Large Message Homomorphic Secret Sharing from DCR and Applications

Lawrence Roy[⋆] and Jaspal Singh

Oregon State University, {`ldr709@gmail.com`, `singjasp@oregonstate.edu`}

Abstract. We present the first homomorphic secret sharing (HSS) construction that simultaneously (1) has negligible correctness error, (2) supports integers from an exponentially large range, and (3) relies on an assumption not known to imply FHE — specifically, the Decisional Composite Residuosity (DCR) assumption. This resolves an open question posed by Boyle, Gilboa, and Ishai (Crypto 2016). Homomorphic secret sharing is analogous to fully-homomorphic encryption, except the ciphertexts are shared across two non-colluding evaluators. Previous constructions of HSS either had non-negligible correctness error and polynomial-size plaintext space or were based on the stronger LWE assumption. We also present two applications of our technique: a two server ORAM with constant bandwidth overhead, and a rate-1 trapdoor hash function with negligible error rate.

1 Introduction

Homomorphic secret sharing is a relaxation of fully-homomorphic encryption (FHE) where the ciphertexts are shared across two non-colluding evaluators, who may homomorphically evaluate functions on their shares. In FHE, if $c \leftarrow \mathsf{Enc}(x)$ then $\mathsf{Hom}(f, c)$ is an encryption of $f(x)$. In HSS, if $s_0, s_1 \leftarrow \mathsf{Share}(x)$ then $\mathsf{Hom}(f, s_1)$ and $\mathsf{Hom}(f, s_0)$ (computed independently) are a sharing of $f(x)$.

Boyle, Gilboa, and Ishai [BGI16] initiated the line of work on secure computation from HSS with a construction based on the Decisional Diffie–Hellman (DDH) assumption. They used their scheme to achieve the first secure two-party computation protocol with *sublinear communication* from an assumption not known to imply FHE. Though their HSS only supports restricted multiplication straight-line (RMS) programs, this is enough at least to evaluate polynomial-size branching programs. All known HSS constructions (including ours) that aren't based on FHE have this same limitation.

The HSS of [BGI16] has two main limitations. First, it achieves correctness with probability only $1 - p$ (for $p = 1/\mathsf{poly}$). Second, it can only support a message space of polynomial size M, as it requires $O(M/p)$ time for a step they call "share conversion". [FGJS17] constructed a similar HSS scheme based on Paillier encryption (from the DCR assumption), with the same limitations and $O(M/p)$-time share conversion technique. The cost of share conversion was later

[⋆] Supported by a DoE CSGF Fellowship.

improved to $O(\sqrt{M/p})$ by [DKK18], which they proved is optimal for these schemes unless faster interval discrete logarithm algorithms are found.

These limitations were eventually removed by [BKS19], using lattice-based cryptography. Their scheme is based the Learning With Errors (LWE) assumption, and achieves homomorphic secret sharing with exponentially small correctness error and exponentially large plaintext space. The LWE assumption is strong enough to construct FHE [BV11], although their HSS scheme uses simpler techniques and can be more efficient than FHE.

Why is correctness error important? Besides the theoretical distinction, it increases the overhead for secure computation: the 2PC protocol of [BGI16] needs to repeat homomorphic evaluation polynomially many times and take a majority vote (using another MPC protocol) on the outcome. Longer programs have higher chance for error, as if any operation errors then the whole computation will fail. Consequently, when evaluating an n-step program on plaintexts bounded by M, they require $O(Mn^2)$ time to get a constant error rate (or $O(Mn^2t)$ time after repeating for $O(t)$ tries to get a negligible error rate of 2^{-t}). The reduced error rate from [DKK18] allows this to be improved to $O(M^{1/2}n^{3/2})$. Ideally, we would want the computation cost of a 2PC protocol to be linear in n.

Supporting exponentially large plaintext space can also improve the 2PC protocol's computational complexity, because it is necessary to represent the HSS scheme's key inside of its messages. [BGI16] manage this by taking the bit-decomposition of the key, though this multiplies the computational cost by the key size. When M can be exponentially large, however, the key can directly fit inside the plaintext space. Additionally, computations can be performed on large chunks of data at a time, further improving efficiency. Finally, there may be some computations that can only be performed with the larger message space bound. Specifically, RMS programs with a polynomial bound on memory values are sufficient to evaluate branching programs [BGI16], while with a large enough message space algebraic branching programs over $\mathbb{Z}$ can be evaluated.

The question of whether negligible correctness error could be achieved from an assumption not known to imply FHE was left as an open problem by [BGI16].[1]

1.1 Our Results

We give an affirmative answer this open question. We construct an HSS scheme based on Damgård–Jurik encryption (under the DCR assumption) that achieves negligible correctness error and exponentially large message space. When our HSS is used for 2PC, there is no need for repeated HSS evaluation to amplify correctness. We can therefore securely evaluate n-step RMS programs in $O(n)$ time. Previous constructions required a polynomial bound on the size of the values in the RMS computation, while our construction natively supports arithmetic operations over exponentially large values.

The main insight in our construction is to define a new "distance function", the key step used for share conversion in HSS schemes. Ours is based on the

[1] A concurrent work [OSY21] has also independently solved this problem.

algebraic properties of the ciphertext group $(\mathbb{Z}/N^{s+1}\mathbb{Z})^\times$, while existing distance functions use the generic technique of searching for a randomly chosen subset of ciphertexts. This allows us to extract an exponentially large result from our distance function, and achieve share conversion with a negligible error rate.

We also present several other applications of our new result and techniques:

ORAM. We propose a novel 2-server malicious secure Oblivious RAM (ORAM) protocol that achieves constant bandwidth. An ORAM protocol allows the client to hide its access pattern on a database outsourced to untrusted server(s). Our protocol is closely based on the single server Onion ORAM protocol [DvDF+16], which leverages server side computation to achieve constant bandwidth blowup. We replace this server side computation with a number of RMS programs, which can be evaluated by the two servers using our HSS scheme.

While there already exist multi-server ORAM constructions with constant client-server bandwidth overhead (*e.g.*, [DvDF+16,FNR+15,HOY+17]), they all require either super-constant server-server communication or a minimum block size of $\Omega(\log^6 N)$, where N is the number of blocks in the ORAM. Whereas, our HSS based 2-server ORAM achieves constant bandwidth for block of size $\omega(\log^4 N)$ and with no server-server communication.

Trapdoor hash functions. Beyond HSS, another construction based on the notion of a distance function is trapdoor hash functions (TDH) [DGI+19]. Rate-1 TDHs are a kind of hash function that have additional properties useful for two-party computation. Specifically, if Alice has some f in a limited class of predicates and Bob has a message x, if Bob sends the hash of x and Alice sends a key generated based on f, they can each compute a single-bit share of $f(x)$. [DGI+19] use rate-1 TDHs to build rate-1 string oblivious transfer (OT), from which they construct efficient private information retrieval and semi-compact homomorphic encryption. They also present several other constructions based on TDHs.

Prior work [DGI+19] constructed rate-1 TDHs from a variety of assumptions (DDH, QR, DCR, and LWE), but only their QR and LWE instantiations achieve negligible correctness error. For DDH and DCR, they had to compensate by using error correcting codes in their construction of rate-1 string OT. We can directly construct a rate-1 trapdoor hash function from DCR using our distance function, achieving negligible correctness error. Our construction also generalizes beyond TDHs, in that it can handle functions f outputting more than a single bit.

HSS definition. We extend the definition of HSS to allow (generalized, to represent RMS operations) circuits to be evaluated one gate at a time. One benefit of this approach is that it allows secure evaluation of online algorithms, which may take input and produce output many times, while maintaining some secret state. The function to evaluate may be chosen adaptively based on previous outputs or shares. We also define malicious security of HSS, in the form of share authentication. These definitions are directly useful for our application to ORAM.

1.2 Technical Overview

Introduction to HSS. HSS schemes work through the interaction of two different homomorphic schemes: additively homomorphic encryption and additive secret sharing. Following the notation of [BGI16], let $[\![x]\!]$ denote an encryption of x. Let $\langle y\rangle$ denote additive shares of y, meaning that party 0 has $\langle y\rangle_0$ and party 1 has $\langle y\rangle_1$ such that $\langle y\rangle_1 - \langle y\rangle_0 = y$. Then $\langle x\rangle + \langle y\rangle \equiv \langle x+y\rangle$, where $\equiv$ means shares that decode to the same value, or ciphertexts that decrypt to the same plaintext. We will write the group operation on the homomorphic encryption multiplicatively, so $[\![x]\!][\![y]\!] \equiv [\![x+y]\!]$. Any additively homomorphic encryption supports multiplication by constants, so we have $[\![x]\!]^c \equiv [\![cx]\!]$.

We have two different additively homomorphic schemes; what happens if we let them interact? If the parties compute $[\![x]\!]^{\langle y\rangle}$, they get $\langle\!\langle [\![xy]\!] \rangle\!\rangle$, where $\langle\!\langle z\rangle\!\rangle$ denotes multiplicative shares of z. More precisely, party i has $\langle\!\langle [\![x]\!]^y \rangle\!\rangle_i = [\![x]\!]^{\langle y\rangle_i}$, and $\langle\!\langle [\![x]\!]^y \rangle\!\rangle_1 / \langle\!\langle [\![x]\!]^y \rangle\!\rangle_0 = [\![x]\!]^{\langle y\rangle_1 - \langle y\rangle_0} \equiv [\![xy]\!]$. What's interesting here is that by combining the two encryption schemes we get a representation of the product. That is, we have a bilinear map. However, we would really like to be able to perform multiple operations in sequence. Is there any way we could make the result instead be $\langle xy\rangle$?

Luckily, many additively homomorphic encryption schemes perform decryption through exponentiation, the same operation as was used for homomorphically multiplying by a constant. For Paillier, decryption is $\phi^{-1}([\![z]\!]^{\varphi}) = z$, where $\phi(z) = 1 + N\varphi z$ is a homomorphism from the plaintext space to the ciphertext space, and N and φ are the public and private keys. Therefore, if we have shares $\langle \varphi y\rangle$ then we can compute $[\![x]\!]^{\langle\varphi y\rangle} \equiv \langle\!\langle \phi(x)^y \rangle\!\rangle \equiv \langle\!\langle \phi(xy)\rangle\!\rangle$. For ElGamal, the decryption of a ciphertext $[\![z]\!] = (A, B)$ is $\phi^{-1}(A^{-k}B)$, where $\phi(z) = g^z$ for a public generator g. Again, ϕ is a homomorphism from the plaintext space. This is slightly more complicated in that it's taking a dot product "in the exponent" with the private key vector $\vec{k} = [-k\ 1]$, but if we take the secret shares to be vectors $\langle \vec{k}y\rangle$ then still we have $[\![x]\!]^{\langle \vec{k}y\rangle} \equiv \langle\!\langle \phi(x)^y \rangle\!\rangle \equiv \langle\!\langle \phi(xy)\rangle\!\rangle$.

The last step of decryption for both schemes is to compute ϕ^{-1}. For HSS we need to do the same, but on the multiplicative shares $\langle\!\langle \phi(z)\rangle\!\rangle$ split across the two parties performing HSS. This is done with a *distance function*, with the property that $\mathrm{Dist}(a\phi(z)) - \mathrm{Dist}(a) = z$, ideally for any ciphertext a and plaintext z. Then $\mathrm{Dist}(\langle\!\langle \phi(xy)\rangle\!\rangle_i) \equiv \langle xy\rangle_i$ gives additive shares of the multiplication result. The idea from [BGI16] for constructing Dist is that both parties agree on a common set of "special points", which they choose randomly. They iteratively compute $a\phi(-1)^j$, starting at $j = 0$ and continuing until $c = a\phi(-1)^j$ is special, then set $\mathrm{Dist}(a)$ to be the distance j. If they find the same special point c,

$$\mathrm{Dist}(a\phi(z)) - \mathrm{Dist}(a) = \mathrm{Dist}(c\phi(j+z)) - \mathrm{Dist}(c\phi(j)) = j + z - j = z,$$

so their distances are additive shares of z. When the special points are chosen randomly and z is small, $\mathrm{Dist}(a\phi(z))$ and $\mathrm{Dist}(a)$ will usually pick the same c.

Putting this all together, HSS consists of a way of homomorphically multiplying a ciphertext $[\![x]\!]$ by a share $\langle y\rangle$, or rather $\langle ky\rangle$ for some private key k, to get $\langle\!\langle \phi(xy)\rangle\!\rangle$, then finally using a distance function to find $\langle xy\rangle$. A circularly secure encryption scheme allows ky to be encrypted, so then $\mathrm{Dist}\big([\![ky]\!]^{kx}\big) \equiv \langle kxy\rangle$, which can feed the input of another multiplication operation, and so on.

