Homomorphic Secret Sharing: Optimizations and Applications

Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Michele Orrù

ABSTRACT

We continue the study of Homomorphic Secret Sharing (HSS), recently introduced by Boyle et al. (Crypto 2016, Eurocrypt 2017). A (2-party) HSS scheme splits an input x into shares (x^0, x^1) such that (1) each share computationally hides x, and (2) there exists an efficient homomorphic evaluation algorithm Eval such that for any function (or "program") P from a given class it holds that $\mathsf{Eval}(x^0, P) + \mathsf{Eval}(x^1, P) = P(x)$. Boyle et al. show how to construct an HSS scheme for branching programs, with an inverse polynomial error, using discrete-log type assumptions such as DDH.

We make two types of contributions.

Optimizations. We introduce new optimizations that speed up the previous optimized implementation of Boyle et al. by more than a factor of 30, significantly reduce the share size, and reduce the rate of leakage induced by selective failure.

Applications. Our optimizations are motivated by the observation that there are natural application scenarios in which HSS is useful even when applied to simple computations on short inputs. We demonstrate the practical feasibility of our HSS implementation in the context of such applications.

KEYWORDS

Homomorphic Secret Sharing, Homomorphic Encryption, Secure Computation

1 INTRODUCTION

Fully homomorphic encryption (FHE) [26, 42] is commonly viewed as a "dream tool" in cryptography, enabling one to perform arbitrary computations on encrypted inputs. In the context of secure multiparty computation (MPC) [6, 17, 31, 45], FHE can be used to minimize the communication complexity and the round complexity, and shift the bulk of the computational work to any subset of the participants.

However, despite exciting progress in the past few years, even the most recent implementations of FHE [18, 24, 32] are still quite slow and require large ciphertexts and keys. This is due in part to the limited set of assumptions on which FHE constructions can be based [15, 27, 44], which are all related to lattices and are therefore susceptible to lattice reduction attacks. As a result, it is arguably hard to find realistic application scenarios in which current FHE implementations outperform optimized versions of classical secure computation techniques (such as garbled circuits) when taking both communication and computation costs into account.

Homomorphic secret sharing. An alternative approach that provides some of the functionality of FHE was introduced in the recent work of Boyle et al. [11] and further studied in [13]. The high level idea is that for some applications, the traditional notion of FHE can be relaxed by allowing the homomorphic evaluation to be distributed among two parties who do not interact with each other.

This relaxation is captured by the following natural notion of *homomorphic secret sharing* (HSS). A (2-party) HSS scheme randomly splits an input x into a pair of shares (x^0, x^1) such that: (1) each share x^b computationally hides x, and (2) there exists a polynomial-time local evaluation algorithm Eval such that for any "program" P (e.g., a boolean circuit, formula or branching program), the output $P(x)$ can be efficiently reconstructed from $\mathsf{Eval}(x^0, P)$ and $\mathsf{Eval}(x^1, P)$.

As in the case of FHE, we require that the output of Eval be *compact* in the sense that its length depends only on the output length $|P(x)|$ but not on the size of P. But in fact, a unique feature of HSS that distinguishes it from traditional FHE is that the output representation can be *additive*. That is, we require that $\mathsf{Eval}(x^0, P) + \mathsf{Eval}(x^1, P) = P(x) \bmod \beta$ for some positive integer $\beta \geq 2$ that can be chosen arbitrarily. This enables an ultimate level of compactness and efficiency of reconstruction that is impossible to achieve via standard FHE. For instance, if P outputs a single bit and $\beta = 2$, then the output $P(x)$ is reconstructed by taking the exclusive-or of two bits.

The main result of [11] is an HSS scheme sfor *branching programs* under the Decisional Diffie-Hellman (DDH) assumption.[1] At a small additional cost, this HSS scheme admits a public-key variant, which enables homomorphic computations on inputs that originate from multiple clients. In this variant, there is a common public key pk and two secret evaluation keys $(\mathsf{ek}_0, \mathsf{ek}_1)$. Each input x_i can now be separately encrypted using pk into a ciphertext ct_i, such that ct_i together with a single evaluation key ek_b do not reveal x_i. The homomorphic evaluation can now apply to any set of encrypted inputs, using only the ciphertexts and one of the evaluation keys. That is, $\mathsf{Eval}(\mathsf{ek}_0, (\mathsf{ct}_1, \ldots, \mathsf{ct}_n), P) + \mathsf{Eval}(\mathsf{ek}_1, (\mathsf{ct}_1, \ldots, \mathsf{ct}_n), P) = P(x) \bmod \beta$.

The HSS scheme from [11] has been later optimized in [13], where the security of the optimized variants relies on other discrete-log style assumptions (including a "circular security" assumption for ElGamal encryption). These HSS schemes for branching programs can only satisfy a relaxed form of δ-correctness, where the (additive) reconstruction of the output may fail with probability δ and where the running time of Eval grows linearly with $1/\delta$. As negative byproducts, the running time of Eval must grow quadratically with the size of the branching program, and one also needs to cope with input-dependent and key-dependent leakage introduced by selective failure. The failure probability originates from a share conversion procedure that *locally* converts multiplicative shares into additive shares. See Section 3 for a self-contained exposition of the HSS construction from [11] that we build on.

[1] HSS for general circuits can be based on LWE via multi-key FHE [23] or even threshold FHE [10, 23]. Since these enhanced variants of FHE are even more inefficient than standard FHE, these constructions cannot get around the efficiency bottlenecks of FHE. We provide a brief comparison with LWE-based approaches in the full version [9].

The main motivating observation behind this work is that unlike standard FHE, HSS can be useful even for *small computations* that involve short inputs, and even in application scenarios in which competing approaches based on traditional secure computation techniques do not apply at all. Coupled with the relatively simple structure of the group-based HSS from [11] and its subsequent optimizations from [13], this gives rise to attractive new applications that motivate further optimizations and refinements.

Before discussing our contribution in more detail, we would like to highlight the key competitive advantages of HSS over alternative approaches.

Optimally compact output. The optimal compactness feature discussed above enables applications in which the communication and computation costs of output reconstruction need to be minimized, e.g., for the purpose of reducing power consumption. For instance, a mobile client may wish to get quickly notified about live news items that satisfy certain secret search criteria, receiving a fast real-time feed that reveals only pointers to matching items. HSS also enables applications in which the parties want to generate large amounts of correlated randomness for the purpose of speeding up an anticipated invocation of a classical secure computation protocol. Generating such correlated randomness non-interactively provides a good protection against traffic analysis attacks that try to obtain information about the identity of the interacting parties, the time of the interaction, and the size of the computation that is about to be performed. This "low communication footprint" feature can be used more broadly to motivate secure computation via FHE. However, the optimal output compactness of HSS makes it the only available option for applications that involve computing long outputs (or many short outputs) from short secret inputs (possibly along with public inputs). We explore several such applications in this work. Other advantages of group-based HSS over existing FHE implementations include smaller keys and ciphertexts and a lower start up cost.

Minimal interaction. HSS enables secure computation protocols that simultaneously offer a minimal amount of interaction and collusion resistance. For instance, following a reusable setup, such protocols can involve a single message from each "input client" to each server, followed by a single message from each server to each "output client." Alternatively, the servers can just publicize their shares of the output if the output is to be made public. The security of such protocols holds even against (semi-honest) adversaries who may corrupt an arbitrary subset of parties that includes only one of the two servers. Such protocols (a special type of 2-round MPC protocols) cannot be obtained using classical MPC techniques and are only known under indistinguishability obfuscation [25], special types of fully homomorphic encryption [23, 39], or HSS [13].

1.1 Our Contribution

We make two types of contributions, extending both the efficiency and the applicability of the recent constructions of group-based HSS from [11, 13].

Optimizations. We introduce several new optimization ideas that further speed up the previous optimized implementation from [13], reduce the key and ciphertext sizes, and reduce the rate of leakage of inputs and secret keys.

Computational optimizations. We show that a slight modification of the share conversion procedure from [13] can reduce the expected computational cost by a factor 16 or more, for the same failure probability. (As in [13], the failure is of a "Las Vegas" type, namely if there is any risk of a reconstruction error this is indicated by the outputs of Eval.) Together with an additional machine-level optimizations, we reduce the computational cost of Eval by more than a factor of 30 compared to the optimized implementation from [13].

Improved key generation. We describe a new protocol for distributing the key generation for public-key HSS, which eliminates a factor-2 computational overhead in all HSS applications that involve inputs from multiple clients.

Ciphertext size reduction. We suggest a method to reduce the ciphertext size of group-based HSS by roughly a factor of 2, relying on a new entropic discrete-log type assumption. In addition, we show how to make HSS ciphertexts extremely succinct (at the cost of a higher evaluation time) using bilinear maps in the full version of this paper [9].

Reducing leakage rate. We suggest several new methods to address input-dependent and key-dependent failures introduced by the share conversion procedure, mitigating the risk of leakage at a lower concrete cost than the previous techniques suggested in [11, 13]. These include "leakage-absorbing pads" that can be used to reduce the effective leakage probability from δ to $O(\delta^2)$ at a low cost.

Extensions and further optimizations. We exploit the specific structure of group-based HSS to enrich its expressiveness, and to improve the efficiency of homomorphic natural types of functions, including low-degree polynomials, branching programs, and boolean formulas. One particularly useful extension allows an efficient evaluation of a function that discloses a short bit-string (say, a cryptographic key) under a condition expressed by a branching program.

Applications. As noted above, our optimizations are motivated by the observation that there are natural application scenarios in which HSS is useful even for simple computations. These include small instances of general secure multiparty computation, as well as distributed variants of private information retrieval, functional encryption, and broadcast encryption. We demonstrate the practical feasibility of our optimized group-based HSS implementation in the context of such applications by providing concrete efficiency estimates for useful choices of the parameters.

Secure MPC with minimal interaction. Using public-key HSS, a set of clients can outsource a secure computation to two non-colluding servers by using the following minimal interaction pattern: each client independently sends a single message to the servers (based on its own input and the public key), and then each server sends a single message to each client. Alternatively, servers can just publish shares of the output if the output is to be made public. The resulting protocol is resilient to any (semi-honest) collusion between one server and a subset of the clients, and minimizes the amount of work performed by the clients. It is particularly attractive in the case where many "simple" computations are performed on the same

inputs. In this case, each additional instance of secure computation involves just local computation by the servers, followed by a minimal amount of communication and work by the clients.

Secure data access. We consider several different applications of HSS in the context of secure access to distributed data. First, we use HSS to construct a 2-server variant of attribute based encryption, in which each client can access an encrypted file only if its (public or encrypted) attributes satisfy an encrypted policy set up by the data owner. We also consider a 2-server private RSS feed, in which clients can get succinct notifications about new data that satisfies their encrypted matching criteria, and 2-server PIR schemes with general boolean queries. The above applications benefit from the optimal output compactness feature of HSS discussed above, minimizing the communication from servers to clients and the computation required for reconstructing the output.

Unlike competing solutions based on classical secure computation techniques, our HSS-based solutions only involve minimal interaction between clients and servers and no direct interaction between servers. In fact, for the RSS feed and PIR applications, the client is free to choose an arbitrary pair of servers who have access to the data being privately searched. These servers do not need to be aware of each other's identity, and do not even need to know they are participating in an HSS-based cryptographic protocol: each server can simply run the code provided by the client on the (relevant portion of) the data, and return the output directly to the client.

Correlated randomness generation. HSS provides a method for non-interactively generating sources of correlated randomness that can be used to speed up classical protocols for secure two-party computation. Concretely, following a setup phase, in which the parties exchange HSS shares of random inputs, the parties can *locally* expand these shares (without any communication) into useful forms of correlated randomness. As discussed above, the non-interactive nature of the correlated randomness generation is useful for hiding the identities of the parties who intend to perform secure computation, as well as the time and the size of the computation being performed. The useful correlations we consider include bilinear correlations (which capture "Beaver triples" as a special case) and truth-table correlations. We also study the question of compressing the communication in the setup phase by using local PRGs, and present different approaches for improving its asymptotic computational complexity. However, this PRG-based compression is still too slow to be realized with good concrete running time using our current implementation of group-based HSS.

For all applications, we discuss the applicability of our general optimization techniques, and additionally discuss specialized optimization methods that target specific applications.

1.2 Related work

The first study of secret sharing homomorphisms is due to Benaloh [7], who presented constructions and applications of additively homomorphic secret sharing schemes.

Most closely relevant to the notion of HSS considered here is the notion of *function secret sharing* (FSS) [10], which can be viewed as a dual version of HSS. Instead of evaluating a given function on a secret-shared input, FSS considers the goal of evaluating a secret-shared function on a given input. For simple function classes, such as point functions, very efficient FSS constructions that rely only on one-way functions are known [10, 12]. However, these constructions cannot be applied to more complex functions as the ones we consider here except via a brute-force approach that scales exponentially with the input length. Moreover, current efficient FSS techniques do not apply at all to computations that involve inputs from two or more clients, which is the case for most of the applications considered in this work.

2 PRELIMINARIES

2.1 Homomorphic Secret Sharing

We consider homomorphic secret sharing (HSS) as introduced in [11]. By default, in this work, the term HSS refers to a public-key variant of HSS (known as DEHE in [11]), with a Las Vegas correctness guarantee. To enable some of the optimizations we consider, we use here a slight variation of the definition from [13] that allows for an output to be computed even when one of the two parties suspects an error might occur.

Definition 2.1 (Homomorphic Secret Sharing). A (2-party, public-key, Las Vegas δ-failure) *Homomorphic Secret Sharing (HSS)* scheme for a class of programs $\mathcal{P}$ consists of algorithms (Gen, Enc, Eval) with the following syntax:

- $\mathsf{Gen}(1^\lambda)$: On input a security parameter 1^λ, the key generation algorithm outputs a public key pk and a pair of evaluation keys $(\mathsf{ek}_0, \mathsf{ek}_1)$.
- $\mathsf{Enc}(\mathsf{pk}, x)$: Given public key pk and secret input value $x \in \{0,1\}$, the encryption algorithm outputs a ciphertext ct. We assume the input length n is included in ct.
- $\mathsf{Eval}(b, \mathsf{ek}_b, (\mathsf{ct}_1, \ldots, \mathsf{ct}_n), P, \delta, \beta)$: On input party index $b \in \{0,1\}$, evaluation key ek_b, vector of n ciphertexts, a program $P \in \mathcal{P}$ with n input bits and m output bits, failure probability bound $\delta > 0$, and an integer $\beta \geq 2$, the homomorphic evaluation algorithm outputs $y_b \in \mathbb{Z}_\beta^m$, constituting party b's share of an output $y \in \{0,1\}^m$, as well as a confidence flag $\gamma_b \in \{\perp, \top\}$ to indicate full confidence ($\top$) or a possibility of failure ($\perp$). When β is omitted it is understood to be $\beta = 2$.

