Large Message Homomorphic Secret Sharing from DCR and Applications

Lawrence Roy* Jaspal Singh*

March 3, 2021

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 multi-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) have this same limitation, other than the trivial scheme based on FHE where both parties use identical FHE ciphertexts as their shares.

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

*Oregon State University, {royl,singjasp}@oregonstate.edu

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) of the outcome. Errors also get harder to deal with the longer the program, because each operation has an independent chance of producing an error that will make the whole computation fail. Consequently, 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}), when evaluating an n-step program on plaintexts bounded by M. 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].

Concurrent result. After we had prepared our manuscript and submitted it for publication, we became aware of concurrent and independent work that also resolves this open problem. See more details in Section 1.4.

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 with 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. Our construction is based on the algebraic properties of the ciphertext group $(\mathbb{Z}/N^{s+1}\mathbb{Z})^\times$, while existing constructions are based on the generic technique of searching for a randomly chosen subset of the ciphertext space. 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 either require server-server super constant communication, or they require the minimum block size to be $\Omega(\log^6 N)$, 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 computation.

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 presented 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 show how to 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 also extend the definition for HSS, to allow (generalized, to represent RMS operations) circuits to be evaluated one gate at a time. One benefit of this approach are that this allows secure evaluation of online algorithms, which may take input and produce output many times while maintaining some secret state. This is directly useful for our application to ORAM. It also let us have subsequent gates for evaluation be chosen adaptively based on past outputs or shares, so if the adversary gets to see one set of shares and then influence the rest of function being evaluated, our scheme will still be correct.

1.2 Technical Overview

Introduction to HSS. HSS schemes take 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 additive encryption scheme supports multiplication by constants, so we can take $[\![x]\!]^c \equiv [\![cx]\!]$.

We have two different additively homomorphic schemes; what happens if we let them interact? If we take $[\![x]\!]^{\langle y \rangle}$ then 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 so $\langle\!\langle [\![x]\!]^y \rangle\!\rangle_1 / \langle\!\langle [\![x]\!]^y \rangle\!\rangle_0 = [\![x]\!]^{\langle y \rangle_1 - \langle y \rangle_0} \equiv [\![xy]\!]$. The interesting thing

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 anyway 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, $\phi^{-1}([\![z]\!]^{\varphi}) = z$, 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 $[\![x]\!]^{\langle \varphi y \rangle} \equiv \langle\!\langle \phi(xy) \rangle\!\rangle$. In ElGamal we have $[\![z]\!] = (A, B)$, and decryption works by finding $\phi^{-1}(A^{-k}B)$, where $\phi(z) = g^z$ for a public generator g, which is also 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 $\vec{k} = \begin{bmatrix} -k & 1 \end{bmatrix}$, but taking the secret shares to be vectors $\langle \vec{k} y \rangle$ then we have $[\![x]\!]^{\langle \vec{k} y \rangle} \equiv \langle\!\langle \phi(xy) \rangle\!\rangle$.

The last step of public key decryption for these 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 for constructing Dist is that both parties agree on a common set of "special points", which [BGI16] choose randomly, then iteratively compute $a\phi(-1)^j$, starting at $j = 0$ and continuing until $c = a\phi(-1)^j$ is special, then setting $\mathrm{Dist}(a)$ 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 the 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 to another multiplication operation, and so on.

Paillier distance function. We now switch to using 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, we encrypt them as $r^{N^3}(1 + N^2 z) \bmod N^4$. This is to allow the plaintext size to be bigger than the private key, which simplifies our construction. We have $\left(r^{N^3}(1 + N^2 z)\right)^{\varphi} = 1 + N^2\varphi z = \phi(z)$. Computing $\phi^{-1}(a)$ is then as easy as finding $(a-1)/N^2$, as a must be a multiple of N^2, then multiplying by $\varphi^{-1} \bmod N^2$.

It turns out that we can design a distance function that based on this ϕ^{-1}. The prior construction of HSS from Paillier encryption, [FGJS17], noticed that $\langle\!\langle \phi(z) \rangle\!\rangle_1 = \langle\!\langle \phi(z) \rangle\!\rangle_0 \mod N^2$ and used this for a minor optimization. The reason is that $\phi(z) = 1 + N^2\varphi z = 1 \mod N^2$, so $\langle\!\langle \phi(z) \rangle\!\rangle_1 / \langle\!\langle \phi(z) \rangle\!\rangle_0 = \phi(z) = 1 \mod N^2$. This means that the both parties will have something in common, and they can use it as their common ciphertext $c = \langle\!\langle \phi(z) \rangle\!\rangle_0 \bmod N^2 = \langle\!\langle \phi(z) \rangle\!\rangle_1 \bmod N^2$.

On input a, let the distance function pick a canonical representative c of $a + N^2\mathbb{Z}$, satisfying $c \in [-\frac{N-1}{2}, \frac{N-1}{2}]$. Then $a/c = 1 + N^2 w$, and we let $\mathrm{Dist}(a) = w$. This means that our "special points" are $[-\frac{N-1}{2}, \frac{N-1}{2}]$, instead of using a random set like [BGI16]. We then have $\mathrm{Dist}(a\phi(z)) - \mathrm{Dist}(a) = \varphi z$, because $a\phi(z)/c = (1 + N^2\varphi z)(1 + N^2 w) = 1 + N^2(\varphi z + w)$. 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 messages.

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 to achieve constant bandwidth overhead, the scheme allows for blocks of size $\tilde{\omega}(\log^6 N)$, with $\tilde{O}(B \log^4 N)$ client computation and with $\tilde{\omega}(B \log^4 N)$ server computation. Comparatively, 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 on the communication and the 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 saving 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 $\tilde{\Omega}(\log^6 N)$. Its 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.

1.4 Concurrent Result

A concurrent and independent work was posted on ePrint on March 3, 2021 that also resolves the open problem of creating an HSS scheme with negligible correctness error without using LWE, after we had prepared a version of this manuscript and submitted it for publication. In [OSY21], Orlandi, Scholl, and Yakoubov (OSY) construct HSS with negligible correctness error and exponentially large plaintext space, based on the Paillier encryption scheme. Qualitatively, their distance function matches ours (Section 4.1), which is the main construction we base our results on. We briefly compare and contrast their results with ours:

- Both our construction and theirs have significant similarities; specifically, when the ciphertexts are modulo N^2 as in Paillier we have identical distance functions.
- OSY use Paillier encryption, while we use the Damgård–Jurik generalization, allowing the ciphertext space to be significantly larger than the private key. At the expense of our smallest

possible ciphertexts being in $(\mathbb{Z}/N^3\mathbb{Z})^\times$ rather than $(\mathbb{Z}/N^2\mathbb{Z})^\times$, we have no need for circular security assumptions related to Paillier encryption, nor for the overhead from the BG provably circular secure encryption scheme [BG10]. Consequently, we do not need multiple ciphertexts to encrypt a single input to the HSS scheme, while OSY needs around 6, assuming circular security, or $\Theta(\ell(\kappa))$ with the BG encryption scheme, where $\ell(\kappa)$ is the bit length of the primes used to generate the public key. Splitting the ciphertexts into chunks is costly in terms of both communication and computation, as each ciphertext chunk must be exponentiated separately during an HSS multiplication. We therefore achieve either a constant factor or a factor of $\Theta(\ell(\kappa))$ improvement in both computation and communication relative to OSY's HSS schemes, depending on the assumption.

- Additive decoding (Definition 18) is naturally satisfied by their scheme, while we need to add an additional complication to achieve it (see Appendix A.2), which more than doubles our ciphertext size. Overall our communication is still cheaper, as we need a single ciphertext in $(\mathbb{Z}/N^7\mathbb{Z})^\times$, instead of 6 ciphertexts in $(\mathbb{Z}/N^2\mathbb{Z})^\times$ for OSY's scheme based on circularly secure Paillier.

- We prove malicious security of our HSS by showing that its shares are authenticated (Theorem 23).

- We give novel HSS definitions (Section 3) and proofs (Section 4) that support online algorithms and adaptively chosen circuits.

- We explore completely different applications: we construct 2-server oblivious RAM (ORAM) with constant overhead and rate-1 trapdoor hash functions (TDH), while OSY instead build pseudorandom correlation generators/functions (PCG/PCF). A key feature of their PCGs and PCFs is "public-key setup", which allows two parties who know each other's public keys to non-interactively compute the setup for HSS. They build this using Paillier–ElGamal encryption; we use a similar technique to construct our TDH, though we avoid the need for picking N to be the product of safe primes, making key generation faster.

2 Preliminaries

2.1 Notation

Modular arithmetic. Let $\mathbb{Z}/N\mathbb{Z}$ be the ring of integers modulo N ($\mathbb{Z}/0\mathbb{Z} = \mathbb{Z}$), and $(\mathbb{Z}/N\mathbb{Z})^\times$ be its group of units. There are a few useful maps between $\mathbb{Z}/N\mathbb{Z}$ and $\mathbb{Z}/M\mathbb{Z}$ if $N \mid M$. Multiplication by $\frac{M}{N}$ gives a injective homomorphism from $\mathbb{Z}/N\mathbb{Z}$ to $\mathbb{Z}/M\mathbb{Z}$, and we let division by the same be the inverse map, where it exists. We will notate the quotient map from $\mathbb{Z}/M\mathbb{Z}$ 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 $a \equiv_N b$. We also need to pick a representative in $\mathbb{Z}/M\mathbb{Z}$ for each element of $\mathbb{Z}/N\mathbb{Z}$. We do this with a symmetric modulus operation $\cdot \bmod N \colon \mathbb{Z}/N\mathbb{Z} \to \mathbb{Z}/M\mathbb{Z}$, where $(x + N\mathbb{Z}) \bmod N = x + M\mathbb{Z}$ when $x \in [-\frac{N}{2}, \frac{N}{2})$.

