Group-Based Secure Computation: Optimizing Rounds, Communication, and Computation

Elette Boyle
IDC Herzliya
eboyle@alum.mit.edu

Niv Gilboa
Ben Gurion University
gilboan@bgu.ac.il

Yuval Ishai
Technion and UCLA
yuvali@cs.technion.ac.il

July 7, 2017

Abstract

A recent work of Boyle et al. (Crypto 2016) suggests that "group-based" cryptographic protocols, namely ones that only rely on a cryptographically hard (Abelian) group, can be surprisingly powerful. In particular, they present succinct two-party protocols for securely computing branching programs and NC^1 circuits under the DDH assumption, providing the first alternative to fully homomorphic encryption.

In this work we further explore the power of group-based secure computation protocols, improving both their asymptotic and concrete efficiency. We obtain the following results.

- **Black-box use of group.** We modify the succinct protocols of Boyle et al. so that they only make a black-box use of the underlying group, eliminating an expensive non-black-box setup phase.
- **Round complexity.** For any constant number of parties, we obtain 2-round MPC protocols based on a PKI setup under the DDH assumption. Prior to our work, such protocols were only known using fully homomorphic encryption or indistinguishability obfuscation.
- **Communication complexity.** Under DDH, we present a secure 2-party protocol for any NC^1 or log-space computation with n input bits and m output bits using $n+(1+o(1))m+\mathsf{poly}(\lambda)$ bits of communication, where λ is a security parameter. In particular, the protocol generates n instances of bit-oblivious-transfer using $(4+o(1))\cdot n$ bits of communication. This gives the first constant-rate OT protocol under DDH.
- **Computation complexity.** We present several techniques for improving the computational cost of the share conversion procedure of Boyle et al., improving the concrete efficiency of group-based protocols by several orders of magnitude.

Keywords: Secure computation, homomorphic secret sharing, share conversion, fully homomorphic encryption

1 Introduction

Gentry's 2009 breakthrough on fully homomorphic encryption (FHE) [RAD78, Gen09] changed the landscape of the theory of secure computation. FHE enables arbitrary computations on encrypted inputs, thereby providing a general-purpose tool for *succinct* secure computation protocols whose communication complexity is smaller than the circuit size of the function being computed. FHE-based protocols were also used to minimize the *round complexity* of secure multiparty computation [LTV12, AJLA$^+$12, MW16, DHRW16].[1]

On the downside, despite impressive recent progress [HS15, DM15, CGGI16], the concrete efficiency of current FHE implementations still leaves much to be desired. Moreover, the set of cryptographic assumptions on which FHE can be based is still quite narrow. These two limitations may in fact be related, in that attempts at efficient implementation are curbed by the limited variety of FHE candidates. Indeed, all such candidates rely on similar *lattice-related* algebraic structures and are subject to lattice reduction attacks that have a negative impact on concrete efficiency. In particular, no FHE construction is known under a discrete-log-type assumption or even in the generic group model. This should be contrasted with standard public-key encryption schemes and non-succinct secure computation protocols that can be easily (and unconditionally) realized in the generic group model.

A recent work of Boyle et al. [BGI16] introduced a new technique for succinct secure computation that can be based on any DDH-hard group. (For better concrete efficiency, it is useful to rely on stronger assumptions than DDH, such as the circular security of ElGamal encryption.) While the results obtained using this group-based approach are weaker than corresponding FHE-based results in several important aspects, they do give hope for better concrete efficiency in useful application scenarios. The present work is motivated in part by this hope.

More concretely, the approach of [BGI16] replaces the use of FHE by a 2-party *homomorphic secret sharing* (HSS) primitive, which turns out to be sufficient for the purpose of succinct secure two-party computation. An HSS scheme is a secret sharing scheme that supports homomorphic computations on the shares, such that the output of the computation is compactly shared between the parties. We in fact make the stronger requirement that the output be *additively* shared between the parties over a finite Abelian group. In particular, if the output is a single bit, each output share can be just a single bit. HSS can be viewed as a dual version of *function section sharing* [BGI15], where the roles of the function and the input are reversed, or a weaker version of *additive-spooky encryption* [DHRW16].

The main result of [BGI16] is a DDH-based HSS scheme for *branching programs*, which in particular captures logspace and NC^1 computations. We provide a high level overview of this HSS scheme in Section 2.6. The HSS scheme of [BGI16] can be used to obtain succinct secure two-party computation protocols for the same classes. One difficulty in applying this HSS scheme towards secure computation is that it has an inverse polynomial error probability, and moreover the event of an error is correlated with the secret input and with bits of the secret key. This difficulty was addressed in [BGI16] by combining error-correcting codes with general-purpose secure two-party computation protocols for recovering the correct output from the encoding with negligible error probability. This approach has a significant overhead in communication and computation, and requires additional rounds of interaction.

The source of the error in the HSS scheme from [BGI16] is a non-interactive *share conversion* procedure, which converts multiplicative shares into additive shares. To perform this conversion with an error probability bound of δ, the procedure requires $O((1/\delta) \cdot \log(1/\delta))$ (or *expected* $O(1/\delta)$) group multiplications.

[1]As in previous related works, our default notion of secure computation refers to security against *passive* (semi-honest) adversaries. In most cases, similar protocols with security against active (malicious) adversaries can be obtained under the same assumptions by using a suitable version of the GMW compiler [GMW87, NN01, HK07].

1.1 Our Contribution

In this work we further explore the power of group-based secure computation protocols, improving both their asymptotic and concrete efficiency. Following is a detailed overview of our results and the underlying techniques.

Black-box use of group. The group-based succinct protocols from [BGI16] use general-purpose secure computation to distribute the key generation of a "public-key" HSS scheme, namely one that allows joint computation on two or more shared inputs. This procedure leads to poor concrete efficiency, and makes a non-black-box use of the underlying cryptographic group. We present a generic approach for obtaining similar results while only making a black-box use of the underlying group. This approach relies on the plaintext- and key-homomorphism properties of ElGamal encryption (or its circular-secure variant [BHHO08]) and can be used for improving the concrete cost of group-based protocols.

Minimizing round complexity. For any constant number of parties, we obtain 2-round MPC protocols based on a Public Key Infrastructure (PKI) setup under the DDH assumption.[2] Prior to our work, such protocols were only known using different flavors of FHE [AJLA^{+}12, MW16, DHRW16] or indistinguishability obfuscation [GGHR14, DHRW16]. (Granted, the latter protocols can further support polynomial number of parties, and with milder setup requirements: PKI setup can be relaxed to a CRS setup by using *multi-key FHE,* which can be based on LWE [MW16, DHRW16], or even eliminated by relying on indistinguishability obfuscation [DHRW16].)

Our 2-round protocol is obtained in three steps. In the first step, we construct a 1-round (PKI-based) distributed HSS scheme, which can be used to jointly share inputs that originate from multiple clients. This can be used to construct a 2-round protocol in the PKI model that allows m clients to compute a function of their inputs with the help of two servers (of which at most one is corrupted), where in this protocol each client sends a single message to each server and each server sends a single message to each client. The protocol only satisfies a weak notion of $1/\mathsf{poly}$ security (i.e., security with inverse-polynomial simulation error), due to the input-dependent error of the HSS scheme (inherited from the share conversion procedure of [BGI16]). The protocol can be used to succinctly evaluate branching programs. Alternatively, it can be used to evaluate general circuits (at the cost of compromising succinctness) by applying the HSS evaluation to a low-complexity randomized encoding of the circuit [Yao86, BMR90, AIK05].

The second step achieves security amplification. That is, we improve the security of the above protocol to hold with negligible simulation error, without increasing the round complexity. This is done by evaluating a *compiled* version of the desired computation, which is resilient to leakage on intermediate computation values. This compilation is obtained by using a virtual "client-server" MPC protocol to make computations locally random, where the initial messages from clients to virtual servers are HSS-shared between the two real servers, and the role of each virtual server is emulated by the two (real) servers via HSS evaluation. This virtual MPC protocol only needs to provide security against a small fraction of corrupted (semi-honest) virtual servers, but additionally needs to be *robust* in the sense that the output can still be computed even when a bounded number of virtual servers fail. The latter feature is important for coping with the error of the underlying HSS.

[2]This implies 3-round protocols in the plain model. Note, however, that unlike the first round in a general 3-round protocol, a PKI setup is *independent of the inputs and the number of parties.*

A technical issue we need to deal with is that the event of failure in the share conversion procedure is correlated not only with the input but also with bits of the secret key. To cope with this type of leakage, we modify the underlying HSS scheme to use a redundant representation of the secret key that makes leakage of a small number of bits harmless.

To make this security amplification step efficient, we need the virtual MPC protocol to have a constant number of rounds, and the next message function computed by each server in each round to be efficiently implementable by branching programs. In particular, we can use 2-round virtual MPC protocols that apply to *constant-degree polynomials* and do not require any server-to-server communication. (Again, general circuits can be handled via randomized encoding.) These protocols are sufficient for our main feasibility result of 2-round MPC from DDH. We can additionally get *succinct* 2-round protocols for NC^1 by applying a different type of virtual MPC protocol that computes NC^1 functions in a constant number of rounds with low client-to-server communication, but additionally requires (a large amount of) server-to-server communication.[3] As a corollary, we get a 2-message 2-party protocol for computing any NC^1 function $f(x, y)$ (with output delivered to one party), where the length of each message is comparable to the length of the corresponding input (and is independent of the complexity of f).

In the third and final step, we use a player virtualization technique [Bra84, HM00] to transform the 2-round (m-client) 2-server protocol into a 2-round protocol with m clients and an *arbitrary constant number of servers k*. At a high level, this is done by iteratively emulating the computations of a single server (beginning with a single server in the 2-server protocol) by two separate servers, via another level of 2-round MPC. Because of the complexity blowup in each iteration, this virtualization step can only be applied a constant number of times. Such a client-server protocol readily implies a 2-round (standard) k-party protocol by letting $m = k$ and having each party emulate the corresponding client and server.

Improving communication complexity. Under DDH, we present a secure 2-party protocol for any NC^1 or log-space computation with n input bits and m output bits using $n+(1+o(1))m+\mathsf{poly}(\lambda)$ bits of communication, where λ is a security parameter. In particular, the protocol can be used to generate n instances of $\binom{2}{1}$-oblivious-transfer (OT) of bits using $4n + o(n) + \mathsf{poly}(\lambda)$ bits of communication. This gives the first constant-rate OT protocol under DDH. Constant-rate OT protocols (with a poor concrete rate) could previously be constructed using a polynomial-stretch local pseudorandom generator [IKOS08] or the Phi-hiding assumption [IKOS09]. A similar result to ours can also be obtained under LWE, via the HSS scheme implied by [DHRW16].

The above result is obtained via a new security amplification technique, which provides a simpler and more efficient alternative to the use of virtual MPC in the second step described above. The downside is that this approach is restricted to the 2-party setting and requires an additional round of interaction. The high level idea is as follows. Denote the two parties by P_0, P_1 and assume that the functionality f delivers an output only to P_1. We rely on a Las-Vegas variant of HSS where the shared output is guaranteed to be correct (i.e., the two output shares add up to the correct output) unless P_1 outputs $\perp$, where the latter occurs with small probability. The idea is to have P_1 use $\binom{m}{m-k}$-OT for $m \gg k$ in order to block itself from the k output shares of P_0 that correspond to the positions in which it outputs $\perp$. Note that the $m - k$ selected output shares can be simulated given the correct output and the output shares of P_1, and thus they do not leak any additional

[3]Interestingly, this approach does not seem to extend to branching programs using known techniques, since in known constant-round protocols for branching programs the next message function cannot be efficiently computed by branching programs.

information about the input. To make up for the k lost output bits, we use an erasure code to encode the output. Since we can make the number of erasures small, we only need to introduce a small amount of redundancy to the output. A crucial observation which makes this approach useful is that the above form of "punctured OT" can be implemented with only $m + o(m)$ bits of communication by combining general-purpose 2PC with a puncturable pseudo-random function [SW14].

Improving computation complexity. We present several techniques for reducing the computational cost of the share conversion procedure from [BGI16], improving the concrete efficiency of group-based protocols (both in [BGI16] and the present work) by several orders of magnitude.