Paillier distance function. We now present a simplified version of our main HSS construction. It uses a variant of Paillier encryption, where instead of encrypting messages as $r^N(1+Nz) \bmod N^2$ for public key N and uniformly random r, it encrypts them as $r^{N^3}(1+N^2z) \bmod N^4$. This is to allow the plaintext size to be bigger than the private key. We have $\left(r^{N^3}(1+N^2z)\right)^{\varphi} = 1+N^2\varphi z = \phi(z)$. To find $\phi^{-1}(a)$, compute $(a-1)/N^2$, as $a-1$ must be a multiple of N^2, then multiply by $\varphi^{-1} \bmod N^2$.

It turns out that we can design a distance function that is based on this ϕ^{-1}. A prior construction of HSS from Paillier encryption, [FGJS17], had a minor optimization based on $\langle\langle\phi(z)\rangle\rangle_1 = \langle\langle\phi(z)\rangle\rangle_0 \mod N^2$, since $\langle\langle\phi(z)\rangle\rangle_1/\langle\langle\phi(z)\rangle\rangle_0 = \phi(z) = 1+N^2\varphi z$. Therefore both parties will have something in common, and they can use it as their common point $c = \langle\langle\phi(z)\rangle\rangle_0 \bmod N^2 = \langle\langle\phi(z)\rangle\rangle_1 \bmod N^2$. On input a, let the distance function pick a canonical representative $c = a \bmod N^2 \in [-\frac{N-1}{2}, \frac{N-1}{2}]$. Then $a/c = 1+N^2w$, and we let $\mathrm{Dist}(a) = w$. This means that our "special points" are $[-\frac{N-1}{2}, \frac{N-1}{2}]$, instead of a random set like [BGI16]. We then have $\mathrm{Dist}(a\phi(z)) - \mathrm{Dist}(a) = \varphi z$, because $a\phi(z)/c = (1+N^2\varphi z)(1+N^2w) = 1+N^2(\varphi z + w)$. This is a slightly different property than what we specified for distance functions, but it is actually even better as we don't need to use circularly secure encryption to get $\langle\varphi xy\rangle$ as the result of multiplication — φ will already be multiplied in the output.

However, there's one last step before we have an HSS. The result from Dist will be in the form of additive shares modulo N^2, and we need them to be additive shares in $\mathbb{Z}$ so that we can use them as an exponent in the next operation. Exponentiating to a power that is modulo N^2 would not make sense, as the multiplicative order of almost any ciphertext does not divide N^2. We use a trick from the LWE HSS construction: additive shares modulo N^2 of a value z much smaller than N^2 (so $|z|/N^2$ is negligible) have overwhelming probability of being additive shares over $\mathbb{Z}$, *without any modulus.* Therefore we can make a distance function that has only negligible failure probability and supports an exponentially large bound on the plaintext.

1.3 Other Related Work

We compare our proposed ORAM construction to Onion ORAM [DvDF+16], which is also based on the Damgård–Jurik public-key encryption. To ensure malicious security and achieve constant bandwidth overhead, the scheme allows for blocks of size $\tilde{\omega}(\log^6 N)$, with $\widetilde{O}(B\log^4 N)$ client computation and $\tilde{\omega}(B\log^4 N)$ server computation. For comparison, our proposed ORAM construction allows for blocks of size $\tilde{\omega}(\log^4 N)$, with $\widetilde{O}(B\log^4 N)$ client computation and $\widetilde{O}(B\log^5 N)$ server computation. To ensure the integrity of server side storage, Onion ORAM uses a verification algorithm that relies on probabilistic checking and error correcting codes. This integrity check adds an overhead to the communication and computation. In our protocol we get this verification check "for free", as the HSS shares held by the two servers satisfy the authenticated property — which ensures that a single corrupt server cannot modify its share without it being detected by the client during the decoding process. This gives major savings in our protocol's communication and client side computation compared to Onion ORAM.

Bucket ORAM proposed by Fletcher et al. [FNR$^+$15] proposes a single server ORAM with constant bandwidth overhead for blocks of size $\tilde{\Omega}(\log^6 N)$. It's a constant round protocol, but asymptotically its client and server computation match that of Onion ORAM. S^{3}ORAM [HOY$^+$17] proposes a multi-server ORAM construction with constant client-server bandwidth overhead. It avoid the evaluation of homomorphic operations on the server side and is based on Shamir Secret Sharing. However, this protocol incurs $O(\log N)$ overhead in server-server communication, which makes the overall communication overhead logarithmic. Another interesting work on designing 2-server ORAMs optimized for secure computation is due to Doerner and Shelat [DS17]. Their construction is based on the notion of function secret sharing, which is closely related to HSS. However, it also incurs an $O(\log N)$ server-server communication overhead.

1.4 Concurrent Result

A concurrent and independent work [OSY21] also constructs an HSS from the DCR assumption and achieves negligible correctness error for an exponentially large plaintext space. Qualitatively, our distance function, which is the main construction we base our results on, matches theirs. There are two main aspects in which our work improves on theirs.

We use Damgård–Jurik encryption, which allows the plaintext space to be significantly larger than the whole private key. OSY instead uses Paillier encryption. They consequently have to split their private key into chunks, requiring either a circular security assumption or a provably circular secure encryption scheme. OSY needs to use around 6 chunks, assuming circular security, or $\Theta(\log(N))$ without. While our ciphertexts are somewhat bigger, we only need a single ciphertext for an input to our HSS scheme. We therefore have either a constant or $\Theta(\ell(\kappa))$ speedup in both computation and communication relative to OSY's HSS scheme, depending on the assumption. While their scheme more naturally supports additive decoding, a variant of our scheme also has this property.

We also give novel HSS definitions and proofs that support running online algorithms, and adaptively choosing functions to evaluate based on previous ciphertexts. We define authenticated HSS, and prove that our construction is authenticated, allowing its use in maliciously secure protocols such as our ORAM.

2 Preliminaries

2.1 Notation

Modular arithmetic. Let $\mathbb{Z}/N\mathbb{Z}$ be the ring of integers modulo N and $(\mathbb{Z}/N\mathbb{Z})^+$ be its additive group. Let $(\mathbb{Z}/N\mathbb{Z})^\times$ be the multiplicative group of all units x of $\mathbb{Z}/N\mathbb{Z}$, i.e. all x coprime to N. Normally multiplication of $\bar{x} = x + N\mathbb{Z} \in \mathbb{Z}/N\mathbb{Z}$ by some integer $K \in \mathbb{Z}$ will just be $K\bar{x} = Kx + N\mathbb{Z} \in \mathbb{Z}/N\mathbb{Z}$; however, we overload this to mean $Kx + KN\mathbb{Z} \in \mathbb{Z}/KN\mathbb{Z}$ as well. We will notate the quotient map from $\mathbb{Z}/KN\mathbb{Z}$ to $\mathbb{Z}/N\mathbb{Z}$ as $\cdot + N\mathbb{Z}$, or omit it when it is clear from context. To say that two values a and b are the same modulo N, i.e., that $a + N\mathbb{Z} = b + N\mathbb{Z}$, we write $a \equiv_N b$. For modulus we assume round to nearest, so $\cdot \bmod N\colon \mathbb{Z}/N\mathbb{Z} \to [-\frac{N}{2}, \frac{N}{2}) \cap \mathbb{Z}$ and $x = (x \bmod N) + N\mathbb{Z}$ for all x.

Algorithm notation. We write our constructions in pseudocode. While the notation should be mostly self-explanatory, there are a few things to take note of. The boolean AND and OR operations are $\wedge$ and $\vee$, and the compliment of a bit b is $\bar{b} = 1 - b$. We give equality testing its own symbol, $\overset{?}{=}$, so $x \overset{?}{=} y$ is 1 if $x = y$, and 0 otherwise. Assignment statements are written as $x := 1$, while sampling is written as $x \leftarrow \{0, 1\}$, to indicate that x is uniformly random in the set $\{0, 1\}$. We use $\rho \leftarrow \$$ to represent sampling a uniformly random bit stream ρ. This notation also applies to subroutine calls, so if f is deterministic then the notation is $y := f(x)$, but if f is randomized then it is $y \leftarrow f(x)$.

We will also write our definitions in pseudocode, expressing our security properties as indistinguishability of two randomized algorithms. Often the adversary $\mathcal{A}$ gets to choose some x partway through a randomized algorithm. To preserve the adversary's state and give it to the distinguisher we use $(\text{view}, x) \leftarrow \mathcal{A}$ and return view from the distribution along with everything else. This way $\mathcal{A}$ can put its state in view and the distinguisher will see it.

2.2 Damgård–Jurik Encryption

Our construction is based on the Damgård–Jurik public-key encryption scheme [DJ01], a generalization of Paillier encryption [Pai99]. At a high level, the plaintexts of Damgård–Jurik are members of an additive group $(\mathbb{Z}/N^s\mathbb{Z})^+$. Encryption applies an isomorphism exp from $(\mathbb{Z}/N^s\mathbb{Z})^+$ to a subgroup of $(\mathbb{Z}/N^{s+1}\mathbb{Z})^\times$, then hides the plaintext by multiplying by a random perfect power of N^s in $(\mathbb{Z}/N^{s+1}\mathbb{Z})^\times$. Decryption uses the private key to cancel out this random value, then applies log, the inverse of exp. This requires that the discrete logarithm be efficiently computable for the subgroup.

This is possible by taking advantage of N being nilpotent in $\mathbb{Z}/N^{s+1}\mathbb{Z}$. Power series in N can have at most $s+1$ nonzero terms because $N^{s+1} \equiv_{N^{s+1}} 0$, allowing us to use the usual Taylor series for e^{Nx} and $\frac{1}{N}\ln(x)$ to define $\exp(x)$ and $\log(x)$.

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

exp is an isomorphism from $(\mathbb{Z}/N^s\mathbb{Z})^+$ to $1 + N(\mathbb{Z}/N^{s+1}\mathbb{Z})$, the subgroup consisting of all $u \in (\mathbb{Z}/N^{s+1}\mathbb{Z})^\times$ such that $u \equiv_N 1$. Specifically, exp and log are inverse functions and $\exp(x + y) = \exp(x)\exp(y)$ (see full version of the paper for the proof of these properties). These functions are sufficient to define Damgård–Jurik encryption.

Definition 1. *Given a security parameter κ and a message size s, define the Damgård–Jurik encryption scheme as follows.*[2]

$(N, \varphi) \leftarrow \mathsf{DJ.KeyGen}(1^\kappa)$**:** *Generate an RSA modulus $N = pq$ where $2^{\ell(\kappa)-1} < p, q < 2^{\ell(\kappa)}$, and ℓ is a polynomial chosen to make the scheme achieve κ-bit security. Let the public key be N and the private key be $\varphi = \varphi(N)$, where $\varphi(N) = (p-1)(q-1)$ is Euler's totient function.*

$c \leftarrow \mathsf{DJ.Enc}_{N,s}(x)$**:** *Given $x \in \mathbb{Z}/N^s\mathbb{Z}$, choose a uniformly random $r \in (\mathbb{Z}/N^{s+1}\mathbb{Z})^\times$ and output $c = r^{N^s} \exp(x)$.*

$x := \mathsf{DJ.Dec}_{N,s,\varphi}(c)$**:** *Given $c \in (\mathbb{Z}/N^{s+1}\mathbb{Z})^\times$, output $x = \frac{1}{\varphi} \log(c^\varphi) \in \mathbb{Z}/N^s\mathbb{Z}$.*

Encryption is clearly additively homomorphic, since $r_1^{N^s} r_2^{N^s} \exp(x) \exp(y) = (r_1 r_2)^{N^s} \exp(x + y)$. Decryption is well defined because $c^\varphi \equiv_N 1$ by Euler's theorem, and because $p-1$ and $q-1$ are each coprime to N since p and q have the same bit length. The order of $(\mathbb{Z}/N^{s+1}\mathbb{Z})^\times$ is $\varphi(N^{s+1}) = p^s(p-1)q^s(q-1) = \varphi N^s$, so $\log(c^\varphi) = \log(r^{\varphi N^s} \exp(x)^\varphi) = \log(\exp(\varphi x)) = \varphi x$, which implies the correctness of decryption.

The security of this encryption scheme is based on the decisional composite residuosity assumption (DCR).

Definition 2. *The* decisional composite residuosity (DCR) *assumption is that the uniform distribution on $(\mathbb{Z}/N^2\mathbb{Z})^\times$ is indistinguishable from the uniform distribution on the subgroup of perfect powers of N in $(\mathbb{Z}/N^2\mathbb{Z})^\times$.*

We will not use the assumption directly, as it will be more convenient use the CPA security of Damgård–Jurik encryption as the basis for our security proofs.