We will often be working with $\mathbb{Z}/N^u\mathbb{Z}$ for a positive integer u and odd N, which can be thought of as a sort of finite power series in N.[1] By abuse of notation, we define extend the functions exp

[1]Specifically, it is a finite approximation to the N-adic numbers. Note that if N is not prime then the N-adics have zero-divisors, so N is usually assumed to be prime, but we do not need to work in an integral domain.

and log to $\mathbb{Z}/N^u\mathbb{Z}$ via their Taylor series:

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

These functions are only well defined in special cases, since there is not any meaningful definition of the convergence of an infinite sum in $\mathbb{Z}/N^u\mathbb{Z}$ other than only finitely many terms being nonzero modulo N^u. For odd N, the series for $\exp(Nx)$ and $\log(1+Nx)$ converge because the power of any prime factor $p > 2$ of N in the numerator grows faster than in the denominator. But in general these series do not converge. The usual properties of these functions follow from their series definition and remain true in $\mathbb{Z}/N^u\mathbb{Z}$ when they converge. See a p-adic numbers textbook for details, such as [Gou20, Sec. 5.7].

Algorithm notation. We will write 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 x is $\overline{x} = 1 - x$. We give equality testing its own symbol, $\stackrel{?}{=}$, so $x \stackrel{?}{=} y$ 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 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 $x \leftarrow \mathcal{A}(1^\kappa)$ in the middle of the definition, so to preserve the adversary's state to give to the distinguisher we instead use $(\text{view}, x) \leftarrow \mathcal{A}$, then 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]. It makes use of the fact that $\mathbb{Z}/N^{s+1}\mathbb{Z}$ allows an efficient discrete logarithm for any number of the form $1 + Nx$.

Definition 1. *Given a security parameter κ and a message size s, define Damgård–Jurik encryption to be the following operations.*[2]

$(N, \varphi) \leftarrow \mathsf{DJ.KeyGen}(1^\kappa)$ *Generate an RSA modulus $N = pq$ where $2^{\ell(\kappa)-1} < p, q < 2^{\ell(\kappa)}$. ℓ is a polynomial chosen to make the scheme achieve the κ-bit security level. N will be the public key and $\varphi = \varphi(N)$ will be the public key, 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 $c = r^{N^s} \exp(Nx)$.*

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

[2] Damgård–Jurik was originally defined using $(1+N)^x$ instead of $\exp(Nx)$. We chose the latter because its inverse can be represented simply and efficiently with the power series for log. In particular, evaluating log with Horner's rule uses $O(s)$ arithmetic operations on $O(s \log_2 N)$-bit numbers, while the original algorithm used Hensel lifting and took $O(s^2)$ operations.

Encryption is clearly additively homomorphic, since $\exp(Nx)\exp(Ny) = \exp(N(x+y))$. The order of $(\mathbb{Z}/N^{s+1}\mathbb{Z})^\times$ is $\varphi(N^{s+1}) = p^s(p-1)q^s(q-1) = \varphi N^s$, since $(\mathbb{Z}/N^{s+1}\mathbb{Z})^\times \simeq (\mathbb{Z}/p^{s+1}\mathbb{Z})^\times \times (\mathbb{Z}/q^{s+1}\mathbb{Z})^\times$ by the Chinese remainder theorem and $|(\mathbb{Z}/p^{s+1}\mathbb{Z})^\times| = p^s(p-1)$. Therefore $\log(c^\varphi) = \log(r^{\varphi N^s}\exp(Nx)^\varphi) = \log(\exp(N\varphi x)) = N\varphi x$. Notice that since p and q have the same bit length, $p-1$ and $q-1$ are each coprime to N, and so φ has a multiplicative inverse in $\mathbb{Z}/N^s\mathbb{Z}$. Therefore decryption is correct.

We also want to show that every ciphertext is valid, i.e. that $\mathsf{Dec}_{N,s,\varphi}$ is always well defined. For this, we need the Taylor series for log to converge in the expression $\log(c^\varphi)$. By Euler's theorem $c^\varphi \equiv_N 1$, so $c^\varphi = 1 + Nu$ for some $u \in \mathbb{Z}/N^s\mathbb{Z}$ and decryption is always well defined. Next we show that c could have come from $\mathsf{Enc}_{N,s}$. Let $x = \mathsf{Dec}_{N,s,\varphi}(c)$ and $v = c\exp(-Nx)$, so that $v^\varphi = c^\varphi\exp(-\log(c^\varphi)) = 1$. Because φ and N are coprime, we can compute $r = v^{N^{-s} \bmod \varphi}$ and see that $r^{N^s} = v$. This shows that every element $c \in (\mathbb{Z}/N^{s+1}\mathbb{Z})^\times$ can be written as $r^{N^s}\exp(Nx)$ for some $x \in \mathbb{Z}/N^s\mathbb{Z}$ and $r \in (\mathbb{Z}/N^s\mathbb{Z})^\times$. This is a surjective group homomorphism $(\mathbb{Z}/N^{s+1}\mathbb{Z})^\times \times \mathbb{Z}/N^s\mathbb{Z} \to (\mathbb{Z}/N^{s+1}\mathbb{Z})^\times$, so every c has exactly the same number of preimages r, x. Therefore the encryption of a uniformly random plaintext is a uniformly random ciphertext.

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

Definition 2. *The* decisional composition 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.*

Because encrypting a uniformly random plaintexts gives a uniformly random ciphertext, CPA$ security is equivalent to CPA security for Damgård–Jurik.

Uniform Difference Property 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 $h \in H$.

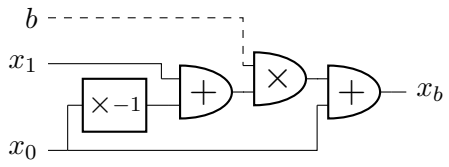
3 Circuit Homomorphic Secret Sharing

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 program. Then we define a notion of HSS based on replacing the gates in a circuit with operations on the shares, which only needs to specify properties of a single gate at a time. These properties compose to allow 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, while maintaining state. This 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.

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

Figure 1: The selection function x_b represented as an RMS program (left, Definition 4) and a RM circuit (right, Definition 9). 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. Notice that x_0 and x_1 don't have input wire type in the RM circuit, allowing us to compose the RM circuit with some other circuit that would produce x_0 and x_1.

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 would possibly incur a linear blowup.

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*

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

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 captions the allowed operations in RMS programs. We have drawn an example in Figure 1b. In Definition 4 there are two types of values: inputs and registers. In the corresponding circuit there are two types of wire. Inputs are drawn with a dashed line, while registers are drawn with a solid line. Gates representing linear operations (addition and multiplication-by-constant) are allowed for either type of wire, and both allow sources for the value 1. However, multiplication is only allowed between a dashed input type wire and a solid register type wire, and must always produce a solid wire. A dashed input type wire may be converted to a solid register type wire, and for convenience this will be shown implicitly, rather than showing the equivalent circuit: a register type wire value 1 source that gets multiplied by the input type wire.

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

Definition 5. *A* circuit prototype $(\mathsf{types}, \mathsf{gates}, \mathsf{in}, \mathsf{out})$ *is a set* $\mathsf{types} \subseteq \{0,1\}^*$ *of wire types, a set* $\mathsf{gates} \subseteq \{0,1\}^*$ *of gate types, and 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* $(\mathsf{types}, \mathsf{gates}, \mathsf{in}, \mathsf{out})$ *is a directed acyclic graph* $(\mathsf{nodes}, \mathsf{wires})$*, a total order on* wires*, subsets* $\mathsf{inputs}, \mathsf{outputs} \subseteq \mathsf{nodes}$*, a node labeling* $\mathsf{type}\colon \mathsf{nodes} \to \mathsf{types}$*, and a non-input node labeling* $\mathsf{gate}\colon \mathsf{nodes} \setminus \mathsf{inputs} \to \mathsf{gates}$*. We require the circuit to be well-formed: for any non-input node* v*,* $\mathsf{type}(v) = \mathsf{out}\big(\mathsf{gate}(v)\big)$*, and for every* $v \in V \setminus \mathsf{inputs}$*, if* $(v_1, v), (v_2, v), \ldots, (v_n, v)$ *are its incoming edges in sorted order then* $\mathsf{type}(v_1)\,\mathsf{type}(v_2) \cdots \mathsf{type}(v_n) = \mathsf{in}\big(\mathsf{gate}(v)\big)$*.*

An edge in a circuit is called a wire, and a non-input node is called a gate.

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. We made this simplification because it is enough for our application, but there are other sorts of circuits best represented by the more general definition.

We would like to evaluate typed circuits one gate at a time, just like any other kind of circuit. To do this, we to need a semantics to define what each gate operation 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* $w \in \mathsf{types}$ *a set of values* $\mathsf{values}(w)$*, and assigns each gate type* $g \in \mathsf{gates}$ *a function* $\mathsf{eval}(g)\colon \mathsf{values}(w_1) \times \mathsf{values}(w_2) \times \cdots \times \mathsf{values}(w_n) \to \mathsf{values}\big(\mathsf{out}(g)\big)$*, 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 inputs $x_v \in \mathsf{values}(\mathsf{type}(v))$ for every $v \in \mathsf{inputs}$, the evaluation proceeds in topological order. The inputs of each gate are its incoming edges, 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 below for the formal algorithm.

