Homomorphic Secret Sharing: Optimizations and Applications*

Elette Boyle**, Geoffroy Couteau***, Niv Gilboa†, Yuval Ishai‡, and 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.

1 Introduction

Fully homomorphic encryption (FHE) [RAD78,Gen09] 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) [Yao86, GMW87, BGW88, CCD88], 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 [HS15, DM15, CGGI16] 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 [vDGHV10,BV14,GSW13], 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. [BGI16a] and further studied in [BGI17]. 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 is a preliminary full version of [BCG+17].

** IDC Herzliya, Israel. Email: eboyle@alum.mit.edu

*** KIT, Germany. Email: geoffroy.couteau@kit.edu. Work mostly done while at ENS Paris.

† Ben-Gurion University, Israel. Email: gilboan@bgu.ac.il

‡ Technion, Israel. Email: yuvali@cs.technion.ac.il. Work mostly done while at UCLA.

§ ENS Paris, France. Email: michele.orru@ens.fr

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 [BGI16a] 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 [BGI16a] has been later optimized in [BGI17], 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 [BGI16a] that we build on.

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 [BGI16a] and its subsequent optimizations from [BGI17], 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.

[1] HSS for general circuits can be based on LWE via multi-key FHE [DHRW16] or even threshold FHE [BGI15, DHRW16]. 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 Appendix B.

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. MPC protocols with a minimal interaction pattern as above cannot be obtained using classical MPC techniques.

Two-round MPC protocols were constructed in a sequence of works under indistinguishability obfuscation [GGHR14] (or witness encryption [GLS15]), special types of fully homomorphic encryption [MW16, DHRW16], and HSS [BGI17, BGI+18]. Very recently, 2-round MPC protocols were constructed via a general "protocol garbling" paradigm under the minimal assumption that 2-round oblivious transfer protocol exists [GS17, GS18, BL18]. However, the latter protocols do not directly apply to the client-server setting considered here, they require a non-black-box use of the underlying cryptographic primitives, and do not enjoy some other efficiency features of HSS-based protocols. See [BGI+18] for discussion.

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 [BGI16a, BGI17].

Optimizations. We introduce several new optimization ideas that further speed up the previous optimized implementation from [BGI17], 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 [BGI17] can reduce the expected computational cost by a factor 16 or more, for the same failure probability. (As in [BGI17], 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 additional machine-level optimizations, we reduce the computational cost of Eval by more than a factor of 30 compared to the optimized implementation from [BGI17].

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. Using bilinear maps, we show how to make HSS ciphertexts extremely succinct (at the cost of a higher evaluation time) by applying a prime-order variant of the Boneh-Goh-Nissim encryption scheme [BGN05].

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 [BGI16a,BGI17]. 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 involving 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 [Ben86], 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) [BGI15], 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 [BGI15, BGI16b]. 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 [BGI16a]. By default, in this work, the term HSS refers to a public-key variant of HSS (known as

DEHE in [BGI16a]), with a Las Vegas correctness guarantee. To enable some of the optimizations we consider, we use here a slight variation of the definition from [BGI17] that allows for an output to be computed even when one of the two parties suspects an error might occur.

Definition 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* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval})$ *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 $\gamma_b \in \{\bot, \top\}$ to indicate full confidence ($\top$) or a possibility of failure ($\bot$). When β is omitted it is understood to be $\beta = 2$.*

The algorithms $\mathsf{Gen}, \mathsf{Enc}$ *are PPT algorithms, whereas* Eval *can run in time polynomial in its input length and in $1/\delta$. The algorithms* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval})$ *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:*

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

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 [BGI16a] for discussion.)*

Remark 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 [Cle91, BGI16a].

Definition 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) [BGI16a]. 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 4.6.

3 Overview of Group-Based HSS

In this section we give a simplified overview of the HSS construction from [BGI16a]. 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, for conjectured 80 bits of security.) 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 $\mathsf{Convert}$ 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 $\mathsf{Convert}$ algorithm works as follows. Each party, on input $h \in \mathbb{G}$, outputs the minimal integer $i \geq 0$ such that $h \cdot g^i$ is "distinguished," where roughly a δ-fraction of the group elements are distinguished. Distinguished elements were picked in [BGI16a] by applying a pseudo-random function to the description of the group element. An optimized conversion procedure from [BGI17] (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 $\mathsf{Convert}$ holds if no group element *between* the two shares $\{z\} \in \mathbb{G}^2$ is distinguished. Finally, $\mathsf{Convert}$ 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 $\mathsf{Convert}$ and for the bound on the memory values in the definition of RMS programs (Definition 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 $\mathsf{PairConv}$ be an algorithm that sequentially executes these two operations: $\mathsf{PairConv}([\![x]\!]_c, \langle\langle y \rangle\rangle_c, \delta) \mapsto \langle xy \rangle$, with error δ. We denote by Mult the following algorithm:

- $\mathsf{Mult}([\![\![x]\!]\!]_c, \langle\langle y \rangle\rangle_c, \delta) \mapsto \langle\langle xy \rangle\rangle_c$
  - Parse $[\![\![x]\!]\!]_c$ as $([\![x]\!]_c, ([\![x \cdot c_i]\!]_c)_{1 \leq i \leq s})$.
  - Let $\langle xy \rangle \leftarrow \mathsf{PairConv}([\![x]\!], \langle\langle y \rangle\rangle_c, \delta')$ for $\delta' = \delta/(s+1)$.
  - For $i = 1$ to s, let $\langle xy \cdot c_i \rangle \leftarrow \mathsf{PairConv}([\![xc_i]\!]_c, \langle\langle y \rangle\rangle_c, \delta')$.
  - 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 $\mathsf{pk} = (\mathbb{G}, g, h, [\![c_i]\!]_c)_{1 \le i \le s}$, 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 $[\![x]\!]_c$.
- 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 $Y_j \leftarrow \mathsf{Mult}([\![x_i]\!]_c, \langle\!\langle 1 \rangle\!\rangle_c, \delta/S)$, 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 $Y_k \leftarrow \mathsf{Mult}([\![x_i]\!]_c, Y_j, \delta/S)$.
  - $(\beta, \hat{O}_j \leftarrow \hat{y}_i)$: 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 $\mathsf{Convert}$ raises a potential error flag, otherwise it is $\top$.

The pseudorandomness required for generating the outputs and for $\mathsf{Convert}$ 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 $[\![x]\!]_c$ 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 $\mathsf{PairConv}$, where each invocation consists of Pair and $\mathsf{Convert}$. The cost of Pair is dominated by one group exponentiation with roughly 200-bit exponent for 80 bits of security. (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 $\mathsf{Convert}$ is linear in BS/δ, where the B factor comes from the fact that the payload z of $\mathsf{Convert}$ 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 Optimizing Share Conversion

In [BGI17], the share conversion algorithm $\mathsf{Convert}$ (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 [BGI16a], the method proposed in [BGI17] 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 [BGI17] 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 [BGI16a], 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.

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 [BGI17], 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).

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 $\mathsf{Convert}$ algorithm from [BGI17] (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, where 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 [BGI17] for a payload z and payload bound M:

Lemma 4. *If* $\mathsf{Convert}^*$ *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* $\varepsilon \leq z \cdot 2^{-(d+1)}$*.*

Proof. The expected number of steps required for finding the first occurrence of 0^d in a random bit-sequence stream is $X = 2^{d+1} - 2$, see, e.g., [Nie73] or Appendix A.2 for a self-contained proof. As it is clear that $T \leq X + 2M$, where the additive $2M$ term captures the cost of checking for distinguished points within $2M$ steps backward, the first part of the lemma follows. Let us now bound ε. Let E_i be the event of party i raising a flag. The first party raises a flag if it finds a distinguished point within M steps of his point, which happens with probability $\Pr[E_1] \leq M/2^{d+1}$. Let t_1 be the number of steps performed by the first party before reaching a distinguished point. The second party raises a flag if it finds a distinguished point within M steps backward from his point; we have

$$\begin{aligned}
\Pr[E_2 \mid E_1] &= \Pr[E_2 \mid t_1 \leq M] && (1)\\
&= \Pr[E_2 \wedge (t_1 \leq z) \mid t_1 \leq M] + \Pr[E_2 \wedge (t_1 > z) \mid t_1 \leq M] && (2)\\
&= \Pr[t_1 \leq z \mid t_1 \leq M] + 0, && (3)
\end{aligned}$$

where Equation 3 comes from the fact that when $t_1 \leq z$, as $z \leq M$, the second party necessarily raises a flag; when $t_1 > z$, as we ensure that the distance between two distinguished points is at least $2M$, there cannot be a distinguished point within M steps backward of the second party. Therefore, we get $\Pr[E_1 \mid E_2] = \Pr[t_1 \leq z \mid t_1 \leq M] = z/M$, hence $\varepsilon \leq M/2^{d+1} \cdot z/M = z/2^{d+1}$, which concludes the proof. □

For comparison, in the Las Vegas variant of the optimized conversion algorithm from [BGI17], 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, and denoting Mult^* the Mult algorithm using $\mathsf{Convert}^*$ instead of $\mathsf{Convert}$, we can now bound the failure probability in an RMS multiplication:

Lemma 5. *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*

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

Note that the payload in the first $\mathsf{Convert}^*$ algorithm is y and the average payload in the s last $\mathsf{Convert}^*$ invocations is $(B-1)y/2$; the failure probability is also taken over the random choice of the secret key.

Proof. The expected running time directly follows from Lemma 4. By a union bound, ε is bounded by $p_0 + p_1$, where p_0 denotes the probability of both parties raising a flag during two different conversions (recall that an RMS multiplication requires $s+1$ conversions), and p_1 denotes the probability of both parties raising a flag during the same conversion algorithm.

The probability for a party to raise a flag in a given conversion is $1/2^{d+1}$, independently of any other information. By the union bound, the probability of raising a flag during an RMS multiplication is bounded by $(s+1)/2^{d+1}$. As different conversions can be made perfectly independent, we can therefore bound p_0 by $((s+1)/2^{d+1})^2$.

By Lemma 5, the probability of both party raising a flag in the first conversion (where the payload is y) is bounded by $y/2^{d+1}$, and the probability of both parties raising a flag in any of the remaining conversions (where each payload is $y \cdot c_i$) is $yc_i/2^{d+1}$. By a union bound, we can upper bound p_1 by $y \cdot (1 + \sum_i c_i)/2^{d+1}$. As the secret key c is uniformly random over $\{0, \cdots, B-1\}^s$, the expected value of $\sum_i c_i$ is $s(B-1)/2$, which concludes the proof. □

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 $\mathsf{Convert}^*$; 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 [BGI16a], 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 $\mathsf{PairConv}^*$ be the $\mathsf{PairConv}$ algorithm modified to use the new $\mathsf{Convert}^*$ 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 $([\![x]\!]_c, ([\![xc_i]\!]_c)_{i \le s})$, and use the public values (b_0, b_1) to compute $[\![b_0 \oplus x]\!]_c$, $([\![(b_0 \oplus x)c_i]\!]_c)_{i \le s}$, and $\langle\!\langle b_1 \oplus y \rangle\!\rangle_c$. 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 5:

Corollary 6. *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*

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

Remark 7. The above method should be avoided when there is an a-priori knowledge that the RMS values are biased towards 0 (or 1). In this case, one can gain better error reduction by applying our optimized conversion directly without any randomization. We also note that 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

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

which is sufficient to improve over the basic $\mathsf{Convert}$ algorithm.

4.2 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 [BGI17], 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).

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 [BGI17], 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.

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 $\left((g^{a_i})g^{(r_i-a_i)}, (g^{a'_i})g^{(x_i-a'_i)}\right)_{i\in[s]}$.

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

Proposition 8. *The protocol Π_{Gen} in Figure 1 securely evaluates the group-based HSS* Gen *algorithm (from Section 3.3).*

Proof (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. 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 [BGI17]. Π_{Gen} requires the additional 2PC execution and $2s$ additional exponentiations per party (from Step 3) over the [BGI17] 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 [BGI17] 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.

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 $(0, -r)$.

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

Proposition 9. *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.

4.3 Compact Ciphertexts from BGN

An issue of the group-based homomorphic secret sharing scheme of [BGI16a, BGI17] is the large size of the ciphertexts. Two natural optimizations in this regard have been already outlined in [BGI16a, BGI17]: the first is to assume the circular security of the ElGamal encryption, leading to a construction with ciphertexts considerably shorter than the original construction of [BGI16a], and the second is to use a larger basis B for the secret key, reducing the ciphertext size by a factor $\log_2(B)$. In addition, another heuristic optimization was suggested in [BGI16a] to further reduce the size of the

ciphertexts by a factor two, by generating all the first components g^r of each ciphertext from a short seed, using a pseudorandom generator. This optimization applies only in the secret key setting, as one need to know the secret key to generate ciphertexts this way.

The ciphertext size of group-based HSS can be strongly reduced using the prime order variant of the Boneh-Goh-Nissim encryption scheme [BGN05, Fre10], which enhances the ElGamal encryption scheme with one level of multiplicative homomorphism using an asymmetric bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \mapsto \mathbb{G}_t$.

The large ciphertext size of group-based HSS schemes essentially comes from the necessity to encrypt all products $x \cdot c_i$ for elements c_i of the secret key, for each input x. We observe that the ciphertexts can be made considerably more succinct by relying on the Boneh-Goh-Nissim (BGN) encryption scheme [BGN05], which extends the homomorphic properties of the ElGamal encryption scheme to support one level of multiplications in composite-order elliptic curves with a bilinear map. While bilinear maps over composite-order elliptic curves are known to be very inefficient, the BGN encryption scheme can be instantiated over prime-order elliptic curves with an asymmetric bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \mapsto \mathbb{G}_t$, using the transformation of [Fre10]. The transformed scheme allows to encrypt exponents in either $\mathbb{G}_1^2$ or $\mathbb{G}_2^2$, and to map pairs of ciphertexts $(c_1, c_2) \in \mathbb{G}_1^2 \times \mathbb{G}_2^2$ encrypting plaintexts (m_1, m_2) to a ciphertext $c_t \in \mathbb{G}_t^4$ encrypting $m_1 m_2$. Decryption in $\mathbb{G}_t^4$ follows the same "linear algebra in the exponent" structure that the ElGamal encryption scheme, which allows to implement the Pair algorithm in this target group in a natural way. While the secret key in the abstract framework of [Fre10] is represented as a tensor product between two 2×2 matrices, and hence consists of 16 exponents, it directly follows from the description of the scheme given in [Fre10] that the first column of this tensor product (a vector of 4 exponents) is in fact sufficient to perform decryption.