First, we present an optimization that improves the asymptotic *worst-case* running time of conversion by an $O(\log(1/\delta))$ factor, where δ is the error probability. In the procedure from [BGI16], a group element h is mapped to the smallest non-negative integer i such that $h \cdot g^i$ (where g is a group generator) belongs to a pseudo-random set of distinguished group elements of density δ. Allowing δ error probability, $O((1/\delta) \cdot \log(1/\delta))$ values of i should be checked, requiring a similar number of group multiplications in the worst case. While the expected number of group multiplications is $O(\log(1/\delta))$, in applications that involve "shallow" computations (where many short sequences of RMS multiplications are performed in parallel) it is the worst-case time that dominates the overall performance. The alternative approach we propose is to apply an integer-valued hash function ϕ to every group element, and return the (first) value of i in an interval of size $O(1/\delta)$ that minimizes the value of $\phi(h \cdot g^i)$. This requires only $O(1/\delta)$ group multiplications. We can also get an *unconditional* implementation of this alternative share conversion by using explicit constructions of "min-wise independent" hash functions [BCM98, Ind01].

Next, we present several optimization ideas that apply "conversion-friendly groups" towards improving the concrete running time of share conversion by several orders of magnitude. These optimizations rely on discrete-log-type assumptions in multiplicative subgroups of $\mathbb{Z}_p^*$ of a prime order q, where $p = 2q + 1$ is a prime which is close to a power of 2, and where $g = 2$ is a generator of the subgroup.We propose several concrete choices of such p. The advantage of such a group is that multiplying a group element h by the generator g can be done by shifting h by one bit to the left, and adding the difference between p and the closest power of 2 in case that the (removed) leftmost bit is 1. In fact, one can multiply h by g^w, where w is comparable to the machine word size (say, $w = 32$) by using a small constant expected number of machine word operations (64-bit additions or multiplications).

A second observation is that by making a seemingly mild heuristic assumption on the MSB sequence of the powers $h \cdot g^i$ (where h is random), it suffices to search for the first position in the sequence that contains a stretch of 0's of length $\approx \log(1/\delta)$. Concretely we need a *combinatorial* pseudo-randomness assumption asserting that such a stretch occurs roughly as often as expected in a totally random sequence.

By using an optimized "lazy" strategy for finding the first such stretch of 0's, the entire share conversion procedure can be implemented with an amortized cost of less than a single machine word operation per step. Concretely, the amortized cost is roughly 0.03 machine word additions and multiplications and 0.2 masking operations per step. This should be compared to a full group multiplication per step in the procedure of [BGI16]. Combining all the optimizations, one can perform thousands of RMS multiplications per second (using strong but standard hardware) with error probability that is small enough for homomorphically computing simple functions.

We note that the latter optimizations do not apply to Elliptic Curve groups, and hence do not provide the optimal level of succinctness. However, the gain in the computational cost of share conversion is arguably much more significant. We leave open the question of implementing similar optimizations for the case of Elliptic Curve groups.

2 Preliminaries

2.1 Restricted-Multiplication Straight-Line programs

The homomorphic evaluation of the HSS scheme from [BGI16] supports the evaluation of programs P in a computational model known as *Restricted Multiplication Straight-line (RMS)* program [Cle91, BGI16].

Definition 2.1 (RMS programs)**.** The class of *Restricted Multiplication Straight-line (RMS)* programs consists of a magnitude bound 1^M and an arbitrary sequence of the four following instructions, each with a unique identifier id:

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

We assume that the RMS program complies with the given bound M in the sense that for every input w, the values of all intermediate memory values in the RMS execution on w are at most M. (For simulating standard branching programs, it suffices to let $M = 1$.) We define the *size* of an RMS program P as the number of its 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) [BGI16].

2.2 Circular Security

We present a simplified (weaker) version of the more general notion of key-dependent security as introduced by Black *et al.* [BRS02].

Definition 2.2 (Circular Security)**.** We say that a public-key encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ with secret key length $\ell(\lambda)$ and message space containing $\{0,1\}$ is *circular secure* if there exists a negligible function $\nu(\lambda)$ for which the following holds for every nonuniform polynomial-time $\mathcal{A}$:

$$\Pr\left[\begin{array}{c}(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda),\\ b \leftarrow \{0,1\},\\ b' \leftarrow \mathcal{A}^{\mathcal{O}_b}(\mathsf{pk})\end{array} : \ b' = b\right] \leq \frac{1}{2} + \nu(\lambda),$$

where the oracle $\mathcal{O}_b$ takes no input and outputs the following (where $\mathsf{sk}^{(i)}$ denotes the ith bit of sk):

$$(C_1, \ldots, C_\ell), \text{ where } \begin{cases}\forall i \in [\ell], C_i \leftarrow \mathsf{Enc}(\mathsf{pk}, 0) & \text{if } b = 0\\ \forall i \in [\ell], C_i \leftarrow \mathsf{Enc}(\mathsf{pk}, \mathsf{sk}^{(i)}) & \text{if } b = 1\end{cases}.$$

We remark that circular security implies standard semantic security.

2.3 Puncturable Pseudorandom Functions

Let $PRF_K(\cdot)$ be a pseudorandom function family with input space $\{0,1\}^*$, output space $\{0,1\}$ and key $K \in \{0,1\}^\lambda$. We call PRF_K a *puncturable PRF family* if there exists a PPT algorithm Puncture that satisfies the following properties:

- **Functionality preserved under puncturing.** Puncture takes as input a PRF key $K \in \{0,1\}^\lambda$, an input length 1^d, set of punctured inputs $X = \{x_1, \ldots, x_k\} \subseteq \{0,1\}^d$ and outputs K_X such that for all $x' \in \{0,1\}^d \setminus X$, $PRF_{K_X}(x') = PRF_K(x')$.

- **Pseudorandom at punctured points.** For every non-uniform polynomial-time adversary $(\mathcal{A}_1, \mathcal{A}_2)$ such that $\mathcal{A}_1(1^\lambda)$ outputs an input set $X = \{x_1, \ldots, x_k\} \subseteq \{0,1\}^d$, consider an experiment where $K \overset{\$}{\leftarrow} \{0,1\}^\lambda$ and $K_X \leftarrow \mathsf{Puncture}(K, X)$. Then there is a negligible function μ such that for all λ, $|\Pr[\mathcal{A}_2(K_X, X, (PRF_K(x_1), \ldots, PRF_K(x_k)) = 1] - Pr[\mathcal{A}_2(K_X, X, U_k) = 1]| \leq \mu(\lambda)$ where U_k is a string drawn uniformly at random from $\{0,1\}^k$.

As observed by [BW13, BGI14, KPTZ13], the GGM construction [GGM86] of PRFs from one-way functions yields puncturable PRFs.

Theorem 2.3 ([GGM86, BW13, BGI14, KPTZ13])**.** *Assuming the existence of a one-way function, there exists a puncturable PRF with key size $|K_X| = O(\lambda kd)$. The PRF can be evaluated at all points given K or all non-punctured point given K_X by using $O(2^d)$ invocations of a PRG $G : \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$. The circuit size required for generating K_X given a λ-bit K and X is $kd \cdot poly(\lambda)$.*

2.4 Homomorphic Secret Sharing and DEHE

As in [BGI16], we consider the case of 2-out-of-2 secret sharing, where an algorithm Share is used to split a secret $w \in \{0,1\}^n$ into two shares, such that each share computationally hides w. The homomorphic evaluation algorithm Eval is used to locally evaluate a program $P \in \mathcal{P}$ on the two shares, such that the two outputs of Eval add up to $P(w)$ modulo a positive integer β (where $\beta = 2$ by default), except with δ error probability. The running time of Eval is polynomial in the size of P and $1/\delta$. Here we formalize a stronger "Las Vegas" notion of HSS where Eval may output $\perp$ with at most δ probability, and the output is guaranteed to be correct as long as no party outputs $\perp$.

Definition 2.4 (Homomorphic Secret Sharing: Las Vegas Variant)**.** A (2-party) *Las Vegas Homomorphic Secret Sharing (*HSS*)* scheme for a class of programs $\mathcal{P}$ consists of algorithms $(\mathsf{Share}, \mathsf{Eval})$ with the following syntax:

- $\mathsf{Share}(1^\lambda, w)$: On security parameter 1^λ and $w \in \{0,1\}^n$, the sharing algorithm outputs a pair of shares $(\mathsf{share}_0, \mathsf{share}_1)$. We assume that the input length n is included in each share.

- $\mathsf{Eval}(b, \mathsf{share}, P, \delta, \beta)$: On input party index $b \in \{0,1\}$, share share (which also specifies an input length n), a program $P \in \mathcal{P}$ with n input bits and m output bits, an error bound $\delta > 0$ and integer $\beta \geq 2$, the homomorphic evaluation algorithm either outputs $y_b \in \mathbb{Z}_\beta^m$, constituting party b's share of an output $y \in \{0,1\}^m$, or alternatively outputs $\perp$ to indicate failure. When β is omitted it is understood to be $\beta = 2$.

The algorithm Share is a PPT algorithm, whereas Eval can run in time polynomial in its input length and in $1/\delta$. The algorithms $(\mathsf{Share}, \mathsf{Eval})$ 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 $w \in \{0,1\}^n$, program $P \in \mathcal{P}$ with input length n, error bound $\delta > 0$ and integer $\beta \geq 2$, where $|P|, 1/\delta \leq p(\lambda)$, we have

$$\Pr[(\mathsf{share}_0, \mathsf{share}_1) \leftarrow \mathsf{Share}(1^\lambda, w); y_b \leftarrow \mathsf{Eval}(b, \mathsf{share}_b, P, \delta, \beta),\ b = 0,1 : (y_0 = \perp) \vee (y_1 = \perp)] \leq \delta + \nu(\lambda),$$

  and

$$\Pr[(\mathsf{share}_0, \mathsf{share}_1) \leftarrow \mathsf{Share}(1^\lambda, w); y_b \leftarrow \mathsf{Eval}(b, \mathsf{share}_b, P, \delta, \beta),\ b = 0,1 : (y_0 \neq \perp) \wedge (y_1 \neq \perp) \wedge y_0 + y_1 \neq P(w)] \leq \nu(\lambda),$$

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

- **Security:** Each share keeps the input semantically secure.

We will also use a stronger *asymmetric* version of Las Vegas HSS where only one party (say, P_1) may output $\perp$. This is defined similarly to the above, except that conditions $y_0 = \perp$ and $y_0 \neq \perp$ in the correctness requirement are removed.

2.4.1 Multi-Evaluation HSS

Similarly to [BGI16] (cf. full version Definition 4.2), it is useful to rely on a stronger flavor of HSS which guarantee that when running polynomially many instances of Eval, the events of error are indistinguishable from being independent. (We do not achieve full independence because all instances share a common PRF key used for share conversion.) To keep Eval deterministic, we include an additional input id where the independence condition is guaranteed as long as all values of id are distinct. Here we need a variant that applies to the notion of (simulatable) Las Vegas HSS. We formalize it for the case of standard Las Vegas HSS, and then describe the required modification for the simulatable variant.

Definition 2.5 (Multi-Evaluation Las Vegas HSS)**.** A (2-party) *Multi-Evaluation Las Vegas* HSS scheme for a class of programs $\mathcal{P}$ consists of algorithms $(\mathsf{Share}, \mathsf{Eval})$ that have the same syntax as standard Las Vegas HSS (Definition 2.4) except that Eval has an additional input id. The algorithm Share should satisfy the same security requirement as in Definition 2.4. The algorithm Eval should satisfy the same correctness requirement as in Definition 2.4 (for every choice of id) and the following additional independence requirement.

- **Independence of failures:** For every $b \in \{0,1\}$, polynomials m, s and nonuniform polynomial-time distinguisher $\mathcal{A}$ there is a negligible function ν such that the following holds. For every positive integer λ, input $w \in \{0,1\}^n$, programs $P_1, \ldots, P_{m(\lambda)} \in \mathcal{P}$ of size $s(\lambda)$ with input length n, error bound $\delta > 0$, integer $\beta \geq 2$ and distinct identifiers $\mathsf{id}_1, \ldots, \mathsf{id}_{m(\lambda)} \in \{0,1\}^\lambda$, there are error probabilities $p_1, \ldots, p_{m(\lambda)} \leq \delta$ such that the advantage of $\mathcal{A}$ in distinguishing between the outputs of the following two experiments is at most $\nu(\lambda)$:

- **Experiment 1**: Output a bit sequence $\tau_1, \ldots, \tau_{m(\lambda)}$ where $\Pr[\tau_i = 1] = p_i$ and the τ_i are statistically independent.
- **Experiment 2**:
  - $(\mathsf{share}_0, \mathsf{share}_1) \leftarrow \mathsf{Share}(1^\lambda, w)$;
  - For $i = 1, \ldots, m(\lambda)$: Let $y_b^i \leftarrow \mathsf{Eval}(\mathsf{id}_i, b, \mathsf{share}_b, P_i, \delta, \beta)$; Output 1 if $y_b^i = \perp$ and 0 otherwise.