Definition 8. *A typed circuit* $(\mathsf{nodes}, \mathsf{wires}, \mathsf{inputs}, \mathsf{outputs}, \mathsf{type}, \mathsf{gate})$ *can be evaluated with a semantics* $(\mathsf{values}, \mathsf{eval})$ *if they are for the same circuit prototype. The evaluation is described by the algorithm* Run *below.*

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

Restricted multiplication circuits. We can now define restricted multiplication circuits by applying the above formal definitions.

Definition 9. *The* Restricted Multiplication (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* $c \in \mathbb{K}$*, and a single nonlinear multiplication operation* $\times\colon \mathsf{IN} \times \mathsf{REG} \to \mathsf{REG}$*. An* RM circuit *is a circuit for this signature.*

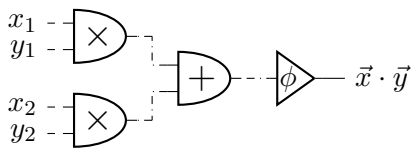


Figure 2: A bounded RM circuit for computing the dot product of a pair of two element vectors. The new wire type MUL is drawn with a - - line, and the new conversion operation ϕ with a triangle.

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

Definition 10. *The* evaluation semantics *for RM circuits over* $\mathbb{K}$ *sets* $\mathsf{values}(\mathsf{IN}) = \mathsf{values}(\mathsf{REG}) = \mathbb{K}$ *and assigns each operation to the corresponding one in the ring* $\mathbb{K}$*.*

However, this will not be 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 circuits: one for the plaintexts and one for the shares. The semantics for the shares is what defines the homomorphic operations that a HSS scheme supports.

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 on the output of every multiplication gate. However, this conversion step can be delayed until after further linear operations, so we generalize RM circuits to add another wire type to allow these operations.

Definition 11. *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 constants and linear operations for each wire type, a multiplication operation* $\times\colon \mathsf{IN} \times \mathsf{REG} \to \mathsf{MUL}$*, and a conversion operation* $\phi\colon \mathsf{MUL} \to \mathsf{REG}$*. A* bounded RM circuit *is a circuit for this signature.*

An example of this new kind of circuit is illustrated in Figure 2.

Definition 12. *The* evaluation semantics *for bounded RM circuits over* $\mathbb{K}$ *bounded in a subset* $M \subseteq \mathbb{K}$ *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 the 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: if any input is* $\bot$ *then the output is* $\bot$*.*

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

Definition 13. *A semantics* $(\mathsf{values}, \mathsf{eval})$ *is called a* failure semantics *if there is a special value* $\bot \in \mathsf{values}(w)$ *called* failure*, for all wire types* $w \in \mathsf{types}$*, which is absorbing for any function in* $\mathsf{eval}(\mathsf{gates})$*. That is, for any* $g \in \mathsf{gates}$*,* $\mathsf{eval}(g)(\ldots, \bot, \ldots) = \bot$*, 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 the types of shares and operations

of them, for each of the two servers. 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.

It turns out that the ring $\mathbb{K}$ that we can support homomorphic operations over will depend on the public key. Since we are using Damgård–Jurik encryption, the group will be $\mathbb{Z}/N^s\mathbb{Z}$ for some public key N, which 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 14. *A $(1-p)$-correct two-server Homomorphic Secret Sharing (HSS) scheme with public-key setup consists of the following 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 $w \in \mathsf{types}$ 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 $w \in \mathsf{types}$.*

such that the following conditions hold:

- *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 $\mathcal{A}$.*

$$\boxed{\begin{array}{l}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1) \leftarrow \mathsf{Setup}(1^\kappa) \\ (w, x) \leftarrow \mathcal{A}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1) \\ (s_0, s_1) \leftarrow \mathsf{Share}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1, w, x) \\ y \leftarrow \mathsf{Decode}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1, w, s_0, s_1) \\ \text{return } x \stackrel{?}{=} y \vee x \stackrel{?}{=} \bot\end{array}}$$

- *Homomorphism: The semantics 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 $\bot$.*

$(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1) \leftarrow \mathsf{Setup}(1^\kappa)$
$(\text{proto}, (\mathsf{values}, \mathsf{eval})) := \mathsf{Eval}(\mathsf{pk})$
$(\text{view}, g, \{(s_{i0}, s_{i1})\}_i) \leftarrow \mathcal{A}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1)$
$r \leftarrow \$$
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 view, r, y

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

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

Formally, the following probability must be negligibly different between $i = 0$ and $i = 1$.

$$\Pr[(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1) \leftarrow \mathsf{Setup}(1^\kappa);\ \mathcal{A}^{\mathcal{O}_{i,\mathsf{pk},\mathsf{sk}_0,\mathsf{sk}_1}}(\mathsf{pk}, \mathsf{sk}_j) = 1]$$

Note that Hom is given a access to a shared stream of randomness. This is to stop an adversary from choosing a circuit that will make the scheme deterministically fail. This could be instantiated with a shared PRG, reseeded whenever the circuit is chosen adaptively in a way that might depend on the seed, or with a random oracle evaluated on a description of the current gate and how the input shares were produced if it is necessary to somehow adaptively change the computation without using any communication at all.

We include an error probability p in our definition for comparison with existing methods of HSS evaluation, even though our scheme has negligible p. The DDH-based construction of [BGI16] satisfies our definition with $p = \frac{1}{\text{poly}(\kappa)}$. 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. However, this structure raises a question: why not do the same with Share and Hom, and require 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, and there is no way for Share to always produce shares of the right size to be indistinguishable from every kind of circuit.

Since our correctness and homomorphism definitions are in terms of just performing a single operation, we need to prove that they can be composed into correctly evaluating a whole circuit. The idea is that we can chain the homomorphism property over and over again, replacing the homomorphic operations one at a time and gradually moving the Decode operation sooner, until the Decode operation is at the start of the circuit. Then if the inputs were from Share the correctness property guarantees that Decode will cancel with Share, and the only thing left is honest evaluation.

Lemma 15. *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 $\text{view}, r, \mathsf{Run}(f, \text{sem}_{pt}, x)$

$(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1) \leftarrow \mathsf{Setup}(1^\kappa)$
$(\text{proto}, \text{sem}_{pt}) := \mathsf{Eval}(\mathsf{pk})$
$(\text{view}, f, \{(s_{0v}, s_{1v})\}_v) \leftarrow \mathcal{A}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1)$
$(\mathsf{nodes}, \mathsf{wires}, \mathsf{inputs}, \mathsf{outputs}, \mathsf{type}, \mathsf{gate}) := f$
$r \leftarrow \$$
for $j \in \{0, 1\}$:
  $s'_j := \mathsf{Run}(f, \mathsf{Hom}(j, \mathsf{pk}, \mathsf{sk}_j), s_j, r)$
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. We 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 will be evaluated using Hom, then its outputs will fed into Decode and used to evaluate h in plaintext. Initially g is the whole circuit and h is nothing, but in each hybrid we shift a gate from g into h, picking one that comes last in topological order. The difference caused the by the switch is that before the gate got evaluated homomorphically, then decoded, while afterwards its inputs get 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, for some wire types. What this means is that we setup 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 run a decode operation. The adversary wins if it manages to obtain a different result than would be obtained with two honest servers.

Definition 16. *An HSS scheme is* authenticated *for wire types $A \subseteq \mathsf{types}$ 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, no PPT $\mathcal{A}$ given oracle access to the interface of the honest*

party $\mathcal{H}$ can cause GUESS *to return* TRUE *with non-negligible probability.*

$\mathcal{H}$

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 $W[i] \in A$
- $s_{\bar{j}} := U[i]_{\bar{j}}$
- $y \leftarrow \mathsf{Decode}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1, W[i], U[i]_0, U[i]_1)$
- $z \leftarrow \mathsf{Decode}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1, W[i], s_0, s_1)$
- return $y \stackrel{?}{\neq} z \wedge y \stackrel{?}{\neq} \perp \wedge z \stackrel{?}{\neq} \perp$

Here, $\mathcal{A}$ is required to call INIT *exactly once, before calling anything else in $\mathcal{H}$.*

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

Definition 17. *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, w, and s_j is output to server j. All messages in the protocol must come from the client.*

Definition 18. *A two-server HSS scheme has* additive decoding *for wire type w if there are PPT algorithms f_0, f_1 such that*

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

with all but negligible probability whenever the left side is not $\perp$, where $\mathsf{values}(w)$ is an abelian group.

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. 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 $c = c \bmod N$. Because $c \bmod N \bmod N = c \bmod N$, the closest special value is $c \bmod N$. The distance can then be computed efficiently using log.