This gives rise to an HSS scheme with extremely compact ciphertexts, at the expense of a bigger public key, and having to perform the $\mathsf{PairConv}$ operation in the target group $\mathbb{G}_t$ (in which operations are more expensive). We have the following lemma:

Lemma 10. *Let $(\mathbb{G}_1, \mathbb{G}_2)$ be prime-order groups equipped with an asymmetric bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \mapsto \mathbb{G}_t$, such that the prime-order variant of the BGN encryption scheme is circularly secure over $(\mathbb{G}_1, \mathbb{G}_2)$. 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 $2(s+1)$ elements of $\mathbb{G}_2$, where s is the length of the BGN secret key (which is 4 times longer than a standard ElGamal secret key).*
- *The ciphertexts consists of 2 group elements over $\mathbb{G}_1$.*
- *The Pair algorithm requires four exponentiations and 3 multiplications over $\mathbb{G}_t$.*
- *A level two share of a value x consist of 4 shares $(\langle x \cdot a_i \rangle)_{i \leq 4}$, where (a_1, a_2, a_3, a_4) is the first column of the tensor product between the two secret-key matrices.*

The public key of the scheme is the public key of a BGN scheme, together with encryptions in $\mathbb{G}_2^2$ of the components of the secret key. An input x is encrypted using the encryption algorithm in $\mathbb{G}_1^2$. Given an encryption of an input x, the parties first extend it by homomorphically computing encryptions in $\mathbb{G}_t$ of $x \cdot c_i$ for $i = 1$ to s using the asymmetric bilinear map between $\mathbb{G}_1$ and $\mathbb{G}_2$ (this step is only performed once per encrypted input).

Randomness Reuse. A common method to reduce the size of ElGamal ciphertexts in exchange for a larger public key is to use randomness reuse, which was introduced

in [Kur02] in the context of multi-recipient encryption schemes: instead of having a short public key (g, h) and encrypting a vector $(x_1, \cdots, x_n)$ as $(g^{r_i}, h^{r_i} g^{x_i})_{i \leq n}$, which requires $2n$ group elements, one can have a large public key $(g, h_1, \cdots, h_n)$ and encrypt $(x_1, \cdots, x_n)$ as $(g^r, (h_i^r g^{x_i})_{i \leq n})$, which requires only $n + 1$ group elements. This method can be used to further reduce the ciphertext size in the BGN-based HSS scheme: increasing the public key size (as well as the amount of computation in pairing operations) by a factor n allows to reduce the size of ciphertexts from 2 group elements to $1 + 1/n$ group elements (asymptotically), saving almost a factor 2. Overall, if one is willing to trade computational efficiency for succinctness, HSS ciphertexts can be made as small as about 200 bits.

4.4 Generic Ciphertext Compression for Public-Key HSS

While we can apply randomness reuse to reduce the size of the ciphertexts in the BGN-based HSS, in many applications one might want to work in groups where conversions will be computationally more efficient (such as the conversion-friendly groups discussed in [BGI17]), and where bilinear maps are not available. However, in such groups, the randomness reuse method cannot be used as each ciphertext is of size proportional to the number of components of the secret key. Having a large number of secret keys would therefore blow up their size. In [BGI16a], 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 of 2 in exchange for a larger key) for public-key HSS in groups that are not equipped with bilinear maps (such as conversion-friendly groups [BGI17], or standard elliptic curves) under a new assumption that we introduce below.

Entropic Span Diffie-Hellman Assumption. Let $\bullet$ denote the inner product operation, and let $B \geq 2$ be a basis. Given a group family $\mathbb{G} = \mathbb{G}(\lambda)$ where $|\mathbb{G}(\lambda)| = q(\lambda)$, the entropic span Diffie-Hellman assumption (ESDH) states the following.

Assumption 11 *For all polynomials $t = t(\lambda), k = k(\lambda)$, and sequence of vector sets $\mathbf{v}_\lambda = \{v_1, \cdots, v_k\} \in (\mathbb{Z}_q^t)^k$ the following holds. Suppose that for any non-zero v in the span of $\mathbf{v}_\lambda$ it holds that $H_\infty(X_v) \geq \omega(\log \lambda)$, where H_∞ denotes min-entropy and X_v is the distribution of $v \bullet c$ for $c \xleftarrow{\$} \{0, \cdots, B-1\}^t$ where the inner product is taken modulo q. Then the following distributions are computationally indistinguishable:*

$$D_0 = \{v_1, \cdots, v_k, g, g^{v_1 \bullet c}, \cdots, g^{v_k \bullet c} \mid c \xleftarrow{\$} \{0, \cdots, B-1\}^t\}$$
$$D_1 = \{v_1, \cdots, v_k, g, g_1, \cdots, g_k \mid (g_1, \cdots, g_k) \xleftarrow{\$} \mathbb{G}^k\}$$

Note that a necessary condition for this assumption to hold is that all non-zero vectors in the span of $v_1, \cdots, v_k$ 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 12. *(Generic Security of ESDH) The entropic span Diffie-Hellman assumption holds in the generic group model.*

A proof of Lemma 12 is sketched in Appendix A.

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

Lemma 13. *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*

Proof (Sketch). The HSS scheme is constructed as previously, with the following modifications: the secret key is a vector $c = (c_i)_{i \leq t}$. The public key now contains k vectors $(v_1, \cdots, v_k) \in (\mathbb{Z}_q^t)^k$ (which can be heuristically 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 $(x, (x \cdot c_i)_{i \leq t})$. We modify level 2 shares to be of the form $(\langle y \rangle, (\langle c_i \cdot y \rangle)_{i \leq t})$ (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 $(\llbracket x \rrbracket_c, \langle\!\langle y \rangle\!\rangle_c)$, the parties compute $\langle v_j \bullet c \rangle_{j \leq q}$ 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.

Reduced Computation for a Given Memory. The randomness reuse method does not only reduces the size of HSS ciphertexts, but also improves the computational efficiency for a given amount of available memory. Evaluating RMS programs involves a large number of exponentiations; more precisely, two kinds of exponentiations are required: exponentiations with a small exponent (involving the second component of an ElGamal ciphertext with the first component of a level 2 share, which is a small integer), and exponentiations with large exponents (involving the first component of an ElGamal ciphertext with the second component of a level 2 share, which is an integer of size at least s bits). To reduce the amount of computation during exponentiations, a standard technique is to use precomputation. A lookup table of size $O(n)$ allows to reduce the amount of computation required for an exponentiation by $O(\log n)$. However, this can require a lot of memory, as a table must be precomputed for each first component of ElGamal ciphertexts. An important advantage of the randomness reuse method is that the same first component of an ElGamal ciphertext, which is the one involved in exponentiations with large exponents, will be reused across many ciphertexts; hence, a single precomputed lookup can be used to speed up exponentiations in many ciphertexts. For a given maximal amount of available memory, this allows to precompute larger lookup tables and provides an important speedup.

4.5 Reducing the Leakage Rate

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.

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 [BGI17] 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 $\mathsf{Convert}(\{xy \oplus b\})$ to get $\langle xy \oplus b \rangle$.

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. $\mathsf{MMult}_{\mathsf{pk}}(\llbracket\!\llbracket x \rrbracket\!\rrbracket_{c,c}, \langle\!\langle b \rangle\!\rangle_c, \langle\!\langle y \oplus b \rangle\!\rangle_c) \mapsto \langle\!\langle xy \oplus b \rangle\!\rangle_c$
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 $\llbracket\!\llbracket x \rrbracket\!\rrbracket_{c,c}$ 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 $(\llbracket x_i \rrbracket_c)_{i \le d}$, which outputs a level-2 share of $P(x_1, \cdots, x_d)$:

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

The algorithm 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 intermediate values 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 can be done using two oblivious transfers: the two parties (P_0, P_1) hold shares (c_0, c_1) of the secret key c, and pick respective random bits (b_0, b_1). With one OT, P_0 transmits $r_0 - 2b_0b_1$ and $(c_0 - 2b_0c_0)b_1 + r'_0$ to P_1, for random $(r_0, r'_0) \in \mathbb{Z}_q^2$, by letting P_1 choose between the pairs (r_0, r'_0) and $(-2b_0 + r_0, c_0(1 - 2b_0) + r'_0)$ with selection bit b_1. Conversely, P_1 transmits $c_1(1 - 2b_1)b_0 + r_1$ to P_0, for a random $r_1 \in \mathbb{Z}_q$, using one OT. Note that $(r_0 + b_0, b_1 + r_0 - 2b_0b_1)$ form additive shares of $b_0 \oplus b_1$, and $(b_0c_0 - r'_0 + c_1(1 - 2b_1)b_0 + r_1, b_1c_1 - r_1 + c_0(1 - 2b_0)b_1 + r'_0)$ form additive shares of $c \cdot (b_0 \oplus b_1)$. Therefore, the two players obtain level 2 shares of a random bit.

Protecting the Secret Key. Leakage pads can be used to equally reduce the leakage rate of both input bits and secret key bits. However, protecting key bits is more important for two reasons. First, key bits are typically involved in a much larger number of conversions than input bits. Second, in applications that involve distributed key generation, replacing a compromised key requires additional interaction. A natural approach suggested in [BGI17] for protecting the secret key c is to split it into k random additive shares $c^i \in \mathbb{Z}_q$ such that $c = \sum_{i=1}^{k} c^i$, and modify the level 1 share of an input x to include encryptions of $x \cdot c^i_j$ for $i \in [k]$ and $j \in [s]$. This ensures that the jth component of c remains unknown unless the k components at the jth positions of the $(c^i_j)_{i \le k}$ 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 [Can97, BR13]). 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 $(v_i)_{i \leq \ell}$ of elements of $\mathbb{Z}_q$ is added to the public key (it can be heuristically compressed to a short seed using a PRG), as well as encryptions of random bits $(c'_i)_{i \leq \ell}$ satisfying $\sum_i c'_i v_i = c \bmod q$. An HSS ciphertext for an input x now encrypts $(x, (xc'_i)_i)$. After an invocation of $\mathsf{Convert}$ 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 is more efficient than the previous one for large values of k and offers unconditional security; 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.

4.6 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.

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 $\mathsf{PairConv}$: 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 $\mathsf{PairConv}$ 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 4.2.

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.

Evaluating Branching Programs and Formulas. As pointed out in [BGI16a], 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 [BGI16a].

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.

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 fuzzy matches with encrypted queries.

5 Applications

In this section, we outline a number of natural scenarios in which group-based HSS seems particularly attractive, and describe how the optimizations from the previous section apply in these scenarios. The efficiency estimates given in this section are based

on the running time of our implementation, described Section 7 (see Remark 24), using 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 roughly 80 bits of security. We summarize 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 4.5) and ciphertext compression (Section 4.4) 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)	Correlations (5.3)
Share Conversion (4.1)	✓	✓	✓	✓	✓
Rand. Conversion (4.1)	✓	✓	✓	✓	✓
Key Generation (4.2)	✓	✗	✗	✗	✓
Compression (4.4)	✓	✓	✗	✗	✗
Leakage (4.5)	✓	✓	✗	✓	✗
Terminal Mult. (4.6)	✓	✓	✓	✓	✓
Large Inputs (4.6)	✓	✓	✗	✗	✗

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 servers, with a simple interaction pattern that involves a message from each input client to each server and a message from each server to each output client. Security should hold as long as the two servers do not collude, and should hold even if when an arbitrary set of clients colludes with one server. HSS provides a natural solution in this scenario. Before the set of clients or their inputs are known, the two servers S_b obtain a common public key pk and local secret evaluation keys ek_b. This (reusable) setup can be implemented via a trusted dealer or via an interactive protocol between the servers or another set of parties. (When the setup is implemented using external parties, the servers do not ever need to interact or know each other's identity.) The clients, who do not need to know which or how many other clients participate in the protocol, can compute a program P on their inputs x_i in the following way.

- Upload inputs. Each client C_i with input x_i computes $\mathsf{ct}_i \leftarrow \mathsf{Enc}(\mathsf{pk}, x_i)$ and sends ct_i to both servers. (Alternatively, the encrypted inputs ct_i can be posted in a public repository and downloaded by the servers.)
- Local evaluation. Each server S_b, given encrypted inputs $(\mathsf{ct}_1, \cdots, \mathsf{ct}_n)$ and a program P, locally computes $z_b \leftarrow \mathsf{Eval}(b, \mathsf{ek}_b, (\mathsf{ct}_1, \cdots, \mathsf{ct}_n), P, \delta)$, where δ is a given failure probability bound.

– Download output. Each server S_b sends z_b to each output client. (Alternatively, z_b can be made public if the output is public.) The output $P(x_1, \ldots, x_n)$ is recovered, except with δ failure probability, by computing $z \leftarrow z_0 \oplus z_1$.

A simple example for this kind of secure computation can be a small-scale vote: multiple clients encrypt their vote and upload them on a public repository. The two servers retrieve the encrypted votes and evaluate the voting function (say, majority, conjunction, or another threshold function), without having to interact. The local nature of this computation mitigates risks of collusions and reduces latency. Shares of the result of the vote are then sent to the clients, who can reconstruct the result by performing a simple XOR. In case of a failure, the vote can be recomputed using the same encrypted inputs.

Managing the Leakage. Note that the event of a failure in our group-based HSS constructions is correlated with both the private inputs of the clients and the secret evaluation keys. 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 events can be acceptably rare.

To further mitigate the risk associated with such leakage, the parties can use the techniques described in Section 4.5 to reduce the dependency between a failure event and a leakage event. The key randomization techniques can be used to ensure that the same setup can be used for many computations without compromising the secret key. Moreover, leakage-absorbing pads can be generated as part of the distributed key setup to protect inputs encrypted with this setup, where Eval is replaced by the MaskedEval algorithm. To minimize the number of pads, the same pad can be used in each computation until one of the servers detects possible failure; 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. Note that while communication between the servers may still be occasionally required for generating new leakage pads, such an interaction will typically be very infrequent and has a small amortized cost.

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 HSS-based approach has the advantage of being particularly efficient for the clients, without requiring interaction between

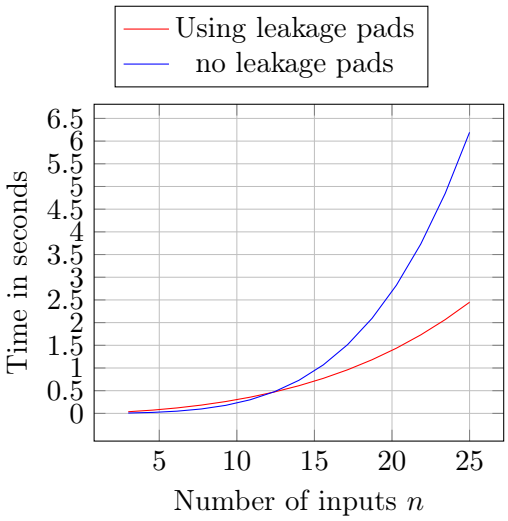

Fig. 3. Time to compute majority of n inputs with 10^{-4} leakage probability, with and without leakage-absorbing pads, on a single thread of an Intel Core i7 CPU. See Remark 24 for further implementation details.

the servers (or requiring infrequent interaction for refreshing secret key or leakage absorbing pads). Standard alternative techniques for performing secure computation in this setting break down if a client colludes with one of the servers. For instance, this applies to solutions where one of the servers generates a garbled circuit, or to solutions that employ a standard FHE scheme whose secret key is known to all clients.

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.

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 policy.

