of* $\mathsf{DJE.KeyGen}$*, and any* $x \in \mathbb{Z}/N^{\zeta}\mathbb{Z}$*.*

CPA-Security: *Both* Enc *and* EncK *are IND-CPA-secure.*

The proof is deferred to Section F.1

DJE Damgård-Jurik-ElGamal Cryptosystem

`Requires:`

- $\zeta \geq 1$ is a parameter defining the plaintext size.
- Group isomorphism $\exp\colon (\mathbb{Z}/N^{\zeta}\mathbb{Z})^+ \to 1 + N(\mathbb{Z}/N^{\zeta+1}\mathbb{Z})$ and its inverse $\log\colon 1 + N(\mathbb{Z}/N^{\zeta+1}\mathbb{Z}) \to (\mathbb{Z}/N^{\zeta}\mathbb{Z})^+$, as defined in [RS21]:

$$\exp(x) = \sum_{k=0}^{\zeta} \frac{(Nx)^k}{k!} \qquad \text{and} \qquad \log(1+Nx) = \sum_{k=1}^{\zeta} \frac{(-N)^{k-1} x^k}{k}$$

$\mathsf{DJE.Setup}(1^\lambda)$:
1. Sample $(N, p, q) \xleftarrow{\$} \mathsf{RSA.Gen}(1^\lambda)$
2. Sample $\boldsymbol{g} \xleftarrow{\$} (\mathbb{Z}/N^{\zeta+1}\mathbb{Z})^\times$
3. Output $\mathsf{pp} = (\boldsymbol{g}, N)$

$\mathsf{DJE.Enc}_{\mathsf{pk}}(x)$:
1. Parse $\mathsf{pk} = (\boldsymbol{g}, \boldsymbol{h}, N)$
2. Sample $r \xleftarrow{\$} [0, N)$
3. Output $c = (\boldsymbol{g}^r, \boldsymbol{h}^r \cdot \exp(x))$

$\mathsf{DJE.EncK}_{\mathsf{pk}}(x)$:
1. Parse $\mathsf{pk} = (\boldsymbol{g}, \boldsymbol{h}, N)$
2. Sample $r \xleftarrow{\$} [0, N)$
3. Output $c = (\boldsymbol{g}^r \cdot \exp(x), \boldsymbol{h}^r)$

$\mathsf{DJE.KeyGen}(1^\lambda, \mathsf{pp})$:
1. Parse pp as $\mathsf{pp} = (\boldsymbol{g}, N)$
2. Sample $k \xleftarrow{\$} [0, N)$
3. Compute $\boldsymbol{h} \leftarrow \boldsymbol{g}^{-k}$
4. Output $(\mathsf{sk} = (k, N), \mathsf{pk} = (\boldsymbol{g}, \boldsymbol{h}, N))$

$\mathsf{DJE.Dec}_{\mathsf{sk}}(c = (\boldsymbol{c}_0, \boldsymbol{c}_1))$:
1. Parse $\mathsf{sk} = (k, N)$
2. Assert $\boldsymbol{c}_0^k \cdot \boldsymbol{c}_1 \equiv 1 \mod N$
3. Output $x \leftarrow \log(\boldsymbol{c}_0^k \cdot \boldsymbol{c}_1)$

Fig. 1: The Damgård-Jurik-ElGamal cryptosystem.

3.2 Distributed Discrete Logarithm

DDLOG Damgård-Jurik Distance Function [RS21]

$\mathsf{DDLog}_N(h \in \mathbb{Z}/N^{\zeta+1}\mathbb{Z})$:
Compute and output $z \leftarrow \log\left(\frac{h}{h \bmod N}\right) \in \mathbb{Z}/N^{\zeta}\mathbb{Z}$

Fig. 2: [RS21]'s distributed discrete logarithm for Damgård-Jurik.

Lemma 2 (Distributed Decryption of Damgård-Jurik [RS21, Theorem 18]). *For all $N \in \mathsf{Supp}(\mathsf{RSA.Gen})$, $c \in (\mathbb{Z}/N^{\zeta+1}\mathbb{Z})^{\times}$, and $x \in \mathbb{Z}$,*

$$\mathsf{DDLog}_N(c\exp(x)) - \mathsf{DDLog}_N(c) \equiv x \bmod \mathbb{N}^{\zeta}$$

We also rely on the following, standard lemma stating that shares of some value x modulo M can be lifted to valid shares over the integers, with a failure probability of $|x|/M$.

Lemma 3 (From e.g. [RS21, Lemma 19]). *For all moduli $M > 1$ and all modulo M shares $\langle x\rangle_0, \langle x\rangle_1 \in \mathbb{Z}/M\mathbb{Z}$ of some $x \in \mathbb{Z}$, we have*

$$\Pr_{r \xleftarrow{\$} \mathbb{Z}/M\mathbb{Z}}\Big[(\langle x\rangle_1 + r) \bmod M - (\langle x\rangle_0 + r) \bmod M = x\Big] = \max\left(1 - \frac{|x|}{M}, 0\right).$$

3.3 Laconic Function Evaluation

Laconic Function Evaluation (or LFE) [QWW18] is a cryptographic primitive which can be seen as a "flipped"[4] variant of fully homomorphic encryption. The server first commits its (large) function f to a small digest d, and the client then uses this digest to encrypt its input x in such a way that the resulting ciphertext can be publicly decoded to $f(x)$.
The original definition of [QWW18] required perfect correctness, which our DDLog-based techniques do not support easily. We therefore adapt a definition from [HLL23], which considers imperfect correctness. Our definition follows their definition of computationally succinct, adaptively correct and secure LFE.
We also define two succinctness properties. Circuit succinctness is the standard succinctness of AB-LFE; nothing except the runtimes of $\mathsf{Compress}$ and Dec should grow linearly with the circuit size. We also consider *attribute succinctness*, which requires the ciphertext (and crs) to not grow linearly with the attribute; this has been referred to previously as laconic function evaluation with rate 1 ciphertexts [Wee24]. We defer the formal correctness, security and succinctness properties to Section G.

Definition 4 (Attribute-Based Laconic Function Evaluation, [QWW18], [HLL23]). *A* Function-Private Attribute-Based Laconic Function Evaluation *(AB-LFE) scheme for a class of arithmetic circuits $\mathcal{C}$ is a tuple of algorithms $\mathsf{LFE} = (\mathsf{crsGen}, \mathsf{Compress}, \mathsf{Enc}, \mathsf{Dec})$ with the following syntax:*

- $\mathsf{crsGen}(1^{\lambda}, \mathsf{params})$: *On input the security parameter and circuit parameters* params*, the common random string generation algorithm* crsGen *outputs a string* crs *(of appropriate length).*

[4] This description is borrowed from [DHMW24].

- $\mathsf{Compress}(\mathsf{crs}, C)$: *On input the common random string* crs*, an arithmetic circuit* $C \in \mathcal{C}$*, the randomized circuit compression algorithm* Compress *outputs a digest* h *and a compression state* ψ*.* Compress *must be reverse sampleable – the random tape must be simulatable using* (C, h, ψ)*.*
- $\mathsf{Enc}(\mathsf{crs}, h, x, m)$: *On input the common random string* crs*, a circuit digest* h*, an attribute* $x \in \mathcal{X}$*, and a message* m*, the encryption algorithm* Enc *outputs a ciphertext* ct*.*
- $\mathsf{Dec}(\mathsf{crs}, C, x, \mathsf{ct}, \psi)$: *On input the common random string* crs*, an arithmetic circuit* $C \in \mathcal{C}$*, an attribute* $x \in \mathcal{X}$*, ciphertext* ct*, and compression state* ψ*, the decryption algorithm* Dec *outputs a message* m*.*

3.4 Constrained Pseudorandom Functions

A Constrained Pseudorandom Function [BW13, KPTZ13, BGI14] is a PRF where one can derive a constrained key that allows for evaluating the PRF only on inputs satisfying a constraint predicate C, without leaking information about the PRF's output on inputs that do not satisfy the constraint. Formally, we use a simulation-based definition in the style of Peikert-Shiehian [PS18], which simultaneously captures pseudorandomness on unauthorised inputs and correctness of constrained evaluation on authorised inputs. Note that we do not require privacy of the constraining function.

Definition 5 (Contrained Pseudorandom Function). *A* constrained PRF *(CPRF) with input space* $\{0,1\}^n$ *and output space* $\{0,1\}^\mu$ *and supporting as constraints a class of class of circuits* $\mathcal{C}$ *(where each circuit in* $\mathcal{C}$ *is from* $\{0,1\}^n$ *to* $\{0,1\}$*) is a tuple of algorithms* $\mathsf{CPRF} = (\mathsf{CPRF.KeyGen}, \mathsf{CPRF.Eval}, \mathsf{CPRF.Constrain}, \mathsf{CPRF.CEval})$ *with the following syntax and properties:*

- $\mathsf{CPRF.KeyGen}(1^\lambda) \to \mathsf{msk}$*: On input a security parameter* 1^λ*, the key generation algorithm* KeyGen *produces a master secret key* msk*.*
- $\mathsf{CPRF.Eval}(1^\lambda, \mathsf{msk}, x) \to y$*: On input a security parameter* 1^λ*, a master secret key* msk*, and an input* $x \in \{0,1\}^n$*, the evaluation algorithm* Eval *output an element* $y \in \{0,1\}^\mu$*.*
- $\mathsf{CPRF.Constrain}(1^\lambda, \mathsf{msk}, C) \to \mathsf{ck}$*: On input a security parameter* 1^λ*, a master secret key* msk*, and a constraining circuit* $C \in \mathcal{C}$*, the constraining algorithm* Constrain *outputs a constrained key* ck*.*
- $\mathsf{CPRF.CEval}(1^\lambda, \mathsf{ck}, x) \to y$*: On input a security parameter* 1^λ*, a constrained secret key* ck*, and an input* $x \in \{0,1\}^n$*, the constrained evaluation algorithm* CEval *output an element* $y \in \{0,1\}^\mu$*.*
- **Single-key selective CPRF security.** *A constrained PRF is* single-key selectively secure *if for any* p.p.t. *adversary* $\mathcal{A}$ *(which we require to never repeat a query) and any* $s \in \mathsf{poly}(\lambda)$ *(the constraint size) there exists a* p.p.t. *simulator* $\mathcal{S}$ *such that:*

$$\left\{\mathsf{Real}_{\mathcal{A}}(1^\lambda, 1^s)\right\}_{\lambda\in\mathbb{N}} \overset{c}{\approx} \left\{\mathsf{Ideal}_{\mathcal{A},\mathcal{S}}(1^\lambda, 1^s)\right\}_{\lambda\in\mathbb{N}}$$

where Real *and* Ideal *are the respective views of* $\mathcal{A}$ *in the experiments of Fig. 3.*

Experiment Constrained PRF Security

$\mathsf{Real}_{\mathcal{A}}(1^\lambda, 1^s)$:

1. $C \xleftarrow{\$} \mathcal{A}(1^\lambda, 1^s)$
2. $\mathsf{msk} \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda)$
3. $\mathsf{ck} \xleftarrow{\$} \mathsf{Constrain}(1^\lambda, \mathsf{msk}, C)$
4. Send ck to $\mathcal{A}$
5. Repeat until $\mathcal{A}$ halts:
   (a) Wait to receive x from $\mathcal{A}$
   (b) $y \xleftarrow{\$} \mathsf{Eval}(1^\lambda, \mathsf{msk}, x)$
   (c) Send y to $\mathcal{A}$

$\mathsf{Ideal}_{\mathcal{A},\mathcal{S}}(1^\lambda, 1^s)$

1. $C \xleftarrow{\$} \mathcal{A}(1^\lambda, 1^s)$
2. $\mathsf{ck} \xleftarrow{\$} \mathcal{S}(1^\lambda, 1^s, C)$
3. Send ck to $\mathcal{A}$
4. Repeat until $\mathcal{A}$ halts:
   (a) Wait to receive x from $\mathcal{A}$
   (b) If $C(x) = 0$: $y \xleftarrow{\$} \mathsf{CEval}(1^\lambda, \mathsf{ck}, x)$
   (c) If $C(x) = 1$: $y \xleftarrow{\$} \{0,1\}^\mu$
   (d) Send y to $\mathcal{A}$

Fig. 3: Real and Ideal worlds of CPRF security (Definition 5).

3.5 Succinct, Bilinear HSS

We use succinct homomorphic secret sharing for bilinear functions, introduced in [ARS24]. Succinct HSS requires a sharing phase consisting of one, simultaneous message from each party, after which the parties can locally run an Eval algorithm to directly obtain secret shares of a bilinear function of the inputs.
We use bilinear HSS for the inner product function $\vec{x}^\top \vec{y}$ over a ring R (with the output required to be in a subset $\mathcal{B} \subseteq \mathcal{R}$), where the *succinctness* requirement is that the message of one party must be independent of the length of the vectors. We further extend this to a *computational* succinctness property for the Eval algorithm of the other party.
Below, we give the formal syntax of this flavour of bilinear HSS. The correctness and security properties are deferred to Section C.

Definition 6 (Bilinear HSS). *Let $\mathcal{R}$ be a commutative ring, and $\mathcal{B} \subseteq \mathcal{R}$. An $\mathcal{R}$-bilinear $\mathcal{B}$-bounded*[5] *HSS (biHSS) scheme is a tuple of PPT algorithms ($\mathsf{crsGen}, \mathsf{Hash}, \mathsf{Matrix}, \mathsf{HasherEval}, \mathsf{EncEval}$) with the following syntax: (for randomised algorithms, we use $\xrightarrow{\$}$, and for deterministic algorithms, $\to$)*

- $\mathsf{biHSS.crsGen}(1^\lambda, 1^\ell) \xrightarrow{\$} \mathsf{crs}$. *On input the security parameter and input length, outputs a common reference string.*
- $\mathsf{biHSS.Hash}(\mathsf{crs}, \vec{x}) \xrightarrow{\$} (d, \psi)$. *On input the CRS and an input vector $\vec{x} \in \mathcal{R}^\ell$, outputs a digest d and secret state ψ.*
- $\mathsf{biHSS.Enc}(\mathsf{crs}, \vec{y}) \xrightarrow{\$} (E, \phi)$. *On input the CRS and a vector $\vec{y} \in \mathcal{R}^\ell$, outputs an encoding E and secret state ϕ.*
- $\mathsf{biHSS.HasherEval}(\mathsf{crs}, E, \psi) \to z^{(0)}$. *On input a vector encoding E, and hasher secret state ψ, outputs a share $z^{(0)} \in \mathcal{R}$.*
- $\mathsf{biHSS.EncEval}(\mathsf{crs}, d, \phi) \to z^{(1)}$. *On input the CRS, a digest d and encoder secret state ϕ, outputs a share $z^{(1)} \in \mathcal{R}$.*

The constructions from [ARS24] are summarised as follows.

Theorem 7 (from [ARS24]). *Assume one of: (1) $\mathcal{R} = \mathbb{Z}_{N^{\zeta-1}}$, for any $\zeta \geq 2$, and the*

[5] If $\mathcal{B}$ is ommitted then $\mathcal{B} = \mathcal{R}$.

DCR assumption over $\mathbb{Z}_{N^2}$; (2) $\mathcal{R} = \mathbb{Z}_q$ for any prime $q = 2^{\Omega(\lambda)}$, and the extended, ℓ-DDH assumption over class groups; or (3) $\mathcal{R} = \mathbb{Z}_p$ for any superpolynomial p, and the LWE assumption with superpolynomial modulus-to-noise ratio.
Then, there exists an $\mathcal{R}$-bilinear HSS scheme with strong hasher privacy, strong encoder privacy, and computational succinctness. In each construction, $\mathsf{biHSS.Enc}$ *runs in time $O(\mathsf{poly}(\lambda, \log M) \cdot \ell)$, where M is either $N^{\zeta-1}$, q, or p, depending on the underlying assumption.*

We remark that in the second case (class groups) and third case (LWE), the CRS is simply a uniformly random string. The first case (DCR) requires an RSA modulus with unknown factorisation.
Like [ARS24], we really are concerned with performing inner products over $\mathbb{Z}$, rather than wrapping around in a finite ring $\mathcal{R}$. Using the share lifting lemma Lemma 3, reducing modulo M the output shares of any of the constructions from Theorem 7 immediately gives a $\mathbb{Z}$-bilinear HSS construction.

Corollary 8. *Under any of the assumptions of Theorem 7, for any $B = 2^{O(\lambda)}$ there exists a $\mathbb{Z}$-bilinear $[-B, B]$-bounded HSS scheme with strong hasher privacy, strong encoder privacy, and computational succinctness, where* $\mathsf{biHSS.Enc}$ *runs in time $O(\mathsf{poly}(\lambda, \log B) \cdot \ell$.*

3.6 Circuits

In order to achieve the desired level of succinctness, our constructions rely on carefully tailored notions of circuits. In particular, we make essential use of *bundled* circuits, which allow operations to be performed in parallel on vectors of values.
To support our constructions, we require a precise and flexible framework for reasoning about circuits. In particular, we need a formalism that can capture different wire types (scalars, vectors), enforce well-typed operations, and cleanly switch between plaintext and homomorphic semantics. We therefore adopt and adapt the typed circuit framework of [RS21], which provides the right level of generality for our purposes. While many of the following definitions are borrowed from prior work, we include them here for completeness and to set up the exact variants needed for our succinct constructions.

Typed Circuits. A *circuit prototype* specifies a class of typed circuits by defining sets $(\mathsf{types}, \mathsf{gates}, \mathsf{in}, \mathsf{out})$, which describe the possible abstract wire types, gate types, and mappings such that $\mathsf{in}(g)$ specifies the input type of gate g, and $\mathsf{out}(g)$ the output type.
A *typed circuit* is a circuit prototype, together with a DAG $(\mathsf{nodes}, \mathsf{wires})$ specifying the circuit structure, subset $\mathsf{inputs}, \mathsf{outputs} \subseteq \mathsf{nodes}$ for the inputs and outputs, and two functions specifying the type of the output wire of a node v, as $\mathsf{type}(v) \in \mathsf{types}$, and the gate type $\mathsf{gate}(v) \in \mathsf{gates}$.
To evaluate a typed circuit, we first need to specify the concrete wire types and gate operations. This is done by assigning an *evaluation semantics* to the circuit. We consider two forms of semantics: a *plaintext evaluation semantics*, for the usual evaluation of a circuit, and a *homomorphic evaluation semantics*, which specifies how one party homomorphically evaluates their shares inside HSS. In each case, the semantics consists of a values function that assigns each wire type $w \in \mathsf{types}$ a set of values, and an eval function such that $\mathsf{eval}(g)$ outputs the gate evaluation function for gate $g \in \mathsf{gates}$. When specifying a homomorphic evaluation semantics, we typically define a separate semantics for each party, that is, distinct evaluation functions eval_0 and eval_1.
We give further, formal details in Section A.

3.6.1 Circuit Classes. In Section A, we define the standard notion of arithmetic circuits over a ring $\mathcal{R}$. Our constructions rely on three other (less standard) circuit classes defined below.

Restricted Multiplication (RM) Circuits. Intuitively, RM circuits distinguish between two kinds of wires: input values IN, which are the "fresh" inputs allowed to participate in non-linear multiplications, and register values REG, which store intermediate results but can only be multiplied by an IN.

Definition 9 (Bounded, Restricted Multiplication circuit [RS21]). *The* RM circuit prototype *over the ring $\mathcal{R}$ has wire types* $\mathsf{types} := \{\mathsf{IN}, \mathsf{REG}\}$*, gate types for constants and linear operations* $\{1_{\mathsf{IN}}, 1_{\mathsf{REG}}, +_{\mathsf{IN}}, +_{\mathsf{REG}}, \times_{\mathsf{IN},c}, \times_{\mathsf{REG},c}\}$ *for all $c \in \mathcal{R}$, as well as the single, non-linear gate* $\times : \mathsf{IN} \times \mathsf{REG} \to \mathsf{REG}$.
*An M-*bounded RM circuit*, for bound $M \subset \mathcal{R}$, is a circuit for this prototype, and is associated with the plaintext evaluation semantics with wire types* $\mathsf{values}(\mathsf{IN}) = \mathsf{values}(\mathsf{REG}) = \mathcal{R} \cup \{\perp\}$*, where $\perp$ is a special failure symbol. Gate operations perform the usual operation in $\mathcal{R}$, except always output $\perp$ if any input x to the gate is $\perp$, or if $x \notin M$.*

The value $\perp$ allows the circuit evaluation to fail if any gate input exceeds the allowed range. As in [RS21], we call such an evaluation semantics a *failure semantics*.

Batched Circuits. Arithmetic circuits (Definition 27) are usually described as being evaluated gate by gate. In practice, many gates can be computed in parallel whenever they do not depend on one another in the circuit's DAG. A *batched representation* makes this parallelism explicit by assigning "colours" to the nodes of the circuit: any two nodes connected by a path in the DAG (i.e., with a sequential dependency) must receive different colours. Each colour class then corresponds to one batch of gates that can be evaluated simultaneously. We say the circuit admits an *ℓ-batching* if every batch contains at most ℓ gates.

Bundled Circuits. Bundled circuits extend the arithmetic model by allowing each wire to carry an ℓ-dimensional vector instead of a single value. While basic ring operations $(+, \times)$ act component-wise across the vector, the model also supports permutations (implemented with networks of bit-flip swaps) that enable non-trivial reordering of components. This flexibility allows bundled circuits to capture both parallel arithmetic and general vector manipulations, but is more restrictive than batched circuits.
We actually use a *levelled* form of bundled circuits, where we explicitly keep track of the depth at which every gate operates. This bookkeeping does not add any new expressive power compared to standard bundled circuits, but is crucial for avoiding circular security assumptions in our constructions. By assigning each value a level and enforcing that multiplications or permutations increase the level in a controlled way, we ensure that authentication keys never form a cycle. Like RM circuits, we assume a magnitude bound on the computation values.

Definition 10 (Levelled, bundled circuit). *Let $\ell \in 2^{\mathbb{N}}$ be a* bundling parameter*, $D \in \mathbb{N}$ a maximum depth parameter and $M \subset \mathcal{R}$ a range bound. The* levelled, ℓ-bundled circuit prototype *over the ring $\mathcal{R}$ for depth D and bound M is defined by wire types* $\mathsf{types} := \{\mathsf{vector}_d : d \in \mathbb{N}\}$*, maps* $\mathsf{in} : \mathsf{gates} \to \mathsf{types}^\star$*,* $\mathsf{out} : \mathsf{gates} \to \mathsf{types}$*, and the set of gate types* $\mathsf{gates} = \{g\}_g$ *defined in Table 1.*
We further specify the plaintext evaluation semantics $\mathsf{eval}(g)$ *for each gate, over a ring $\mathcal{R}$, when considering* $\mathsf{values}(\mathsf{vector}) = \mathcal{R}^\ell \cup \{\perp\}$.[6]

[6] As with RM circuits, if any gate input is $\perp$ or outside the range M, the output is $\perp$.

Gate type, g	Input type, $\mathsf{in}(g)$	Output type, $\mathsf{out}(g)$	$\mathsf{eval}(g)$
$\{\mathsf{constant}_{v,d}\}_{v\in\mathcal{R}^\ell}$	$\varnothing$	vector_d	$\{\} \mapsto v$
add_d	$\mathsf{vector}_d \times \mathsf{vector}_d$	vector_d	$(\vec{x},\vec{y}) \mapsto \vec{x}+\vec{y}$
mult_d	$\mathsf{vector}_d \times \mathsf{vector}_{d+1}$	vector_{d+1}	$(\vec{x},\vec{y}) \mapsto \vec{x}\circ\vec{y}$
$\{\mathsf{scalar\text{-}mult}_{c,d}\}_{c\in\mathcal{R}}$	vector_d	vector_d	$\vec{x} \mapsto c\vec{x}$
$\{\mathsf{bitflip\text{-}swap}_{t,d}\}_{t\in[0,\log\ell)}$	vector_d	vector_{d+1}	$\vec{x} \mapsto \mathfrak{S}_t(\vec{x})$
$\mathsf{level\text{-}switch}_{d,d'}$	vector_d	$\mathsf{vector}_{d'}$	$\vec{x} \mapsto \vec{x}$

Table 1: Syntax and evaluation semantics for levelled, bundled circuits. Each gate is defined for any depth parameter $d \in \{1,\dots,D\}$, and further for $\mathsf{level\text{-}switch}$, any $d' \in \{d,\dots,D\}$.

Levelled, Bundled-then-RMS. The reason for introducing yet another elaborate circuit model is that our applications inherently mix two very different kinds of computation. On the one hand, we need to evaluate predicates of the form $C(x)$, where both C and x are known to Alice; these can be handled within the bundled framework. On the other hand, once $C(x)$ is obtained, it must control a further computation that involves values secret to Bob, such as multiplying by m in the AB-LFE setting or by $\mathsf{PRF}_k(x)$ in the CPRF application. Looking ahead, our HSS construction will support exactly this combination of (levelled) bundled circuits and RM circuits.
In particular, bundled circuits provide a powerful way to apply arbitrary vector operations to vector wire types, but in our setting they come with the restriction that the corresponding wire values must be known to one party. RM circuits remove this restriction and allow computations on secret values, though at the cost of being limited to linear operations and restricted multiplications on IN/REG (scalar) wire types. To bridge these two worlds, we introduce a conversion gate that maps a vector wire type into ℓ wires of type REG, thus enabling a seamless transition from bundled computations to RM computations.

Definition 11 (Levelled, Bundled-then-RMS). *Let $\ell \in 2^{\mathbb{N}}$ be a* bundling parameter*, $D \in \mathbb{N}$ a depth bound and $M \subset \mathcal{R}$ a range bound. The* levelled, bundled-then-RMS circuit prototype *over a ring $(\mathcal{R},+,\times)$ is defined by wire types $\mathsf{types} \coloneqq \{\mathsf{vector}_d\}_{d\in\mathbb{N}} \cup \{\mathsf{REG},\mathsf{IN}\}$. Its gate types are the union of those in Definition 9 and Table 1, together with the conversion gates, for $d \in [1,D]$ and $i \in [0,\ell)$:*

$$\mathsf{vector\text{-}to\text{-}REG}_{d,i} : \mathsf{vector}_d \to \mathsf{REG}$$

This gate has the natural plaintext evaluation semantics, which maps $\vec{x} \in \mathcal{R}^\ell$ to the scalar $x_i \in \mathcal{R}$ (while also respecting $\perp$ and the bound M).

4 A Bundled Universal Circuit

In this section we introduce new techniques for constructing bundled universal circuits. Our goal is to analyze both the size and the depth of such constructions, as both are crucial for later applications.