The algorithms Gen, Enc are PPT algorithms, whereas Eval can run in time polynomial in its input length and in $1/\delta$. The algorithms (Gen, Enc, Eval) should satisfy the following correctness and security requirements:

- **Correctness:** For every polynomial p there is a negligible ν such that for every positive integer λ, input $x \in \{0,1\}^n$, program $P \in \mathcal{P}$ with input length n, failure bound $\delta > 0$ and integer $\beta \geq 2$, where $|P|, 1/\delta \leq p(\lambda)$, we have:

$$\Pr[(\gamma_0 = \perp) \wedge (\gamma_1 = \perp)] \leq \delta + \nu(\lambda),$$

and

$$\Pr[((\gamma_0 = \top) \vee (\gamma_1 = \top)) \wedge y_0 + y_1 \neq P(x_1, \ldots, x_n)] \leq \nu(\lambda),$$

where probability is taken over

$$(\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \leftarrow \mathsf{Gen}(1^\lambda); \mathsf{ct}_i \leftarrow \mathsf{Enc}(\mathsf{pk}, x_i),\ i \in [n];$$

$$(y_b, \gamma_b) \leftarrow \mathsf{Eval}(b, \mathsf{ek}_b, (\mathsf{ct}_1, \ldots, \mathsf{ct}_n), P, \delta, \beta),\ b \in \{0,1\},$$

and where addition of y_0 and y_1 is carried out modulo β.

- **Security:** For $b = 0, 1$, the distribution ensembles $C_b(\lambda, 0)$ and $C_b(\lambda, 1)$ are computationally indistinguishable, where $C_b(\lambda, x)$ is obtained by sampling $(\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \leftarrow \mathsf{Gen}(1^\lambda)$, sampling $\mathsf{ct}_x \leftarrow \mathsf{Enc}(\mathsf{pk}, x)$, and outputting $(\mathsf{pk}, \mathsf{ek}_b, \mathsf{ct}_x)$.

We implicitly assume each execution of Eval to take an additional nonce input, which enables different invocations to have (pseudo)-independent failure probabilities. (See [11] for discussion.)

Remark 2.2 (Variant HSS Notions). *Within applications, we additionally consider the following HSS variants:*

(1) *Secret-Key HSS: a weaker notion where the role of the public key* pk *is replaced by a* secret *key* sk*, and where security requires indistinguishability of* $(\mathsf{ek}_b, \mathsf{Enc}(\mathsf{sk}, x_1) \ldots \mathsf{Enc}(\mathsf{sk}, x_n))$ *from* $(\mathsf{ek}_b, \mathsf{Enc}(\mathsf{sk}, x'_1) \ldots \mathsf{Enc}(\mathsf{sk}, x'_n))$ *for any pair of inputs* $x = (x_1, \ldots, x_n)$ *and* $x' = (x'_1, \ldots, x'_n)$*. Here we also allow* Enc *to produce a pair of shares of* x*, where each share is sent to one of the parties. This variant provides better efficiency when all inputs originate from a single client.*

(2) *Non-binary values: in some applications it is useful to evaluate programs with non-binary inputs and outputs, typically integers from a bounded range* $[0..M]$ *or* $[-M..M]$*. The above definition can be easily modified to capture this case.*

2.2 Computational Models

The main HSS scheme we optimize and implement naturally applies to programs P in a computational model known as *Restricted Multiplication Straight-line (RMS)* program [11, 20].

Definition 2.3 (RMS programs). An RMS program consists of a magnitude bound 1^M and an arbitrary sequence of the four following instructions, sorted according to a unique identifier id:

- Load an input into memory: $(\mathsf{id}, \hat{y}_j \leftarrow \hat{x}_i)$.
- Add values in memory: $(\mathsf{id}, \hat{y}_k \leftarrow \hat{y}_i + \hat{y}_j)$.
- Multiply memory value by input: $(\mathsf{id}, \hat{y}_k \leftarrow \hat{x}_i \cdot \hat{y}_j)$.
- Output from memory, as $\mathbb{Z}_\beta$ element: $(\mathsf{id}, \beta, \hat{O}_j \leftarrow \hat{y}_i)$.

If at any step of execution the size of a memory value exceeds the bound M, the output of the program on the corresponding input is defined to be $\perp$. Otherwise the output is the sequence of $\hat{O}_j$ values modulo β, sorted by id. We define the *size* (resp., *multiplicative size*) of an RMS program P as the number of instructions (resp., multiplication and load input instructions).

RMS programs with $M = 2$ are powerful enough to efficiently simulate boolean formulas, logarithmic-depth boolean circuits, and deterministic branching programs (capturing logarithmic-space computations) [11]. For concrete efficiency purposes, their ability to perform arithmetic computations on larger inputs can also be useful. We present an optimized simulation of formulas and branching programs by RMS programs in Section 4.5.

3 OVERVIEW OF GROUP-BASED HSS

In this section we give a simplified overview of the HSS construction from [11]. For efficiency reasons, we assume circular security of ElGamal encryption with a 160-bit secret key. This assumption can be replaced by standard DDH, but at a significant concrete cost.

3.1 Encoding $\mathbb{Z}_q$ Elements

Let $\mathbb{H}$ be a prime order group, with a subgroup $\mathbb{G}$ of prime order q. Let g denote a generator of $\mathbb{G}$. For any $x \in \mathbb{Z}_q$, we consider the following 3 types of two-party encodings:

Level 1: "Encryption." For $x \in \mathbb{Z}_q$, we let $[x]$ denote g^x, and $[\![x]\!]_c$ denote $([r], [r \cdot c + x])$ for a uniformly random $r \in \mathbb{Z}_q$, which corresponds to an ElGamal encryption of x with a secret key $c \in \mathbb{Z}_q$. (With short-exponent ElGamal, c is a 160-bit integer.) We assume that c is represented in base B ($B = 2$ by default) as a sequence of digits $(c_i)_{1 \le i \le s}$ (where $s = \lceil 160/\log_2 B \rceil$). We let $[\![\![x]\!]\!]_c$ denote $([\![x]\!]_c, ([\![x \cdot c_i]\!]_c)_{1 \le i \le s})$. All level-1 encodings are known to both parties.

Level 2: "Additive shares." Let $\langle x \rangle$ denote a pair of shares $x_0, x_1 \in \mathbb{Z}_q$ such that $x_0 = x_1 + x$, where each share is held by a different party. We let $\langle\!\langle x \rangle\!\rangle_c$ denote $(\langle x \rangle, \langle x \cdot c \rangle) \in (\mathbb{Z}_q^2)^2$, namely each party holds one share of $\langle x \rangle$ and one share of $\langle x \cdot c \rangle$. Note that both types of encodings are additively homomorphic over $\mathbb{Z}_q$, namely given encodings of x and x' the parties can locally compute a valid encoding of $x + x'$.

Level 3: "Multiplicative shares." Let $\{x\}$ denote a pair of shares $x_0, x_1 \in \mathbb{G}$ such that the difference between their discrete logarithms is x. That is, $x_0 = x_1 \cdot g^x$.

3.2 Operations on Encodings

We manipulate the above encodings via the following two types of operations, performed locally by the two parties:

(1) $\mathsf{Pair}([\![x]\!]_c, \langle\!\langle y \rangle\!\rangle_c) \mapsto \{xy\}$. This pairing operation exploits the fact that $[a]$ and $\langle b \rangle$ can be locally converted to $\{ab\}$ via exponentiation.

(2) $\mathsf{Convert}(\{z\}, \delta) \mapsto \langle z \rangle$, with failure bound δ. The implementation of Convert is also given an upper bound M on the "payload" z ($M = 1$ by default), and its expected running time grows linearly with M/δ. We omit M from the following notation.

The Convert algorithm works as follows. Each party, on input $h \in \mathbb{G}$, outputs the minimal integer $i \ge 0$ such that $h \cdot g^i$ is "distinguished," where roughly a δ-fraction of the group elements are distinguished. Distinguished elements were picked in [11] by applying a pseudo-random function to the description of the group element. An optimized conversion procedure from [13] (using special "conversion-friendly" choices of $\mathbb{G} \subset \mathbb{Z}_p^*$ and $g = 2$) applies the heuristic of defining a group element to be distinguished if its bit-representation starts with $d \approx \log_2(M/\delta)$ leading 0's. Note that this heuristic only affects the running time and not security, and thus it can be validated empirically. Correctness of Convert holds if no group element *between* the two shares $\{z\} \in \mathbb{G}^2$ is distinguished. Finally, Convert signals that there is a potential failure if there is a distinguished point in the "danger zone." Namely, Party $b = 0$ (resp., $b = 1$) raises a potential error flag if $h \cdot g^{-i}$ (resp., $h \cdot g^{i-1}$) is distinguished for some $i = 1, \ldots, M$. Note that we used the notation M both for the payload upper bound in Convert and for the bound on the memory values in the definition of RMS programs (Definition 2.3). In the default case of RMS program evaluation using base 2 for the secret key c in level 1 encodings, both values are indeed the same; however, when using larger basis, they will differ. To

avoid confusion, in the following we will denote M_{RMS} the bound on the memory values, and M the bound on the payload.

Let PairConv be an algorithm that sequentially executes these two operations: $\mathsf{PairConv}(\llbracket x \rrbracket_c, \langle\!\langle y \rangle\!\rangle_c, \delta) \mapsto \langle xy \rangle$, with error δ. We denote by Mult the following algorithm:

- **Functionality:** $\mathsf{Mult}(\llbracket\!\llbracket x \rrbracket\!\rrbracket_c, \langle\!\langle y \rangle\!\rangle_c, \delta) \mapsto \langle\!\langle xy \rangle\!\rangle_c$
  - Parse $\llbracket\!\llbracket x \rrbracket\!\rrbracket_c$ as $(\llbracket x \rrbracket_c, (\llbracket x \cdot c_i \rrbracket_c)_{1 \le i \le s})$.
  - Let $\langle xy \rangle \leftarrow \mathsf{PairConv}(\llbracket x \rrbracket, \langle\!\langle y \rangle\!\rangle_c, \delta')$ for $\delta' = \delta/(s+1)$.
  - For $i = 1$ to s, let $\langle xy \cdot c_i \rangle \leftarrow \mathsf{PairConv}(\llbracket x c_i \rrbracket_c, \langle\!\langle y \rangle\!\rangle_c, \delta')$.
  - Let $\langle xy \cdot c \rangle = \sum_{i=1}^{s} B^{i-1} \langle xy \cdot c_i \rangle$.
  - Return $(\langle xy \rangle, \langle xy \cdot c \rangle)$.

3.3 HSS for RMS programs

Given the above operations, an HSS for RMS programs is obtained as follows.

- Key generation: $\mathsf{Gen}(1^\lambda)$ picks a group $\mathbb{G}$ of order q with λ bits of security, generator g, and secret ElGamal key $c \in \mathbb{Z}_q$. It output $\mathsf{pk} = (\mathbb{G}, g, h, \llbracket c_i \rrbracket_c)_{1 \le i \le s}$, where $h = g^c$, and $(\mathsf{ek}_0, \mathsf{ek}_1) \leftarrow \langle c \rangle$, a random additive sharing of c.
- Encryption: $\mathsf{Enc}(\mathsf{pk}, x)$ uses the homomorphism of ElGamal to compute and output $\llbracket\!\llbracket x \rrbracket\!\rrbracket_c$.
- RMS program evaluation: For an RMS program P of multiplicative size S, the algorithm $\mathsf{Eval}(b, \mathsf{ek}_b, (\mathsf{ct}_1, \ldots, \mathsf{ct}_n), P, \delta, \beta)$ processes the instructions of P, sorted according to id, as follows. We describe the algorithm for both parties b jointly, maintaining the invariant that whenever a memory variable $\hat{y}$ is assigned a value y, the parties hold level-2 shares $Y = \langle\!\langle y \rangle\!\rangle_c$.
  - $\hat{y}_j \leftarrow \hat{x}_i$: Let $Y_j \leftarrow \mathsf{Mult}(\llbracket\!\llbracket x_i \rrbracket\!\rrbracket_c, \langle\!\langle 1 \rangle\!\rangle_c, \delta/S)$, where $\langle\!\langle 1 \rangle\!\rangle_c$ is locally computed from $(\mathsf{ek}_0, \mathsf{ek}_1)$ using $\langle 1 \rangle = (1, 0)$.
  - $\hat{y}_k \leftarrow \hat{y}_i + \hat{y}_j$: Let $Y_k \leftarrow Y_i + Y_j$.
  - $\hat{y}_k \leftarrow \hat{x}_i \cdot \hat{y}_j$: Let $Y_k \leftarrow \mathsf{Mult}(\llbracket\!\llbracket x_i \rrbracket\!\rrbracket_c, Y_j, \delta/S)$.
  - $(\beta, \hat{O}_j \leftarrow \hat{y}_i)$: Parse Y_i as $(\langle y_i \rangle, \langle y_i \cdot c \rangle)$ and output $O_j = \langle y_i \rangle + (r, r) \mod \beta$ for a fresh (pseudo-)random $r \in \mathbb{Z}_q$.

  The confidence flag is $\perp$ if any of the invocations of Convert raises a potential error flag, otherwise it is $\top$.

The pseudorandomness required for generating the outputs and for Convert is obtained by using a common pseudorandom function key that is (implicitly) given as part of each ek_b, and using a unique nonce as an input to ensure that different invocations of Eval are indistinguishable from being independent.

The secret-key HSS variant is simpler in two ways. First, Enc can directly generate $\llbracket\!\llbracket x \rrbracket\!\rrbracket_c$ from the secret key c. More significantly, an input loading instruction $\hat{y}_j \leftarrow \hat{x}_i$ can be processed directly, without invoking Mult, by letting Enc compute $Y_j \leftarrow \langle\!\langle x_i \rangle\!\rangle_c$ and distribute Y_j as shares to the two parties. Note that in this variant, unlike our main public key variant, the size of the *secret* information distributed to each party grows with the input size.