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

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

Theorem 19. *For any $c \in (\mathbb{Z}/N^{s+1}\mathbb{Z})^\times$ and $x \in \mathbb{Z}/N^s\mathbb{Z}$,*

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

Proof. First, we need to show that $\mathrm{Dist}_{N,s}(c)$ is always well defined. We have $\frac{c}{c \bmod N} \equiv_N \frac{c}{c} \equiv_N 1$, which implies that the Taylor series for log converges in this function. Then,

$$\begin{aligned}&\mathrm{Dist}_{N,s}(c\exp(Nx)) - \mathrm{Dist}_{N,s}(c) \\ &= \frac{1}{N}\left(\log\left(\frac{c\exp(Nx)}{c\exp(Nx) \bmod N}\right) - \log\Big(\frac{c}{c \bmod N}\Big)\right) \\ &= \frac{1}{N}\left(\log\left(\frac{c\exp(Nx)}{c \bmod N}\right) - \log\Big(\frac{c}{c \bmod N}\Big)\right) \\ &= \frac{1}{N}\Big(\log\Big(\frac{c}{c \bmod N}\Big) + Nx - \log\Big(\frac{c}{c \bmod N}\Big)\Big) \\ &= x.\end{aligned}$$

□

Corollary 20. *The distribution $\mathrm{Dist}_{N,s}(\mathsf{DJ.Enc}_{N,s}(x))$ for uniformly random $x \in \mathbb{Z}/N^{s+1}\mathbb{Z}$ is identical to the uniform distribution on $\mathbb{Z}/N^s\mathbb{Z}$.*

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

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

Proof. The condition may equivalently be written as

$$r \bmod N + x = \big(r \bmod N + x\big) \bmod N.$$

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. □

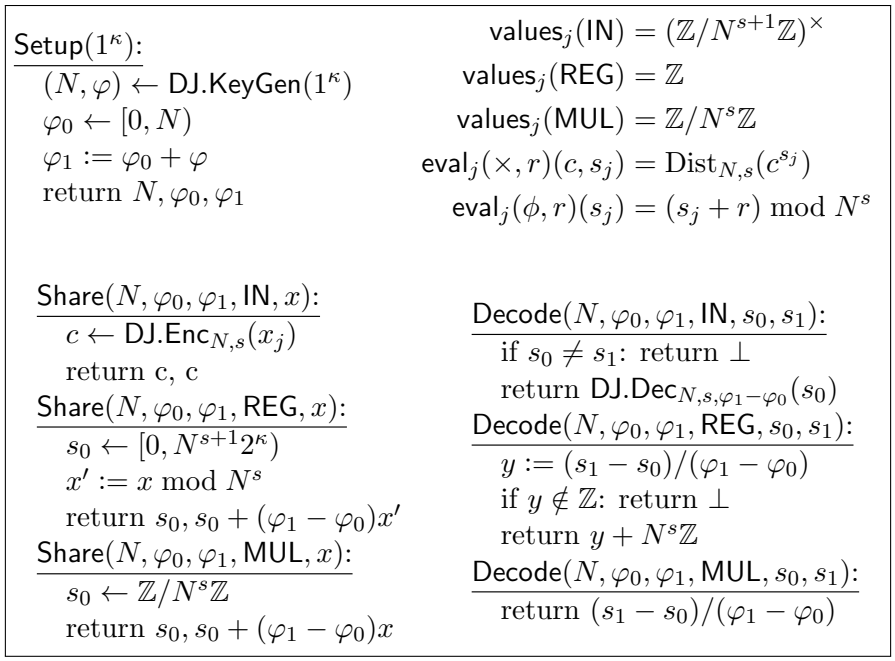


Figure 3: Our HSS scheme for bounded RM circuits. In the top left the encryption is setup and the secret key shared between the two parties. The secret share sets are in the top right, along with the non-trivial homomorphic that may be performed on them. The linear operations are just the abelian group structure of the shares are in, and we omit them. Share and Decode for the three types of shares are shown in the bottom right.

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 among the two parties in Setup (Figure 3). The plaintext evaluation semantics Eval(N) are then the evaluation semantics (Definition 12) for bounded RM circuits over $\mathbb{Z}/N^s\mathbb{Z}$ bounded in $[-M, M] + N^s\mathbb{Z}$.

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 3). 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 shares of 1.

The homomorphic multiplication function eval$_j(\times, r)$ in Figure 3 is based on $c^{\varphi x}$ essential 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 eval$_j(\phi, r)$ to pick a representative in $\mathbb{Z}$, allowing the result to be converted to shares in $\mathbb{Z}$.

We now prove the HSS scheme's correctness and privacy. Note that its error rate is a negligible function of κ and s.

Theorem 22. *Figure 3 describes a $(1 - MN^{1-s})$-correct HSS scheme (Definition 14) 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 $\varphi_1 - \varphi_0$ in Decode gives the correct decoding.

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

$$\begin{aligned}\frac{c^{s_1}}{c^{s_0}} &= r^{N^s(s_1-s_0)} \exp(Nx(s_1 - s_0)) \\ &= r^{N^s\varphi y} \exp(Nx\varphi y) \\ &= \exp(N\varphi xy),\end{aligned}$$

for some $r \in (\mathbb{Z}/N^{s+1}\mathbb{Z})^\times$, where $x = \mathsf{DJ.Dec}_{N,s,\varphi}(c)$ and $y = \frac{s_1-s_0}{\varphi}$ are the decodings of the two input shares. Then Theorem 19 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 21. 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 $\mathcal{O}_{0,\mathsf{pk},\mathsf{sk}_0,\mathsf{sk}_1}$.

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 $\frac{p+q-1}{N} \le \frac{2^{\ell(\kappa)+1}}{2^{2(\ell(\kappa)-1)}} = 2^{-\ell(\kappa)+3}$, which is negligible.
3. Notice that the private key φ is now totally unused. We can then apply Theorem 3 to show that Share for wire type IN can return a dummy encryption of 0 and be indistinguishable.

We have reached the halfway point, where the adversary's view does not depend on their inputs to Share. Going through almost the same hybrids in reverse then takes us to the distribution where $\mathcal{A}$ is given oracle access to $\mathcal{O}_{1,\mathsf{pk},\mathsf{sk}_0,\mathsf{sk}_1}$. □

Shares of type IN are trivially authenticated, as both parties always have the same share. REG authentication comes from the shares always being of a multiple of φ, so to create a fake share the adversary would have to guess a multiple of φ to offset their share by.

Theorem 23. *The HSS scheme in Figure 3 is authenticated for wire types* $\{\mathsf{IN}, \mathsf{REG}\}$.

Proof. We defer this proof to the appendix. See Appendix A.1. □

Public-key sharing. Our construction also satisfies public-key sharing (Definition 17). 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 parties, and then they do some local computation to find the shares.

Additive decoding. A variant of our scheme also satisfies additive decoding. The idea is to send an encryption of φ^{-1} under a second encryption key, and use it to remove the division step from Decode (on wire type REG) so that it only does subtraction. See Appendix A.2 for details.

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 data blocks on the server while hiding the access pattern [Ost92, Gol87]. While traditionally ORAM protocols were designed assuming single server which stores data passively, recent works have considered more general settings, allowing for multiple non-colluding 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. Allowing for computation on the server side one can break this bandwidth lower bound, and even 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, which is updated after each ORAM operation. We use the following syntax to represent each ORAM operation:

$$(out, st'_c, , st'_{s1}, st'_{s2}) \leftarrow f(in, st_c, st_{s1}, st_{s2}) \tag{1}$$

When performing the f ORAM function, the client receives as input in and its output is out. After this function execution, the states of the client and the two servers are updated from st_c, st_{s0}, st_{s1} to $st'_c, st'_{s0}, st'_{s1}$ respectively.

Definition 24. *A distributed 2 server ORAM construction with security parameter λ consists of the following two interactive protocols:*

- $(\perp, st'_c, , st'_{s1}, st'_{s2}) \leftarrow \mathsf{Setup}(D, \perp, \perp, \perp)$: 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.
- $(data, st'_c, , st'_{s1}, st'_{s2}) \leftarrow \mathsf{Access}(op, st_c, st_{s1}, st_{s2})$: The client receives as input an ORAM operation $(op, idx, data)$, 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 Appendix B.1.).

5.2 An Overview of Onion ORAM

Our protocol is based on 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 the Bounded Feedback ORAM and how it can be modified to give the original single server Onion ORAM construction.

Bounded Feedback ORAM Similar to other tree-based ORAMs, its single 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 meta-data $(addr, label)$, 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 remaped 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.

Recursion The position map stored on the client is super-linear in the size of the database. To avoid the large client memory, we can recursively stores 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 allows for server-side computation, and at a high level it differs from Bounded Feedback ORAM in the following ways:

1. The server side storage is encrypted using an aditively homomorphic encryption (AHE) scheme, where blocks at depth l in the tree at encrypted using l layers of the AHE scheme.

2. Downloading only meta-data: Rather than downloading all blocks on a path during the access operations, it downloads only the meta-data of all the blocks on this path. It locally computes the location of the block that it wants to read/write. The client and the server run a *homomorphic select operation* to just fetch the needed data-block from this path. We describe this selection operation below in greater detail. Similarly the selection operation is also used in the eviction algorithm.

3. To ensure integrity of data, the protocol uses memory checking to ensure integrity of the read only meta-data blocks. To ensure the integrity of the data blocks the protocol uses a new verification algorithm that relies on probabilistic checking and error correcting codes.

The key building block for their protocol is the homomorphic selection operation, where the client has an index idx and the server has ciphertexts $ct_1, \ldots ct_m$ (under l layers of AHE $\mathcal{E}_l$). The client sends an $l+1$ layer encrypted m length bit vector b to the server, which is 1 at location idx. The server can compute $ct = \oplus_i \mathcal{E}_{l+1}(b_i).ct_i = \mathcal{E}_{l+1}(ct_{idx})$. This allows the server to obliviously select the "correct" ciphertext with an additional layer of encryption. This operation can be used to ensure $O(B)$ bandwidth for the access and the eviction protocols for blocks of size at least $\tilde{\Omega}(\log^4 N)$.

5.3 Our HSS based ORAM construction

Our construction largely has the same structure as the single server Onion ORAM construction [DvDF+16], with the server side computation in Onion ORAM divided across the 2 servers in our scheme using our HSS construction.