Definition 12 (Universal Circuit). Let $\pi = (\mathsf{types},\mathsf{gates},\mathsf{in},\mathsf{out})$ and $\pi_{\mathsf{UC}} = (\mathsf{types}_{\mathsf{UC}},\mathsf{gates}_{\mathsf{UC}},\mathsf{in}_{\mathsf{UC}},\mathsf{out}_{\mathsf{UC}})$ be circuit prototypes, with evaluation semantics $(\mathsf{values},\mathsf{eval})$ and $(\mathsf{values}_{\mathsf{UC}},\mathsf{eval}_{\mathsf{UC}})$, respectively. Let $\mathcal{C}$ be a set of a π-typed circuits, all sharing the same number of input nodes n and output nodes m. Every input node must have the same type $w_0 \in \mathsf{types}$, and every output must have the same type $w_1 \in \mathsf{types}$. A *π-universal π_{UC}-circuit* for a set of π-typed circuits $\mathcal{C}$ is a π_{UC}-typed circuit UC with n_{UC} inputs of type $w_{\mathsf{UC},0}$, s_{UC} inputs of type $w_{\mathsf{UC},C}$, and m_{UC} outputs of type $w_{\mathsf{UC},1}$, together with a tuple of maps

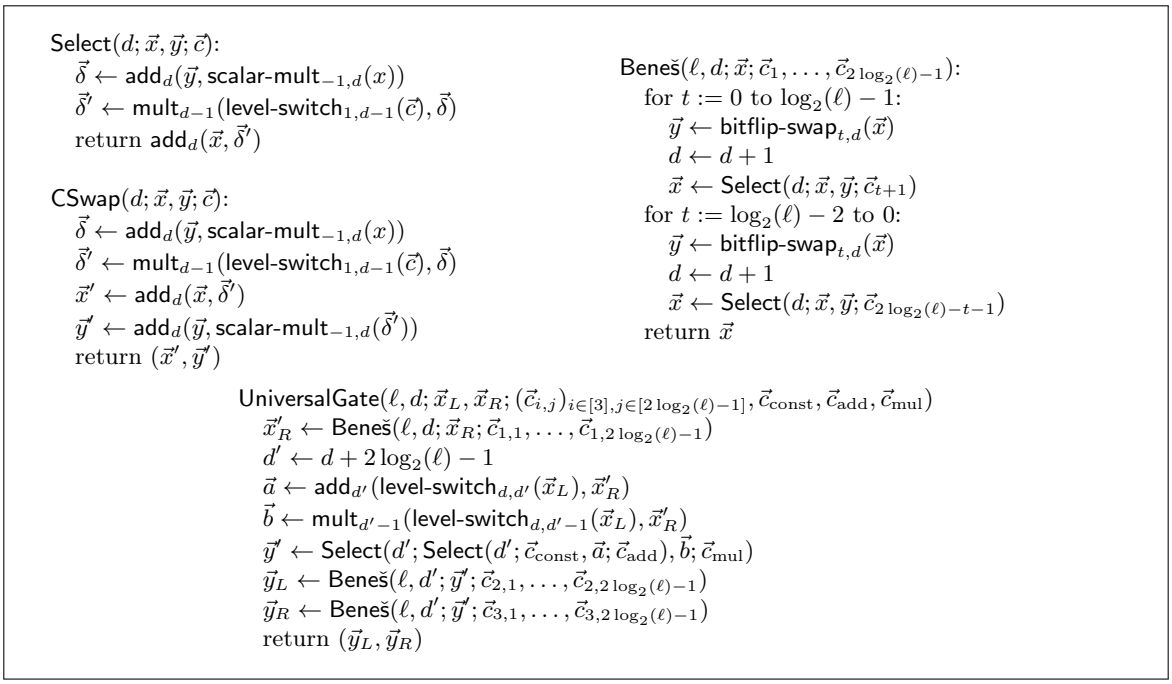

Fig. 4: Bundled universal circuit subroutines: select elements from two bundles according a control $\vec{c} \in \{0,1\}^\ell$, conditionally swap elements between two bundles according to $\vec{c} \in \{0,1\}^\ell$, apply an arbitrary permutation within a bundle using a Beneš network, and put these together until a bundled universal gate. Inputs to functions are divided into (Constants; Inputs; Controls), where Constants are fixed for a particular universal circuit, Inputs depend on the actual inputs to the circuit, and Controls depend only on the circuit to emulate. All functions expect depth d bundles as Inputs, and depth 1 bundles as Controls. Select and CSwap output bundles of the same depth d, while Beneš outputs a depth $d + 2\log_2(\ell) - 1$ bundle, and UniversalGate outputs bundles of depth $d + 4\log_2(\ell) - 2$.

- $\mathsf{ConvertInputs}\colon \mathsf{values}(w_0)^n \to \mathsf{values}_{\mathsf{UC}}(w_{\mathsf{UC},0})^{n_{\mathsf{UC}}}$ (*input pre-processing*)
- $\mathsf{ConvertCircuit}\colon \mathcal{C} \to \mathsf{values}_{\mathsf{UC}}(w_{\mathsf{UC},C})^{s_{\mathsf{UC}}}$ (*circuit pre-processing*)
- $\mathsf{ConvertOutput}\colon \mathsf{values}_{\mathsf{UC}}(w_{\mathsf{UC},1})^{m_{\mathsf{UC}}} \to \mathsf{values}(w_1)^m$ (*output post-processing*)

such that for all $C \in \mathcal{C}$ and for all $(x_1, \ldots, x_n) \in \mathsf{values}(\mathsf{type}(\mathsf{inputs}_i))$,

$$\mathsf{ConvertOutput}\left(\begin{array}{l}\mathsf{CircuitEval}(\,\mathsf{UC}, \mathsf{values}_{\mathsf{UC}}, \mathsf{eval}_{\mathsf{UC}},\\ \qquad (\mathsf{ConvertInputs}(x_1, \ldots, x_n), \mathsf{ConvertCircuit}(C))\end{array}\right) = \mathsf{CircuitEval}\,(C, \mathsf{values}, \mathsf{eval}, (x_1, \ldots, x_n))\,.$$

In the following theorem, proven in Section B, we give a detailed analysis of an optimized, bundled universal circuit.

Theorem 13 (Size of Bundled Universal Circuit). *Let ℓ be a power of two. There is an arithmetic-circuit-universal levelled ℓ-bundled circuit for the class of all arithmetic circuits with n inputs, m outputs, fan-out*[7] *2, and an ℓ-batched representation of at most S colors. The universal circuit has input and output wire types $w_{\mathsf{UC},0} = w_{\mathsf{UC},C} = \mathsf{vector}_1$ and $w_{\mathsf{UC},1} =$*

[7] For gates that are also outputs of the circuit, being an output counts towards the fan-out.

$\mathsf{vector}_{1+S(4\log_2(\ell)-2)}$. *Its input and output sizes are bounded by* $n_{\mathsf{UC}} = \lceil \frac{n}{\ell} \rceil$, $m_{\mathsf{UC}} = \lceil \frac{m}{\ell} \rceil$, *and*

$$\begin{aligned} s_{\mathsf{UC}} &\leq \frac{19}{4}(n_{\mathsf{UC}} + S + m_{\mathsf{UC}}) \log_2(n_{\mathsf{UC}} + S + m_{\mathsf{UC}}) + (m_{\mathsf{UC}} + 3S + n_{\mathsf{UC}}) 2 \log_2(\ell) \\ &\leq O((n_{\mathsf{UC}} + S + m_{\mathsf{UC}}) \log_2(n + \ell S + m)) \end{aligned}$$

and it has $O(s_{\mathsf{UC}})$ *gates.*

Unfortunately, the above cannot be used directly when doing key-switching, as the universal circuit's depth grows with the worst case circuit depth of S. Fortunately, we can get a lower depth version of the universal circuit by building a universal circuit for a larger bundle size, then splitting the bundles back up again.

Corollary 14. *Let* $\ell \leq \ell'$ *be powers of two. There is an arithmetic-circuit-universal levelled* ℓ*-bundled circuit for the class of all arithmetic circuits with* n *inputs,* m *outputs, fan-out* 2*, and a* ℓ' *batched representation of at most* S *colors. The universal circuit has input and output wire types* $w_{\mathsf{UC},0} = w_{\mathsf{UC},C} = \mathsf{vector}_1$ *and* $w_{\mathsf{UC},1} = \mathsf{vector}_{1+S(4\log_2(\ell')-2)}$*. Its input and output sizes are bounded by* $n_{\mathsf{UC}} = \lceil \frac{n}{\ell} \rceil$, $m_{\mathsf{UC}} = \lceil \frac{m}{\ell} \rceil$, *and* $s_{\mathsf{UC}} \leq O(\frac{1}{\ell}(n+\ell' S+m) \log_2(n+\ell' S+m))$*, and it has* $O(s_{\mathsf{UC}})$ *gates.*

Proof. Use Theorem 13 to get a universal ℓ'-bundled circuit. Every ℓ'-bundled wire can be represented as $\frac{\ell'}{\ell}$ ℓ-bundled wires, and similarly every ℓ'-bundled gate can be implemented with $\frac{\ell'}{\ell}$ ℓ-bundled gates. This gives the arithmetic-circuit-universal levelled ℓ-bundled circuit. □

To use the universal circuit, there is one other, main technicality we need to take care of: Theorem 13 requires all gates to have fan-out at most 2. We need to add gates to the circuit as value duplicator to make it fan-out 2, and show that they don't make the circuit much bigger or deeper. In Section B, we give a procedure to do this with only a constant factor blow-up in size, and the depth increases from D to $O(D + \log m)$, where m is the number of outputs.
We obtain the following theorem (proven in Section B).

Theorem 15. *Let* ℓ *be a power of two. There is an arithmetic-circuit-universal levelled* ℓ*-bundled circuit for the class of all arithmetic circuits with* n *inputs,* m *outputs, size* s*, and depth* D*, where* $\ell \leq 2^{\lceil \log_2(s/D) \rceil}$*. Then input wire types are* $w_{\mathsf{UC},0} = w_{\mathsf{UC},C} = \mathsf{vector}_1$*, and for some* $D_{\mathsf{UC}} = O((D + \log_2(m)) \log_2(s/D))$ *the output wire type is* $w_{\mathsf{UC},1} = \mathsf{vector}_{D_{\mathsf{UC}}}$*. Its input and output sizes are bounded by* $n_{\mathsf{UC}} = \lceil \frac{n}{\ell} \rceil$, $m_{\mathsf{UC}} = \lceil \frac{m}{\ell} \rceil$, *and* $s_{\mathsf{UC}} \leq O(\frac{1}{\ell}(n+s+\frac{s}{D} \log_2(m)+ m) \log_2(n + s + \frac{s}{D} \log_2(m) + m))$*, and it has* $O(s_{\mathsf{UC}})$ *gates.*

5 Semi-private Bundled HSS (from DCR)

In this section we introduce and instantiate our HSS framework for the class of *levelled, bundled-then-RMS* circuits (Definition 11).

5.1 Definition

The main difference between our HSS definition and that of [RS21] is the introduction of vector wire types and their combination with RMS circuits. In addition, unlike [RS21], we split out the sharing procedures to be independent of the rest of the scheme, and do not require a sharing algorithm to exist for all wire types.

Definition 16 (Gate-by-Gate Homomorphic Secret Sharing, adapted from [BGI16, RS21]). *A two-party* gate-by-gate homomorphic secret sharing (HSS) scheme *for a circuit prototypes* $(\mathsf{types}, \mathsf{gates}, \mathsf{in}, \mathsf{out})$ *with* plaintext evaluation semantics $(\mathsf{values}, \mathsf{eval})$ *is a tuple of* p.p.t. *algorithms* $\mathsf{HSS} = (\mathsf{crsGen}, \mathsf{KeyGen}, \mathsf{HGateEval}, \mathsf{Dec})$ *with the following syntax and properties:*

- $\mathsf{HSS.crsGen}(1^\lambda) \xrightarrow{\$} \mathsf{crs}$. *On input the security parameter, samples a common reference string* crs.

- $\mathsf{HSS.KeyGen}(1^\lambda, \mathsf{crs}) \xrightarrow{\$} (\mathsf{pp}, \mathsf{sk}, \mathsf{ek}_0, \mathsf{ek}_1)$. *On input the security parameter and common reference string, outputs public parameters* pp*, a secret key* sk*, and a pair of evaluation keys* $(\mathsf{ek}_0, \mathsf{ek}_1)$.

  We require pp *and* ek_σ *together to define a space* $\mathsf{values}_\sigma(w)$ *of shares for each wire type* $w \in \mathsf{types}$ *and* $\sigma \in \{0, 1\}$.

  We also require **weak transparency:** *given the* crs *and the full output* $(\mathsf{pp}, \mathsf{sk}, (\mathsf{ek}_0, \mathsf{ek}_1))$ *of* KeyGen*, it must be possible to reverse sample* KeyGen*'s randomness.*[8]

- $\mathsf{HSS.HGateEval}(\sigma, \mathsf{pp}, \mathsf{ek}_\sigma, g, (x_j^{(\sigma)})_{j\in[k]}; r) \to y^{(\sigma)}$. *On input a party index* σ*, public parameters* pp*, an evaluation key* ek_σ*, a gate type* g *of arity* k*, a* k*-tuple of shares* $(x_1^{(\sigma)}, \ldots, x_k^{(\sigma)})$*, the homomorphic gate evaluation algorithm* HEval *outputs a share* $y^{(\sigma)}$*. We explicitly write the random tape* r *for this algorithm, as the same tape must be provided by both parties for correctness.*

  HGateEval *(together with* values_σ*) must define a* homomorphic evaluation semantics*, i.e., if* $(w_1, \ldots, w_k) = \mathsf{in}(g)$ *and* $x_i^{(\sigma)} \in \mathsf{values}_\sigma(w_i)$ *for* $i \in [k]$*, then* $y^{(\sigma)} \in \mathsf{values}_\sigma(\mathsf{out}(g))$.

- $\mathsf{HSS.Dec}(\mathsf{pp}, \mathsf{sk}, w, y^{(0)}, y^{(1)}) \to y$. *On input public parameters, the secret key, a wire type* $w \in \mathsf{types}$*, and shares* $y^{(0)} \in \mathsf{values}_0(w)$ *and* $y^{(1)} \in \mathsf{values}_1(w)$*, the decoding algorithm outputs a value* $y \in \mathsf{values}(w)$.

For convenience, we define the full homomorphic evaluation algorithm HSS.HCircuitEval *to run* HSS.HGateEval *for each gate of a circuit:*

$$\begin{aligned}&\mathsf{HSS.HCircuitEval}(\sigma, \mathsf{pp}, \mathsf{ek}_\sigma, C, x_1^{(\sigma)}, \ldots, x_n^{(\sigma)}; r)\\ &= \mathsf{CircuitEval}(C, \mathsf{values}_\sigma, \mathsf{HSS.HGateEval}(\sigma, \mathsf{pp}, \mathsf{ek}_\sigma, \cdot, \cdot), x_1^{(\sigma)}, \ldots, x_n^{(\sigma)}; r).\end{aligned}$$

Gate-by-gate correctness. For each gate type $g \in \mathsf{gates}$, we require a correctness property, which says that applying $\mathsf{HSS.Dec}(\mathsf{pp}, \mathsf{sk}, \mathsf{in}(g), \cdot, \cdot)$ to a set of input shares, and then applying the gate function $\mathsf{eval}(g)$ to the result, gives the same answer as running HSS.HGateEval on the input shares, followed by HSS.Dec on the result. Note that this gate-by-gate correctness property naturally composes, that is, it suffices to ensure correctness of the circuit evaluation algorithm HSS.HCircuitEval on any circuit [RS21]. We give further details and the formal correctness definition in Appendix C.2.

Sharing inputs. Our HSS construction supports three ways of creating shares of input values:

- $\mathsf{Share}_{\mathsf{priv},\mathsf{IN}}(1^\lambda, \mathsf{pp}, x) \to (x^{(0)}, x^{(1)})$: *Public-key sharing* of a private input, which outputs a pair of shares for an IN wire type.
- $\mathsf{Share}_{\mathsf{sinct},\sigma,\mathsf{vector}_1}(\mathsf{pp}, \mathsf{ek}_0, d)/\mathsf{Share}_{\mathsf{sinct},1,\mathsf{vector}_1}(\mathsf{pp}, \mathsf{ek}_1, \vec{x}, \psi)$: *Succinct, public key sharing* of a *semi-private* vector of inputs. These algorithms are used in conjunction with

[8] This can easily be satisfied by including KeyGen's full randomness in sk.

$\mathsf{HSS.Hash}(\mathsf{crs}, \vec{x}) \to (d, \psi)$, which uses succinct HSS (Section 3.5) to compress $\vec{x}$ into a digest d and secret state ψ. The two Share algorithms output *semi-private* shares, where party 1's share contains $\vec{x}$.

– $\mathsf{Share}_{\mathsf{s\text{-}priv},0,\mathsf{vector}_1}(\mathsf{pp})$ / $\mathsf{Share}_{\mathsf{s\text{-}priv},1,\mathsf{vector}_1}(\mathsf{pp}, \mathsf{sk}, \vec{x}, \vec{x}^{(0)})$: *Private-key sharing* of a *semi-private* vector of inputs. Here, one "offline" share, $\vec{x}^{(0)}$, is generated ahead of time, and the other share $x^{(1)}$ is derived based on $\vec{x}^{(0)}$ and the secret key, sk, and contains $\vec{x}$.

For each sharing algorithm, we formalise the relevant correctness and privacy properties in Appendix C.3. Essentially, correctness requires that decoding the shares using Dec gives the correct result, while privacy requires that each share hides the original input.
By default, the $\mathsf{HSS.Dec}$ algorithms require the secret key in order to decode an output. We also consider a public-key variant of decoding, which requires that shares of a particular wire type can be locally converted into additive shares without knowledge of the secret key. In our construction, additive decoding is trivial, since each share always contains an additive share of the message.

Definition 17 (Additive Decoding). HSS *has* additive decoding *for wire type* w *if* $\mathsf{values}(w)$ *is an abelian group (written additively) and there are PPT algorithms* $\mathsf{HSS.Dec}_w^{(0)}, \mathsf{HSS.Dec}_w^{(1)}$ *such that, for every PPT adversary* $\mathcal{A}$*,*

$$\Pr\left[\begin{array}{c} y \neq y^{(1)} - y^{(0)} \\ \wedge \\ y \neq \bot \end{array} \;\middle|\; \begin{array}{r} \mathsf{crs} \xleftarrow{\$} \mathsf{HSS.crsGen}(1^\lambda) \\ (\mathsf{pp}, \mathsf{sk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda, \mathsf{crs}) \\ (x^{(0)}, x^{(1)}) \xleftarrow{\$} \mathcal{A}(1^\lambda, \mathsf{crs}, \mathsf{pp}, \mathsf{sk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \\ y \leftarrow \mathsf{HSS.Dec}(\mathsf{pp}, \mathsf{sk}, w, x^{(0)}, x^{(1)}) \\ y^{(0)} \xleftarrow{\$} \mathsf{HSS.Dec}_w^{(0)}(\mathsf{pp}, \mathsf{ek}_0, x^{(0)}) \\ y^{(1)} \xleftarrow{\$} \mathsf{HSS.Dec}_w^{(1)}(\mathsf{pp}, \mathsf{ek}_1, x^{(1)}) \end{array}\right] \leq \mathsf{negl}(\lambda).$$

5.2 Construction from DCR

We now present our main construction, which achieves HSS for B-bounded, bundled-then-RMS circuits under the DCR assumption. In this section, we use the following vector/matrix notation.

– $\boldsymbol{g}$: group element (in $\mathbb{Z}_{N^\zeta}$).
– $\vec{\boldsymbol{g}}$: vector of group elements (column vector by default).
– $\boldsymbol{G}$: matrix of group elements.
– $(\vec{\boldsymbol{g}}^\mathsf{T})^{\vec{x}} = \prod_i \boldsymbol{g}_i^{x_i}$: dot product in exponent.
– $\boldsymbol{G}^{\vec{x}} = \left[(\vec{\boldsymbol{g}}_1)^{\vec{x}} \cdots (\vec{\boldsymbol{g}}_\ell)^{\vec{x}}\right]^\top$: matrix-vector product in exponent ($\vec{\boldsymbol{g}}_i$ is row i).
– ${}^{\vec{x}^\mathsf{T}}\vec{\boldsymbol{g}}$: dot product in exponent, with the exponent on the left. Defined for consistency with:
– ${}^{\vec{x}^\mathsf{T}}\boldsymbol{G}^{\vec{y}}$: Use $\boldsymbol{G}$ to define a bilinear operation on $\vec{x}$ and $\vec{y}$. Exponent version of $\vec{x}^\mathsf{T} G \vec{y}$.

In this section we also make limited use of colour purely as a visual aid. All constructions and notation remain formally well-defined without colour, and the scheme can be fully understood in black-and-white. Specifically, orange values such as K are associated with RMS programs, blue values $\boldsymbol{g}_{t,d}$ with vector-swapping ciphertexts, and red values $\boldsymbol{g}_{d \to d'}$ with level-switching ciphertexts.

The Key Switching Graph. In our bundled circuit evaluation, the input wires are of type vector_1, corresponding to level-1 shares, and progressively increase to types of vector_d, where $d \leq D$ for some maximum depth bound D. To help shift wires from one level to the next, we

use key-switching gadgets generated in the setup. Define $G_{\mathsf{KS}} = ([D], E_{\mathsf{KS}})$ to be a directed acyclic graph called the "key-switching graph" (it contains an edge from d to d' iff there is a key-switching gadget to switch a value from level d to level d'). It should satisfy the following connectivity condition: for every pair $(d, d') \in [D]^2$ such that $d < d'$, there should exist a path from d to d' in G_{KS}. In Section D, we present a construction with a linear number of edges such that all shortest paths are at most logarithmic.

Setup and Key Generation. Our HSS setup algorithms, shown in Fig. 5, start by generating a CRS for succinct, bilinear HSS. Then, we sample a fresh modulus N for the group $\mathbb{Z}_{N^\zeta}$, and keys $(\vec{k}_1, \ldots, \vec{k}_d)$ for each level of the bundled circuit, as well as a key $\vec{k}$ used for succinct input sharing.

We then generate three forms of evaluation key material. For vector multiplication gates, we sample helper values $(\vec{\boldsymbol{h}}_d, \boldsymbol{H}_d)$ for each level d, using the idea described in Section 2, except we assume the inputs to any multiplication gate are shared at levels d and $d+1$, and obtain an output at level $d+1$. For bit-flip swap gates ($\mathcal{R}_t$), we use a variant of this setup, specialised for the simpler task of permuting the input vector, where now the level-d input is always mapped to level-$d+1$. We also give out general, key-switching gadgets, used to translate level-d vectors into level $d' > d$.

Finally, we also sample shares of a random key K, used as a Damgård-Jurik-ElGamal secret key for the RM circuit part of HSS evaluation.

HSS Setup for B-bounded, Bundled-then-RMS Circuits

$\mathsf{HSS.crsGen}(1^\lambda)$: return $\mathsf{crs} \xleftarrow{\$} \mathsf{biHSS.crsGen}(1^\lambda, 1^\ell)$

$\mathsf{HSS.KeyGen}(1^\lambda, \mathsf{crs})$:

1. $(N, p, q) \xleftarrow{\$} \mathsf{RSA.Gen}(1^\lambda)$
2. $\zeta \xleftarrow{\$} \min\{\zeta \in \mathbb{Z} \mid N^{\zeta-1} > 2^\lambda \ell B\}$
3. $\boldsymbol{g} \xleftarrow{\$} (\mathbb{Z}/N^{\zeta+1}\mathbb{Z})^\times$

4. For $d \in [D]$:
   $\vec{k}_d \xleftarrow{\$} [0, N)^\ell$ // Key used to authenticate bundles at depth d

5. $\vec{k}^{(0)} \xleftarrow{\$} [0, N2^\lambda)$ // Key for succinct HSS input sharing
6. $\vec{k}^{(1)} \leftarrow \vec{k}^{(0)} + \vec{k}_1$
7. $(E, \phi) \xleftarrow{\$} \mathsf{biHSS.Enc}(\mathsf{crs}, \vec{k}^{(0)})$

8. For $d \in [D]$: // Setup key-switching ciphertexts for multiplication gates
   $\vec{\boldsymbol{h}}_d \leftarrow \boldsymbol{g}^{-\vec{k}_d}$
9. For $d \in [D-1]$:
   $\boldsymbol{H}_d \leftarrow \boldsymbol{g}^{\vec{k}_d \vec{k}_{d+1}^\top} \exp(\mathsf{diag}(\vec{k}_{d+1}))$

   // Setup key-switching ciphertexts for bitflip-swap gates
10. For $d \in [D]$ and $t \in [0, \log \ell)$:

$$r_{t,d} \xleftarrow{\$} [0, N), \quad \boldsymbol{g}_{t,d} \leftarrow \boldsymbol{g}^{r_{t,d}}, \quad \vec{\boldsymbol{h}}_{t,d} \leftarrow \boldsymbol{g}_{t,d}^{-\vec{k}_d^\top} \exp(\mathsf{diag}(\mathcal{R}_t(\vec{k}_{d+1})))$$

11. For $(d, d') \in E_{\mathsf{KS}}$: // Setup key-switching ciphertexts for level-switching gates

$$r_{d\to d'} \xleftarrow{\$} [0, N), \quad \boldsymbol{g}_{d\to d'} \leftarrow \boldsymbol{g}^{r_{d\to d'}}, \quad \vec{\boldsymbol{h}}_{d\to d'} \leftarrow \boldsymbol{g}_{d\to d'}^{-\vec{k}_d^\top} \exp(\mathsf{diag}(\vec{k}_{d'}))$$

12. Set $K \leftarrow \vec{k}_D[0]$ // Setup key shares for RM circuit
13. Sample $K^{(0)} \xleftarrow{\$} [0, 2^\lambda N)$ // $\langle K \rangle_0$
14. Set $K^{(1)} \leftarrow K + K^{(0)}$ // $\langle K \rangle_1$

15. $\mathsf{pp} \leftarrow (N, \boldsymbol{g}, \mathsf{crs}, E, (\vec{\boldsymbol{h}}_d, \boldsymbol{H}_d)_d, (\boldsymbol{g}_{t,d}, \vec{\boldsymbol{h}}_{t,d})_{t,d}, (\boldsymbol{g}_{d\to d'}, \vec{\boldsymbol{h}}_{d\to d'})_{(d,d')})$
16. $(\mathsf{ek}_0, \mathsf{ek}_1) \leftarrow ((\vec{k}^{(0)}, K^{(0)}, \phi), (\vec{k}^{(1)}, K^{(1)}))$
17. $\mathsf{sk} \leftarrow ((k_d)_{d\in[D]}, K, (r_{t,d})_{t\in[0,\log \ell)],d\in[D]}, (r_{d\to d'})_{(d,d')\in E_{\mathsf{KS}}}, p, q)$
18. Output $(\mathsf{pp}, \mathsf{sk}, \mathsf{ek}_0, \mathsf{ek}_1)$

Fig. 5: HSS CRS and key generation algorithms

RM Circuit evaluation. We briefly recap the paradigm of HSS for restricted multiplication circuits (also called RMS programs), instantiated using Damgård-Jurik-ElGamal encryption and the DCR assumption as in [RS21]. Intermediate values in the computation are represented by REG wire types, and consist of subtractive sharings $\langle K\, x\rangle, \langle x\rangle$. Input wires (of type IN) contain a pair of Damgård-Jurik-ElGamal ciphertexts encrypting x and $K\, x$, given to both parties. Multiplication can only be performed between a REG type and an IN type, and is done using the DDLog algorithm.
These algorithms, together with the decoding algorithms for REG and IN wires, are shown in Fig. 11.

Bundled circuit evaluation. Each bundled type vector_d is represented as a subtractive sharing $\langle \vec{k}_d^\top \vec{x}\rangle$. Linear and constant gates are straightforward, and deferred to Section C.4, along with a vector-to-REG conversion gate. In Fig. 6, we present the gates for vector multiplication, $\mathsf{bitflip\text{-}swap}_{t,d}$ and key-switching (called level-switch).
Correctness of mult_d follows from the same argument presented in Section 2, the only difference being that we now have two keys $\vec{k}_d, \vec{k}_{d+1}$, instead of just one key k. For the swap gate, observe that, for shares $\overline{x}^{(0)}, \overline{x}^{(1)}$ input to $\mathsf{bitflip\text{-}swap}_{t,d}$, we have:

$$\left(\boldsymbol{g}_{t,d}^{\overline{x}^{(1)}} \left(\vec{\boldsymbol{h}}_{t,d}^{\mathsf{T}}\right)^{\vec{x}}\right) / \left(\boldsymbol{g}_{t,d}^{\overline{x}^{(0)}}\right) = \boldsymbol{g}_{t,d}^{\overline{x}^{(1)}-\overline{x}^{(0)}} \boldsymbol{g}_{t,d}^{-\vec{k}_d^\mathsf{T}\vec{x}} \exp(\mathsf{diag}(\boldsymbol{\circlearrowright}_t(\vec{k}_{d+1})))^{\vec{x}} = \exp\left(\vec{k}_{d+1}^{\mathsf{T}} \cdot \boldsymbol{\circlearrowright}_t(\vec{x})\right)$$

By Lemma 2, applying DDLog to the top and bottom leads to subtractive shares of $\vec{k}_{d+1}^\top \cdot \boldsymbol{\circlearrowright}_t(\vec{x})$ mod N^ζ. Furthermore, applying Lemma 3, these shares are correct shares *over the integers* with high probability, provided the payload does not exceed the magnitude bound B.
Correctness of level-switch follows by the same argument, repeated for each i where the switch operation is applied. We defer full proofs of gate-by-gate correctness to Section C.4.