Performance. The cost of each RMS multiplication or input loading is dominated by $s + 1$ invocations of PairConv, where each invocation consists of Pair and Convert. The cost of Pair is dominated by one group exponentiation with roughly 200-bit exponent. (The basis of the exponent depends only on the key and the input, which allows for optimized fixed-basis exponentiations when the same input is involved in many RMS multiplications.) When the RMS multiplications apply to 0/1 values (this is the case when evaluating branching programs), the cost of Convert is linear in BS/δ, where the B factor comes from the fact that the payload z of Convert is bounded by the size of the basis. When δ is sufficiently small, the overall cost is dominated by the $O(BS^2 s/\delta)$ conversion steps, where each step consists of multiplying by g and testing whether the result is a distinguished group element.

4 OPTIMIZATIONS

4.1 Distributed Protocols

In this section, we suggest new protocols to improve the key generation, and to distributively generate level 2 shares of inputs under a shared key. The former protocol allows to save a factor two compared to the solution outlined in [13], while the latter is extremely useful for computation of degree-two polynomials (intuitively, this allows to avoid encoding each input with a number of group elements proportional to the size of the secret key – see e.g. Section 5.3.1).

4.1.1 Distributed Key Generation. When using HSS within secure computation applications, the parties must generate an HSS public key in a secure distributed fashion. Applying general-purpose secure computation to do so has poor concrete efficiency and requires non-black-box access to the underlying group. A targeted group-based key generation protocol was given in [13], where each party samples an independent ElGamal key, and the system key is generated homomorphically in a threshold ElGamal fashion. However, a negative side-effect of this procedure is that encryptions of key bits from different parties combine to encrypted values in $\{0, 1, 2\}$ instead of $\{0, 1\}$ (since homomorphism is over $\mathbb{Z}_q$, not $\mathbb{Z}_2$), and these larger payloads incur a factor of 2 greater runtime in homomorphic multiplications to maintain the same failure probability.

We present an alternative distributed key generation procedure which avoids this factor of 2 overhead, while maintaining black-box use of the group, at the expense of slightly greater (one-time) setup computation and communication. We focus here on the primary challenge of generating encryptions of the bits of a shared ElGamal secret key c. We use a binary basis for concreteness, but the protocol can be easily generalized to an arbitrary basis. Roughly the idea is to run an underlying (standard) secure protocol to sample exponents of the desired ElGamal ciphertext group elements, but which reveals the exponents *masked* by a random value (a_i or b_i) generated by the other party. The parties then exchange g^{a_i} and g^{b_i}, which enables each to locally reconstruct the ElGamal ciphertext, while computationally hiding the final exponents. Most importantly, the resulting protocol requires only black-box operations in the group.

Proposition 4.1. *The protocol Π_{Gen} in Figure 1 securely evaluates the group-based HSS* Gen *algorithm (from Section 3.3).*

Proof Sketch. By construction (and correctness of the underlying 2PC), both parties will correctly output ElGamal ciphertexts $(g^{r_i}, g^{x_i})_{i \in [s]}$ of each bit c_i of the generated secret key, as desired.

HSS Distributed Key Generation Π_{Gen}

(1) For each $i \in [s]$:
A samples $a_i, a'_i \xleftarrow{\$} \mathbb{Z}_q$, sends $g^{a_i}, g^{a'_i}$ to B; B samples $b_i, b'_i \xleftarrow{\$} \mathbb{Z}_q$, sends $g^{b_i}, g^{b'_i}$ to A.
(2) Execute secure 2PC for (randomized) functionality:
Input: A inputs $(a_i, a'_i)_{i\in[s]}$. B inputs $(b_i, b'_i)_{i\in[s]}$.
Compute:
Sample s random key bits: $\forall i \in [s],\ c_i \leftarrow \{0,1\}$.
Let $c = \sum_{i=1}^{s} 2^{i-1} c_i \in \mathbb{Z}_q$.
For each $i \in [s]$:
(a) Sample encryption randomness $r_i \leftarrow \mathbb{Z}_q$.
(b) Compute $x_i = r_i c + c_i \in \mathbb{Z}_q$.
Output: A receives $(r_i - b_i, x_i - b'_i)_{i\in[s]}$; B receives $(r_i - a_i, x_i - a'_i)_{i\in[s]}$.
(3) A outputs $\left((g^{b_i})g^{(r_i-b_i)}, (g^{b'_i})g^{(x_i-b'_i)}\right)_{i\in[s]}$, B outputs $\left((g^{a_i})g^{(r_i-a_i)}, (g^{a'_i})g^{(x_i-a'_i)}\right)_{i\in[s]}$.

Figure 1: 2-party protocol Π_{Gen} for distributed HSS public key generation.

Regarding security, the view of each party consists of a collection of random group elements (received from the other party) together with the exponent offsets from each value and its target. This can be directly simulated given freshly sampled target ciphertexts, by choosing a random offset and computing the group elements in the first step accordingly. □

Observe that it is immediate to modify the protocol Π_{Gen} to additionally output additive shares (c_A, c_B) of the secret key c.

Comparison to [13]. Π_{Gen} requires the additional 2PC execution and $2s$ additional exponentiations per party (from Step 3) over the [13] solution. The 2PC is composed of s linear operations over $\mathbb{Z}_2$, and s multiplications and $2s$ additions over $\mathbb{Z}_q$. In exchange, Π_{Gen} guarantees the encrypted system key bits c_i remain in $\{0,1\}$, whereas in [13] the corresponding terms c_i will take values in $\{0,1,2\}$, yielding x2 speedup in homomorphic evaluation of RMS multiplications.

We remark that while one may be able to effectively address this larger payload in specific cases (e.g., leveraging that the value $c_i \in \{0,1,2\}$ is 1 with probability 1/2), such fixes will not extend to general usage settings, or when implementing further HSS optimizations, such as using a larger basis for the key.

4.1.2 Distributed Generation of Level 2 Shares. In this section, we present a simple distributed protocol for generation of level 2 shares (additive shares) of a secret input under a shared key. Namely, we consider two parties, A and B, holding additive shares c_A and c_B of the secret key c (where addition is in $\mathbb{Z}_q$). We assume that each share statistically masks c over the integers (for 80 bits of security, each share can be 240 bits long, instead of requiring 1536 bits to describe a truly random element in $\mathbb{Z}_q$). The protocol is represented in Figure 2; it assumes access to an oblivious transfer primitive.

Distributed Level 2 Shares Generation Π_{L2S}

(1) A's input is (x, c_A), with $x \in \{0,1\}$, and B's input is c_B, such that $c_A + c_B = c$. Let t be the bitlength of c_A and c_B.
(2) B picks $r \xleftarrow{\$} \mathbb{Z}_{2^{t+\lambda}}$ and runs as sender in an oblivious transfer protocol with input $(r, r + c_B)$. A runs as receiver with selection bit x and get an output r'.
(3) A outputs $(x, r' + x \cdot c_A)$. B outputs $(0, -r)$.

Figure 2: 2-party protocol Π_{L2S} for distributed level 2 shares generation.

Proposition 4.2. *The protocol Π_{L2S} in Figure 2 securely generates level 2 shares $\langle\!\langle x \rangle\!\rangle_c$ of the input x.*

Correctness follows easily by inspection, and the (statistical) privacy of the inputs directly reduces to the security properties of the oblivious transfer (OT).

For each input bit x encoded in this fashion, the required communication corresponds to a single 1-out-of-2 string OT, with string length $\ell = 240$ bits and security parameter $\lambda = 80$. Leveraging OT extension techniques, n input bits can then be encoded with $2n(\lambda + \ell) = 640n$ bits of communication.

4.2 Generic Ciphertext Compression for Public-Key HSS

In [11], a heuristic method to compress the ciphertext size by a factor two was suggested, by generating all first components g^r of ciphertexts using a PRG; however, this method only applies to secret-key HSS. In this section, we outline a method to achieve a comparable trade-off (ciphertexts size reduced by a factor two in exchange for a larger key) for public-key HSS, under a new assumption that we introduce below.

Entropic Span Diffie-Hellman Assumption over $\mathbb{Z}_q$. Let $\bullet$ denote the inner product operation, and let B denote any basis. For any integer t and any vector $v \in Z_q^t$, we let $X_{\lambda,t,v}$ denote the distribution ensemble

$$X_{\lambda,t,v} = \{x \in \mathbb{Z}_q \mid c \xleftarrow{\$} \{0,\cdots,B-1\}^t, x \leftarrow v \bullet c\}$$

The entropic span Diffie-Hellman assumption (ESDH) states that

Assumption 4.3. *For any integers $t = t(\lambda), k = k(\lambda)$, we let $(v_1,\cdots,v_k) \in (Z_q^t)^k$ be k vectors of size t such that for any non-trivial linear combination v of $(v_1,\cdots,v_k)$, it holds that $H_\infty(X_{\lambda,t,v}) \geq \omega(\log\lambda)$, where H_∞ denotes the min-entropy (when such vectors exist). Then the following distributions are indistinguishable:*

$$D_0 = \{v_1,\cdots,v_k, g, g^{v_1\bullet c},\cdots,g^{v_k\bullet c} \mid c \leftarrow \{0,\cdots,B-1\}^t\}$$

$$D_1 = \{v_1,\cdots,v_k, g, g_1,\cdots,g_k \mid (g_1,\cdots,g_k) \xleftarrow{\$} \mathbb{G}^k\}$$

Note that a necessary condition for this assumption to hold is that all non-zero vectors in the span of $(v_1,\cdots,v_k)$ must have $\omega(\log\lambda)$ exponentially large non-zero entries. If s denotes the length of a standard ElGamal secret key (e.g. using base 2, $s = 160$ for 80 bits of security), natural parameters for the ESDH assumption are $t \approx s + \sqrt{s}$, $\lambda = s$, and $k \approx \sqrt{s}$, and each component of each vector is s-bit

long: with overwhelming probability, the vector with the smallest Hamming weight in the span of random vectors $(v_1, \cdots, v_k)$ has s large coefficients.

Lemma 4.4. *(Generic Security of* ESDH*) The entropic-span Diffie-Hellman assumption holds in the generic group model.*

A proof of Lemma 4.4 is given in the full version [9].

Randomness Reuse under ESDH**.** Under the above assumption, we get the following lemma:

Lemma 4.5. *Let $\mathbb{G}$ be a group and (t, k) be two integers such that the ElGamal encryption scheme is circularly secure, and the* ESDH *assumption with parameters (t, k) holds over $\mathbb{G}$. Then there exists a correct and secure 2-party public-key Las Vegas homomorphic secret sharing scheme with the following parameters:*

- *The public key* pk *consists of $k+1$ elements of $\mathbb{G}$, and a short prg seed*
- *The ciphertexts are of length $t + \lceil t/k \rceil + 1$ group elements*

Sketch. The HSS scheme is constructed as previously, with the following modifications: the secret key is a vector $c = (c_i)_{i \le t}$. The public key now contains k vectors $(v_1, \cdots, v_k) \in (\mathbb{Z}_q^t)^k$ (which can be compressed using a pseudorandom generator) and group elements $(h_1, \cdots, h_k) \leftarrow (g^{v_1 \bullet c}, \cdots, g^{v_k \bullet c})$. Encryption is done with the standard randomness reuse method, using a single random coin and the k public keys to encrypt k consecutive values of $(x, (x \cdot c_i)_{i \le t})$. We modify level 2 shares to be of the form $(\langle y \rangle, (\langle c_i \cdot y \rangle)_{i \le t})$ (which simply means that the reconstruction with powers of 2 is not executed at the end of the Mult algorithm). To evaluate the pairing algorithm Pair on an input $([\![x]\!]_c, \langle\!\langle y \rangle\!\rangle_c)$, the parties compute $\langle v_j \bullet c \rangle_{j \le q}$ and use the jth share to decrypt components of level 1 shares encrypted with the key h_j. Using the natural parameters previously mentioned, this optimization reduces the ciphertext size from $2s+1$ group elements to $s + 2\lceil\sqrt{s}\rceil + 1$ group elements. For $s = 160$, this corresponds to a reduction from 321 to 187 group elements, whereas for $s = 40$ (obtained by using a base-16 representation) this corresponds to a reduction from 81 to 55 group elements. □

4.3 Optimizing Share Conversion

In [13], the share conversion algorithm Convert (see Section 3.2) was heuristically improved by changing the way in which distinguished group elements are defined. Instead of independently deciding whether a group element is distinguished by applying a PRF to its description, as originally proposed in [11], the method proposed in [13] considers the sequence stream of most significant bits of the group elements $h, hg, hg^2, hg^3, \ldots$, where h is the given starting point, and looks for the first occurrence of the pattern 0^d in stream.

The advantage of this approach is that stream can be computed very efficiently for a suitable choice of "conversion-friendly" group. Concretely, the groups proposed in [13] are of the form $\mathbb{G} \subseteq \mathbb{Z}_p^*$, where p is close to a power of 2 and $g = 2$ is a generator. Combined with an efficient implementation of searching for the pattern 0^d in stream, a single conversion step can be implemented at an amortized cost of less than one *machine word* operation per step. This provides several orders of magnitude improvement over a generic implementation of the original conversion algorithm from [11], which requires a full group multiplication and PRF evaluation per step.

In this section, we describe two simple modifications that allow us to further improve over this method. In the context of RMS multiplications, the improvement is by at least a factor of 16.

4.3.1 Separating Distinguished Points. The first optimization ensures that an actual failure happens in the computation if and only if the two parties raise a flag. This is done simply by declaring any point in the danger zone (which corresponds to M points forward for the first party, and M points backward for the second party, where M is the payload bound) to be non-distinguished if it is located less than $2M$ steps after a distinguished point. This modification has only a marginal impact on the running time as it only affects the start of the Convert algorithm, where the parties search for distinguished points in the danger zone. Before starting the conversion, we also let both parties multiply their local share by g^M (this avoids having to compute inversions when looking for distinguished points backward). This is to be compared with [13], where roughly half of the distinguished points are immediately followed by another distinguished point (this happens if the bit following the 0^d pattern is 0). Hence, the event of two parties raising a flag was highly correlated with the event of the first party raising a flag, even when the actual payload is 0 (which corresponds to a case where no actual failure can occur).

4.3.2 Changing the Pattern. We suggest a second, seemingly trivial, modification of the Convert algorithm: searching for the pattern 10^d instead of 0^d. We explain below how this improves the construction.

First, recall that the conversion algorithm proceeds by looking for the first distinguished point in a sequence stream defined by the most significant bits of the group elements $h, hg, hg^2, \ldots$. Searching for the modified pattern is almost the same: as before, we search for the first occurrence of 0^d in the sequence; when this sub-sequence is found, it necessarily holds that the bit that precedes it is 1. The only actual change is in the initial check, which ignores an initial sequence of 0's and searches the danger zone for the pattern 10^m (instead of 0^m) when deciding whether to raise a potential error flag. Changing the pattern 0^d to 10^d improves the failure probability by a factor of 2 (since it reduces the probability of a distinguished point in the danger zone by a factor of 2) without significantly changing the running time. Thus, it effectively reduces the expected running time required for achieving a given failure probability by a factor of 2.