Basic Setting. A 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 4.6 to allow the program P to handle the large input r in a terminal multiplication. In practice, this will correspond to setting $K \leftarrow R(g^r)$, where g is part of the public key pk and $R(\cdot)$ is a randomness extractor, evaluating Pair between $\mathsf{Enc}(\mathsf{pk}, r)$ and $\langle\!\langle b \rangle\!\rangle_c$, where b is a bit that determines whether access should be granted, and outputting the result $\{b \cdot r\} = (v, vg^{rb})$ for some group element v, without performing the final $\mathsf{Convert}$ procedure. From the multiplicative shares of g^{rb}, the client can recover $K = R(g^r)$ (and therefore access the file m) if and only if $b = 1$. This basic scenario is represented in Figure 4.

Enhanced Scenarios. The key reconstruction in the above procedure will fail with some probability δ even for an authorized client. While the client can then simply ask the

servers to re-run the authorization procedure, this reveals that a failure occurred, which leaks information on the encrypted permission vector, as well as on the secret key. A simple solution to that is to set the failure parameter to be small enough, so that a very large number of messages can be delivered before the leakage accumulates and the data owner must refresh the setup.

A better solution is to use the leakage-absorbing pads introduced in Section 4.5. The data owner simply generates some number of leakage-absorbing pads and sends them to the servers. The two servers use each leakage-absorbing pad for some fixed number of computations, and throw them away afterward. These pads are very easy to generate (they essentially consist in either random shares of zero, or random shares of the secret key), so the data owner can just regularly regenerate some pads and send them to the server. That way, the key setup must be refreshed only when too many leakage-absorbing pads have been involved in two simultaneous failures.

The protocol can easily be enhanced to allow for multiple data owners, each providing their own permission vector, by using public-key HSS instead of secret-key. In that case, the data owner can use the distributed key generation protocol described in Section 4.2, as well as generate leakage-absorbing pads using standard interactive protocols.

Efficiency Estimations. A leakage-absorbing pad is about 200 bit long. Assuming that the policy is, e.g., sending files subject to approval of two managers (conjunction of two encrypted bits – considering the simplest possible scenario), answering a file request with failure probability 10^{-6} takes about 0.05 seconds on a server running with 32 cores. Assuming an initial pool of a thousand leakage-absorbing pads (of total size 25kB), where each pad is used for answering 10 requests, the pool must be recharged after answering about 10^5 requests (which requires a single 25kB message from the data owner), and the key setup must be refreshed only after 10^{10} request answers.

Policy-Based File System for a data oner D and two servers (S_0, S_1):

Setup: D runs $(\mathsf{pk}, \mathsf{ek}_0, \mathsf{ek}_1) \xleftarrow{\$} \mathsf{Gen}(1^\lambda)$ and sends $(\mathsf{pk}, \mathsf{ek}_b)$ to each server S_b.

File Upload: Pick a large value r and compute $\mathsf{ct} \leftarrow \mathsf{Enc}(\mathsf{pk}, r)$. Upload $(\mathsf{ct}, E_K(m))$, where m is the file to upload, E is a symmetric encryption algorithm, and K is a key derived from $\{r\}$ via some randomness extraction procedure.

Permission Upload: D sends to (S_0, S_1) an encrypted vector ct_p corresponding to some permission.

Query Answer: For all possible pairs (ct, a), where a is a public vector of attributes associated to some client C, each server S_b precomputes $K_b \leftarrow \mathsf{Eval}(b, \mathsf{ek}_b, \mathsf{ct}_p, \mathsf{ct}, P[a], \delta)$, where $P[a]$ is a policy function with a hardcoded whose output allows to reconstruct K depending on the permission vector. When C wants to access the file m, he retrieves $E_K(m)$ from a public repository, and each server S_b sends him K_b.

Fig. 4. Policy-Based File System Protocol.

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 5. 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 [GI14] 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.

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 4.4.

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 [GI14] (DPF), which can be implemented very efficiently using block ciphers such as AES [BGI16b], by letting the servers match the

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 $(r_i)_{i \leq n}$ hardcoded 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 $(\gamma_j^0, \gamma_j^1) \neq (\perp, \perp)$.

Fig. 5. Private RSS Feed Protocol.

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.

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 [CGKS95, KO97]. 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: 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 $(m_i^b)_{i \in [N]}$ be the output shares of each server S_b obtained by matching the encrypted query with each vector of attributes a_i from the database. Let $(m_i)_{i \in [N]}$ be the corresponding outputs. Each server S_b interprets his shares $(m_i^b)_{i \leq N}$ 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 each item m_i for which a failure occurred will be delivered except with 2^{-k} 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 on the number of such entries, using a suitable linear sketch, each sending a vector v^b, $b \in \{0, 1\}$, to the client. For instance, using the syndrome of a Reed-Solomon code we have $v^b \in \mathbb{F}^{2n}$ and using the power sum method from [CBM15] we have $v^b \in \mathbb{F}^n$. Alternatively, if there is no publicly known bound n on the number of matches, a sample of matching items can be obtained by repeating the above procedure using successive powers of 2 as guesses for the bound. Concretely, for $n = 1, 2, 4, 8, ..., N$, the servers use 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., [OS05]).

The generalized PIR protocol described above provides a tunable tradeoff between communication and computation. Concretely, if the HSS failure probability is set to, say 1%, this results in a similar expected number of false matches, which requires increasing the bound n by roughly $N/100$. This can still provide a good enough compression rate in practice. Decreasing the failure probability results in better compression rate but proportionally higher running time.

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 4.2), yielding keys $(\mathsf{pk}, \mathsf{ek}_0, \mathsf{ek}_1)$.

Bilinear Form Correlations. Consider the following 2-party "bilinear form" correlation, parameterized by abelian groups G_x, G_y, G_z and a bilinear map $M : G_x \times G_y \to G_z$. In the correlation, Party A holds a random $x \in G_x$, party B holds a random $y \in G_y$, and the parties hold additive secret shares (over G_z) of the image $M(x, y) \in G_z$. 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. Consider for simplicity $G_x = G_y = G$ (the protocol extends straightforwardly). The parties will begin with random *bit*-strings $a, b \in \{0,1\}^m$, for m somewhat larger than $\log |G|$, and generate the correlation via two primary steps.

First: Shares of pairwise the $a_i b_j$ products can be computed via m^2 *terminal* RMS multiplications, using the procedure described in Section 4.2 for "loading" the inputs b_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 4.6 (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 Party A's bits and to perform the OT-based protocol for encoding the bits of Party B.

For correctness, after the first step, the parties exchange and discard indices $i, j \in [m]$ with error. However, this may leak information on a subset of non-discarded values a_i, b_j.

Second: The (partly leaked) $a, b \in \{0,1\}^m$ bits are converted to random G elements x, y, while removing the effects of leakage, by taking the corresponding subset sums of fixed public random G elements. The output shares of $M(x, y)$ can then be computed locally from shares of the $\{a_i b_j\}$, relying on bilinearity of M.

More explicitly, consider the following protocol, for $m = \lceil \log |G| \rceil + 4\sigma + E + L$, where $\sigma = 40$ is statistical security parameter, L, E are chosen parameters, and $(r_i)_{i\in[m]}, (s_j)_{j\in[m]} \in G$ are randomly chosen public parameters.

1. Each party samples a respective vector, $a, b \leftarrow \{0,1\}^m$.
2. Party A encodes his input a bitwise using HSS: i.e., $\forall i \in [m], \mathsf{ct}_i^a \leftarrow \mathsf{Enc}(\mathsf{pk}, a_i)$, and sends the resulting ciphertexts $(\mathsf{ct}_i^{a_i})_{i\in[m]}$ to Party B.
3. The parties run the OT-based protocol described in Section 4.2 (Figure 2) to load Party B's input b bitwise into HSS memory as level 2 encodings.
4. Locally, each party runs Las Vegas homomorphic evaluation of the RMS program P_{bilin} that computes m^2 RMS multiplications between input value a_i and memory value b_j, for each $i, j \in [m]$, and outputs the value modulo $\beta = q$ (DDH group modulus). The error for each multiplication is set to E/m^2. Each result is $\mathsf{share}_{i,j} \in \mathbb{Z}_q \cup \{\perp\}$.
5. Party B: Let $\mathsf{Err} = \{(i,j) : \mathsf{share}_{i,j}^B = \perp\}$. Send Err to party A. Let $\mathsf{Err}_a, \mathsf{Err}_b$ be the respective projections of Err onto the 1st and 2nd coordinate.
6. (Discard errs): Locally, for every $i \in \mathsf{Err}_a$ and $j \in \mathsf{Err}_b$: Party A sets $a_i = 0$, $\mathsf{share}_{i,j}^A = 0$, Party B sets $b_j = 0, \mathsf{share}_{i,j}^B = 0$.
7. Party A: Output $x = \sum_{i\in[m]} a_i r_i \in G$ and $(M(x,y))^A = \sum_{i,j\in[m]} M(r_i, s_j)(\mathsf{share}_{i,j}^A) \in G_z$ (in G_z as $\mathbb{Z}$-module).
   Party B: Output $y = \sum_{j\in[m]} b_j s_j \in G$, and corresponding $(M(x,y))^B \in G_z$. Note each $M(r_i, s_j)$ is publicly computable.

Correctness: By Las Vegas correctness (Definition 1), with overwhelming probability $(1/q)$ the HSS shares of $a_i b_j$ for all kept positions $i \notin \mathsf{Err}_a, j \notin \mathsf{Err}_b$ each satisfy $\mathsf{share}_{i,j}^A = \mathsf{share}_{i,j}^B + a_i b_j$ over the *integers* $\mathbb{Z}$ (instead of just $\mathbb{Z}_q$). Suppose this is the case. Then we have $M(x,y) = M(\sum_i a_i r_i, \sum_j b_j s_j) = \sum_{i,j} a_i b_j M(r_i, s_j)$ (over G_z) $= \sum_{i,j}(\mathsf{share}_{i,j}^A - \mathsf{share}_{i,j}^B) M(r_i s_j) = (M(x,y))^A - (M(x,y))^B$.

Secrecy: Entropy loss in a, b comes from (i) discarding erred positions (Step 6), and (ii) leakage on non-discarded b_j values from learning Err. For (i): HSS correctness gives $|\mathsf{Err}| \le E + \sigma$ except with probability $\sim 2^{-\sigma}$. For (ii): leaked values are restricted to b_j for which $j \notin \mathsf{Err}_b$ but $\mathsf{share}_{i,j}^A = \perp$ for some $i \in [m]$ ("danger zone" but no error), also bounded in number by $L + \sigma$ $(= E + \sigma)$ with probability $\sim 2^{-\sigma}$. So, conditioned on Party A's view, b maintains min-entropy $\lceil \log |G| \rceil + 2\sigma$ (and vice versa for a). Thus, the linear combinations $x = \sum_i a_i r_i$ and $y = \sum_j b_j s_j$ are $2^{-\sigma}$-close to uniform over G, conditioned on the public r_i, s_j values and view.

Communication: $640 \times m$ bits for the input-encoding OTs for Party B (see Section 4.2), plus m ElGamal ciphertexts for Party A, which correspond to $2m$ group elements (each 1536 bits). In total, $640 \times m + 1536 \times 2m = 3712m$ bits. (Note that the random elements $r_i, s_j \in G_z$ can be generated pseudorandomly from a short shared seed and need not be directly communicated.)

Computation: We focus on the required cryptographic operations (e.g., dominating the subset sums over G). The local HSS evaluation runtime corresponds to m^2 *terminal* RMS multiplications, i.e. m^2 total exponentiations and share conversions. Each terminal multiplication is performed with failure probability E/m^2.

Applications of Bilinear Form Correlations. This bilinear correlation distribution can aid the following sample applications.

Generating Beaver triples over rings. Beaver triple correlations [Bea95] 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., [Bea92, BDOZ11, DPSZ12, KOS16]).

Beaver triples over R are exactly bilinear correlations with $G_x = G_y = G_z = R$ and bilinear map M multiplication over the ring, thus generating a triple from HSS can be achieved with costs as described above. Let $n = \lceil \log |R| \rceil$, and consider for instance $L = E = n/8$ in the parameters above. Then the HSS approach requires $3712(5n/4 + 160)$ bits of communication.

For $n \geq 128$, required computation is less than $(9n/4)^2$ terminal RMS multiplications, each with failure $(n/8)/(9n/4)^2 = 2/(81n)$. In this regime, the RMS multiplications are dominated by conversions. Estimating a baseline of 5×10^9 conversion steps per second (see Section 7), 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),[2] to generate an m-bit Beaver triple $\sim m^3/2^{26.5}$ seconds; for example, for 128-bit inputs (i.e., $n = 2^7$) this is roughly 135kB communication and 22ms computation time.

Consider the following alternative Beaver triple approaches.

- *Paillier based.* Beaver triples can be generated using an encryption scheme that supports homomorphic addition and multiplication by scalars, such as the Paillier cryptosystem.[3] 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 some applications), as it yields a qualitatively different protocol structure. In this approach, the parties must exchange information, perform a heavy "public key" computation (homomorphic evaluation), 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 pseudorandomness 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 [NP06,

[2] 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 here shares are over $\mathbb{Z}_q$.

[3] 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 $ax + b$.

IPS09, ADI+17, GNN17]. 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 [Gil99, KOS16]. 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 $n \geq 4096 = 2^{12}$ this is greater communication than our approach (and we expect this crossover to drop substantially with future optimizations); note in our current implementation (on a single core of a standard laptop), a 2^{12}-bit Beaver triple correlation can be generated via HSS in ~ 12.1 minutes.
We remark that the crossover point is lower when instantiating the HSS using ElGamal over elliptic-curve groups. As discussed in Section 7.3, 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 $1504n$ bits of communication (in the place of $3712n$), yielding a crossover of $n = 672 \approx 2^{9.4}$. The current implementation of HSS over elliptic curves would run notably longer at this size (~ 4.5 hours), 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 [Gil99,KOS16]. 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 $x^T A y$ 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 $r^x_i r^y_j \in \{0,1\}$), then for each desired $A = (a_{ij})$ the parties can take the appropriate linear combination of their $r^x_i r^y_j$ 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 [KK13], this will require more than $100n^2$ bits of communication, indicating that an HSS-based approach wins in communication already for $n \geq 84$. The computation required for a 84-bit Beaver triple correlation can be generated via HSS in ~ 6.3ms.

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 [IKM$^+$13, DNNR16]. Executing this technique on small chosen sub-computations within a larger secure computation can provide helpful speedups [DNNR16].

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 [IKOS08, App13, AL16], 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 4.5), 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 [BGI17], 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 [BGI17] to $O(\sqrt{m/n})$ using our new techniques. This application being more theoretical and involving non-trivial additional tools, we devote a full section to its study.