Bundled HSS: Gate Evaluation and Wire Decoding

mult_d gate:

$$\mathsf{eval}_0 : (\overline{x}_1^{(0)}, \overline{x}_2^{(0)}) \mapsto \mathsf{DDLog}\left(\boldsymbol{g}^{\overline{x}_1^{(0)}\overline{x}_2^{(0)}}\right) + r \mod N^\zeta$$

$$\mathsf{eval}_1 : (\overline{x}_1^{(1)}, \vec{x}_1), (\overline{x}_2^{(1)}, \vec{x}_2) \mapsto \Big(\mathsf{DDLog}\Big(\boldsymbol{g}^{\overline{x}_1^{(1)}\overline{x}_2^{(1)}} \vec{x}_1^{\mathsf{T}} \boldsymbol{H}_d^{\vec{x}_2} \left(\vec{\boldsymbol{h}}_d^{\mathsf{T}}\right)^{\overline{x}_2^{(1)}\vec{x}_1} \left(\vec{\boldsymbol{h}}_{d+1}^{\mathsf{T}}\right)^{\overline{x}_1^{(1)}\vec{x}_2}\Big) + r \mod N^\zeta,\ \vec{x}_1 \circ \vec{x}_2\Big)$$

$\mathsf{bitflip\text{-}swap}_{t,d}$ gate:

$\mathsf{eval}_0 : \bar{x}^{(0)} \quad \mapsto \mathsf{DDLog}\left(\boldsymbol{g}_{t,d}^{\bar{x}^{(0)}}\right) + r \mod N^\zeta$

$\mathsf{eval}_1 : (\bar{x}^{(1)}, \vec{x}) \mapsto \left(\mathsf{DDLog}\left(\boldsymbol{g}_{t,d}^{\bar{x}^{(1)}}\left(\vec{\boldsymbol{h}}_{t,d}^{\mathsf{T}}\right)^{\vec{x}}\right) + r \mod N^\zeta,\ \boldsymbol{\circlearrowright}_t(\vec{x})\right)$

$\mathsf{level\text{-}switch}_{d,d'}$: Let $(d_0, \ldots d_m)$ be the shortest path from d to d', with $d_0 = d$ and $d_m = d'$. For $i \in [1..m)$, define:

$$\bar{x}_{i+1}^{(0)} = \mathsf{DDLog}\left(\boldsymbol{g}_{d_i \to d_{i+1}}^{\bar{x}_i^{(0)}}\right)$$

$$\bar{x}_{i+1}^{(1)} = \mathsf{DDLog}\left(\boldsymbol{g}_{d_i \to d_{i+1}}^{\bar{x}_i^{(1)}}\left(\vec{\boldsymbol{h}}_{d_i \to d_{i+1}}^{\mathsf{T}}\right)^{\vec{x}}\right)$$

$\mathsf{eval}_0 : \bar{x}^{(0)} \mapsto \bar{x}_m^{(0)} + r \bmod N^\zeta$

$\mathsf{eval}_1 : \bar{x}^{(1)} \mapsto (\bar{x}_m^{(1)} + r \bmod N^\zeta, \vec{x})$

$\mathsf{HSS.Dec}(\mathsf{pp}, \mathsf{sk}, \mathsf{vector}_d, \bar{y}^{(0)}, (\bar{y}^{(1)}, \vec{y}))$:

- **assert** $k_d^{\mathsf{T}} \vec{y} = \bar{y}^{(1)} - \bar{y}^{(0)}$
- return $\vec{y}$ // or $\perp$ if assertion fails

Fig. 6: Bundled HSS gate evaluation

Input sharing algorithms. In Fig. 7, we detail the three types of input sharing algorithms used in our HSS scheme. The public-key, IN-sharing algorithm is the same as in standard HSS for RM circuits. We use $\mathsf{HSS.Hash}$ to apply succinct, bilinear HSS to hash a vector input. This is used together with the $\mathsf{HSS.Share}_{\mathsf{sinct},\sigma}$ algorithms to output level-1 bundled shares of $\vec{x}$, that is, subtractive shares of $\vec{k}_1^\top \vec{x}$, where $\vec{k}_1$ is the level-1 secret key. Finally, the $\mathsf{HSS.Share}_{\mathsf{s\text{-}priv},\sigma}$ algorithms create semi-private shares of $\vec{x}$, by sampling them in the natural way using the level-1 secret key.

HSS Input Sharing Algorithms

$\mathsf{HSS.Share}_{\mathsf{priv,IN}}(\mathsf{pp}, x)$

1. Parse $\mathsf{pp} = (N, \boldsymbol{g}, E, (\vec{\boldsymbol{h}}_d, \boldsymbol{H}_d)_{d\in[D]}, \ldots)$
2. $\boldsymbol{h} \leftarrow \vec{\boldsymbol{h}}_D[0]$
3. $c^{\mathsf{auth}} \xleftarrow{\$} \mathsf{DJE.Enc}\mathsf{K}_{(\boldsymbol{g},\boldsymbol{h},N)}(x)$ // $\mathsf{Enc}(K\, x)$
4. $c \xleftarrow{\$} \mathsf{DJE.Enc}_{(\boldsymbol{g},\boldsymbol{h},N)}(x)$ // $\mathsf{Enc}(x)$
5. Output $((c^{\mathsf{auth}}, c), (c^{\mathsf{auth}}, c))$

$\mathsf{HSS.Hash}_{\mathsf{vector}_1}(\mathsf{crs}, x)$

1. $(d, \psi) \leftarrow \mathsf{biHSS.Hash}(\mathsf{crs}, \vec{x})$
2. Output (d, ψ)

$\mathsf{HSS.Share}_{\mathsf{sinct},0,\mathsf{vector}_1}(\mathsf{pp}, \mathsf{ek}_0, d)$

1. Parse $\mathsf{pp} = (N, \boldsymbol{g}, \mathsf{crs}, E, \ldots)$
2. Parse $\mathsf{ek}_0 = (\vec{k}^{(0)}, K^{(0)}, \phi)$
3. $\bar{x}^{(0)} \xleftarrow{\$} \mathsf{biHSS.EncEval}(\mathsf{crs}, d, \phi)$ // $\langle \vec{k}_1^{\mathsf{T}} \vec{x} \rangle_0 = \langle (-\vec{k}^{(0)})^{\mathsf{T}} \vec{x} \rangle_0$
4. Output $\bar{x}^{(0)}$

$\mathsf{HSS.Share}_{\mathsf{sinct},1,\mathsf{vector}_1}(\mathsf{pp}, \mathsf{ek}_1, x, \psi)$

1. Parse $\mathsf{pp} = (N, \boldsymbol{g}, \mathsf{crs}, E, \ldots)$

2. Parse $\mathsf{ek}_1 = (\vec{k}^{(1)}, K^{(1)})$
3. $\bar{x}_0^{(1)} \xleftarrow{\$} \mathsf{biHSS.EncEval}(\mathsf{crs}, d, \phi)$ // $\langle(-\vec{k}^{(1)})^\mathsf{T}\vec{x}\rangle_1$
4. $\bar{x}^{(1)} \leftarrow (\vec{k}^{(1)})^\mathsf{T}\vec{x} + x_0^{(1)}$ // $\langle\vec{k}_1^\mathsf{T}\vec{x}\rangle_1 = (\vec{k}^{(1)})^\mathsf{T}\vec{x} + \langle(-\vec{k}^{(0)})^\mathsf{T}\vec{x}\rangle_1$
5. Output $(\bar{x}^{(1)}, \vec{x})$

$\mathsf{HSS.Share}_{\mathsf{s\text{-}priv},0,\mathsf{vector}_1}(\mathsf{pp}, \mathsf{sk})$
1. Parse $\mathsf{pp} = (N, \ldots)$
2. $x^{(0)} \xleftarrow{\$} [0, 2^\lambda B\ell N)$ // $\langle\vec{k}_1^\mathsf{T}\vec{x}\rangle_0$; note that $|\vec{k}_1^\mathsf{T}x| < B\ell N$
3. $\mathsf{st} \leftarrow x^{(0)}$
4. Output $(x^{(0)}, \mathsf{st})$

$\mathsf{HSS.Share}_{\mathsf{s\text{-}priv},1,\mathsf{vector}_1}(\mathsf{pp}, \mathsf{sk}, x, \mathsf{st})$
1. Parse $\mathsf{sk} = ((k_d)_{d\in[D]}, \ldots)$
2. $x^{(1)} \leftarrow \vec{k}_1^\mathsf{T}\vec{x} + \mathsf{st}$ // $\langle\vec{k}_1^\mathsf{T}\vec{x}\rangle_1$
3. Output $x^{(1)}$

Fig. 7: HSS input sharing algorithms

Theorem 18 (HSS scheme for levelled, bundled-then-RMS circuits from DCR). *Let $B = \mathcal{O}(2^{\mathsf{poly}(\lambda)})$, and let* biHSS *be a $\mathbb{Z}$-bilinear $2^\lambda\ell B$-bounded HSS scheme. Then the HSS construction defined by* (HSS.crsGen, HSS.KeyGen) *(Fig. 5),* (HSS.HGateEval, HSS.Dec) *(Figs. 6 and 11 to 13) is a correct HSS scheme for B-bundled levelled, bundled-then-RMS circuits (Definition 11). When instantiated with the key-switching graph in Section D and a computationally succinct* biHSS *where* biHSS.Enc *runs in $O(\mathsf{poly}(\lambda, \log B)\cdot\ell^2$ time, the computation of* HSS.KeyGen *and the size of* pp *are $O(\mathsf{poly}(\lambda, \log|B|)\ell^2 D)$, and the runtimes of* $\mathsf{HSS.sinct}, 0, \mathsf{vector}_1$, $\mathsf{HSS.Share}_{\mathsf{s\text{-}priv},0,\mathsf{vector}_1}$, $\mathsf{HSS.Share}_{\mathsf{priv}}$, *and* $\mathsf{HSS.HGateEval}(0,\ldots)$ *and the output size of* $\mathsf{HSS.Hash}_{\mathsf{vector}_1}$ *are all* $\mathsf{poly}(\lambda, \log|B|)$ *(independent of ℓ).*
Furthermore, assuming DCR and the strong encoder and hasher privacy properties of biHSS*, the sharing algorithms in Fig. 7 satisfy the correctness and privacy properties in Definitions 33 to 35.*

The proof is given in Section E.4. Most of the properties are straightforward, following the individual correctness properties of the gates. The only real challenge is proving privacy of the encrypted inputs from $\mathsf{HSS.Share}_{\mathsf{priv},\mathsf{IN}}$. This requires a careful hybrid argument, relying on the security of succinct HSS, the DJE encryption scheme, and the DCR assumption.

6 Attributed-based laconic function evaluation from semi-private succinct HSS

We are now ready to present our main contribution: a construction of function-private AB-LFE from HSS. At a high level, the protocol is quite simple once the semi-private, bundled HSS framework from the previous section is in place: The CRS generation and compression algorithms of AB-LFE are essentially just the corresponding HSS procedures, combined with the appropriate conversion of inputs and circuits into the bundled format. The crucial step is that $\mathcal{P}_B$ can obtain a share $\widetilde{C}_0$ of the circuit relative to the digest h *succinctly* and without knowing C, thanks to the succinct on-the-fly generation of shares of our HSS. With this in hand, $\mathcal{P}_B$ homomorphically evaluates his share of $C(x)\cdot\mu$ using the universal-circuit construction, again in a succinct manner. Importantly, the use of a universal circuit also ensures *function privacy* Finally, $\mathcal{P}_B$ sends to $\mathcal{P}_A$ his output share of $C(x)\cdot\mu$, together with

$\mathcal{P}_A$'s share of μ and the public input x. Now $\mathcal{P}_A$ can perform the matching HSS evaluation to reconstruct the final value $z = C(x) \cdot \mu$.

Function-Private AB-LFE

// Supports $2^{\mathsf{poly}(\lambda)}$-bounded arithmetic circuits.

Requires:

- Let $B = \mathcal{O}(2^{\mathsf{poly}(\lambda)})$
- UC is an B-bounded ℓ-levelled bundled circuit which is universal (Definition 12) for the class of arithmetic circuits (Definition 27); it is associated with the conversion functions $\mathsf{ConvertInputs}$, $\mathsf{ConvertCircuit}$, and $\mathsf{ConvertOutput}$.
- $\mathsf{UC\text{-}then\text{-}mult}_{\tilde{x}}$ is the B-bounded bundled-then-RMS circuit obtained by
  1. taking a copy of UC (whose gates and wires can naturally be mapped from levelled bundled syntax to bundled-then-RMS syntax), replacing the first n_{UC} input gates with the constant gates $\mathsf{ConvertInputs}(x) = (\tilde{x}_1, \tilde{x}_2, \dots)$
  2. adding a $\mathsf{vector\text{-}to\text{-}REG}$ gate to convert the output of UC to a register (REG)
  3. adding a multiplication gate between this register and a new input wire typed IN

  Informally, it takes as input an appropriately formatted boolean circuit C and a value μ, then computes $C(x) \cdot \mu$ (if the hardcoded value is $\tilde{x} = \mathsf{ConvertInputs}(x)$).
- $\mathsf{HSS} = (\mathsf{crsGen}, \mathsf{KeyGen}, \mathsf{HGateEval}, \mathsf{Dec})$ is a two-party gate-by-gate HSS scheme (Definition 16) for B-bounded bundled-then-RMS circuits (Definition 11). Let R be (a tight enough bound on) the runtime of $\mathsf{HSS.HCircuitEval}$ (as defined at the end Definition 16)
- The wire type vector_1 should support a succinct semi-private sharing (Definition 34) triple $(\mathsf{HSS.Hash}, \mathsf{Share}_{\mathsf{sinct},0}, \mathsf{Share}_{\mathsf{sinct},1})$
- The wire type IN should support a public-key private sharing (Definition 33) algorithm $\mathsf{HSS.Share}_{\mathsf{priv}}$.
- The wire type REG should support an additive decoding pair (Definition 17) $(\mathsf{HSS.Dec}^{(0)}, \mathsf{HSS.Dec}^{(1)})$.

$\mathsf{crsGen}(1^\lambda, \mathsf{params})$:

1. $r \xleftarrow{\$} \{0,1\}^R$ // R being a bound on the runtime of $\mathsf{HSS.HCircuitEval}$, it is still smaller than the size of the constraint
2. $\mathsf{crs}_{\mathsf{HSS}} \xleftarrow{\$} \mathsf{HSS.crsGen}(1^\lambda)$
3. Output $\mathsf{crs} = (\mathsf{crs}_{\mathsf{HSS}}, r)$

$\mathsf{Compress}(\mathsf{crs}, C)$:

1. $\widetilde{C} \leftarrow \mathsf{ConvertCircuit}(C)$
2. $(h, \psi) \xleftarrow{\$} \mathsf{HSS.Hash}(\widetilde{C})$
3. Output (h, ψ)

$\mathsf{Enc}(\mathsf{crs}, h, x, \mu)$:

1. Parse $\mathsf{crs} = (\mathsf{crs}_{\mathsf{HSS}}, r)$
2. $\tilde{x} \leftarrow \mathsf{ConvertInputs}(x)$
3. $(\mathsf{pp}, \mathsf{sk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \xleftarrow{\$} \mathsf{HSS.KeyGen}(1^\lambda, \mathsf{crs}_{\mathsf{HSS}})$
4. $\widetilde{C}_0 \xleftarrow{\$} \mathsf{Share}_{\mathsf{sinct},0}(\mathsf{pp}, \mathsf{ek}_0, h)$
5. $(\mu_0, \mu_1) \xleftarrow{\$} \mathsf{HSS.Share}_{\mathsf{priv}}(\mathsf{pp}, \mu)$

6. $y_0 \xleftarrow{\$} \mathsf{HSS.HCircuitEval}(0, \mathsf{pp}, \mathsf{ek}_0, \mathsf{UC\text{-}then\text{-}mult}_{\tilde{x}}, (\widetilde{C}_0, \mu_0); r)$
7. $z_0 \xleftarrow{\$} \mathsf{HSS.Dec}^{(0)}(\mathsf{pp}, \mathsf{ek}_0, y_0)$
8. Output $\mathsf{ct} = (\mathsf{pp}, \mathsf{ek}_1, d, \psi, \mu_1, z_0)$

$\mathsf{Dec}(\mathsf{crs}, C, x, \mathsf{ct}, \psi)$:

1. Parse $\mathsf{crs} = (\mathsf{crs}_{\mathsf{HSS}}, r)$
2. Parse $\mathsf{ct} = (\mathsf{pp}, \mathsf{ek}_1, h, \psi, \mu_1, y_0)$
3. $\widetilde{C} \leftarrow \mathsf{ConvertCircuit}(C)$
4. $\widetilde{C}_1 \xleftarrow{\$} \mathsf{HSS.Share}_{\mathsf{sinct},1}(\mathsf{pp}, \mathsf{ek}_1, \widetilde{C}, \psi)$
5. $y_1 \xleftarrow{\$} \mathsf{HSS.HCircuitEval}(1, \mathsf{pp}, \mathsf{ek}_1, \mathsf{UC\text{-}then\text{-}mult}_{\tilde{x}}, (\widetilde{C}_1, \mu_1); r)$
6. $z_1 \xleftarrow{\$} \mathsf{HSS.Dec}^{(1)}(\mathsf{pp}, \mathsf{ek}_1, y_1)$
7. Compute $z \leftarrow z_1 - z_0$
8. Output z

Fig. 8: Function-Private Attribute-Based Laconic Function Evaluation from HSS.

Theorem 19 (Function-Private AB-LFE from HSS). *Let $B = \mathcal{O}(2^{\mathsf{poly}(\lambda)})$. The construction of Fig. 8 is a function-private attribute-based laconic function evaluation scheme for the class of B-bounded arithmetic circuits.*

Correctness, security, and function privacy all reduce to the various correctness and security properties of the HSS scheme. We refer to Section H for the proof of Theorem 19.

Corollary 20 (Function-Private AB-LFE from DCR). *Let $B = \mathcal{O}(2^{\mathsf{poly}(\lambda)})$. Assuming DCR, there exists a function-private attribute-based laconic function evaluation scheme for the class of B-bounded arithetic circuits achieving circuit succinctness $\tilde{O}(|x| + |C|^{2/3}\mathsf{depth}(C)^{1/3})$ and ciphertext size $\tilde{O}(|C|^{2/3}\mathsf{depth}(C)^{1/3})$ (ignoring $\mathsf{poly}(\lambda, \log B)$ factors). Therefore, it is attribute-succinct as well when the attribute length is at least $\tilde{\omega}(|C|^{2/3}\mathsf{depth}(C)^{1/3})$.*

Proof. The construction of Fig. 10 and its requirements can be instantiated under DCR by combining the universal circuit of Theorem 15 with bundling parameter $\ell := \big(|C|/\mathsf{depth}(C)\big)^{1/3}$, the HSS construction from DCR of Theorem 18, and the construction of succinct $\mathbb{Z}$-bilinear $B2^{\mathsf{poly}(\lambda)}$-bounded HSS from DCR of Theorem 7. Note that, as required for Theorem 15, $\ell = \big(|C|/\mathsf{depth}(C)\big)^{1/3} \leq |C|/\mathsf{depth}(C)$ because $\mathsf{depth}(C) \leq |C|$.
Next, we analyze the communication and computation complexities, ignoring factors of $\mathsf{poly}(\lambda, \log B)$. The universal circuit takes a input circuit description of $s_{\mathsf{UC}} = \tilde{O}((|x| + |C|)/\ell) = \tilde{O}((|x|/|C| + 1)\,|C|^{2/3}\mathsf{depth}(C)^{1/3})$ bundles, computes $O(s_{\mathsf{UC}})$ gates, and has depth $D_{\mathsf{UC}} = \tilde{O}(\mathsf{depth}(C))$. The runtime of $\mathsf{ConvertInputs}$ in Enc is $O(|x|)$, to compute a dot product between the first key vector and each input bundle. The runtime of KeyGen and size of pp are both $O(\ell^2 \cdot D_{\mathsf{UC}}) = \tilde{O}(|C|^{2/3}\mathsf{depth}(C)^{1/3})$. The output size of $\mathsf{Compress}$ is $O(s_{\mathsf{UC}})$, as is the runtime of $\mathsf{Share}_{\mathsf{sinct},0}$ in Enc. The call to $\mathsf{HSS.HCircuitEval}$ in Enc takes $O(s_{\mathsf{UC}})$ computation. Therefore, the runtime of Enc is $O(|x| + \ell^2 \cdot D_{\mathsf{UC}} + s_{\mathsf{UC}}) = \tilde{O}(|x| + |C|^{2/3}\mathsf{depth}(C)^{1/3})$, and the size of ct is $O(\ell^2 D_{\mathsf{UC}}) = \tilde{O}(|C|^{2/3}\mathsf{depth}(C)^{1/3})$. □

7 Computationally succinct constrained PRF from semi-private succinct HSS

In this section, we demonstrate a second application of our techniques by constructing the first constrained pseudorandom function (CPRF) whose master evaluation algorithm is com-

putationally succinct in the size of the constraint. Our starting point is the general template of Couteau–Meyer–Passelègue–Riahinia [CMPR23] for building CPRFs from HSS. When instantiated with our bundled-then-RMS HSS, the construction inherits our computational succinctness guarantees. Concretely, the CPRF evaluation can be viewed as a two-party computation between the master-key holder and the constrained-key holder, who must compute additive shares of $C(x) \cdot \mathsf{PRF}_k(x)$ for any input x. This is achieved by homomorphically evaluating the circuit $(A, B) \mapsto \mathsf{UC}(A, x) \cdot \mathsf{PRF}_B(x)$ (with x hardcoded), where UC denotes a universal circuit. For completeness, we define the circuit $\mathsf{UC\text{-}then\text{-}PRF}_x$ using our typed-circuit framework in Fig. 9.

Typed Circuit $\mathsf{UC\text{-}then\text{-}PRF}_x$

// Informally, $\mathsf{UC\text{-}then\text{-}PRF}_x$ takes as input a boolean circuit C and a PRF key k, and computes $C(x) \cdot \mathsf{PRF}_k(x)$

Requires:

- Let $\ell \leftarrow 2^{\lceil \log_2(s^{1/3}D^{-1/3}) \rceil}$ be the bundling parameter.
- $x = (x_1, \ldots, x_\lambda) \in \{0,1\}^\lambda$ is a hardcoded value.
- UC is an $(n_{\mathsf{UC}} + s_{\mathsf{UC}})$-input, ℓ-levelled bundled circuit which is universal (Definition 12) for the class of arithmetic circuits (Definition 27); it is associated with the conversion functions $\mathsf{ConvertInputs}$, $\mathsf{ConvertCircuit}$, and $\mathsf{ConvertOutput}$.
- $\mathsf{PRF}\colon \{0,1\}^\lambda \times \{0,1\}^\lambda \to [0, 2^\lambda)$ is a pseudorandom function which can be evaluated as an RMS program.

The bundled-then-RMS circuit $\mathsf{UC\text{-}then\text{-}PRF}_x$ is obtained by

1. taking a copy of UC (whose gates and wires can naturally be mapped from levelled bundled syntax to bundled-then-RMS syntax), replacing the first n_{UC} input gates with the constant gates $\mathsf{ConvertInputs}(x) = (\tilde{x}_1, \tilde{x}_2, \ldots)$
2. adding a $\mathsf{vector\text{-}to\text{-}REG}$ gate to convert the output of UC to a register (REG)
3. taking a copy of PRF (whose gates and wires can naturally be mapped from RMS syntax to bundled-then-RMS syntax), and replacing the PRF input gates with the hardcoded constant registers $x_1, x_2, \ldots, x_\lambda$
4. replacing every wire typed 1_{REG} with a wire (typed REG) starting from the output of UC (converted to REG)

Fig. 9: Typed circuit for evaluating a universal circuit then multiplying its output with a PRF.

Computationally Succinct CPRF

Requires:

- $\mathsf{PRF}\colon \{0,1\}^\lambda \times \{0,1\}^\lambda \to [0, 2^\lambda)$ is a pseudorandom function which can be evaluated as an RMS program.
- For $x \in \{0,1\}^\lambda$, $\mathsf{UC\text{-}then\text{-}PRF}_x$ is the circuit of Fig. 9, as parameterised by PRF.
- $\mathsf{HSS} = (\mathsf{crsGen}, \mathsf{KeyGen}, \mathsf{HGateEval}, \mathsf{Dec})$ is a two-party gate-by-gate HSS scheme (Definition 16) for B-bounded bundled-then-RMS circuits (Definition 11). Let R be (a tight enough bound on) the runtime of $\mathsf{HSS.HCircuitEval}$ (as defined at the end

Definition 16)
- The wire type vector_1 should support an offline-online semi-private sharing (Definition 35) pair $(\mathsf{HSS.Share}_{\mathsf{s\text{-}priv},0}, \mathsf{HSS.Share}_{\mathsf{s\text{-}priv},1})$
- The wire type IN should support a public-key private sharing (Definition 33) algorithm $\mathsf{HSS.Share}_{\mathsf{priv}}$.
- The wire type REG should support an additive decoding pair (Definition 17) $(\mathsf{HSS.Dec}^{(0)}, \mathsf{HSS.Dec}^{(1)})$.
- Let $B = \mathcal{O}(2^{\mathsf{poly}(\lambda)})$ such that every $(x, k) \in \{0,1\}^\lambda$ is B-admissible for PRF. The class of supported constraints is then the class of all arithmetic circuits with output in $\{0,1\}$ such that every $x \in \{0,1\}^\lambda$ is B-admissible for C.

$\mathsf{KeyGen}(1^\lambda)$:

1. $r \xleftarrow{\$} \{0,1\}^R$ // R being a bound on the runtime of $\mathsf{HSS.HCircuitEval}$, it is still smaller than the size of the constraint
2. $k^{\mathsf{sec}}_{\mathsf{PRF}} \xleftarrow{\$} \{0,1\}^\lambda$
3. $\mathsf{crs} \xleftarrow{\$} \mathsf{HSS.crsGen}(1^\lambda)$
4. $(\mathsf{pp}, \mathsf{sk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \xleftarrow{\$} \mathsf{HSS.KeyGen}(1^\lambda, \mathsf{crs})$
5. $(\widetilde{C}_0, \mathsf{st}) \leftarrow \mathsf{HSS.Share}_{\mathsf{s\text{-}priv},0}(\mathsf{pp}, \mathsf{sk})$
6. $(k_0, k_1) \xleftarrow{\$} \mathsf{HSS.Share}_{\mathsf{priv}}(\mathsf{pp}, k^{\mathsf{sec}}_{\mathsf{PRF}})$
7. Output $\mathsf{msk} = (k^{\mathsf{sec}}_{\mathsf{PRF}}, r, \mathsf{crs}, \mathsf{pp}, \mathsf{sk}, \mathsf{ek}_0, \mathsf{ek}_1, \widetilde{C}_0, \mathsf{st}, k_0, k_1)$

$\mathsf{Eval}(1^\lambda, \mathsf{msk}, x)$:

1. Parse $\mathsf{msk} = (k^{\mathsf{sec}}_{\mathsf{PRF}}, r, \mathsf{crs}, \mathsf{pp}, \mathsf{sk}, \mathsf{ek}_0, \mathsf{ek}_1, \widetilde{C}_0, \mathsf{st}, k_0, k_1)$
2. $y_0 \xleftarrow{\$} \mathsf{HSS.HCircuitEval}(0, \mathsf{pp}, \mathsf{ek}_0, \mathsf{UC\text{-}then\text{-}PRF}_x, (\widetilde{C}_0, k_0); r)$
3. Output $y_0 \bmod 2^\lambda$

$\mathsf{Constrain}(1^\lambda, \mathsf{msk}, C)$:

1. Parse $\mathsf{msk} = (k^{\mathsf{sec}}_{\mathsf{PRF}}, r, \mathsf{crs}, \mathsf{pp}, \mathsf{sk}, \mathsf{ek}_0, \mathsf{ek}_1, \widetilde{C}_0, \mathsf{st}, k_0, k_1)$
2. $\widetilde{C} \leftarrow \mathsf{ConvertCircuit}(C)$
3. $\widetilde{C}_1 \xleftarrow{\$} \mathsf{HSS.Share}_{\mathsf{s\text{-}priv},1}(\mathsf{pp}, \mathsf{sk}, \widetilde{C}, \mathsf{st})$
4. Output $\mathsf{ck} = (r, \mathsf{crs}, \mathsf{pp}, \mathsf{ek}_1, k_1, \widetilde{C}_1)$

$\mathsf{CEval}(1^\lambda, \mathsf{ck}, x)$:

1. Parse $\mathsf{ck} = (r, \mathsf{crs}, \mathsf{pp}, \mathsf{ek}_1, k_1, \widetilde{C}_1)$
2. $y_1 \xleftarrow{\$} \mathsf{HSS.HCircuitEval}(1, \mathsf{pp}, \mathsf{ek}_1, \mathsf{UC\text{-}then\text{-}PRF}_x, (\widetilde{C}_1, k_1); r)$
3. Output $y_1 \bmod 2^\lambda$

Fig. 10: Computationally Succinct CPRF for $\mathsf{P/poly}$ from HSS.