One can similarly define a multi-evaluation variants of the other flavors of HSS, including the simulatable Las Vegas variant. In the latter case, we make the additional requirement that the events of outputting $\top$ are indistinguishable from being independent. This is formalized similarly to the above.

2.5 Distributed Evaluation Homomorphic Encryption (DEHE)

In this section we define a public-key variant of HSS from [BGI16], referred to as Distributed Evaluation Homomorphic Encryption (DEHE). A DEHE scheme supports homomorphic computations on inputs contributed by different clients using a common public key and secret evaluation keys. The corresponding security notion guarantees computational secrecy of an encrypted input given the public key, the ciphertext, and evaluation key of any single server. In a setting consisting of two servers and an arbitrary number of clients, the above security notion implies that inputs contributed by a set of uncorrupted clients remain secure even if one of the two servers colludes with all the remaining clients.

Definition 2.6 (Distributed-Evaluation Homomorphic Encryption)**.** A (2-party, $1/\mathsf{poly}$-error) *Distributed Evaluation Homomorphic Encryption (DEHE)* 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 $(\mathsf{ek}_0, \mathsf{ek}_1)$.
- $\mathsf{Enc}(\mathsf{pk}, w)$: On a public key pk and a secret input value $w \in \{0,1\}$, the encryption algorithm outputs a ciphertext 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, vector of n ciphertexts, a program $P \in \mathcal{P}$ with n input bits and m output bits, error 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 $y \in \{0,1\}^m$. When β is omitted it is understood to be $\beta = 2$.

The algorithms Gen and 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 $(w_1, \ldots, w_n) \in \{0,1\}^n$, program $P \in \mathcal{P}$ with input length n, error bound $\delta > 0$ and integer $\beta \geq 2$, where $|P|, 1/\delta \leq p(\lambda)$, we have

$$\Pr\left[\begin{array}{c} (\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \leftarrow \mathsf{Gen}(1^\lambda); \\ (\mathsf{ct}_1, \ldots, \mathsf{ct}_n) \leftarrow (\mathsf{Enc}(\mathsf{pk}, w_1), \ldots, \mathsf{Enc}(\mathsf{pk}, w_n)); \\ y_b \leftarrow \mathsf{Eval}(b, \mathsf{ek}_b, (\mathsf{ct}_1, \ldots, \mathsf{ct}_n), P, \delta, \beta),\ b = 0,1 \end{array} : y_0 + y_1 = P(w_1, \ldots, w_n)\right] \geq 1 - \delta - \nu(\lambda),$$

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

- **Security:** For $b = 0, 1$, the two distribution ensembles $C_0(\lambda)$ and $C_1(\lambda)$ are computationally indistinguishable, where $C_w(\lambda)$ is obtained by letting $(\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \leftarrow \mathsf{Gen}(1^\lambda)$ and outputting $(\mathsf{pk}, \mathsf{ek}_b, \mathsf{Enc}(\mathsf{pk}, w))$.

Similarly to the case of HSS, one can define the multi-evaluation and Las Vegas variants of DEHE in the natural way.

2.6 The BGI Construction [BGI16]

The work of [BGI16] constructs 2-party HSS (and DEHE) that directly supports homomorphic evaluation of "Restricted-Multiplication Straight-line" (RMS) programs over small integers. Such programs support four operations: Load Input to Memory, Add Values in Memory, Multiply Input by Memory Value, and Output Value. (See Definition 2.1 for RMS syntax.) We provide here a high-level description of the [BGI16] construction, which serves as a starting point for many of our results. In what follows, let $\mathbb{G}$ be a DDH-hard group of prime order q with generator $g \in \mathbb{G}$, and let $\ell = \lceil \log q \rceil$. We begin with the BGI construction of HSS based on circular-secure ElGamal:

Secret shares: To secret share a (small integer) input w, the BGI construction samples an ElGamal key pair $(c, e = g^c) \in \mathbb{Z}_q \times \mathbb{G}$, and outputs shares as follows: (1) Each party gets an additive secret share over $\mathbb{Z}_q$ of the input w and of the product cw (viewed as an element of $\mathbb{Z}_q$). (2) Each party also gets (copies of the same) $(\ell + 1)$ ElGamal ciphertexts, one encrypting w and one encrypting each product $c^{(t)}w$ of w with the tth bit of the secret key for $t \in [\ell]$.

Homomorphic evaluation: Evaluation maintains the invariant that (after each instruction) for each memory value x in the RMS program execution, the value of x and of cx are each held as an additive secret sharing across the two parties. This directly holds for any "Load Input to Memory" instruction, and can straightforwardly be achieved for each "Add Values in Memory" instruction by linear homomorphism of additive secret shares. "Output Value From Memory" to a target group $\mathbb{Z}_\beta$ (for some integer $\beta \le q$ specified in the RMS program) is achieved by having each party shift his current share of the relevant memory value by a common rerandomization value and then output this share mod β.

The primary challenge is in supporting "Multiply Input by Value in Memory." Recall in such situation the parties hold additive secret shares of x and cx for the memory value x, and ElGamal ciphertexts of w and $\{c^{(t)}w\}_{t\in[\ell]}$ for the input w. Evaluation takes place in two steps, repeated for each ciphertext; for example, for the ciphertext encrypting w, we convert the common ElGamal ciphertext of w and additive secret shares of x and cx to additive secret shares of wx:

1. Use additive secret shares of x and cx to perform distributed ElGamal decryption via "linear algebra in the exponent," yielding multiplicative secret shares of g^{wx}. For ciphertext (g^r, g^{cr+w}), the multiplicative share of g^{wx} is $(g^r)^{-[\text{share of cx}]}(g^{cr+w})^{[\text{share of x}]}$.

2. To return the computed shares of g^{wx} back to additive shares of wx, the parties execute a share conversion procedure referred to as "Distributed Discrete Log," wherein the parties output the distance (measured by powers of g) of their share value g^{z_b} from the nearest point in an agreed-upon "distinguished set" in $\mathbb{G}$. Error occurs in this step if parties output with respect to *different* distinguished points, which occurs if a distinguished point lies "between" the parties' two shares $g^{z_0}, g^{z_1} = g^{z_0+wx}$.

A tradeoff between computation and error can be made, by decreasing the density of distinguished points δ, and scaling computation as $1/\delta$; the resulting error probability is roughly δM, where M is the maximal value of the "payload" wx (corresponding to the "distance" between the parties' shares).

By repeating the above 2 steps for w and for each $c^{(t)}w$, the parties receive additive secret shares of wx and of each $c^{(t)}wx$. As a final step, the shares of $\{c^{(t)}wx\}_{t\in[\ell]}$ are combined by the appropriate powers-of-2 linear combination to yield a single set of additive shares of cwx, yielding the desired invariant for the new memory value wx.

Remark 2.7 (Obtaining DEHE)**.** The DEHE construction from [BGI16] is obtained by modifying the HSS construction as follows. Recall that the DEHE setup results in a public key pk and evaluation keys $\mathsf{ek}_0, \mathsf{ek}_1$. In the BGI construction, $\mathsf{ek}_0, \mathsf{ek}_1$ are additive secret shares of a system secret key c (which corresponds to 1 being "loaded to memory"); and pk contains the corresponding ElGamal public key e as well as encryptions of the bits $c^{(t)}$ of the secret key. To "encrypt" his input w into the system, a client can convert the encryptions of $c^{(t)}$ to encryptions of $c^{(t)}w$ using the homomorphic properties of ElGamal. The only change to homomorphic evaluation is that "Load Input to Memory" is no longer immediate: instead, additive secret shares of w must be *computed*, through the same process as before for multiplying the value 1 loaded into memory with the input value w.

Remark 2.8 (Removing the ElGamal circular security assumption)**.** This can be done by one of two methods: (1) a standard "leveled" approach, using a sequence of secret keys (growing the HSS share size by the depth of computation); alternatively, (2) by replacing ElGamal with the "BHHO" encryption scheme of Boneh, Halevi, Hamburg, and Ostrovsky [BHHO08], which is *provably* circular secure based on DDH. Roughly, BHHO ciphertexts are an $O(\lambda)$-element extension of ElGamal, where the first elements are of the form $g_1^r, \ldots, g_\ell^r$ (for fixed generators $g_1, \ldots, g_\ell$ and encryption randomness r), and the final element contains the message as g^{msg} masked by a subset-product of the previous elements as dictated by the secret key $s \in \{0,1\}^\ell$. In particular, BHHO decryption follows a direct analog of "linear algebra in the exponent" as in ElGamal, and thus can be leveraged in the same manner within homomorphic share evaluation, where the new invariant for each memory value x is holding additive secret shares of x as well as each product $s_t x$, for the secret key bits s_t, $t \in [\ell]$. In addition, BHHO supports the same form of plaintext homomorphism required for DEHE, as discussed above. We refer the reader to [BGI16] for a detailed formal treatment. We note that for the purpose of optimizing concrete efficiency, one can simply assume the ElGamal encryption scheme to be circularly secure. Indeed, this is the assumption we make for the purpose of optimizing the computational cost in Section 6.3. We will also use the circular security assumption to simplify the presentation of the main results of the paper; however, all of our asymptotic theorem statements can be based on DDH by using the BHHO encryption instead of ElGamal encryption.

2.7 Secure Multiparty Computation

We consider two types of protocols for secure multiparty computation (MPC): standard k-party MPC protocols and client-server protocols. We refer the reader to [Can00, Gol04] for standard definitions of MPC protocols and only highlight here the aspects that are particularly relevant to this work.

In a standard MPC protocol there are k parties who interact with each other in order to compute a function of their inputs. We say that such a protocol is *secure* if it is computationally secure against a static, passive adversary who may corrupt any strict subset of the parties. We use 2PC to refer to the case $k = 2$.

Client-server protocols. In a client-server protocol there are m clients and k servers. Only the clients have inputs and get an output. Clients and servers can communicate over secure point-to-point channels. We assume protocols in the client-server model to take the following canonical form: in the first round each client sends a message to each server. Then there may $r \geq 0$ rounds of interaction in which each server can send a message to each other server. We assume the servers to be deterministic, so that every message sent by a server in a given round is determined by the messages it received in previous rounds. Finally, there is an output reconstruction round in which each server sends a message to each client, and where each client computes an output by applying a local decoding function to the k messages it received.

We specify such a client-server protocol by $\Pi = (\mathsf{Encode}, \mathsf{NextMsg}, \mathsf{Decode})$, where $\mathsf{Encode}(i, x_i)$ is a randomized function mapping the input of Client i to the k messages it sends in the first round, $\mathsf{NextMsg}(i, \vec{m})$ is a next message function which determines the messages sent by Server i in in the current round given the messages $\vec{m}$ it received in previous rounds, and $\mathsf{Decode}(i, \vec{m})$ denote the output of Client i given the messages $\vec{m}$ it received in the final round. Finally, we will consider by default protocols for functionalities that deliver the same output to all clients. In such a case, we can assume that each server sends the same message to all clients, and $\mathsf{Decode}(i, \cdot)$ is the same for all i.

Security and robustness. We say that Π is a *t-secure* protocol for f if it is secure against a static, passive (semi-honest) adversary who may corrupt any set of parties that includes at most t servers and an arbitrary number of clients. Security is defined by the existence of a simulator $\mathsf{Sim}(1^\lambda, T, 1^n, y)$ that given a security parameter λ (in the computational case), a set T of corrupted parties, an input length n, and an output y of f (in the case at least one client is corrupted) outputs a simulated view of the parties in T. Simulation should be either perfect or computational, depending on the type of security. We assume *computational* $(k-1)$-security by default, but will also consider protocols that offer perfect t-security for smaller values of t. Note that any secure k-client k-server protocol for f implies a standard k-party MPC for f by letting Party i simulate both Client i and Server i.

A *t-robust* protocol for f is a t-secure protocol with the following additional feature: the clients obtain the correct output of f even if t servers fail to send messages. Equivalently, the function Decode outputs the correct output of f at the end of the protocol execution even if up to t of its inputs are replaced by $\perp$.