In our construction the two servers store two ORAM binary trees (T_0, T_1) similar to that in Onion ORAM, 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 $(b_1, b_2, \ldots, b_C)$, 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 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 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

Notation	Meaning
N	number of ORAM blocks
B	size of each block in bits
C	number of share chunks of each block
T_0, T_1	ORAM trees stored on Servers S_0 and S_1 respectively
L	depth of the ORAM trees
Z	number of blocks per bucket in the ORAM tree
$\mathcal{P}(l)$	path from root to lead l also a shorthand for the array of blocks on the same path
$\mathsf{md}_0, \mathsf{md}_1$	shares of meta-data stored on Servers S_0 and S_1
$\mathsf{md}[l]$	the array of meta-data for root to leaf blocks on $\mathcal{P}(l)$
$h([a_1, \ldots, a_k])$	same as the list $(h(a_1), h(a_2), \ldots, h(a_k))$, where h is a hash function
G	eviction counter
M'	bound on value of secret shares
M	bound on share conversion in RM circuit

Table 1: ORAM Notation and shorthand used in the protocol description

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 4).

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

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

$$\begin{aligned} |S| \le \sum_{j=0}^{m-1} \frac{(m-1)!}{j!} 2^j m^j M' &\le (m-1)!\,M' \sum_{j=0}^{\infty} \frac{(2m)^j}{j!} \\ &= (m-1)!\,M' e^{2m} \le M \end{aligned} \tag{2}$$

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 5.

$\underline{\text{SELECT}(i \in \mathsf{IN}, y_0 \in \mathsf{REG}, \ldots, y_{m-1} \in \mathsf{REG}):}$
$D[j] := \frac{(m-1)!}{j!} \sum_{k=0}^{j} \binom{j}{k} (-1)^{j-k} \times_{\mathsf{REG}} y_k$
$z := 0_{\mathsf{REG}}$
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

$\underline{\mathsf{SelectShare}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1, i, \{y_{0k}\}_k, \{y_{1k}\}_k):}$
$in[\text{“}i\text{”}] := \mathsf{Share}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1, \mathsf{IN}, i)$
for $j \in \{0,1\}$:
  for each chunk index c:
    for $k := 0$ to $m-1$:
      $in[\text{“}y\text{”} \parallel k] := y_{jk}[c]$
    $s_j[c] := \mathsf{Run}(\text{SELECT}, \mathsf{Hom}(j, \mathsf{pk}, \mathsf{sk}_j), in)$
return s_0, s_1

$\underline{\mathsf{Select}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1, i, \{y_{0k}\}_k, \{y_{1k}\}_k):}$
$(s_0, s_1) \leftarrow \mathsf{SelectShare}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1, i, \{y_{0k}\}_k, \{y_{1k}\}_k)$
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

Figure 4: 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 3 to be modified slightly, to not take its output modulo N^s, and we call this modification Decode′.

Using these two algorithms, we describe our Setup and Access function for our proposed ORAM scheme in Figure 6 and Figure 7 respectively.

Secure 2-party computation. The main technique for secure computation of RAM programs is to start with a circuit MPC protocol and use a multi-server ORAM protocol for every RAM access [GKK+12]. The client of the ORAM protocol must be run inside the MPC protocol, which puts a premium on reducing its computational complexity. The bottlenecks of client computation for the Access protocol are in Select, which must encrypt Damgård–Jurik ciphertexts as part of Share, and divide by φ as part of Decode. We can alleviate the former with the public-key sharing property (Definition 17): use generic MPC to give out shares $i_1 - i_0 = i$ of the plaintext, then use the public-key sharing protocol to compute $\mathsf{Share}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1, \mathsf{IN}, i_j)$ without using the secret key. The shares can then be combined as the first step of the RMS circuit being evaluated. Additive decoding (Definition 18) avoids the second problem by directly giving additive shares of the result, without any division.

These optimizations put the malicious security of the scheme in jeopardy, however. One server could encrypt an incorrect value instead of their share of i, or they could exploit the fact that the additive decoding algorithm does not authenticate its input. The solutions to these problems are very similar to how we authenticate the meta-data. The generic MPC protocol can provide an information theoretic MAC of the shares (e.g. shares of $H(i)$ where H satisfies the uniform difference property), which can then be checked inside the RMS circuit. The RMS circuit can produce an information theoretic MAC of its output as well, which then after decoding can be checked inside of MPC.

$\mathsf{Evict}(\mathsf{pk}, \mathsf{sk}_0, \mathsf{sk}_1, l_e, \mathsf{md}, \{x_{0k}\}_k, \{x_{1k}\}_k, \{y_{0k}\}_k, \{y_{1k}\}_k)$:

$remap :=$ array of zeros
for each block b of parent node $\lfloor \frac{l_e}{2} \rfloor$.
  if 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 $\{y_{0k}\}_k, \{y_{1k}\}_k$

Figure 5: 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.

5.4 Proof of Security

Intuitively, the adversary learns nothing looking at one servers binary tree's 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 25. *The distributed ORAM construction described in Figure 6 and Figure 7 satisfy the security Definition 23.*

Proof. See Appendix B.2. □

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 (2)) 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 $\Theta(\ell(\kappa))$ 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.

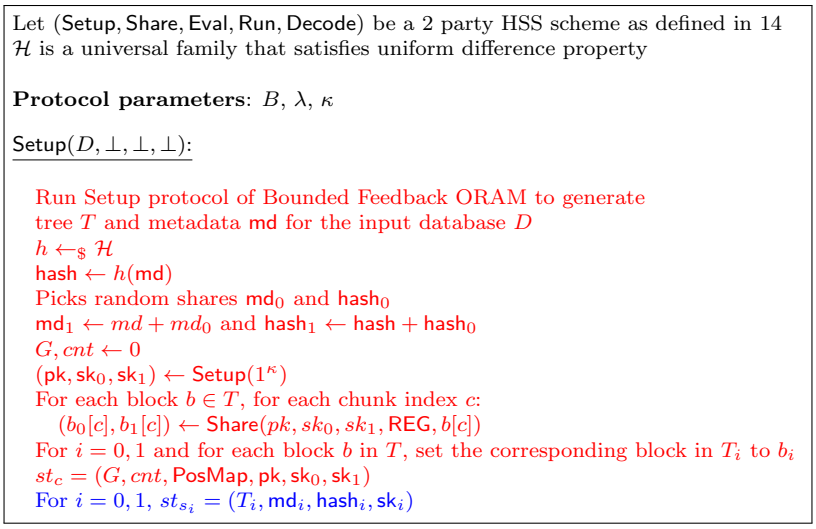


Figure 6: The 2-server distributed ORAM Setup function.
Client computation is colored red and server computation is colored blue

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 $\widetilde{O}(C\lambda \log N \ell^2(\kappa))$ time.

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 $\widetilde{O}(\log^3 N \ell^2(\kappa)) = \widetilde{O}(\log^9 N) = \widetilde{O}(B \log^5 N)$.

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 8. See also Appendix C.1, where we review the definition

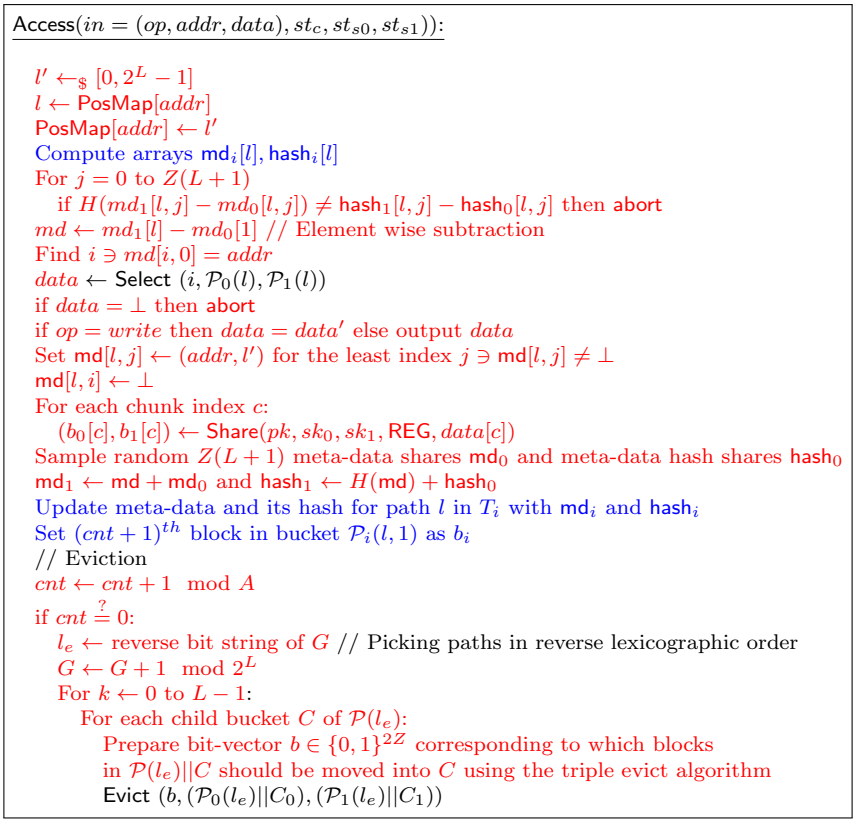


Figure 7: The 2-server distributed ORAM Access function.
Client computation is colored red and server computation is colored blue