Theorem 21 (Computationally Succinct CPRF for $\mathsf{P/poly}$ from HSS). *Let $\lambda \in \mathbb{N}$ be a security parameter. Let $B = 2^{\mathsf{poly}(\lambda)}$. Assume HSS has subexponential correctness for all the following properties: homomorphism (Section C.2), public-key private sharing (Definition 33), succinct semi-private sharing (Definition 34), and additive decoding for REG (Definition 17). If the parameters of HSS are chosen (using complexity leveraging) so that all the above correctness properties hold with probability at least $1 - 2^{-2\lambda}$, then the construction of Fig. 10 is a single-key, selectively secure constrained PRF supporting as constraints the class of all*

polynomial-sized circuits with output in $\{0,1\}$ *such that every* $x \in \{0,1\}^\lambda$ *is* B*-admissible for* C*.*

We refer to Section I for the proof of Theorem 21.

Corollary 22 (Computationally Succinct CPRF for P/poly from DCR). *Assuming DCR, there exists a selectively secure constrained PRF supporting as constraints the class of all polynomial-sized circuits. Moreover the computational complexity of* KeyGen *and* Eval *is sublinear in the circuit-size of the constraint. Specifically, for constraints of size* s *and depth* D*,* KeyGen *and* Eval *run in time* $\widetilde{O}(s^{2/3}D^{1/3})$*.*

Acknowledgments

This research was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme under grant agreement number 101124977 (DECRYPSIS) and the Danish Independent Research Council under Grant-IDs DFF-3103-00077B (CryptoDigi) and DFF-0165-00107B (C3PO).
Part of this work was conducted while P. Meyer, L. Roy, and P. Scholl were visiting the Simons Institute for the Theory of Computing.

References

ADOS22. Damiano Abram, Ivan Damgård, Claudio Orlandi, and Peter Scholl. An algebraic framework for silent preprocessing with trustless setup and active security. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part IV*, volume 13510 of *LNCS*, pages 421–452. Springer, Cham, August 2022.

ARS24. Damiano Abram, Lawrence Roy, and Peter Scholl. Succinct homomorphic secret sharing. In Marc Joye and Gregor Leander, editors, *EUROCRYPT 2024, Part VI*, volume 14656 of *LNCS*, pages 301–330. Springer, Cham, May 2024.

BCC+23. Dung Bui, Haotian Chu, Geoffroy Couteau, Xiao Wang, Chenkai Weng, Kang Yang, and Yu Yu. An efficient ZK compiler from SIMD circuits to general circuits. Cryptology ePrint Archive, Report 2023/1610, 2023.

Ben64. Václad E Beneš. Optimal rearrangeable multistage connecting networks. *The Bell System Technical Journal*, 43(4):1641–1656, 1964.

BGI14. Elette Boyle, Shafi Goldwasser, and Ioana Ivan. Functional signatures and pseudorandom functions. In Hugo Krawczyk, editor, *PKC 2014*, volume 8383 of *LNCS*, pages 501–519. Springer, Berlin, Heidelberg, March 2014.

BGI16. Elette Boyle, Niv Gilboa, and Yuval Ishai. Breaking the circuit size barrier for secure computation under DDH. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part I*, volume 9814 of *LNCS*, pages 509–539. Springer, Berlin, Heidelberg, August 2016.

BGN05. Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. Evaluating 2-DNF formulas on ciphertexts. In Joe Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 325–341. Springer, Berlin, Heidelberg, February 2005.

BMO+25. Amik Raj Behera, Pierre Meyer, Claudio Orlandi, Lawrence Roy, and Peter Scholl. Privately constrained PRFs from DCR: Puncturing and bounded waring rank. Cryptology ePrint Archive, Report 2025/230, 2025.

BV11. Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In Rafail Ostrovsky, editor, *52nd FOCS*, pages 97–106. IEEE Computer Society Press, October 2011.

BW13. Dan Boneh and Brent Waters. Constrained pseudorandom functions and their applications. In Kazue Sako and Palash Sarkar, editors, *ASIACRYPT 2013, Part II*, volume 8270 of *LNCS*, pages 280–300. Springer, Berlin, Heidelberg, December 2013.

CHKV25. Henry Corrigan-Gibbs, Alexandra Henzinger, Yael Tauman Kalai, and Vinod Vaikuntanathan. Somewhat homomorphic encryption from linear homomorphism and sparse LPN. In Serge Fehr and Pierre-Alain Fouque, editors, *EUROCRYPT 2025, Part II*, volume 15602 of *LNCS*, pages 3–33. Springer, Cham, May 2025.

CMPR23. Geoffroy Couteau, Pierre Meyer, Alain Passelègue, and Mahshid Riahinia. Constrained pseudorandom functions from homomorphic secret sharing. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part III*, volume 14006 of *LNCS*, pages 194–224. Springer, Cham, April 2023.

DGI+19. Nico Döttling, Sanjam Garg, Yuval Ishai, Giulio Malavolta, Tamer Mour, and Rafail Ostrovsky. Trapdoor hash functions and their applications. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 3–32. Springer, Cham, August 2019.

DHM+24. Fangqi Dong, Zihan Hao, Ethan Mook, Hoeteck Wee, and Daniel Wichs. Laconic function evaluation and ABE for RAMs from (ring-)LWE. In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024, Part III*, volume 14922 of *LNCS*, pages 107–142. Springer, Cham, August 2024.

DHMW24. Fangqi Dong, Zihan Hao, Ethan Mook, and Daniel Wichs. Laconic function evaluation, functional encryption and obfuscation for RAMs with sublinear computation. In Marc Joye and Gregor Leander, editors, *EUROCRYPT 2024, Part II*, volume 14652 of *LNCS*, pages 190–218. Springer, Cham, May 2024.

DIK10. Ivan Damgård, Yuval Ishai, and Mikkel Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 445–465. Springer, Berlin, Heidelberg, May / June 2010.

DJ01. Ivan Damgård and Mats Jurik. A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. In Kwangjo Kim, editor, *PKC 2001*, volume 1992 of *LNCS*, pages 119–136. Springer, Berlin, Heidelberg, February 2001.

Gen09. Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *41st ACM STOC*, pages 169–178. ACM Press, May / June 2009.

GHS12. Craig Gentry, Shai Halevi, and Nigel P. Smart. Fully homomorphic encryption with polylog overhead. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 465–482. Springer, Berlin, Heidelberg, April 2012.

GMW87. Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.

HLL23. Yao-Ching Hsieh, Huijia Lin, and Ji Luo. Attribute-based encryption for circuits of unbounded depth from lattices. In *64th FOCS*, pages 415–434. IEEE Computer Society Press, November 2023.

ILL25a. Yuval Ishai, Hanjun Li, and Huijia Lin. Succinct homomorphic macs from groups and applications. In *FOCS 2025*, 2025.

ILL25b. Yuval Ishai, Hanjun Li, and Huijia Lin. A unified framework for succinct garbling from homomorphic secret sharing. In *CRYPTO 2025*, 2025.

IP07. Yuval Ishai and Anat Paskin. Evaluating branching programs on encrypted data. In Salil P. Vadhan, editor, *TCC 2007*, volume 4392 of *LNCS*, pages 575–594. Springer, Berlin, Heidelberg, February 2007.

IW14. Yuval Ishai and Hoeteck Wee. Partial garbling schemes and their applications. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *ICALP 2014, Part I*, volume 8572 of *LNCS*, pages 650–662. Springer, Berlin, Heidelberg, July 2014.

KPTZ13. Aggelos Kiayias, Stavros Papadopoulos, Nikos Triandopoulos, and Thomas Zacharias. Delegatable pseudorandom functions and applications. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 669–684. ACM Press, November 2013.

LLL25. Hanjun Li, Huijia Lin, and George Lu. Succinct garbled circuits with low-depth garbling algorithms. Cryptology ePrint Archive, 2025.

LLM17. Eric Lehman, Frank Thomson Leighton, and Albert R. Meyer. *Mathematics for computer science.* 12th Media Services, 2017.

MORS24. Pierre Meyer, Claudio Orlandi, Lawrence Roy, and Peter Scholl. Rate-1 arithmetic garbling from homomorphic secret sharing. In Elette Boyle and Mohammad Mahmoody, editors, *TCC 2024, Part IV*, volume 15367 of *LNCS*, pages 71–97. Springer, Cham, December 2024.

MORS25. Pierre Meyer, Claudio Orlandi, Lawrence Roy, and Peter Scholl. Silent circuit relinearisation: Sublinear-size (boolean and arithmetic) garbled circuits from DCR. In Yael Tauman Kalai and Seny F. Kamara, editors, *CRYPTO 2025, Part IV*, volume 16003 of *LNCS*, pages 426–458. Springer, Cham, August 2025.

OSY21. Claudio Orlandi, Peter Scholl, and Sophia Yakoubov. The rise of paillier: Homomorphic secret sharing and public-key silent OT. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part I*, volume 12696 of *LNCS*, pages 678–708. Springer, Cham, October 2021.

PS18. Chris Peikert and Sina Shiehian. Privately constraining and programming PRFs, the LWE way. In Michel Abdalla and Ricardo Dahab, editors, *PKC 2018, Part II*, volume 10770 of *LNCS*, pages 675–701. Springer, Cham, March 2018.

QWW18. Willy Quach, Hoeteck Wee, and Daniel Wichs. Laconic function evaluation and applications. In Mikkel Thorup, editor, *59th FOCS*, pages 859–870. IEEE Computer Society Press, October 2018.

RS21. Lawrence Roy and Jaspal Singh. Large message homomorphic secret sharing from DCR and applications. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part III*, volume 12827 of *LNCS*, pages 687–717, Virtual Event, August 2021. Springer, Cham.

Val76. Leslie G. Valiant. Universal circuits (preliminary report). In *Proceedings of the Eighth Annual ACM Symposium on Theory of Computing*, STOC '76, page 196–203, New York, NY, USA, 1976. Association for Computing Machinery.

Wee24. Hoeteck Wee. Circuit ABE with $\mathsf{poly}(\mathsf{depth}, \lambda)$-sized ciphertexts and keys from lattices. In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024, Part III*, volume 14922 of *LNCS*, pages 178–209. Springer, Cham, August 2024.

WYY+22. Chenkai Weng, Kang Yang, Zhaomin Yang, Xiang Xie, and Xiao Wang. AntMan: Interactive zero-knowledge proofs with sublinear communication. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 2901–2914. ACM Press, November 2022.

Yao86. Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, October 1986.

A Details on Circuit Definitions

Definition 23 (Circuit Prototype [RS21, Definition 5]). A *circuit prototype* $(\mathsf{types}, \mathsf{gates}, \mathsf{in}, \mathsf{out})$ consists of a set $\mathsf{types} \subseteq \{0,1\}^*$ of *wire types* and a set $\mathsf{gates} \subseteq \{0,1\}^*$ of *gate types*, together with two maps $\mathsf{in}\colon \mathsf{gates} \to \mathsf{types}^*$ and $\mathsf{out}\colon \mathsf{gates} \to \mathsf{types}$ (assigning to each gate type the expected wire types of its inputs and output, respectively).

Definition 24 (Typed Circuit [RS21, Definition 6]). A *typed circuit* $(\mathsf{nodes}, \mathsf{wires}, \mathsf{inputs}, \mathsf{outputs}, \mathsf{type}, \mathsf{gate})$ for a circuit prototype $(\mathsf{types}, \mathsf{gates}, \mathsf{in}, \mathsf{out})$ consists of

1. a directed acyclic graph $(\mathsf{nodes}, \mathsf{wires})$
2. a total order $\preceq$ on wires
3. distinguished subsets $\mathsf{inputs}, \mathsf{outputs} \subseteq \mathsf{nodes}$, where nodes in inputs must be sources of the graph.
4. a node-labelling function $\mathsf{type}\colon \mathsf{nodes} \to \mathsf{types}$ mapping nodes into their output's wire type.
5. a node-labelling function for the non-input nodes (which we call *gates*) $\mathsf{gate}\colon \mathsf{nodes} \setminus \mathsf{inputs} \to \mathsf{gates}$

We say a typed circuit is *well-formed* if for every gate $v \in \mathsf{nodes} \setminus \mathsf{inputs}$, the incoming and outgoing wires of v match its signature type, *i.e.*:

- $\mathsf{type}(v) = \mathsf{out}(\mathsf{gate}(v))$;
- if $(v_1, v), \ldots, (v_k, v)$ are v's $\preceq$-ordered incoming wires, then $(\mathsf{type}(v_1), \ldots, \mathsf{type}(v_k)) = \mathsf{in}(\mathsf{gate}(v))$.

Unless otherwise specified, we only consider typed circuits which are *well-formed*.
In order to evaluate a typed circuit, it needs to come with a semantics that defines the wire types and gate operations. Looking ahead, a semantics could be used to define either a standard, plaintext evaluation of a circuit, or a *homomorphic evaluation* semantics to define how homomorphic operations are performed on HSS values.

Definition 25 (Typed Circuit – Semantics [RS21, definition 7]). *A* semantics $(\mathsf{values}, \mathsf{eval})$ *for a circuit prototype* $(\mathsf{types}, \mathsf{gates}, \mathsf{in}, \mathsf{out})$ *assigns each wire type* $w \in \mathsf{types}$ *a set of values* $\mathsf{values}(w)$*, and assigns each gate type* $g \in \mathsf{gates}$ *a function* $\mathsf{eval}(g)\colon \mathsf{values}(w_1) \times \cdots \times \mathsf{values}(w_k) \to \mathsf{values}(\mathsf{out}(g))$*, where* k *is the incoming arity of* g *and* $(w_1, \ldots, w_k) = \mathsf{in}(g)$ *are the input wire types of the gate.*

Given a semantics for a typed circuit, we define the complete circuit evaluation algorithm in the natural manner, as follows.

Definition 26. *Let* $C = (\mathsf{nodes}, \mathsf{wires}, \mathsf{inputs}, \mathsf{outputs}, \mathsf{type}, \mathsf{gate})$ *be a typed circuit and* $(\mathsf{values}, \mathsf{eval})$ *be a semantics for a circuit prototype* $(\mathsf{types}, \mathsf{gates}, \mathsf{in}, \mathsf{out})$*. Let* $n = |\mathsf{inputs}|$ *and* $m = |\mathsf{outputs}|$*, and denote the input nodes by* $\mathsf{inputs}_1, \ldots, \mathsf{inputs}_n$ *and the output nodes by* $\mathsf{outputs}_1, \ldots, \mathsf{outputs}_m$*. The* full evaluation algorithm $(y_1, \ldots, y_m) \xleftarrow{\$} \mathsf{CircuitEval}(C, \mathsf{values}, \mathsf{eval}, x_1, \ldots, x_n)$ *maps the* n *circuit input values* $x_i \in \mathsf{values}(\mathsf{type}(\mathsf{inputs}_i))$ *to* m *outputs* $y_j \in \mathsf{values}(\mathsf{type}(\mathsf{outputs}_i))$*, by proceeding through the circuit in topological order, and evaluating each node* v *by running* $\mathsf{eval}(\mathsf{gate}(v))$ *on the values of its predecessor nodes (ordered by* $\preceq$*).*

Arithmetic Circuits. We begin by defining the familiar case of basic arithmetic circuits over a ring $\mathcal{R}$.

Gate Type, g	Input type, $\mathsf{in}(g)$	Output type, $\mathsf{out}(g)$	$\mathsf{eval}(g)$
$c \in \mathcal{R}$	$\varnothing$	$\mathsf{element}_{\mathcal{R}}$	$\bot \mapsto c$
$+$	$\mathsf{element}_{\mathcal{R}} \times \mathsf{element}_{\mathcal{R}}$	$\mathsf{element}_{\mathcal{R}}$	$(x, y) \mapsto x + y$
$\times$	$\mathsf{element}_{\mathcal{R}} \times \mathsf{element}_{\mathcal{R}}$	$\mathsf{element}_{\mathcal{R}}$	$(x, y) \mapsto xy$

Definition 27 (Arithmetic Circuits). The *arithmetic circuit prototype* over a ring $(\mathcal{R}, +, \times)$ is defined by wires types $\mathsf{types} := \{\mathsf{element}_{\mathcal{R}}\}$, and the basic gates with input/output wire types defined as follows:
An *arithmetic circuit* is any typed circuit (Definition 24) for this circuit prototype. The *plaintext evaluation semantics* for arithmetic circuits is defined by setting $\mathsf{values}(\mathsf{element}_{\mathcal{R}}) = \mathcal{R}$ and $\mathsf{eval}(g)$ and using the $\mathsf{eval}(g)$ operations above.

B Details on Universal Circuit Construction

B.1 Proof of Theorem 13

Proof. We start by describing in Fig. 4 the basic subroutines used to build the universal circuit. Select and CSwap implement independent selection and conditional swap gates for each element of the bundle. The Beneš network [Ben64] allows any permutation on the elements of $\vec{x}$ to be implemented, according to some (efficiently computable) setting of the control bit vectors $\vec{c}_1, \ldots, \vec{c}_{2\log_2(\ell)-1}$.[9]
UniversalGate implements a batch of ℓ arithmetic circuit gates of fan-in and fan-out at most 2. If we have a batch of ℓ such gates, each gate will be either constant, addition, or multiplication, so we can use addition, multiplication, and selection to implement it. Let x_L be a bundle containing all of the left inputs to gates, and x_R be a bundle containing all the right inputs.[10] In these bundles, the inputs might be provided in a different order for x_L than for x_R, so we first use a Beneš network on x_R to reorder it to match $\vec{x}_L$. Finally, the output bundle is duplicated into $\vec{y}_L$ and $\vec{y}_R$, and each is permuted into an arbitrary order to help the routing process.
To build the universal circuit, we first create n_{UC} circuit input nodes, S instances of UniversalGate, at depths $1, 1 + (4\log_2(\ell) - 2), \ldots, 1 + (S-1)(4\log_2(\ell) - 2)$, and finally m_{UC} circuit output nodes. Each circuit input node and circuit output node is associated with a Beneš subcircuit, to adjust the ordering of the inputs and outputs within a bundle to match the original circuit. Next, we will describe how to connect these nodes together. We consider the left inputs and left outputs on their own (and do the same to the right inputs and outputs). We will connect the circuit input nodes, the left inputs, and the left outputs using Valiant's edge-universal graph [Val76] on $n_{\mathsf{UC}} + S + m_{\mathsf{UC}}$ poles, which describes a network of at most $\frac{19}{8}(n_{\mathsf{UC}} + S + m_{\mathsf{UC}}) \log_2(n_{\mathsf{UC}} + S + m_{\mathsf{UC}})$ CSwap gates that implements an arbitrary set of disjoint order-respecting connections between the $n_{\mathsf{UC}} + S$ circuit input nodes and gate left outputs and the $S + m_{\mathsf{UC}}$ circuit output nodes and gate left inputs. The edge-universal graph is acyclic, so this network does not introduce any paths from a later UniversalGate to an earlier one. Additionally, CSwap does not raise the depth of its inputs, so depths will line up when connecting the CSwaps with the UniversalGates.
Now that we have described how to construct the universal circuit (i.e., UC), next we need to show how to compile the input circuit C to run on it (i.e., describe ConvertCircuit). Go through the S colors of the batched circuit in topological order, and assign them to the S UniversalGate instances. Label each wire in the circuit as being either on the "left" or "right" side, such that no gate has two inputs or two outputs on the same side. This can be done by

[9] See [LLM17] for a modern exposition of Beneš networks.
[10] All gates in our set are symmetric, so the choice of which input is left versus right is arbitrary.

finding the bipartite graph of inputs and outputs, adding dummy edges to make it 2-regular, computing a perfect matching (which exists by Hall's marriage theorem), and noticing that the remaining edges also form a perfecting matching. Label the edges from the first matching "left", and the remaining edges "right". This justifies why we routed the left edges and right edges separately.
Next, label each edge with a "bundle position" in $[0, \ell)$, such that no two gates in the same batch (i.e., color) have inputs or outputs labeled with the same side and same bundle position. Such a labeling can be found similarly to labeling the edges with a side: find the bipartite graph with input batches and output batches as vertices, include an edge connecting two vertices if there is a left edge connecting gates within the two corresponding batches, pad it with dummy edges to be ℓ-regular, then divide it into ℓ perfect matchings and assign a bundle position to each perfect matching; then do the same for the right edges as well. These "bundle positions" define permutations for the Beneš networks in the UniversalGateS and the input and output nodes.
Finally, we just need to route the edges with each side and bundle position. Valiant showed how to efficiently compute settings for the conditional swaps in the edge-universal graph that will implement an arbitrary set of disjoint order-respecting connections, and our connections are disjoint because we computed them as a matching. Compute the routing information for Valiant's edge-universal graphs independently for each side and bundle position, and then concatenate the control bits on the same side into bundles of control bits. We have now fully routed the circuit.
The bound on s_{UC} follows from adding up all of the CSwap, Beneš, and UniversalGate calls' control bit bundles. The size of UC is $O(s_{\mathsf{UC}})$, as almost the whole circuit is built out of Select and CSwap, and each of these subcircuits takes a control bit bundle. □

B.2 Remaining Details

Following Corollary 14, we show how a not-too-deep circuit can always be batched efficiently, so that these universal circuits can be applied to arbitary circuits (of bounded depth).

Lemma 28. *There is an efficient algorithm that, for any $\ell > 0$, will compute an ℓ-batched representation of a circuit C of size s and depth D using at most $\frac{s}{\ell} + D$ colors.*

Proof. Let the graph defining C be $(\mathsf{nodes}, \mathsf{wires})$. For each gate node v in nodes, let the "level" $l(v) \in [1, D]$ be the length of the longest path from an input to v; this can be computed efficiently using dynamic programming since C is a DAG. Notice that for any pair of gates v, v' in the same level, there cannot be a path from v to v', as otherwise there would be an even longer path to v' that would put v' in a later level than v. Therefore, it is always valid to assign gates of the same level the same color. For each $d \in [1, D]$, create $\left\lceil \frac{|l^{-1}(d)|}{\ell} \right\rceil \leq 1 + \frac{|l^{-1}(d)|}{\ell}$ colors, and arbitrarily assign them to gates in $l^{-1}(d)$ such that no color covers more than ℓ gates. This defines an ℓ-batched representation of C, using at most $\sum_{d=1}^{D} \left(1 + \frac{|l^{-1}(d)|}{\ell}\right) = D + \frac{s}{\ell}$ colors. □

Next, we show how to transform a circuit into fan-out 2, without significantly increasing the size or depth.

Lemma 29. *Let C be an arithmetic circuit with n inputs, m outputs, size s, and depth D. There is an efficient procedure to transform C into C', where C' has the same inputs, same outputs, same behavior, fan-out 2, size at most $2s + m$, and depth at most $2D - 1 + \log_2(m)$.*

Proof. The general technique is to insert a tree of duplication gates at the outputs of each

gate with fan-out greater than 2. Previous work [DIK10] used balanced binary trees, blowing up the depth of C by a log factor in the worst case. We now present an algorithm that instead uses an unbalanced tree designed to minimize the overall circuit depth, and a more technical analysis of the depth to prove that it only blows up by a factor of 2.

Given a list of numbers $(w_1, \ldots, w_{f-1}, w_f)$, define the tree-max of them by the following procedure (inspired by Huffman trees). First, sort them to get $w_{i_1} \geq \cdots \geq w_{i_{f-1}} \geq w_{i_f}$, then recurse:

$$\begin{aligned}\mathsf{tree\text{-}max}(w_1, \ldots, w_{f-1}, w_f) &= \mathsf{tree\text{-}max}(w_{i_1}, \ldots, w_{i_{f-1}}, w_{i_f}) \\ &= \mathsf{tree\text{-}max}(w_{i_1}, \ldots, 2w_{i_{f-1}}).\end{aligned}$$

The base cases are $\mathsf{tree\text{-}max}(w_1, w_2) = 2\max(w_1, w_2)$ and $\mathsf{tree\text{-}max}(w_1) = 2w_1$.

Claim. For $f \geq 2$, $\quad \mathsf{tree\text{-}max}(w_1, \ldots, w_{f-1}, w_f) \leq 2\sum_i w_i - 2\min_i(w_i)$.

Proof. We prove this by induction. In the base case, $f = 2$, and we have

$$\mathsf{tree\text{-}max}(w_1, w_2) = 2\max(w_1, w_2) = 2(w_1 + w_2 - \min(w_1, w_2)) = 2\sum_i w_i - 2\min_i(w_i).$$

For induction, assume without loss of generality that $(w_1, \ldots, w_{f-1}, w_f)$ are in weakly descending order. Let $(w'_1 = w_1, \ldots, w'_{f-2} = w_{f-2}, w'_{f-1} = 2w_{f-1})$ be the parameters of the recursive call. Then

$$\begin{aligned}&\mathsf{tree\text{-}max}(w_1, \ldots, w_{f-1}, w_f) \\ &= \mathsf{tree\text{-}max}(w'_1, \ldots, w'_{f-1}) \\ &\leq 2\sum_i w'_i - 2\min_i(w'_i) \\ &= 2\sum_{i=1}^{f-2} w_i + 4w_{f-1} - 2\min(w_{f-2}, 2w_{f-1}) \\ &= 2\sum_{i=1}^{f-2} w_i + 2w_{f-1} - 2\min(w_{f-2} - w_{f-1}, 2w_{f-1} - w_{f-1}) \\ &= 2\sum_{i=1}^{f-1} w_i - 2\min(w_{f-2} - w_{f-1}, w_{f-1}) \\ &\leq 2\sum_{i=1}^{f-1} w_i = 2\sum_i w_i - 2\min_i(w_i),\end{aligned}$$

since $w_{f-2} \geq w_{f-1} \geq w_f \geq 0$. □

Parse $C = (\mathsf{nodes}, \mathsf{wires}, \mathsf{inputs}, \mathsf{outputs}, \mathsf{type}, \mathsf{gate})$. Define the sum-weight and tree-weight of every node v in nodes according to the following recursions:

$$w_{\mathbf{\Sigma}}(v) = 2 \cdot \begin{cases} 1 + \sum_{(v,v') \in \mathsf{wires}} w_{\mathbf{\Sigma}}(v') & \text{if } v \in \mathsf{outputs} \\ \sum_{(v,v') \in \mathsf{wires}} w_{\mathbf{\Sigma}}(v') & \text{otherwise} \end{cases}$$

$$w_{\mathsf{tree}}(v) = \begin{cases} \mathsf{tree\text{-}max}(1, w_{\mathsf{tree}}(v') \mid (v,v') \in \mathsf{wires}) & \text{if } v \in \mathsf{outputs} \\ \mathsf{tree\text{-}max}(\quad w_{\mathsf{tree}}(v') \mid (v,v') \in \mathsf{wires}) & \text{otherwise} \end{cases}$$

Now, we are ready to describe how to reduce the fan-out of C. For each gate (or input) v

with fanout $f > 2$, follow the execution of $\mathsf{tree\text{-}max}$ in the recursion for $w_{\mathsf{tree}}(v)$. Whenever $(\ldots, w_{i_{f-1}}, w_{i_f})$ gets replaced by $(\ldots, 2w_{i_{f-1}})$, add a duplication gate (i.e, a gate that does nothing other than providing fan-out, such as addition with 0) connecting to destinations (either subsequent gates or circuit outputs) indexed i_{f-1} and i_{f-2}, and connect v to this duplication gate instead. In each step we reduced the fan-out of v by 1, and added a gate to the circuit. The process for v stops when the fan-out reaches 2, i.e., when we've added $f-2$ gates to the circuit. This defines a fan-out 2 circuit $C' = (\mathsf{nodes}', \mathsf{wires}', \mathsf{inputs}', \mathsf{outputs}', \mathsf{type}', \mathsf{gate}')$ that is functionally equivalent to C.

We can upper-bound the size of C' essentially by using that the average fan-out is the same as the average fan-in, which is at most 2. In more detail, we count the number of added duplication gates as follows. Let x be the number of fan-out 1 gates. By counting the fan-in and the outputs, the circuit has at most $2s + m$ wires, $2s + m - x$ of which are not attached to a fan-out 1 gate. There are $s - x$ gates with fan-out at least 2, and if they were reduced to fan-out 2 they would only be attached to $2(s - x)$ wires. Every duplication gate adds 1 extra fan-out to a gate or input, so we need to add at most

$$2s + m - x - 2(s - x) = m + x \leq m + s$$

duplication gates. Therefore, the circuit size of C' is at most $2s + m$.