Succinct MPC. We will consider MPC protocols for a class of programs $\mathcal{P}$, where all parties are given a "program" $P \in \mathcal{P}$ (say, a boolean circuit, boolean formula or branching program) as an input, and their running time should be polynomial in the size of P. See Section 4 of [BGI16] for a full definition. We refer to an MPC protocol for $\mathcal{P}$ as being *succinct* if the communication complexity is bounded by a fixed polynomial in the total length of inputs and outputs and the security parameter, independently of the program size.

MPC with PKI setup. For both flavors of MPC protocols, we consider round complexity with a public key infrastructure (PKI) setup. A PKI setup allows a one-time global choice of parameters $\mathsf{params} \leftarrow \mathsf{ParamGen}(1^\lambda)$, followed by independent choices of a key pair $(\mathsf{sk}_i, \mathsf{pk}_i) \leftarrow$

$\mathsf{KeyGen}(1^\lambda, \mathsf{params})$ by each party P_i.[4] We assume that each party knows the public keys of all parties with whom it wants to interact as well as its own secret key. Note that the public keys are generated independently of any inputs or even the number of other parties in the system. For this reason we do not count the PKI setup towards the round complexity of our protocols.

3 Black-Box Client-Server HSS and MPC

In order to use HSS or its public-key DEHE variant to obtain secure computation, the secret sharing procedure (or DEHE key setup) must be performed in a secure distributed fashion. Applying general-purpose secure computation to do so, as suggested in [BGI16], has poor concrete efficiency and requires non-black-box access to the underlying group.

To avoid this, we introduce the notion of *client-server HSS* (Π, Eval), defined as standard HSS, except that the input is distributed between multiple clients and the centralized sharing algorithm Share is replaced by a distributed protocol Π. That is, Π allows m clients, each holding a secret input w_i, to share the joint input $(w_1, \ldots, w_m)$ between the servers in a way that supports homomorphic computations via Eval. We will be interested in constructing client-server HSS (and DEHE) that only make a black-box access to the underlying group.

Intuitively, in our construction of the joint secret sharing protocol Π, each client C_i will generate an independent ElGamal key pair (c_i, e_i), and the joint keys of the system will correspond to $c = \sum c_i \in \mathbb{Z}_q$ and $e = \prod e_i \in \mathbb{G}$, leveraging the key homomorphism of ElGamal. The primary challenge (mirroring the BGI HSS) is how to generate encryptions of the products $c^{(t)} w_i$, where $c^{(t)}$ are the bits of the *joint* secret key $c = \sum c_i$ (where addition is in $\mathbb{Z}_q$). To solve this, we leverage the fact that the BGI construction does not strictly require $\{0,1\}$ values for this $c^{(t)}$, but rather can support computations on any sufficiently small values at the expense of greater computation during the share conversion procedure. We will thus use the (possibly non-Boolean) values $\sum_i c_i^{(t)}$ in the place of $c^{(t)}$.

In the following subsections, we provide a formal definition of client-server HSS (and DEHE), we present our corresponding black-box client-server DEHE construction, and then we demonstrate how to build on top of this construction to obtain succinct secure computation of branching programs which makes black-box use of the DDH-hard group.

3.1 Defining Client-Server HSS & DEHE

We will restrict our attention to 2-server protocols. The security requirement is that the view of an adversary who corrupts a subset of clients/servers, leaving at least one client and one server uncorrupted, can be simulated given the inputs of corrupted clients, *without* knowledge of the inputs of uncorrupted clients. A "multi-evaluation" version enables independent executions of Eval without re-executing Π.

Definition 3.1 (Client-Server HSS)**.** An m-client 2-server ($1/\mathsf{poly}$-error) *Client-Server Homomorphic Secret Sharing (HSS)* for a class of programs $\mathcal{P}$ consists of a distributed protocol Π and PPT algorithm Eval, with the following syntax:

[4] We will only use params to specify a group for ElGamal encryption; hence, we can let params be a common random string, or even pick params deterministically under a suitable variant of DDH.

- Π specifies an interactive protocol between m clients $C_1, \ldots, C_m$ and two servers S_0, S_1, where each client C_i begins with input w_i, and in the end of executing Π the servers S_0, S_1 output homomorphic secret shares $\mathsf{share}_0, \mathsf{share}_1$, respectively, of the joint input $(w_1, \ldots, w_m)$.

- $\mathsf{Eval}(b, \mathsf{share}, P, \delta, \beta)$ has the same syntax as in standard HSS. That is, on input party index $b \in \{0,1\}$, share share (which also specifies an input length n), a program $P \in \mathcal{P}$ with n input bits and ℓ output bits, an error bound $\delta > 0$ and integer $\beta \geq 2$, the homomorphic evaluation algorithm either outputs $y_b \in \mathbb{Z}_\beta^\ell$, constituting party b's share of an output $y \in \{0,1\}^\ell$, or alternatively outputs $\perp$ to indicate failure. When β is omitted it is understood to be $\beta = 2$.

The pair (Π, Eval) should satisfy the following correctness and security properties:

- **Correctness:** For every polynomial p there is a negligible ν such that for every positive integer λ, input $\vec{w} = (w_1, w_2, \ldots, w_m)$ of total length n, program $P \in \mathcal{P}$ with input length n, error bound $\delta > 0$ and integer $\beta \geq 2$, where $|P|, 1/\delta \leq p(\lambda)$, we have

$$\Pr[(\mathsf{share}_0, \mathsf{share}_1) \leftarrow \Pi(1^\lambda, \vec{w}); y_b \leftarrow \mathsf{Eval}(b, \mathsf{share}_b, P, \delta, \beta),\ b = 0,1 : \\ y_0 + y_1 \neq P(\vec{w})] \leq \delta + \nu(\lambda),$$

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

- **Security:** For $\mathsf{Corrupt} \subset \{C_1, \ldots, C_m\} \cup \{S_0, S_1\}$ and $\vec{w} = (w_1, \ldots, w_m) \in \{0,1\}^n$, let $\mathsf{view}(1^\lambda, \mathsf{Corrupt}, \vec{w})$ denote the joint view of the corrupted parties within the following experiment:

  - Clients $C_1, \ldots, C_m$ interact as prescribed by Π, with inputs $w_1, \ldots, w_m$, respectively, using fresh randomness. Each server S_b outputs share_b.

  Then security requires that there exist a PPT simulator Sim such that for any corrupted set $\mathsf{Corrupt} \subset \{C_1, \ldots, C_m\} \cup \{S_0, S_1\}$ of clients and servers with $\{C_1, \ldots C_m\}, \{S_0, S_1\} \not\subset \mathsf{Corrupt}$ (i.e., for which at least one server and one client are uncorrupted), every polynomial p, and sequence of input vectors $\vec{w}^\lambda = (w_1^\lambda, \ldots, w_m^\lambda) \in (\{0,1\}^*)^m$ such that $|w_i^\lambda| \leq p(\lambda)$, it holds that $\mathsf{view}(1^\lambda, \mathsf{Corrupt}, \vec{w}^\lambda) \overset{c}{\cong} \mathsf{Sim}(1^\lambda, \mathsf{Corrupt}, \{w_i\}_{C_i \in \mathsf{Corrupt}}, \{|w_i|\}_{C_i \notin \mathsf{Corrupt}})$.

For completeness, we define a "public key" version of the above, which we refer to as client-server DEHE.

Definition 3.2 (Client-Server DEHE)**.** An m-client 2-server (1/poly-error) *Client-Server Distributed-Evaluation Homomorphic Encryption (DEHE)* for a class of programs $\mathcal{P}$ consists of a distributed protocol Π and algorithms $\mathsf{Enc}, \mathsf{Eval}$, with the following syntax.

- Π specifies an interactive protocol between m clients $C_1, \ldots, C_m$ and two servers S_0, S_1 whose output is a public key pk to each client, and an evaluation key $\mathsf{ek}_0, \mathsf{ek}_1$ to each respective server S_0, S_1.

- $\mathsf{Enc}, \mathsf{Eval}$ have the same syntax as in standard DEHE (see Definition 2.6).

The tuple $(\Pi, \mathsf{Enc}, \mathsf{Eval})$ satisfies the same correctness property as in standard DEHE, as well as the following security property:

- **Security:** Intuitively, for any corrupted set of clients and at most one server, there exists a PPT simulator who simulates the collective view of the corrupt entities in Π, together with an encrypted input $w \in \{0,1\}$, without knowledge of w.

  Namely, for $\mathsf{Corrupt} \subset \{C_1, \dots, C_m\} \cup \{S_0, S_1\}$ and $w \in \{0,1\}$, let $\mathsf{view}(1^\lambda, \mathsf{Corrupt}, w)$ denote the joint view of the respective clients and/or server within the following experiment:

  - Clients $C_1, \dots, C_m$ interact as prescribed by Π, using fresh randomness; each client learns pk and each server S_b learns ek_b.
  - The input w is encrypted with the resulting public key, as $\mathsf{ct} \leftarrow \mathsf{Enc}(\mathsf{pk}, w)$. The ciphertext ct is given to all clients and servers.

  Then security requires that there exists a PPT simulator Sim such that for any corrupted set $\mathsf{Corrupt} \subset \{C_1, \dots, C_m\} \cup \{S_0, S_1\}$ of clients and servers with $\{C_1, \dots C_m\}, \{S_0, S_1\} \not\subset \mathsf{Corrupt}$ (i.e., for which at least one server and one client is uncorrupted), and every input $w \in \{0,1\}$, it holds $\mathsf{view}(1^\lambda, \mathsf{Corrupt}, w) \overset{c}{\cong} \mathsf{Sim}(1^\lambda, \mathsf{Corrupt})$.

3.2 Black-Box Client-Server HSS Construction

We attain a *black-box* client-server HSS (in fact, even the stronger notion of multi-evaluation client-server DEHE), building from the HSS of BGI [BGI16] by leveraging three homomorphic properties of ElGamal:

1. Key hom: Given key pairs (c_1, e_1) and (c_2, e_2), then encryptions under public key $e_1 \cdot e_2$ correspond to secret key $c_1 + c_2$.

2. Plaintext hom 1: Given ElGamal encryptions of plaintexts x_1 and x_2, under the *same* public key e, their componentwise product is an encryption of the plaintext $x_1 + x_2$ under e.

3. Plaintext hom 2: Given an ElGamal encryption (h_1, h_2) of unknown value x under public key e, together with a known value w, one can generate a (freshly distributed) encryption of the product wx.

Recall that in the BGI construction, the secret sharing of an input w consists of: (1) additive secret share over $\mathbb{Z}_q$ of the input w and of the product cw, for a system ElGamal secret key c; (2) identical copies of $(\ell+1)$ ElGamal ciphertexts, one encrypting w and one encrypting each product $c^{(t)}w$ of w with the t'th bit of the secret key, for $t \in [\ell]$. (See discussion in Section 2.6 for a more detailed overview.)

The construction. Within the joint secret sharing protocol Π, each client C_i will generate an independent ElGamal key pair (c_i, e_i), and the joint keys of the system will correspond to $c = \sum c_i \in \mathbb{Z}_q$ and $e = \prod e_i \in \mathbb{G}$, leveraging the key homomorphism of ElGamal. Additive secret shares of the combined key c can be obtained by having each party additively secret share his key c_i. Given the collection of public keys $\{e_i\}$, each client can directly encrypt his input w_i under the combined public key $e = \prod e_i$. It then remains to generate encryptions of the products $c^{(t)}w_i$, where $c^{(t)}$ are the bits of the *joint* secret key $c = \sum c_i$ (where addition is in $\mathbb{Z}_q$). Note that by Plaintext homomorphism 2 (above), it suffices to generate encryptions of the bits $c^{(t)}$ themselves (since client C_i can convert such an encryption to one of $c^{(t)}w_i$, given w_i in the clear).

The other plaintext-homomorphism property of ElGamal (above) gets us partway toward this goal. Each client C_j can encrypt the bits $c_j^{(t)}$ of *his own* secret key under the joint key e, which can be combined via Plaintext homomorphism 1 (above) to an encryption of $\sum_{j\in[m]} c_j^{(t)}$ under e. Note that this is not the same as the t'th bit of the combined key $\sum_{j\in[m]} c_j$ (indeed, this value likely will not even lie in $\{0,1\}$). However, we observe that the BGI construction does not strictly require $\{0,1\}$ values for this $c^{(t)}$, but rather used the bit representation for convenience and can support computations on any sufficiently small values at the expense of greater computation during the share conversion procedure.