We now formally describe and analyze the optimized conversion algorithm that incorporates the above two modifications.

$\mathsf{Convert}^*(\{z\}, M, d) \mapsto \langle z \rangle$. Let $\mathsf{Convert}^*$ denote the Convert algorithm from [13] (see Section 3.2) modified as follows: given a payload bound M and failure parameter d, the algorithm searches for the pattern 10^d instead of 0^d, and points in the danger zone within $2M$ steps backward of a distinguished point are considered to be non-distinguished.

Referring by "failure" to the event of *both parties* raising a potential failure flag, we can therefore state the following lemma, which

corresponds to a factor-$(2M/z)$ improvement over the conversion algorithm of [13] for a payload z and payload bound M:

Lemma 4.6. *If* Convert* *is run on a random* stream *with payload z, payload bound M, and failure parameter d, the expected number of steps performed by each party is $T \leq 2^{d+1} + 2M$ and the failure probability is $\varepsilon \leq z \cdot 2^{-(d+1)}$.*

A proof of Lemma 4.6 is given in the full version [9]. For comparison, in the Las Vegas variant of the optimized conversion algorithm from [13], the expected running time is the same, whereas the failure probability bound is $\varepsilon \leq M \cdot 2^{-d}$.

Note that our heuristic assumption that stream is uniformly random has no impact on security, it only affects efficiency and has been empirically validated by our implementation. Given Lemma 4.6, and denoting Mult* the Mult algorithm using Convert* instead of Convert, we can now bound the failure probability in an RMS multiplication:

Lemma 4.7. *If* Mult* *is run with base B, length s for the secret key c, payload bound M, and outputs y, the expected number of conversion steps performed by each party is $T \leq (s+1) \cdot 2^{d+1}$, the failure probability ε, expected over the randomness of the secret key c, satisfies*

$$\varepsilon \leq y \cdot \frac{1 + \frac{s(B-1)}{2}}{2^{d+1}} + \left(\frac{s+1}{2^{d+1}}\right)^2.$$

A proof of Lemma 4.7 is given in the full version [9]. Note that the payload in the first Convert* algorithm is y and the average payload in the s last Convert* invocations is $(B-1)y/2$; the failure probability is also taken over the random choice of the secret key.

4.3.3 Randomizing the Conversion of Bit Inputs. Using the above method, the two parties raise a flag if a failure actually occurs or if both parties raise a flag in different executions of Convert*; the latter situation occurs only with quadratically smaller probability $((s+1)/2^{d+1})^2$. In addition, let z be a shared value used in a conversion step with failure parameter δ. Observe that the actual probability of a failure occurring is δz. In [11], the failure probability was analyzed by using a bound on the maximal size of the shared value. A typical conversion occurs after a pairing between an encryption of a value $x \cdot c_i$, where x is an input and c_i is a component of the secret key (in some base B), and a level 2 share of a value y; in most applications, x and y are bits (this corresponds to using memory bound $M_{\mathsf{RMS}} = 1$ for the RMS program), hence the maximum value of xyc_i is $B-1$. As the secret key is random, we previously observed that the average size of c_i is $(B-1)/2$.

In addition, we will show in this section that we can randomize the conversion algorithm, so as to ensure that each of x and y is equal to 0 with probability $1/2$. This ensures that the average size of $z = xyc_i$ in a typical conversion step is $(B-1)/8$, hence that the event of a failure occurring is on average $\delta(B-1)/8$, independently of the actual distribution of the inputs. Because of our previous optimization, which ensures that a failure occurs if and only if two flags are raised, this allows to set the parameter δ to be 8 times bigger to reach a fixed failure probability, effectively reducing the number of steps in a conversion algorithm by a factor of 8. Therefore, cumulated with the previous optimization, this improves the computational efficiency of conversions in most applications by a factor 16.

We now describe our randomization technique. First, we modify the key generation algorithm as follows: we set the evaluation keys $(\mathsf{ek}_0, \mathsf{ek}_1)$ to be $(\langle c_i \rangle)_{i \leq s}$ (the parties hold shares of each bit of c over the integers, rather than holding integer shares of c). Second, we assume that the parties have access to a stream of common random bits (which can be heuristically produced by a PRG), and that they hold level 2 shares of each input bit. In the case of secret key HSS, these level two shares can be part of the encryption algorithm of the HSS; for public key HSS, they can be computed (with some failure probability) from level 1 shares and the shares of the secret key. Let PairConv* be the PairConv algorithm modified to use the new Convert* algorithm.

Functionality: $\mathsf{RandMult}(\mathsf{pk}, [\![[x]]\!]_c, \langle\!\langle x \rangle\!\rangle_c, \langle\!\langle y \rangle\!\rangle_c, \delta, b_0, b_1) \mapsto \langle\!\langle xy \rangle\!\rangle_c$

Description: Parse $[\![[x]]\!]_c$ as $([\![x]\!]_c, ([\![xc_i]\!]_c)_{i \leq s})$, and use the public values (b_0, b_1) to compute

$$[\![b_0 \oplus x]\!]_c, ([\![(b_0 \oplus x)c_i]\!]_c)_{i \leq s}, \langle\!\langle b_1 \oplus y \rangle\!\rangle_c.$$

Let $c_0 = 1$. For $i = 0$ to s, call

$$\mathsf{PairConv}^*([\![(b_0 \oplus x)c_i]\!]_c, \langle\!\langle b_1 \oplus y \rangle\!\rangle_c, \delta),$$

which returns $\langle (b_0 \oplus x)(b_1 \oplus y)c_i \rangle$. Compute

$$\langle xyc_i \rangle \leftarrow (-1)^{b_0+b_1}(\langle (b_0 \oplus x)(b_1 \oplus y)c_i \rangle - b_0 b_1 \langle c_i \rangle - b_0(-1)^{b_1} \langle yc_i \rangle - b_1(-1)^{b_0} \langle xc_i \rangle).$$

Reconstruct $\langle\!\langle xy \rangle\!\rangle_c \leftarrow (\langle xyc_0 \rangle, \sum_i 2^{i-1} \langle xyc_i \rangle)$.

The correctness immediately follows from the fact that $b_0 \oplus x$ and $b_1 \oplus y$ are uniform over $\{0,1\}$ if (b_0, b_1) are random bits. Therefore, we get the following corollary to Lemma 4.7:

Corollary 4.8. *The (Las Vegas) probability ε of a failure event occurring in an RMS multiplication on bit inputs using base B and length s for the secret key is*

$$\varepsilon \leq \frac{1 + \frac{s(B-1)}{2}}{2^{d+3}} + \left(\frac{s+1}{2^{d+1}}\right)^2.$$

Remark 4.9. *The above method does not generalize immediately to $M_{\mathsf{RMS}} > 1$: while xoring with a public value can be done homomorphically in the case $M_{\mathsf{RMS}} = 1$, this does not extend to general modular addition. However, a weaker version of the result can be achieved, using $(r_0, r_1) \xleftarrow{\$} \{0, \cdots, M_{\mathsf{RMS}} - 1\}^2$ and randomizing (x, y) as $(x', y') = (x - r_0, y - r_1)$. While (x', y') are not uniformly distributed and belong to a larger set $\{1 - M_{\mathsf{RMS}}, \cdots, M_{\mathsf{RMS}} - 1\}$, we can lower bound the probability of $x'y' = 0$ as*

$$\Pr[x'y' = 0] \geq 1 - \left(\frac{M_{\mathsf{RMS}} - 1}{M_{\mathsf{RMS}}}\right)^2,$$

which is sufficient to improve over the basic Convert *algorithm.*

4.4 Dealing with the Leakage

A crucial issue with current group-based HSS schemes is that the failure event depends on secret information, which may depend both on the inputs and the secret key. Therefore, in scenarios where the computing parties get to know whether the computation failed, the secrecy of the inputs and the key can be compromised. The

amount of private information that leaks during a computation is directly proportional to the failure probability δ. We discuss methods to mitigate the leakage in this section.

4.4.1 Leakage-Absorbing Pads. In this section, we introduce a technique to reduce the dependency between the failure probability and the amount of leakage, from linear to quadratic. Note that a technique was also proposed in [13] to deal with the leakage, by compiling the RMS program into a leakage-resilient circuit. This technique is incomparable to our new technique: it allows reduce the leakage on the input by any desired amount, but it gives no security guarantee for the secret key and it comes at the cost of a large computational overhead, while the current technique we develop allows only to square the leakage probability, but it does protect both the input and the secret key with essentially no additional computation.

Masked Pairing Algorithm. To handle the leakage more efficiently, we introduce a masked pairing algorithm, which takes in addition some level 2 share of a pseudorandom bit b, which we call leakage-absorbing pad, so that any value that can leak during a conversion is XOR-masked with b. This ensures that failures do not leak private information, unless two failure events occur on computation involving the same pad. In various scenarios, this allows us to make the amount of leakage quadratically smaller than the failure probability.

Functionality. $\mathsf{MPair}(\llbracket x \rrbracket_c, \langle\!\langle b \rangle\!\rangle_c, \langle\!\langle y \oplus b \rangle\!\rangle_c) \mapsto \langle xy \oplus b \rangle$
Description. Compute $\llbracket 1 - x \rrbracket_c$ from $\llbracket x \rrbracket_c$ homomorphically. Compute $\mathsf{Pair}(\llbracket x \rrbracket_c, \langle\!\langle b \rangle\!\rangle_c) \times \mathsf{Pair}(\llbracket 1 - x \rrbracket_c, \langle\!\langle y \oplus b \rangle\!\rangle_c)$ to get $\{x(y \oplus b)\} \times \{(1 - x)b\} = \{xy \oplus b\}$, and compute

$$\langle xy \oplus b \rangle = \mathsf{Convert}(\{xy \oplus b\}).$$

We extend this masked pairing algorithm to a masked multiplication algorithm, that returns $\langle\!\langle xy \oplus b \rangle\!\rangle_c$. However, the latter is more involved, as we must compute $\langle c(b \oplus xy) \rangle$ using only MPair to avoid non-masked leakage. In addition to pk, we assume that the parties hold shares $(\langle c_i \rangle)_{i \le s}$ of the coordinates of c.

Functionality. $\mathsf{MMult}_{\mathsf{pk}}(\llbracket\!\llbracket x \rrbracket\!\rrbracket_c, \langle\!\langle b \rangle\!\rangle_c, \langle\!\langle y \oplus b \rangle\!\rangle_c) \mapsto \langle\!\langle xy \oplus b \rangle\!\rangle_c$
Description. Compute for $i = 1$ to s

$$\langle b \oplus c_i \rangle \leftarrow \mathsf{MPair}(\llbracket c_i \rrbracket_c, \langle\!\langle b \rangle\!\rangle_c, \langle\!\langle 1 \oplus b \rangle\!\rangle_c)$$

This part correspond to a precomputation phase, which depends only on the pad b and can be reused in any execution of MMult with the same pad. Parse $\llbracket\!\llbracket x \rrbracket\!\rrbracket_c$ as

$$(\llbracket x \rrbracket_c, (\llbracket x \cdot c_i \rrbracket_c)_{i \le s})$$

and perform the following operations:

(1) $\langle b \oplus xy \rangle \leftarrow \mathsf{MPair}(\llbracket x \rrbracket_c, \langle\!\langle b \rangle\!\rangle_c, \langle\!\langle y \oplus b \rangle\!\rangle_c)$
(2) $\langle b \oplus c_i xy \rangle \leftarrow \mathsf{MPair}(\llbracket xc_i \rrbracket_c, \langle\!\langle b \rangle\!\rangle_c, \langle\!\langle y \oplus b \rangle\!\rangle_c)$ for $i = 1$ to s
(3) $2\langle c_i(b \oplus xy) \rangle \leftarrow 2 \cdot \langle b \oplus xyc_i \rangle + \langle c_i \rangle - (\langle b \rangle + \langle b \oplus c_i \rangle)$ for $i = 1$ to s
(4) $\langle c(b \oplus xy) \rangle \leftarrow \sum_{i=1}^{s} 2^{i-1} \langle c_i(b \oplus xy) \rangle$
(5) Return $(\langle b \oplus xy \rangle, \langle (b \oplus xy) \cdot c \rangle)$.

Masked Evaluation of an RMS Program. Let P be an RMS program with d inputs, which we assume to be a circuit with XOR gates and restricted AND gates. We denote by $\mathsf{MaskedEval}$ an algorithm that takes as input pk, a bit t, an evaluation key ek, a failure parameter δ, an RMS program P, a leakage-absorbing pad $\langle\!\langle b \rangle\!\rangle_c$, and d encoded inputs $(\llbracket\!\llbracket x_i \rrbracket\!\rrbracket_c)_{i \le d}$, which outputs a level-2 share of $P(x_1, \cdots, x_d)$:

$$\mathsf{MaskedEval}(t, \langle\!\langle b \rangle\!\rangle_c, (\llbracket\!\llbracket x_i \rrbracket\!\rrbracket_c)_{i \le d}, P, \delta) \mapsto \langle\!\langle P(x_1, \cdots, x_d) \rangle\!\rangle_c$$

The algorithm $\mathsf{MaskedEval}$ proceeds as follows: each masked monomial is computed using the MMult algorithm for each product of the monomial. To compute a masked XOR of two monomials M_1 and M_2,

(1) Compute $\langle\!\langle b \oplus M_1 \rangle\!\rangle_c$, $\langle\!\langle b \oplus M_2 \rangle\!\rangle_c$, and $\langle\!\langle b \oplus M_1 M_2 \rangle\!\rangle_c$ using several invocations of the MMult algorithm
(2) Compute $\langle\!\langle b \oplus (M_1 \oplus M_2) \rangle\!\rangle_c$ as

$$\langle\!\langle b \rangle\!\rangle_c + \langle\!\langle b \oplus M_1 \rangle\!\rangle_c + \langle\!\langle b \oplus M_2 \rangle\!\rangle_c - 2\langle\!\langle b \oplus M_1 M_2 \rangle\!\rangle_c .$$

Generating the Pads. In scenarios where secret-key HSS is sufficient, the leakage absorbing pads can simply be generated as part of any HSS ciphertext. For scenarios that require public-key HSS, a number of leakage-absorbing pads can be generated as part of the key distribution protocol, and re-generated later on if too many pads have been compromised. Generating a pad is relatively simple: it can be done using two oblivious transfers.