6 Cryptographic Capsules

In this section, we expand on the concept of cryptographic capsules, outlined in Section 5. A cryptographic capsule is a method to encode a long stream of correlated pseudorandomness into a short string. More specifically, it allows two players to generate, as an interactive protocol with $O(n)$ communication (up to $\mathsf{poly}(\lambda)$ factors), a *capsule* of size n, from which the two players can later non-interactively extract $m \gg n$ bits of correlated randomness, that can later be used to speedup secure computation protocols. Due to the inverse-polynomial failure probability of group-based HSS, a sanitization phase is necessary to remove faulty outputs. For efficiency, we require this phase to be information-theoretic and with communication $O(m)$, with small constant factors.

Advantages over Standard Preprocessing. Correlated randomness is typically generated as part of preprocessing protocols for multiparty computation; special forms of correlated randomness allow for information-theoretic secure computation of arbitrary circuits, with optimal communication up to small constant factors. Standard preprocessing protocols usually involve a larger communication, typically $O(\lambda \cdot |C|)$ for a circuit C. We outline below some theoretical advantages of cryptographic capsules over standard preprocessing protocol (such as oblivious-transfer-based preprocessing protocols).

Low communication. Generating a capsule involves a sublinear communication $O(n) \ll |C|$. Therefore, generating a cryptocapsule can be done without investing a large amount of resources. In addition to being communication-efficient, which saves bandwidth, this suggests that cryptocapsule can be used to hide the communication fingerprint of standard preprocessing protocols: two parties can generate a cryptocapsule without revealing whether they will actually want to perform a joint computation. On the other hand, standard preprocessing protocols leak non-trivial informations to any external observer: it leaks the fact that the two parties are willing to execute a secure protocol, as well as an upper bound on the size of the computation they will perform.

Universality. The purpose of preprocessing is to speedup secure computation. However, the need to execute a secure computation protocol with someone is typically something that might arise without anticipation. In a classical business context, parties willing to execute a secure computation protocol might not have planned it long before the actual computation should take place. With standard preprocessing, this leaves them with two alternatives:

- executing the protocol directly when the need for it arise, loosing the advantage of using preprocessed material to speed it up, or
- performing preprocessing protocols in advance with all parties that could someday want to execute a secure computation with them, which requires to invest a large computational effort to generate material that might never be actually used.

Cryptographic capsules offer a better alternative to the above. Generating a capsule involves only a short communication, way smaller than the amount of correlated randomness that can be extracted from it, and very little computation. This suggests to use cryptocapsules as a form of "business cards" that parties who might later become interested in performing a computation would exchange: two parties can jointly generate a cryptocapsule without actually knowing what amount of preprocessed material they will need, what kind of correlated randomness will be appropriate, nor even whether they will eventually want to perform a secure computation protocol. They do not commit to doing a computation by generating a cryptocapsule, but this gives them a *universal* source of a variable amount of near-arbitrary types of correlated randomness, which they can extract once needed without any further interaction.

General Structure. Homomorphic secret sharing suggests a natural approach for building cryptographic capsule: the capsule is generated as a level 1 encoding of the seed of a pseudorandom generator, together with shares of the appropriate secret key. Extracting correlated randomness is done by locally evaluating a function on this encoding that stretches pseudorandom bits with a PRG and compute a simple function; the shares of the output that the players get are exactly the correlated randomness. Consider for example the following function: on input a short seed x, f outputs many triples $(a_i, b_i, a_i b_i)_i$, where the $(a_i, b_i)_i$ are pseudorandom bits stretched from x. Shares of those triples are called Beaver triples, and can be used to speedup a variety of secure computation protocols for boolean circuits.

By the structure of group-based HSS, one must choose an appropriate PRG from a low-complexity class: the resulting function f must be efficiently expressed as an RMS program. The inverse-polynomial failure probability implies that the parties also have to sanitize the output to remove faulty triples. Even though the parties are notified of failures in the computation, they cannot simply inform each other about these failures: as the amount of pseudorandom bits m stretched from the capsule of size $O(n)$ is very large ($m \gg n$), the number of failures can potentially be way larger than n, hence revealing these failures could leak the entire seed x, or even the ElGamal secret key of the HSS scheme. Therefore, sanitizing the output must be done carefully to manage the leakage while maintaining the security. Our constructions focus on this natural approach, and our goal will be to design efficient algorithms to manage the leakage of cryptocapsules that follow this design strategy.

6.1 Definition

In this section, we define cryptographic capsules and introduce the efficiency and security notions it must satisfy. In the following, we call a *correlation* an efficiently computable function that, on input a uniformly random string, returns two correlated strings which are independently random. For example, the function $Q : (x, y, x', y', z) \mapsto ((x, y, x'y' \oplus z), (x \oplus x', y \oplus y', z))$ returns independently random strings that form xor-shares of $(x', y', x'y')$, which corresponds to Beaver triples.

Definition 14 (Cryptographic Capsule for a Correlation Q). *A two-party cryptographic capsule* CC *for a correlation* Q *is a triple of polynomial-time algorithms* $(\mathsf{CC.Generate}, \mathsf{CC.Extract}, \mathsf{CC.Sanitize})$ *(which are implicitely assumed to take a security parameter* 1^λ *as input) such that*

- $\mathsf{Generate}(m)$: *on common input a bound* m*, as a randomized interactive protocol between parties* (P_1, P_2)*, outputs* c_1 *to* P_1 *and* c_2 *to* P_2*;*
- $\mathsf{Extract}(Q, m, c_i)$: *on input a correlation* Q*, a bound* m*, and a capsule* c_i*, outputs* r_1 *to* P_1 *and* r_2 *to* P_2*, with* $|r_1| = |r_2| = O(m)$*;*
- $\mathsf{Sanitize}(Q, m, r_1, r_2)$: *on input a correlation* Q*,* r_1 *from* P_1*, and* r_2 *from* P_2*, as a randomized interactive protocol between* (P_1, P_2)*, outputs* r'_1 *to* P_1 *and* r'_2 *to* P_2*, with* $|r_1| = |r_2| = m$*;*

which satisfies the security and efficiency requirements defined below.

We stress that $\mathsf{Generate}$ and $\mathsf{Sanitize}$ are interactive protocols that have a randomized outputs (alternatively, they can take as input random coins from the parties), and $\mathsf{Extract}$ is a local procedure. In the following, we denote by $(r'_1, r'_2) \xleftarrow{\$} \mathsf{CC}[Q, m]$ the randomized two-party protocol obtained by successively running $(c_1, c_2) \xleftarrow{\$} \mathsf{Generate}(m)$, $(r_i)_{i\leq 2} \leftarrow (\mathsf{Extract}(Q, m, c_i))_{i\leq 2}$, and $(r'_1, r'_2) \xleftarrow{\$} \mathsf{Sanitize}(Q, m, r_1, r_2)$, and outputting (r'_1, r'_2).

Definition 15 (Security of Cryptographic Capsules). *A two-party cryptographic capsule* CC *for a correlation* Q *with bound* m *is* secure *if the protocol* $\mathsf{CC}[Q, m]$ *securely realizes the ideal functionality* $\mathcal{F}_{\mathsf{corr}}$ *represented Figure 6.*

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

The functionality is parametrized by a correlation Q and a bound m. On input $(\mathsf{start}, \mathsf{sid})$ from both parties, where start is a token and sid is a session identifier, pick a uniformly random string x such that $|Q(x)| = 2m$, and compute $(y_1, y_2) \leftarrow Q(x)$. Return y_1 to P_1 and y_2 to P_2.

Fig. 6. Ideal functionality for the generation of m correlated random bits for a correlation function Q

The above definition can be defined with respect to passive adversaries, or active adversaries; however, we will focus on the case of passively secure cryptographic capsules in this work.

Definition 16 (Efficiency of Cryptographic Capsules). *A two-party cryptographic capsule* CC *for a correlation* Q *is* efficient *if the following conditions hold:*

1. *the protocol* $\mathsf{Generate}(m)$ *has communication complexity* $o(m)$*;*
2. *the protocol* $\mathsf{Sanitize}(Q, m, r_1, r_2)$ *is optimally efficient regarding both communication and computation, up to small constants (*i.e.*, it has communication and computation* $O(m)$ *with small constants).*

Regarding item 2, we will in fact construct cryptographic capsules that use no public-key operations whatseever in the $\mathsf{Sanitize}$ protocol. Below, we introduce several tools that will be used in our constructions.

6.2 Preliminaries on Local Pseudorandom Generators

Our first ingredient for cryptographic capsules are *local pseudorandom generators*, which will be used to homomorphically expand the capsule into a long stream of correlated bits with low-complexity computation.

Definition of Local PRGs. Pseudorandom generators (PRGs) allow to expand a short seed seed into a long bitstring r, so that the distribution obtained by applying a PRG on a random short seed is computationally indistinguishable from the uniform distribution. An important efficiency measure for pseudorandom generators is their parallelizability. Pseudorandom generators achieving the most extreme form of parallelizability are known as *local pseudorandom generators*.

Definition 17. *(Local Pseudorandom Generator) A local pseudorandom generator* $\mathsf{PRG} : \{0,1\}^n \mapsto \{0,1\}^m$ *with locality* ℓ *is a pseudorandom generator in which each output bit depends on at most* ℓ *input bits.*

It follows from the definition that local PRGs are highly parallelizable: all output bits can be computed in parallel from the input bits. This also puts local PRGs into the low complexity class NC_0 of constant-depth fan-in-2 boolean circuits, which implies in particular that each output bit of a local PRG can be computed by an RMS program of constant size.

Candidate Local PRGs. The study of local one-way functions (later generalized to local PRGs) was initiated by Goldreich in [Gol00]. The goal of Goldreich was to construct a one-way function with the simplest possible design, to facilitate cryptanalytic attempts and to allow for a better understanding of the source of hardness in cryptographic primitives. All known candidate local one-way functions and local pseudorandom generators follow the same simple template advocated by Goldreich in his paper, that we describe below. Let $\mathsf{PRG} : \{0,1\}^n \mapsto \{0,1\}^m$ be a pseudorandom generator with locality ℓ.

- Select m size-ℓ subsets $S_1, \ldots, S_m$ of $[n]$ (each subset S_i correspond to the indices of the input bits that will be used to compute the ith output bit).
- On input $x = x_1 \ldots x_n$, compute the ith output bit y_i as $P(x_{S_i})$, where $P : \{0,1\}^\ell \mapsto \{0,1\}$ is a fixed boolean predicate, and x_{S_i} denotes the ℓ bits of x whose indices are in S_i.

It is conjectured that if the bipartite hypergraph $G = (S_1, \ldots, S_m)$, with an hyperedge from each $i \in [m]$ to the corresponding subset S_i of $[n]$, is *highly expanding*, and if the predicate P is sufficiently *non-degenerate*, then the above template leads to a secure pseudorandom generator. Highly expanding means that every subset of hyperedges that is not too large is almost pair-wise disjoint; in practice, one usually picks the subsets S_i uniformly at random, which generates a good expander graph with reasonable probability (see e.g. [ADI+17]). The choice of an appropriate predicate P requires more care and has been the subject of a large body of research [MST03, AHI04, CEMT09, CEMT14, OW14, AL16]. Below, we outline two families of predicates which satisfy the properties that are currently conjectured to lead to secure local PRGs, and which can be efficiently evaluated with group-based HSS.

Example 1: The XOR-MAJ *Predicate.* The (α, β)-XOR-MAJ predicate is defined as

$$P : (x_1, \ldots, x_{\alpha+\beta}) \mapsto x_1 \oplus \ldots \oplus x_\alpha \oplus \mathsf{majority}(x_{\alpha+1}, \ldots, x_{\alpha+\beta})$$

Evaluating the majority of β inputs requires $(\beta+1)^2/4$ RMS multiplications, see Section 4.6. Evaluating a xor is free when the output will not be reused in further computation (the parties can perform local integer addition and a final modular reduction). If the output must be reused (hence the parties have to compute level 2 shares of the output), a xor requires one RMS multiplication and integer addition, using the formula $x \oplus y = x + y - 2xy$. Therefore, evaluating the (α, β)-XOR-MAJ predicate requires at most $\alpha + (\beta+1)^2/4 + 1$ RMS multiplications. This candidate predicate was considered in several works, and was advocated e.g. in [AL16] for its good properties.

Example 2: The XOR-Δ *Predicate.* The (α, β)-XOR-Δ predicate is defined as

$$P : (x_1, \ldots, x_{\alpha+\beta(\beta+1)/2}) \mapsto x_1 \oplus \ldots \oplus x_\alpha \oplus \Delta_\beta(x_{\alpha+1}, \ldots, x_{\alpha+\beta(\beta+1)/2})$$

Where Δ_β denotes the βth triangular function, which takes $\beta(\beta+1)/2$ inputs and returns

$$\Delta_\beta : (x_1, \ldots, x_{\beta(\beta+1)/2}) \mapsto x_1 \oplus x_2x_3 \oplus x_4x_5x_6 \oplus \ldots \oplus \prod_{i=\beta(\beta-1)/2}^{\beta(\beta+1)/2} x_i.$$

The (α, β)-XOR-Δ predicate can be evaluated with $\alpha + \beta(\beta+1)/2 + 1$ RMS multiplication; however, if the output needs not be reused, it requires only $(\beta-1)(\beta-2)/2$ non-terminal RMS multiplications. When the XOR-MAJ and the XOR-Δ predicates are instantiated with parameters that give them comparable resistance to known attacks, the XOR-MAJ predicate (which was considered more often in the literature) has a better locality, while the XOR-Δ predicate requires less RMS multiplications. Boolean functions based on triangular functions were recently studied as natural candidate for the design of symmetric primitives well suited for homomorphic evaluation, see [MJSC16].

6.3 Preliminaries on Pseudorandom Functions

Our second ingredient for cryptographic capsules are special types of pseudorandom functions (PRFs) known as *puncturable pseudorandom functions* and *constrained pseudorandom functions*; they will be used to mask faulty outputs (that might leak information) in a computationally efficient way.

Definitions. Let $\mathcal{K}$ denote the key space, $\mathcal{X}$ denote the domain, and $\mathcal{Y}$ denote the range. A pseudorandom function is an efficiently computable deterministic function $F : \mathcal{K} \times \mathcal{X} \mapsto \mathcal{Y}$, together with a key generation algorithm $F.\mathsf{KeyGen}$ which, on input 1^λ, outputs a random key from $\mathcal{K}$. The core property of PRFs is that, on a random choice of key K, no probabilistic polynomial-time adversary should be able to distinguish $F(K, \cdot)$ from a truly random function, when given black-box access to it. Puncturable PRFs (pPRFs) have the additional property that some keys can be generated *punctured* at some point, so that they allow to evaluate the PRF at all points except the punctured point.