As such, we can replace the role of the *bits* of the joint key $c^{(t)}$ with any combination of small-magnitude values which can be combined over $\mathbb{Z}_q$ to reconstruct c. In our case, we will use exactly the sums $c_{\mathsf{sum}}^{(t)} = \sum_{i\in[m]} c_i^{(t)} \in \{0,\dots,m\}$ of the bits of the keys c_i of the m clients. We have $c = \sum_{t=1}^{\ell} 2^{t-1} c_{\mathsf{sum}}^{(t)} \in Z_q$ as desired, and each sum is bounded in magnitude by the number of clients m. This means that if each intermediate memory value in the computation of program P is bounded in size by M, then the maximum magnitude of multiplicatively values we will need to convert to additive shares will be mM, corresponding to the product $c_{\mathsf{sum}}^{(t)} y$ for some memory value y.

A formal description of our black-box client-server DEHE construction is given in Figure 1.

In our construction, we make use of the notations of [BGI16] to denote ElGamal ciphertexts, additive secret shares, and "multiplicative" secret shares, as well as two of their "share pairing" operations $\mathsf{MultShares}$, $\mathsf{ConvertShares}$, and their core share conversion sub-routine $\mathsf{DistributedDLog}$. Definitions of these items are given in Figure 2.

Remark 3.3 (ElGamal Circular Security vs. DDH)**.** For simplicity, throughout the present work we describe our constructions based on circular security of ElGamal. However, in each case we may directly remove this circular security assumption, as in [BGI16], by either considering a leveled variant or replacing ElGamal with a circular-secure variant due to BHHO [BHHO08], as described in Remark 2.8. Our theorem statements implicitly apply this transformation directly.

Proposition 3.4 (Black-box client-server DEHE)**.** *Assuming circular security of ElGamal, there exists a client-server DEHE protocol for branching programs that makes a black-box access to any DDH-hard group.*

Remark 3.5 (Multi-eval black-box client-server DEHE)**.** We can directly obtain a multi-evaluation version of the construction, as per Section 2.4.1, where several independent executions of Eval can take place cheaply without requiring further communication. This is because the randomness of Eval is determined by a pseudo-random function ϕ, which can be leveraged to give "fresh" additional randomness for a second execution by simply evaluating ϕ each time on a disjoint set of its input space.

Proof. Correctness. The ultimate construction $(\Pi, \mathsf{Enc}, \mathsf{Eval})$ differs from the BGI DEHE construction $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval})$ in only two aspects: (1) In BGI, the ciphertexts $[\![c^{(t)}]\!]_c$ in the public key corresponded to encryptions of *bits* of the secret key c, whereas here they correspond to larger-magnitude values (up to m) satisfying the same linear reconstruction property, and (2) Each execution of $\mathsf{ConvertShares}$ within homomorphic Eval is run with magnitude bound mM, instead of M. The only difference introduced by (1) is that the share conversion procedure will err with higher probability, scaling with the new larger magnitude; however, this is directly taken care of by (2), in effect increasing the runtime complexity of $\mathsf{DistributedDLog}$ to maintain the same error guarantees.

Black-Box Client-Server DEHE: m-client protocol Π.
Inputs: Global parameters $\mathbb{G}, g, q$. Let $\ell := \lceil \log q \rceil$.
Outputs: All clients learn pk. Each server $b \in \{0,1\}$ learns evaluation key ek_b.

Client Round 1: Each client $i \in [m]$ performs the following:

1. Sample an ElGamal secret key $c_i \leftarrow \mathbb{Z}_q$.
2. Compute the corresponding ElGamal public key $e_i = g^{c_i} \in \mathbb{G}$ and send e_i to all clients.
3. Secret share $\langle c_i \rangle \leftarrow \mathsf{AdditiveShare}(c_i)$ and send each resulting share $\langle c_i \rangle_b$ to the corresponding server b.
4. Sample a random string $r_i \leftarrow \{0,1\}^\lambda$ and send r_i to both servers.

Client Round 2: Each client $i \in [m]$ performs the following:

1. Let $e = \prod_{j \in [m]} e_j$, where e_j is the value sent by client j in the previous round.
2. Each party P_i encrypts the bits of his respective secret key c_i: namely, for every $t = 1, \ldots, \ell$, let $[\![c_i^{(t)}]\!]_c \leftarrow \mathsf{Enc}_{\mathsf{ElGamal}}(e, c_i^{(t)})$. (Recall ElGamal encryption is black-box in $\mathbb{G}$.)
3. Send $([\![c_i^{(t)}]\!]_c)_{t \in [\ell]}$ to all clients.

Client Output: Each client $i \in [m]$ performs the following:

1. For each bit $t \in [\ell]$ of the secret key, combine ciphertexts as $[\![c^{(t)}]\!]_c = \prod_{j \in [m]} [\![c_j^{(t)}]\!]_c$, where multiplication is performed coordinate-wise.
2. Output $\mathsf{pk} = \big(e, ([\![c^{(t)}]\!]_c)_{t \in [\ell]}\big)$.

Server Output: Each server $b \in \{0,1\}$ performs the following:

1. Let $\langle c_j \rangle_b$ and r_j denote the values received by client $j \in [m]$ in Round 1.
2. Take $\langle c \rangle_b = \sum_{j \in [m]} \langle c_j \rangle_b \in \mathbb{Z}_q$.
3. Take $r = \sum_{j \in [m]} r_j$ and compute a PRF key with input $\{0,1\}^\lambda \times \mathbb{G}$ and output $\{0,1\}^\ell$ using randomness r: i.e., $\phi = \mathsf{PRFGen}(1^\lambda; r)$.
4. Output $\mathsf{ek}_b = (m, \langle c \rangle_b, \phi)$.

Black-Box Client-Server DEHE: algorithms $\mathsf{Enc}, \mathsf{Eval}$

See Appendix A (Figure 9) for a full description of the BGI [BGI16] DEHE construction.

$\mathsf{Enc}_{\mathbb{G},g}(\mathsf{pk}, w)$: Same as BGI. That is,

1. Parse $\mathsf{pk} = \big(e, ([\![c^{(t)}]\!]_c)_{t \in [\ell]}\big)$.
2. Compute the following ElGamal ciphertexts:
   (a) Of $w \in \mathbb{Z}$: let $[\![w]\!]_c \leftarrow \mathsf{Enc}_{\mathsf{ElGamal}}(e, w)$.
   (b) Of $c^{(t)}w \in \mathbb{Z}$: for each $t \in [\ell]$, parse $[\![c^{(t)}]\!]_c = (h_1^{(t)}, h_2^{(t)})$, sample a fresh encryption of 0 $(h_1', h_2') \leftarrow \mathsf{Enc}_{\mathsf{ElGamal}}(e, 0)$, and let $[\![c^{(t)}w]\!]_c = ((h_1^{(t)})^w \cdot h_1', (h_2^{(t)})^w \cdot h_2')$.
3. Output $([\![w]\!]_c, \{[\![c^{(t)}w]\!]_c\}_{t \in [\ell]})$.

$\mathsf{Eval}_{\mathbb{G},g}(b, \mathsf{ek}, \mathsf{ct}, P, \delta)$:

1. Parse $\mathsf{ek} = (m, \langle c \rangle, \phi)$. Parse P as a sequence of RMS program instructions.
2. Execute as specified in the BGI construction (Figure 9), except for the following modification:

Security. We construct the desired simulator algorithm below. Essentially, Sim simulates honest parties faithfully, except that all encryptions (of $w, c_i^{(t)}$) under the joint public key are replaced by encryptions of 0.

$\mathsf{Sim}(1^\lambda, \mathsf{Corrupt})$:

1. Denote $S_b = \{S_0, S_1\} \cap \mathsf{Corrupt}$ (i.e., either a corrupt server or empty).
2. Client Round 1: Perform the following on behalf of each uncorrupted client $i \in [m]$.
   (a) Sample a fresh ElGamal key pair $c_i \leftarrow \mathbb{Z}_q$, $e = g^{c_i}$. Send c_i, e_i to all clients.
   (b) Sample a random "secret share" $v_i \leftarrow \mathbb{Z}_q$ and a random string $r_i \leftarrow \{0,1\}^\lambda$, and send both v_i, r_i to server S_b.
3. Client Round 2: Perform the following on behalf of each uncorrupted client $i \in [m]$.
   (a) Let $e = \prod_{j \in [m]} e_j$, where e_j is the value sent by client j in the previous round.
   (b) Sample ℓ encryptions of 0 (instead of the bits of his key c_i): i.e., for $t \in [\ell]$, let $[\![c_i^{(t)}]\!]_c \leftarrow \mathsf{Enc}_{\mathsf{ElGamal}}(e, 0)$.
   (c) Send $([\![c_i^{(t)}]\!]_c)_{t \in [\ell]}$ to all clients.
4. Client Output:
   (a) For each $t \in [\ell]$, let $[\![c^{(t)}]\!]_c = \prod_{j \in [m]} [\![c_i^{(t)}]\!]_c$ be the coordinate-wise product in $\mathbb{G}^2$.
   (b) Output $\mathsf{pk} = \big(e, ([\![c^{(t)}]\!]_c)_{t \in [\ell]}\big)$.
5. Simulation of $\mathsf{Enc}(w)$:
   (a) Sample an encryption of 0 (instead of w): i.e., let $[\![w]\!]_c \leftarrow \mathsf{Enc}_{\mathsf{ElGamal}}(e, 0)$.
   (b) Sample ℓ encryptions of 0 (instead of homomorphically generating an encryption of $c^{(t)}w$): i.e., for $t \in [\ell]$, let $[\![c^{(t)}w]\!]_c \leftarrow \mathsf{Enc}_{\mathsf{ElGamal}}(e, 0)$.
   (c) Output $\big([\![w]\!]_c, ([\![c^{(t)}w]\!]_c)_{t \in [\ell]}\big)$ as the simulated encryption of w.

Claim 3.6. *Assume circular security of ElGamal encryption. Then for any legal set of corruptions* $\mathsf{Corrupt} \subset \{C_1, \ldots, C_m\} \cup \{S_0, S_1\}$ *(i.e., with* $\{C_1, \ldots, C_m\}, \{S_0, S_1\} \not\subset \mathsf{Corrupt}$*) and any* $w \in \{0,1\}$*, it holds that* $\mathsf{view}(1^\lambda, \mathsf{Corrupt}, w) \overset{c}{\cong} \mathsf{Sim}(1^\lambda, \mathsf{Corrupt})$.

Proof. Indistinguishability of the simulated view for two inputs $w, w' \in \{0,1\}$ follows by three hybrids.

In the first hybrid, (if there exists a corrupted server S_b), we replace all additive secret shares given to S_b by truly random values in $\mathbb{Z}_q$. By perfect security of the secret sharing, these distributions are identical.

In the second hybrid, we replace the vectors of encryptions $([\![c_i^{(t)}]\!]_c)_{t \in [\ell]}$ with encryptions of 0 under key c, making use of the circular security property of ElGamal. Note that although these ciphertexts are under the *joint* key $c = \sum_i c_i$, by the key homomorphism of ElGamal within the reduction we can convert ciphertexts under key c_i to ones under key c while preserving the plaintext.

In the third hybrid, now that all side information on the secret keys c_i is removed, we may appeal to standard semantic security of ElGamal to remove all remaining encryptions under the joint key c with encryptions of 0. As above, this can be done as long as at least one client C_i is uncorrupted, as there is still an unknown key contribution c_i. □

□

Note that a client-server DEHE directly yields a construction of client-server HSS: In the HSS protocol Π_{HSS}, the clients and servers first run Π_{DEHE} of the DEHE to generate a global pk and evaluation keys $\mathsf{ek}_0, \mathsf{ek}_1$ to the respective servers, and then the final secret shares share_b of the clients' inputs will consist of $\mathsf{pk}, \mathsf{ek}_b$, and encryptions $\hat{w}_1 \dots, \hat{w}_m$ of the clients' inputs, where each $\hat{w}_i$ is generated by client C_i as $\hat{w}_i \leftarrow \mathsf{Enc}(\mathsf{pk}, w_i)$ and sent to both servers.

Because of this, we immediately obtain the following corollary.

Corollary 3.7 (Black-box client-server HSS)**.** *Assume ElGamal is circular secure. There exists a client-server HSS protocol for branching programs that makes a black-box access to any DDH-hard group.*

3.3 Black-Box Succinct Secure Computation

Given a black-box m-client 2-server multi-evaluation HSS ($\Pi_{\mathsf{HSS}}, \mathsf{Eval}_{\mathsf{HSS}}$) as above, and an arbitrary general 2PC protocol Π_{MPC}, we obtain succinct secure m-client 2-server computation for branching programs based on DDH which makes only black-box use of the DDH group. Namely, to securely evaluate a program P: (1) the clients and servers interact via Π_{HSS} to share the clients' inputs, (2) the servers homomorphically evaluate λ copies of the desired program P on the resulting shares, and then (3) run the generic protocol Π_{MPC} to securely evaluate the most common combined output.