4.4.2 Sharing the Secret Key. A natural approach to protect the secret key is to share it into k shares by picking uniformly random $c^i \in \mathbb{Z}_q$ subject to $c = \sum_{i=1}^{k} c^i$, and modify the level 1 share of an input x to include encryptions of $x \cdot c_j^i$ for $i \le k$ and $j \le s$. This ensures that the jth component of c remains unknown unless the k components at the jth positions of the $(c_j^i)_{i \le k}$ are compromised. However, this increases the ciphertext size and the evaluation time by a factor k. In this section, we discuss more efficient sharing methods to protect the secret key, that offer comparable security at a cost which is only additive in k.

Computational Approach. The simplest method is to increase the size of the secret key, and to rely on entropic variants of the Diffie-Hellman assumption, stating that indistinguishability holds as long as the secret exponent has sufficient min-entropy (see [14, 16]). Assume for simplicity that the secret key is written in base 2; let s be the key length corresponding to the desired security level. Extending the key to be of size $s + k$ ensures, under an appropriate variant of the Diffie-Hellman assumption, that a leakage of up to k bits of the secret key does not compromise the security.

Information Theoretic Approach. The above method becomes inefficient if one wants to be able to handle a very large amount of leakage. We outline a better approach to protect the secret key c against an amount of leakage bounded by k. Let $\ell \leftarrow \lceil \log q \rceil + k + 2\lceil \log(1/\varepsilon) \rceil$, where ε denotes a bound on the statistical distance between the distribution of the secret key and the uniform distribution from the view of an adversary getting up to k bits of information. In the key setup, a large vector $(v_i)_{i \le \ell}$ of elements of $\mathbb{Z}_q$ is added to the public key (it can be compressed to a short seed using a PRG), as well as encryptions of random bits $(c'_i)_{i \le \ell}$ satisfying $\sum_i c'_i v_i = c \bmod q$. An HSS ciphertext for an input x now encrypt $(x, (xc'_i)_i)$. After an invocation of Convert with input y, $\langle yc \rangle$ can be reconstructed as $\sum_i v_i \langle yc'_i \rangle$. By the leftover hash lemma, an arbitrary leakage of up to k bits of information on the c'_i can be allowed, without compromising the key c. This method

improves over the previous one for large values of k; as a byproduct, it offers information-theoretic security and allows to handle leakage of arbitrary form, which simplifies the concrete security analysis of protocols that use it.

4.5 Extending and Optimizing RMS Programs

In this section, we describe optimizations that take advantage of the specificities of group-based HSS schemes when evaluating RMS programs, to allow for richer semantics and efficiency improvements for certain types of computation.

4.5.1 Terminal Multiplications. The Mult algorithm, which allows to multiply a level 1 share of x with a level 2 share of y and produces a level 2 share of xy, involves $s+1$ calls to PairConv: one to generate $\langle xy \rangle$, and s to generate $\langle xy \cdot c \rangle$. We make the following very simple observation: let us call *terminal multiplication* a multiplication between values that will not be involved in further multiplications afterward. Then for such multiplications, it is sufficient to call PairConv a single time, as the second part $\langle xy \cdot c \rangle$ of a level 2 share is only necessary to evaluate further multiplications. For low depth computation with a large number of outputs, this results in large savings (in particular, it reduces the amount of computation required to evaluate degree-two polynomials with some fixed failure probability by a factor $(s+1)^2$). Moreover, terminal multiplications have additional benefits that we outline below, which provides further motivation for treating them separately.

Short Ciphertexts for Evaluation of Degree-Two Poynomial with Secret-Key HSS. Unlike public-key HSS, a ciphertext in a secret-key HSS scheme can be directly generated together with a level 2 share of its plaintext. This implies that it is not necessary to "download" the inputs at all to reconstruct such level 2 shares. Therefore, when computing degree-two polynomials with secret-key HSS, which involves only terminal multiplications, it is not necessary anymore to encrypt the products between the bits of the secret key and the input: a single ElGamal encryption of the input is sufficient.

For public-key HSS, level 2 shares of secret inputs cannot be generated by a party directly, as no party knows the HSS secret key. However, if we are in a setting with two parties who hold shares of the secret key, then the parties can jointly generate level 2 shares of their input by the protocol described in Section 4.1.2.

Handling Large Inputs in Terminal Multiplications. In general, all inputs manipulated in RMS programs must be small, as the running time of conversion steps depend on the size of the inputs. However, the semantic of RMS programs can be extended to allow for a terminal multiplication where one of the inputs can be large, by outputting the result of the pairing operation without executing the final conversion step. This simple observation has interesting applications: it allows to design RMS programs in which a large secret key will be revealed if and only if some predicate is satisfied. More specifically, it allows to evaluate programs with outputs of the form $K^{F(x_1,\cdots,x_n)}$ where K is a large input, and $(x_1,\cdots,x_n)$ are short input: the key K will be revealed if and only if F evaluates to 1 on $(x_1,\cdots,x_n)$.

Reduced Failure Probability in Terminal Multiplications. Consider terminal multiplications in the evaluation of an RMS program where the output is computed modulo β. If a party detects a risk of failure, he must return a flag $\perp$. However, observe that such a failure occurs when the two parties end up on different distinguished points in a conversion step; but if the distance between the two possible distinguished points happens to be a multiple of β in a terminal multiplication, then the reduction modulo β of the result will cancel this failure. In this case, the party can simply ignore the risk of failure. For the most commonly used special case of computation modulo 2, this observation reduces the number of failures in terminal multiplication by a factor 2.

4.5.2 Evaluating Branching Programs and Formulas. As pointed out in [11], a branching program can be evaluated using two RMS multiplications for each node. A simple observation shows that in fact, a single RMS multiplication per node is sufficient. Each node N is computed as $x \cdot N_0 + y \cdot N_1$, where (N_0, N_1) are values on the two parent nodes, and (x, y) are multipliers on the edges (N_0, N) and (N_1, N). Observe that the two edges leaving N_0 carry the values x and $\bar{x}$, and that given $(N_0, x \cdot N_0)$, the value $\bar{x} \cdot N_0$ can be computed as $N_0 - x \cdot N_0$ at no cost. Therefore, the two RMS multiplications used to compute N can be reused in the computation of two other nodes, saving a factor two on average compared to the simulation of a branching program by an RMS program given in Claim A.2 of [11].

As boolean formulas can be efficiently simulated by branching programs, a fan-in-2 boolean formula with n internal AND and OR gates can be evaluated using exactly n RMS multiplication in the setting of secret-key HSS. In the setting of public-key HSS, where the encryption of the inputs must be converted to level 2 shares, and additional RMS multiplication per input is required. In both cases, NOT gates incur no additional cost.

4.5.3 Evaluating Threshold Formulas. Threshold functions (that return 1 if at least some number n of inputs, out of N, are equal to 1) are useful in many applications. An n-out-of-N threshold function can be evaluated using $(N-n+1)\cdot n$ non-terminal RMS multiplications, and 1 terminal RMS multiplication (for example, the majority function requires essentially $(N+1)^2/4 - 1$ RMS multiplications), using their natural branching program representation. Applying an n-out-of-N threshold function to the N outputs of N size-k boolean formulas requires $k(N-n+1)\cdot n$ non-terminal RMS multiplications. This class of functions captures a large number of interesting applications, such as evaluating policies on encrypted data, or searching for matches with encrypted queries.

5 APPLICATIONS

In this section, we outline a number of natural scenarios in which homomorphic secret sharing schemes can prove particularly useful, and describe solutions based on the HSS scheme of Boyle *et al.*, enhanced with the relevant optimizations described in the previous section. Efficiency estimations given in this section are based on the running time of our implementation, described Section 6, ran on a single thread of an Intel Core i7 CPU. Our implementation could perform roughly 5×10^9 conversion steps per second on average, and 6.4×10^5 modular multiplications per second, on a conversion-friendly group with a pseudo-Mersenne modulus $p = 2^{1536} - 11510609$, which is estimated to provide 80 bits of security for the HSS scheme. We summary in Table 1 the optimizations

of Section 4 that apply to each application described in this section. Some of the subsections of Section 4 refer to several distinct possible optimizations; a ✓ mark indicates that at least one of the optimizations apply to the application. Note also that leakage-absorbing pads (Section 4.4.1) and ciphertext compression (Section 4.2) cannot be used simultaneously; for applications where both optimizations possibly apply, only one of the two optimizations can be used in a given instantiation. Finally, for some applications, there are optimizations that are not relevant in general, but could be applied in some specific scenario; those optimizations are still marked with a ✗ for simplicity.

	MPC (5.1)	File System (5.2)	RSS Feed (5.2)	PIR (5.2)	Correl. (5.3)
Key Generation (4.1)	✓	✗	✗	✗	✓
Compression (4.2)	✓	✓	✗	✗	✗
Share Conversion (4.3)	✓	✓	✓	✓	✓
Rand. Conversion (4.3)	✓	✓	✓	✓	✓
Leakage (4.4)	✓	✓	✗	✓	✗
Terminal Mult. (4.5)	✓	✓	✓	✓	✓
Large Inputs (4.5)	✓	✓	✗	✗	✗

Table 1: Summary of the optimizations of Section 4 that apply to the applications of Section 5.

5.1 Secure MPC with Minimal Interaction

Suppose that a set of clients wish to outsource some simple MPC computation to two non-colluding servers, with the following very simple communication pattern: the clients encrypt their inputs, and send them to the servers, who perform local computation and send back a single message to some output client, from which the output can be reconstructed. The servers do not need to interact with each other, or even to know each other. Homomorphic secret sharing provides a very natural solution in this scenario: a one-time key setup is executed, and each server S_b for $b \in \{0,1\}$ gets an evaluation key ek_b. All parties get the public key pk. Each computation can be performed as follows:

- **Send input to Servers.** Each client C_i with input w_i computes $\mathsf{ct}_i \leftarrow \mathsf{Enc}(\mathsf{pk}, w_i)$ and uploads ct_i on some public repository.
- **Program Evaluation.** Each server S_b recovers $(\mathsf{ct}_1, \cdots, \mathsf{ct}_n)$ and computes $z_b \leftarrow \mathsf{Eval}(b, \mathsf{ek}_b, (\mathsf{ct}_1, \cdots, \mathsf{ct}_n), P, \delta)$, where P is a public program, and δ is set depending of the application.
- **Send output to Client.** Each server S_b sends z_b to the output client, who recovers the output $z \leftarrow z_0 \oplus z_1$.

A natural scenario for this kind of computation can be a vote: multiple users encrypt their vote and upload them on a public repository. The two servers retrieve the encrypted votes and evaluate a voting function on them (it can be for example a threshold function, or a conjunction), without having to interact (which mitigates risks of collusions and latency issues). Shares of the result of the vote are then sent to the client, who can reconstruct the result by performing a simple XOR. In case of failure, the vote can be re-started.

Managing the Leakage. The event of a failure is directly related to the private inputs of the clients and the secret key of the scheme. In some cases, this might not be an issue: the private inputs are compromised only when a leakage occurs while a server is corrupted. In scenarios where a server has a low probability of being corrupted, this conjunction of event can be acceptably rare.

In scenarios where this would be an issue, the parties can use the approach outlined Section 4.4.1 to reduce the dependency between a failure event and a leakage event: some number N of leakage-absorbing pads is generated as part of the distributed key setup. Computations are performed on the inputs using the MaskedEval algorithm. The same pad is used in each computation until the servers get noticed of a failure event; when this happens, the compromised pad is replaced by a new pad in subsequent computations. This makes the leakage probability quadratically smaller than the failure probability.

Efficiency Estimations. Consider for example the case of n clients who want to compute the majority of their private inputs. The majority function can be implemented using an RMS program with $(n+1)^2/4 - 1$ non-terminal multiplications. Each client sends one ciphertext encrypting his input, with basis $B = 2$ if using leakage-absorbing pads (for XOR-masking), and $B = 16$ otherwise. Figure 3 shows the time required to compute the majority function on n inputs, using either Eval directly, or using leakage-absorbing pads and MaskedEval. Without leakage-absorbing pads, a ciphertext is of size 10.6kB. With leakage-absorbing pads, a ciphertext is of size 35.9kB. The parameters are chosen to ensure a 10^{-4} leakage probability, and allow for the evaluation of about 10^4 functions before refreshing the key. In the setting with leakage-absorbing pads, this requires generating a number $N = 100$ of pads during the setup. Note that the failure probability corresponding to a 10^{-4} leakage probability is 1% with leakage pads, and 0.01% without leakage pads. However, one can easily mitigate this issue by setting the leakage probability of the pad-based protocol to $10^{-4}/2$ and re-running it when a failure occurs, which allows to maintain a 10^{-4} leakage probability while making the failure probability comparable to that of the protocol without pads, at essentially no cost in efficiency (as the protocol is re-run only when a failure actually occurs).

Advantage over alternative approaches. This approach has the advantage of being particularly efficient for the clients. An alternative method would be to let the two servers share some correlated randomness (e.g. garbled circuits distributed at the key setup phase) to perform privately the computation, or to rely on a single-server solution with fully-homomorphic encryption. However, these solutions would completely break down if a client colludes with a server, while the HSS-based solution outlined above is resistant to collusions between any number of clients and a single server.

5.2 Secure Data Access

In this section, we discuss three natural applications of HSS to secure data access: policy-based file systems, private RSS feeds, and private information retrieval.

5.2.1 Policy-Based File System. Consider the following scenario: a data owner wants to maintain a file system where users, identified by a set of attributes, can access encrypted files according to some

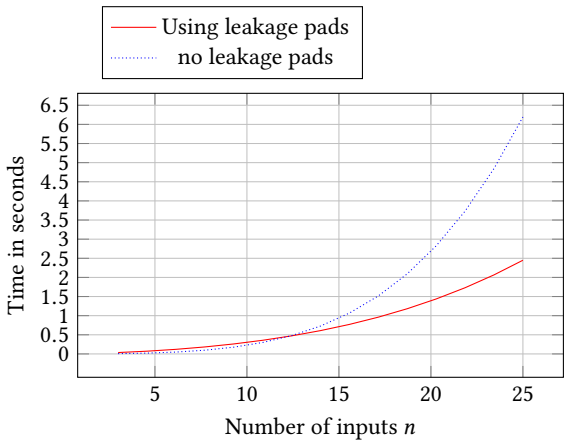

Figure 3: Time to compute majority of n inputs with 10^{-4} leakage probability, with and without leakage-absorbing pads, on a single thread of an Intel Core i7 CPU. See Remark 6.1 for further implementation details.

policy. Let us outline a brief intuition of an HSS-based solution: the data owner D generates the keys of a secret-key Las Vegas HSS and sends them to two servers (S_0, S_1), together with some encrypted vectors that indicates how permissions to access the files should be granted given the vector of attributes of some client. A public repository contains encrypted files $E_K(m)$, where the key K is derived from a large value r encrypted by the data owner. An RMS program P determines whether access should be granted to a client. We use the enhanced semantic of section 4.5.1 to allow the program P to handle the large input r in a terminal multiplication. We discuss this application in more details in the full version [9].