It is significantly trickier to bound the depth of C'. Start by defining the max-weight for every node in this modified circuit according to the recursion:

$$w_{\mathsf{max}}(v) = 2 \cdot \begin{cases} \max(1, w_{\mathsf{max}}(v') \mid (v, v') \in \mathsf{wires}') & \text{if } v \in \mathsf{outputs} \\ \max(\quad w_{\mathsf{max}}(v') \mid (v, v') \in \mathsf{wires}') & \text{otherwise} \end{cases}$$

Since our procedure for constructing C' matches with the computation of $\mathsf{tree\text{-}max}$, we have that $w_{\mathsf{max}}(v) = w_{\mathsf{tree}}(v)$ for all $v \in \mathsf{nodes}$. And we have that the depth of C' is exactly $\max_{v \in S} \log_2(w_{\mathsf{max}}(v))$, where S is the set of source nodes in C (both inputs and nullary gates), since the recurrence for w_{max} is essentially just the dynamic program for finding the longest path in C'. From the claim, we have $\mathsf{tree\text{-}max}(w_1, \ldots, w_{f-1}, w_f) \leq 2\sum_i w_i$ for all $f \geq 1$. Since $\mathsf{tree\text{-}max}$ is weakly increasing, this implies $w_{\mathsf{tree}}(v) \leq w_{\mathbf{\Sigma}}(v)$. Therefore, to upper bound the depth of C', we just need to upper bound $w_{\mathbf{\Sigma}}$.

Define the reversed sum-weight according to the recursion

$$w_{\mathsf{rev\text{-}}\mathbf{\Sigma}}(v) = 2 \cdot \begin{cases} 1 & \text{if } v \in S \\ \sum_{(v',v) \in \mathsf{wires}} w_{\mathsf{rev\text{-}}\mathbf{\Sigma}}(v') & \text{otherwise} \end{cases}$$

Then, $\sum_{v \in \mathsf{outputs}} w_{\mathsf{rev\text{-}}\mathbf{\Sigma}}(v) = \sum_{v \in S} w_{\mathbf{\Sigma}}$, as both sums count the number of labeled paths in C from a source to an output, where the labeling assigns a bit to each node on the path. Equivalently, they both count the number of paths from a source to an output, weighted by 2^L for a path of length L. Next, $w_{\mathsf{rev\text{-}}\mathbf{\Sigma}}(v) \leq 2^{2d-1}$ if v is has depth d (counting from 1), as in the base case inputs in C have reversed sum-weight 2, and in the worst case a node takes two inputs that are maximal for the previous depth. Therefore, $\sum_{v \in \mathsf{outputs}} w_{\mathsf{rev\text{-}}\mathbf{\Sigma}}(v) \leq m \cdot 2^{2D-1}$, and the depth of C' is upper bounded by

$$\begin{aligned} \max_{v \in S} \log_2(w_{\mathsf{max}}(v)) &\leq \log_2\Big(\sum_{v \in S} w_{\mathbf{\Sigma}}\Big) \\ &= \log_2\Big(\sum_{v \in \mathsf{outputs}} w_{\mathsf{rev\text{-}}\mathbf{\Sigma}}(v)\Big) \\ &\leq \log_2(m \cdot 2^{2D-1}) \\ &= 2D - 1 + \log_2(m). \end{aligned}$$

□

B.3 Proof of Theorem 15

Proof. First, use Lemma 29 to get a circuit C' with fan-out 2, size $2s+m$, and depth $2D-1+\log_2(m)$. Then set $\ell' = 2^{\lceil \log_2(s/D) \rceil}$, and run Lemma 28 to get an ℓ'-batched representation of C' with $S = \frac{2s+m}{\ell'} + 2D - 1 + \log_2(m) = O(D + \log_2(m))$ colors. Finally, plug this batched representation of C' into Corollary 14 (note that $\ell \le \ell'$) to get a universal circuit with input wire types $w_{\mathsf{UC},0} = w_{\mathsf{UC},C} = \mathsf{vector}_1$, and output wire type $w_{\mathsf{UC},1} = \mathsf{vector}_{D_{\mathsf{UC}}}$ where $D_{\mathsf{UC}} = 1 + S(4\log_2(\ell') - 2) = O((D + \log_2(m))\log_2(s/D))$. The circuit description takes

$$\begin{aligned} s_{\mathsf{UC}} &\le O\Big(\frac{1}{\ell}(n + \ell' S + m)\log_2(n + \ell' S + m)\Big) \\ &\le O\Big(\frac{1}{\ell}\big(n + \frac{s}{D}(D + \log_2(m)) + m\big)\log_2\big(n + \frac{s}{D}(D + \log_2(m)) + m\big)\Big) \\ &\le O\Big(\frac{1}{\ell}\big(n + s + \frac{s}{D}\log_2(m) + m\big)\log_2\big(n + s + \frac{s}{D}\log_2(m) + m\big)\Big) \end{aligned}$$

bundles, and the universal circuit has $O(s_{\mathsf{UC}})$ gates. □

C Details on HSS Definition

C.1 Succinct HSS Definition

Below, we give the formal correctness and privacy properties for succinct HSS.

Definition 30 (Bilinear HSS: Correctness and Privacy).

1. ***(Correctness).*** *For every $\ell \in \mathbb{N}$, $\vec{y} \in \mathcal{R}^\ell$, and PPT adversary $\mathcal{A}$ which outputs a vector $\vec{x} \in R^\ell$, we have:*

$$\Pr\left[\begin{array}{l|r} \vec{x}^{\mathsf{T}}\vec{y} \neq z^{(0)} - z^{(1)} \wedge & \mathsf{crs} \xleftarrow{\$} \mathsf{biHSS.crsGen}(1^\lambda, 1^\ell) \\ \vec{x}^{\mathsf{T}}\vec{y} \in \mathcal{B} & (E, \phi) \xleftarrow{\$} \mathsf{biHSS.Enc}(\mathsf{crs}, \vec{y}) \\ & \vec{x} \xleftarrow{\$} \mathcal{A}(1^\lambda, \mathsf{crs}, E, \phi) \\ & (d, \psi) \xleftarrow{\$} \mathsf{biHSS.Hash}(\mathsf{crs}, \vec{x}) \\ & z^{(0)} \leftarrow \mathsf{biHSS.HasherEval}(\mathsf{crs}, E, \psi) \\ & z^{(1)} \leftarrow \mathsf{biHSS.EncEval}(\mathsf{crs}, d, \phi) \end{array}\right] \le \mathsf{negl}(\lambda).$$

2. ***(Strong Hasher Privacy).*** *For every $\ell \in \mathbb{N}$ and PPT adversary $\mathcal{A}$ outputting a state and two vectors in R^ℓ, the following distributions are computationally indistinguishable:*

$\mathsf{crs} \xleftarrow{\$} \mathsf{biHSS.crsGen}(1^\lambda, 1^\ell)$	$\mathsf{crs} \xleftarrow{\$} \mathsf{biHSS.crsGen}(1^\lambda, 1^\ell)$
$(\mathsf{view}, \vec{x}_0, \vec{x}_1) \xleftarrow{\$} \mathcal{A}(1^\lambda, \mathsf{crs})$	$(\mathsf{view}, \vec{x}_0, \vec{x}_1) \xleftarrow{\$} \mathcal{A}(1^\lambda, \mathsf{crs})$
$(d, \psi) \xleftarrow{\$} \mathsf{biHSS.Hash}(\mathsf{crs}, \vec{x}_0)$	$(d, \psi) \xleftarrow{\$} \mathsf{biHSS.Hash}(\mathsf{crs}, \vec{x}_1)$
return $(\mathsf{view}, d, \mathsf{crs})$	return $(\mathsf{view}, d, \mathsf{crs})$

3. ***(Strong Encoder Privacy).*** *For every $\ell \in \mathbb{N}$ and PPT adversary $\mathcal{A}$ outputting a state and two vectors in R^ℓ, the following distributions are computationally indistinguishable:*

$\mathsf{crs} \xleftarrow{\$} \mathsf{biHSS.crsGen}(1^\lambda, 1^\ell)$	$\mathsf{crs} \xleftarrow{\$} \mathsf{biHSS.crsGen}(1^\lambda, 1^\ell)$
$(\mathsf{view}, \vec{y}_0, \vec{y}_1) \xleftarrow{\$} \mathcal{A}(1^\lambda, \mathsf{crs})$	$(\mathsf{view}, \vec{y}_0, \vec{y}_1) \xleftarrow{\$} \mathcal{A}(1^\lambda, \mathsf{crs})$
$(E, \phi) \xleftarrow{\$} \mathsf{biHSS.Enc}(\mathsf{crs}, \vec{y}_0)$	$(E, \phi) \xleftarrow{\$} \mathsf{biHSS.Enc}(\mathsf{crs}, \vec{y}_1)$
return $(\mathsf{view}, E, \mathsf{crs})$	return $(\mathsf{view}, E, \mathsf{crs})$

4. ***(Computational Succinctness).*** *The size of the hash d, and runtime of* $\mathsf{EncEval}$ *when given random access to* crs[11]*, are both* $\mathsf{poly}(\lambda)$*, independently of the length* ℓ*.*

Remark 31 (Adaptive correctness/privacy). Compared with previous definitions, our bilinear HSS definition allows for *adaptive* choice of the inputs $\vec{x}$ and $\vec{y}$ by the adversary, after seeing the CRS. By inspection of the constructions from [ARS24], it is straightforward to see that privacy still holds in this case, since the hasher and encoder messages are standard Pedersen commitments or ElGamal-like encryptions of the input, which remain hiding in this setting. The correctness of the constructions, on the other hand, relies on the randomness of the hasher's secret state ψ. Since we only give the adversary the crs and encoder's state, and not the hasher state, correctness still holds.

One further, minor difference in our definition is that we do not allow for weaker variants where crsGen additionally outputs some private information to one of the parties.

C.2 HSS Correctness Definition

The formal, gate-by-gate correctness property of HSS is given below.
Homomorphism Correctness: The homomorphic evaluation semantics must commute with Dec. That is, for every PPT adversary $\mathcal{A}$, the following probability must be negligible.

$$\Pr\left[\begin{array}{c|r} y \neq y' \\ \wedge \\ y \neq \bot \end{array} \middle| \begin{array}{r} \mathsf{crs} \xleftarrow{\$} \mathsf{HSS.crsGen}(1^\lambda) \\ (\mathsf{pp}, \mathsf{sk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda, \mathsf{crs}) \\ (g, \{(x_i^{(0)}, x_i^{(1)})\}_{i\in[k]}) \xleftarrow{\$} \mathcal{A}(1^\lambda, \mathsf{crs}, \mathsf{pp}, \mathsf{sk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \\ \forall i \in [k].\ x_i \leftarrow \mathsf{HSS.Dec}(\mathsf{pk}, \mathsf{sk}, \mathsf{in}(g)_i, x_i^{(0)}, x_i^{(1)}) \\ y \leftarrow \mathsf{eval}(g)(x_1, \ldots, x_k) \\ r \xleftarrow{\$} \$ \\ \forall \sigma \in \{0,1\}.\ y'^{(\sigma)} \leftarrow \mathsf{HSS.HGateEval}(\sigma, \mathsf{pp}, \mathsf{ek}_\sigma, g, (x_i^{(\sigma)})_{i\in[k]}; r) \\ y' \leftarrow \mathsf{HSS.Dec}(\mathsf{pk}, \mathsf{sk}, \mathsf{out}(g), y'^{(0)}, y'^{(1)}) \end{array}\right]$$

As shown in [RS21], this property suffices to ensure correctness of the circuit evaluation algorithm, $\mathsf{HSS.HCircuitEval}$, for any circuit that is chosen by the adversary, potentially *after* seeing the public parameters and keys, but *before* seeing the randomness used for the gate evaluation procedures.

Lemma 32 (Circuit Homomorphism Correctness [RS21, Lemma 14]). *Let* HSS *be a two-party gate-by-gate HSS scheme for a circuit prototype* $(\mathsf{types}, \mathsf{gates}, \mathsf{in}, \mathsf{out})$ *with plaintext evaluation semantics* $(\mathsf{values}, \mathsf{eval})$*. Then decodings the input shares and evaluating an arbitrary circuit on the plaintexts must (except with negligible probability) give the same result as evaluating the circuit homomorphically, then decoding the results. That is, for any PPT adversary* $\mathcal{A}$*, the following probability must be negligible.*

$$\Pr\left[\begin{array}{c|r} y \neq y' \\ \wedge \\ y \neq \bot \end{array} \middle| \begin{array}{r} \mathsf{crs} \xleftarrow{\$} \mathsf{HSS.crsGen}(1^\lambda) \\ (\mathsf{pp}, \mathsf{sk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda, \mathsf{crs}) \\ (C, \{(x_i^{(0)}, x_i^{(1)})\}_{i\in[n]}) \xleftarrow{\$} \mathcal{A}(1^\lambda, \mathsf{crs}, \mathsf{pp}, \mathsf{sk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \\ \forall i \in [n].\ x_i \leftarrow \mathsf{HSS.Dec}(\mathsf{pk}, \mathsf{sk}, C.\mathsf{type}(C.\mathsf{inputs}_i), x_i^{(0)}, x_i^{(1)}) \\ y \leftarrow \mathsf{CircuitEval}(C, \mathsf{values}, \mathsf{eval}, x_1, \ldots, x_n) \\ r \xleftarrow{\$} \$ \\ \forall \sigma \in \{0,1\}.\ y'^{(\sigma)} \leftarrow \mathsf{HSS.HCircuitEval}(\sigma, \mathsf{pp}, \mathsf{ek}_\sigma, C, x_1^{(\sigma)}, \ldots, x_n^{(\sigma)}; r) \\ \forall j \in [m].\ y'_j \leftarrow \mathsf{HSS.Dec}(\mathsf{pk}, \mathsf{sk}, C.\mathsf{type}(C.\mathsf{outputs}_j), y_j'^{(0)}, y_j'^{(1)}) \\ y' \leftarrow (y'_1, \ldots, y'_m) \end{array}\right]$$

[11] In the constructions from [ARS24], $\mathsf{EncEval}$ only needs to be given a small part of the CRS containing a group description; the rest can be ignored.

C.3 HSS Privacy Definitions

Definition 33 (Public-key Private Sharing). *Let* HSS *be an HSS for a circuit prototype* $(\mathsf{types}, \mathsf{gates}, \mathsf{in}, \mathsf{out})$*. Then wire type* $w \in \mathsf{types}$ *supports* public-key private sharing *if there exists a PPT algorithm* $\mathsf{HSS.Share}_{\mathsf{priv},w}(\mathsf{pp}, x)$ *mapping from* $x \in \mathsf{values}(w) \setminus \{\bot\}$ *to* $(y^{(0)}, y^{(1)}) \in \mathsf{values}_0(w_0) \times \mathsf{values}_1(w_1)$*, satisfying the following properties:*

1. **Correctness:** *For every* p.p.t. *adversary* $\mathcal{A}$*,*

$$\Pr\left[x \neq y \;\middle|\; \begin{array}{r} \mathsf{crs} \xleftarrow{\$} \mathsf{HSS.crsGen}(1^\lambda) \\ (\mathsf{pp}, \mathsf{sk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda, \mathsf{crs}) \\ x \xleftarrow{\$} \mathcal{A}(1^\lambda, \mathsf{crs}, \mathsf{pp}, \mathsf{sk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \\ (x^{(0)}, x^{(1)}) \xleftarrow{\$} \mathsf{Share}_{\mathsf{priv},w}(1^\lambda, \mathsf{pp}, x) \\ y \leftarrow \mathsf{Dec}(\mathsf{pp}, \mathsf{sk}, w, x^{(0)}, x^{(1)}) \end{array}\right] \leq \mathsf{negl}(\lambda).$$

   In other words, honestly generated shares should by decoded to the expected value with all but negligible probability.

2. **Privacy:** *For every* p.p.t. *adversary* $\mathcal{A}$ *outputting* $x, y \in \mathsf{values}(w)$ *and every corrupt party* $\sigma \in \{0, 1\}$*, the following distributions must be indistinguishable:*

$\mathsf{crs} \xleftarrow{\$} \mathsf{HSS.crsGen}(1^\lambda)$	$\mathsf{crs} \xleftarrow{\$} \mathsf{HSS.crsGen}(1^\lambda)$
$(\mathsf{pp}, \mathsf{sk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda, \mathsf{crs})$	$(\mathsf{pp}, \mathsf{sk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda, \mathsf{crs})$
$(\mathsf{view}, x, y) \xleftarrow{\$} \mathcal{A}(1^\lambda, \mathsf{crs}, \mathsf{pp}, \mathsf{ek}_\sigma)$	$(\mathsf{view}, x, y) \xleftarrow{\$} \mathcal{A}(1^\lambda, \mathsf{crs}, \mathsf{pp}, \mathsf{ek}_\sigma)$
$(x^{(0)}, x^{(1)}) \xleftarrow{\$} \mathsf{Share}_{\mathsf{priv},w}(1^\lambda, \mathsf{pp}, x)$	$(x^{(0)}, x^{(1)}) \xleftarrow{\$} \mathsf{Share}_{\mathsf{priv},w}(1^\lambda, \mathsf{pp}, y)$
return $(\mathsf{view}, x^{(\sigma)})$	return $(\mathsf{view}, x^{(\sigma)})$

Definition 34 (Succinct Semi-Private Sharing). *Let* HSS *be an HSS for a circuit prototype* $(\mathsf{types}, \mathsf{gates}, \mathsf{in}, \mathsf{out})$*. Then wire type* $w \in \mathsf{types}$ *supports* succinct semi-private sharing *if there exists PPT algorithms:*

1. $\mathsf{HSS.Hash}_w(\mathsf{crs}, x)$*: Given the* crs *and an input* $x \in \mathsf{values}(w) \setminus \{\bot\}$*, output a randomized digest* d *of* x*, along with the randomness* ψ*.*
2. $\mathsf{HSS.Share}_{\mathsf{sinct},0,w}(\mathsf{pp}, \mathsf{ek}_0, d)$*: compute the succinct party's share* $x^{(0)} \in \mathsf{values}_0(w)$ *of* x*, using its evaluation key* ek_0 *and the digest* d*.*
3. $\mathsf{HSS.Share}_{\mathsf{sinct},1,w}(\mathsf{pp}, \mathsf{ek}_1, x, \psi)$*: compute the non-succinct party's share* $x^{(1)} \in \mathsf{values}_1(w)$ *of* x*, using its evaluation key* ek_1*, the full input* x*, and the digest randomness* ψ*.*

satisfying the following properties:

1. **Correctness:** *For every* p.p.t. *adversary* $\mathcal{A}$*,*

$$\Pr\left[x \neq y \;\middle|\; \begin{array}{r} \mathsf{crs} \xleftarrow{\$} \mathsf{HSS.crsGen}(1^\lambda) \\ (\mathsf{pp}, \mathsf{sk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda) \\ x \xleftarrow{\$} \mathcal{A}(1^\lambda, \mathsf{crs}, \mathsf{pp}, \mathsf{sk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \\ (d, \psi) \xleftarrow{\$} \mathsf{HSS.Hash}_w(\mathsf{crs}, x) \\ x^{(0)} \xleftarrow{\$} \mathsf{HSS.Share}_{\mathsf{sinct},0,w}(\mathsf{pp}, \mathsf{ek}_0, d) \\ x^{(1)} \xleftarrow{\$} \mathsf{HSS.Share}_{\mathsf{sinct},1,w}(\mathsf{pp}, \mathsf{ek}_1, x, \psi) \\ y \leftarrow \mathsf{Dec}(\mathsf{pp}, \mathsf{sk}, w, x^{(0)}, x^{(1)}) \end{array}\right] \leq \mathsf{negl}(\lambda).$$

   That is, shares generated by the succinct semi-private sharing should be decrypt correctly, with all but negligible probability.

2. **Hasher Privacy:** *For every* p.p.t. *adversary* $\mathcal{A}$ *outputting* $x, y \in \mathsf{values}(w)$*, the following distributions must be indistinguishable:*

$\mathsf{crs} \xleftarrow{\$} \mathsf{HSS.crsGen}(1^\lambda)$	$\mathsf{crs} \xleftarrow{\$} \mathsf{HSS.crsGen}(1^\lambda)$
$(\mathsf{pp}, \mathsf{sk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda, \mathsf{crs})$	$(\mathsf{pp}, \mathsf{sk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda, \mathsf{crs})$
$(\mathsf{view}, x, y) \xleftarrow{\$} \mathcal{A}(1^\lambda, \mathsf{crs}, \mathsf{pp}, \mathsf{ek}_0)$	$(\mathsf{view}, x, y) \xleftarrow{\$} \mathcal{A}(1^\lambda, \mathsf{crs}, \mathsf{pp}, \mathsf{sk}, (\mathsf{ek}_0, \mathsf{ek}_1))$
$(d, \psi) \xleftarrow{\$} \mathsf{HSS.Hash}_w(\mathsf{crs}, x)$	$(d, \psi) \xleftarrow{\$} \mathsf{HSS.Hash}_w(\mathsf{crs}, y)$
return (view, d)	return (view, d)

Definition 35 (Offline-Online Semi-Private Sharing). *Let* HSS *be an HSS for a circuit prototype* $(\mathsf{types}, \mathsf{gates}, \mathsf{in}, \mathsf{out})$*. Then wire type* $w \in \mathsf{types}$ *supports* offline-online semi-private sharing *if there exists PPT algorithms:*

1. $\mathsf{HSS.Share}_{\mathsf{s\text{-}priv},0,w}(\mathsf{pp}, \mathsf{sk})$: *On input the public parameters* pp *and a secret key* sk*, sample the offline share* $x^{(0)}$ *(of a value that is not yet chosen) and a state* st*.*
2. $\mathsf{HSS.Share}_{\mathsf{s\text{-}priv},1,w}(\mathsf{pp}, \mathsf{sk}, x, \mathsf{st})$: *On input the public parameters* pp*, a secret key* sk*, a semi-private input* x*, and a state* st*, sample the online share* $x^{(1)}$*.*

satisfying the following properties:

1. **Correctness:** *The shares must decode correctly. That is, for every* p.p.t. *adversary* $\mathcal{A}$*,*

$$\Pr\left[(x \neq y) \;\middle|\; \begin{array}{r} (\mathsf{pp}, \mathsf{sk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda) \\ x \xleftarrow{\$} \mathcal{A}(1^\lambda, \mathsf{pp}, \mathsf{sk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \\ (x^{(0)}, \mathsf{st}) \xleftarrow{\$} \mathsf{Share}_{0,\mathsf{s\text{-}priv}}(1^\lambda, \mathsf{pp}, \mathsf{sk}) \\ x^{(1)} \xleftarrow{\$} \mathsf{Share}_{1,\mathsf{s\text{-}priv}}(1^\lambda, \mathsf{pp}, \mathsf{sk}, x, \mathsf{st}) \\ y \leftarrow \mathsf{Dec}(\mathsf{pp}, \mathsf{sk}, w, x^{(0)}, x^{(1)}) \end{array}\right] \leq \mathsf{negl}(\lambda).$$

2. **Security:** *Each share individually must leak nothing about the secret key. That is, there must be a PPT simulator* Sim *such that, for every PPT adversary* $\mathcal{A}$ *and* $\sigma \in \{0, 1\}$*, the following distributions are indistinguishable*

$(\mathsf{pp}, \mathsf{sk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda)$	$(\mathsf{pp}, \mathsf{sk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda)$
$(\mathsf{view}, x) \xleftarrow{\$} \mathcal{A}(1^\lambda, \mathsf{pp}, \mathsf{sk}, (\mathsf{ek}_0, \mathsf{ek}_1))$	$(\mathsf{view}, x) \xleftarrow{\$} \mathcal{A}(1^\lambda, \mathsf{pp}, \mathsf{sk}, (\mathsf{ek}_0, \mathsf{ek}_1))$
$(x^{(0)}, \mathsf{st}) \xleftarrow{\$} \mathsf{Share}_{0,\mathsf{s\text{-}priv}}(1^\lambda, \mathsf{pp}, \mathsf{sk})$	$x^{(\sigma)} \xleftarrow{\$} \mathsf{Sim}(\sigma, \mathsf{crs}, \mathsf{pp}, \mathsf{ek}_\sigma, x)$
$x^{(1)} \xleftarrow{\$} \mathsf{Share}_{1,\mathsf{s\text{-}priv}}(1^\lambda, \mathsf{pp}, \mathsf{sk}, x, \mathsf{st})$	return $(\mathsf{view}, x^{(\sigma)})$
return $(\mathsf{view}, x^{(\sigma)})$	

C.4 Details on HSS Construction

C.5 Restricted Multiplication HSS Operations

In Fig. 11, we present the gate operations for RM circuits, with wire types IN and REG, used as part of our main construction.

HSS Restricted Multiplication: Gate Evaluation and Wire Decoding

Gate, g	$\mathsf{eval}_\sigma(g) : x^{(\sigma)}$	$\to$	$y^{(\sigma)}$
1_{IN} :	$\perp$	$\mapsto$	$(\underbrace{(\exp(1),1)}_{\mathsf{DJE.Enc}(K)}, \underbrace{(1,\exp(1))}_{\mathsf{DJE.Enc}(1)})$
1_{REG} :	$\perp$	$\mapsto$	$(K^{(\sigma)}, \sigma)$
$+_{\mathsf{REG}}$:	$(x_1^{(\sigma)}, x_2^{(\sigma)})$	$\mapsto$	$x_1^{(\sigma)} + x_2^{(\sigma)}$
$\times_{\mathsf{IN},v}$:	$(\underbrace{(\boldsymbol{c}_0^{\mathsf{auth}}, \boldsymbol{c}_1^{\mathsf{auth}})}_{\mathsf{DJE.Enc}(K\,x)}, \underbrace{(\boldsymbol{c}_0, \boldsymbol{c}_1)}_{\mathsf{DJE.Enc}(x)})$	$\mapsto$	$(((\boldsymbol{c}_0^{\mathsf{auth}})^v, (\boldsymbol{c}_1^{\mathsf{auth}})^v), (\boldsymbol{c}_0^v, \boldsymbol{c}_1^v))$
$\times_{\mathsf{REG},c}$:	$(\langle Kx\rangle_\sigma, \langle x\rangle_\sigma)$	$\mapsto$	$(c\langle Kx\rangle_\sigma, c\langle x\rangle_\sigma)$
$\times_{\mathsf{IN,REG}}$:	$(((\boldsymbol{c}_0^{\mathsf{auth}}, \boldsymbol{c}_1^{\mathsf{auth}}), (\boldsymbol{c}_0, \boldsymbol{c}_1)), (\langle Kx\rangle_\sigma, \langle x\rangle_\sigma))$	$\mapsto$	$(\mathsf{DDLog}\left((\boldsymbol{c}_0^{\mathsf{auth}})^{\langle Kx\rangle_\sigma}(\boldsymbol{c}_1^{\mathsf{auth}})^{\langle x\rangle_\sigma}\right) + r \bmod N^\zeta,$ $\mathsf{DDLog}\left(\boldsymbol{c}_0^{\langle Kx\rangle_\sigma}\boldsymbol{c}_1^{\langle x\rangle_\sigma}\right) + r \bmod N^\zeta)$

$\mathsf{HSS.Dec}(\mathsf{pp}, \mathsf{sk}, \mathsf{REG}, (\langle K\,x\rangle_0, \langle x\rangle_0), \langle K\,x\rangle_1, \langle x\rangle_1)$:

- $x \leftarrow \langle x\rangle_1 - \langle x\rangle_0$
- **assert** $\langle K\,x\rangle_1 - \langle K\,x\rangle_0 = K\,x$
- return x // Dec always returns $\perp$ if any assertion fails

$\mathsf{HSS.Dec}(\mathsf{pp}, \mathsf{sk}, \mathsf{IN}, (c^{\mathsf{auth}}, c), (\hat{c}^{\mathsf{auth}}, \hat{c}))$:

- $x^{\mathsf{auth}} = \mathsf{DJE.Dec}_{K,N}(c^{\mathsf{auth}})$
- $x = \mathsf{DJE.Dec}_{K,N}(c)$
- **assert** $x^{\mathsf{auth}} = Kx$ and $(c^{\mathsf{auth}}, c) = (\hat{c}^{\mathsf{auth}}, \hat{c})$
- return x

Fig. 11: HSS restricted multiplication gate evaluation

C.6 Bundled HSS Operations

In Fig. 12, we describe the missing linear gates and vector-to-register conversion gate, for the vector wire types used in our construction. Given all the individual gate specifications so far, in Fig. 13, we then describe the general, gate evaluation algorithm, HSS.HGateEval.