Note that the procedure for combining evaluated shares and taking the majority (in Step 3) does not require any $\mathbb{G}$ group operations (only operations over the output space $\mathbb{Z}_\beta$), so that general secure computation of this function is still black-box in the DDH group $\mathbb{G}$.

The formal construction is given below. For simplicity, we treat the case where clients' inputs w_i are each a single bit.

Construction 3.8 (**Black-Box Secure Computation for Branching Progs Π_{BB}**)**.** Parties: Clients $C_1, \dots, C_m$, servers S_0, S_1.
Global inputs: DDH group $\mathbb{G}, g, q$, program P, parameter δ.
Inputs: Clients $C_1, \dots, C_m$ hold secret inputs $w_1, \dots, w_m \in \{0, 1\}$.
Output: Each client outputs $P(w_1, \dots, w_m)$.
Tools: m-client 2-server Multi-Eval HSS ($\Pi_{\mathsf{HSS}}, \mathsf{Eval}_{\mathsf{HSS}}$), and generic MPC protocol Π_{MPC}.

1. **Share inputs:** The clients and servers execute the client-server HSS protocol Π_{HSS}, where each client C_i and server S_b emulates the corresponding role in Π_{HSS}. As a result, each server S_b learns a respective HSS share value, share_b.

2. **Homomorphically Evaluate:** Each server performs λ multi-evaluation homomorphic evaluations on their respective share: i.e., for $j = 1, \dots, \lambda$, let $y_b^j \leftarrow \mathsf{Eval}_{\mathbb{G},g}(b, \mathsf{share}_b, P, \delta)$ using the HSS scheme. Recall each $y_b^j \in \mathbb{Z}_\beta$ for some $\beta \in \mathbb{Z}$ as specified in P.

3. **Reconstruct:** The servers participate in generic secure computation protocol Π_{MPC} on respective inputs $(y_0^j)_{j\in[\lambda]}, (y_1^j)_{j\in[\lambda]}$ to evaluate function f defined by $f\left((y_0^j)_{j\in[\lambda]}, (y_0^j)_{j\in[\lambda]}\right) = \mathsf{majority}_{j\in[\lambda]}(y_0^j - y_1^j) \in \mathbb{Z}_\beta$. The output of this computation is sent to and output by all clients.

Theorem 3.9 (Black-box succinct secure computation for branching programs)**.** *There exists a constant-round succinct m-client 2-server protocol* Π_{BB} *for branching programs that makes only black-box access to any DDH-hard group.*

Proof. Let $\mathcal{A}$ be a PPT adversary in the client-server secure computation protocol, who corrupts a subset of entities $\mathsf{Corrupt} \subset \{C_1, \ldots, C_m\} \cup \{S_0, S_1\}$, such that at least one client and one server is uncorrupt. Consider the following simulator $\mathsf{Sim}_{\mathsf{BB}}(1^\lambda, \{w_i\}_{C_i\in\mathsf{Corrupt}})$, which receives from the ideal functionality the correct final output $y = P(w_1, \ldots, w_m)$. First, $\mathsf{Sim}_{\mathsf{BB}}$ executes the HSS simulator,

$$\mathsf{view}^1_{\mathcal{A}} \leftarrow \mathsf{Sim}_{\mathsf{HSS}}(1^\lambda, \mathsf{Corrupt}, \{w_i\}_{C_i\in\mathsf{Corrupt}}, \{1\}_{C_i\notin\mathsf{Corrupt}}).$$

If $\mathsf{Corrupt} \cap \{S_0, S_1\} = \emptyset$, the simulator $\mathsf{Sim}_{\mathsf{BB}}$ outputs the value of $\mathsf{view}^1_{\mathcal{A}}$, together with the final output y. Otherwise, if $S_b \in \mathsf{Corrupt}$, then $\mathsf{Sim}_{\mathsf{BB}}$ locates from within $\mathsf{view}^1_{\mathcal{A}}$ the value share_b, corresponding to S_b's (simulated) HSS share (which is used as S_b's input to the final MPC execution), executes the MPC simulator as $\mathsf{view}^2_{\mathcal{A}} \leftarrow \mathsf{Sim}_{\mathsf{MPC}}(1^\lambda, \{S_b\}, \mathsf{share}_b)$, and outputs $(\mathsf{view}^1_{\mathcal{A}}, \mathsf{view}^2_{\mathcal{A}}, y)$.

By a straightforward hybrid argument, indistinguishability of the real and $\mathsf{Sim}_{\mathsf{BB}}$-simulated views follows by the security of the underlying HSS and MPC primitives.

□

Remark 3.10 (1/poly security tradeoff)**.** The round complexity of Π_{BB} is given by the round complexity HSS sharing protocol Π_{HSS} plus that of the generic MPC to evaluate the reconstruction-majority. If one is willing to accept 1/poly security, the MPC reconstruction phase can be replaced by a direct exchange of the output shares computed in the homomorphic evaluation. The corresponding simulator will follow the same simulation strategy, but will fail with inverse-polynomial probability, in the event that a homomorphic evaluation error occurs. The resulting protocol will have $\mathsf{rounds}(\Pi_{\mathsf{HSS}}) + 1$ rounds.

From here on, all of our protocols make a black-box access to the group except for protocols that involve $k \geq 3$ servers (in client server model) or parties (in the MPC model).

4 DDH-Based 2-Round Protocols over PKI

In this section we present a 2-round secure computation protocol in the PKI setup model for a constant number of parties and arbitrary polynomial-size circuits, based on DDH. Our starting point will be the general secure client-server protocol structure given in Theorem 3.9.

As discussed in the Introduction, our final 2-round solution removes the extra rounds of interaction by means of three main technical steps, which we present in the following three sections: (1) Constructing a Client-Server HSS whose secret sharing protocol Π can be executed in a *single* round of interaction in the PKI model; (2) Amplifying the resulting 2-round client-server protocol (Remark 3.10) from 1/poly to full security using techniques in leakage resilience; and (3) Compiling from 2 to any constant number k of servers by iteratively emulating a server's computation securely by 2 separate servers.

4.1 Succinct 2-Server Protocol with $1/\mathsf{poly}$ Security

We begin by constructing m-client 2-server HSS whose secret sharing protocol Π takes place via a *single message* from each client within the PKI model.

Our construction takes a similar approach to the black-box client-server HSS of the previous section, where each client owns an independent ElGamal key pair (c_i, e_i). However, the approach does not quite work as is. The primary challenge is in agreeing on common encryptions of the *cross*-products $c_j^{(t)} w_i$ for different clients C_i, C_j. Recall that HSS evaluation requires not only that each party holds an encryption of the same value, but in fact the exact *same* ciphertext.

This remains a problem even if we consider the setting with a public-key infrastructure (PKI). Namely, even given all clients' public keys, it is not clear how in a single message of communication all clients can agree on the same ciphertext of $c_i^{(t)} w_j$ under the joint key $\prod_i e_i$ when $c_i^{(t)}$ and w_j are known by two different clients, and $c_i^{(t)}$ and w_j themselves must remain hidden.

This goal *can* be achieved, however, for the i, j "pairwise" combination of public keys $e_i e_j$, by including an encryption of $c_i^{(t)}$ under key e_i as part of an expanded public key of client C_i. (Note that the value of $c_i^{(t)}$ depends only on C_i's keys themselves and not on inputs or number of parties, hence this is a valid contribution to the PKI setup.) Namely, given an encryption $[\![c_i^{(t)}]\!]_{c_i}$ of $c_i^{(t)}$ (using notation from [BGI16], as per Figure 2), client C_j can use the homomorphic properties of ElGamal to first shift this to an encryption under e_i of the product $c_i^{(t)} w_j$, and then shift this ciphertext to an encryption of the same value under key $e_i e_j$ by coordinate-wise multiplying in an encryption of 0 under key e_j. (Note that the second step is necessary in order to hide w_j from client C_i.)

We demonstrate that generating these pairwise $c_i^{(t)} w_j$ ciphertexts under the respective pairwise keys is enough to support full homomorphic evaluation capability. The new invariant maintained throughout homomorphic evaluation is that for each memory variable $\hat{y}$, the correct value y of this variable is held as an additive secret sharing $\langle y \rangle$, and as a *collection of* m additive secret sharings $\langle c_i y \rangle$, one for the key c_i of each client $i \in [m]$. Whenever we wish to perform an RMS multiplication using a ciphertext $[\![c_i^{(t)} w_j]\!]_{c_i + c_j}$, we can combine the corresponding pair of secret shares $\langle (c_i + c_j) y \rangle = \langle c_i y \rangle + \langle c_j y \rangle$, and then proceed as usual as if the secret key were the sum $c_i + c_j$.

As one additional change (which will be useful in future sections), we replace the bit decomposition $(c^{(t)})_{t \in [\ell]}$ of a key c with a more general, possibly randomized, representation $(\hat{c}^{(t)})_{t \in [\ell']} \leftarrow \mathsf{Decomp}(c)$. The only requirements for correctness are: (1) each value $\hat{c}^{(t)}$ has small magnitude; and (2) there exists a $\mathbb{Z}_q$-*linear* reconstruction procedure Recomp for which $c = \mathsf{Recomp}((\hat{c}^{(t)})_{t \in [\ell']})$.[5]

The formal descriptions of $(\Pi_{\mathsf{1r}}, \mathsf{Eval}_{\mathsf{1r}})$ are given in Figures 3 and 4.

Lemma 4.1 (One-Round Client-Server HSS). *Assume hardness of DDH. Then for any polynomial* $m = m(\lambda)$*, there exists an* m*-client 2-server HSS* $(\Pi_{\mathsf{1r}}, \mathsf{Eval}_{\mathsf{1r}})$ *for which* Π_{1r} *is a single round in the PKI model.*

Proof. Correctness. From the BGI HSS construction and analysis, it is known that the share conversion procedure properly coverts a ciphertext $[\![x]\!]_c$ under key c and additive shares $\langle y \rangle, \langle cy \rangle$ collectively to additive shares of the product xy, except with δ' error probability. We leverage

[5]Note that bit decomposition can be expressed in this form, where $\mathsf{Decomp}(c) := (c^{(t)})_{t \in [\ell]}$ and $\mathsf{Recomp}((c^{(t)})_{t \in [\ell]}) := \sum_{t=1}^{\ell} 2^{t-1} c^{(t)}$.

this exact property within our construction, except with different choices of the key c: either a single party's key c_i, or the pairwise joint key $(c_i + c_j)$ of two parties. Following along each of the homomorphic evaluation steps, and combining the linear homomorphism of the additive secret shares to obtain shares of $\langle (c_i+c_j)y \rangle$ given $\langle c_i y \rangle$, $\langle c_j y \rangle$, correctness aside from total error probability δ follows by a union bound.

Security. The proof of security follows that of the client-server DEHE from section 3.2. The only core difference is that secrets are published encrypted under pairwise *subsets* of the secret keys of clients. However, any secret value owned by a client C_i is encrypted under a combination of keys that includes his own key, c_i. Note that all such released ciphertexts are distributed as honestly sampled encryptions of the corresponding plaintext under the joint key, because of the step of re-randomization. Then, by the key homomorphism of ElGamal, this directly implies semantic security hiding of the corresponding ciphertexts. This extends as well to the case where the secret plaintexts are key bits, assuming ElGamal circular security (again, since in a reduction we can translate challenge ciphertexts $(\llbracket c_i^{(t)} \rrbracket_{c_i})_{t \in [\ell]}$ to $(\llbracket c_i^{(t)} \rrbracket_{c_i + c_j})_{t \in [\ell]}$ for any $c_j \in \mathbb{Z}_q$). $\square$

Plugging in the client-server HSS $(\Pi_{\mathsf{1r}}, \mathsf{Eval}_{\mathsf{1r}})$ to the framework of Theorem 3.9, together with the round-savings-for-1/poly tradeoff described in Remark 3.10, we directly obtain the following proposition.

Proposition 4.2 (Succinct 2-server protocol with 1/poly security for branching programs)**.** *Assuming PKI setup and DDH, for any polynomial $p(\cdot)$ and $m = m(\lambda)$ there is a (succinct) 2-round m-client 2-server client-server protocol for branching programs with $1/p(\lambda)$ security.*

4.2 Amplifying Security via Leakage Resilience

The 1/poly security loss in the protocol of Section 4.1 is due to the noticeable probability of (input-dependent) error in the homomorphic evaluation of the client-server HSS, revealed when evaluated output shares are directly exchanged. We now develop techniques for addressing this information leakage *without* additional communication rounds.