Definition 18 (Puncturable Pseudorandom Function). *A puncturable pseudorandom function is a pseudorandom function F with an additional punctured key space $\mathcal{K}_p$ and three probabilistic polynomial-time algorithms $(F.\mathsf{KeyGen}, F.\mathsf{Puncture}, F.\mathsf{Eval})$ such that*

- *$F.\mathsf{KeyGen}(1^\lambda)$ outputs a random key $K \in \mathcal{K}$,*
- *$F.\mathsf{Puncture}(K, x)$, on input $K \in \mathcal{K}$, $x \in \mathcal{X}$, outputs a punctured key $K\{x\} \in \mathcal{K}_p$,*
- *$F.\mathsf{Eval}(K\{x\}, x')$, on input a key $K\{x\}$ punctured at a point x, and a point x', outputs $F(K, x')$ if $x' \neq x$, and $\perp$ otherwise,*

such that no probabilistic polynomial-time adversary wins the experiment Exp-s-pPRF *represented Figure 7 with non-negligible advantage over the random guess.*

Definition 18 corresponds to a selective security notion for puncturable pseudorandom functions; adaptive security can also be considered, but will not be required in our work. Note also that we only considered puncturing at a single point in the above definition; one can easily extend it to define t-puncturable pseudorandom functions, in which a key can be punctured at up to t points. In this work, will we use a natural construction of a t-pPRF from t instances of a 1-pPRF. Puncturable pseudorandom functions are a special type of constrained pseudorandom functions, that we define below, for the class of point functions.

Experiment Exp-s-pPRF

Setup Phase. The adversary $\mathcal{A}$ sends $x^* \in \mathcal{X}$ to the challenger. When it receives x^*, the challenger picks $K \xleftarrow{\$} F.\mathsf{KeyGen}(1^\lambda)$ and a random bit $b \xleftarrow{\$} \{0,1\}$.

Challenge Phase. The challenger sends $K\{x^*\} \leftarrow F.\mathsf{Puncture}(K, x^*)$ to $\mathcal{A}$. If $b = 0$, the challenger additionaly sends $F(K, x^*)$ to $\mathcal{A}$; otherwise, if $b = 1$, the challenger picks a random $y^* \xleftarrow{\$} \mathcal{Y}$ and sends it to $\mathcal{A}$.

Experiment Exp-s-cPRF

Setup Phase. The adversary $\mathcal{A}$ sends $x^* \in \mathcal{X}$ to the challenger. When it receives x^*, the challenger picks $K \xleftarrow{\$} F.\mathsf{KeyGen}(1^\lambda)$ and a random bit $b \xleftarrow{\$} \{0,1\}$.

Query Phase. Each time $\mathcal{A}$ queries a circuit $C \in \mathcal{C}$ such that $C(x^*) = 0$, the challenger sends $K\{C\} \leftarrow F.\mathsf{Constrain}(K, C)$ to $\mathcal{A}$.

Challenge Phase. If $b = 0$, the challenger sends $F(K, x^*)$ to $\mathcal{A}$; otherwise, if $b = 1$, the challenger picks a random $y^* \xleftarrow{\$} \mathcal{Y}$ and sends it to $\mathcal{A}$.

Fig. 7. Selective security game for puncturable pseudorandom functions and constrained pseudorandom functions. At the end of each experiment, $\mathcal{A}$ sends a guess b' and wins if $b' = b$.

Definition 19 (Constrained Pseudorandom Function). *A constrained pseudorandom function for a class of circuits $\mathcal{C}$ a pseudorandom function F with an additional punctured key space $\mathcal{K}_\mathcal{C}$ and three probabilistic polynomial-time algorithms $(F.\mathsf{KeyGen}, F.\mathsf{Constrain}, F.\mathsf{Eval})$ such that*

- *$F.\mathsf{KeyGen}(1^\lambda)$ outputs a random key $K \in \mathcal{K}$,*
- *$F.\mathsf{Puncture}(K, C)$, on input $K \in \mathcal{K}$, $C \in \mathcal{C}$, outputs a constrained key $K\{C\} \in \mathcal{K}_\mathcal{C}$,*
- *$F.\mathsf{Eval}(K\{C\}, x)$, on input a key $K\{C\}$ constrained at a circuit C, and a point x, outputs $F(K, x)$ if $C(x) = 1$, and $\perp$ otherwise,*

such that no probabilistic polynomial-time adversary wins the experiment Exp-s-cPRF *represented Figure 7 with non-negligible advantage over the random guess.*

In this work, we will consider constrained PRFs for the class of prefix functions, *i.e.*, the functions C_z such that $C_z(x)$ returns 1 if z is a prefix of x, and 0 otherwise. For simplicity, we will denote $K\{z\}$ a constrained key for the prefix function C_z.

Construction of pPRFs and cPRFs. We recall below the GGM construction [GGM86] of a pseudorandom function $F : \{0,1\}^k \times \{0,1\}^\ell \mapsto \{0,1\}^\ell$ from any length-doubling pseudorandom generator $G : \{0,1\}^k \mapsto \{0,1\}^{2k}$. As observed in [BGI14,BW13,KPTZ13], this construction also leads a puncturable PRF, and a constrained PRF for prefix functions. On input a key K and a point $x = x_1 x_2 \dots x_\ell$, set $K^{(0)} \leftarrow K$ and perform the following iterative evaluation procedure: for $i = 1$ to ℓ, compute $(K_0^{(i)}, K_1^{(i)}) \leftarrow G(K^{(i-1)})$, and set $K^{(i)} \leftarrow K_{x_i}^{(i)}$. Output $K^{(\ell)}$.

- $F.\mathsf{KeyGen}(1^\lambda)$: output a random seed for G.
- $F.\mathsf{Constrain}(K, z)$: on input a key $K \in \{0,1\}^k$ and a prefix $z \in \{0,1\}^{\ell'}$ (with $\ell' \leq \ell$), apply the above procedure for ℓ' steps and return $K\{z\} = (K_{1-z_1}^{(1)}, \dots, K_{1-z_{\ell'}}^{(\ell')})$.
- $F.\mathsf{Eval}(K\{z\}, x)$, on input a constrained key $K\{z\}$ with $z \in \{0,1\}^{\ell'}$ and a point $x \in \{0,1\}^\ell$, if z is a prefix for x, output $\perp$. Otherwise, parse $K\{z\}$ as $(K_{1-z_1}^{(1)}, \dots, K_{1-z_{\ell'}}^{(\ell')})$ and start the iterative evaluation procedure from the first $K_{1-z_i}^{(i)}$ such that $x_i = 1 - z_i$.

Note that the above construction gives a puncturable PRF by constraining at a prefix of length ℓ. To obtain a t-puncturable PRF, run t instances of the above puncturable PRF and set the output of the PRF to be the bitwise xor of the output of each instance. With this construction, the length of a key punctured at t points is $tk\ell$.

6.4 First Construction

While our work is the first to consider cryptographic capsules, the work of [BGI17] develops tools that provide a natural (although already non-trivial) construction of cryptocapsules. In this section, we describe this natural candidate, which we use later as a baseline to compare our improved construction to.

Naive Construction. As a starting point, let us consider the following simple construction. As previously suggested, the $\mathsf{Generate}$ algorithm is constructed as a two-party generation of a level 1 share of a random seed x of size n (this involves $O(n)$ communication, up to $\mathsf{poly}(\lambda)$ factors). Let s be the length of the secret key of the HSS scheme. Given a bound $m \gg n$ on the number of correlated coins to generate, the $\mathsf{Extract}$ algorithm locally evaluates a function that stretches x into $O(m)$ pseudorandom bits and evaluates a simple correlation function on these random bits, with failure probability p per output bit. We assume that the function used to compute each output bit can be described by a constant-size circuit.

We then let the parties execute a sanitization phase, in which they securely remove the faulty correlated coins. To sanitize the coins, one of the parties simply notifies his opponent of faulty outputs, which are dropped. On average, this leaks $O(pm)$ bits to this party. To make this leakage harmless, the parties set p to $O(1/m)$ (this costs ignores the factor coming from the size of the ElGamal secret key, counting it leads to $p = O(1/sm)$): this ensures that with overwhelming probability, the amount of leakage will be at most logarithmic in n (as well as in the length of the secret key). Assuming

some entropic variant of the security assumption for the ElGamal ciphertexts, and using leakage-resilient local PRGs, one can simply instantiate this construction with primitives (encryption and PRG) which ensure that this amount of leakage can be tolerated. In spite of its simplicity, this solution has an obvious issue: computing each output involves a computation proportional to $O(s/p) = O(s^2 m)$. Therefore, to compute *each output bit*, each party must perform a computation proportional to the *total number of output bits* that he wants to generate. This makes this solution highly inefficient as soon as the parties wish to generate a large amount of correlated randomness. Below, we outline an improved strategy that stems from the work of [BGI17].

Punctured Oblivious Transfer. A k-out-of-m oblivious transfer (OT) protocol involves a sender, with a database $D = (d_1, \ldots, d_m)$, and a receiver holding a subset $S \subset [m]$ of size $|S| = k$. The receiver should learn all entries $(d_i)_{i \in S}$, without learning the entries indexed by $[m] \setminus S$, while the sender should not learn which entries the receiver got. Standard k-out-of-m OT protocols involve $O(\lambda \cdot (m + k))$ bits of communication. In [BGI17], the authors observed that when k is very close to m ($m \gg m - k$), this primitive can be implemented more efficiently, using only $m + o(m)$ bits of communication. We outline the construction below; it relies on a general two-party computation protocol (modeled as an oracle Π) and a k-puncturable PRF F with domain $[m]$.

1. The parties invoke Π on a randomized functionality that, on input S from the receiver, outputs a random PRF key K to the sender, and the key $K\{[m] \setminus S\}$ punctured at all points in $[m] \setminus S$ to the receiver.
2. For $i = 1$ to m, the sender computes and sends $d'_i \leftarrow d_i \oplus F(K, i)$.
3. The receiver outputs $(i, d'_i \oplus F.\mathsf{Eval}(K\{S\}, i))$ for all $i \in [m] \setminus S$.

Plugging in Yao's protocol for Π, and the GGM construction for F, this leads to a protocol with $m + (m - k) \cdot \log m \cdot \mathsf{poly}(\lambda)$ bits of communication. Setting $(m - k)$ to be sufficiently small leads to $(m - k) \cdot \log m \cdot \mathsf{poly}(\lambda) = o(m)$, hence the result.

Cryptocapsule from Punctured OT. Punctured OTs offer a way to sanitize the output while hiding faulty outputs, limiting the leakage by letting a party reconstruct non-faulty outputs while preventing himself from seeing faulty outputs. To ensure correct reconstruction of the output, we rely on an efficient error-correcting code with the following properties:

Lemma 20 (Lemma 5.2 from [BGI17]). *There is a randomized linear encoding function $E_r : \{0,1\}^m \mapsto \{0,1\}^{m+m/\lambda}$ that can correct a $1/\lambda^2$ rate of random erasures, with all but $m \cdot \mathsf{negl}(\lambda)$ probability.*

Using the above lemma, we proceed with the description of a cryptographic capsule from punctured OT. Let m be the bound. Let $\mathsf{PRG} : \{0,1\}^n \mapsto \{0,1\}^{m+m/\lambda}$ be a local pseudorandom generator, with seed length $n \ll m$. Let Q be the target correlation.

– $\mathsf{Generate}(m)$. As an interactive protocol, the two parties generate level 1 shares of a uniformly random seed $\rho || \rho_1$ of length $2n$, where ρ_1 is known to P_1 and ρ is unknown to both parties. Denote c_i the encoding share of party P_i.

- $\mathsf{Extract}(Q, m, c_i)$. Party P_i locally evaluates, on his share c_i of the cryptographic capsule, an RMS program for the function f that stretches x and x_1 into long pseudorandom strings z, z_1 using PRG, evaluates the correlation Q on (z, z_1), and encodes it with a randomized linear encoding function E_r, using a failure parameter $p = O(n/m)$ (this ignores $\mathsf{poly}(\lambda)$ factors). It obtains as output a string r_i, which contains on average $n' = O(n)$ faulty outputs.
- $\mathsf{Sanitize}(Q, m, r_1, r_2)$. The parties execute a punctured oblivious transfer protocol to let P_2 recover all outputs of P_1 excepts the ones for which P_2 raised a flag, using $m + o(m)$ bits of communication (as the number of faulty outputs is $n' \gg m$). P_2 reconstruct all outputs, correcting the n' erasures with the decoding procedure of E_r, and get a sanitized output r'_2. P_1 sets $r'_1 \leftarrow \mathsf{PRG}(\rho_1)$.

Remark 21 ($\mathsf{poly}(\lambda)$ *Factor).* To guarantee a total number $n' = O(n)$ of faulty outputs, the parties must use $1/p = O(c \cdot s \cdot m/n)$, where c denotes the size of the RMS program computing each output bit, and s denotes the length of the ElGamal secret key. As PRG, Q, and E_r are computed by constant-size boolean circuits, c is constant, hence one can use $p = O(n/sm)$. Note that the erasure correcting code requires $p < 1/\lambda^2$, which holds as soon as $sm > \lambda^2 n$.

Remark 22 (Preprocessing the Sanitizing Phase). In the above protocol, the Eval protocol is not information-theoretically secure, as it involves a punctured oblivious transfer; hence, it requires public-key operations. To remove all cryptographic operations from this protocol, the parties can preprocess the step 1 of the punctured OT protocol, by letting the receiver input a uniformly random subset S of the appropriate size $O(n)$. This preprocessing adds $O(n \log m \cdot \mathsf{poly}(\lambda)) = \tilde{O}(n \cdot \mathsf{poly}(\lambda))$ bits to the communication complexity of the $\mathsf{Generate}$ protocol, preserving its sublinearity. In the sanitization phase, the receiver can first send a permutation σ of $[m]$ that maps all flagged outputs to S, and the sender applies σ to his output before executing the step 2. As S is random, this leaks nothing about failures. However, this increases the communication of the sanitization phase (in the Eval protocol) to $m \log m$, contradicting the efficiency requirement of cryptographic capsules.

The increased communication of Eval can be avoided as follows. The parties partition $[m]$ into $m/t \log m$ blocks (for some constant t), indexed by $[m/t \log m]$, and execute the first step of a k-out-of-$m/t \log m$ punctured OT, where the receiver's input is a uniformly random subset of $[m/t \log m]$. In the extraction phase, the parties set $p = O(n/(sm \log m)) = \tilde{O}(n/sm)$, which ensures that at most $n' = O(n)$ *blocks* contain a faulty output. In the sanitization phase, the receiver first sends a permutation σ of the set $[m/t \log m]$ of indices of the blocks, which maps each block containing a flagged output to S. This permutation can be described with m/t bits.

Asymptotic Complexity. Computing each output bit in the extraction phase requires $O(c \cdot s \cdot 1/p) = O(s/p)$ conversion steps, which dominate the total cost. Using $p = O(n/sm)$ leads to a cost of $O(s^2 m/n)$, improving over the naive approach by a factor of n. As the punctured OT protocol involves $m + n \log m \cdot \mathsf{poly}(\lambda)$ bits of communication, the sanitization phase has the required efficiency as soon as $m/n \gg \log m \cdot \mathsf{poly}(\lambda)$, where the $\mathsf{poly}(\lambda)$ factor depends on the two-party computation protocol used to implement step 1 of the punctured OT. With the above preprocessing for the sanitization phase, the total number of bits exchanged in the $\mathsf{Sanitize}$ protocol is $m + m/t$, for some arbitrary constant t.