HSS: Vector-to-register conversion and additional linear operations

vector-to-REG_d gate: eval_σ works as follows[a]

- If $d < D$: $x_1^{(\sigma)} \xleftarrow{\$} \mathsf{level\text{-}switch}_{d,D}(x_1^{(\sigma)})$
- $\vec{v} \leftarrow \left[1\,0\cdots 0\right]^{\mathsf{T}}$ // Represented as a sparse vector.
- $x_2^{(\sigma)} \xleftarrow{\$} \mathsf{constant}_{\vec{v},D-1}()$
- return $y^{(\sigma)} \xleftarrow{\$} \mathsf{mult}_{D-1}(x_1^{(\sigma)}, x_2^{(\sigma)})$

add_d:

$$\begin{aligned}\mathsf{eval}_0 &: (\overline{x}_1^{(0)}, \overline{x}_2^{(0)}) && \mapsto \overline{x}_1^{(0)} + \overline{x}_2^{(0)}\\ \mathsf{eval}_1 &: (\overline{x}_1^{(1)}, \vec{x}_1), (\overline{x}_2^{(1)}, \vec{x}_2) \mapsto (\overline{x}_1^{(0)} + \overline{x}_2^{(0)}, \vec{x}_1 + \vec{x}_2)\end{aligned}$$

$\mathsf{scalar\text{-}mult}_{c,d}$:

$$\begin{aligned}\mathsf{eval}_0 &: \overline{x}^{(0)} && \mapsto c\overline{x}^{(0)}\\ \mathsf{eval}_1 &: (\overline{x}^{(1)}, \vec{x}) \mapsto (c\overline{x}^{(1)}, c \cdot \vec{x})\end{aligned}$$

$\mathsf{constant}_{\vec{v},d}$: eval_σ proceeds as follows: // $\vec{v}$ is represented as a sparse vector

- Compute

$$y^{(\sigma)} \leftarrow \begin{cases} \left((\vec{k}^{(1)})^\mathsf{T}\vec{v}, \vec{v}\right) & \text{if } \sigma = 1 \\ (\vec{k}^{(0)})^\mathsf{T}\vec{v} & \text{if } \sigma = 0 \end{cases}$$

- If $d > 1$: $y^{(\sigma)} \xleftarrow{\$} \mathsf{level\text{-}switch}_{1,d}(y^{(\sigma)})$
- return $y^{(\sigma)}$

[a] We abuse notation slightly, writing e.g. $\mathsf{level\text{-}switch}_{d,D}(x_1^{(\sigma)})$ to mean: let $f = \mathsf{eval}_\sigma(\mathsf{level\text{-}switch}_{d,D})$ and return $f(x_1^{(\sigma)})$.

Fig. 12: Bundled HSS conversion and linear gates

HSS Gate Evaluation Algorithm

$\mathsf{HSS.HGateEval}(\sigma, \mathsf{pp}, \mathsf{ek}_\sigma, k, \mathsf{gate}, (x_i^{(\sigma)})_{i=1}^k; r)$

1. Depending on (k, gate), apply the relevant $\mathsf{eval}(\mathsf{gate})$ algorithm from Figs. 6, 11 and 12 and return the result.

Fig. 13: HSS gate evaluation algorithm

D Construction of Key Switching Graphs

Theorem 36 (Key Switching Graph). *For any power of two $D \geq 2$, there exists a DAG $G_\mathsf{KS} = ([D], E_\mathsf{KS})$ such that $|E| = 2D - 3$, and for every pair of vertices $d < d' \in [D]$ there is a directed path from d to d' of length $\lfloor 2\log_2(d' - d + 1)\rfloor + 1$.*

Proof. We prove by induction that we can construct a family $(G_1, \ldots)$ of graphs, where $G_i = ([2^i], E_i)$, $|E_i| = 2^{i+1} - 3$, $G_i \subseteq G_{i-1}$, G_i is symmetric under reversing the order of the vertices and the direction of all edges, G_i connects $(d, d+1)$ for all $d \in [2^i - 1]$, and G_i satisfies the following routing requirement: (a) for any $d < d' \in [2^i]$ with d' a power of two, there is a path from d to d' of length at most $\lfloor \log_2(d' - d)\rfloor + 1$. Note that putting symmetry together with the routing requirment implies a symmetrical statement: (b) for any $j \leq i$ and $d < d' \in [2^j]$ with $2^j + 1 - d$ a power of two, there is a path from d to d' of length at most $\lfloor \log_2(d' - d)\rfloor + 1$.

Base case: The first graph is $G_1 = ([2], E_1 = \{(1, 2)\})$. The only path has length 1, which is consistent with all required properties.

Induction: Let $G + X$ be the operation of shifting numbers labeling the vertices in G by X, and otherwise leaving X G unchanged. Then we construct G_i in terms of G_{i-1} as

$$G_i = G_{i-1} \cup \left(G_{i-1} + 2^{i-1}\right) \cup \left\{\left(2^{i-1}, 2^{i-1}+1\right), \left(2^{i-1}, 2^i\right), \left(1, 2^{i-1}+1\right)\right\}.$$

We have $|E_i| = 2|E_{i-1}| + 3 = 2 \cdot (2^{i-1+1} - 3) + 3 = 2^{i+1} - 3$. Reversal symmetry is preserved, as the two copies of G_{i-1} are placed symmetrically, and the added edges are also symmetric. Next, we prove the routing requirement: $d < d' \in [2^i]$ with d' a power of two. If d and d' are both in $[1, 2^{i-1}]$, or both in $[2^{i-1}+1, 2^i]$ then the induction hypothesis gives us the path we need, so assume that $d \in [1, 2^{i-1}]$ and $d' \in [2^{i-1}+1, 2^i]$. This implies that $d' = 2^i$. Therefore we have a length $\lceil \log_2(2^{i-1} - d) \rceil + 1$ path given by first going to 2^{i-1} using the induction hypothesis, then following the added edge $(2^{i-1}, 2^i)$. And

$$\lceil \log_2(2^{i-1} - d) \rceil + 1 = \lceil \log_2(2(2^{i-1} - d)) \rceil = \lceil \log_2(2^i - 2d) \rceil \leq \lceil \log_2(d' - d) \rceil.$$

General Routing Finally, we show the existence of short paths between $d < d'$, without restrictions on d and d' being powers of two. Choose j such that $d \leq 2^j \leq d' < 2^{j+1}$. Then we can go from d to 2^j using at most $\lfloor \log_2(2^j - d + 1) \rfloor + 1$ steps (by routing (a)).[12] Next, we can go from 2^j to $2^j + 1$ using 1 step. Finally, $d' < 2^j$ and $2^{j+1} + 1 - (2^j + 1) = 2^j$ is a power of two, so we can use routing (b) to get a path of length at most $\lfloor \log_2(d' - 2^j - 1 + 1) \rfloor + 1$ from $2^j + 1$ to d'. Adding these up, we have a path of length at most

$$\begin{aligned}
\lfloor \log_2(2^j - d + 1) \rfloor + \lfloor \log_2(d' - 2^j) \rfloor + 3 &\leq \lfloor \log_2((2^j - d + 1)(d' - 2^j)) \rfloor + 3 \\
&\leq \left\lfloor \log_2 \left(\left(\frac{d' - d + 1}{2} \right)^2 \right) \right\rfloor + 3 \\
&= \lfloor 2 \log_2(d' - d + 1) \rfloor + 1. \qquad \square
\end{aligned}$$

E Semi-Private HSS, Gate-by-Gate Correctness

E.1 Correctness of component-wise multiplication

Lemma 37 (Homomorphic component-wise multiplication). *Let λ be a security parameter. Let $B > 0$, $\zeta \in \mathbb{N} \setminus \{0, 1\}$, and $\ell \in \mathbb{N}^*$.*

- *For all $\vec{x} = (x_0, \ldots, x_{\ell-1}) \in \mathbb{Z}^\ell$ and $\vec{y} = (y_0, \ldots, y_{\ell-1}) \in \mathbb{Z}^\ell$ such that $\|\vec{x}\|_\infty, \|\vec{y}\|_\infty, \|\vec{x} \odot \vec{y}\|_\infty < B$,*
- *For all $N \in \mathsf{Supp}(\mathsf{RSA.Gen}(1^\lambda))$ such that $N^{\zeta-1} > 2^\lambda \ell B$,*
- *For all $g \in (\mathbb{Z}/N^\zeta\mathbb{Z})^\times$,*
- *For all $\vec{k} = (k_0, \ldots, k_{\ell-1}) \in [0, N)^\ell$, and $\vec{k'} = (k'_0, \ldots, k'_{\ell-1}) \in [0, N)^\ell$,*
- *For all $(x^{(0)}_{\mathsf{share}}, x^{(1)}_{\mathsf{share}}) \in \mathbb{Z}^2$ such that $x^{(1)}_{\mathsf{share}} - x^{(0)}_{\mathsf{share}} = \vec{k}^\mathsf{T} \cdot \vec{x}$,*
- *For all $(y^{(0)}_{\mathsf{share}}, y^{(1)}_{\mathsf{share}}) \in \mathbb{Z}^2$ such that $y^{(1)}_{\mathsf{share}} - y^{(0)}_{\mathsf{share}} = (\vec{k'})^\mathsf{T} \cdot \vec{y}$,*

$$\Pr\left[z_1 - z_0 = (\vec{k'})^\mathsf{T} \cdot (\vec{x} \odot \vec{y}) \;\middle|\; \begin{array}{c} r \xleftarrow{\$} \mathbb{Z}/N^\zeta\mathbb{Z} \\ (z_0, z_1) \coloneqq \mathsf{ShareConv}_\odot(N, g, \vec{k}, \vec{k'}, \vec{x}, \vec{y}, x^{(0)}_{\mathsf{share}}, y^{(0)}_{\mathsf{share}}, r) \end{array} \right] \geq 1 - 2^{-\lambda}$$

where the deterministic algorithm $\mathsf{ShareConv}_\odot$ is defined in Fig. 14.

[12] We added 1 to the input to the logarithm, to avoid taking $\log_2(0)$ in the case where $d = 2^j$.

Algorithm $\mathsf{ShareConv}_{\odot}$

$\mathsf{ShareConv}_{\odot}(N, g, \vec{k}, \vec{k}', \vec{x}, \vec{y}, x^{(0)}_{\mathsf{share}}, y^{(0)}_{\mathsf{share}}, r)$:

1. $x^{(1)}_{\mathsf{share}} \leftarrow \vec{k}^{\mathsf{T}} \cdot \vec{x} + x^{(0)}_{\mathsf{share}}$
2. $y^{(1)}_{\mathsf{share}} \leftarrow (\vec{k}')^{\mathsf{T}} \cdot \vec{y} + y^{(0)}_{\mathsf{share}}$
3. *For* $i \in [0, \ell-1]$, $h_i \leftarrow g^{-k_i}$
4. *For* $i \in [0, \ell-1]$, $h'_i \leftarrow g^{-k'_i}$
5. *For* $(i,j) \in [0, \ell-1]^2$, $h_{i,j} \leftarrow g^{k_i k'_j} \exp(\delta_{i,j}\ k'_i)$
6. $z_1 \leftarrow \left\lfloor \mathsf{DDLog}\left(g^{x^{(1)}_{\mathsf{share}} y^{(1)}_{\mathsf{share}}} \left(\prod_{(i,j)\in[0,\ell)^2} h_{i,j}^{x_i y_j} \right) \left(\prod_{i\in[0,\ell)} h_i^{x_i y^{(1)}_{\mathsf{share}}} \right) \left(\prod_{i\in[0,\ell)} (h'_i)^{y_i x^{(1)}_{\mathsf{share}}} \right) \right) + r \right\rceil \bmod N^{\zeta}$
7. $z_0 \leftarrow \left\lfloor \mathsf{DDLog}\left(g^{x^{(0)}_{\mathsf{share}} y^{(0)}_{\mathsf{share}}} \right) + r \right\rceil \bmod N^{\zeta}$
8. *Output* (z_0, z_1)

Fig. 14: Share conversion procedure for homomorphic component-wise multiplication.

Proof. Our starting point is the following algebraic identity:

$$\begin{aligned} x^{(0)}_{\mathsf{share}}\ y^{(0)}_{\mathsf{share}} &= (x^{(1)}_{\mathsf{share}} - k^{\mathsf{T}} \cdot \vec{x})\ (y^{(1)}_{\mathsf{share}} - k^{\mathsf{T}} \cdot \vec{y}) \\ &= x^{(1)}_{\mathsf{share}} y^{(1)}_{\mathsf{share}} \\ &\quad + (k^{\mathsf{T}} \cdot \vec{x})(k^{\mathsf{T}} \cdot \vec{y}) && // \textstyle\sum_{(i,j)\in[0,\ell)^2} k_i k'_j\ x_i y_j \\ &\quad - k^{\mathsf{T}} \cdot (y^{(1)}_{\mathsf{share}} \vec{x}) && // \textstyle\sum_{i\in[0,\ell)} k_i\ x_i y^{(1)}_{\mathsf{share}} \\ &\quad - (k')^{\mathsf{T}} \cdot (x^{(1)}_{\mathsf{share}} \vec{y}) && // \textstyle\sum_{i\in[0,\ell)} k'_i\ y_i x^{(1)}_{\mathsf{share}} \end{aligned}$$

If we further observe that

1. $g^{(k^{\mathsf{T}}\cdot\vec{x})(k^{\mathsf{T}}\cdot\vec{y})} = g^{\sum_{(i,j)\in[0,\ell)^2} k_i k'_j\ x_i y_j} = \prod_{(i,j)\in[0,\ell)^2} (\overbrace{g^{k_i k'_j}}^{= h_{i,j}/\exp(\delta_{i,j} k'_i)})^{x_i y_j}$
$$= \left(\prod_{(i,j)\in[0,\ell)^2} h_{i,j}^{x_i y_k} \right) \exp(\vec{k'}(\vec{x} \odot \vec{y}))$$
2. $g^{-k^{\mathsf{T}}\cdot(y^{(1)}_{\mathsf{share}}\vec{x})} = g^{\sum_{i\in[0,\ell)} k_i\ x_i y^{(1)}_{\mathsf{share}}} = \prod_{i\in[0,\ell)} h_i^{x_i y^{(1)}_{\mathsf{share}}}$
3. $g^{-(k')^{\mathsf{T}}\cdot(x^{(1)}_{\mathsf{share}}\vec{y})} = g^{\sum_{i\in[0,\ell)} k'_i\ y_i x^{(1)}_{\mathsf{share}}} = \prod_{i\in[0,\ell)} (h'_i)^{y_i x^{(1)}_{\mathsf{share}}}$

then it follows that $g_1 \coloneqq \left(g^{x^{(1)}_{\mathsf{share}} y^{(1)}_{\mathsf{share}}} \left(\prod_{(i,j)\in[0,\ell)^2} h_{i,j}^{x_i y_j} \right) \left(\prod_{i\in[0,\ell)} h_i^{x_i y^{(1)}_{\mathsf{share}}} \right) \left(\prod_{i\in[0,\ell)} (h'_i)^{y_i x^{(1)}_{\mathsf{share}}} \right) \right)$ and $g_0 \coloneqq \left(g^{x^{(0)}_{\mathsf{share}} y^{(0)}_{\mathsf{share}}} \right)$ are divisive shares of $\exp(\vec{k'}^{\mathsf{T}} \cdot (\vec{x} \odot \vec{y}))$. By Lemma 2, $\mathsf{DDLog}(g_1) - \mathsf{DDLog}(g_0) \equiv (\vec{k'})^{\mathsf{T}} \cdot (\vec{x} \odot \vec{y}) \bmod N^{\zeta}$. By Lemma 3, $\Pr[z_1 - z_0 = (\vec{k'})^{\mathsf{T}} \cdot (\vec{x} \odot \vec{y})] \geq 1 - \frac{|(\vec{k'})^{\mathsf{T}} \cdot (\vec{x} \odot \vec{y})|}{N^{\zeta}} \geq 1 - \frac{\ell N \|\vec{x} \odot \vec{y}\|_{\infty}}{N^{\zeta}} \geq 1 - \frac{\ell B}{N^{\zeta-1}} \geq 1 - 2^{-\lambda}$. □

E.2 Correctness of bitflip-swaps

Lemma 38 (Homomorphic bitflip-swap). *Let λ be a security parameter. Let $B > 0$, $\zeta \in \mathbb{N} \setminus \{0, 1\}$, $\ell \in 2^{\mathbb{N}}$, and $t \in [0, \log \ell)$.*

– *For all $\vec{x} = (x_0, \ldots, x_{\ell-1}) \in \mathbb{Z}^{\ell}$ and $\vec{y} = (y_0, \ldots, y_{\ell-1}) \in \mathbb{Z}^{\ell}$ such that $\|\vec{x}\|_{\infty}, \|\vec{y}\|_{\infty} < B$,*

– *For all* $N \in \mathsf{Supp}(\mathsf{RSA.Gen}(1^\lambda))$ *such that* $N^{\zeta-1} > 2^\lambda \ell B$,
– *For all* $g \in (\mathbb{Z}/N^\zeta\mathbb{Z})^\times$,
– *For all* $\vec{k} = (k_0, \ldots, k_{\ell-1}) \in [0, N)^\ell$, *and* $\vec{k'} = (k'_0, \ldots, k'_{\ell-1}) \in [0, N)^\ell$,
– *For all* $(x^{(0)}_{\mathsf{share}}, x^{(1)}_{\mathsf{share}}) \in \mathbb{Z}^2$ *such that* $x^{(1)}_{\mathsf{share}} - x^{(0)}_{\mathsf{share}} = \vec{k}^\mathsf{T} \cdot \vec{x}$,

$$\Pr\left[z_1 - z_0 = (\vec{k'})^\mathsf{T} \cdot \mathbf{\mathfrak{C}}_t(\vec{x}) \;\middle|\; \begin{array}{c} r \xleftarrow{\$} \mathbb{Z}/N^\zeta\mathbb{Z} \\ (z_0, z_1) \coloneqq \mathsf{ShareConv}_{\mathbf{\mathfrak{C}}_t}(N, g, \vec{k}, \vec{k'}, \vec{x}, x^{(0)}_{\mathsf{share}}, r) \end{array}\right] \geq 1 - 2^{-\lambda}$$

where the deterministic algorithm $\mathsf{ShareConv}_{\mathbf{\mathfrak{C}}_t}$ *is defined in Fig. 15.*

Algorithm $\mathsf{ShareConv}_{\mathbf{\mathfrak{C}}_t}$

$\mathsf{ShareConv}_{\mathbf{\mathfrak{C}}_t}(N, g, \vec{k}, \vec{k'}, \vec{x}, x^{(0)}_{\mathsf{share}}, r)$:

1. $x^{(1)}_{\mathsf{share}} \leftarrow \vec{k}^\mathsf{T} \cdot \vec{x} + x^{(0)}_{\mathsf{share}}$
2. *For* $i \in [0, \ell)$, $h_i \leftarrow g^{-k_i} \exp(k'_{i \oplus 2^t})$
3. $z_1 \leftarrow \left\lceil \mathsf{DDLog}\left(g^{x^{(1)}_{\mathsf{share}}} \prod_{i \in [0,\ell)} h_i^{x_i}\right) + r \right\rceil \bmod N^\zeta$
4. $z_0 \leftarrow \left\lceil \mathsf{DDLog}\left(g^{x^{(0)}_{\mathsf{share}}}\right) + r \right\rceil \bmod N^\zeta$
5. *Output* (z_0, z_1)

Fig. 15: Share conversion procedure for homomorphic bitflip-swap.

Proof. Observe that

$$\left(g^{x^{(1)}_{\mathsf{share}}} \prod_{i \in [0,\ell)} h_i^{x_i}\right) / \left(g^{x^{(0)}_{\mathsf{share}}}\right) = \left(g^{x^{(1)}_{\mathsf{share}} - x^{(0)}_{\mathsf{share}}}\right) g^{-(\vec{k})^\mathsf{T} \cdot \vec{x}} g^{\sum_{i=0}^{\ell-1} k_{i \oplus 2^t} x_i} = \exp\left((\vec{k})^\mathsf{T} \cdot \mathbf{\mathfrak{C}}_t(\vec{x})\right)$$

and so, by Lemma 2, $\mathsf{DDLog}\left(g^{x^{(1)}_{\mathsf{share}}} \prod_{i \in [0,\ell)} h_i^{x_i}\right) - \mathsf{DDLog}\left(g^{x^{(0)}_{\mathsf{share}}}\right) \equiv (\vec{k'})^\mathsf{T} \cdot \mathbf{\mathfrak{C}}_t(\vec{x}) \bmod N^\zeta$.
By Lemma 3, $\Pr[z_1 - z_0 = (\vec{k'})^\mathsf{T} \cdot \mathbf{\mathfrak{C}}_t(\vec{x})] \geq 1 - \frac{|(\vec{k'})^\mathsf{T} \cdot \mathbf{\mathfrak{C}}_t(\vec{x})|}{N^\zeta} \geq 1 - \frac{\ell N \|\mathbf{\mathfrak{C}}_t(\vec{x})\|_\infty}{N^\zeta} \geq 1 - \frac{\ell B}{N^{\zeta-1}} \geq 1 - 2^{-\lambda}$. □

E.3 Correctness of key/level-switching

Lemma 39 (Key-switching). *Let* λ *be a security parameter. Let* $B > 0$, $\zeta \in \mathbb{N} \setminus \{0, 1\}$, *and* $\ell \in \mathbb{N}^*$.

– *For all* $\vec{x} = (x_0, \ldots, x_{\ell-1}) \in \mathbb{Z}^\ell$ *and* $\vec{y} = (y_0, \ldots, y_{\ell-1}) \in \mathbb{Z}^\ell$ *such that* $\|\vec{x}\|_\infty, \|\vec{y}\|_\infty < B$,
– *For all* $N \in \mathsf{Supp}(\mathsf{RSA.Gen}(1^\lambda))$ *such that* $N^{\zeta-1} > 2^\lambda B \ell$,
– *For all* $g \in (\mathbb{Z}/N^\zeta\mathbb{Z})^\times$,
– *For all* $\vec{k} = (k_0, \ldots, k_{\ell-1}) \in [0, N)^\ell$, *and* $\vec{k'} = (k'_0, \ldots, k'_{\ell-1}) \in [0, N)^\ell$,
– *For all* $(x^{(0)}_{\mathsf{share}}, x^{(1)}_{\mathsf{share}}) \in \mathbb{Z}^2$ *such that* $x^{(1)}_{\mathsf{share}} - x^{(0)}_{\mathsf{share}} = \vec{k}^\mathsf{T} \cdot \vec{x}$,

$$\Pr\left[z_1 - z_0 = (\vec{k'})^\mathsf{T} \cdot \vec{x} \;\middle|\; \begin{array}{c} r \xleftarrow{\$} \mathbb{Z}/N^\zeta\mathbb{Z} \\ (z_0, z_1) \coloneqq \mathsf{ShareConv}_{\mathsf{KS}}(N, g, \vec{k}, \vec{k'}, \vec{x}, x^{(0)}_{\mathsf{share}}, r) \end{array}\right] \geq 1 - 2^{-\lambda}$$

where the deterministic algorithm $\mathsf{ShareConv}_{\mathsf{KS}}$ *is defined in Fig. 16.*

Algorithm $\mathsf{ShareConv}_{\mathsf{KS}}$

$\mathsf{ShareConv}_{\mathsf{KS}}(N, g, \vec{k}, \vec{k}', \vec{x}, x^{(0)}_{\mathsf{share}}, r)$:

1. $x^{(1)}_{\mathsf{share}} \leftarrow \vec{k}^{\mathsf{T}} \cdot \vec{x} + x^{(0)}_{\mathsf{share}}$
2. *For* $i \in [0, \ell)$, $h_i \leftarrow g^{-k_i} \exp(k'_i)$
3. $z_1 \leftarrow \left\lceil \mathsf{DDLog}\left(g^{x^{(1)}_{\mathsf{share}}} \prod_{i\in[0,\ell)} h_i^{x_i}\right) + r \right\rceil \bmod N^{\zeta}$
4. $z_0 \leftarrow \left\lceil \mathsf{DDLog}\left(g^{x^{(0)}_{\mathsf{share}}}\right) + r \right\rceil \bmod N^{\zeta}$
5. *Output* (z_0, z_1)

Fig. 16: Share conversion procedure for homomorphic key-switching.

Proof. Observe

$$\left(g^{x^{(1)}_{\mathsf{share}}} \prod_{i\in[0,\ell)} h_i^{x_i}\right) / \left(g^{x^{(0)}_{\mathsf{share}}}\right) = \left(g^{x^{(1)}_{\mathsf{share}} - x^{(0)}_{\mathsf{share}}}\right) g^{-(\vec{k})^{\mathsf{T}}\cdot\vec{x}} g^{(\vec{k}')^{\mathsf{T}}\cdot\vec{x}} = g^{(\vec{k}')^{\mathsf{T}}\cdot\vec{x}}$$

and so, by Lemma 2, $\mathsf{DDLog}\left(g^{x^{(1)}_{\mathsf{share}}} \prod_{i\in[0,\ell)} h_i^{x_i}\right) - \mathsf{DDLog}\left(g^{x^{(0)}_{\mathsf{share}}}\right) \equiv (\vec{k}')^{\mathsf{T}} \cdot \vec{x} \bmod N^{\zeta}$. By Lemma 3, $\Pr[z_1 - z_0 = (\vec{k}')^{\mathsf{T}} \cdot \vec{x}] \geq 1 - \frac{|(\vec{k}')^{\mathsf{T}}\cdot\vec{x}|}{N^{\zeta}} \geq 1 - \frac{\ell N \|\vec{x}\|_\infty}{N^{\zeta}} \geq 1 - \frac{\ell B}{N^{\zeta-1}} \geq 1 - 2^{-\lambda}$. □

E.4 Proof of Theorem Theorem 18

Theorem 40 (Theorem 18, restated). *Let* $B = \mathcal{O}(2^{\mathsf{poly}(\lambda)})$*, and let* biHSS *be a* $\mathbb{Z}$*-bilinear* $2^{\lambda}\ell B$*-bounded HSS scheme. Then the HSS construction defined by* $(\mathsf{HSS.crsGen}, \mathsf{HSS.KeyGen})$ *(Fig. 5),* $(\mathsf{HSS.HGateEval}, \mathsf{HSS.Dec})$ *(Figs. 6 and 11 to 13) is a correct HSS scheme for* B*-bundled levelled, bundled-then-RMS circuits (Definition 11). When instantiated with the key-switching graph in Section D and a computationally succinct* biHSS *where* $\mathsf{biHSS.Enc}$ *runs in* $O(\mathsf{poly}(\lambda, \log B) \cdot \ell^2$ *time, the computation of* $\mathsf{HSS.KeyGen}$ *and the size of* pp *are* $O(\mathsf{poly}(\lambda, \log|B|)\ell^2 D)$*, and the runtimes of* $\mathsf{HSS.sinct}, 0, \mathsf{vector}_1$*,* $\mathsf{HSS.Share}_{\mathsf{s\text{-}priv},0,\mathsf{vector}_1}$*,* $\mathsf{HSS.Share}_{\mathsf{priv}}$*, and* $\mathsf{HSS.HGateEval}(0, \ldots)$ *and the output size of* $\mathsf{HSS.Hash}_{\mathsf{vector}_1}$ *are all* $\mathsf{poly}(\lambda, \log|B|)$ *(independent of* ℓ*).*
Furthermore, assuming DCR and the strong encoder and hasher privacy properties of biHSS*, the sharing algorithms in Fig. 7 satisfy the correctness and privacy properties in Definitions 33 to 35.*

Proof. We prove each property separately.

Homomorphic correctness (gate-by-gate).

- Correctness of restriction multiplication gate evaluation (Fig. 11), *i.e.* gates $(1_{\mathsf{IN}}, 1_{\mathsf{REG}}, +_{\mathsf{REG}}, \times_{\mathsf{IN},v}, \times_{\mathsf{REG},c}, \times_{\mathsf{IN,REG}})$, has been established in *e.g.* [MORS25, Lemmata 59-63].
- Correctness of bundled wire gate evaluation (Fig. 6) is shown: in Lemma 37 for mult_d, in Lemma 38 for $\mathsf{bitflip\text{-}swap}_{t,d}$, and in Lemma 39 for $\mathsf{level\text{-}switch}_{d,d'}$.
- For vector-to-register conversion and additional linear operations (Fig. 12): $\mathsf{vector\text{-}to\text{-}REG}_d$ is a special case of mult_d where the component-wise multiplication is done with respect to the constant vector $(1, 0 \ldots, 0)$ (and so correctness follows from Lemma 37), while correctness of add_d, $\mathsf{scalar\text{-}mult}_{c,d}$, and $\mathsf{constant}_{\vec{v},d}$ immediately follow from inspection.