Theorem 3 (Damgård and Jurik [DJ01, Thm. 1]). *Damgård–Jurik encryption is CPA secure if and only if the DCR assumption holds. That is, the oracles $\mathcal{O}_{i,N,s}(x_0, x_1) = \mathsf{DJ.Enc}_{N,s}(N, x_i)$ for $i \in \{0, 1\}$ must be indistinguishable, meaning that for any PPT $\mathcal{A}$ the following probability must be negligibly different between the two values of i.*

$$\Pr[(N, \varphi) \leftarrow \mathsf{DJ.KeyGen}(1^\kappa);\ \mathcal{A}^{\mathcal{O}_{i,N,s}}(N) = 1]$$

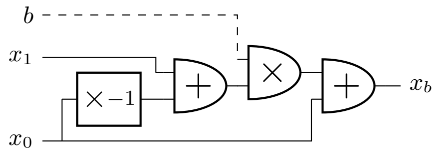
2.3 Universal Hashing

In our ORAM construction we will assume a family of hash function $H = \{h : U \to [m]\}$, which satisfied the *uniform difference property*, which states: for any two unequal $x, y \in U$, the number $(h(x) - h(y)) \mod m$ is uniformly random over all hash functions $h \in H$.

[2] Damgård–Jurik was originally defined using $\exp(x) = (1+N)^x$, which required log to use Hensel lifting. We instead chose to use Taylor series because it simplifies the description of log, and only require $O(s)$ additions and multiplications to evaluate with Horner's rule, while the Hensel lifting algorithm took $O(s^2)$ arithmetic operations.

```
SELECT$(b, x_0, x_1)$:
    $v := b(x_1 - x_0)$
    $w := v + x_0$
    return $w$
```

(a) As an RMS program



(b) As an RM circuit

Fig. 1: The selection function x_b represented as an RMS program (left, Definition 4) and a RM circuit (right, Definition 8). In the RM circuit, dashed wires (wire type IN) correspond to inputs in an RMS program, while solid wires (wire type REG) correspond to registers.

3 Circuit Homomorphic Secret Sharing

In this section we present a definition of homomorphic secret sharing (HSS) based on evaluating (generalized) circuits. We first present a notion of circuit that is general enough to capture the operations that our HSS scheme can perform, Restricted Multiplication Straight-line programs. Then we define a notion of HSS based on replacing each gate in a circuit with an operation that works on shares. We will only need to specify properties of a single gate at a time; these properties compose to become secure evaluation of a whole circuit.

The benefits of this approach are threefold. The piecewise definition allows the evaluation of online algorithms, where some output may need to be produced before the rest of the inputs can be taken, with state maintained throughout. It also allows the circuit to be chosen adaptively, based on previous outputs or even shares. Finally, it simplifies the proof of our HSS construction to be able to prove properties of individual gates and have them compose.

3.1 Restricted Multiplication Circuits

First, we give a definition for restricted multiplication straight-line programs, which were first defined in [Cle90]. We give a slight generalization however, allowing inputs to be added together before multiplication with a register. Polynomially sized RMS programs under the new definition could still be written in polynomial size in the traditional definition by applying the distributive property, but this may multiply the number of steps by n.

Definition 4. *A* Restricted Multiplication Straight-line (RMS) program *over a ring $\mathbb{K}$ is a sequential program taking with inputs $x_1, \ldots, x_n \in \mathbb{K}$ and registers $z_1, \ldots,$ where the outputs are a subset of the registers. Each instruction must take the form*

$$z_k := (A_0 + \sum_{i \leq n} A_i x_i)(B_0 + \sum_{i<k} B_i z_i),$$

for some constants $A_0, \ldots, A_N, B_0, \cdots B_{k-1}$.

For convenience we take the first n registers to be the inputs, to avoid explicitly writing out a conversion like $z_1 := 1$; $z_{i+1} := x_i z_1$. An example of an RMS program is shown in Figure 1a. We want to define a kind of circuit that captures the allowed operations in RMS programs. In an RMS program there are two types of values: inputs and registers. This suggests defining circuits with two types of wire, called IN and REG. The circuit for the example is shown in Figure 1b, where IN wires are drawn with a dashed line, and REG wires are drawn with a solid line. Gates representing linear operations (addition and multiplication-by-constant) are allowed for either type of wire, and both wire types allow sources for the value 1. However, multiplication is only allowed between the IN wire type and the REG wire type, and must always produce a REG wire.

Typed circuits. To make this formal, we need to define circuits with multiple types of wire. First we define circuit prototypes, which specify what types of wires and gates are allowed, then we define circuits for a given prototype.

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 maps* $\mathsf{in}\colon \mathsf{gates} \to \mathsf{types}^*$ *and* $\mathsf{out}\colon \mathsf{gates} \to \mathsf{types}$ *assigning to each gate type the wire types of its inputs and output.*

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* ***a)*** *a directed acyclic graph* $(\mathsf{nodes}, \mathsf{wires})$*,* ***b)*** *a total order on* wires*,* ***c)*** *subsets* $\mathsf{inputs}, \mathsf{outputs} \subseteq \mathsf{nodes}$*,* ***d)*** *a node labeling* $\mathsf{type}\colon \mathsf{nodes} \to \mathsf{types}$*, and* ***e)*** *a non-input node labeling* $\mathsf{gate}\colon \mathsf{nodes}\setminus\mathsf{inputs} \to \mathsf{gates}$*. A non-input node is called a gate. We require the circuit to be well-formed: for any gate v,* $\mathsf{type}(v) = \mathsf{out}\big(\mathsf{gate}(v)\big)$*, and if $(v_1, v), (v_2, v), \ldots, (v_n, v)$ are v's incoming wires in sorted order,* $\mathsf{type}(v_1)\,\mathsf{type}(v_2)\cdots\mathsf{type}(v_n) = \mathsf{in}\big(\mathsf{gate}(v)\big)$*.*

Note that in the above definition we followed the more common practice of only using single output gates and letting fan-out be an implicit operation represented by a gate having multiple outgoing edges. A more general definition would allow gates with multiple outputs and disallow implicit fanout, so that fanout can be controlled by what gates are allowed. The simplified definition is enough for our application, but e.g. quantum circuits would be better represented by a more general definition.

We would like to evaluate typed circuits, just like any other kind of circuit. To do this, we to need a semantics to define what each gate does.

Definition 7. *A* semantics $(\mathsf{values}, \mathsf{eval})$ *for a circuit prototype* $(\mathsf{types}, \mathsf{gates}, \mathsf{in}, \mathsf{out})$ *assigns each wire type $w \in$* types *a set of values* $\mathsf{values}(w)$*, and assigns each gate type $g \in$* gates *a function* $\mathsf{eval}(g)\colon \mathsf{values}(w_1)\times\mathsf{values}(w_2)\times\cdots\times\mathsf{values}(w_n) \to \mathsf{values}\big(\mathsf{out}(g)\big)$*, where $w_1\, w_2 \cdots w_n = \mathsf{in}(g)$ are the input wire types of the gate.*

We can evaluate a typed circuit using a semantics. Given values $x_v \in \mathsf{values}(\mathsf{type}(v))$ for all circuit inputs $v \in \mathsf{inputs}$, the evaluation proceeds in topological order. The inputs of each gate are its incoming wires, and the input order is given by the total order on the edges. Every gate $g \in \mathsf{nodes} \setminus \mathsf{inputs}$ gets evaluated as $x_g = \mathsf{eval}(\mathsf{gate}(g))(x_{v_1}, x_{v_2}, \ldots, x_{v_n})$ where $(v_1, g), (v_2, g), \ldots, (v_n, g) \in \mathsf{wires}$ are the incoming wires of g in sorted order. The outputs are then x_v for $v \in \mathsf{outputs}$. See Figure 2 for the formal algorithm.

```
$\mathsf{Run}(f, s, x)$:
    $(\mathsf{nodes}, \mathsf{wires}, \mathsf{inputs}, \mathsf{outputs}, \mathsf{type}, \mathsf{gate}) := f$
    $(\mathsf{values}, \mathsf{eval}) := s$
    for $v \in \mathsf{nodes} \setminus \mathsf{inputs}$ in topological order:
        $u :=$ empty list
        for $e \in \mathsf{wires}$ in sorted order:
            if $(w, v) = e$: append $w$ to $u$
        $x_v := \mathsf{eval}(\mathsf{gate}(g))(x_{u[1]}, x_{u[2]}, \ldots, x_{u[|u|]})$
    return $\{x_v\}_{v \in \mathsf{outputs}}$
```

Fig. 2: Algorithm for evaluating a circuit f with a semantics s. The circuit and the semantics must share the same circuit prototype.

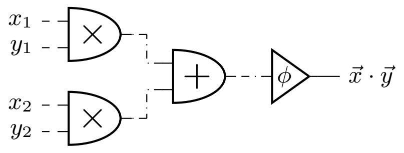


Fig. 3: A bounded RM circuit for computing the dot product of a pair of two element vectors. The new wire type MUL is drawn with a -·- line, and the new conversion operation ϕ with a triangle.

Restricted multiplication (RM) circuits. We can now define restricted multiplication circuits using the above definitions.

Definition 8. *The* RM circuit prototype *over a ring* $\mathbb{K}$ *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* $c \in \mathbb{K}$*, and a single nonlinear multiplication operation* $\times \colon \mathsf{IN} \times \mathsf{REG} \to \mathsf{REG}$*. An* RM circuit *is a circuit for this circuit prototype.*

An RM circuit can be evaluated just like an RMS program.

Definition 9. *The* evaluation semantics *for RM circuits over* $\mathbb{K}$ *has* $\mathsf{values}(\mathsf{IN}) = \mathsf{values}(\mathsf{REG}) = \mathbb{K}$ *and performs each gate operation in the ring* $\mathbb{K}$.

However, this is not the only semantics assigned to RM programs. In fact, our HSS definition is based on the idea of giving multiple different semantics to the same circuit: one for the plaintexts and one for the shares. The latter define what shares are and how homomorphic operations are evaluated on them.

Bounded RM circuits. Unfortunately, our construction will not be capable of evaluating all RM circuits. Similarly to [BGI16], we have a share conversion step that only works for values of bounded size. This conversion step is normally done on the output of every multiplication, but it can be delayed until after further linear operations. We generalize RM circuits with another wire type to represent unconverted values.

Definition 10. *The* bounded RM circuit prototype *over a ring* $\mathcal{R}$ *has wire types* $\mathsf{types} = \{\mathsf{IN}, \mathsf{REG}, \mathsf{MUL}\}$ *and gate types for* ***a)*** *the constant* 1 *for all* *wire types,* ***b)*** *linear operations for all wire types,* ***c)*** *a multiplication operation* $\times\colon \mathsf{IN} \times \mathsf{REG} \to \mathsf{MUL}$*, and* ***d)*** *a conversion operation* $\phi\colon \mathsf{MUL} \to \mathsf{REG}$*. A* bounded RM circuit *is a circuit for this circuit prototype.*

An example of this new kind of circuit is illustrated in Figure 3.

Definition 11. *The* evaluation semantics *for bounded RM circuits over* $\mathbb{K}$*, given a bound* $M \subseteq \mathbb{K}$*, sets* $\mathsf{values}(\mathsf{IN}) = \mathsf{values}(\mathsf{REG}) = \mathsf{values}(\mathsf{MUL}) = \mathbb{K} \cup \{\bot\}$ *and assigns the usual operations in* $\mathbb{K}$ *for linear operations and multiplication.* $\mathsf{eval}(\phi)(x)$ *is* x *if* $x \in M$*, or* $\bot$ *otherwise.* $\bot$ *is an absorbing element for all operations, so if any input is* $\bot$ *then the output is* $\bot$*.*

The value $\bot$ allows the circuit evaluation to fail if the input to the conversion operation isn't bounded. This idea is generalized by the following definition.

Definition 12. *A semantics* $(\mathsf{values}, \mathsf{eval})$ *is called a* failure semantics *if, for all wire types* $w \in \mathsf{types}$*, there is a special value* $\bot \in \mathsf{values}(w)$ *called* failure *that is absorbing for all functions in* $\mathsf{eval}(\mathsf{gates})$*. That is, for any* $g \in \mathsf{gates}$*,* $\mathsf{eval}(g)(\dots, \bot, \dots) = \bot$*, no matter what the other arguments are.*

The evaluation semantics of bounded RM circuits is a failure semantics.

3.2 Homomorphic Secret Sharing

Instead of taking a whole circuit to evaluate at once, our two-server HSS definition works piecemeal, by assigning three different semantics to the same circuit prototype. The first semantics is the usual one that works over the plaintexts, while the other two define, for each of the two servers, the what values the shares may take and how homomorphic operations may be computed on them. In a sense, these share semantics define compilers that turn the circuit into something that can be evaluated on shares, one gate at a time. The idea is that if we require that the plaintext semantics and share semantics be compatible with each other in a certain way, it implies that the homomorphic operations correctly evaluate the circuit to the sames result as if it were evaluated on the plaintext.