Simulatable Las Vegas HSS. Toward this goal, we first consider and realize two beneficial properties of a client-server HSS:

– *Las Vegas correctness.* In such an HSS scheme, servers can output a special symbol $\perp$ if they identify a possible error situation in the homomorphic evaluation. Las Vegas correctness guarantees that if both servers output a non-$\perp$ value then correct reconstruction will hold.

– *Simulatability of errors.* Unfortunately, it will be the case in constructions that servers do not always agree on whether an error is possible to occur (otherwise error could be removed completely by having each server recompute in such situation), and learning whether the other server reaches $\perp$ may reveal secret information. To address this, we consider a further "simulatability" property which formally characterizes what information is leaked through this process.

We construct simulatable Las Vegas HSS where the information leakage depends *locally* on values of a small number of memory values within the computation of the RMS program and/or symbols $\hat{c}^{(t)}$ of the secret key representation.

In the following two subsections, we present our construction of a simulatable Las Vegas HSS whose secret-sharing protocol is a single round given PKI, and then use this construction as a tool together with leakage-resilient techniques to obtain a (fully) secure 2-round 2-party computation protocol in the PKI model.

4.2.1 Defining and Obtaining Simulatable Las Vegas HSS

We define a "simulatable" variant of client-server Las Vegas HSS (LV-HSS), where each server has a secondary output in Eval that represents its knowledge about the other server's primary output. The secondary output can either be $\top$, indicating that it is certain that the other server does not output $\perp$, or a predicate Pred (represented by a circuit) that specifies a function of the clients' inputs $\vec{w}$ and randomness $\vec{r}$ such that the other party outputs $\perp$ if and only if $\mathsf{Pred}(\vec{w}, \vec{r}) = 1$. We require that the secondary output is $\top$ except with at most δ probability. Note that Pred may depend on the program P being homomorphically evaluated.

Definition 4.3 (Simulatable Client-Server Las Vegas HSS)**.** A (m-client, 2-server) *Simulatable Client-Server Las Vegas HSS* scheme for class of programs $\mathcal{P}$ consists of a distributed protocol Π and PPT algorithm Eval, with syntax:

- Π specifies an interactive protocol between m clients $C_1, \ldots, C_m$ and two servers S_0, S_1, where each client C_i begins with input w_i, and in the end of executing Π the servers S_0, S_1 output homomorphic secret shares $\mathsf{share}_0, \mathsf{share}_1$, respectively, of the joint input $(w_1, \ldots, w_m)$.
- Eval has a second output z such that z is either the symbol $\top$ or a predicate $\mathsf{Pred} : \{0,1\}^n \to \{0,1\}$ represented by a boolean circuit.

  We denote by $(\mathsf{share}_0, \mathsf{share}_1) \leftarrow \Pi(\vec{w}; \vec{r}, R_0, R_1)$ where $\vec{w} = (w_1, \ldots, w_m)$ and $\vec{r} = (r_1, \ldots, r_m)$ the execution of Π in which each client $i \in [m]$ uses input w_i and randomness r_i, each server $b \in \{0,1\}$ uses randomness R_b, and the output to each server S_b is share_b.

The pair (Π, Eval) should satisfy the correctness of Definition 2.4 (with respect to the first output of Eval), and the following additional requirements:

- **Security:** There exists a PPT simulator Sim such that for any corrupted set $\mathsf{Corrupt} \subset \{C_1, \ldots, C_m\} \cup \{S_0, S_1\}$ of clients and servers for which at least one server and one client are uncorrupted, for every polynomial p, and sequence of input vectors $\vec{w}^\lambda = (w_1^\lambda, \ldots, w_m^\lambda) \in (\{0,1\}^{p(\lambda)})^m$, it holds that $\mathsf{view}(1^\lambda, \mathsf{Corrupt}, \vec{w}^\lambda) \overset{c}{\cong} \mathsf{Sim}(1^\lambda, \mathsf{Corrupt}, \{w_i\}_{C_i \in \mathsf{Corrupt}}, \{|w_i|\}_{C_i \notin \mathsf{Corrupt}})$.
- **Error simulation:** For every polynomial p there is a negligible ν such that for every $\lambda \in \mathbb{N}$, input $w \in \{0,1\}^n$, program $P \in \mathcal{P}$ with input length n, error bound $\delta > 0$ and integer $\beta \geq 2$, where $|P|, 1/\delta \leq p(\lambda)$, then for every $b \in \{0,1\}$,

  $$\begin{aligned}\Pr[&(\mathsf{share}_0, \mathsf{share}_1) \leftarrow \Pi(\vec{w}; \vec{r}, R_0, R_1);\\ &\qquad (y_b, z_b) \leftarrow \mathsf{Eval}(b, \mathsf{share}_b, P, \delta, \beta) : z_b \neq \top] \leq \delta + \nu(\lambda),\end{aligned}$$

  and for every circuit Pred and $c \in \{0,1\}$:

  $$\begin{aligned}\Pr[&(\mathsf{share}_0, \mathsf{share}_1) \leftarrow \Pi(\vec{w}; \vec{r}, R_0, R_1); (y_b, z_b) \leftarrow \mathsf{Eval}(b, \mathsf{share}_b, P, \delta, \beta),\ b = 0, 1 :\\ &\qquad (z_c = \mathsf{Pred}) \wedge (\chi(y_{1-c} = \perp) \neq \mathsf{Pred}(\vec{w}, \vec{r}))] \leq \nu(\lambda),\end{aligned}$$

  where $\chi(y_{1-c} = \perp)$ evaluates to 1 if $y_{1-c} = \perp$ and evaluates to 0 otherwise.

Algorithm 1 Simulatable $\mathsf{SLVDistribDLog}_{\mathbb{G},g}(b, h, \delta, M, \phi)$

```
1:  Let $\mathsf{DangerZone} := \{h, hg^{(-1)^b}, \ldots, hg^{(-1)^b M}\}$.
2:  Let $\mathsf{SimDangerZone} := \{hg^{-M+1}, \ldots, h, \ldots, hg^M\}$ and initialize $\mathsf{BadValues} \leftarrow \emptyset$.
3:  **if** $\exists h' \in \mathsf{SimDangerZone}$ with $\phi(h') = 0^{\lceil \log(2M/\delta) \rceil}$ **then** Let $\mathsf{BadValues}$ be the set of $z \in [M]$ for which $\{hg^{(-1)^b z}, hg^{(-1)^b z + (-1)^{b-1}}, \ldots, hg^{(-1)^b z + (-1)^{b-1} M}\}$ contains some $h'$ with $\phi(h') = 0^{\lceil \log(2M/\delta) \rceil}$). If $\mathsf{BadValues} = \emptyset$, set $\mathsf{BadValues} \leftarrow \top$.
4:  **end if**
5:  **if** $\exists h' \in \mathsf{DangerZone}$ with $\phi(h') = 0^{\lceil \log(2M/\delta) \rceil}$ **then** Let $i = \perp$.
6:  **else**
7:      Set $h' \leftarrow h$, $i \leftarrow 0$. Let $T = 2M\lambda/\delta$.
8:      **while** ($\phi(h') \neq 0^{\lceil \log(2M/\delta) \rceil}$ and $i < T$) **do**
9:          $h' \leftarrow h' \cdot g$, $i \leftarrow i + 1$.
10:     **end while**
11: **end if**
12: Return $(i, \mathsf{BadValues})$.
```

Constructing simulatable Las Vegas HSS. Our construction of simulatable (client-server) LV-HSS will be a variant of the 1-round Client-Server HSS construction, with a modified core share-conversion sub-routine DistributedDLog (called within ConvertShares), which enables each party to convert a multiplicative share of $g^z \in \mathbb{G}$ to an additive share of $z \in \mathbb{Z}_q$ (for small z).

Following [BGI16], the procedure DistributedDLog takes as input a share $h \in \mathbb{G}$ and outputs the distance on the cycle generated by $g \in \mathbb{G}$ between h and the first "distinguished" point $h' \in \mathbb{G}$ such that a pseudo-random function (PRF) outputs 0 on h'. Two invocations on inputs h and $h \cdot g^z$ for a small z result, with good probability (over the initial choice of PRF seed), in outputs i and $i - z$ for some $i \in \mathbb{Z}_q$. In such case, the DistributedDLog procedure converts a difference of small z in the cycle generated by g in $\mathbb{G}$ to the same difference over $\mathbb{Z}$.

For any $h, h \cdot g^z \in \mathbb{G}$, DistributedDLog yields an error in two cases:

1. When there exists a distinguished point h' *between* the two inputs h, hg^z: i.e., $h' = hg^i$ for some $i \in \{0, \ldots, z-1\}$.

2. When there does not exist a distinguished point within a fixed polynomial-size range after which the party will abort.

We construct a simulatable Las Vegas version of this sub-routine, SLVDistribDLog, described in Algorithm 1. This algorithm has three primary differences from the original procedure DistributedDLog.

1. For simplicity, the end-case abort threshold T is set large enough ($2M\lambda/\delta$) so that the probability of abort over the choice of distinguished points (via the PRF ϕ) is negligible. Recall the choice of T gives a tradeoff between error probability and required computation (in [BGI16], and in our complexity-optimized versions in later sections, the threshold is set to a lower value).

2. Given an input share $h \in \mathbb{G}$, maximum magnitude bound M, and "party id" $b \in \{0, 1\}$, the algorithm will now output $\perp$ if there is a distinguished point h' within M steps of h in the

direction dictated by b. Recall that this sub-routine will be called simultaneously by party P_0 (the "behind" party) holding share h and party P_1 (the "ahead" party) holding share $h \cdot g^z$. In the new procedure, P_0 will output $\perp$ if any of $h \cdot g, \ldots, h \cdot g^{M-1}$ is distinguished, and P_1 will output $\perp$ if any of $h \cdot g^{z-M+1}, \ldots, h \cdot g^{z-1}$ is distinguished. This will guarantee (no matter the value of $z \in [M]$) that if there is a distinguished point between the two parties' shares then both parties will output $\perp$.

This zone of values is denoted $\mathsf{DangerZone}$ in $\mathsf{SLVDistribDLog}$.

3. $\mathsf{SLVDistribDLog}$ now outputs two values: (1) a $\mathbb{Z}_q$-element (or $\perp$) as usual, corresponding to the output additive share, and (2) a subset $\mathsf{BadValues} \subset [M]$ of values z such that the other party $1-b$ will have a distinguished point h' within his $\mathsf{DangerZone}$ (and output $\perp$) if and only if he runs $\mathsf{SLVDistribDLog}$ with input $hg^{(-1)^b z}$ (i.e., our respective inputs $h, g^{(-1)^b z}$ are multiplicative shares of g^z for some $z \in \mathsf{BadValues}$).

   Basically, for each possible share of the other party, we can directly determine if it would result in $\perp$, and record the corresponding secret shared value $z \in [M]$ if it would. In the notation of $\mathsf{SLVDistribDLog}$, the window $\mathsf{SimDangerZone}$ is of size $2M$ and captures all possible shifted windows of size M which could be the $\mathsf{DangerZone}$ of the other party, depending on which of the M possible values of z is the current offset between shares.

In Figure 5 we present a construction of simulatable Las Vegas HSS, using $\mathsf{SLVDistribDLog}$ as a sub-routine. Roughly: At every share conversion step of homomorphic evaluation in $\mathsf{Eval}^{\mathsf{SLV}}$, with some probability there will exist a bad set of plaintext values $z \in [M]$ such that if the newly computed shared value is equal to z then the other party would output $\perp$. These sets of bad values are identified within $\mathsf{SLVDistribDLog}$ and are stored as $\mathsf{BadValues}$'s within Eval. A pair $(k, \mathsf{BadValues}_k) \in \mathbb{Z} \times 2^{[M]}$ is added to $\mathsf{LeakageInfo}$ if partial computation value $y_k = z \in \mathsf{BadValues}$ would lead to the other party outputting $\perp$. This corresponds to a share conversion for some $\langle y_k \rangle$. Similarly, a pair $((k, \gamma, t), \mathsf{BadValues}_{k,\gamma,t}) \in (\mathbb{Z} \times [m] \times [\ell]) \times 2^{[M]}$ is added to $\mathsf{LeakageInfo}$ if partial computation value $\hat{c}_\gamma^{(t)} y_k = z \in \mathsf{BadValues}_{k,\gamma,t}$ would lead to the other party outputting $\perp$. This corresponds to a share conversion for some $\langle \hat{c}_\gamma^{(t)} y_k \rangle$. Note that the values y_k are defined as a function of the program P and a given input $\vec{w}$. The choice of Pred incorporates the P dependency, and operates on input $\vec{w}$ as well as a subset of (at most λ values of) $\hat{c}^{(t)}$.