Correctness of offline-online semi-private sharing ($\mathsf{HSS.Share}_{0,\text{s-priv},\mathsf{vector}_1}$ *and* $\mathsf{HSS.Share}_{1,\text{s-priv},\mathsf{vector}_1}$*).* This follows from inspection.

Correctness of succinct semi-private sharing ($\mathsf{HSS.Hash}_{\mathsf{vector}_1}$, $\mathsf{HSS.Share}_{\mathsf{sinct},0,\mathsf{vector}_1}$ *and* $\mathsf{HSS.Share}_{\mathsf{sinct},1,\mathsf{vector}_1}$*).* This follows immediately from the correctness of bilinear HSS.

Correctness of public-key private sharing ($\mathsf{HSS.Share}_{\mathsf{priv},\mathsf{vector}_1}$*).* Immediate from correctness of $\mathsf{DJE.Enc}$ and $\mathsf{DJE.EncK}$ (Theorem 1).

Security of ($\mathsf{HSS.Share}_{0,\text{s-priv},\mathsf{vector}_1}$ *and* $\mathsf{HSS.Share}_{1,\text{s-priv},\mathsf{vector}_1}$*).* For $\sigma = 0$, we have $\mathsf{HSS.Share}_{0,\text{s-priv},\mathsf{vector}_1}$ already outputs $x^{(0)} \xleftarrow{\$} [0, 2^\lambda B\ell N)$, which is independently random from everything else. For $\sigma = 1$, it is statistically indistinguishable to sample $x^{(1)} \xleftarrow{\$} [0, 2^\lambda B\ell N)$ instead of $\vec{k}_1^\mathsf{T}\vec{x} + x^{(0)}$ and $x^{(0)} \xleftarrow{\$} [0, 2^\lambda B\ell N)$, as $|\vec{k}_1^\mathsf{T}\vec{x}| \leq NB\ell \leq 2^{-\lambda}(2^\lambda B\ell N)$. Therefore, sampling from $[0, 2^\lambda B\ell N)$ is a valid simulation in either case.

Hasher Privacy of $\mathsf{HSS.Hash}_{\mathsf{vector}_1}$ Immediate from Strong Hasher Privacy of biHSS.

Privacy of $\mathsf{HSS.Share}_{\mathsf{priv},\mathsf{IN}}$. We show a hybrid proof, starting with sharing x, and ending at a hybrid where the distinguisher's view is independent from x. Running the hybrids forwards on x and then backwards on y then proves the two distributions defining privacy of HSS are indistinguishable.

$\mathcal{H}_1$. If $\sigma = 1$:
Inside $\mathsf{HSS.KeyGen}$, replace $(E, \phi) \xleftarrow{\$} \mathsf{biHSS.Enc}(\mathsf{crs}, \vec{k}^{(0)})$ with $(E, \phi) \xleftarrow{\$} \mathsf{biHSS.Enc}(\mathsf{crs}, 0^\ell)$. This is indistinguishable by Strong Encoder Privacy of biHSS.

$\mathcal{H}_2$. If $\sigma = 1$:
Instead of sampling $\vec{k}^{(0)} \xleftarrow{\$} [0, N2^\lambda)$ and defining $\vec{k}^{(1)} \leftarrow \vec{k}^{(0)} + \vec{k}_1$, sample $\vec{k}^{(1)} \xleftarrow{\$} [0, N2^\lambda)$ and define $\vec{k}^{(0)} \leftarrow \vec{k}^{(1)} - \vec{k}_1$. This is identical to shifting the distribution of $\vec{k}^{(0)}$ by $-\vec{k}_1$, which has ∞-norm less than N. Therefore, the statical advantage is at most $\frac{\ell N}{N2^\lambda} \leq \ell 2^{-\lambda}$.

$\mathcal{H}_3$. If $\sigma = 1$:
Instead of sampling $K^{(0)} \xleftarrow{\$} [0, 2^\lambda N)$ and setting $K^{(1)} \leftarrow K + K^{(0)}$, sample $K^{(1)} \xleftarrow{\$} [0, 2^\lambda N)$ and set $K^{(0)} \leftarrow -K + K^{(1)}$. This is statistically indistinguishable for the same reason as for the previous hybrid.

$\mathcal{H}_4$. Delete $\vec{k}^{(1-\sigma)}$ and $K^{(1-\sigma)}$, as they no longer used when computing the adversary's view.

$\mathcal{H}_5$. Perform the following sequence of hybrids once for each $d \in [D]$, starting from $d = 1$ and continuing in order.

$\mathcal{H}_{5,d,1}$. At this point, $\vec{k}_d$ is only used in the exponent of $\boldsymbol{g}$ (and powers of it). This lets us use Lemma 44. For each d' such that $(d, d') \in E_{\mathsf{KS}}$, replace $\vec{\boldsymbol{h}}_{d\to d'} \leftarrow \boldsymbol{g}_{d\to d'}^{-\vec{k}_d} \exp(\vec{k}_{d'})$ with $\vec{\boldsymbol{h}}_{d\to d'} \leftarrow \boldsymbol{g}_{d\to d'}^{-\vec{k}_d}$. This is indistinguishable by reduction to Lemma 44: the challenge instance is placed into $\boldsymbol{g}_{d\to d'}$ and $\vec{\boldsymbol{h}}_{d\to d'}$, and all other usage of $\vec{k}_d$ can be simulated using $\boldsymbol{g}^{\vec{k}_d}$ by raising it to various powers. These powers include $r_{d\to d''}$ for other key-switching destinations d'', $r_{t,d}$ for bitflip-swap key-switching ciphertexts, and $\vec{k}_{d+1}^\mathsf{T}$ for multiplication gate key-switching ciphertexts.

$\mathcal{H}_{5,d,2}$. For each $t \in [0, \log \ell)$, replace $\vec{\boldsymbol{h}}_{t,d} \leftarrow \boldsymbol{g}_t^{-\vec{k}_d} \exp(\boldsymbol{\circlearrowright}_t(\vec{k}_{d+1}))$ with $\vec{\boldsymbol{h}}_{t,d} \leftarrow \boldsymbol{g}_t^{-\vec{k}_d}$. This is indistinguishable by a similar argument to the previous hybrids.

$\mathcal{H}_{5,d,3}$. Replace $\boldsymbol{H}_d \leftarrow \boldsymbol{g}^{\vec{k}_d\vec{k}_{d+1}^\mathsf{T}} \exp(\mathsf{diag}(\vec{k}_{d+1}))$ with sampling a vector $\vec{s} \xleftarrow{\$} [0, N^\zeta)^\ell$ and setting $\boldsymbol{H}_d \leftarrow \boldsymbol{g}^{\vec{k}_d\vec{k}_{d+1}^\mathsf{T}} \exp(\mathsf{diag}(\vec{s}))$. This is indistinguishable by Lemma 43.
After this change, $\vec{k}_{d+1}$ is only used in the exponent of $\boldsymbol{g}$ (and powers of it), preserving the loop invariant.

$\mathcal{H}_6$. Use IND-CPA security of $\mathsf{DJE.Enc}$ (Lemma 41) to replace $c \xleftarrow{\$} \mathsf{DJE.Enc}_{(\boldsymbol{g},\boldsymbol{h},N)}(x)$ in $\mathsf{HSS.Share}_{\mathsf{priv,IN}}$ with $c \xleftarrow{\$} \mathsf{DJE.Enc}_{(\boldsymbol{g},\boldsymbol{h},N)}(0)$. Recall that $\boldsymbol{h} \leftarrow \vec{\boldsymbol{h}}_D[0]$. Since $\vec{k}_d$ is only ever used in the exponent of $\boldsymbol{g}$ (or a power of $\boldsymbol{g}$), the rest of the adversary's view can be simulated using just a DJE public key, without access to $\vec{k}_d[0]$. That is, it is sufficient to have $g^{\vec{k}_d[0]} = (\vec{\boldsymbol{h}}_D[0])^{-1} = \boldsymbol{h}^{-1}$. Therefore, we can reduce to IND-CPA security.

$\mathcal{H}_7$. Use IND-CPA security of $\mathsf{DJE.EncK}$ (Lemma 41) to replace $c^{\mathsf{auth}} \xleftarrow{\$} \mathsf{DJE.EncK}_{(\boldsymbol{g},\boldsymbol{h},N)}(x)$ in $\mathsf{HSS.Share}_{\mathsf{priv,IN}}$ with $c^{\mathsf{auth}} \xleftarrow{\$} \mathsf{DJE.EncK}_{(\boldsymbol{g},\boldsymbol{h},N)}(0)$. The reduction is similar to the previous hybrid.
After this change, x is no longer used, so the hybrid proof is complete.

Communication and computational complexities. The runtime of $\mathsf{HSS.KeyGen}$ is dominated by computing – and the size of pp is dominated by storing – the key-switching ciphertexts $\boldsymbol{H}_d$, which consist of $\ell^2 D$ group elements. That is, the call to $\mathsf{biHSS.Enc}$ takes at most $O(\mathsf{poly}(\lambda, \log B) \cdot \ell^2$ time (and output size), and the other costs grow subquadratically with ℓ. The runtimes of $\mathsf{HSS.Share}_{\mathsf{s\text{-}priv},0,\mathsf{vector}_1}$, $\mathsf{HSS.Share}_{\mathsf{priv}}$, and $\mathsf{HSS.HGateEval}(0,\ldots)$ are all trivially independent of ℓ. The runtime of $\mathsf{HSS.sinct}, 0, \mathsf{vector}_1$ and the output size of $\mathsf{HSS.Hash}_{\mathsf{vector}_1}$ are both independent of ℓ by the computational succinctness of biHSS. □

F DCR Reductions

F.1 DJE Encryption

Lemma 41 ([BMO+25, Theorem 2]). *Damgård–Jurik–ElGamal is KDM-secure for affine functions of the form $f_{x,y} : \mathsf{sk} \mapsto x \cdot \mathsf{sk} + y$, where $a, b \in \mathbb{Z}/N^\zeta\mathbb{Z}$. That is, for all* p.p.t. *adversaries $\mathcal{A}$, the oracles $\mathcal{O}^{\mathsf{KDM}}_{\mathsf{sk},\mathsf{pk},R}$ and $\mathcal{O}^{\mathsf{KDM}}_{\mathsf{sk},\mathsf{pk},\$}$ are indistinguishable in the following experiment*

$$\boxed{\begin{array}{l}(\mathsf{sk},\mathsf{pk}) \xleftarrow{\$} \mathsf{DJE.KeyGen}(1^\lambda)\\ \text{output } \mathcal{A}^{\mathcal{O}^{\mathsf{KDM}}_{\mathsf{sk},\mathsf{pk},R/\$}}(\mathsf{pk})\end{array}}$$

where these oracles are defined as

$$\boxed{\begin{array}{l}\mathcal{O}^{\mathsf{KDM}}_{\mathsf{sk},\mathsf{pk},R}(x,y)\text{:}\\ \quad (k,N) \leftarrow \mathsf{sk}\\ \quad z \leftarrow x \cdot k + y\\ \quad \text{return } \mathsf{DJE.Enc}_{\mathsf{pk}}(z)\end{array}} \qquad \boxed{\begin{array}{l}\mathcal{O}^{\mathsf{KDM}}_{\mathsf{sk},\mathsf{pk},\$}(x,y)\text{:}\\ \quad (k,N) \leftarrow \mathsf{sk}\\ \quad z \xleftarrow{\$} \mathbb{Z}/N^\zeta\mathbb{Z}\\ \quad \text{return } \mathsf{DJE.Enc}_{\mathsf{pk}}(z)\end{array}}$$

Proof of Theorem 1. We prove the four properties separately.

Correctness of Enc. Parse $\mathsf{pk} = (\boldsymbol{g}, \boldsymbol{h}, N)$. Then,

$$\begin{aligned}\mathsf{DJE.Dec}_{\mathsf{sk}}(\mathsf{DJE.Enc}_{\mathsf{pk}}(x)) &= \mathsf{DJE.Dec}_{\mathsf{sk}}(\boldsymbol{g}^r, \boldsymbol{h}^r \cdot \exp(x))\\ &= \log(\boldsymbol{g}^{rk} \cdot \boldsymbol{h}^r \cdot \exp(x)) = \log((\boldsymbol{g}^k \cdot \boldsymbol{h})^r \cdot \exp(x)) = x.\end{aligned}$$

Correctness of EncK. Parse $\mathsf{pk} = (\boldsymbol{g}, \boldsymbol{h}, N)$. Then,

$$\begin{aligned}\mathsf{DJE.Dec}_{\mathsf{sk}}(\mathsf{DJE.Enc}_{\mathsf{pk}}(x)) &= \mathsf{DJE.Dec}_{\mathsf{sk}}(\boldsymbol{g}^r \cdot \exp(x), \boldsymbol{h}^r)\\ &= \log(\boldsymbol{g}^{rk} \cdot \exp(x)^k \cdot \boldsymbol{h}^r) = \log((\boldsymbol{g}^k \cdot \boldsymbol{h})^r \cdot \exp(kx)) = kx.\end{aligned}$$

IND-CPA Security of Enc. Immediate from KDM security of Enc (Lemma 41): the real encryption is indistinguishable from a simulation $\mathcal{O}^{\mathsf{KDM}}_{\mathsf{sk},\mathsf{pk},\$}$ that is independent of which message (y_0 or y_1) gets encrypted.

IND-CPA Security of EncK. We give a hybrid proof, starting from $\mathsf{EncK}(x)$ and ending at a hybrid that is independent of x. Running the hybrids forwards on x_0 and then backwards on x_1 then proves that $\mathsf{EncK}(x_0)$ and $\mathsf{EncK}(x_1)$ are indistinguishable (i.e., IND-CPA security).

$\mathcal{H}_1$. Change $\mathsf{EncK}(x)$ to $\mathsf{Enc}(0) \cdot (\exp(x), 1)$, which is identically distributed.

$\mathcal{H}_2$. Use KDM security of Enc (Lemma 41) to change this to $\mathsf{Enc}(z) \cdot (\exp(x), 1)$ for uniform z. The ciphertext is now

$$(\boldsymbol{g}^r \cdot \exp(x), \boldsymbol{h}^r \exp(z))$$

$\mathcal{H}_3$. Use DCR to change $\boldsymbol{g}$ to be sampled from the order $\varphi(N)$ subgroup of perfect N^ζ powers.

$\mathcal{H}_4$. Sample $r \leftarrow [0, 2^\lambda N^{\zeta+1})$ instead of $r \leftarrow [0, N)$. This is statistically indistinguishable because both are statistically close to uniform modulo the order of $\boldsymbol{g}$ (which is a factor of $\varphi(N)$). For the former, the statistical distance from uniform is at most $\varphi(N)2^{-\lambda}N^{-\zeta-1} \leq 2^{-\lambda}$ The latter has distance at most $(N \bmod \varphi(N))/N = (p+q-1)/N = O(N^{-\frac{1}{2}})$ from uniform.

$\mathcal{H}_5$. Replace $\boldsymbol{g}$ with $\boldsymbol{g}\exp(1)$. This is indistinguishable by DCR. The ciphertext is now

$$(\boldsymbol{g}^r \cdot \exp(r+x), \boldsymbol{h}^r \exp(-kr+z))$$

$\mathcal{H}_6$. Separate r into two variables: $r_1 \xleftarrow{\$} [0, N)$ and $r_2 \xleftarrow{\$} \mathbb{Z}/N^\zeta\mathbb{Z}$, and use CRT to set r such that $r \equiv r_1 \mod \varphi(N)$ and $r \equiv r_2 \mod N^\zeta$. This CRT trick is valid because $\varphi(N)$ and N are coprime, as p and q are of similar sizes. This change is indistinguishable because r was $2^{-\lambda}$-close to uniform modulo $\varphi(N^{\zeta+1}) < N^{\zeta+1}$, while r_2 is now uniform modulo N^ζ and r_1 is $O(N^{-\frac{1}{2}})$-close to uniform modulo $\varphi(N)$. The ciphertext is now

$$(\boldsymbol{g}^{r_1} \cdot \exp(r_2+x), \boldsymbol{h}^{r_1} \exp(-kr_2+z))$$

$\mathcal{H}_7$. Let $z' = -kr_2 + z$, and sample z' uniformly instead of z.

$\mathcal{H}_8$. Let $r_2' = r_2 + x$, and sample r_2' uniformly instead of r_2.
The ciphertext is now

$$(\boldsymbol{g}^{r_1} \cdot \exp(r_2'), \boldsymbol{h}^{r_1} \exp(z'))$$

Notice that this is independent of x, so we are done. □

F.2 Privacy of Key Switching Gadget Ciphertexts

Lemma 42 ([MORS25, Lemma 4]). *Let* $N \xleftarrow{\$} \mathsf{RSA.Gen}(1^\lambda)$, $\boldsymbol{g} \xleftarrow{\$} (\mathbb{Z}/N^{\zeta+1}\mathbb{Z})^\times$, $x, y \xleftarrow{\$} [0, N)$ *and* $r \xleftarrow{\$} [0, N^\zeta)$. *Then, the distributions*

$$(N, \boldsymbol{g}, \boldsymbol{g}^x, \boldsymbol{g}^y, \boldsymbol{g}^{xy} \cdot \exp(y)) \qquad \textit{and} \qquad (N, \boldsymbol{g}, \boldsymbol{g}^x, \boldsymbol{g}^y, \boldsymbol{g}^{xy} \cdot \exp(r))$$

are computationally indistinguishable under the DCR assumption.

Lemma 43 (Bilinear Key Switching Ciphertexts). *Let* $N \xleftarrow{\$} \mathsf{RSA.Gen}(1^\lambda)$, $\boldsymbol{g} \xleftarrow{\$} (\mathbb{Z}/N^{\zeta+1}\mathbb{Z})^\times$, $\vec{x}, \vec{y} \xleftarrow{\$} [0, N)^\ell$, *and* $\vec{r} \xleftarrow{\$} [0, N^\zeta)^\ell$. *Then, the distributions*

$$(N, \boldsymbol{g}, \boldsymbol{g}^{\vec{x}}, \boldsymbol{g}^{\vec{y}}, \boldsymbol{g}^{\vec{x}\vec{y}^\intercal} \cdot \exp(\mathsf{diag}(\vec{y}))) \qquad \textit{and} \qquad (N, \boldsymbol{g}, \boldsymbol{g}^{\vec{x}}, \boldsymbol{g}^{\vec{y}}, \boldsymbol{g}^{\vec{x}\vec{y}^\intercal} \cdot \exp(\mathsf{diag}(\vec{r})))$$

are computationally indistinguishable under the DCR assumption.

Proof. We present a hybrid proof, where in the ith hybrid distribution the $\exp(\mathsf{diag}(\vec{y}))$ term becomes $\exp(\mathsf{diag}(\vec{r}_1, \ldots, \vec{r}_i, \vec{y}_{i+1}, \ldots, \vec{y}_\ell))$. Hybrid 0 is the same as the first distribution from the lemma statement, and hybrid ℓ is the same as the second distribution. The difference from hybrid $i-1$ to hybrid i is just that $\vec{y}_i$ gets replaced with $\vec{r}_i$.

We now give a reduction to Lemma 42 to show that each hybrid i is indistinguishable from the previous hybrid $i-1$. First, receive a tuple $(N, \boldsymbol{g}, \boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w})$ from the indistinguishability game of Lemma 42. Sample $\vec{x}_T, \vec{y}_T \leftarrow [0,N)^{i-1}$, $\vec{x}_B, \vec{y}_B \leftarrow [0,N)^{\ell-i}$, and $\vec{r} \leftarrow [0, N^\zeta)^{i-1}$. Then run the distinguisher on $(N, \boldsymbol{g}, \vec{\boldsymbol{u}}, \vec{\boldsymbol{v}}, \boldsymbol{W})$ and return its output, where

$$\vec{\boldsymbol{u}} = \begin{bmatrix} \boldsymbol{g}^{\vec{x}_T} \\ \boldsymbol{u} \\ \boldsymbol{g}^{\vec{x}_B} \end{bmatrix} \qquad \vec{\boldsymbol{v}} = \begin{bmatrix} \boldsymbol{g}^{\vec{y}_T} \\ \boldsymbol{v} \\ \boldsymbol{g}^{\vec{y}_B} \end{bmatrix} \qquad \boldsymbol{W} = \begin{bmatrix} \boldsymbol{g}^{\vec{x}_T \vec{y}_T^\intercal} \exp(\mathsf{diag}(\vec{r})) & \boldsymbol{v}^{\vec{x}_T} & \boldsymbol{g}^{\vec{x}_T \vec{y}_B^\intercal} \\ \boldsymbol{u}^{\vec{y}_T^\intercal} & \boldsymbol{w} & \boldsymbol{u}^{\vec{y}_B^\intercal} \\ \boldsymbol{g}^{\vec{x}_B \vec{y}_T^\intercal} & \boldsymbol{v}^{\vec{x}_B} & \boldsymbol{g}^{\vec{x}_B \vec{y}_B^\intercal} \exp(\mathsf{diag}(\vec{y}_B)) \end{bmatrix}. \tag{4}$$

If the game provides a sample from the first distribution in Lemma 42, then the reduction will behave identically to the hybrid $i-1$ distribution. That is, plugging in $\boldsymbol{u} = \boldsymbol{g}^x$, $\boldsymbol{v} = \boldsymbol{g}^y$, and $\boldsymbol{w} = \boldsymbol{g}^{xy} \exp(y)$ into (4) is the same as setting

$$\vec{x} = \begin{bmatrix} \vec{x}_T \\ x \\ \vec{x}_B \end{bmatrix} \qquad \vec{y} = \begin{bmatrix} \vec{y}_T \\ y \\ \vec{y}_B \end{bmatrix}$$

in hybrid $i-1$. Similarly, if the reduction receives a sample from the second distribution then it will behave identically to the hybrid i distribution, as w is now $\boldsymbol{g}^{xy} \exp(r)$, which plugs a random value into W_{ii}. □

Lemma 44 (Linear Key Switching Ciphertexts). *Let* $N \xleftarrow{\$} \mathsf{RSA.Gen}(1^\lambda)$, $\boldsymbol{g} \xleftarrow{\$} (\mathbb{Z}/N^{\zeta+1}\mathbb{Z})^\times$, $r \xleftarrow{\$} [0,N)$, $\vec{x} \xleftarrow{\$} [0,N)^\ell$. *Then, for any vector* $\vec{y} \in [0, N^\zeta)^\ell$, *the distributions*

$$(N, \boldsymbol{g}, \boldsymbol{g}^{\vec{x}}, \boldsymbol{g}^r, \boldsymbol{g}^{-r\vec{x}} \cdot \exp(\mathsf{diag}(\vec{y}))) \qquad \textit{and} \qquad (N, \boldsymbol{g}, \boldsymbol{g}^{\vec{x}}, \boldsymbol{g}^r, \boldsymbol{g}^{-r\vec{x}}$$

are computationally indistinguishable under the DCR assumption.

Proof. We present an ℓ-step hybrid proof. In the ith hybrid distribution the $\exp(\mathsf{diag}(\vec{y}))$ term becomes $\exp(\mathsf{diag}(0, \ldots, 0, \vec{y}_{i+1}, \ldots, \vec{y}_\ell))$, which makes the real distribution the same as hybrid 0 and the random distribution the same as hybrid 1. In the transition from hybrid $i-1$ to hybrid i, $\vec{y}_i$ gets swapped out for 0.

We now show that this change is indistinguishable, by reduction to the IND-CPA security of $\mathsf{DJE.Enc}$ (Theorem 1). The reduction receives a public key $\mathsf{pk} = (\boldsymbol{g}, \boldsymbol{h}, N)$ from the challenger. Replace $\boldsymbol{g}^{\vec{x}_i}$ with $\boldsymbol{h}^{-1}$, essentially setting $\vec{x}_i = k$. Query the challenger for an encryption $(\boldsymbol{c}_0, \boldsymbol{c}_1)$ of either $\vec{y}_i$ or 0; our goal is to distinguish which one the challenger encrypts. Replace $\boldsymbol{g}^r$ with $\boldsymbol{c}_0$ and $\boldsymbol{g}^{-r\vec{x}_i} \exp(\vec{y}_i)$ with $\boldsymbol{c}_1$. Then the tuple output by the reduction will exactly the match hybrid $i-1$ if the challenger chooses $\vec{y}_i$ to be the plaintext, and will exactly match hybrid i if the challenger chooses 0 to be the plaintext. □

G Details on LFE Definition

Definition 45 (AB-LFE: Correctness, Security and Succinctness).

- **Correctness:** *For every* params *and PPT adversary* $\mathcal{A}$,

$$\Pr\left[\begin{array}{c} C(x) = 1 \\ \wedge \\ m' \neq m \end{array} \;\middle|\; \begin{array}{r} \mathsf{crs} \xleftarrow{\$} \mathsf{crsGen}(1^\lambda, \mathsf{params}) \\ (\mathsf{view}, C) \xleftarrow{\$} \mathcal{A}_1(1^\lambda, \mathsf{crs}) \\ (h, \psi) \xleftarrow{\$} \mathsf{Compress}(\mathsf{crs}, C) \\ x, m \xleftarrow{\$} \mathcal{A}_2(\mathsf{view}, h, \psi) \\ \mathsf{ct} \xleftarrow{\$} \mathsf{Enc}(\mathsf{crs}, h, x, m) \\ m' \xleftarrow{\$} \mathsf{Dec}(\mathsf{crs}, C, x, \mathsf{ct}, \psi) \end{array}\right] \leq \mathsf{negl}(\lambda).$$

- **Security:** *For every* p.p.t. *adversary* $\mathcal{A}$*, the following distributions are indistinguishable.*

$\mathsf{crs} \xleftarrow{\$} \mathsf{crsGen}(1^\lambda, \mathsf{params})$ $(\mathsf{view}, C) \xleftarrow{\$} \mathcal{A}_1(1^\lambda, \mathsf{crs})$ $(h, \psi) \xleftarrow{\$} \mathsf{Compress}(\mathsf{crs}, C)$ $(\mathsf{view}', x, m_0, m_1) \xleftarrow{\$} \mathcal{A}_2(\mathsf{view}, h, \psi)$ abort if $C(x) \neq 0$ $\mathsf{ct} \xleftarrow{\$} \mathsf{Enc}(\mathsf{crs}, h, x, m_0)$ return $(\mathsf{view}', \mathsf{ct})$	$\mathsf{crs} \xleftarrow{\$} \mathsf{crsGen}(1^\lambda, \mathsf{params})$ $(\mathsf{view}, C) \xleftarrow{\$} \mathcal{A}_1(1^\lambda, \mathsf{crs})$ $(h, \psi) \xleftarrow{\$} \mathsf{Compress}(\mathsf{crs}, C)$ $(\mathsf{view}', x, m_0, m_1) \xleftarrow{\$} \mathcal{A}_2(\mathsf{view}, h, \psi)$ abort if $C(x) \neq 0$ $\mathsf{ct} \xleftarrow{\$} \mathsf{Enc}(\mathsf{crs}, h, x, m_1)$ return $(\mathsf{view}', \mathsf{ct})$

- **Function Privacy:** *For every* p.p.t. *adversary* $\mathcal{A}$*, the following distributions are indistinguishable.*

$\mathsf{crs} \xleftarrow{\$} \mathsf{crsGen}(1^\lambda, \mathsf{params})$ $(\mathsf{view}, C, C') \xleftarrow{\$} \mathcal{A}(1^\lambda, \mathsf{crs})$ $(h, \psi) \xleftarrow{\$} \mathsf{Compress}(\mathsf{crs}, C)$ return (view, h)	$\mathsf{crs} \xleftarrow{\$} \mathsf{crsGen}(1^\lambda, \mathsf{params})$ $(\mathsf{view}, C, C') \xleftarrow{\$} \mathcal{A}(1^\lambda, \mathsf{crs})$ $(h, \psi) \xleftarrow{\$} \mathsf{Compress}(\mathsf{crs}, C')$ return (view, h)

An AB-LFE scheme may have the following succinctness properties:

- **Circuit Succinctness:** *The sizes of* crs*,* h*, and* ct*, and the runtime of* Enc*, must all be sublinear in the circuit size of* C*.*
- **Attribute Succinctness:** *The size* ct *is sublinear in the size of the attribute* x*.*

H Proof of Theorem 19

Theorem 19 (Function-Private AB-LFE from HSS). *Let* $B = \mathcal{O}(2^{\mathsf{poly}(\lambda)})$*. The construction of Fig. 8 is a function-private attribute-based laconic function evaluation scheme for the class of* B*-bounded arithmetic circuits.*

Proof. – *Correctness.* Let $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ be a pair of *p.p.t.* adversaries. Consider the probability experiment of AB-LFE correctness, and the event $X\colon C(x) = 1 \wedge m' \neq m$. Our goal is to show that $\Pr[X] \leq \mathsf{negl}(\lambda)$. Without loss of generality, we can assume $\mathcal{A}$ only outputs correctly formatted values x and C, as otherwise one of the algorithms LFE.Compress, LFE.Enc, or LFE.Dec would return $\perp$ and the event X cannot occur. This is the experiment:

$\mathsf{crs} \xleftarrow{\$} \mathsf{LFE.crsGen}(1^\lambda, \mathsf{params})$
$(\mathsf{view}, C) \xleftarrow{\$} \mathcal{A}_1(1^\lambda, \mathsf{crs})$
$(h, \psi) \xleftarrow{\$} \mathsf{LFE.Compress}(\mathsf{crs}, C)$
$x, m \xleftarrow{\$} \mathcal{A}_2(\mathsf{view}, h, \psi)$
$\mathsf{ct} \xleftarrow{\$} \mathsf{LFE.Enc}(\mathsf{crs}, h, x, m)$
$m' \xleftarrow{\$} \mathsf{LFE.Dec}(\mathsf{crs}, C, x, \mathsf{ct}, \psi)$
$X := \text{“}C(x) = 1 \wedge m' \neq m''$

The experiment remains unchanged if we inline the definitions of LFE.crsGen, LFE.Compress, LFE.Enc, and LFE.Dec (and reorder some of the lines of code).