5.2.2 Private RSS Feed. Consider the following scenario: a client has subscribed to a (potentially large) number of RSS feeds, and would like to receive regular updates on whether new data might interest him. Typical examples could be getting newspapers relevant to his center of interest, or job offers corresponding to his abilities. Each data is categorized by a set of tags, and the client wishes to retrieve data containing specific tags (one can also envision retrieving data according to more complex predicates on the tags) in a private way (without revealing to the servers his topics of interest, or his curriculum vitae).

The trivial solution in this scenario would be to let the servers send regular digests to the client, containing the list of all tags attached to each newly arrived data, so as to let the client determine which data interest him. But with the potentially large number of tags associated to each data, and the large quantity of new data, the client would have to receive a large volume of essentially non-relevant information, which consumes bandwidth and power.

In this section, we show how homomorphic secret sharing can be used to optimally compress such digest while maintaining the privacy of the client. After a setup phase, in which the client encrypts a query that indicates his area of interests, he will receive on average a *two bits* from each of two servers maintaining the database, from which he can learn whether a new record is likely to interest him.

Basic Setting. A database publicly maintained by two servers (S_0, S_1), who hold respective evaluation keys $(\mathsf{ek}_0, \mathsf{ek}_1)$ for a secret-key Las Vegas HSS scheme, is regularly updated with new records R. Each record comes with a size-n string of bits, indicating for each possible tag whether it is relevant to this record. In the most basic scenario, the client sends an encryption of the list of bits indicating all tags that interest him in a setup phase. For each new record, the client wants to know whether the record contains all tags that interest him. The protocol is represented Figure 4. Each string $(r_i)_{i\leq n}$ associated to a record contains typically a very large number of zeros, and the corresponding RMS program $P[r_1, \cdots, r_n]$ is essentially a conjunction of n' inputs, where n' is close to n.

A nice feature of this private RSS feed protocol is that the servers do not need to interact at all – they do not even need to know each other, which strongly reduces the risk of collusions and can be used in the setup phase to mitigate hacking, by secretly choosing the two servers.

Enhanced Scenario. Once he finds out that a new record interests him, the client will likely want to retrieve it privately. This can be done very efficiently using the two-server PIR protocol of [30] that relies on distributed point functions (which can be built from any one-way function). The servers can also apply more complex permission policy functions, such as a disjunction of conjunctions, which can be easily translated to RMS programs. The group-based public-key HSS scheme also easily supports inputs from multiple clients, which allows to append for example an encrypted permission string, coming from e.g. the news provider, to the encrypted query of the client. The RMS program would then indicate to the client that a record is likely to interest him only if his permission data indicates that he is authorized to get this record.

Private RSS feed for two servers S_0, S_1 and one client C:

Global Setup: Let $(\mathsf{pk}, \mathsf{ek}_0, \mathsf{ek}_1) \xleftarrow{\$} \mathsf{Gen}(1^\lambda)$. S_0 gets $(\mathsf{pk}, \mathsf{ek}_0)$, S_1 gets $(\mathsf{pk}, \mathsf{ek}_1)$, and C gets pk.

Client Setup: For each of n possible tags, C computes $\mathsf{ct}_i \leftarrow \mathsf{Enc}(\mathsf{pk}, w_i)$ where $w_i = 1$ if the ith tag matches the interests of C, and $w_i = 0$ otherwise. C sends $(\mathsf{ct}_i)_{i\leq n}$ to (S_0, S_1).

Digest Generation: For each new record R_j added to the database, associated to a list of n bits $(r_i)_{i\leq n}$ identifying the tags of the record, each server S_b computes $(x_j^b, \gamma_j^b) \leftarrow \mathsf{Eval}(b, \mathsf{ek}_b, (\mathsf{ct}_1, \ldots, \mathsf{ct}_n), P[r_1, \cdots, r_n], \delta)$ where $P[r_1, \cdots, r_n]$ is an RMS program with $(r_i)_{i\leq n}$ hard-coded that returns 1 iff it holds that $r_i = 1$ for all j such that $w_i = 1$. Once N new records have been added, each server S_b sends $(I, (x_j^b, \gamma_j^b)_{j\leq N})$ to C, where I is a unique identifier of the digest.

Parsing the Digest: C computes $x_j \leftarrow x_j^0 \oplus x_j^1$ for each j such that $(\gamma_j^0, \gamma_j^1) \neq (\bot, \bot)$.

Figure 4: Private RSS Feed Protocol.

Efficiency Estimations. Using the algorithmic optimizations of Section 4 together with our optimized implementation, an RMS program with 50 non-terminal multiplicative gates (e.g. a conjunction of 51 inputs, a majority of 13 inputs, or any branching program or boolean formula with 51 gates) can be evaluated with a 1% failure probability on an encrypted query in less than 0.1 second on a single thread of an Intel Core i7 CPU, using $B = 16$ as the basis for the ElGamal secret key. An encrypted bit amounts to about 10kB, using the generic ciphertext compression method of section 4.2.

Comparison with Alternative Approaches. A number of alternative approaches can be envisioned for the above application. An attractive approach for small values of n is to use distributed point functions [30] (DPF), which can be implemented very efficiently using block ciphers such as AES [12], by letting the servers match the private query with all 2^n possible vectors of length n. This solution becomes clearly impractical as soon as n becomes large, while our HSS-based solution can handle values of n ranging from a few dozens to a few hundreds.

5.2.3 Private Information Retrieval. Private Information Retrieval (PIR) allows a client to query items in a database held by one or more servers, while hiding his query from each server. This problem has been extensively studied in the cryptographic community, see [19, 38]. In this section, we outline how homomorphic secret sharing can be used to construct efficient 2-server PIR schemes supporting rich queries that can be expressed by general formulas or branching programs.

The setting is comparable to the setting of the private RSS feed protocol described in Section 5.2.2: the client applies the HSS sharing algorithm to split the query q between the servers. (Here the more efficient secret-key variant of HSS suffices.) The servers use the HSS evaluation algorithm to non-interactively compute, for each attribute vector of database item, a secret-sharing of 0 (for no match) or 1 (match). The main challenge is for the servers to send a single succinct answer to the client, from which he can retrieve all items that matched his query (possibly with some additional items). We describe below a method to achieve this.

Retrieving a Bounded Number of Items. We start by assuming that the client wishes to retrieve items matching his query, up to some public bound n on the number of matching items. let N be the size of the database, and let $(m_i^b)_{i \le N}$ be the output shares of each server S_b obtained by matching the encrypted query with each vector of attribute a_i of the database. Let $(m_i)_{i \le N}$ be the corresponding outputs. Each server S_b interprets his shares $(m_i^b)_{i \le N}$ as a vector over $(\mathbb{F}_{2^k})^N$, for some large enough k (e.g. $k = 40$). Both servers replace each share for which they raised a flag, indicating a potential failure, by a uniformly random value over $\mathbb{F}_{2^k}$. This ensures that the elements of $(m_i)_{i \le N}$ for which a failure occurred will not be equal to 0, with very high probability.

Then, the servers can non-interactively reconstruct shares of the database entries D_i for with $m_i \neq 0$, up to the bound n of the number of such entries, using a syndrome of a linear error-correcting codes (interpreting the vector $(m_i)_{i \le N}$ with failures as a noisy codeword), and send the resulting vectors $(v_i^b)_{i \le n}$ for $b \in \{0, 1\}$ to the client. Eventually, the requirement of a bound n on the number of matches can be removed by repeating the above procedure using successive powers of 2 as guesses for the bound. Concretely, for $n = 1, 2, 4, 8, \ldots, N$, the servers use a common (pseudo-)randomness to replace each entry in the vector by 0 except with $1/n$ probability, repeating several times to reduce the failure probability (see, e.g., [41]).

5.3 Generating Correlated Randomness

Special forms of correlated randomness serve as useful resources for speeding up cryptographic protocols. HSS techniques provide a promising means for generating *large* instances of certain correlations while requiring only a *small* amount of communication. This approach is particularly effective for correlations evaluable in low depth, and with long output, by homomorphically "expanding" out encoded input values into shares of the output.

In this section, we discuss a few sample correlation classes that are HSS amenable. In each case, when generating the correlation, we assume the parties have run a (one-time) distributed HSS key generation (as in Section 4.1), yielding keys $(\mathsf{pk}, \mathsf{ek}_0, \mathsf{ek}_1)$.

5.3.1 Bilinear Form Correlations. Consider the following 2-party "bilinear form" correlation: party P_0 holds a random vector $x \in \{0,1\}^n$, P_1 holds a random vector $y \in \{0,1\}^n$, and the parties hold additive secret shares (over $\mathbb{Z}_\beta$) of each of the n^2 products $x_i y_j \in \{0, 1\}$. Intuitively, by taking appropriate linear combinations of the shares of $x_i y_j$ (which can be performed locally), this correlation encodes additive sharings of all possible bilinear forms on x and y.

Generating Bilinear Form Correlations via HSS. A core observation for efficiency is that the computation can be completed via just n^2 *terminal* RMS multiplications, by using the procedure described in Section 4.1.2 for "loading" the inputs x_i, y_j as level 2 HSS shares via an OT-based protocol (avoiding the need for an additional homomorphic multiplication to do so). As described in Section 4.5 (Terminal Multiplication discussion), this means just a single pairing and conversion is required per multiplication, and the HSS encoding of each bit can be given by a single ElGamal ciphertext. More specifically, it suffices to send ElGamal encryptions of just Party A's input bits and to perform the OT-based protocol for encoding the input bits of just Party B. Explicitly:

(1) Each party samples a respective vector, $x, y \leftarrow \{0,1\}^n$.
(2) Party A encodes his input x bitwise using HSS: i.e., $\forall i \in [n], \mathsf{ct}_i^x \leftarrow \mathsf{Enc}(\mathsf{pk}, x_i)$, and sends the resulting ciphertexts $(\mathsf{ct}_i^x)_{i \in [n]}$ to Party B.
(3) The parties run the OT-based protocol described in Section 4.1.2 (Figure 2) to load Party B's input y bitwise into HSS memory as level 2 encodings.
(4) Locally, each party then homomorphically evaluates the RMS program P_{bilin} that computes n^2 RMS multiplications between input value x_i and memory value y_j, for each $i, j \in [n]$, and outputs the value modulo $\beta \in \mathbb{N}$ (determined by the specific application). The error for each multiplication is set to δ/n^2. (That is, each party executes $\mathsf{Eval}'(b, \mathsf{ek}_b, ((\mathsf{ct}_i^x, \mathsf{ct}_j^y)_{i,j \in [n]}), P_{\mathsf{bilin}}, \delta)$, where Eval' denotes Eval with the first homomorphic "load to memory" step skipped.)

(5) Each party outputs its vector (x or y), together with its evaluated share (as an element of $\mathbb{Z}_\beta$) of the each of the n^2 products.

The total required communication is $640 \times n$ bits for the input-encoding OTs for Party B (see Section 4.1.2), plus n ElGamal ciphertexts for Party A, which correspond to $2n$ group elements (each 1536 bits). In total, $640 \times n + 1536 \times 2n = 3712n$ bits.

The local HSS evaluation runtime corresponds to n^2 *terminal* RMS multiplications, i.e. n^2 total exponentiations and share conversions. To obtain overall failure probability δ, each terminal multiplication must be performed with failure probability δ/n^2.

Applications of Bilinear Form Correlations. This bilinear correlation distribution can aid the following sample applications. For each, we discuss how competitive the HSS-based approach currently is compared to existing solutions.

Generating Beaver triples over rings. Beaver triple correlations [5] over a ring R are comprised of a pair of random elements $x, y \in R$ where each element is known by one party, as well as additive secret shares of their product $xy \in R$ (where addition and multiplication are over the ring). Given such multiplication triples, one can obtain secure computation protocols for computations over R with near-optimal computational complexity (e.g., [4, 8, 22, 36]).

We demonstrate how to generate a Beaver triple over $\mathbb{Z}_m$ for arbitrary m (as well as for general finite fields $\mathbb{F}$), from HSS via bilinear form correlations. For concreteness, consider $R = \mathbb{Z}_{2^n}$.

Given an n-bit representation of $x, y \in \mathbb{Z}_{2^n}$, the product $xy \in \mathbb{Z}_{2^n}$ can be obtained by: (1) applying a fixed bilinear form $M \in \mathbb{Z}_2^{n \times n}$ on the bits of x and y corresponding to "grade-school" multiplication of multi-digit numbers, (2) followed by a modular reduction by 2^n.[2] Step (1) is exactly a special case of a bilinear form correlation (for fixed linear combination of the terms $x_i y_j$); and, the reduction in step (2) can be *locally* applied on the individual output shares of each party.

Overall, this means generating each Beaver triple over $\mathbb{Z}_{2^n}$ with failure δ requires communication of $3712n$ bits, and computation of n^2 terminal RMS multiplications (output as shares in $\mathbb{Z}_\beta$ for $\beta = 2^n$), each with failure δ/n^2. (Over field $\mathbb{F}_{p^n}$, the same holds with $n' := n\lceil \log_2 p \rceil$.) In this failure regime, the RMS multiplications are dominated by conversions. Estimating a baseline of 5×10^9 conversion steps per second (see Section 6), together with effective $\times 8$ speedup from the relevant optimizations in Section 4 ($\times 4$ for expected payloads, $\times 2$ for 10^{d-1} distinguished points),[3] to generate an n-bit Beaver triple with overall failure probability $1/16$ takes $\sim n^4/2^{30.2}$ seconds; for example, for 128-bit inputs (i.e., $n = 2^7$) this is roughly 60kB communication and 0.22s computation time.

Consider the following alternative methods for Beaver triple generation.

- *Paillier based.* Beaver triples can be generated using an encryption scheme that supports homomorphic addition and multiplication by scalars, such as the Paillier cryptosystem.[4] This approach requires notably less communication than the HSS-based approach, as only 2 ciphertexts are required as opposed to one ciphertext per input bit (where Paillier ciphertexts with 80 bits of security are comparable size to ours), and computationally requires a small constant number of group operations.