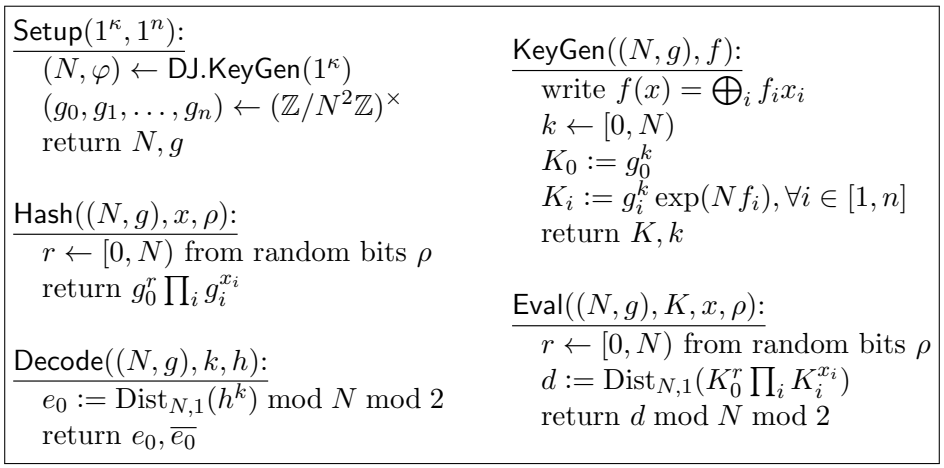


Figure 8: Trapdoor hash function for linear predicates from DCR based on based on our distance function, which achieves a negligible error rate.

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 26. *The construction in Figure 8 is a* $(1 - nN^{-1})$*-correct trapdoor hash function with rate* 1*.*

Proof. See Appendix C.2. □

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 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. The above generalization of TDH would instead allow TDH to output large chunks of data, with nearly $|h|$ bits of output per evaluation.

References

[BG10] Zvika Brakerski and Shafi Goldwasser. Circular and leakage resilient public-key encryption under subgroup indistinguishability (or: Quadratic residuosity strikes back). Cryptology ePrint Archive, Report 2010/226, 2010. `http://eprint.iacr.org/2010/226`.

[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.

[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.

[GKK+12] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 513–524. ACM Press, October 2012.

[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.

[Gou20] Fernando Q. Gouvêa. *p-adic Numbers: An Introduction*. Springer, Heidelberg, 3rd edition, 2020.

[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. Cryptology ePrint Archive, Report 2021/262, 2021. `https://eprint.iacr.org/2021/262`.

[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.

A HSS Proofs and Extras

A.1 Proofs for HSS Construction

Theorem 23. *The HSS scheme in Figure 3 is authenticated for wire types* $\{\mathsf{IN}, \mathsf{REG}\}$.

Proof. Let $\mathcal{A}$ be an adversary that makes calls to $\mathcal{H}$ from Definition 16 that has a non-negligible probability of making GUESS return TRUE. Clearly any call to GUESS that returns TRUE must use a dishonest share, i.e. $s_j \neq U[i]_j$. This immediately rules out guessing a share of type IN since Decode checks that the two shares are identical.

With REG shares, for neither call to Decode to return $\perp$ we must have $U[i]_1 - U[i]_0 \equiv_\varphi 0 \equiv_\varphi s_1 - s_0$. Since $s_{\bar{j}} = U[i]_{\bar{j}}$, this implies that $s_j \equiv_\varphi U[i]_j$. We will use this fact to build an adversary $\mathcal{A}'$ against the privacy of the HSS scheme. $\mathcal{A}'$ runs $\mathcal{A}$ while simulating its access to SHARE using the Share oracle $\mathcal{O}_{i,\mathsf{pk},\mathsf{sk}_0,\mathsf{sk}_1}$. It keeps track of the half of the honest shares $U[i]_j$ that it can compute, saving them in SHARE and updating them in EVAL, the same as $\mathcal{H}$ does. It always returns FALSE from GUESS, which is indistinguishable until $\mathcal{A}$ succeeds in causing $s_j \equiv_\varphi U[i]_j$.

$\mathcal{A}'$ checks for this case by computing the difference $\mu = s_j - U[i]_j$, and if it is nonzero seeing whether it is a multiple of φ by checking if $c^\mu \equiv_N 1$. If so, it computes the largest k such that $N^k \mid \mu$, and finds $\mu' = N^{-k}\mu$. μ' is sufficient to decrypt Damgård–Jurik ciphertexts:

$$\frac{\log(c^{\mu'})}{N\mu'} = \frac{\mu'}{\varphi}\frac{\log(c^\varphi)}{N\mu'} = \frac{\log(c^\varphi)}{N\varphi} = \mathsf{DJ.Dec}_{N,s,\varphi}(c).$$

Then a single query to $\mathcal{O}_{i,\mathsf{pk},\mathsf{sk}_0,\mathsf{sk}_1}(\mathsf{IN}, 0, 1)$ can be decrypted to find i, successfully attacking the privacy of the HSS scheme. □

A.2 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? We would need the all of the REG shares to be of multiplies of $\varphi\varphi'$ so that ciphertexts encrypted with either key can be decrypted. This would still leave the output as a multiple of φ' at the end, after the $\varphi\varphi^{-1}$ factor cancels, and it seems like we are back where we started. However, there is a fix: make N^s be much larger than $N'^{s'}$, and every REG share be a multiple of $\varphi\varphi'(\varphi'^{-1} \bmod N'^{s'})$, while still keeping them within the bound. Then this last decryption step will output modulo $N'^{s'}$, so φ' and $\varphi'^{-1} \bmod N'^{s'}$ will cancel.

We show the modifications to the HSS scheme in Figure 9. The biggest change is to Setup, which now computes the second key pair (N', φ') and gives out an encryption c' of $\mu = \varphi^{-1}$ under the second key. Because the only time we have an upper bound on the size of a plaintext value x is during $\mathsf{eval}_j(\phi, r)$, we take that opportunity to compute additive shares of x, using c' and shares of φx. We change $\mathsf{values}_j(\mathsf{REG})$ $\mathbb{Z} \times \mathbb{Z}$ to store both additive shares. Finally, we change Share and Decode to match, so that they encode and decode shares in $\mathbb{Z} \times \mathbb{Z}$.

Lemma 23. *Assuming DCR, the modified scheme in Figure 9 is a* $(1-p)$*-correct HSS scheme that is authenticated for wire types* $\{\mathsf{IN}, \mathsf{REG}\}$ *and has additive decoding for* REG*, where* $p = M\left(N^{1-s}N'^{s'+1} + N'^{-s'}\right)$.

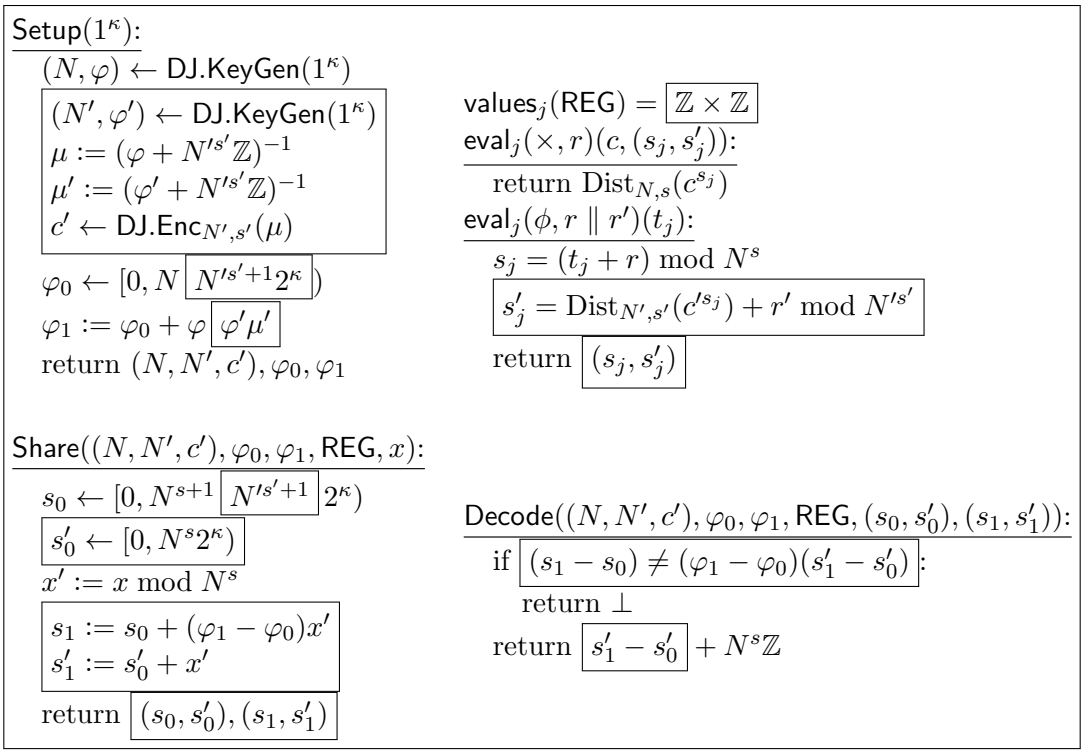


Figure 9: Modifications to the HSS scheme in Figure 3 needed to support additive decoding. The changes are boxed, and only those functions that have been modified are shown.