In our construction we are using Damgård–Jurik encryption, so $\mathbb{K}$ will be $\mathbb{Z}/N^s\mathbb{Z}$, which depends on the public key N and cannot be fixed in advance. This means that the operations we can perform have to be sampled randomly, at the same time as the public key, even though it is more usual to define homomorphic secret sharing in terms of some fixed operations (see e.g. [BGI+17]). Therefore, the plaintext evaluation will depend on the public key. We give the homomorphic operations access to shares of the secret key as well, as some of our operations (such as getting shares of 1) will depend on them.

Definition 13. *A* $(1-p)$-correct two-server Homomorphic Secret Sharing (HSS) scheme *with public-key setup consists of PPT algorithms:*

– $(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1) \leftarrow \mathsf{Setup}(1^\kappa)$ *outputs the keys and the circuit prototype, where* κ *is the security parameter.*

– $((\mathsf{types}, \mathsf{gates}, \mathsf{in}, \mathsf{out}), (\mathsf{values}, \mathsf{eval})) := \mathsf{Eval}(\mathsf{pk})$ *gives the circuit prototype and the plaintext evaluation semantics. This must be a failure semantics.*
– $(\mathsf{values}_j, \mathsf{eval}_j) := \mathsf{Hom}(j, \mathsf{pk}, \mathsf{sk}_j)$ *outputs the homomorphic evaluation semantics for server j, except that* eval_j *takes an extra argument r, which is a stream of random coins.*
– $(s_0, s_1) \leftarrow \mathsf{Share}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1, w, x)$*, given a wire type $w \in$* types *and a value $x \in \mathsf{values}(w)$, outputs shares $s_j \in \mathsf{values}_j(w)$.*
– $y \leftarrow \mathsf{Decode}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1, w, s_0, s_1)$ *decodes an output $y \in \mathsf{values}(w)$ from shares $s_j \in \mathsf{values}_j(w)$, where $w \in \mathsf{types}$.*

The following conditions are imposed.

– *Correctness: Running* Decode *on the shares from* Share *must output the original input x when x is not failure. More precisely, the following distribution outputs* TRUE *with probability at least $1 - p$, for any PPT adversary $\mathcal{A}$.*

```
$(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1) \leftarrow \mathsf{Setup}(1^\kappa)$
$(w, x) \leftarrow \mathcal{A}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1)$
$(s_0, s_1) \leftarrow \mathsf{Share}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1, w, x)$
$y \leftarrow \mathsf{Decode}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1, w, s_0, s_1)$
return $x \overset{?}{=} y \vee x \overset{?}{=} \perp$
```

– *Homomorphism: The semantics must commute with* Decode*. That is, the following distributions are indistinguishable except with advantage p, for any PPT adversary $\mathcal{A}$ such that the first distribution never returns $\perp$.*

```
$(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1) \leftarrow \mathsf{Setup}(1^\kappa)$
$(\mathrm{proto}, (\mathsf{values}, \mathsf{eval})) := \mathsf{Eval}(\mathsf{pk})$
$(\mathrm{view}, g, \{(s_{i0}, s_{i1})\}_i) \leftarrow \mathcal{A}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1)$
$r \leftarrow \$$
for $i := 1$ to $n$:
    $x_i \leftarrow \mathsf{Decode}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1, \mathsf{in}(g)_i, s_{i0}, s_{i1})$
$y := \mathsf{eval}(g)(x_1, \ldots, x_n)$
return $\mathrm{view}, r, y$
```

```
$(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1) \leftarrow \mathsf{Setup}(1^\kappa)$
$(\mathrm{proto}, (\mathsf{values}, \mathsf{eval})) := \mathsf{Eval}(\mathsf{pk})$
$(\mathrm{view}, g, \{(s_{i0}, s_{i1})\}_i) \leftarrow \mathcal{A}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1)$
$r \leftarrow \$$
$(\mathsf{values}_j, \mathsf{eval}_j) := \mathsf{Hom}(j, \mathsf{pk}, \mathsf{sk}_j), \forall j \in \{0, 1\}$
$s'_j := \mathsf{eval}_j(g, r)(s_1, \ldots, s_n), \forall j \in \{0, 1\}$
$y \leftarrow \mathsf{Decode}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1, \mathsf{out}(g), s'_0, s'_1)$
return $\mathrm{view}, r, y$
```

– *Privacy:* Share *must give each server no information about x. More precisely, we need the oracles $\mathcal{O}_{0,\mathsf{pk},\mathsf{sk}_0,\mathsf{sk}_1}$ and $\mathcal{O}_{1,\mathsf{pk},\mathsf{sk}_0,\mathsf{sk}_1}$ to be indistinguishable, for any PPT adversary $\mathcal{A}$ and any compromised server $j \in \{0, 1\}$.*

```
$\underline{\mathcal{O}_{i,\mathsf{pk},\mathsf{sk}_0,\mathsf{sk}_1}(w, x_0, x_1):}$
    $(s_0, s_1) \leftarrow \mathsf{Share}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1, w, x_i)$
    return $s_j$
```

Formally, $\Pr[(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1) \leftarrow \mathsf{Setup}(1^\kappa);\ \mathcal{A}^{\mathcal{O}_{i,\mathsf{pk},\mathsf{sk}_0,\mathsf{sk}_1}}(\mathsf{pk}, \mathsf{sk}_j) = 1]$ *must be negligibly different between $i = 0$ and $i = 1$.*

There are some important differences from the existing HSS definition such as [BGI+17]. In order to split the evaluation up into gates, we give a definition of homomorphism correctness that works on individual gates. We cannot simply use their definition for each gate, because their correctness property assumes that the shares input to Eval come directly from Share, not from other homomorphic operations. By allowing the shares to be chosen adversarially, we can accurately model online computation, where the adversary may dynamically choose what to evaluate based on the shares and keys.

However, HSS is not 100% correct. What's to stop the adversary from choosing shares (or even a sequence of gates that would generate those shares) that cause the HSS to fail? In [BGI16] this is solved by sampling a PRF provided to both parties, as part of Share, and using it to randomize the conversion operation. This works since the circuit is chosen before the shares. But our adversary gets to choose the shares, so we have to explicitly introduce into the homomorphism property a stream of randomness r that is sampled after the input shares have been determined. It could be instantiated with a shared PRG, reseeded whenever the circuit might be chosen adaptively based on the previous seed. If it were necessary to somehow adaptively change the circuit without using any communication at all, a random oracle evaluated on a description of the current gate and where the input shares came from would be an alternative.

For compatibility with existing constructions of HSS, we include an error probability p in our definition, even though in our HSS scheme p is negligible. The DDH-based construction of [BGI16] satisfies our definition with $p = \frac{1}{\mathrm{poly}(\kappa)}$. We do not prove this, but it should become clear that the same techniques we use to prove that our HSS scheme satisfies the definition would also work when applied to theirs. The LWE-based construction of [BKS19] should also work — this time with p a negligible function of κ.

The homomorphism property requires that decoding then performing a plaintext operation must work the same as doing the operation homomorphically, then decoding.[3] Why not go the other way round using Share and Hom, by requiring that the output of the homomorphic operation be indistinguishable from sharing the plaintext value? It turns out that this property is harder to achieve, as it is actually a form of circuit privacy. It asserts that the real distribution, where the shares are produced from a homomorphically evaluated circuit, is indistinguishable from an ideal distribution where the shares are simulated just using Share. Unfortunately, we cannot achieve this property because our construction involves holding shares of integers that may grow in size as they pass through the circuit. There's no way for Share to always produce shares of the right size.

Since our correctness and homomorphism definitions are in terms of performing a single operation, we need to prove that they can be composed into correctly evaluating a whole circuit.

[3] This structure may seem familiar to readers interested in category theory. In fact, we hit on these definitions by thinking of circuit semantics as functors. The homomorphism property then requires that Decode be a natural transformation from the homomorphic evaluation semantics to the plaintext evaluation semantics.

Lemma 14. *In any $(1-p)$-correct two-server HSS scheme, evaluating an arbitrary circuit on shares and then decoding the result vs. decoding the inputs and evaluating the circuit has distinguisher advantage at most np if the circuit has n gates. More precisely, following distributions are distinguishable with advantage at most np if the PPT $\mathcal{A}$ outputs a circuit f of at most n gates.*

```
$(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1) \leftarrow \mathsf{Setup}(1^\kappa)$
$(\text{proto}, \text{sem}_{pt}) := \mathsf{Eval}(\mathsf{pk})$
$(\text{view}, f, \{(s_{0v}, s_{1v})\}_v) \leftarrow \mathcal{A}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1)$
$(\mathsf{nodes}, \mathsf{wires}, \mathsf{inputs}, \mathsf{outputs}, \mathsf{type}, \mathsf{gate}) := f$
$r \leftarrow \$$
for $v \in \mathsf{inputs}$:
    $x_v \leftarrow \mathsf{Decode}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1, \mathsf{type}(v), s_{0v}, s_{1v})$
return view, $r$, $\mathsf{Run}(f, \text{sem}_{pt}, \{x_v\}_v)$
```

```
$(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1) \leftarrow \mathsf{Setup}(1^\kappa)$
$(\text{proto}, \text{sem}_{pt}) := \mathsf{Eval}(\mathsf{pk})$
$(\text{view}, f, \{(s_{0v}, s_{1v})\}_v) \leftarrow \mathcal{A}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1)$
$(\mathsf{nodes}, \mathsf{wires}, \mathsf{inputs}, \mathsf{outputs}, \mathsf{type}, \mathsf{gate}) := f$
$r \leftarrow \$$
for $j \in \{0, 1\}$:
    $s'_j := \mathsf{Run}(f, \mathsf{Hom}(j, \mathsf{pk}, \mathsf{sk}_j), \{s_{jv}\}_v, r)$
for $v \in \mathsf{outputs}$:
    $y_v \leftarrow \mathsf{Decode}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1, \mathsf{type}(v), s'_{0v}, s'_{1v})$
return view, $r$, $y$
```

In the second distribution, the extra parameter r to Run *represents giving each homomorphic gate evaluating its own piece of the random stream r.*

Proof. We give a hybrid proof starting from the right distribution and going to the left. Partition the circuit f into two parts g and h, where everything in g comes before everything in h in topological order. The circuit g is evaluated using Hom, then its outputs are fed into Decode and used to evaluate h in plaintext. Initially g is the whole circuit and h is nothing, and in each hybrid we shift a gate from g into h, picking a gate that comes last in topological order. The difference caused by the change is that before the gate was evaluated homomorphically, then decoded, while afterwards its inputs are decoded and then it is evaluated in plaintext. Since r is a freshly random string for each gate, the homomorphism property shows that this change has advantage at most p.

After all gates have been moved from g to h, we are at the left distribution. Since there are n gates to shift over, the total advantage is bounded by np.

An important property of our HSS scheme is that Decode authenticates its shares, at least for some wire types. More precisely, we can set up an experiment where shares are provided honestly to both the adversary and an honest server, the honest server performs some homomorphic operations on its shares, then they each provide an input to a decode operation. The adversary wins if it manages to obtain a different result than would be obtained with two honest servers.

Definition 15. *An HSS scheme is* authenticated *for wire types $A \subseteq$* types *if it is impossible for a single party to find a share of a wire type in A that decodes to a different result than would be obtained if they were honest. Formally, PPTs can only win* AuthGame *(Figure 4) with negligible probability.*

Some applications have a single trusted client, who can run the Share and Decode operations themselves. Others might not trust the client, or have numerous mutually distrusting clients and so need to implement these algorithms

```
AuthGame
INIT($j \in \{0,1\}$):
    $(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1) \leftarrow \mathsf{Setup}(1^\kappa)$
    $((\mathsf{types}, \mathsf{gates}, \mathsf{in}, \mathsf{out}), (\mathsf{values}, \mathsf{eval})) := \mathsf{Eval}(\mathsf{pk})$
    $(\mathsf{values}_k, \mathsf{eval}_k) := \mathsf{Hom}(k, \mathsf{pk}, \mathsf{sk}_k), \forall k \in \{0,1\}$
    $U, W :=$ empty list
    return $\mathsf{pk}, \mathsf{sk}_j$
SHARE($w \in \mathsf{types}, x \in \mathsf{values}(w)$):
    $(s_0, s_1) \leftarrow \mathsf{Share}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1, w, x)$
    append $(s_0, s_1)$ to $U$ and $w$ to $W$
    return $s_j$
EVAL($g \in \mathsf{gates}, i_1, \ldots, i_n$):
    assert $W[i_1]\, W[i_2] \cdots W[i_n] = \mathsf{in}(g)$
    $r \leftarrow \$$
    $s_k := \mathsf{eval}_k(g, r)(U[i_1]_k, \ldots, U[i_n]_k), \forall k \in \{0,1\}$
    append $(s_0, s_1)$ to $U$ and $\mathsf{out}(g)$ to $W$
    return $r$
GUESS($i, s_j \in \mathsf{values}_j(W[i])$):
    assert $W[i] \in A$
    $s_{\bar{j}} := U[i]_{\bar{j}}$
    $y \leftarrow \mathsf{Decode}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1, W[i], U[i]_0, U[i]_1)$
    $z \leftarrow \mathsf{Decode}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1, W[i], s_0, s_1)$
    win if $y \overset{?}{\neq} z \wedge y \overset{?}{\neq} \bot \wedge z \overset{?}{\neq} \bot$
```

Fig. 4: Game defining authentication for wire types $A \subseteq \mathsf{types}$. An adversary $\mathcal{A}$ is given oracle access to the interface of AuthGame, which emulates an honest party. $\mathcal{A}$ is required to call INIT exactly once, before calling anything else in AuthGame, and only wins by making a successful call to GUESS.

with MPC. We define a couple special cases where these operations can be implemented more easily, without the need for generic MPC.