6.5 Second Construction

As the main issue of cryptographic capsules is the need to deal with the leakage efficiently, it is natural to examine to what extend leakage-absorbing pads, that we introduced in Section 4.5, could be used for this application. The ideal situation one could hope for would be to use leakage-absorbing pads to reduce the amount of leakage by a quadratic factor: $O(mp)$ faulty outputs would correspond to only $O(mp^2)$ bits of leakage, which leads to quadratic savings in efficiency. However, a naive use of leakage-absorbing pads fails to result in such significant savings. The reason for that is that the pads are *consumed* by masked evaluation algorithm, hence to compute m outputs with these algorithms, one would have to use $O(m)$ leakage-absorbing pads, jointly generated ahead of time by the parties. Including these pads in the cryptographic capsule would increase its size to $O(m)$, hence the capsule would become at least as long as the entire output that must be extracted from it.

Alternatively, the parties can jointly generate $O(n)$ leakage-absorbing pads as part of the cryptocapsule generation procedure, and use each pad in the computation of $O(m/n)$ output bits with the masked evaluation algorithms $\mathsf{MaskedEval}$ of Section 4.5. A quick analysis shows that this method leads to an overall complexity of $O(s^2 m/n)$ per output bit, comparable to the above method with punctured OT. We now describe an improved strategy that combine leakage-absorbing pads with a generalization of punctured OT in a non-trivial way. This strategy allows to obtain optimal efficiency improvements from leakage-absorbing pads, reducing the cost by a quadratic factor. We introduce below the core ingredients of our improved strategy.

Bootstrapping Leakage-Absorbing Pads. The first idea is to use a kind of bootstrapping approach to generate a large number of (pseudorandom) leakage-absorbing pads from an initial pool of $O(n)$ pads. Specifically, recall that a leakage absorbing pad is essentially a level 2 share of a random bit. The cryptocapsule will contain an encoding of a random seed and some number $N = O(n)$ pads. Each pad will be used in $\mathsf{MaskedEval}$ to homomorphically stretch the seed into a slightly larger number of pseudorandom bits, say $2N$; by the structure of the group-based HSS scheme, the output of this evaluation is shared between the parties in the form of level 2 shares, hence these outputs can be directly seen as leakage-absorbing pads. The initial pool of N pads is then dropped, and replaced by the $2N$ new pseudorandom leakage pads. The process is repeated, generating $4N$ pseudorandom pads from the $2N$ previous pseudorandom pads, and so on, until the pool contains m pseudorandom pads. Eventually, the m target output bits are computed with $\mathsf{MaskedEval}$, where each output bit is computed with a different pseudorandom pad from the pool. The computation of the pool of pseudorandom pads follows a tree structure, that is represented Figure 8.

Directly applying the above bootstrapping strategy would not result in any saving by itself. The reason is that a random pad does reduce leakage, but if a failure occur in the computation of a pseudorandom pad, then it does not offer any security guarantee anymore: *any* output computed from *any* pad contained in the subtree starting from this pad could be compromised. Our second ingredient is therefore an improved puncturing strategy that allows to puncture compromised pads directly at the tree level, so that puncturing a node of a tree also removes all nodes of the subtree starting from this node.

Prefix-Punctured Oblivious Transfer. Prefix-punctured oblivious transfers generalize the construction of punctured oblivious transfer given in [BGI17]. In a k-out-of-m

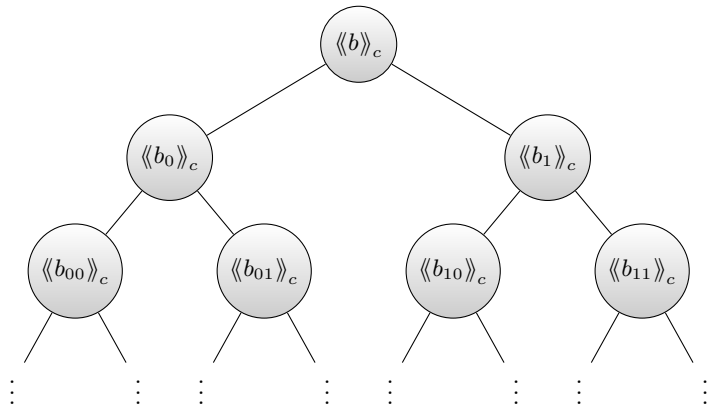

Fig. 8. Creation of the pool of pseudorandom pads. Each pad from the initial size-N pool is used in masked evaluations to stretch two new pads from the encoded seed, and this process is recursively applied. Each pad of the initial pool eventually generates m/N pseudorandom pads, which are the leaves of a tree of depth $\log(m/N)$.

prefix-punctured OT, the sender holds a database $D = (d_1, \dots d_m)$. We identify the set $[m]$ to $\{0,1\}^{\log m}$. The input of the receiver is a size-k set S of *prefixes*, each prefix being of length bounded by $\log m$ (the prefixes need not have all the same length). The receiver should learn all entries $d_i \in D$ with $i = i_1 i_2 \dots i_{\log m}$ for which there exists ℓ such that $i_1 i_2 \dots i_\ell \in S$ (*i.e.*, S contains a prefix of i). It is immediate to obtain a prefix-punctured oblivious transfer protocol, by taking the BGI construction of punctured OT and replacing the puncturable PRF by a constrained PRF for the class of prefix functions. As the GGM construction of puncturable PRF is already a prefix-constrained PRF, the instantiation suggested in [BGI17] and recalled in Section 6.4 can be directly used as a prefix-punctured OT.

Putting Pieces Together. We now combine the bootstrapping approach on leakage-absorbing pads with prefix-punctured oblivious transfer. As in the first construction, we let m be the bound, Q be a correlation, and $\mathsf{PRG} : \{0,1\}^n \mapsto \{0,1\}^{m+m/\lambda}$ be a local pseudorandom generator, with seed length $n \ll m$.

- $\mathsf{Generate}(m)$. As an interactive protocol, the two parties generate level 1 shares of a uniformly random seed $\rho||\rho_1||\rho'$ of length $3n$, where ρ_1 is known to P_1 and (ρ, ρ') are unknown to both parties, and a number $N = O(n)$ of leakage-absorbing pads. Denote c_i the encoding share of party P_i.
- $\mathsf{Extract}(Q, m, c_i)$. Party P_i recursively uses each pad of the initial size-N pool of leakage-absorbing pads as input to $\mathsf{MaskedEval}$, to stretch two pseudorandom pads from the encoded seed ρ', and replaces the used pad by the two newly generated pads, until a total number of m pads have been generated that way. The m pseudorandom pads are ordered in N blocks of m/N pads, each block containing all pads computed from the same pad of the initial pool.
  Afterward, party P_i locally evaluates an RMS program for the function f that stretches ρ and ρ_1 into long pseudorandom strings z, z_1 using PRG, evaluates the correlation function Q on z, xor the result with z_1, and encodes it with a randomized linear encoding function E_r, using a failure parameter $p = O(\sqrt{n/m} \cdot 1/s)$. The computation is performed with $\mathsf{MaskedEval}$, using a different pseudorandom pad from the pool to compute each output bit. P_i obtains as output a string r_i.

– $\mathsf{Eval}(Q, m, r_1, r_2)$. The parties execute a prefix-punctured oblivious transfer protocol to let P_2 recover all outputs of P_1, at the exception of all outputs involved in a $\mathsf{MaskedEval}$ computation with a faulty leakage-absorbing pad, using the *puncturing strategy* described below. The purpose of the puncturing strategy will be to carefully puncture at a minimal number of points, to hide only faulty outputs that could create leakage. This uses $m + o(m)$ bits of communication. P_2 reconstruct all outputs, correcting the erasures with the decoding procedure of E_r, and get a sanitized output r'_2. P_1 sets $r'_1 \leftarrow \mathsf{PRG}(\rho_1)$.

Puncturing Strategy. Let p be the probability of an error occuring when computing a pseudorandom pad from the pad associated to its parent node. Let us call *black nodes* the nodes of the tree for which (at least) an error occured, and *white nodes* the remaining nodes. Suppose now that we encounter some black node, let us call it BN, for the first time when analysing the tree from the root to the leaves. As an error occured on BN, it will leak a values of the form $b \oplus X$, where X is some private value, and b is the pseudorandom pad associated to its parent (which is indeed pseudorandom as no error occured yet). We need to distinguish three cases:

– If the brother node of this black node is also black, then we have to puncture their parent node, as two leakages from those two nodes would cancel the pad.
– If at least one of the node in the subtree starting from BN is also black (or if two errors occured on BN), then we have to puncture BN, as the error in BN implies that the pseudorandom pads derived from BN are incorrect and cannot be assumed to mask the private inputs.
– If none of the above happens, we do not need to puncture.

Asymptotic Complexity. In each of the N trees (each tree having m/N leaves and being of depth $\log(m/N)$), there are 2^{i-1} pairs of nodes (with the same parent) at level i, and for each such pair we need to puncture on average $\log(m/N)2^{1-i}p^2$ values, hence the total number of points to puncture is $m/N \cdot \log(m/N) \cdot p^2$ on average. Puncturing a point at level i costs $O(\lambda i)$ bits (the λ factor comes from the size of a GGM PRF key), hence the total cost of puncturing the tree is on average:

$$\begin{aligned}\lambda \left(\sum_{i=1}^{\log(m/N)} i \right) \cdot m/N \cdot p^2 &= \lambda \frac{\log(m/N)(\log(m/N)-1)}{2} m/N \cdot p^2 \\ &= O(\lambda \log^2(m/N) m/N p^2) \\ &= \tilde{O}(\lambda m p^2/N) = \tilde{O}(\lambda m p^2/n) \text{ (as } N = O(n))\end{aligned}$$

Therefore, to hide all failures in the computation (except perhaps for a constant number of failures, which can be handled by assuming some leakage-resilience from the primitives), the parties must set p so that the total number of points to puncture $N \cdot \tilde{O}(mp^2/n) = \tilde{O}(mp^2)$ remains bounded by $O(n)$ (up to logarithmic and $\mathsf{poly}(\lambda)$ factors), which requires to set p to $\tilde{O}(\sqrt{n/m} \cdot 1/s)$. Therefore, the cost of computing each output bit is reduced to $\tilde{O}(s^2\sqrt{m/n})$, which improves quadratically over the previous method (up to logarithmic factors).

Remark 23 (Preprocessing the Sanitization Phase). As in the previous method, the sanitization phase can be preprocessed during the cryptocapsule generation phase by applying the prefix-punctured OT protocol on a random subset S of prefixes, without

significantly increasing the communication, and increasing the computational overhead by a $O(\log m) = \tilde{O}(1)$ factor.

7 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 [BGI17], 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 [BGI17], 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 [BGI17] 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 4.1, we improve on the approach of [BGI17] 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 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 6 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 [BGI17]. 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 7.2 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 [BGI17].

Three optimizations that were introduced in [BGI16a, BGI17] 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 op-

timization 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.

7.1 Analytic Expression

We describe the number of RMS multiplications per second as a function of the target error probability ε and the feasible trade-offs. Loosely speaking, if ε is very small then the number of conversion steps until finding a distinguished point is expected to be high and is the bottleneck of each RMS multiplication. Conversely, if ε is relatively high then the exponentiations are the performance bottleneck.

Let s be the length of the secret key; note that $s = s(\lambda, B)$ is a function of the security parameter and the base B. Typically, $s = \lceil 2\lambda / \log B \rceil$ gives λ bits of security; setting $\lambda = 80$ gives $s(B) = \lceil 160 / \log B \rceil$. The number of ElGamal ciphertexts involved in an RMS multiplication is $s + 1$. In a non-terminal RMS multiplication there is one pairing operation $\mathsf{PairConv}^*$ for each ciphertext, including one conversion and two exponentiations. We refer to a partial RMS multiplication in which only the conversion (or exponentiation) component is executed as a *conversion semi-RMS* (or an *exponentiation semi-RMS*).

Let N_ε be the number of steps per $\mathsf{Convert}^*$ operation required to get an average failure probability ε per RMS multiplication ($N_\varepsilon = 2^{d+1}$ for an appropriate pattern length parameter d). By Corollary 6, we can bound N_ε by solving

$$\varepsilon \leq \frac{1 + \frac{s \cdot (B-1)}{2}}{4N_\varepsilon} + \left(\frac{s+1}{N_\varepsilon}\right)^2 = \frac{1}{4N_\varepsilon} + \frac{20 \cdot \lceil \frac{B-1}{\log B} \rceil}{N_\varepsilon} + \left(\frac{\left\lceil \frac{160}{\log B} \right\rceil + 1}{N_\varepsilon}\right)^2.$$

Solving for N_ε, we get

$$N_\varepsilon \leq \frac{1}{8\varepsilon} \cdot \left(1 + \frac{s \cdot (B-1)}{2}\right) + \sqrt{\left(\frac{1 + \frac{s \cdot (B-1)}{2}}{8\varepsilon}\right)^2 + \frac{(s+1)^2}{\varepsilon}}.$$

For a small enough failure probability ε, a good approximation of the left side of the inequality is $(1 + s \cdot (B-1)/2)/4\varepsilon$, hence the pattern length d can be taken as

$$d = \left\lceil \log \left(\frac{1}{4\varepsilon} \cdot \left(1 + \frac{s \cdot (B-1)}{2}\right)\right)\right\rceil - 1.$$

An RMS multiplication involves $s + 1$ $\mathsf{Convert}^*$ operations. If μ denotes the number of conversion steps per second in the underlying architecture then the number of conversion semi-RMS multiplications per second is

$$\rho_c = \frac{\mu \cdot 4\varepsilon}{(1+s) \cdot (1 + s \cdot (B-1)/2)}.$$

In order to estimate the performance of an exponentiation semi-RMS we note that one of the exponentiations in $\mathsf{PairConv}^*$ raises an element to the power of an additive share of cy and the other raises an element to the power of additive share of y for a memory variable y. Each share of y is the number of conversion steps until reaching a distinguished point, which can be bound in practice by 2^{32}. Therefore, the length of the share of y is at most 32 bits and the length of the share of cy is at most $160 + 32 = 192$

bits. This implies that number of multiplications for one PairConv is at most $\lceil 224/R \rceil$. If ν is the number of modular multiplications per second in our architecture then the number of exponentiation semi-RMS multiplications per second is

$$\rho_e = \frac{\nu}{\lceil \frac{224}{R} \rceil \cdot (s+1)}.$$

For fixed values B and R the number of RMS multiplications per second is $\frac{\rho_c \rho_e}{\rho_c + \rho_e}$. The more the failure probability is reduced, the more the conversions become the bottleneck; the crossover failure probability $\bar{\varepsilon}$ for which $\rho_c = \rho_e$ is given by

$$\bar{\varepsilon} = \frac{\nu}{4\mu} \cdot \left(1 + \left\lceil \frac{80 \cdot (B-1)}{\log B} \right\rceil\right) \cdot \frac{1}{\lceil \frac{224}{R} \rceil}.$$

As the error grows and exponentiations become the bottleneck, the value of B should be increased to reduce the overall time. As ε tends to 0 the value of B should be set to $B = 4$ or $B = 5$ to minimize the running time of the conversions.