Proof. The proof is very similar to Theorem 22, so we will only give the differences. For correctness, we have that $s_1 - s_0 = (\varphi_1 - \varphi_0)x' = (\varphi_1 - \varphi_0)(s'_1 - s'_0)$, so Decode will output $x' + N^s\mathbb{Z} = x$. For homomorphism we need to prove that the new eval$_j(\phi, r)$ is correct. The same analysis as before gives that the probability of failure when finding s_j is at most $\frac{(\varphi_1 - \varphi_0)M}{N^s}$, which this time is upper bounded by $MN^{1-s}N'^{s'+1}$, because $\varphi' < N'$ and $\mu' < N'^{s'}$.

Calculating s'_j gives a second source of error. While the it will be correct modulo $N'^{s'}$ because of a similar distance function argument to before, based on $s_1 - s_0$ always being a multiple of φ', there is still the modulo operation at the end. Because $s'_1 - s'_0 \equiv_{N'^{s'}} \varphi\varphi'\mu'x\mu \equiv_{N'^{s'}} x$, where x is the decoded value of the share, Lemma 21 implies that $s'_1 - s'_0 = x$ other than with probability at most $\frac{x}{N'^{s'}} \leq MN'^{-s'}$. Adding these two error probabilities gives the correctness bound.

We show privacy using a sequence of hybrids.

1. Replace φ_1 with a uniformly random integer in $[0, NN'^{s+1}2^\kappa)$. If $j = 0$ this makes no difference from the adversaries perspective. If $j = 1$ then φ_1 was uniformly random in $[\varphi\varphi'\mu', \varphi\varphi'\mu' + NN'^{s+1}2^\kappa)$. This is only distinguishable with advantage $\frac{\varphi\varphi'\mu'}{NN'^{s+1}2^\kappa} < 2^{-\kappa}$ because the distributions are identical in all but that fraction of the possibilities.

2. Do the same to the share (s_1, s'_1) produced by Share on REG wire type, and to the share s_1 from MUL. A similar argument gives a distinguisher advantage bound of $2^{-\kappa}$ per Share call.

3. Sample $c' \leftarrow$ DJ.Enc$_{N',s'}(0)$ instead of encrypting μ. This is indistinguishable by the CPA security of Damgård–Jurik encryption as φ' and μ' are unused.

4. In Share, generate c as DJ.Enc$_{N',s'}(0)$. Again, this is indistinguishable by the CPA security of Damgård–Jurik encryption, because φ and μ are now unused.

In this final hybrid the input x to Share is unused, so $\mathcal{O}_i$ can be swapped for $\mathcal{O}_{\bar{i}}$ in the privacy distinguisher game.

Authentication follows a very similar argument to before. It's impossible to fake a IN share because the two parties have identical shares, and any adversary that can fake a REG share can find $\varphi\varphi'\mu'$, then use it to decrypt the ciphertexts and break privacy.

Finally, we have additive decoding for wire type REG because when Decode does not error it must return $s'_1 - s'_0 + N^s\mathbb{Z}$. □

Choosing appropriate parameters is a bit trickier than the previous construction, but since N and N' are of approximately the same size we can roughly minimize the error probability by choosing $s = 2s' + 2$. Then $p \approx 2MN^{-s'}$, which is clearly negligible as a function of N and s'.

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 everything else was based on that one share type.

B ORAM Details

B.1 Security Definition

We define the ideal and real worlds as follows:

Ideal World The ideal functionality $\mathcal{F}_{ORAM}$ maintains the updated vector D and it correctly answers each of client's queries.

- Setup The environment $\mathcal{Z}$ sends the initial database D to the client. The client forwards this database D to $\mathcal{F}_{ORAM}$, which sends $N = |D|$ to the simulator $\mathcal{S}$ and the fact that the Setup function was being invoked. The simulator returns ok or abort to $\mathcal{F}_{ORAM}$, and for both these cases $\mathcal{F}_{ORAM}$ forwards to the client ok or $\perp$ respectively.
- Access The environment $\mathcal{Z}$ sends the operation $op = (read, idx)$ or $op = (write, idx, data)$ to the client, which it forwards to the ideal functionality $\mathcal{F}_{ORAM}$. $\mathcal{F}_{ORAM}$ notifies $\mathcal{S}$ that the Access function is being invoked without leaking the operation op. If $\mathcal{S}$ says ok, the $\mathcal{F}_{ORAM}$ outputs $D[idx]$ to the client if it was the read operation, and for a write operation it updates $D[idx] \leftarrow data$. If $\mathcal{S}$ returns abort, $\mathcal{F}_{ORAM}$ returns $\perp$. The client forwards the message from $\mathcal{F}_{ORAM}$ to $\mathcal{Z}$.

Real World The environment $\mathcal{Z}$ sends the initial database D to the client. The client runs the Setup protocol with the two servers S_0 and S_1. The adversary $\mathcal{A}$ controls the behavior of only one of these two servers. Whenever the environment $\mathcal{Z}$ sends the operation $op = (read, idx)$ or $op = (write, idx, data)$ to the client, it runs the Access protocol with the two servers. The client forwards its output of the Access protocol to $\mathcal{Z}$ and the adversary outputs its view to $\mathcal{Z}$ after each operation.

Definition 23. *(Simulation based security definition for privacy+verifiability) A protocol $\prod_{ORAM}$ securely realizes the ideal functional $\mathcal{F}_{ORAM}$ if for any probabilistic polynomial-time real-world adversary $\mathcal{A}$ there exist a simulator $\mathcal{S}$, such that for all non-uniform, polynomial-time environment $\mathcal{Z}$ we have that:*

$$|Pr[REAL_{\prod_{ORAM},\mathcal{A},\mathcal{Z}} = 1] - Pr[IDEAL_{\mathcal{F}_{ORAM},\mathcal{S},\mathcal{Z}} = 1]| \leq negl(\lambda)$$

For some negligible function $negl$ and security parameter λ.

B.2 ORAM Proof

Theorem 25. *The distributed ORAM construction described in Figure 6 and Figure 7 satisfy the security Definition 23.*

Proof. We describe the simulator and the sequence of hybrids to show that the real world and the ideal world simulation are indistinguishable.

Simulator $\mathcal{S}$: The simulator runs the Setup protocol on behalf of the honest client and the honest server on a dummy database D' of size N containing all zero blocks. For each Access operation the simulator runs the honest Access protocol using a dummy index $idx' = 0$ for both the honest client and server. If the simulated client ever aborts during the protocol, then the simulator sends abort to $\mathcal{F}_{ORAM}$ and stops responding to future Access operations, else it sends ok to $\mathcal{F}_{ORAM}$.

Sequence of hybrids. Next we show a sequence of hybrids which prove that the real and the ideal simulation are indistinguishable.

Game 0 This is the real game $REAL_{\prod_{ORAM},\mathcal{A},\mathcal{Z}}$.

Game 1 The client simulates both servers honestly, and uses the shares and shared meta-data returned from these honest servers instead of from the real servers. The real shares are only used to determine if the protocol should abort, with the same condition as before.

Game 0 and Game 1 can only differ if the honest shares and real shares (of either the data or the meta-data) decode to distinct non-aborting results. For the data, this would violate the authentication property Definition 16. Similarly, the meta-data is authenticated using the universal hash function.

Let $\mathsf{md}_0, \mathsf{md}_1$ be the honest meta-data shares, and let $\mathsf{md}'_0, \mathsf{md}'_1$ be the real meta-data shares. Similarly, let $H_{\mathsf{md}_0}, H_{\mathsf{md}_1}$ and $H'_{\mathsf{md}_0}, H'_{\mathsf{md}_1}$ be the honest and real shares of the universal hash of the meta-data. Because at most one party has been corrupted, we have $\mathsf{md}_i = \mathsf{md}'_i$ and $H_{\mathsf{md}_i} = H'_{\mathsf{md}_i}$ for some $i \in \{0, 1\}$. We have $H(\mathsf{md}_1 - \mathsf{md}_0) = H_{\mathsf{md}_1} - H_{\mathsf{md}_0}$ and $H(\mathsf{md}'_1 - \mathsf{md}'_0) = H'_{\mathsf{md}_1} - H'_{\mathsf{md}_0}$ because the honest shares always authenticate correctly and because the protocol must not abort for a difference to occur between the two games. Subtracting these two equations, we get

$$H(\mathsf{md}'_1 - \mathsf{md}'_0) - H(\mathsf{md}_1 - \mathsf{md}_0) = H'_{\mathsf{md}_1} - H'_{\mathsf{md}_0} - H_{\mathsf{md}_1} + H_{\mathsf{md}_0} = (-1)^i (H'_{\mathsf{md}_{\bar{i}}} - H_{\mathsf{md}_{\bar{i}}})$$

Notice that the right-hand side of the equation is known to the corrupted party $\bar{i}$, as one is the share they provided and the other is the share that they would have given had they been honest. Given only the honest plaintexts $\mathsf{md}_1 - \mathsf{md}_0$ and a single party's shares, which contains no information about the chosen hash H from the universal family, it must have been possible to find the difference between two hashes. Therefore, this event has negligible probability by the uniform differences property of the universal hash family.

Game 2 Instead of simulating two honest servers and decoding the results of their shares, the client will now instead stores the data and meta-data plaintexts and operate on them instead. For the meta-data, this is equivalent by the correctness of secret sharing. For the data, this follows from the HSS scheme's correctness, as applied to a whole circuit in Lemma 15.

Game 3 Introduce an ideal functionality $\mathcal{F}_{ORAM}$ which stores the correct database D and the client sends it the Access query whenever it doesn't abort. The equivalence of this computation comes from the correctness of the select and eviction operations, and the correctness of Onion ORAM.