Definition 16. *A two-server HSS scheme has* public-key sharing *if there is a UC secure 3-party protocol to compute* $(s_0, s_1) \leftarrow \mathsf{Share}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1, w, x)$*, where* x *is provided by the client,* sk_j *is input by server* j*, all parties know* pk *and* w*, and* s_j *is output to server* j*. All protocol messages must come from the client.*

Definition 17. *A two-server HSS scheme has* additive decoding *for wire type* w *if* $\mathsf{values}(w)$ *is an abelian group and there are PPT algorithms* f_0, f_1 *such that*

$$\mathsf{Decode}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1, w, s_0, s_1) = f_1(\mathsf{pk}, \mathsf{sk}_1, s_1) - f_0(\mathsf{pk}, \mathsf{sk}_0, s_0).$$

4 Main Construction

4.1 Distance Function

Similarly to [BGI16], share conversion for our HSS scheme works by picking a subset of ciphertexts to be "special", and measuring the "distance" from the nearest special point. Here "distance" means the number of times some generator must be divided to reach the special point. We pick the subset of values in $\left[-\frac{N}{2}, \frac{N}{2}\right)$ to be special, i.e. those $c \in \mathbb{Z}/N^{s+1}\mathbb{Z}$ where $c = c \bmod N$. The generator in our case is $\exp(1)$. The only special point that can be reached is $c \bmod N$ because $\exp(1) \bmod N = 1$. This choice of generator allows the distance to be computed efficiently using log.

$$\begin{aligned}\mathrm{Dist}_{N,s} \colon (\mathbb{Z}/N^{s+1}\mathbb{Z})^\times &\to \mathbb{Z}/N^s\mathbb{Z} \\ c &\mapsto \log\Bigl(\frac{c}{c \bmod N}\Bigr)\end{aligned}$$

This is justified by the following theorem, which shows that $\mathrm{Dist}_{N,s}$ preserves the distance between two ciphertexts.

Theorem 18. *For any $c \in (\mathbb{Z}/N^{s+1}\mathbb{Z})^\times$ and $x \in \mathbb{Z}/N^s\mathbb{Z}$,*

$$\mathrm{Dist}_{N,s}(c\exp(x)) - \mathrm{Dist}_{N,s}(c) = x.$$

Proof. First, we need to show that $\mathrm{Dist}_{N,s}(c)$ is always well defined. We have $\frac{c}{c \bmod N} \equiv_N \frac{c}{c} \equiv_N 1$, so $\log\bigl(\frac{c}{c \bmod N}\bigr)$ is well defined. Then,

$$\begin{aligned}&\mathrm{Dist}_{N,s}(c\exp(x)) - \mathrm{Dist}_{N,s}(c) \\ &= \log\left(\frac{c\exp(x)}{c\exp(x) \bmod N}\right) - \log\Bigl(\frac{c}{c \bmod N}\Bigr) \\ &= \log\left(\frac{c\exp(x)}{c \bmod N}\right) - \log\Bigl(\frac{c}{c \bmod N}\Bigr) = x.\end{aligned}$$

Note that we have only shown the correctness of the distance function modulo N^s. Our construction will in fact need to convert its outputs to be in $\mathbb{Z}$, as there is no consistent way to exponentiate to a power that is in $\mathbb{Z}/N^s\mathbb{Z}$ when the multiplicative order of the base does not divide N^s. The following lemma will be used to show that using $\cdot \bmod N^s$ to convert shares to $\mathbb{Z}$ works with all but negligible probability.

Lemma 19. *For any $N \in \mathbb{Z}^+$, $x \in \mathbb{Z}$, and uniformly random $r \in \mathbb{Z}/N\mathbb{Z}$, we have*

$$\Pr\bigl[x = (r + x) \bmod N - r \bmod N\bigr] = \max\left(1 - \frac{|x|}{N}, 0\right).$$

Proof. The condition may equivalently be written as

$$r \bmod N + x = \bigl(r \bmod N + x\bigr) \bmod N.$$

This clearly holds if and only if $-\frac{N}{2} \le r \bmod N + x < \frac{N}{2}$, i.e. if it is already reduced so taking the modulus will not change it. If $x \ge 0$ then this is equivalent to $r \in [-\frac{N}{2}, \frac{N}{2} - x)$, which contains $N - x$ (or none, if $x > N$) of the N possible integer values for $r \bmod N$. The case of negative x is symmetric, so the probability is either $\frac{N-|x|}{N} = 1 - \frac{|x|}{N}$, or 0 if it would otherwise be negative.

```
Setup($1^\kappa$):
    $(N, \varphi) \leftarrow \mathsf{DJ.KeyGen}(1^\kappa)$
    $\varphi_0 \leftarrow [0, N)$
    $\varphi_1 := \varphi_0 + \varphi$
    return $N, \varphi_0, \varphi_1$

Share($N, \varphi_0, \varphi_1, \mathsf{IN}, x$):
    $c \leftarrow \mathsf{DJ.Enc}_{N,s}(x_j)$
    return c, c
Share($N, \varphi_0, \varphi_1, \mathsf{REG}, x$):
    $s_0 \leftarrow [0, N^{s+1} 2^\kappa)$
    $x' := x \bmod N^s$
    return $s_0, s_0 + (\varphi_1 - \varphi_0)x'$
Share($N, \varphi_0, \varphi_1, \mathsf{MUL}, x$):
    $s_0 \leftarrow \mathbb{Z}/N^s\mathbb{Z}$
    return $s_0, s_0 + (\varphi_1 - \varphi_0)x$
```

```
$\mathsf{values}_j(\mathsf{IN}) = (\mathbb{Z}/N^{s+1}\mathbb{Z})^\times$
$\mathsf{values}_j(\mathsf{REG}) = \mathbb{Z}$
$\mathsf{values}_j(\mathsf{MUL}) = \mathbb{Z}/N^s\mathbb{Z}$
$\mathsf{eval}_j(\times, r)(c, s_j) = \mathrm{Dist}_{N,s}(c^{s_j})$
$\mathsf{eval}_j(\phi, r)(s_j) = (s_j + r) \bmod N^s$

Decode($N, \varphi_0, \varphi_1, \mathsf{IN}, s_0, s_1$):
    if $s_0 \neq s_1$: return $\perp$
    return $\mathsf{DJ.Dec}_{N,s,\varphi_1-\varphi_0}(s_0)$
Decode($N, \varphi_0, \varphi_1, \mathsf{REG}, s_0, s_1$):
    if $s_1 - s_0 \notin (\varphi_1 - \varphi_0)\mathbb{Z}$:
        return $\perp$
    return $(s_1 - s_0)/(\varphi_1 - \varphi_0) + N^s\mathbb{Z}$
Decode($N, \varphi_0, \varphi_1, \mathsf{MUL}, s_0, s_1$):
    return $(s_1 - s_0)/(\varphi_1 - \varphi_0)$
```

Fig. 5: Our HSS scheme for bounded RM circuits. In the top left the encryption is setup and the secret key shared between the two parties. The secret share sets are in the top right, along with the non-trivial homomorphic that may be performed on them. The linear operations are given by the abelian group structure that the shares are in, so we omit them. Share and Decode for the three types of shares are shown in the bottom.

4.2 HSS Construction

Now we have everything required to define our main HSS scheme, which will be parameterized by a ciphertext size s and a bound M on the values. To start, we generate a random Damgård–Jurik key pair (N, φ) and share φ between the two parties in Setup (Figure 5). The plaintext evaluation semantics $\mathsf{Eval}(N)$ are then the evaluation semantics (Definition 11) for bounded RM circuits over $\mathbb{Z}/N^s\mathbb{Z}$ bounded in $[-M, M]$.

Our three types of shares of a value x will be ciphertexts in $(\mathbb{Z}/N^{s+1}\mathbb{Z})^\times$, additive shares of φx in $\mathbb{Z}$, and additive shares of φx in $\mathbb{Z}/N^s\mathbb{Z}$ (see values in Figure 5). We let Share encrypt or generate these shares and Decode decrypt or decode them, while checking for consistency between the two parties' shares. The share types are all abelian groups, allowing the circuit's linear operations to be defined on the shares easily. We omit these, other than noting that constructing 1_{REG} and 1_{MUL} requires secret shares of the private key φ. In fact, additive secret shares of φ are exactly the same as our REG and MUL shares of 1.

The homomorphic multiplication function $\mathsf{eval}_j(\times, r)$ in Figure 5 is based on $c^{\varphi x}$ essentially decrypting x times the plaintext, so when performed on additive shares s_0, s_1 of φx this gives multiplicative shares of the decryption. We then use the distance function to convert them to additive shares. As these shares are only in $\mathbb{Z}/N^s\mathbb{Z}$, we define $\mathsf{eval}_j(\phi, r)$ to pick a representative in $\mathbb{Z}$, allowing the result to be converted to shares in $\mathbb{Z}$.

Theorem 20. *Figure 5 describes a $(1 - MN^{1-s})$-correct HSS scheme (Definition 13) under DCR.*

Proof. There are three properties to be proved.

Correctness: For the IN wire type, this is just the correctness of Damgård–Jurik encryption. For REG and MUL we have $s_1 - s_0 = (\varphi_1 - \varphi_0)x$, so dividing out $\varphi_1 - \varphi_0$ inside Decode gives the correct decoding.

Homomorphism: We omit the trivial proofs for the linear operations allowed in bounded RM circuits. For multiplication, we have

$$\frac{c^{s_1}}{c^{s_0}} = c^{s_1 - s_0} = c^{\varphi y} = \exp(\varphi x)^y = \exp(\varphi xy),$$

where $x = \mathsf{DJ.Dec}_{N,s,\varphi}(c)$ and $y = \frac{s_1 - s_0}{\varphi}$ are the two input share decodings. Then Theorem 18 shows that $\mathsf{eval}_1(\times, r)(c, s_1) - \mathsf{eval}_0(\times, r)(c, s_0) = \varphi xy$.

The correctness of share conversion $\mathsf{eval}_j(\times, r)(\phi)$ with probability $1 - \frac{\varphi M}{N^s}$ follows directly from Lemma 19. Adding r to both shares before taking the modulus guarantees that s_0 is uniformly random, as is required by the lemma, and does not change $s_1 - s_0 \equiv_{N^s} \varphi x$. This is the only step with imperfect correctness, so because $\varphi < N$ we get that the overall scheme is $(1 - MN^{1-s})$-correct.

Privacy: We must show that Share leaks nothing about the value being shared to any individual server. We present a hybrid proof, starting with the adversary $\mathcal{A}$ having access to $\mathcal{O}_{0,\mathsf{pk},\mathsf{sk}_0,\mathsf{sk}_1}$, and ending $\mathcal{A}$ accessing $\mathcal{O}_{1,\mathsf{pk},\mathsf{sk}_0,\mathsf{sk}_1}$.