  However, this approach does not fully subsume HSS techniques (and may be less preferred in certain applications), because of the qualitatively different structure of the resulting protocol. In this approach, the parties must exchange information, perform a heavy "public key" computation (homomorphic evaluation), then exchange information once again, and then perform another heavy computation (ciphertexts to be locally decrypted). In particular, the computation and second exchange must be performed if there is a chance the parties will wish to engage in secure computation in the future.

  In contrast, using HSS, the parties need only exchange information once; this means a party can exchange HSS shares with many others, and only later decide which from among these he wishes to expend the computation to "expand" the shares into correlated randomness. The expansion of shares only involves local computation without communication, which can be useful for mitigating traffic analysis attacks. Another advantage of the HSS-based approach is that it can use the same setup for generating correlations over different rings. This can be useful, for instance, for secure computation over the integers where the bit-length of the inputs is not known in advance.
- *Coding based.* Assuming coding-based intractability assumptions such as the pseudo-randomness of noisy Reed-Solomon codes, there are protocols for generating Beaver triples of n-bit field elements at an *amortized* cost of $O(n)$ bits per triple [2, 28, 35, 40]. These constructions rely on relatively nonstandard assumptions whose choice of parameters may require further scrutiny. Moreover, amortization only kicks in when the number of instances is large (at least a few hundreds). In contrast, the HSS-based approach can apply to a small number of instances and, as noted before, can use the same setup for generating correlations over different fields.
- *OT based.* Perhaps the best comparison approach for generating Beaver triples of n-bit ring elements (without requiring amortization across a very large number of instances) is achieved by evaluating n 1-out-of-2 OTs of n-bit strings [29, 36]. While this computation can be heavily optimized for large n using OT extension, it requires communication of $2n(\lambda + \ell)$ bits per such OT, for $\lambda = 80$ and $\ell = n$. For $n \geq 1776 \approx 2^{10.8}$ this is greater communication than our approach (and we expect this crossover to drop substantially with future optimizations); note that in our

[2] For finite fields $\mathbb{F} \cong \mathbb{Z}_p[X]/(f)$, M corresponds to polynomial multiplication, and modular reduction is taken mod (f).

[3] Note we cannot take advantage of the $\times 2$ speedup for even/odd failure recovery since this requires shares in a field of characteristic 2 whereas in this example shares are over $\mathbb{Z}_{2^n}$.

[4] For example, a Beaver triple can be generated from 2 executions of oblivious linear evaluation (OLE), each of which achieved as: Party A generates a key pair $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}_{\mathsf{Enc}}(1^\lambda)$ and sends an encryption $\mathsf{Enc}(x)$ of $x \in R$ to Party B, who replies with the homomorphic evaluation $\mathsf{Enc}(ax + b)$ for his $a, b \in R$, back to Party A who can decrypt and learn $ax + b$.

current implementation (on a single core of a standard laptop), a $2^{10.8}$-bit Beaver triple correlation can be generated via HSS with overall failure probability $1/4$ in ~ 33.6 minutes.

We remark that the crossover point is lower when instantiating the HSS using ElGamal over elliptic-curve groups. As discussed in Section 6.2, homomorphic evaluation over an elliptic-curve group presently runs slower than over a conversion-friendly group by roughly a factor of 5×10^3 (approx 10^6 conversions per second as opposed to 5×10^9), but the corresponding ciphertext size is approximately 8 times smaller. In this setting, the HSS-based solution requires $1024n$ bits of communication (in the place of $3712n$), yielding a crossover of $n = 432 \approx 2^{8.8}$. The current implementation of HSS over elliptic curves would run notably longer at this size ($\sim$ 10 hours for failure probability $1/4$), but discovery of "conversion-friendly" elliptic curve techniques may make this approach more competitive.

Preprocessing for matrix multiplication. Similar online speedups can be achieved for secure $n \times n$ matrix multiplication given an extension of the bilinear form correlations with $2n$ random (n-bit) input vectors; Generating this correlation via HSS with failure probability δ requires communication of $2n^2$ group elements (one ElGamal CT for each input bit of one party) plus $640 \times n^2$ bits (for n^2 input-loading OTs, for the other party), altogether $3712n^2$ bits. The required computation is n^3 terminal RMS multiplications each with failure δ/n^3.

To our knowledge, the best existing approach is via n^2 1-out-of-2 OTs of n-bit strings [29, 36]. Using OT extension, this requires communication of $2n^2(\lambda + \ell)$ bits, for $\lambda = 80$ and $\ell = n$. For the same crossover value as above $n \geq 1776 \approx 2^{10.8}$ this is greater communication than our approach. The required computation time of the HSS-based solution at this crossover point is presently quite large; however, this will improve greatly with time and additional computing power.

Universal bilinear forms. An appealing property of the HSS-based generation procedure that sets it apart from competing techniques is its universality: The same fixed communication and computation can be used to speed up online evaluation of *any collection* of bilinear maps on a set of inputs, and the identity of the maps need not be known during the preprocessing phase.

For example, suppose parties hold respective inputs $x, y \in \{0,1\}^n$, and wish to securely evaluate $x^T Ay$ for a collection of many different matrices $A \in \{0,1\}^{n \times n}$, possibly not known at setup time. For instance, each A may be an adjacency matrix representing possible connectivity structures between n locations, so that the above product computes correlation information along the graph between the resource distribution of the two parties (encoded by x and y). Given an instance of the bilinear form correlation (shares of $r^x, r^y \in \{0,1\}^n$ and each $r_i^x r_j^y \in \{0,1\}$), then for each desired $A = (a_{ij})$ the parties can take the appropriate linear combination of their $r_i^x r_j^y$ shares (with coefficients a_{ij}) to yield a corresponding "bilinear Beaver triple." This can be done even if the identity of matrices A is not determined until runtime.

To the best of our knowledge, in this regime of universality, the best competition is generic Yao/GMW for securely evaluating all n^2 products. Even utilizing optimized OT extension techniques [37], this will require more than $100n^2$ bits of communication, indicating that an HSS-based approach wins in communication already for $n \geq 38$. The computation required for a 38-bit Beaver triple correlation can be generated via HSS with overall failure probability $1/4$ in ~ 0.42ms.

5.3.2 Truth Table Correlations. Given access to a preprocessed "one-time truth-table" correlation, one can securely evaluate any function with polynomial-size domain by a single memory lookup and short message exchange [21, 33]. Executing this technique on small chosen sub-computations within a larger secure computation can provide helpful speedups [21].

For function $f : [N_1] \times [N_2] \to [M]$, a one-time truth-table correlation gives the two parties a random x-offset $r_x \leftarrow [N_1]$ and y-offset $r_y \leftarrow [N_2]$, respectively, and gives the parties additive secret shares (over $\mathbb{Z}_M$) of the *shifted* truth table $(f(w - r_x, z - r_y))_{w \in [N_1], z \in [N_2]}$. To securely evaluate f on inputs x, y at runtime, the parties exchange the masked values $(x - r_x), (y - r_y)$, and then use the shares in the corresponding position of their respective truth tables as their output shares of the f computation.

HSS can be used to generate one-time truth table correlations for functions of relatively small size domains, e.g. 128×128. We assume the truth table of f is public. One approach is a further instance of bilinear form correlations. Each party samples his random offset, e.g. $r_x \leftarrow [128]$, HSS-encodes the corresponding 128-bit unit vector, and sends the encoding. Locally, each party homomorphically expands the $(128)^2$-bit tensor of the encoded vectors, to obtain shares $a_{i,j}$. His output share for the (x, y)th position in the shifted truth table is the linear combination $\sum_{i,j \in [128]} f(i,j) a_{i-x, j-y}$ over the output space of f (say $\{0,1\}$). The cost analysis of the RMS portion of this procedure is equivalent to that of the Beaver triples.

5.4 Cryptographic Capsules

As a direction of future research, we propose HSS as a promising approach for generating many (pseudo-)independent instances of useful correlations given a *short, one-time* communication between parties. The idea is for parties to exchange a single short "capsule" of HSS-encoded randomness, then locally apply HSS evaluation of the computation that first expands the seed into a long sequence of pseudo-random bits and then uses the resulting bits within the sampling algorithm for the desired correlation. Combined with high-stretch local PRGs [1, 3, 34], this may yield compression schemes for many useful types of correlations. A natural application of cryptographic capsules is to execute the preprocessing phase of a multiparty computation protocol, using short communication (of size $O(C^{1/k})$ for generating the material for evaluating a circuit of size C, where k is the locality parameter of the high-stretch local PRG; all known protocols for generating such preprocessing material have communication $O(C)$).

Additional challenges arise in this setting when dealing with HSS error, since the number of homomorphic multiplications will be much greater than the size of the HSS-encoded seed. We introduce two new techniques for addressing the effects of leakage.

The first is a method of "bootstrapping" leakage pads (as in Section 4.4.1), enabling the parties to homomorphically generate fresh pseudorandom pads from a small starting set via homomorphic evaluation. The second is a more sophisticated variant of punctured OT from [13], making use of prefix-punctured PRFs. Combined, we are able to drop the cost of expanding an n-bit seed to m bits of correlation (for $m \gg n$) from $O(m/n)$ per output using [13] to $O(\sqrt{m/n})$ using our new techniques. We devote a section to the study of this tool in the full version [9].

6 CONCRETE EFFICIENCY

In this section we discuss the concrete performance of our HSS implementation, providing both analytical predictions and empirical data. Our implementation builds on the optimized conversion algorithm from [13], but incorporates additional optimizations that significantly improve the system's performance. The optimizations include the algorithmic improvements discussed in Section 4 and some additional machine-level optimizations we describe in this section.

We assume RMS multiplications are performed in the context of an application which specifies a target error probability ε for each multiplication. The performance of an RMS multiplication given ε is determined by the performance of its two main components, exponentiations in the underlying group $\mathbb{G}$ (we will use multiplicative notation for the group operation) and multiplicative-to-additive share conversions in this group.

Similarly to [13], we take $\mathbb{G}$ to be a large sub-group of $\mathbb{Z}_p^*$ for a prime p that is pseudo-Mersenne, safe and $\pm 1 \bmod 8$. That is, $p = 2^n - \gamma$ for a small γ and $p = 2q + 1$ for a prime q. If p is such a prime then 2 is a generator of a group of size q in which the DDH problem is assumed to be hard. One specific prime of this type on which we ran our measurements is $2^{1536} - 11510609$.

The optimized implementation from [13] viewed any element with d leading zeros, i.e. an integer in the range $0, \ldots, 2^{n-d} - 1$, as a distinguished point. The problem of locating a distinguished point in the sequence $h, 2h, \ldots, 2^{w-1}h$, where w is the word size of the underlying computer architecture, is reduced to searching for the pattern 0^d in the first word of the representation of h. Computing $h2^w$ from h requires with high probability only one multiplication and one addition, if $\gamma < 2^w$.

As discussed in Section 4.3, we improve on the approach of [13] for conversion in several ways. First a distinguished point begins with the pattern 10^d, i.e. all integers in the range $2^{n-1}, \ldots, 2^{n-1}2^{n-d} - 1$. By Lemma 4.6 the probability of error is $z \cdot 2^{-d-1}$ for a payload z while the expected running time is 2^{d+1}. Based on this lemma and on Corollary 4.8 the average probability of error in a single conversion on bit inputs is $(B-1)/16$. This is a factor 16 improvement over the worst-case analysis of [13]. In fact, replacing the pattern 0^d by 10^d is necessary for this improvement. Finally, a series of low level optimizations, described in Section 6.1 reduces the running time by another factor of two. Altogether, we improve the running time of the conversion procedure for a given failure probability by a factor of 32 over the conversion procedure of [13].

Three optimizations that were introduced in [11, 13] and which we use are short-keys, time-memory trade-off for fixed-base exponentiation and large-basis for key representation. The secret key c which we used for ElGamal encryption is *short*, 160 bits in our implementation, which is sufficiently secure given known cryptanalytic attacks. Trading memory for time in fixed base exponentiation for base h, and maximum exponent length e is possible for a parameter R by storing $h^{2^{Ri}+j}$ for $i = 0, \ldots, \lceil e/R \rceil - 1, j = 0, \ldots, R-1$. Exponentiation can be computed by roughly $\lceil e/R \rceil - 1$ modular multiplications of stored elements. The secret key c can be represented in base B instead of in binary, reducing the number of ElGamal ciphertexts encrypting integers of the form $xc^{(i)}$ from 160 per input bit to $160/\log B$. This optimization reduces the storage and the number of exponentiations at the expense of increasing the number of conversion steps required for the same error probability δ by a factor of B.

Table 2 sums up the parameters of a single RMS multiplication.

Parameter	Analytic expression
Failure probability	$(1 + s(B-1)/2)/2^{d+3}$
Group operations	$(s+1)\frac{\ell+2d}{R}$
Expected conv. steps	$(s+1)2^{d+1}$
Public key size (DEHE)	$s + 3\lceil\sqrt{s}\rceil + 2$
Share size per input (HSS)	$s + 2$
Ciphertext size per input (DEHE)	$s + 2\lceil\sqrt{s}\rceil + 1$
Preprocessing memory	$(s+1)(\ell + 2d)(2^R - 1)/R$

Table 2: Parameters of a single RMS multiplication of binary values as a function of B (basis size for representing secret key), $s = \lceil 160/\log B \rceil$ (for 160-bit ElGamal secret key), R (modular exponentiation preprocessing parameter), and d (zero-sequence length for the conversion algorithm). All sizes are measured in group elements.

6.1 Low Level Optimizations

We were able to obtain substantial - more than double - improvements in the implementation of conversion algorithm compared to the method described in [13]. Boyle et al. look for the distinguishing pattern by considering "windows" of size $w = 32$ bits in the binary representation of the group element. Each window, once fixed, is divided into strips of length $d/2$ and to look first for a zero-strip of length $d/2$, then incrementally count zeros on the left and on the right.

A straightforward improvement over the reported implementation of [13] is to extend the window size to $w = 64$, and use the 64-bit arithmetic offered by any modern CPU. Furthermore, we noticed that with the aid of a *partial match* lookup table, it is possible to avoid counting zeros on the left and on the right.

For an integer i, let $l(i)$ be the number of trailing zeros in the binary representation of i. Consider a table T of $2^{d/2}$ elements such that $T(i) = 2^{l(i)} - 1$ for all $0 \le i < 2^{d/2}$. If the j-th strip is $0^{d/2}$, the value of the preceding strip is i and the value of the subsequent strip is k then a strip of d zero occurs if and only if $k < T(i)$, as the binary representation k of the next strip has at least $d/2 - l(i)$ leading zeros. With the above optimization, and with a word size $w = 64$ bits, we were able to achieve roughly 5 billion conversion steps per second,

using seven cycles for pattern-matching and eight for the modular multiplication by 2^w (relying on the synthetic 128-bit arithmetic offered by the compiler).