For secret key HSS, using the heuristic optimization of [BGI16a], the ciphertext size is $\lceil \log p \rceil \cdot \lceil \frac{160}{\log B} + 2 \rceil$ bits. For public key HSS, using the optimization of Section 4.4 under the ESDH assumption, the ciphertext size is $\lceil \log p \rceil \cdot \left(\lceil \frac{160}{\log B} + 1 \rceil + 2 \left\lceil \sqrt{\frac{160}{\log B}} \right\rceil \right)$ bits. 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.

7.2 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 [BGI17]. 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 [BGI17] 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 $0 \leq i < 2^{d/2}$.

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[4], 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.

7.3 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` [SEC10], 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 24. 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.

Figure 9 provides the experimental validation of the heuristic running time estimate of our optimized conversion procedure; the estimate is heuristic in that it assumes that the sequence stream (namely, the MSB sequence of the group elements

[4] `https://gmplib.org/`

$h, hg, hg^2, hg^3, \ldots$, for a random starting point h), behaves like a totally random bit sequence. The graph shows that the empirical measurements closely match the heuristic prediction: the average number of steps needed to find the pattern 10^d of length $d+1$ is $2^{d+1}-2$. A similar behavior can be observed for the standard deviation.

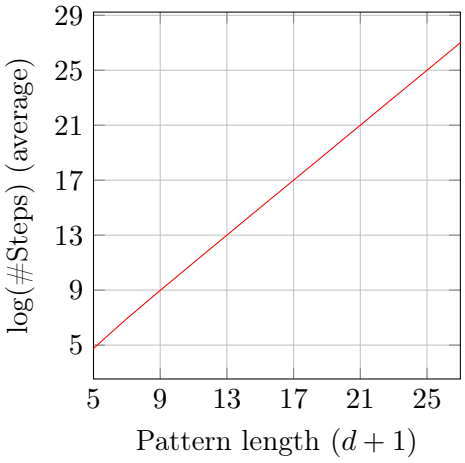

Fig. 9. Experimental validation of the random stream assumption. using the group $\mathbb{G} \subseteq \mathbb{Z}_p^*$ specified in Remark 24. The graph represents the average number of steps to find the pattern 10^d in the MSB sequence of group elements $h, hg, hg^2, hg^3, \ldots$, with a random starting point $h \in \mathbb{G}$.

We also compared our implementation using a partial match table with a naïvely optimized version of [BGI17] with extended window size, obtaining roughly a 50% factor of improvement for the `Intel Core i7` cpu.

We conclude with Figure 10, which 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.

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 24 for implementation details.

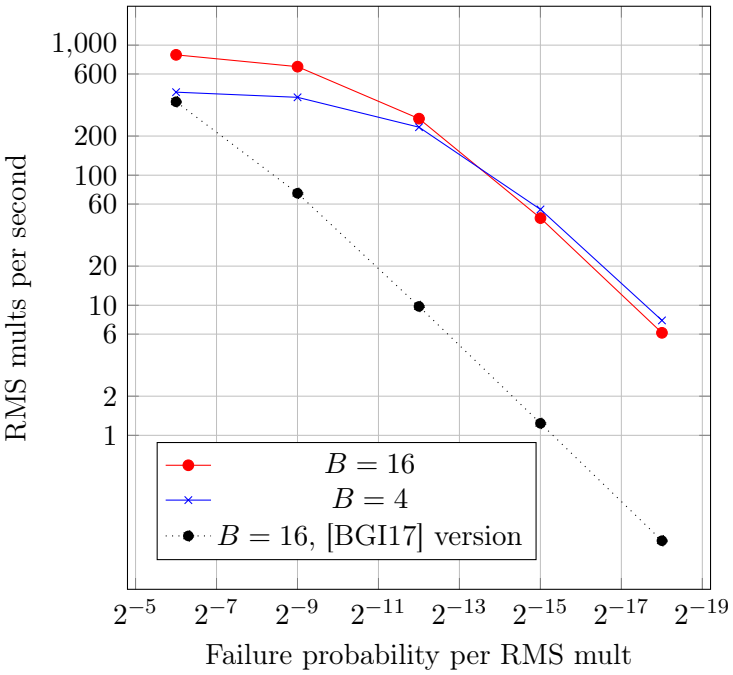

Fig. 10. Number of RMS multiplication per second given the failure probability per RMS multiplication with $R = 8$. The ciphertext size for $B = 4$ (resp., $B = 16$) is 18.8kB (resp., 10.6kB). See Remark 24 for implementation details. The [BGI17] version assumes 2×10^9 conversion steps per second.

Acknowledgements. We thank Josh Benaloh, Florian Bourse, Ilaria Chillotti, Henry Corrigan-Gibbs, Ranjit Kumaresan, Pierrick Meaux, and Victor Shoup for helpful discussions, comments, and pointers.

First, third, and fourth authors supported by ERC grant 742754 (project NTSC). First author additionally supported by ISF grant 1861/16, AFOSR Award FA9550-17-1-0069, and ERC grant 307952. Second author supported by ERC grant 339563 (project CryptoCloud). Third author supported by ISF grant 1638/15, a grant by the BGU Cyber Center, and by the European Union's Horizon 2020 ICT program (Mikelangelo project). Fourth author was supported by a DARPA/ARL SAFEWARE award, DARPA Brandeis program under Contract N66001-15-C-4065, NSF Frontier Award 1413955, NSF grants 1619348, 1228984, 1136174, and 1065276, NSF-BSF grant 2015782, ISF grant 1709/14, BSF grant 2012378, a Xerox Faculty Research Award, a Google Faculty Research Award, an equipment grant from Intel, and an Okawa Foundation Research Grant. This material is based upon work supported by the Defense Advanced Research Projects Agency through the ARL under Contract W911NF-15-C-0205. Fifth author supported by ERC grant 639554 (project aSCEND).

References

ADI+17. B. Applebaum, I. Damgård, Y. Ishai, M. Nielsen, and L. Zichron. Secure arithmetic computation with constant computational overhead. In *Crypto'17*, pages 223–254, 2017.

AHI04. M. Alekhnovich, E. A. Hirsch, and D. Itsykson. Exponential lower bounds for the running time of DPLL algorithms on satisfiable formulas. In *ICALP 2004*, *LNCS* 3142, pages 84–96. Springer, Heidelberg, July 2004.

AJLA+12. G. Asharov, A. Jain, A. López-Alt, E. Tromer, V. Vaikuntanathan, and D. Wichs. Multiparty computation with low communication, computation and interaction via threshold FHE. In *Eurocrypt'12*, pages 483–501, 2012.

AL16. B. Applebaum and S. Lovett. Algebraic attacks against random local functions and their countermeasures. In *STOC*, pages 1087–1100, 2016.

App13. B. Applebaum. Pseudorandom generators with long stretch and low locality from random local one-way functions. *SIAM J. Comput.*, 42(5):2008–2037, 2013.

BCG+17. E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, and M. Orrù. Homomorphic secret sharing: Optimizations and applications. In *CCS 2017*, pages 2105–2122, 2017.

BDOZ11. R. Bendlin, I. Damgård, C. Orlandi, and S. Zakarias. Semi-homomorphic encryption and multiparty computation. In *EUROCRYPT 2011*, *LNCS* 6632, pages 169–188. Springer, Heidelberg, May 2011.

BDPMW16. F. Bourse, R. Del Pino, M. Minelli, and H. Wee. FHE circuit privacy almost for free. In *Crypto'16*, pages 62–89, 2016.

Bea92. D. Beaver. Foundations of secure interactive computing. In *CRYPTO'91*, *LNCS* 576, pages 377–391. Springer, Heidelberg, August 1992.

Bea95. D. Beaver. Precomputing oblivious transfer. In *CRYPTO'95*, *LNCS* 963, pages 97–109. Springer, Heidelberg, August 1995.

Ben86. J. C. Benaloh. Secret sharing homomorphisms: Keeping shares of A secret sharing. In *CRYPTO*, pages 251–260, 1986.

BGI14. E. Boyle, S. Goldwasser, and I. Ivan. Functional signatures and pseudorandom functions. In *PKC 2014*, *LNCS* 8383, pages 501–519. Springer, Heidelberg, March 2014.

BGI15. E. Boyle, N. Gilboa, and Y. Ishai. Function secret sharing. In *EUROCRYPT*, pages 337–367, 2015.

BGI16a. E. Boyle, N. Gilboa, and Y. Ishai. Breaking the circuit size barrier for secure computation under DDH. In *CRYPTO*, pages 509–539, 2016. Full version: IACR Cryptology ePrint Archive 2016: 585 (2016).

BGI16b. E. Boyle, N. Gilboa, and Y. Ishai. Function secret sharing: Improvements and extensions. In *ACM CCS*, pages 1292–1303, 2016.

BGI17. E. Boyle, N. Gilboa, and Y. Ishai. Group-based secure computation: Optimizing rounds, communication, and computation. In *Eurocrypt'17*, pages 163–193, 2017.

BGI+18. E. Boyle, N. Gilboa, Y. Ishai, H. Lin, and S. Tessaro. Foundations of homomorphic secret sharing. In *ITCS 2018*, pages 21:1–21:21, 2018.

BGN05. D. Boneh, E.-J. Goh, and K. Nissim. Evaluating 2-DNF formulas on ciphertexts. In *TCC 2005*, *LNCS* 3378, pages 325–341. Springer, Heidelberg, February 2005.

BGW88. M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *STOC*, pages 1–10, 1988.

BL18. F. Benhamouda and H. Lin. k-round multiparty computation from k-round oblivious transfer via garbled interactive circuits. In *EUROCRYPT 2018, Part II*, pages 500–532, 2018.

BR13. Z. Brakerski and G. N. Rothblum. Obfuscating conjunctions. In *CRYPTO 2013, Part II*, *LNCS* 8043, pages 416–434. Springer, Heidelberg, August 2013.

BV14. Z. Brakerski and V. Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. *SIAM J. Comput.*, 43(2):831–871, 2014.

BW13. D. Boneh and B. Waters. Constrained pseudorandom functions and their applications. In *ASIACRYPT 2013, Part II*, *LNCS* 8270, pages 280–300. Springer, Heidelberg, December 2013.

Can97. R. Canetti. Towards realizing random oracles: Hash functions that hide all partial information. In *CRYPTO'97*, *LNCS* 1294, pages 455–469. Springer, Heidelberg, August 1997.

CBM15. H. Corrigan-Gibbs, D. Boneh, and D. Mazières. Riposte: An anonymous messaging system handling millions of users. In *2015 IEEE Symposium on Security and Privacy, SP*, pages 321–338, 2015.

CCD88. D. Chaum, C. Crépeau, and I. Damgård. Multiparty unconditionally secure protocols (extended abstract). In *STOC*, pages 11–19, 1988.

CEMT09. J. Cook, O. Etesami, R. Miller, and L. Trevisan. Goldreich's one-way function candidate and myopic backtracking algorithms. In *TCC 2009*, *LNCS* 5444, pages 521–538. Springer, Heidelberg, March 2009.

CEMT14. J. Cook, O. Etesami, R. Miller, and L. Trevisan. On the one-way function candidate proposed by goldreich. *ACM Transactions on Computation Theory (TOCT)*, 6(3):14, 2014.

CGGI16. I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *Asiacrypt'16*, pages 3–33, 2016.

CGKS95. B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *FOCS'95*, pages 41–50, 1995.

Cle91. R. Cleve. Towards optimal simulations of formulas by bounded-width programs. *Computational Complexity*, 1:91–105, 1991.

DHRW16. Y. Dodis, S. Halevi, R. D. Rothblum, and D. Wichs. Spooky encryption and its applications. In *CRYPTO*, pages 93–122, 2016.

DM15. L. Ducas and D. Micciancio. FHEW: bootstrapping homomorphic encryption in less than a second. In *EUROCRYPT*, pages 617–640, 2015.

DNNR16. I. Damgård, J. B. Nielsen, M. Nielsen, and S. Ranellucci. Gate-scrambling revisited - or: The TinyTable protocol for 2-party secure computation. Cryptology ePrint Archive, Report 2016/695, 2016. `http://eprint.iacr.org/2016/695`.

DPSZ12. I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO 2012*, *LNCS* 7417, pages 643–662. Springer, Heidelberg, August 2012.

Fre10. D. M. Freeman. Converting pairing-based cryptosystems from composite-order groups to prime-order groups. In *EUROCRYPT 2010*, *LNCS* 6110, pages 44–61. Springer, Heidelberg, May 2010.

Gen09. C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178, 2009.

GGHR14. S. Garg, C. Gentry, S. Halevi, and M. Raykova. Two-round secure MPC from indistinguishability obfuscation. In *TCC*, pages 74–94, 2014.

GGM86. O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *Journal of the ACM*, 33(4):792–807, October 1986.

GI14. N. Gilboa and Y. Ishai. Distributed point functions and their applications. In *EUROCRYPT 2014*, *LNCS* 8441, pages 640–658. Springer, Heidelberg, May 2014.

Gil99. N. Gilboa. Two party RSA key generation. In *CRYPTO'99*, *LNCS* 1666, pages 116–129. Springer, Heidelberg, August 1999.

GLS15. S. D. Gordon, F. Liu, and E. Shi. Constant-round MPC with fairness and guarantee of output delivery. In *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, pages 63–82, 2015.

GMW87. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *STOC*, pages 218–229, 1987.

GNN17. S. Ghosh, J. B. Nielsen, and T. Nilges. Maliciously secure oblivious linear function evaluation with constant overhead. *IACR Cryptology ePrint Archive*, page 409, 2017.

Gol00. O. Goldreich. Candidate one-way functions based on expander graphs. Cryptology ePrint Archive, Report 2000/063, 2000. `http://eprint.iacr.org/2000/063`.

GS17. S. Garg and A. Srinivasan. Garbled protocols and two-round MPC from bilinear maps. In *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017*, pages 588–599, 2017.

GS18. S. Garg and A. Srinivasan. Two-round multiparty secure computation from minimal assumptions. In *EUROCRYPT 2018 , Part II*, pages 468–499, 2018.

GSW13. C. Gentry, A. Sahai, and B. Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Crypto'13*, pages 75–92, 2013.

HS15. S. Halevi and V. Shoup. Bootstrapping for HElib. In *EUROCRYPT*, pages 641–670, 2015.

IKM+13. Y. Ishai, E. Kushilevitz, S. Meldgaard, C. Orlandi, and A. Paskin-Cherniavsky. On the power of correlated randomness in secure computation. In *TCC 2013*, *LNCS* 7785, pages 600–620. Springer, Heidelberg, March 2013.

IKOS08. Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. Cryptography with constant computational overhead. In *40th ACM STOC*, pages 433–442. ACM Press, May 2008.

IPS09. Y. Ishai, M. Prabhakaran, and A. Sahai. Secure arithmetic computation with no honest majority. In *TCC'09*, pages 294–314, 2009.