1. Use dummy shares of 0 in Share for wire types REG and MUL. For MUL, s_0 and s_1 individually are uniformly random, independent of x, so this is indistinguishable to the adversary, who only gets to see s_j. Similarly, the distribution for s_0 when sharing a REG value does not depend on x, while s_1 is uniform in the range $[\varphi x', \varphi x' + N^{s+1}2^\kappa)$, which is statistically indistinguishable from being uniform in $[0, N^{s+1}2^\kappa)$ because the distributions are identical in all but a negligible fraction $\frac{|\varphi x'|}{N^{s+1}2^\kappa} < 2^{-\kappa}$ of the possibilities. After this change, φ is unused by Share.
2. Instead of setting $\varphi_1 = \varphi_0 + \varphi$, sample $\varphi_1 \leftarrow [N, 2N)$. This is indistinguishable because φ_0 is uniform in $[0, N)$, the adversary only gets to see φ_j, and $[N, 2N)$ and $[\varphi, \varphi + N)$ overlap in all but $N - \varphi = p + q - 1$ out of N possibilities. Therefore, the adversary has advantage at most $\frac{p+q-1}{N} \leq \frac{2^{\ell(\kappa)+1}}{2^{2(\ell(\kappa)-1)}} = 2^{-\ell(\kappa)+3}$, which is negligible.
3. Notice that the private key φ is now totally unused. Therefore, swapping the oracle to $\mathcal{O}_{1,\mathsf{pk},\mathsf{sk}_0,\mathsf{sk}_1}$ is indistinguishable. Specifically, Share for wire types REG and MUL already ignores its input, while for wire type IN, Theorem 3 shows that Share encrypts its input securely.
4. Undo hybrids 2 and 1. We are now at a distribution where $\mathcal{A}$ is given oracle access to $\mathcal{O}_{1,\mathsf{pk},\mathsf{sk}_0,\mathsf{sk}_1}$, and Setup and Share once more have their real implementations.

We proved that each operation in our HSS scheme has an error rate of at most MN^{1-s}. Normally s should be chosen to be the smallest such that $MN^{1-s} \leq 2^{-k}$, to make the error rate negligible. For many applications (including ORAM), $M \leq 2^{-\kappa}N$, in which case $s = 2$ is most efficient. Concretely, at the 128-bit security level $N \approx 2^{3072}$, so $s = 2$ is sufficient for plaintexts of up to 2944 bits.

Authentication. Shares of type IN are trivially authenticated, as both parties always have the same share. REG values are always multiples of φ, so to create a fake share the adversary would have to guess a multiple of φ to offset their share by. Finding a multiple of φ would give an attack against privacy.

Theorem 21. *The HSS scheme in Figure 5 is authenticated for wire types* $\{\mathsf{IN}, \mathsf{REG}\}$.

Proof. See the full version of this paper.

Public-key sharing. Our construction also satisfies public-key sharing (Definition 16). This is easiest to see for IN shares, because they are just encryptions under the public key N. We can build public-key sharing for the other share types from this. To share out a MUL share of x, just give out IN shares of x, then run the RM circuit to compute $x \times 1_{\mathsf{REG}}$, which produces MUL wire type shares. Finally, REG shares of x can be given out by splitting x into pieces small enough to guarantee that ϕ will succeed (so $x = \sum_i x_i M^i$), then doing public key sharing on every x_i. Then they are converted back to REG type with ϕ, and $x = \sum_i x_i M^i$ is then computed inside an RM circuit. Note that in all cases the client only needs to send a message to both servers, who then do some local computation to find the shares.

4.3 Additive Decoding

Notice how in the previous HSS scheme, decoding REG shares is almost additive. The only flaw is that we need to divide by φ. With circular security we could simply encrypt φ^{-1} and multiply it as the last step. It's a little trickier without.

Instead, we generate a second key (N', φ') and use it to encrypt $\varphi^{-1} \bmod N'^{s'}$, avoiding the need for a circular security assumption. But then how do we decrypt this ciphertext? If the shares were multiplied by $\varphi'(\varphi'^{-1} \bmod N'^{s'})$ then during decryption the $\varphi'\varphi'^{-1}$ would cancel, since it is modulo $N'^{s'}$, and similarly φ would cancel with φ^{-1}. But $\varphi'^{-1} \bmod N'^{s'}$ is nearly as large as $N'^{s'}$, requiring s to be around double s' and making the scheme less efficient.

There's a trick to avoid this, however. Let $\nu = N'^{-s'} \bmod \varphi'$. Then,

$$1 - N'^{s'}\nu \equiv_{\varphi' N'^{s'}} \varphi'(\varphi'^{-1} \bmod N'^{s'})$$

by the Chinese remainder theorem, since modulo φ' they are both 0, and modulo $N'^{s'}$ they are both 1. Therefore, $1 - N'^{s'}\nu$ is just as good for decoding the result, because the final decryption is of a ciphertext in $(\mathbb{Z}/N'^{s'}\mathbb{Z})^\times$, which has order $\varphi' N'^{s'}$. If for every value x we maintain shares of φx and $\varphi\nu x$ (which are both relatively small), we can do additive decoding by first computing shares

```
$\mathsf{values}_j(\mathsf{REG}) = \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$
$\mathsf{Setup}(1^\kappa)$:
    $(N, \varphi) \leftarrow \mathsf{DJ.KeyGen}(1^\kappa)$
    $(N', \varphi') \leftarrow \mathsf{DJ.KeyGen}(1^\kappa)$
    $\mu := \varphi^{-1} \bmod N'^{s-2}$
    $\nu := \left(N'^{s-2}\right)^{-1} \bmod \varphi'$
    $c' \leftarrow \mathsf{DJ.Enc}_{N', s-2}(\mu)$
    $\varphi_0, \varphi'_0 \leftarrow [0, NN'2^\kappa)$
    return $(N, N', c'), (\varphi_0, \varphi'_0), (\varphi_0 + \varphi, \varphi'_0 + \varphi\nu)$

$\mathsf{Share}(\ldots, \mathsf{REG}, x)$:
    $x' := x \bmod N^s$
    $s_0, s'_0 \leftarrow [0, N^{s+1}N'2^\kappa)$
    $v_0 \leftarrow [0, N^s 2^\kappa)$
    $s_1 := s_0 + (\varphi_1 - \varphi_0)x'$
    $s'_1 := s'_0 + (\varphi'_1 - \varphi'_0)x'$
    return $(s_0, s'_0, v_0), (s_1, s'_1, v_1)$
$\mathsf{Share}(\ldots, \mathsf{MUL}, x)$:
    $(s_0, s'_0, v_0), (s_1, s'_1, v_1) \leftarrow \mathsf{Share}(\ldots, \mathsf{REG}, x)$
    return $(s_0, s'_0), (s_1, s'_1)$
```

```
$\mathsf{values}_j(\mathsf{MUL}) = \mathbb{Z}/N^s\mathbb{Z} \times \mathbb{Z}/N^s\mathbb{Z}$
$\mathsf{eval}_j(\times, r)(c, (s_j, s'_j))$:
    return $\mathrm{Dist}_{N,s}(c^{s_j}), \mathrm{Dist}_{N,s}(c^{s'_j})$
$\mathsf{eval}_j(\phi, r \parallel r' \parallel r'')((t_j, t'_j))$:
    $s_j := (t_j + r) \bmod N^s$
    $s'_j := (t'_j + r') \bmod N^s$
    $v_j := \mathrm{Dist}_{N', s-2}\Big(c'^{s_j - N'^{s-2}s'_j}\Big)$
    $v_j := (v_j + r'') \bmod N'^{s-2}$
    return $(s_j, s'_j, v_j)$

$\mathsf{Decode}(\ldots, \mathsf{REG}, (s_0, s'_0, v_0), (s_1, s'_1, v_1))$:
    if $(s_1 - s_0) \neq (\varphi_1 - \varphi_0)(v_1 - v_0)$
        $\vee\ (s'_1 - s'_0) \neq (\varphi'_1 - \varphi'_0)(v_1 - v_0)$:
        return $\perp$
    return $v_1 - v_0 + N^s\mathbb{Z}$
$\mathsf{Decode}(\ldots, \mathsf{MUL}, (s_0, s'_0), (s_1, s'_1))$:
    $\nu := (\varphi'_1 - \varphi'_0)/(\varphi_1 - \varphi_0)$
    if $(s'_1 - s'_0) \neq \nu(s_1 - s_0)$:
        return $\perp$
    return $(s_1 - s_0)/(\varphi_1 - \varphi_0)$
```

Fig. 6: Modifications to the HSS scheme in Figure 5 needed to support additive decoding. Only those functions that have been modified are shown. For compactness, the public and private keys in Share and Decode have been omitted with an ellipsis, rather than writing out $(N, N', c'), (\varphi_0, \varphi'_0), (\varphi_1, \varphi'_1)$ every time.

of $\varphi(1 - N'^{s'}\nu)x$, then doing a final multiply by the encryption of φ^{-1} to get additive shares of x.

We show the modified HSS scheme in Figure 6. Setup now computes the second key pair (N', φ') and gives out an encryption c' of $\mu = \varphi^{-1}$ under the second key. It also returns secret shares $\varphi'_1 - \varphi'_0 = \varphi\nu$ alongside the secret shares of φ. The REG shares have the biggest changes. Not only does they keep track of shares of both φx and $\varphi\nu x$, but they also keep shares (v_0, v_1) of x. This is because the only time we have an upper bound on the size of a plaintext value x is during $\mathsf{eval}_j(\phi, r)$, so we compute additive shares of x then and cache them. The MUL shares also needed to be changed to keep shares of both φx and $\varphi\nu x$. We set $s' = s - 2$ because the additive decoding value x is much smaller (by a factor of nearly NN') than $\varphi\nu x$. The former is computed modulo N'^{s-2} while the latter is found modulo N^s, which makes the error probabilities similar.

Theorem 22. *Assuming DCR, the modified scheme in Figure 6 is a $(1 - p)$-correct HSS scheme that is authenticated for all wire types and has additive decoding for* REG *wires, where $p = MN'\big(N^{1-s} + N'^{1-s}\big)$.*

Proof. See the full version of this paper.

Choosing s to achieve a negligible error rate is essentially the same as for the previous construction. Because N and N' are of approximately the same size,

s should be chosen such that $p \approx 2MN^{2-s} \leq 2^{-k}$. Roughly, s just needs to be incremented. Public key sharing works for this new protocol in exactly the same way as before, since we did not change the sharing process for IN shares, and public key sharing of everything else was based on that one share type.

5 Distributed Oblivious RAM

An oblivious RAM (ORAM) allows a client to outsource its data (a sequence of N blocks) to an untrusted server, such that it can access any sequence of data blocks on the server while hiding the access pattern [Ost92,Gol87]. While traditionally ORAM protocols were designed assuming a single server which stores data passively, recent works have considered more general settings, allowing for multiple servers with computational capabilities [DvDF$^+$16,HOY$^+$17,FNR$^+$15]. Given the result in [Gol87], all passive server ORAM protocols incur at least $\Omega(\log N)$ bandwidth overhead. However, if we allow for server side computation, constant bandwidth blowup can be achieved for large block sizes [DvDF$^+$16]. In this section we propose a new malicious secure ORAM construction based on 2 party HSS. Our construction achieves constant bandwidth blowup for blocks of size at least $\Omega(\log^4 N)$ bits.

5.1 Definition: Distributed ORAM

We consider a 3 party distributed ORAM model with a single client and 2 non-colluding servers. All the parties maintain a state (st_c, st_{s0}, st_{s1}) which is updated after each ORAM operation, where st_c is the client state and st_{s0}, st_{s1} are the states of the two servers respectively.

Definition 23. *A distributed 2-server ORAM construction with security parameter κ consists of the following two interactive protocols:*

– $(st'_c, st'_{s0}, st'_{s1}) \leftarrow \mathsf{Setup}(D)$: The client inputs an N sized array D of blocks, where each block is of length B bits. This function initializes the ORAM with the array D.
– $(data, st'_c, st'_{s0}, st'_{s1}) \leftarrow \mathsf{Access}(in, st_c, st_{s0}, st_{s1})$: The client receives as input an ORAM operation $in = (op, idx, data)$, where $op = \{read, write\}$, $idx \in [1 \dots N]$ and $data \in \{0,1\}^B \cup \{\perp\}$. If $op = read$ then the client should return the block $D[idx]$. If $op = write$, then this protocol should update the content of block $D[idx]$ in the ORAM with $data$.

We use the simulation based definition for a malicious secure ORAM as was considered in [DvDF$^+$16] (see the full version of this paper for details).

5.2 An Overview of Bounded Feedback and Onion ORAM

Our protocol is inspired by the Onion ORAM protocol proposed in [DvDF$^+$16], which in turn is based on the passive server Bounded Feedback ORAM protocol from the same paper. In this subsection we describe Bounded Feedback ORAM and how it can be modified to give the single server Onion ORAM construction.

Bounded Feedback ORAM Similar to other tree-based ORAMs, its server memory is organized in the form of an L depth binary tree T, where each node of the tree (also referred to as a *bucket*) contains Z blocks. The leaves of the tree are numbered from 0 to $2^L - 1$. $\mathcal{P}(l)$ represent the blocks on the path to leaf l on this tree and $\mathcal{P}(l, k)$ represents the k^{th} bucket from the root node on this same path respectively.

As is the case for all tree based ORAMs, each block is mapped to a unique random leaf node in this tree. And this mapping is stored in a position map (PosMap) by the client. The **key invariant** that's maintained is that each block (with index $addr$) is present in some bucket on the path $\mathcal{P}(\mathsf{PosMap}[addr])$.