All remaining arithmetic operations were based on the GNU Multiple Precision library[5], on top of which we optimized some primitives. For example, modular reductions can be improved by a factor of 2 just by exploiting the structure of the prime modulus, using a single 64-bit multiplication and one addition.

Basic HSS operations, such as conversions and fixed-base group operations, add up to less than 150 lines of code and run on a single thread, meaning that all the following results can be easily scaled linearly with the number of available processors. Our library is publicly available, together with the raw benchmarks data. Our implementation is released into the public domain.

6.2 Measured Results

Using our optimized implementation for modular multiplication, we were able to report about 10^6 modular multiplications per second. In order to obtain time estimates for conversions on elliptic-curve groups, we benchmarked OpenSSL's implementation of SECG curves `secp160k1` and `secp160r1`[43], both providing 80 bits of security. In both cases, we were able to measure about $4.1 \cdot 10^6$ group operations per second. This is three orders of magnitude slower than what can be achieved on conversion-friendly groups; in the low-error regime, where conversions dominate, elliptic-curve based HSS should therefore be about a thousand times slower that their counterpart based on conversion-friendly groups. On the other hand, it is worth noting that the size of the HSS ciphertexts in the elliptic curve implementation are smaller by roughly a factor of 10 (1.1kB vs. 10.6kB).

Remark 6.1. *We summarize below the parameters and assumptions on which our concrete efficiency analysis is based.*

- **Processor:** *Benchmarks have been performed on an* `Intel Core i7-3537U @ 2.00GHz` *processor running Debian stretch - Linux 4.9 patched with Grsecurity, and on a* `Intel Xenon E5-2650 @ 2.00GHz` *running Ubuntu 16.04.2 LTS.*
- **Group:** *We used a conversion-friendly group with a pseudo-Mersenne modulus* $p = 2^{1536} - 11510609$ *(hence group elements are* 1536 *bits long).*
- **Cost of operations:** *With the above settings, we were able to perform roughly* $5 \cdot 10^9$ *conversion steps per second and* 10^6 *mod-*p *multiplications per second on average.*
- **Optimizations:** *Improvements from Section 4 where assumed when relevant, such as ciphertext size reduction under the* ESDH *assumption, and randomized conversions.*
- **Parameters:** *Experiments were run for bases* $B = 2, 4, 16$ *for the secret key, and with precomputation parameter* $R = 1, 8, 12$ *for exponentiations.*

We also compared our implementation using a partial match table with a naïvely optimized version of [13] with extended window size, obtaining roughly a 50% factor of improvement for the `Intel Core i7` cpu.

Figure 5 shows the number of full RMS multiplications that can be performed in one second for a given failure probability per RMS multiplication. The curves are based on an analytical formula derived from the data obtained in the previous experiment. Additional analysis, graphs, and benchmarks, are available at [URL omitted to maintain author anonymity]. Some concrete numbers are also given in Table 3.

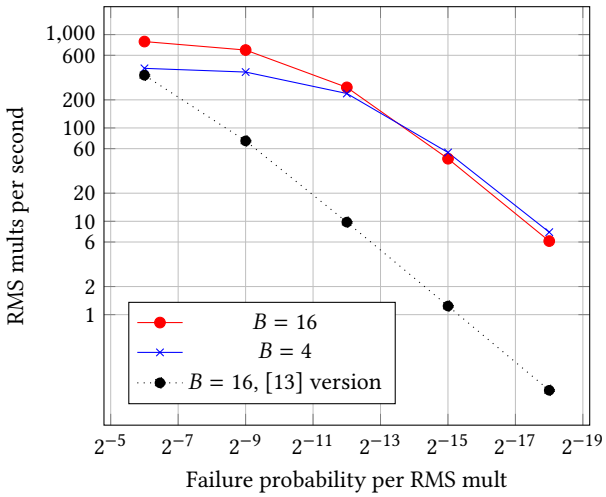

Figure 5: Number of RMS multiplication per second given the failure probability per RMS multiplication with $R = 8$. The ciphertext size for $B = 4$ (resp., $B = 16$) is 18.8kB (resp., 10.6kB). See Remark 6.1 for implementation details. The version of [13] assumes 2×10^9 conversion steps per second.

Failure	Base	Tradeoff param.	Length of dist. point	Share (kB)	RMS mult. per second
$\epsilon = 2^{-5}$	$B = 4$	$R = 1$	$d = 9$	18.8	55
	$B = 4$	$R = 8$	$d = 9$	18.8	438
	$B = 16$	$R = 1$	$d = 11$	10.6	109
	$B = 16$	$R = 8$	$d = 11$	10.6	856
$\epsilon = 2^{-10}$	$B = 4$	$R = 1$	$d = 14$	18.8	54
	$B = 4$	$R = 8$	$d = 14$	18.8	361
	$B = 16$	$R = 1$	$d = 16$	10.6	101
	$B = 16$	$R = 8$	$d = 16$	10.6	562
$\epsilon = 2^{-15}$	$B = 4$	$R = 1$	$d = 19$	18.8	29
	$B = 4$	$R = 8$	$d = 19$	18.8	55
	$B = 16$	$R = 1$	$d = 21$	10.6	34
	$B = 16$	$R = 8$	$d = 21$	10.6	47

Table 3: Performance of RMS multiplications, see Remark 6.1 for implementation details.

[5] https://gmplib.org/

REFERENCES

[1] Benny Applebaum. 2013. Pseudorandom Generators with Long Stretch and Low Locality from Random Local One-Way Functions. *SIAM J. Comput.* 42, 5 (2013), 2008–2037. https://doi.org/10.1137/120884857

[2] Benny Applebaum, Ivan Damgård, Yuval Ishai, Michael Nielsen, and Lior Zichron. 2017. Secure Arithmetic Computation with Constant Computational Overhead. In *Crypto 2017*.

[3] Benny Applebaum and Shachar Lovett. 2016. Algebraic attacks against random local functions and their countermeasures. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*. 1087–1100. https://doi.org/10.1145/2897518.2897554

[4] Donald Beaver. 1992. Foundations of Secure Interactive Computing. In *CRYPTO'91 (LNCS)*, Joan Feigenbaum (Ed.), Vol. 576. Springer, Heidelberg, 377–391.

[5] Donald Beaver. 1995. Precomputing Oblivious Transfer. In *CRYPTO'95 (LNCS)*, Don Coppersmith (Ed.), Vol. 963. Springer, Heidelberg, 97–109.

[6] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. 1988. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract). In *STOC*. 1–10.

[7] Josh Cohen Benaloh. 1986. Secret Sharing Homomorphisms: Keeping Shares of A Secret Sharing. In *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*. 251–260. https://doi.org/10.1007/3-540-47721-7_19

[8] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. 2011. Semi-homomorphic Encryption and Multiparty Computation. In *EUROCRYPT 2011 (LNCS)*, Kenneth G. Paterson (Ed.), Vol. 6632. Springer, Heidelberg, 169–188.

[9] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, and Michele Orrù. 2017. Homomorphic Secret Sharing: Optimizations and Applications. In *To appear*.

[10] E. Boyle, N. Gilboa, and Y. Ishai. 2015. Function Secret Sharing. In *EUROCRYPT*. 337–367.

[11] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2016. Breaking the Circuit Size Barrier for Secure Computation Under DDH. In *CRYPTO*. 509–539. Full version: IACR Cryptology ePrint Archive 2016: 585 (2016).

[12] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2016. Function Secret Sharing: Improvements and Extensions. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. 1292–1303. https://doi.org/10.1145/2976749.2978429

[13] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2017. Group-Based Secure Computation: Optimizing Rounds, Communication, and Computation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 163–193.

[14] Zvika Brakerski and Guy N. Rothblum. 2013. Obfuscating Conjunctions. In *CRYPTO 2013, Part II (LNCS)*, Ran Canetti and Juan A. Garay (Eds.), Vol. 8043. Springer, Heidelberg, 416–434. https://doi.org/10.1007/978-3-642-40084-1_24

[15] Zvika Brakerski and Vinod Vaikuntanathan. 2014. Efficient Fully Homomorphic Encryption from (Standard) LWE. *SIAM J. Comput.* 43, 2 (2014), 831–871. https://doi.org/10.1137/120868669

[16] Ran Canetti. 1997. Towards Realizing Random Oracles: Hash Functions That Hide All Partial Information. In *CRYPTO'97 (LNCS)*, Burton S. Kaliski Jr. (Ed.), Vol. 1294. Springer, Heidelberg, 455–469.

[17] David Chaum, Claude Crépeau, and Ivan Damgård. 1988. Multiparty Unconditionally Secure Protocols (Extended Abstract). In *STOC*. 11–19.

[18] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2016. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *Advances in Cryptology–ASIACRYPT 2016: 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I 22*. Springer, 3–33.

[19] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. 1995. Private information retrieval. In *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on*. IEEE, 41–50.

[20] Richard Cleve. 1991. Towards Optimal Simulations of Formulas by Bounded-Width Programs. *Computational Complexity* 1 (1991), 91–105. https://doi.org/10.1007/BF01200059

[21] Ivan Damgård, Jesper Buus Nielsen, Michael Nielsen, and Samuel Ranellucci. 2016. Gate-scrambling Revisited - or: The TinyTable protocol for 2-Party Secure Computation. *IACR Cryptology ePrint Archive* 2016 (2016), 695. http://eprint.iacr.org/2016/695

[22] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. 2012. Multiparty Computation from Somewhat Homomorphic Encryption. In *CRYPTO 2012 (LNCS)*, Reihaneh Safavi-Naini and Ran Canetti (Eds.), Vol. 7417. Springer, Heidelberg, 643–662.

[23] Yevgeniy Dodis, Shai Halevi, Ron D. Rothblum, and Daniel Wichs. 2016. Spooky Encryption and Its Applications. In *CRYPTO*. 93–122.

[24] Léo Ducas and Daniele Micciancio. 2015. FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second. In *EUROCRYPT*. 617–640.

[25] Sanjam Garg, Craig Gentry, Shai Halevi, and Mariana Raykova. 2014. Two-Round Secure MPC from Indistinguishability Obfuscation. In *TCC*. 74–94.

[26] Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices. In *STOC*. 169–178.

[27] Craig Gentry, Amit Sahai, and Brent Waters. 2013. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*. 75–92. https://doi.org/10.1007/978-3-642-40041-4_5

[28] Satrajit Ghosh, Jesper Buus Nielsen, and Tobias Nilges. 2017. Maliciously Secure Oblivious Linear Function Evaluation with Constant Overhead. *IACR Cryptology ePrint Archive* 2017 (2017), 409. http://eprint.iacr.org/2017/409

[29] Niv Gilboa. 1999. Two Party RSA Key Generation. In *CRYPTO'99 (LNCS)*, Michael J. Wiener (Ed.), Vol. 1666. Springer, Heidelberg, 116–129.

[30] Niv Gilboa and Yuval Ishai. 2014. Distributed Point Functions and Their Applications. In *EUROCRYPT 2014 (LNCS)*, Phong Q. Nguyen and Elisabeth Oswald (Eds.), Vol. 8441. Springer, Heidelberg, 640–658. https://doi.org/10.1007/978-3-642-55220-5_35

[31] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *STOC*. 218–229.

[32] Shai Halevi and Victor Shoup. 2015. Bootstrapping for HElib. In *EUROCRYPT*. 641–670. https://doi.org/10.1007/978-3-662-46800-5_25

[33] Yuval Ishai, Eyal Kushilevitz, Sigurd Meldgaard, Claudio Orlandi, and Anat Paskin-Cherniavsky. 2013. On the Power of Correlated Randomness in Secure Computation. In *TCC 2013 (LNCS)*, Amit Sahai (Ed.), Vol. 7785. Springer, Heidelberg, 600–620. https://doi.org/10.1007/978-3-642-36594-2_34

[34] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. 2008. Cryptography with constant computational overhead. In *40th ACM STOC*, Richard E. Ladner and Cynthia Dwork (Eds.). ACM Press, 433–442.

[35] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. 2009. Secure Arithmetic Computation with No Honest Majority. In *Theory of Cryptography, 6th Theory of Cryptography Conference, TCC 2009, San Francisco, CA, USA, March 15-17, 2009. Proceedings (Lecture Notes in Computer Science)*, Omer Reingold (Ed.), Vol. 5444. Springer, 294–314. https://doi.org/10.1007/978-3-642-00457-5_18

[36] Marcel Keller, Emmanuela Orsini, and Peter Scholl. 2016. MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 830–842. https://doi.org/10.1145/2976749.2978357

[37] Vladimir Kolesnikov and Ranjit Kumaresan. 2013. Improved OT Extension for Transferring Short Secrets. In *CRYPTO 2013, Part II (LNCS)*, Ran Canetti and Juan A. Garay (Eds.), Vol. 8043. Springer, Heidelberg, 54–70. https://doi.org/10.1007/978-3-642-40084-1_4

[38] Eyal Kushilevitz and Rafail Ostrovsky. 1997. Replication is NOT Needed: SINGLE Database, Computationally-Private Information Retrieval. In *38th FOCS*. IEEE Computer Society Press, 364–373.

[39] Pratyay Mukherjee and Daniel Wichs. 2016. Two Round Multiparty Computation via Multi-key FHE. In *Proc. EUROCRYPT 2016*. 735–763. https://doi.org/10.1007/978-3-662-49896-5_26

[40] Moni Naor and Benny Pinkas. 2006. Oblivious Polynomial Evaluation. *SIAM J. Comput.* 35, 5 (2006), 1254–1281. https://doi.org/10.1137/S0097539704383633

[41] Rafail Ostrovsky and William E. Skeith III. 2005. Private Searching on Streaming Data. In *Proc. CRYPTO 2005*. 223–240.

[42] Ronald L. Rivest, Len Adleman, and Michael L. Dertouzos. 1978. On data banks and privacy homomorphisms. In *Foundations of secure computation (Workshop, Georgia Inst. Tech., Atlanta, Ga., 1977)*. Academic, New York, 169–179.

[43] SECG. 2010. SEC 2: Recommended Elliptic Curve Domain Parameters, Version 2. http://www.secg.org. (2010).

[44] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. 2010. Fully Homomorphic Encryption over the Integers. In *Proc. EUROCRYPT 2010*. 24–43.

[45] Andrew Chi-Chih Yao. 1986. How to Generate and Exchange Secrets (Extended Abstract). In *FOCS*. 162–167.