KK13. V. Kolesnikov and R. Kumaresan. Improved OT extension for transferring short secrets. In *CRYPTO 2013, Part II*, *LNCS* 8043, pages 54–70. Springer, Heidelberg, August 2013.

KO97. E. Kushilevitz and R. Ostrovsky. Replication is NOT needed: SINGLE database, computationally-private information retrieval. In *38th FOCS*, pages 364–373. IEEE Computer Society Press, October 1997.

KOS16. M. Keller, E. Orsini, and P. Scholl. MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 830–842, 2016.

KPTZ13. A. Kiayias, S. Papadopoulos, N. Triandopoulos, and T. Zacharias. Delegatable pseudorandom functions and applications. In *ACM CCS 13*, pages 669–684. ACM Press, November 2013.

Kur02. K. Kurosawa. Multi-recipient public-key encryption with shortened ciphertext. In *PKC 2002*, *LNCS* 2274, pages 48–63. Springer, Heidelberg, February 2002.

LTV12. A. López-Alt, E. Tromer, and V. Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *44th ACM STOC*, pages 1219–1234. ACM Press, May 2012.

MJSC16. P. Méaux, A. Journault, F.-X. Standaert, and C. Carlet. Towards stream ciphers for efficient FHE with low-noise ciphertexts. In *Eurocrypt'16*, pages 311–343, 2016.

MSS11. S. Myers, M. Sergi, and A. Shelat. Threshold fully homomorphic encryption and secure computation. IACR Cryptology ePrint Archive, 2011.

MST03. E. Mossel, A. Shpilka, and L. Trevisan. On e-biased generators in NC0. In *44th FOCS*, pages 136–145. IEEE Computer Society Press, October 2003.

MW16. P. Mukherjee and D. Wichs. Two round multiparty computation via multi-key FHE. In *Proc. EUROCRYPT 2016*, pages 735–763, 2016.

Nie73. P. T. Nielsen. On the expected duration of a search for a fixed pattern in random data (corresp.). *IEEE Trans. Information Theory*, 19(5):702–704, 1973.

NP06. M. Naor and B. Pinkas. Oblivious polynomial evaluation. *SIAM J. Comput.*, 35(5):1254–1281, 2006.

OS05. R. Ostrovsky and W. Skeith III. Private searching on streaming data. In *Proc. CRYPTO 2005*, pages 223–240, 2005.

OW14. R. ODonnell and D. Witmer. Goldreich's prg: evidence for near-optimal polynomial stretch. In *Computational Complexity (CCC), 2014 IEEE 29th Conference on*, pages 1–12. IEEE, 2014.

PS16. C. Peikert and S. Shiehian. Multi-key FHE from LWE, revisited. In *TCC'16*, pages 217–238, 2016.

RAD78. R. L. Rivest, L. Adleman, and M. L. Dertouzos. On data banks and privacy homomorphisms. In *Foundations of secure computation (Workshop, Georgia Inst. Tech., Atlanta, Ga., 1977)*, pages 169–179. Academic, New York, 1978.

SEC10. SECG. Sec 2: Recommended elliptic curve domain parameters, version 2. `http://www.secg.org`, 2010.

vDGHV10. M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. In *Proc. EUROCRYPT 2010*, pages 24–43, 2010.

Yao86. A. C.-C. Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, pages 162–167, 1986.

A Deferred Proofs

A.1 Proof Sketch of Lemma 12

The generic group model allows to provide heuristic security arguments by restricting the adversary to perform only basic group operations (multiplications and inversions) on a given group element, without being able to exploit any further information from the particular structure of the group. A security proof for an assumption in the generic group model shows that any attack on the assumption must use the group in a non-black-box way, suggesting that the assumption might be secure when isntantiated on an appropriate group. In this section, we sketch a proof that the entropic-span Diffie-Hellman assumption (Assumption 11 from Section 4) holds in the generic group model.

Generic Group Model. We implement the generic group model by choosing a random deterministic encoding $\sigma : \mathbb{G} \mapsto \{0,1\}^n$ for some large enough n. For two group elements (g_1, g_2), given their encodings $(\sigma(g_1), \sigma(g_2))$, the only information available is whether $g_1 = g_2$. Group operations are made available through two oracles, $\mathcal{O}_{\mathsf{mult}}$ (which, on input $\sigma(g_1), \sigma(g_2)$, returns $\sigma(g_1 g_2)$) and $\mathcal{O}_{\mathsf{inv}}$ (which, on input $\sigma(g)$, returns $\sigma(g^{-1})$). By choosing n to be sufficiently large so that a random element of $\{0,1\}^n$ is a valid encoding of a group element with negligible probability, we can assume without loss of generality that any adversary in the generic group model will only submit to the oracles encodings of elements that he has previously received (as input or as answers from the oracles).

ESDH Assumption. For a distribution X with finite support, we denote

$$H_\infty(X) = -\log\left(\max_x \left(\Pr[X = x]\right)\right)$$

the min-entropy of X. We recall the ESDH assumption below. Let $\bullet$ denote the inner product operation, and let $B \geq 2$ denote any basis. For any integer t and any vector $v \in Z_q^t$, we let X_v denote the distribution ensemble

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

The entropic span Diffie-Hellman assumption (ESDH) states that for any polynomials $t = t(\lambda), k = k(\lambda)$, and any $v_1, \cdots, v_k \in Z_q^t$ such that for any non-trivial linear combination v of $(v_1, \cdots, v_k)$, it holds that $H_\infty(X_v) \geq \omega(\log \lambda)$, the following distributions are indistinguishable:

$$D_0 = \{v_1, \cdots, v_k, g, g^{v_1 \bullet c}, \cdots, g^{v_k \bullet c} \mid c \leftarrow \{0, \cdots, B-1\}^t\}$$
$$D_1 = \{v_1, \cdots, v_k, g, g_1, \cdots, g_k \mid (g_1, \cdots, g_k) \xleftarrow{\$} \mathbb{G}^k\}$$

Security in the Generic Group Model. Let $\mathcal{A}$ be a polynomial-time adversary in the generic group model. Consider a simulator Sim which is given as input k vectors $(v_1, \cdots, v_k)$ of size t (satisfying the above requirements), playing the following game. Sim picks $k+1$ uniformly random strings $\sigma_0, \cdots, \sigma_k \xleftarrow{\$} \{0,1\}^n$ and sends $(v_1, \cdots, v_k, \sigma_0, \cdots, \sigma_k)$ to $\mathcal{A}$. Then, Sim simulates the oracle queries as follows:

- Sim maintains a list L of pairs (P, σ), where P is a multivariate polynomial over $\mathbb{F}_q[X_1, \cdots, X_k]$, and σ is an encoding. Sim initially sets

$$L \leftarrow ((\sigma_0, 1), (\sigma_1, X_1), \ldots, (\sigma_k, X_k))$$

- On input (σ, σ') from $\mathcal{A}$, Sim simulates $\mathcal{O}_{\mathsf{mult}}$ as follows: he retrieves (P, P') associated to (σ, σ') from L, and search L for an encoding σ'' associated to $P + P'$. If Sim finds such an encoding, he returns σ''; otherwise, Sim picks $\sigma'' \xleftarrow{\$} \{0,1\}^n$, returns σ'' and adds $(\sigma'', P + P')$ to L.
- On input σ from $\mathcal{A}$, Sim simulates $\mathcal{O}_{\mathsf{inv}}$ as follows: he retrieves P associated to σ from L, and search L for an encoding σ associated to $-P$. If Sim finds such an encoding, he returns σ'; otherwise, Sim picks $\sigma' \xleftarrow{\$} \{0,1\}^n$, returns σ' and adds $(\sigma', -P)$ to L.

After making Q queries to the oracle, $\mathcal{A}$ returns a guess bit b'. Then, Sim picks a uniformly random bit b. If $b = 0$, Sim picks $c \leftarrow \{0, \cdots, B-1\}^t$ and sets

$$(X_1, \cdots, X_k) = (v_1 \bullet c, \cdots, v_k \bullet c);$$

otherwise, Sim picks uniformly random $(x_1, \cdots, x_k)$ and sets

$$(X_1, \cdots, X_k) = (x_1, \cdots, x_k).$$

We now show that the simulation is correct with overwhelming probability in the case $b = 0$; it follows immediately that no information leaks about b from the game.

In the case $b = 0$, all the encodings $\mathcal{A}$ can get by querying the oracles are of the form $\sigma(\mu_0 + \sum_{i=1}^{k} \mu_i (v_i \bullet c))$, for some coefficients $\mu_0, \cdots, \mu_k$. Let us denote by $V = [v_1 | \ldots | v_k]$ the matrix of dimension $t \times k$ whose column-vectors are the various v_i-s. For any query with constant term μ_0 and coefficients column-vector $\mu \in \mathbb{Z}_q^k$, it is easy to see that the simulation fails only if a collision happen, *i.e.*, $\mathcal{A}$ obtains two encodings $\sigma(\mu_0 + cV\mu$ and $\sigma(\mu_0' + cV\mu')$ such that $\mu \neq \mu'$ and

$$\mu_0 + cV\mu = \mu_0' + cV\mu'.$$

The above equation reduces to $cV(\mu - \mu') = \mu_0' - \mu_0'$. The probability that two queries out of the Q queries of $\mathcal{A}$ satisfy this equality can be bounded by

$$\binom{Q}{2} \cdot 2^{-H_\infty(X_{\lambda,t,V(\mu-\mu')})}.$$

As $V(\mu - \mu')$ is a non-trivial vector (as $\mu \neq \mu'$) in the span of $(v_1, \cdots, v_k)$, by assumption it holds that $H_\infty(X_{V(\mu-\mu')}) = \omega(\log \lambda)$. As $\mathcal{A}$ runs in polynomial time, it makes a polynomial number of queries to the oracle, hence the probability that the simulation fails is bounded by

$$\binom{\mathsf{poly}(\lambda)}{2} \cdot 2^{-\omega(\log \lambda)} = \mathsf{negl}(\lambda)$$

which concludes the proof.

A.2 Expected Number of Steps to Find the Sequence 0^d

In this section we give a self-contained proof that the expected number of steps X for finding the first pattern 0^d in a random bit-sequence stream is $2^{d+1} - 2$.

- For $k = 1 \cdots d$, if the k'th bit is the first 1 encountered, then it will take an expected number X of additional steps to find the pattern from this point. In this case, which happens with probability $1/2^k$, k steps have been already performed.
- If the first d bits are all 0, which happens with probability $1/2^d$, then the pattern is found after d steps.

By linearity of expectation, we can therefore obtain X by solving

$$X = \frac{d}{2^d} + \sum_{k=1}^{d} \frac{X+k}{2^k}$$

from which we get $X = 2^{d+1} - 2$.

B Homomorphic Secret Sharing from LWE

As previously observed in [BGI16a, BGI17], an alternative approach to build a homomorphic secret sharing scheme is to rely on (threshold) fully-homomorphic encryption [MSS11, AJLA+12]. This gives rise to an HSS scheme for any circuit under the LWE assumption. In this section, we discuss the specificities of the LWE-based approach for building HSS schemes, and provide some comparison to the group-based approach studied in this paper.

The intuition behind the threshold FHE-based approach for HSS is the following: key generation and encryption for FHE-based HSS are just the standard key generation and encryption algorithms of the FHE scheme. Distributed evaluation is performed by letting the two computing parties homomorphically evaluate the desired program over the ciphertexts, then performing a local threshold decryption procedure. Using existing LWE-based FHE scheme, where ciphertexts follow a simple linear algebra structure, applying the standard decryption algorithm with shares of the secret key leads to noisy shares of the output. Then, the computing parties locally round their noisy share of the outputs, obtaining boolean shares of the result (with some probability that depends on the parameters of the scheme).

We provide below further details on two natural directions that can be envisioned to implement this approach, using fully homomorphic encryption with bootstrapping, or using leveled fully homomorphic encryption.

B.1 Fully Homomorphic Encryption Approach

To our knowledge, the fastest existing implementation of fully homomorphic encryption was described in [CGGI16]. It relies on a "bootstrapping at each gate" paradigm, and allows to perform a bootstrapping operation at the impressive speed of 20ms on a standard computer (the paper reports 52ms, but these number where improved in the implementation reported on the corresponding github project). The paper reports a ciphertext size of 8kB, comparable to what we get using group-based HSS, by relying on an external product procedure to considerably reduce the ciphertext size compared with previous constructions of FHE. However, the FHE-based approach also has several downsides, that we outline below.

Large size of the Keys. A well known issue with known fully homomorphic encryption schemes is the total size of the keys (bootstrapping key and key switching key), which exceeds 50MB in [CGGI16], making this approach considerably less communication-efficient than the group-based approach in most scenarios.

Distributed Key Generation. Our group-based HSS protocol enjoys a simple distributed key generation protocol. Distributed key generation for threshold FHE has been previously studied in the context of multikey FHE [LTV12,PS16]. However, existing methods rely on very expensive noise-flooding procedure, to compensate for the fact

that locally generated shares of the FHE secret key do not add up to a well-distributed secret key, and are mainly of theoretical interest. We point out, nonetheless, that recent works on circuit-private FHE [BDPMW16] seem to suggest that more efficient alternative methods could be envisioned, adding a small gaussian noise to the shares of the key.

Error Probability. The local decryption and rounding procedure incur an error probability proportional to the inverse of the modulus. In efficient FHE implementations such as [CGGI16], the modulus is typically a small number (e.g. 2048), which means that, as for group-based HSS, efficient FHE-based HSS have a non-negligible error probability (and similarly, those errors can leak information on the secret key). However, unlike in the group-based approach, those errors cannot be detected by the players, which makes it considerably harder to correct them; the techniques that we developed to deal with the leakage crucially rely on the fact that the parties get notified of risks of errors, which is not the case for the FHE-based approach, hence requiring expensive protocols for obliviously reconstructing the erroneous outputs. In addition, the error probability ε in group-based HSS can be decreased at a cost linear in $1/\varepsilon$. On the other hand, decreasing the error for the FHE based approach requires increasing the modulus size, which results in a very large overhead in the running time of the bootstrapping, and in both the size of the ciphertexts and of the keys.

B.2 Leveled Homomorphic Encryption Approach

Due to the previously mentioned issues, HSS based on fully homomorphic encryption are likely to provide poorer performances than the group-based approach in most scenarios. An alternative to the above approach is to rely on level homomorphic encryption, avoiding the need to use bootstrapping. However, this comes at the cost of larger ciphertexts, as the bootstrapping keys in the previous approach allow to implement the homomorphic operations as external products between LWE samples and GSW samples, where the ciphertexts are encoded with LWE samples while GSW samples (which are way larger) are confined to the bootstrapping key. GSW-based leveled homomorphic encryption enjoys faster homomorphic operations, but has larger ciphertexts. A common approach to mitigate this issue is to use leveled FHE in a batch setting, where each ciphertext contains a large number of slots, and computation is performed in a SIMD fashion. For most applications that we considered in this paper, the batch setting is not relevant, but it might be interesting in other scenarios, if threshold decryption and local rounding can be made compatible with the techniques used to encode several plaintexts in a single ciphertexts.