Game 4 Make the client run the setup procedure with the real servers using the dummy database D', as used by the simulator, instead of the one used by the environment. Additionally, only share dummy information in SelectShare, so i is always set to 0. These changes are indistinguishable by the privacy property of the HSS scheme, which implies that changing the input to Share is indistinguishable from the point of view of a single server.

Game 5 For each Access function call, the client just runs a dummy read operation on index $idx' = 0$ instead of using the operation input by the environment. Game 4 and 5 are indistinguishable given that the physical memory access pattern on the server is indistinguishable for any polynomial sequence of operations. This follows directly from the privacy of Onion ORAM.

After these hybrids, we are now at the simulator. □

C Trapdoor Hash Function Details

C.1 Definition

We adapt the following definition from [DGI+19]. The only changes are to the notation, and that we allow the challenge parameters for the security properties to be chosen adaptively depending on the common reference string. The main notational differences are that what they call the hash key we call the common reference string, their encoding key is our public key, and their trapdoor is our private key. We believe that these changes make the resemblance between trapdoor hashes and homomorphic encryption more clear. We also write the properties as being algorithms producing indistinguishable distributions.

Definition 25. *A $(1-p)$-correct trapdoor hash function $\mathcal{H}$ for a class of predicates $\{\mathcal{F}_n\}_n$ for each input size n ($\mathcal{F}_n \colon \{0,1\}^n \to \{0,1\}$) consists of the algorithms*

- $\mathsf{crs} \leftarrow \mathsf{Setup}(1^\kappa, 1^n)$ *samples the common reference string.*
- $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(\mathsf{crs}, f)$ *creates a key pair that can be used to evaluate $f \in \mathcal{F}_n$ and decode the result, respectively.*
- $h := \mathsf{Hash}(\mathsf{crs}, x; \rho)$ *hashes an input $x \in \{0,1\}^n$ using random bit stream ρ.*
- $e := \mathsf{Eval}(\mathsf{crs}, \mathsf{pk}, x; \rho)$ *evaluates the predicate represented by* pk *on x, returning the encoded result.*
- $(e_0, e_1) \leftarrow \mathsf{Decode}(\mathsf{crs}, \mathsf{sk}, h)$ *takes a hash h and outputs the encodings corresponding to the possible predicate outputs, 0 and 1.*

such that the following conditions hold.

Correctness *For any $x \in \{0,1\}^n$, $\mathsf{Eval}(\mathsf{crs}, \mathsf{pk}, x; \rho)$ must output $e_{f(x)}$, where pk was chosen using f. That is, for any efficient adversary $\mathcal{A}$ and any n, the distribution*

$$\boxed{\begin{array}{l}\mathsf{crs} \leftarrow \mathsf{Setup}(1^\kappa, 1^n)\\ (x, f) \leftarrow \mathcal{A}(\mathsf{crs})\\ \rho \leftarrow \$\\ h := \mathsf{Hash}(\mathsf{crs}, x, \rho)\\ (\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(\mathsf{crs}, f)\\ e := \mathsf{Eval}(\mathsf{crs}, \mathsf{pk}, x, \rho)\\ e_0, e_1 \leftarrow \mathsf{Decode}(\mathsf{crs}, \mathsf{sk}, h)\\ \text{return } e, e_0, e_1\end{array}}$$

has only negligible probability of giving $e \neq e_{f(x)}$, and $\Pr\left[e = e_{\overline{f(x)}}\right] \leq p$. Here, $\rho \leftarrow \$$ denotes generating a uniformly random bit stream.

Function Privacy *f cannot be determined from pk. More precisely, the distributions*

$$\boxed{\begin{array}{l}\mathsf{crs} \leftarrow \mathsf{Setup}(1^\kappa, 1^n)\\ (\text{view}, f_0, f_1) \leftarrow \mathcal{A}(\mathsf{crs})\\ (\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(\mathsf{crs}, f_0)\\ \text{return view}, \mathsf{pk}\end{array}} \quad \boxed{\begin{array}{l}\mathsf{crs} \leftarrow \mathsf{Setup}(1^\kappa, 1^n)\\ (\text{view}, f_0, f_1) \leftarrow \mathcal{A}(\mathsf{crs})\\ (\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(\mathsf{crs}, f_1)\\ \text{return view}, \mathsf{pk}\end{array}}$$

must be indistinguishable, for any n and PPT adversary $\mathcal{A}$.

Input Privacy Hash *leaks nothing about its input x. Formally, for any efficient adversary $\mathcal{A}$ the following distributions must be indistinguishable.*

$$\boxed{\begin{array}{l}\mathsf{crs} \leftarrow \mathsf{Setup}(1^\kappa, 1^n)\\ (\text{view}, x_0, x_1) \leftarrow \mathcal{A}(\mathsf{crs})\\ \rho \leftarrow \$\\ h := \mathsf{Hash}(\mathsf{crs}, x_0, \rho)\\ \text{return view}, h\end{array}} \quad \boxed{\begin{array}{l}\mathsf{crs} \leftarrow \mathsf{Setup}(1^\kappa, 1^n)\\ (\text{view}, x_0, x_1) \leftarrow \mathcal{A}(\mathsf{crs})\\ \rho \leftarrow \$\\ h := \mathsf{Hash}(\mathsf{crs}, x_1, \rho)\\ \text{return view}, h\end{array}}$$

Compactness *The output of Hash must have size bounded by a polynomial in κ, independent of n.*

The rate *of $\mathcal{H}$ is $|e|$, the length in bits of Eval's output.*

C.2 Proof

The proof of function privacy for our scheme hinges on the following statement.

Lemma 25. *For any PPT adversary $\mathcal{A}$ and any positive integers n, s the following distributions are indistinguishable.*

$$\boxed{\begin{array}{l}(N, \varphi) \leftarrow \mathsf{DJ.KeyGen}(1^\kappa)\\ (g_1, \ldots, g_n) \leftarrow (\mathbb{Z}/N^{s+1}\mathbb{Z})^\times\\ \text{view}, (a_1, \ldots, a_n) \leftarrow \mathcal{A}(N, g_1, \ldots, g_n)\\ r \leftarrow [0, N)\\ \text{return view}, a_1 g_1^r, \ldots, a_n g_n^r\end{array}} \quad \boxed{\begin{array}{l}(N, \varphi) \leftarrow \mathsf{DJ.KeyGen}(1^\kappa)\\ (g_1, \ldots, g_n) \leftarrow (\mathbb{Z}/N^{s+1}\mathbb{Z})^\times\\ \text{view}, (a_1, \ldots, a_n) \leftarrow \mathcal{A}(N, g_1, \ldots, g_n)\\ r \leftarrow [0, N)\\ \text{return view}, g_1^r, \ldots, g_n^r\end{array}}$$

Proof. This is almost exactly [BG10, Lem. B.1]. The only differences are that here each g_i is sampled in $(\mathbb{Z}/N^{s+1}\mathbb{Z})^\times$ instead of in the subgroup of perfect powers of N^s, and r is sampled in $[0, N)$ instead of $[1, N^{2(s+1)}]$. But the perfect powers of N^s have multiplicative order dividing φ, and modulo φ the uniform distributions on both $[0, N)$ and $[1, N^{2(s+1)}]$ have negligible statistical distance to uniform. After applying this change in a hybrid, we then use that choosing g_i to be a random perfect power of N^s is equivalent to sampling $g_i \leftarrow \mathsf{DJ.Enc}_{N,s}(0)$, so the CPA\$ security of Damgård–Jurik implies that it is indistinguishable to sample $g_i \leftarrow (\mathbb{Z}/N^{s+1}\mathbb{Z})^\times$. □

Theorem 26. *The construction in Figure 8 is a $(1 - nN^{-1})$-correct trapdoor hash function with rate 1.*

Proof. There are four conditions that must be met.

Correctness Let $d_0 = \mathrm{Dist}_{N,1}(h^k)$ and $d_1 = \mathrm{Dist}_{N,1}(K_0^r \prod_i K_i^{x_i})$. Then by Theorem 19, we have

$$\begin{aligned} d_1 - d_0 &= \mathrm{Dist}_{N,1}\left(g_0^{kr} \prod_i g_i^{kx_i} \exp(N f_i x_i)\right) - \mathrm{Dist}_{N,1}(g_0^{kr} \prod_i g_i^{kx_i}) \\ &= \sum_i f_i x_i. \end{aligned}$$

Lemma 25 shows that g_0^r is indistinguishable from a uniformly random element of $(\mathbb{Z}/N^2\mathbb{Z})^\times$, so Corollary 20 implies that d_0 is uniformly random. Then by Lemma 21, $d_1 \bmod N - d_0 \bmod N = \sum_i f_i x_i$ with probability at least $1 - \frac{n}{N}$. Assuming that this holds, $e - e_0 \equiv_2 \sum_i f_i x_i \equiv_2 f(x)$, and so $e = e_{f(x)}$ and $e \neq e_{\overline{f(x)}}$. Therefore the correctness properties fail with probability at most $\frac{n}{N}$.

Function Privacy By Lemma 25, it would be indistinguishable to sample every K_i as g_i^k, i.e. to set every f_i to be zero. Now the distribution of pk does not depend on f, so privacy follows.

Input Privacy By Lemma 25, g_0^r is indistinguishable from uniformly random in $(\mathbb{Z}/N^2\mathbb{Z})^\times$, so it completely hides every x_i.

Compactness The output of Hash is in $(\mathbb{Z}/N^2\mathbb{Z})^\times$, which has size bounded by $4\ell(\kappa)$, a polynomial in κ.

□