For each block in the tree, the server also stores the corresponding metadata $(addr, label)$, where $addr$ is the logical address of the block and $label = \mathsf{PosMap}[addr]$. The corresponding metadata tree is referred to as md. We use the shorthand $\mathsf{md}[l]$ to represent the list of all metadata present on the path l in md.

ORAM Access To read/write a block $addr$ the client looks up the corresponding leaf label $\mathsf{PosMap}[addr]$ from the position map. It further downloads all the blocks on the path $\mathsf{PosMap}[addr]$ in tree T from the server. The client can now locally read and update the block $addr$. The block $addr$ is remapped to a new random leaf label and is inserted in the root bucket. All the downloaded blocks on path l are re-encrypted and stored back on the server. To ensure that no bucket overflows except with negligible probability, after every A (a parameter) Access operation the blocks are percolated towards the leaves in the tree while maintaining the key invariant. This process is also called the **eviction algorithm**. Most tree based ORAMs often differ in their eviction procedures.

Triplet Eviction Algorithm As is the case for other tree based ORAMs, eviction is performed along a specific path (let say l). For $k = 0$ to L, the algorithm pushes all the blocks in bucket $\mathcal{P}(l, k)$ into one of its two children buckets. This process can be carried out without violating the key invariant. After every A ORAM accesses, the next eviction path is chosen in the reverse lexicographic order of G (a variable), which is initialized to 0 and incremented by 1 after each eviction procedure. Given the analysis in [DvDF+16], the parameters $Z = A = \Theta(\lambda)$ ensure negligible overflow probability for each bucket, where λ is the statistical security parameter.

Recursion Storing the position map requires space $O(N \log N)$. To avoid the large client memory, we can recursively store the position map in a smaller ORAM on the server. This recursive approach used in all tree based ORAMs does not incur any additional asymptotic cost for blocks of size $\Omega(\log^2 N)$, where N is the size of the database. For all the ORAM protocols we describe ahead, we will ignore the cost of recursion for larger block sizes.

How Onion ORAM Differs The Onion ORAM protocol is similar to the Bounded Feedback ORAM with the key distinction that all computation on the data blocks is performed by the server locally. For this purpose the ORAM data structure is encrypted using an additively homomophic encryption scheme, which

```
$\textsc{select}(i \in \mathsf{IN}, y_0 \in \mathsf{REG}, \ldots, y_{m-1} \in \mathsf{REG})$:
    $D[j] := \frac{(m-1)!}{j!} \sum_{k=0}^{j} \binom{j}{k} (-1)^{j-k} \times_{\mathsf{REG}} y_k$
    $z := 0_{\mathsf{REG}}$
    for $j := m-1$ to $0$:
        $z := z +_{\mathsf{REG}} D[j]$
        $z := \phi(z \times (i +_{\mathsf{IN}} (1-j)))$
    $z := \frac{1}{(m-1)!} \times_{\mathsf{REG}} z$
    return $z$
```

```
$\mathsf{SelectShare}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1, i, \{y_{0k}\}_k, \{y_{1k}\}_k)$:
    $in[“i”] := \mathsf{Share}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1, \mathsf{IN}, i)$
    for $j \in \{0,1\}$:
        for each chunk index $c$:
            for $k := 0$ to $m-1$:
                $in[“y” \parallel k] := y_{jk}[c]$
            $s_j[c] := \mathsf{Run}(\textsc{select}, \mathsf{Hom}(j, \mathsf{pk}, \mathsf{sk}_j), in)$
    return $s_0, s_1$
$\mathsf{Select}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1, i, \{y_{0k}\}_k, \{y_{1k}\}_k)$:
    $(s_0, s_1) \leftarrow \mathsf{SelectShare}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1,$
                                    $i, \{y_{0k}\}_k, \{y_{1k}\}_k)$
    for $j \in \{0,1\}$:
        $s'_j := \sum_c M'^c s_j[c]$
    $z' := \mathsf{Decode}'(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1, \mathsf{REG}, s'_0, s'_1)$
    for each chunk index $c$:
        $z[c] := \lfloor \frac{z'}{M'^c} \rfloor \bmod M'$
    return $z$
```

Fig. 7: Left: Selection operation pseudocode. The pseudocode follows the wire-type rules of a bounded RM circuit, and could easily be unrolled into a circuit. Right: The distributed Select algorithm, which runs the SELECT RM circuit on the given shares, then decodes the result to find y_i. Client computation is colored red and server computation is colored blue. Because in our HSS the REG secret shares do not depending at all on the ciphertext size parameter s, we can pack together several shares (by treating them as a base M' number) and decode them all at once, which reduces the overhead of the secret sharing step. However, we need Decode from Figure 5 to be modified slightly, to not take its output modulo N^s, and we call this modification Decode$'$.

allows the server to perform access and evict algorithms locally. More details on the Onion ORAM construction can be found in full version of this paper.

5.3 Our HSS based ORAM construction

In our construction the server side computation of Onion ORAM is divided across 2 non-colluding servers using our HSS construction.

The two servers store two ORAM binary trees (T_0, T_1) respectively, and they also have additive shares of authenticated meta-data $(\mathsf{md}, H(\mathsf{md}))$ corresponding to each block in the tree. Each block b in our scheme is a sequence of chunks $(b_1, b_2, \ldots, b_C)$ (for some parameter C), where each chunk can be secret shared as wires of type REG using HSS.

The server side computation in Onion ORAM can be replaced with homomorphic computation on the HSS shares by the two servers, where the client sends an encrypted index as a wire type IN. For the eviction procedure, we conceptually use the same technique as used in Onion ORAM, which uses $\Theta(ZL)$ select operations. We next describe the selection and evict algorithms in a little more detail.

Selection. An advantage of using HSS is being able to evaluate a limited kind of arithmetic circuit, so we can encode more than just a single bit in a ciphertext. In fact, we can do a 1-of-m select operation by sending just a single ciphertext to the servers. Suppose we want to select the ith element of a sequence $y_0, \ldots, y_{m-1}$, for

```
$\mathsf{Evict}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1, l_e, \mathsf{md}, \{x_{0k}\}_k, \{x_{1k}\}_k, \{y_{0k}\}_k, \{y_{1k}\}_k)$:
  $remap :=$ array of zeros
  for each block $b$ of parent node $\lfloor \frac{l_e}{2} \rfloor$.
    if $\mathsf{md}$ says $b$ is present and needs to move to $l_e$:
      find next empty location $b'$ in $l_e$
      $remap[b'] := b$
  for each block $b$ of node $l_e$:
    $in_j := (y_{jb}, x_{j0}, \ldots, x_{j(Z-1)}) \forall j \in \{0, 1\}$
    $y_{0b}, y_{1b} \leftarrow \mathsf{SelectShare}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1, remap[b], in_0, in_1)$
  return $\{y_{0k}\}_k, \{y_{1k}\}_k$
```

Fig. 8: The distributed Evict algorithm. Inputs are l_e, the location of the node to evict into, the shares $\{x_{0k}\}_k, \{x_{1k}\}_k$ of the blocks in the parent node of l_e, and shares $\{y_{0k}\}_k, \{y_{1k}\}_k$ of node l_e.

some $i \in [0, m-1]$. Then if we interpolate a polynomial $p(X)$ through the points $p(0) = y_0, \ldots, p(m-1) = y_{m-1}$, then we can evaluate $p(i)$ to find y_i. Polynomial interpolation is a linear operation, and so can be performed separately by each server, on its own share of $\{y_i\}_i$.

However, there's one small issue that we've skipped over. We can only evaluate *bounded* RM circuit, and representing a fraction in the ring is very likely to produce a large number that is outside of the bound. We instead use the Newton polynomial interpolation, representing p as

$$p(X) = \sum_{j=0}^{m-1} \frac{\Delta^j[y]}{j!}(X)_j \qquad \text{where} \qquad \Delta^j[y] = \sum_{k=0}^{j} \binom{j}{k} (-1)^{j-k} y_k,$$

where $(X)_j = X(X-1)\cdots(X-j+1)$ is the falling factorial. Although we only show the direct formula for computing the differences $\Delta^j[y]$, faster FFT-based methods would also work. Notice that the finite differences $\Delta^j[y]$ are all integers, so only need to evaluate $(m-1)!\, p(i)$ to remove all of the fractions, and then divide by $(m-1)!$ at the last step, which works since $p(i)$ is an integer. We can evaluate this polynomial using a variant of Horner's rule, which is efficient inside an RM circuit (Figure 7).

$$p(X) = \left(\left(\frac{\Delta^{m-1}[y]}{(m-1)!}(X - m + 2) + \frac{\Delta^{m-2}[y]}{(m-2)!} \right) (X - m + 3) + \cdots \right) X + \frac{\Delta^0[y]}{0!}$$

We need to compute a size bound M on the values in this computation, given the known bound M' on every y_i. We have $\left|\Delta^j[y]\right| \leq M' \sum_k \binom{j}{k} = 2^j M'$. Let S be a subexpression in the evaluation of $(m-1)!\, p(X)$. Then $|S| \leq \sum_{j=0}^{m-1} \left| \frac{(m-1)!}{j!} \Delta^j[y] m^j \right|$, because every $(x - j + 1) \leq m$, and going from S to this we only add more nonnegative terms and multiply more factors of $m \geq 1$. This can be turned into an upper bound, which we will use to set M.

$$|S| \leq \sum_{j=0}^{m-1} \frac{(m-1)!}{j!} 2^j m^j M' \leq (m-1)!\, M' \sum_{j=0}^{\infty} \frac{(2m)^j}{j!} = (m-1)!\, M' e^{2m} \leq M \quad (1)$$

Eviction. We need to move up to Z blocks in a parent node in the tree into a child node, which has locations for Z blocks. We do this by performing Z instances of 1-of-$(Z+1)$ Select, allowing each block location in a child node to select any of its parent node's blocks, or its existing value if it was already filled. This algorithm is shown in Figure 8.

Using these two algorithms, we describe our Setup and Access function for our proposed ORAM scheme in Figure 9 and Figure 10 respectively.

Our ORAM construction can also be used for implementing 2 party secure computation of RAM programs. In full version of this paper we further discuss how the public-key sharing property (Definition 16) and additive decoding (Definition 17) of our HSS scheme help realize this protocol more efficiently.

5.4 Proof of Security

Intuitively, the adversary learns nothing looking at one server's binary tree data - which consists of one share of each corresponding plaintext block chunks. Hence the view of the adversary in this case can be simulated given the privacy guarantee of our HSS scheme. Our scheme satisfies the authenticated shares property, hence any tampering of the shares by the adversary would make the protocol abort. The meta data is authenticated using a universal hash function that satisfies uniform difference property.

Theorem 24. *The distributed ORAM construction described in Figure 9 and Figure 10 is a secure ORAM.*

Proof. See full version of this paper.

5.5 Complexity Analysis

First, we must determine the dependence between the parameters. Each share stores a number in $[0, M'-1)$, and since there are C share chunks per block this gives $B = C\log_2 M'$. For the HSS parameters, we choose the smallest possible ciphertext size ($s = 2$) as this will decrease the communication bandwidth of data sent to the servers. Therefore, we should set $M\mathcal{N}^{-1} = 2^{-\lambda}$, where $\mathcal{N} = 2^{\Theta(\ell(\kappa))}$ is the Damgård–Jurik public key, to have a statistical correctness error negligible in λ. We set M' to be as large as possible (as determined by Equation 1) in order to reduce the number of chunks (which take extra computation) while keeping the same block size and ciphertext size. So we set $M' = \frac{1}{(m-1)!}e^{-2m}M$, where $m = Z(L+1)$ is the largest number of options in a select operation, and get $\log_2 M' = \Theta(\ell(\kappa) - \lambda - ZL\log(ZL)) = \Theta(\ell(\kappa) - \lambda\log N\log(\lambda))$, where we have assumed that $\lambda = \Omega(\log(N))$.

Next, we analyze the complexity of each part.