$\mathsf{crs}_{\mathsf{HSS}} \xleftarrow{\$} \mathsf{HSS.crsGen}(1^\lambda)$
$(\mathsf{pp}, \mathsf{sk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \xleftarrow{\$} \mathsf{HSS.KeyGen}(1^\lambda, \mathsf{crs}_{\mathsf{HSS}})$
$(\mathsf{view}, C) \xleftarrow{\$} \mathcal{A}_1(1^\lambda, \mathsf{crs})$
$\widetilde{C} \leftarrow \mathsf{ConvertCircuit}(C)$
$(h, \psi) \xleftarrow{\$} \mathsf{HSS.Hash}(\widetilde{C})$
$x, m \xleftarrow{\$} \mathcal{A}_2(\mathsf{view}, h, \psi)$
$\tilde{x} \leftarrow \mathsf{ConvertInputs}(x)$
$\widetilde{C}_0 \xleftarrow{\$} \mathsf{Share}_{\mathsf{sinct},0}(\mathsf{pp}, \mathsf{ek}_0, h)$
$(\mu_0, \mu_1) \xleftarrow{\$} \mathsf{HSS.Share}_{\mathsf{priv}}(\mathsf{pp}, \mu)$
$r \xleftarrow{\$} \{0,1\}^R$
$y_0 \xleftarrow{\$} \mathsf{HSS.HCircuitEval}(0, \mathsf{pp}, \mathsf{ek}_0, \mathsf{UC\text{-}then\text{-}mult}_{\tilde{x}}, (\widetilde{C}_0, \mu_0); r)$
$z_0 \xleftarrow{\$} \mathsf{HSS.Dec}^{(0)}(\mathsf{pp}, \mathsf{ek}_0, y_0)$
$\widetilde{C}_1 \xleftarrow{\$} \mathsf{HSS.Share}_{\mathsf{sinct},1}(\mathsf{pp}, \mathsf{ek}_1, \widetilde{C}, \psi)$
$y_1 \xleftarrow{\$} \mathsf{HSS.HCircuitEval}(1, \mathsf{pp}, \mathsf{ek}_1, \mathsf{UC\text{-}then\text{-}mult}_{\tilde{x}}, (\widetilde{C}_1, \mu_1); r)$
$z_1 \xleftarrow{\$} \mathsf{HSS.Dec}^{(1)}(\mathsf{pp}, \mathsf{ek}_1, y_1)$
$z \leftarrow z_1 - z_0$
$X :=$ "$C(x) = 1 \wedge z \neq \mu$

By additive decoding (Definition 17), we can replace the highlighted lines above by "$z \leftarrow \mathsf{HSS.Dec}(\mathsf{pp}, \mathsf{sk}, \mathsf{REG}, y_0, y_1)$" without noticeably changing the probability of X. We are left with the following experiment:

$\mathsf{crs}_{\mathsf{HSS}} \xleftarrow{\$} \mathsf{HSS.crsGen}(1^\lambda)$
$(\mathsf{pp}, \mathsf{sk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \xleftarrow{\$} \mathsf{HSS.KeyGen}(1^\lambda, \mathsf{crs}_{\mathsf{HSS}})$
$(\mathsf{view}, C) \xleftarrow{\$} \mathcal{A}_1(1^\lambda, \mathsf{crs})$
$\widetilde{C} \leftarrow \mathsf{ConvertCircuit}(C)$
$(h, \psi) \xleftarrow{\$} \mathsf{HSS.Hash}(\widetilde{C})$
$x, m \xleftarrow{\$} \mathcal{A}_2(\mathsf{view}, h, \psi)$
$\tilde{x} \leftarrow \mathsf{ConvertInputs}(x)$
$\widetilde{C}_0 \xleftarrow{\$} \mathsf{Share}_{\mathsf{sinct},0}(\mathsf{pp}, \mathsf{ek}_0, h)$
$(\mu_0, \mu_1) \xleftarrow{\$} \mathsf{HSS.Share}_{\mathsf{priv}}(\mathsf{pp}, \mu)$
$r \xleftarrow{\$} \{0,1\}^R$
$y_0 \xleftarrow{\$} \mathsf{HSS.HCircuitEval}(0, \mathsf{pp}, \mathsf{ek}_0, \mathsf{UC\text{-}then\text{-}mult}_{\tilde{x}}, (\widetilde{C}_0, \mu_0); r)$
$\widetilde{C}_1 \xleftarrow{\$} \mathsf{HSS.Share}_{\mathsf{sinct},1}(\mathsf{pp}, \mathsf{ek}_1, \widetilde{C}, \psi)$
$y_1 \xleftarrow{\$} \mathsf{HSS.HCircuitEval}(1, \mathsf{pp}, \mathsf{ek}_1, \mathsf{UC\text{-}then\text{-}mult}_{\tilde{x}}, (\widetilde{C}_1, \mu_1); r)$
$z \leftarrow \mathsf{HSS.Dec}(\mathsf{pp}, \mathsf{sk}, \mathsf{REG}, y_0, y_1)$
$X :=$ "$C(x) = 1 \wedge z \neq \mu$

Observe (focusing on the highlighted lines) that, by *circuit homomomorphism correctness* (Lemma 32), we can switch the line "$z \leftarrow \mathsf{HSS.Dec}(\mathsf{pp}, \mathsf{sk}, \mathsf{REG}, y_0, y_1)$" to the following three:

$y'_1 \xleftarrow{\$} \mathsf{HSS.Dec}(\mathsf{pp}, \mathsf{sk}, \mathsf{vector}_0, \tilde{C}_0, \tilde{C}_1)$
$y'_2 \xleftarrow{\$} \mathsf{HSS.Dec}(\mathsf{pp}, \mathsf{sk}, \mathsf{IN}, \mu_0, \mu_1)$
$z \leftarrow \mathsf{CircuitEval}(\mathsf{UC\text{-}then\text{-}mult}_{\tilde{x}}, y'_1, y'_2)$

while only changing the probability of X by a negligible amount. After removing the (now unused) lines defining $r, y_0, y_1, z_0, y_1, z_1$, we are left with the following probability experiment:

$\mathsf{crs}_{\mathsf{HSS}} \xleftarrow{\$} \mathsf{HSS.crsGen}(1^\lambda)$
$(\mathsf{pp}, \mathsf{sk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \xleftarrow{\$} \mathsf{HSS.KeyGen}(1^\lambda, \mathsf{crs}_{\mathsf{HSS}})$
$(\mathsf{view}, C) \xleftarrow{\$} \mathcal{A}_1(1^\lambda, \mathsf{crs})$
$\widetilde{C} \leftarrow \mathsf{ConvertCircuit}(C)$
$(h, \psi) \xleftarrow{\$} \mathsf{HSS.Hash}(\widetilde{C})$
$x, m \xleftarrow{\$} \mathcal{A}_2(\mathsf{view}, h, \psi)$
$\tilde{x} \leftarrow \mathsf{ConvertInputs}(x)$
$\widetilde{C}_0 \xleftarrow{\$} \mathsf{Share}_{\mathsf{sinct},0}(\mathsf{pp}, \mathsf{ek}_0, h)$
$(\mu_0, \mu_1) \xleftarrow{\$} \mathsf{HSS.Share}_{\mathsf{priv}}(\mathsf{pp}, \mu)$
$\widetilde{C}_1 \xleftarrow{\$} \mathsf{HSS.Share}_{\mathsf{sinct},1}(\mathsf{pp}, \mathsf{ek}_1, \widetilde{C}, \psi)$
$y'_1 \xleftarrow{\$} \mathsf{HSS.Dec}(\mathsf{pp}, \mathsf{sk}, \mathsf{vector}_0, \tilde{C}_0, \tilde{C}_1)$
$y'_2 \xleftarrow{\$} \mathsf{HSS.Dec}(\mathsf{pp}, \mathsf{sk}, \mathsf{IN}, \mu_0, \mu_1)$
$z \leftarrow \mathsf{CircuitEval}(\mathsf{UC\text{-}then\text{-}mult}_{\tilde{x}}, y'_1, y'_2)$
$X :=$ "$C(x) = 1 \wedge z \neq \mu$

By correctness of the succinct semi-private sharing (Definition 34), the experiment's outcome changes negligibly if if we switch "$y'_1 \xleftarrow{\$} \mathsf{HSS.Dec}(\mathsf{pp}, \mathsf{sk}, \mathsf{vector}_0, \tilde{C}_0, \tilde{C}_1)$" to "$y'_1 \leftarrow \widetilde{C}$". Similarly, by correctness of the public-key private sharing (Definition 33), there is again no noticeable change if we switch "$y'_2 \xleftarrow{\$} \mathsf{HSS.Dec}(\mathsf{pp}, \mathsf{sk}, \mathsf{IN}, \mu_0, \mu_1)$" to "$y'_2 \leftarrow \mu$. Finally, by correctness of UC, we can switch "$z \leftarrow \mathsf{CircuitEval}(\mathsf{UC\text{-}then\text{-}mult}_{\tilde{x}}, y'_1, y'_2)$" to $z \leftarrow C(x)\ \mu$. Clearly, $\Pr[X] = 0$ in this last experiment. Therefore, $\Pr[X] \leq \mathsf{negl}(\lambda)$ int the original LFE correctness experiment.

– *Security.* Let $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ be a *p.p.t.* adversary.

1. Our starting point is the first experiment of the AB-LFE security game.

$\mathsf{crs} \xleftarrow{\$} \mathsf{LFE.crsGen}(1^\lambda, \mathsf{params})$
$(\mathsf{view}, C) \xleftarrow{\$} \mathcal{A}_1(1^\lambda, \mathsf{crs})$
$(h, \psi) \xleftarrow{\$} \mathsf{LFE.Compress}(\mathsf{crs}, C)$
$(\mathsf{view}', x, m_0, m_1) \xleftarrow{\$} \mathcal{A}_2(\mathsf{view}, h, \psi)$
abort if $C(x) \neq 0$
$\mathsf{ct} \xleftarrow{\$} \mathsf{LFE.Enc}(\mathsf{crs}, h, x, m_0)$
return $(\mathsf{view}', \mathsf{ct})$

2. We then inline the definition of $\mathsf{LFE.crsGen}$, $\mathsf{LFE.Compress}$, and $\mathsf{LFE.Enc}$.

$r \xleftarrow{\$} \{0,1\}^R$
$\mathsf{crs}_{\mathsf{HSS}} \xleftarrow{\$} \mathsf{HSS.crsGen}(1^\lambda)$
$(\mathsf{view}, C) \xleftarrow{\$} \mathcal{A}_1(1^\lambda, \mathsf{crs})$
$\widetilde{C} \leftarrow \mathsf{ConvertCircuit}(C)$
$(h, \psi) \xleftarrow{\$} \mathsf{HSS.Hash}(\widetilde{C})$
$(\mathsf{view}', x, m_0, m_1) \xleftarrow{\$} \mathcal{A}_2(\mathsf{view}, h, \psi)$
abort if $C(x) \neq 0$
$\tilde{x} \leftarrow \mathsf{ConvertInputs}(x)$
$(\mathsf{pp}, \mathsf{sk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \xleftarrow{\$} \mathsf{HSS.KeyGen}(1^\lambda, \mathsf{crs}_{\mathsf{HSS}})$
$\widetilde{C}_0 \xleftarrow{\$} \mathsf{Share}_{\mathsf{sinct},0}(\mathsf{pp}, \mathsf{ek}_0, h)$
$(\mu_0, \mu_1) \xleftarrow{\$} \mathsf{HSS.Share}_{\mathsf{priv}}(\mathsf{pp}, \mu)$
$y_0 \xleftarrow{\$} \mathsf{HSS.HCircuitEval}(0, \mathsf{pp}, \mathsf{ek}_0, \mathsf{UC\text{-}then\text{-}mult}_{\tilde{x}}, (\widetilde{C}_0, \mu_0); r)$
$z_0 \xleftarrow{\$} \mathsf{HSS.Dec}^{(0)}(\mathsf{pp}, \mathsf{ek}_0, y_0)$
$\mathsf{ct} \leftarrow (x, \mathsf{pp}, \mathsf{ek}_1, d, \psi, \mu_1, z_0)$
return $(\mathsf{view}', \mathsf{ct})$

3. If we add the lines
   $\widetilde{C}_1 \xleftarrow{\$} \mathsf{HSS.Share}_{\mathsf{sinct},1}(\mathsf{pp}, \mathsf{ek}_1, \widetilde{C}, \psi)$
   $y_1 \xleftarrow{\$} \mathsf{HSS.HCircuitEval}(0, \mathsf{pp}, \mathsf{ek}_1, \mathsf{UC\text{-}then\text{-}mult}_{\tilde{x}}, (\widetilde{C}_1, \mu_1); r)$
   to the experiment (which changes nothing to the outcome) then by circuit-homomorphism correctness (Lemma 32) y_0 and y_1 will be shares of $C(x)$ μ_0 (which is 0 when $C(x) = 0$) with all but negligible probability.

4. If we add the line
   $z_1 \xleftarrow{\$} \mathsf{HSS.Dec}^{(1)}(\mathsf{pp}, \mathsf{ek}_1, y_1)$
   to the experiment (which again changes nothing to the outcome), then by additive decoding (Definition 17), it should hold with all but negligible probability that $z_1 - z_0 = 0$ (because with all but negligible probability y_0 and y_1 decode to 0). So we only incur a negligible security loss if we switch to reverse sampling z_0 as
   $y_1 \xleftarrow{\$} \mathsf{HSS.HCircuitEval}(0, \mathsf{pp}, \mathsf{ek}_1, \mathsf{UC\text{-}then\text{-}mult}_{\tilde{x}}, (\widetilde{C}_1, \mu_1); r)$
   $z_1 \xleftarrow{\$} \mathsf{HSS.Dec}^{(1)}(\mathsf{pp}, \mathsf{ek}_1, y_1)$
   $z_0 \leftarrow z_1 + 0$

5. By privacy of the public-key private sharing ($def : pk - priv - sharing$), we can switch from $(\mu_0, \mu_1) \xleftarrow{\$} \mathsf{HSS.Share}_{\mathsf{priv}}(\mathsf{pp}, \mu)$ to $(\mu_0, \mu_1) \xleftarrow{\$} \mathsf{HSS.Share}_{\mathsf{priv}}(\mathsf{pp}, \mu')$. From there we can reverse the hybrids one-by-one (switching back to sampling z_0 instead of reverse-sampling it, then reintroduce $\mathsf{LFE.crsGen}$, $\mathsf{LFE.Compress}$, and $\mathsf{LFE.Enc}$) and obtain the second experiment from AB-LFE privacy:

$\mathsf{crs} \xleftarrow{\$} \mathsf{LFE.crsGen}(1^\lambda, \mathsf{params})$
$(\mathsf{view}, C) \xleftarrow{\$} \mathcal{A}_1(1^\lambda, \mathsf{crs})$
$(h, \psi) \xleftarrow{\$} \mathsf{LFE.Compress}(\mathsf{crs}, C)$
$(\mathsf{view}', x, m_0, m_1) \xleftarrow{\$} \mathcal{A}_2(\mathsf{view}, h, \psi)$
abort if $C(x) \neq 0$
$\mathsf{ct} \xleftarrow{\$} \mathsf{LFE.Enc}(\mathsf{crs}, h, x, m_1)$
return $(\mathsf{view}', \mathsf{ct})$

– *Function privacy.* Let $\mathcal{A}$ be a *p.p.t.* adversary.
1. We start with the first experiment of the definition of AB-LFE function privacy.

$$\boxed{\begin{array}{l}\mathsf{crs} \xleftarrow{\$} \mathsf{LFE.crsGen}(1^\lambda, \mathsf{params}) \\ (\mathsf{view}, C, C') \xleftarrow{\$} \mathcal{A}(1^\lambda, \mathsf{crs}) \\ (h, \psi) \xleftarrow{\$} \mathsf{LFE.Compress}(\mathsf{crs}, C) \\ \text{return } (\mathsf{view}, h)\end{array}}$$

2. If we then inline the definition of LFE.crsGen and LFE.Compress then experiments are still code-equivalent.

$$\boxed{\begin{array}{l} r \xleftarrow{\$} \{0,1\}^R \\ \mathsf{crs}_{\mathsf{HSS}} \xleftarrow{\$} \mathsf{HSS.crsGen}(1^\lambda) \\ (\mathsf{view}, C, C') \xleftarrow{\$} \mathcal{A}(1^\lambda, \mathsf{crs}_{\mathsf{HSS}}, r) \\ \widetilde{C'} \leftarrow \mathsf{ConvertCircuit}(C') \\ (h, \psi) \xleftarrow{\$} \mathsf{HSS.Hash}(\widetilde{C'}) \\ \text{return } (\mathsf{view}, h)\end{array}}$$

3. Because having an adversary internally sample $r \xleftarrow{\$} \{0,1\}^R$ then run $\mathcal{A}_1(1^\lambda, \mathsf{crs}_{\mathsf{HSS}}, r)$ cannot constitute a distinguisher against the hasher privacy game of Definition 34, we can switch to the following hybrid.

$$\boxed{\begin{array}{l} r \xleftarrow{\$} \{0,1\}^R \\ \mathsf{crs}_{\mathsf{HSS}} \xleftarrow{\$} \mathsf{HSS.crsGen}(1^\lambda) \\ (\mathsf{view}, C, C') \xleftarrow{\$} \mathcal{A}(1^\lambda, \mathsf{crs}_{\mathsf{HSS}}, r) \\ \widetilde{C} \leftarrow \mathsf{ConvertCircuit}(C) \\ (h, \psi) \xleftarrow{\$} \mathsf{HSS.Hash}(\widetilde{C}) \\ \text{return } (\mathsf{view}, h)\end{array}}$$

4. Finally, be reversing the inlining of LFE.crsGen and LFE.Compress.

$$\boxed{\begin{array}{l}\mathsf{crs} \xleftarrow{\$} \mathsf{crsGen}(1^\lambda, \mathsf{params}) \\ (\mathsf{view}, C, C') \xleftarrow{\$} \mathcal{A}_1(1^\lambda, \mathsf{crs}) \\ (h, \psi) \xleftarrow{\$} \mathsf{Compress}(\mathsf{crs}, C') \\ \text{return } (\mathsf{view}, h)\end{array}}$$

□

I Proof of Theorem 21

Theorem 21 (Computationally Succinct CPRF for P/poly from HSS). *Let $\lambda \in \mathbb{N}$ be a security parameter. Let $B = 2^{\mathsf{poly}(\lambda)}$. Assume* HSS *has subexponential correctness for all the following properties: homomorphism (Section C.2), public-key private sharing (Definition 33), succinct semi-private sharing (Definition 34), and additive decoding for* REG *(Definition 17). If the parameters of* HSS *are chosen (using complexity leveraging) so that all the above correctness properties hold with probability at least $1 - 2^{-2\lambda}$, then the construction of Fig. 10 is a single-key, selectively secure constrained PRF supporting as constraints the class of all polynomial-sized circuits with output in $\{0,1\}$ such that every $x \in \{0,1\}^\lambda$ is B-admissible for C.*

We start by proving a correctness lemma.

Lemma 46 (A "strong correctness" property of the CPRF of Fig. 10). *Let $B = \mathcal{O}(2^{\mathsf{poly}(\lambda)})$. For every arithmetic circuit C with output in $\{0,1\}$ such that every $x \in \{0,1\}^\lambda$*

is B-admissible for C,

$$\Pr\left[\begin{array}{l}\forall x \in \{0,1\}^\lambda, \\ \mathsf{CEval}(1^\lambda, \mathsf{ck}, x) - \mathsf{Eval}(1^\lambda, \mathsf{msk}, x) = C(x)\ \mathsf{PRF}_{k^{\mathsf{sec}}_{\mathsf{PRF}}}(x)\end{array}\middle|\begin{array}{r}\mathsf{msk} = (k^{\mathsf{sec}}_{\mathsf{PRF}}(x), \dots) \xleftarrow{\$} \mathsf{CPRF.KeyGen}(1^\lambda) \\ \mathsf{ck} \xleftarrow{\$} \mathsf{CPRF.Constrain}(1^\lambda, \mathsf{msk}, C)\end{array}\right] \geq 1 - \mathsf{negl}(\lambda)$$

Proof. The proof of Lemma 46 boils down to showing that correctness holds with probability at least $\mathsf{negl}(\lambda)/2^\lambda$ for *each* $x \in \{0,1\}^\lambda$, and then conclude by a union bound. To that end, let $x \in \{0,1\}^\lambda$, and consider the event E_x defined by the following probability experiment:

$$\begin{array}{l}\mathsf{msk} = (k^{\mathsf{sec}}_{\mathsf{PRF}}(x), \dots) \xleftarrow{\$} \mathsf{CPRF.KeyGen}(1^\lambda) \\ \mathsf{ck} \xleftarrow{\$} \mathsf{CPRF.Constrain}(1^\lambda, \mathsf{msk}, C) \\ E_x : \text{“}\mathsf{CEval}(1^\lambda, \mathsf{ck}, x) - \mathsf{Eval}(1^\lambda, \mathsf{msk}, x) = C(x)\ \mathsf{PRF}_{k^{\mathsf{sec}}_{\mathsf{PRF}}(x)}\text{”}\end{array}$$

After inlining the definitions of $\mathsf{CPRF.KeyGen}$, $\mathsf{CPRF.Constrain}$, $\mathsf{CPRF.Eval}$, and $\mathsf{CPRF.CEval}$:

$$\begin{array}{l}r \xleftarrow{\$} \{0,1\}^R \\ k^{\mathsf{sec}}_{\mathsf{PRF}} \xleftarrow{\$} \{0,1\}^\lambda \\ \mathsf{crs} \xleftarrow{\$} \mathsf{HSS.crsGen}(1^\lambda) \\ (\mathsf{pp}, \mathsf{sk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \xleftarrow{\$} \mathsf{HSS.KeyGen}(1^\lambda, \mathsf{crs}) \\ (\widetilde{C}_0, \mathsf{st}) \leftarrow \mathsf{HSS.Share}_{\mathsf{s\text{-}priv},0}(\mathsf{pp}, \mathsf{sk}) \\ (k_0, k_1) \xleftarrow{\$} \mathsf{HSS.Share}_{\mathsf{priv}}(\mathsf{pp}, k^{\mathsf{sec}}_{\mathsf{PRF}}) \\ \widetilde{C} \leftarrow \mathsf{ConvertCircuit}(C) \\ \widetilde{C}_1 \xleftarrow{\$} \mathsf{HSS.Share}_{\mathsf{s\text{-}priv},1}(\mathsf{pp}, \mathsf{sk}, \tilde{C}, \mathsf{st}) \\ y_1 \xleftarrow{\$} \mathsf{HSS.HCircuitEval}(1, \mathsf{pp}, \mathsf{ek}_1, \mathsf{UC\text{-}then\text{-}PRF}_x, (\widetilde{C}_1, k_1); r) \\ y_0 \xleftarrow{\$} \mathsf{HSS.HCircuitEval}(0, \mathsf{pp}, \mathsf{ek}_0, \mathsf{UC\text{-}then\text{-}PRF}_x, (\widetilde{C}_0, k_0); r) \\ E_x : \text{“}y_1 \bmod 2^\lambda - y_0 \bmod 2^\lambda = C(x)\ \mathsf{PRF}_{k^{\mathsf{sec}}_{\mathsf{PRF}}}(x)\text{”}\end{array}$$

In this form, this clearly correspond to a standard HSS correctness game, which follows from the various notions of HSS correctness (homomorphism, public-key private sharing, succinct semi-private sharing, and additive decoding for REG), and so $\Pr[y_1 - y_0 \neq C(x)\ \mathsf{PRF}_{k^{\mathsf{sec}}_{\mathsf{PRF}}}(x)] \leq 4 \cdot 2^{-2\lambda}$. This means $\Pr[E_x] \leq \mathsf{negl}(\lambda)/2^\lambda$, as desired. □

We are now ready to prove the theorem.

Proof of Theorem 21. We now use this lemma to prove single-key selective CPRF security (Definition 5). Our simulator generates ck as in the real experiment (by generating a master key and then constraining it), except that it always sets the PRF key $k^{\mathsf{sec}}_{\mathsf{PRF}}(x)$ to be all zeroes. We now present a hybrid proof to show that the real experiment is indistinguishable from the ideal experiment, using this simulation.

$\mathcal{H}_0$. This is the real experiment for single-key selective CPRF security. Master key $\mathsf{msk} \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda)$ and constrained key $\mathsf{ck} \xleftarrow{\$} \mathsf{Constrain}(1^\lambda, \mathsf{msk}, C)$ are sampled by the challenger, and ck is given to the adversary. The adversary then receives oracle access to the master key evaluation algorithm $x \mapsto \mathsf{Eval}(1^\lambda, \mathsf{msk}, x)$.

$\mathcal{H}_1$. Replace the master key evaluation oracle with

$$x \mapsto \mathsf{CEval}(1^\lambda, \mathsf{ck}, x) - C(x)\ \mathsf{PRF}_{k^{\mathsf{sec}}_{\mathsf{PRF}}(x)} \bmod 2^\lambda.$$

By Lemma 46, this is an identical function, with all but negligible probability.

$\mathcal{H}_2$. Inside $\mathsf{KeyGen}(1^\lambda)$, replace

$$(k_0, k_1) \xleftarrow{\$} \mathsf{HSS.Share}_{\mathsf{priv}}(\mathsf{pp}, k^{\mathsf{sec}}_{\mathsf{PRF}}) \quad \Longrightarrow \quad (k_0, k_1) \xleftarrow{\$} \mathsf{HSS.Share}_{\mathsf{priv}}(\mathsf{pp}, 0^\lambda).$$

Note that since $\mathcal{H}_1$, the master key msk is not used directly — the adversary's view only depends on ck and $k^{\mathsf{sec}}_{\mathsf{PRF}}$. ck only contains party 1's view of evaluation key and shares, so we can reduce this change to the privacy of $\mathsf{HSS.Share}_{\mathsf{priv}}$ (Definition 33).

$\mathcal{H}_3$. Replace

$$x \mapsto \mathsf{CEval}(1^\lambda, \mathsf{ck}, x) - C(x)\ \mathsf{PRF}_{k^{\mathsf{sec}}_{\mathsf{PRF}}(x)} \bmod 2^\lambda$$
$$\Longrightarrow$$
$$x \mapsto \mathsf{CEval}(1^\lambda, \mathsf{ck}, x) - C(x)R(x) \bmod 2^\lambda,$$

for a fresh uniformly random function $R\colon \{0,1\}^\lambda \to [0, 2^\lambda)$. Since $\mathcal{H}_2$ this is the only use of $k^{\mathsf{sec}}_{\mathsf{PRF}}(x)$, so this reduces to the security of PRF.

$\mathcal{H}_4$. Answer the master key evaluation queres as in the ideal experiment, with $\mathsf{CEval}(1^\lambda, \mathsf{ck}, x)$ when $C(x) = 0$, and alternatively uniformly from $[0, 2^\lambda)$ when $C(x) = 1$. This is exactly the previous hybrid in the former case, and uses $R(x)$ as a random mask (to cover up $\mathsf{CEval}(1^\lambda, \mathsf{ck}, x)$) in the latter case, so this is identically distributed. We are now at the ideal experiment. □