Communication complexity The communication complexity from client to the servers consists of $\Theta(ZL)$ ciphertexts sent on every eviction (once every A accesses), plus 1 sent for every access. From the server to the client, we get $B + \Theta(\ell(\kappa))$ bits sent from each server, for the shares we decode plus the extra

```
Let (Setup, Share, Eval, Run, Decode) be a 2 party HSS scheme as in Definition 13
$\mathcal{H}$ is a universal family that satisfies uniform difference property

Protocol parameters: $B$, $\lambda$, $\kappa$

Setup($D$):

  $(T, \mathsf{md}) \leftarrow$ Bounded-Feedback-ORAM-Setup on input $D$
  $h \leftarrow_{\$} \mathcal{H}$
  $\mathsf{hash} \leftarrow h(\mathsf{md})$
  Picks random shares $\mathsf{md}_0$ and $\mathsf{hash}_0$
  $\mathsf{md}_1 \leftarrow md + \mathsf{md}_0$ and $\mathsf{hash}_1 \leftarrow \mathsf{hash} + \mathsf{hash}_0$
  $G, cnt \leftarrow 0$
  $(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1) \leftarrow \mathsf{Setup}(1^\kappa)$
  For each block $b \in T$, for each chunk index $c$:
    $(b_0[c], b_1[c]) \leftarrow \mathsf{Share}(pk, sk_0, sk_1, \mathsf{REG}, b[c])$
  For $i = 0, 1$, and for each block $b$ in $T$:
    Set corresponding block in $T_i$ as $b_i$
  $st_c = (G, cnt, \mathsf{PosMap}, \mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1)$
  For $i = 0, 1$, $st_{s_i} = (T_i, \mathsf{md}_i, \mathsf{hash}_i, \mathsf{sk}_i)$
```

Fig. 9: The 2-server distributed ORAM Setup function. In this protocol we assume the ORAM Setup protocol for the Bounded Feedback ORAM given in [DvDF+16]. Client side computation is colored red and server side computation is colored blue.

$\Theta(\ell(\kappa))$ coming from the fact that the shares were already multiplied by the private key before they were sent back. This comes to a total of $2B + \Theta(\ell(\kappa)\log N)$ amortized communication for each access.

Client Computation The client computation is dominated by the Share function calls in the Select operations in the protocol. This is dominated by eviction, where it invokes $Z(L+1)$ instances of Share, each taking time $\widetilde{O}(\ell^2(\kappa))$ because they are dominated by exponentiation. This takes a total of $\widetilde{O}(\log N \ell^2(\kappa))$ amortized time per access.

Server Computation For the server the most computationally intensive step is the computation in the Select operations. We require evaluation of a $O(m)$ gate RM circuit for a m-way select. This is dominated by the Evict step, which requires $CZ(L+1)$ evaluations of a $Z+1$-way selection. The cost of evaluating a gate is dominated by exponentiation, so we get an amortized cost of $\widetilde{O}(C\lambda \log N \ell^2(\kappa))$.

We use a similarly parameter regime to Onion ORAM, where we set the statistical security parameter $\lambda = \omega(\log N)$ and computational security parameter $\kappa = \omega(\log N)$, and based on the best known attacks on Damgård–Jurik encryption (from factoring), set $\ell(\kappa) = \Theta(\kappa^3)$. The communication complexity is then $2B + O(\log^4 N)$, so we set the minimum block size to be $B = \omega(O(\log^4 N))$ to get constant communication overhead. Then the number of chunks is determined to be $C = \frac{B}{\log_2 M'} = \Theta(\frac{\log^4 N}{\log^3 N}) = \Theta(\log N)$. Finally, we find the client side computation $\widetilde{O}(\log^7 N) = \widetilde{O}(B \log^4 N)$, and the server-side computation $\widetilde{O}(\log^3 N \ell^2(\kappa)) = \widetilde{O}(\log^9 N) = \widetilde{O}(B \log^5 N)$.

```
$\underline{\mathsf{Access}(in = (op, addr, data), st_c, st_{s0}, st_{s1}))\text{:}}$

  $l' \leftarrow_\$ [0, 2^L - 1]$
  $l \leftarrow \mathsf{PosMap}[addr]$
  $\mathsf{PosMap}[addr] \leftarrow l'$
  Compute arrays $\mathsf{md}_i[l], \mathsf{hash}_i[l]$
  For $j = 0$ to $Z(L+1)$:
    if $H(md_1[l,j] - md_0[l,j]) \neq \mathsf{hash}_1[l,j] - \mathsf{hash}_0[l,j]$ then abort
  $md \leftarrow md_1[l] - md_0[1]$ // Element wise subtraction
  Find $i \ni md[i,0] = addr$
  $data \leftarrow \mathsf{Select}\ (i, \mathcal{P}_0(l), \mathcal{P}_1(l))$
  if $data = \perp$ then abort
  if $op = write$ then $data = data'$ else output $data$
  Set $\mathsf{md}[l,j] \leftarrow (addr, l')$ for the least index $j \ni \mathsf{md}[l,j] \neq \perp$
  $\mathsf{md}[l,i] \leftarrow \perp$
  For each chunk index $c$:
    $(b_0[c], b_1[c]) \leftarrow \mathsf{Share}(pk, sk_0, sk_1, \mathsf{REG}, data[c])$
  Sample new random $\mathsf{md}_0$ and $\mathsf{hash}_0$
  $\mathsf{md}_1 \leftarrow \mathsf{md} + \mathsf{md}_0$ and $\mathsf{hash}_1 \leftarrow H(\mathsf{md}) + \mathsf{hash}_0$
  $\mathsf{md}[l] \leftarrow \mathsf{md}_i$ and $h(\mathsf{md}[l]) \leftarrow \mathsf{hash}_i$
  Set $(cnt+1)^{th}$ block in bucket $\mathcal{P}_i(l,1)$ as $b_i$
  // Eviction
  $cnt \leftarrow cnt + 1 \mod A$
  if $cnt \overset{?}{=} 0$:
    $l_e \leftarrow$ reverse bit string of $G$ // Picking paths in reverse lexicographic order
    $G \leftarrow (G + 1 \mod 2^L)$
    For $k \leftarrow 0$ to $L - 1$:
      For each child bucket $C$ of $\mathcal{P}(l_e)$:
        $b \in 0^{2Z}$
        For $i \in [0, Z-1]$ : Set $b[i] \leftarrow 1$ if $i^{th}$ block in $\mathcal{P}(l_e)$ can be moved into $C$
        For $i \in [0, Z-1]$ : Set $b[Z+i] \leftarrow 1$ if $i^{th}$ block in $C$ is real
        $\mathsf{Evict}\ (b, (\mathcal{P}_0(l_e)||C_0), (\mathcal{P}_1(l_e)||C_1))$
```

Fig. 10: The 2-server distributed ORAM Access function. Client side computation is colored red and server side computation is colored blue.

6 Trapdoor Hash Functions

The idea of using a distance function to compute a distributed discrete logarithm has been applied to more than just HSS. One such application is to trapdoor hash functions, which have applications to rate-1 OT, PIR, and private matrix-vector products, among others [DGI+19]. In this section we present a new trapdoor hash function based on DCR and our distance function, and show that it has negligible error probability. We then talk about possible generalizations allowed by our construction.

We present our trapdoor hash in Figure 11. See also the full version of this paper, where we review the definition of a trapdoor hash function, with some notational changes. We support linear predicates, $\mathcal{F}_n := \{f(x) = \bigoplus_i f_i x_i \mid f_i \in \{0,1\}\}$. [DGI+19] also gave a DCR-based construction that was in many ways similar, but since they used the distance function of [BGI16] they had an inverse polynomial error rate. We instead achieve a negligible error probability.

Theorem 25. *The construction in Figure 11 is a $(1 - nN^{-1})$-correct trapdoor hash function with rate 1.*

Proof. See the full version of the paper.

```
Setup(1^κ, 1^n):
    $(N, \varphi) \leftarrow \mathsf{DJ.KeyGen}(1^\kappa)$
    $(g_0, g_1, \ldots, g_n) \leftarrow (\mathbb{Z}/N^2\mathbb{Z})^\times$
    return $N, g_0, \ldots, g_n$

Hash((N, g_0, ..., g_n), x, ρ):
    $r \leftarrow [0, N)$ from random bits $\rho$
    return $g_0^r \prod_i g_i^{x_i}$

Decode((N, g_0, ..., g_n), k, h):
    $e_0 := \mathrm{Dist}_{N,1}(h^k) \bmod N \bmod 2$
    return $e_0, \overline{e_0}$

KeyGen((N, g_0, ..., g_n), f):
    write $f(x) = \bigoplus_i f_i x_i$
    $k \leftarrow [0, N)$
    $K_0 := g_0^k$
    $K_i := g_i^k \exp(f_i), \forall i \in [1, n]$
    return $K, k$

Eval((N, g_0, ..., g_n), K, x, ρ):
    $r \leftarrow [0, N)$ from random bits $\rho$
    $d := \mathrm{Dist}_{N,1}\big(K_0^r \prod_i K_i^{x_i}\big)$
    return $d \bmod N \bmod 2$
```

Fig. 11: Trapdoor hash function for linear predicates from DCR based on our distance function, which achieves a negligible error rate.

6.1 Generalizations

Trapdoor hash functions are only defined to output a single bit, but our construction is really suited to producing a longer output. A possible generalization would be to allow output in any abelian group $\mathbb{G}$, so the correctness property would be that if $e \leftarrow \mathsf{Eval}(\mathsf{crs}, \mathsf{pk}, x; \rho)$ and $e_0 \leftarrow \mathsf{Decode}(\mathsf{crs}, \mathsf{sk}, h)$ then $e - e_0 = f(x)$. Then we could achieve $\mathbb{G} = \mathbb{Z}$ (as long as we have a bound on $|f(x)|$) by simply removing the last mod 2 step from Eval and Decode. And $\mathbb{G} = \mathbb{Z}/N^s\mathbb{Z}$ would work with perfect correctness if the mod N were removed as well.

This is useful for constructing rate-1 string OT efficiently. [DGI+19] build 1-out-of-k OT in batches of n elements, then having the receiver send n TDH public keys selecting the n bits they are interested in. The same hash h is shared among these n evaluations of the TDH, so if $n \gg |h|$ (the bit length of h) then the scheme is rate 1. However, this requires sending many public keys. Generalizing TDH to output large chunks of data would only need batches of $n \gg 1$ to achieve rate 1, as it would provide nearly $|h|$ bits of output per evaluation.

References

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, Heidelberg, August 2016.

BGI+17. Elette Boyle, Niv Gilboa, Yuval Ishai, Huijia Lin, and Stefano Tessaro. Foundations of homomorphic secret sharing. Cryptology ePrint Archive, Report 2017/1248, 2017. `https://eprint.iacr.org/2017/1248`.

BKS19. Elette Boyle, Lisa Kohl, and Peter Scholl. Homomorphic secret sharing from lattices without FHE. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part II*, volume 11477 of *LNCS*, pages 3–33. Springer, Heidelberg, May 2019.

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.

Cle90. Richard Cleve. Towards optimal simulations of formulas by bounded-width programs. In *22nd ACM STOC*, pages 271–277. ACM Press, May 1990.

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, Heidelberg, August 2019.

DJ01. Ivan Damgård and Mads Jurik. A generalisation, a simplification and some applications of paillier's probabilistic public-key system. In *Public Key Cryptography, 4th International Workshop on Practice and Theory in Public Key Cryptography, PKC 2001, Cheju Island, Korea, February 13-15, 2001, Proceedings*, volume 1992 of *Lecture Notes in Computer Science*, pages 119–136. Springer, 2001.

DKK18. Itai Dinur, Nathan Keller, and Ohad Klein. An optimal distributed discrete log protocol with applications to homomorphic secret sharing. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 213–242. Springer, Heidelberg, August 2018.

DS17. Jack Doerner and Abhi Shelat. Scaling oram for secure computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 523–535, 2017.

DvDF$^+$16. Srinivas Devadas, Marten van Dijk, Christopher W Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. Onion oram: A constant bandwidth blowup oblivious ram. In *Theory of Cryptography Conference*, pages 145–174. Springer, 2016.

FGJS17. Nelly Fazio, Rosario Gennaro, Tahereh Jafarikhah, and William E. Skeith III. Homomorphic secret sharing from paillier encryption. In Tatsuaki Okamoto, Yong Yu, Man Ho Au, and Yannan Li, editors, *ProvSec 2017*, volume 10592 of *LNCS*, pages 381–399. Springer, Heidelberg, October 2017.

FNR$^+$15. Christopher W Fletcher, Muhammad Naveed, Ling Ren, Elaine Shi, and Emil Stefanov. Bucket oram: Single online roundtrip, constant bandwidth oblivious ram. *IACR Cryptol. ePrint Arch.*, 2015:1065, 2015.

Gol87. Oded Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 182–194, 1987.

HOY$^+$17. Thang Hoang, Ceyhun D Ozkaptan, Attila A Yavuz, Jorge Guajardo, and Tam Nguyen. S3oram: A computation-efficient and constant client bandwidth blowup oram with shamir secret sharing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 491–505, 2017.

Ost92. Rafail Ostrovsky. *Software protection and simulation on oblivious RAMs*. PhD thesis, Massachusetts Institute of Technology, 1992.

OSY21. Claudio Orlandi, Peter Scholl, and Sophia Yakoubov. The rise of paillier: Homomorphic secret sharing and public-key silent ot. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 678–708. Springer, 2021.

Pai99. Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 223–238. Springer, Heidelberg, May 1999.