Proposition 4.4. *Assume hardness of DDH. Then for any polynomial $m = m(\lambda)$, the scheme $(\Pi_{\mathsf{SLV}}, \mathsf{Eval}_{\mathsf{SLV}})$ described above is an m-client 2-server simulatable Las Vegas HSS, where Π_{SLV} is a single round in the PKI model. Moreover, with overwhelming probability in λ over the randomness of Π, the predicate Pred depends on at most λ intermediate variables of the evaluation of the RMS program P and values $\hat{c}^{(t)}$.*

Proof. Correctness and security follow as in the previous constructions. We now consider the property of error simulation. We must demonstrate two items.

First, we bound the probability over the randomness of $(\mathsf{share}_0, \mathsf{share}_1) \leftarrow \Pi^{\mathsf{SLV}}$ that we have $\mathsf{SimOutput} \neq \top$. Recall that $\mathsf{SimOutput} \neq \top$ exactly when there exists at least one pair $(k, \mathsf{BadValues})$ in the set $\mathsf{LeakageInfo}$. In each execution of $\mathsf{SLVDistribDLog}$, such a pair is added when there exists a distinguished point within the $2M$-size window $\mathsf{SimDangerZone} \subset \mathbb{G}$. Since each group element $h \in \mathbb{G}$ is selected as a distinguished point via a pseudo-random function indistinguishably from being selected independently at random, the probability of this event is bounded by

$1-(1-\delta/(2M))^{2M} \leq 1-(1-(2M)\delta/(2M)) = \delta$. Note that although future share values depend on previously computed values, and thus on the choice of PRF, they depend only on the *output* values of the PRF, and no input is used more than once, thus pseudo-randomness is guaranteed to hold.

Second, we claim that if $\mathsf{SimOutput} \neq \top$, then the output predicate Pred correctly simulates the output share ($\perp$ or not) of the second party. Recall the second party P_{1-b} will output $\perp$ if and only if he receives $\perp$ as the output in any execution of $\mathsf{SLVDistribDLog}$. Given the share h_b of party P_b in an $\mathsf{SLVDistribDLog}$ execution, there are M possible values for the share h_{1-b} of P_{1-b} into the corresponding execution, and each such value corresponds to a window $\mathsf{DangerZone}_z$ ($z \in [M]$) for P_{1-b}. If there is no distinguished point within any of these M $\mathsf{DangerZone}_z$ windows, then it is guaranteed that P_{1-b} will not output $\perp$ within this $\mathsf{SLVDistribDLog}$ execution. Further, if there is a distinguished point within one or multiple $\mathsf{DangerZone}_z$'s, then the identification of whether P_{1-b} will output $\perp$ will be exactly determined by whether the true value converted in this step is equal to one of the bad choices of $z \in [M]$. This is directly captured by the sets $\mathsf{BadValues}$. Finally, note that Pred can indeed be computed given the inputs and randomness $(\vec{w}, \vec{r})$ of clients within the execution of Π^{SLV}.

The true memory value y_k of a partial computation can be computed as a function of the program P, share_b (as described) and inputs $\vec{w}$. The true value of $c_\gamma^{(t)}$ for some $\gamma \in [m]$ can be directly computed as a function of the randomness r_γ used by the *single* client C_γ. Now, the error parameter of Eval is set so that the probability of an error occurring in any share conversion procedure is δ, and error occurs in each conversion independently. This means that with overwhelming probability in λ, no more than λ conversions will result in error in total, meaning $\mathsf{LeakageInfo}$ will contain at most λ pairs. Altogether, this implies as desired that (with overwhelming probability) Pred will depend on at most λ intermediate memory values y_k and on the randomness r_γ of at most λ different clients. $\square$

Remark 4.5 (Asymmetric Las Vegas HSS)**.** In some of our later applications (see Section 5), it will be advantageous to have an *asymmetric* notion of Las Vegas HSS, where only one of the two parties might output $\perp$. In these applications, simulatability will not be required. We can achieve such notion via a simple tweak of our construction by simply removing the option of outputting $\perp$ for party P_1 within the sub-routine $\mathsf{SLVDistribDLog}$.

4.2.2 Secure 2-server Computation from Simulatable LV-HSS and Leakage Resilience

We now combine the simulatable LV-HSS of Proposition 4.4, which yields 2-server protocols with partial leakage, together with techniques for protecting computation against this leakage, to obtain a 2-round (m-client 2-server) secure computation protocol (in the PKI model) with *standard* security.

More concretely, the simulatable LV-HSS $(\Pi_{\mathsf{SLV}}, \mathsf{Eval}_{\mathsf{SLV}})$ guaranteed leakage (with high probability) of up to λ intermediate RMS computation memory values y_i and secret-key representation values $\hat{c}^{(t)}$.

To protect against leakage of intermediate computation values, we can replace homomorphic evaluation of the program P with evaluation of a new ("leakage-resilient") program that takes as input *secret shares* $w_i^{(1)}, \dots, w_i^{(k)}$ of clients' inputs w_i, and emulates a k-server secure computation of the program (whose $\mathsf{NextMsg}$ computation is in NC^1) that recombines secret shares and evaluates P, *while guaranteeing correctness and security against* λ *out of* k *server corruptions* (referred to as "λ-robustness"). Indeed, the λ leaked/erred intermediate computation values from HSS evaluation now correspond directly to revealing/losing the view of up to λ (virtual) servers in the emulated protocol. For simplicity, we use client-server protocols with no server-server communication, and so we can even emulate servers by *independent* HSS executions. Such protocols are known to exist for secure computation of low-degree polynomials [IK00]; in turn, this yields a solution for secure computation of general circuits P by instead generating a randomized encoding of the circuit P, computable in low degree [Yao86, AIK05].

To deal with the leakage on the values $\hat{c}^{(t)}$, we further refine the above approach. It will no longer be sufficient to take the $\hat{c}^{(t)}$ directly as the bits of the ElGamal secret key c (as in [BGI16]), since this leakage will compromise the security of the encryptions and thus the HSS. Instead, we take $(\hat{c}^{(t)})_{t\in[\ell']} \leftarrow \mathsf{Decomp}(c)$ defined by first additively secret sharing c over $\mathbb{Z}_q$ into $\lambda+1$ shares, and then taking the $\ell' := (\lambda+1)\ell$ bits of these separate values. Note that the $\hat{c}^{(t)}$ themselves are bits (in particular, have small magnitude) and reconstruction is linear over $\mathbb{Z}_q$ (first perform powers-of-2 bit reconstruction, then add the resulting values). But, further, any subset of λ values $\hat{c}^{(t)}$ are *statistically independent* of c.

Theorem 4.6 (Security amplification via virtual client-server protocols). *Let* $(\Pi_{\mathsf{SLV}}, \mathsf{Eval}_{\mathsf{SLV}})$ *be the one-round simulatable Las Vegas client-server HSS from Proposition 4.4, and let* $(\mathsf{Encode}, \mathsf{NextMsg}, \mathsf{Decode})$ *be a λ-robust client-server secure computation protocol with no server-server communication with* $\mathsf{NextMsg} \in \mathsf{NC}^1$ *(see Section 2.7). Then for any polynomial $m = m(\lambda)$, the protocol Π given in Construction 4.7 is a secure m-client 2-server protocol for general circuits that executes in* 2 rounds *in the PKI model.*

Construction 4.7 (Secure 2-round m-client 2-server protocol (with PKI)). Input: Each client begins with input w_i.
Tools:

- ("Virtual") 2λ-robust m-client k-server single-round secure computation protocol $(\mathsf{Encode}, \mathsf{NextMsg}, \mathsf{Decode})$, with *no* server-server interaction (i.e., server computation is a single execution of $\mathsf{NextMsg} \in \mathsf{NC}^1$).

- One-round simulatable LV-HSS $(\Pi_{\mathsf{SLV}}, \mathsf{Eval}_{\mathsf{SLV}})$ from Proposition 4.4.

Protocol:

0. PKI: The new PKI consists of k independent copies of the PKI distribution from the simulatable LV-HSS; denote each copy by $\mathsf{PKI}^{(j)}$.

1. Each client C_i encodes his input as $(\mathsf{msg}_i^{(1)}, \ldots, \mathsf{msg}_i^{(k)}) \leftarrow \mathsf{Encode}(i, w_i)$.

2. **Communication Round 1:** In k parallel executions (one for each virtual server in the underlying secure computation protocol), using fresh randomness, the clients each send the corresponding single message as dictated by the one-round sharing protocol Π_{SLV}, where in the j'th execution ($j \in [k]$), client C_i uses $\mathsf{PKI}^{(j)}$ and input $\mathsf{msg}_i^{(j)}$.

3. As a result of the previous step, each (real) HSS server S_b learns k shares $\mathsf{share}_b^{(1)}, \ldots, \mathsf{share}_b^{(k)}$, one for each *virtual* server in the secure computation protocol, where $\mathsf{share}_b^{(j)}$ is one share of all clients' messages to virtual server j.

4. Each server S_b performs k independent homomorphic evaluations: For each virtual server $j \in [k]$, let $(\mathsf{output}_b^{(j)}, z_b^{(j)}) = \mathsf{Eval}_{\mathbb{G},g}^{\mathsf{SLV}}(b, \mathsf{share}_b^{(j)}, \mathsf{NextMsg}, 1/2k\lambda)$, with allowable error probability $1/2k\lambda$. Let $\mathsf{output}_b = (\mathsf{output}_b^{(1)}, \ldots, \mathsf{output}_b^{(k)})$, i.e. S_b's secret share (with possible $\perp$ symbols) of the encoded output of the client-server protocol.

5. **Communication Round 2:** Each server $b \in \{0, 1\}$ sends his evaluated share, output_b, to all clients.

6. Each client outputs $\mathsf{Decode}(\mathsf{output}_0 + \mathsf{output}_1)$: i.e., recombining the HSS output shares (where $\perp + h$ is defined as $\perp$) and running the decoding procedure of the client-server protocol on the resulting output.

Proof Sketch. We briefly outline the simulator $\mathsf{Sim}_{2r}(1^\lambda, \{w_i\}_{C_i \in \mathsf{Corrupt}}, y)$, where $\mathsf{Corrupt} \subset \{C_1, \ldots, C_m\} \cup \{S_0, S_1\}$ is the set of corrupted clients/servers, and y is the output $P(w_1, \ldots, w_m)$ received by the ideal functionality.

Assume wlog that $S_b \in \mathsf{Corrupt}$. Sim_{2r} simulates the HSS shares sent to S_b in the first round on behalf of each honest client C_i, by generating an HSS sharing with respect to $\mathsf{PKI}^{(j)}$ of 0 for each virtual server $j \in [k]$. For $j \in [k]$, Sim_{2r} computes $(\mathsf{output}_b^{(j)}, z_b^{(j)}) = \mathsf{Eval}_{\mathbb{G},g}^{\mathsf{SLV}}(b, \mathsf{share}_b^{(j)}, \mathsf{NextMsg}, 1/2k\lambda)$ on S_b's shares. Let $\mathsf{Corrupt}_S^{\mathsf{Virt}} = \{j \in [k] : z_b^{(j)} = \mathsf{Pred}_b^{(j)} \neq \top\}$ be the virtual servers j for which $\mathsf{output}_{1-b}^{(j)}$ might be $\perp$ (thus leaking information). By Proposition 4.4, with overwhelming probability $|\mathsf{Corrupt}^{\mathsf{Virt}}| \leq \lambda$ (by correctness and independence of executions) and each $\mathsf{Pred}_b^{(j)}$ depends on the input and at most λ values of $\hat{c}^{(t)}$ for the key c within the corresponding j'th HSS execution.

Sim_{2r} then runs the simulator for the underlying (virtual) m-client k-server protocol, for corrupted clients $\mathsf{Corrupt}_C^{\mathsf{Virt}} = \mathsf{Corrupt} \cap \{C_1, \ldots, C_m\}$ and corrupted (virtual) servers $\mathsf{Corrupt}_S^{\mathsf{Virt}}$, for corrupted inputs $\{w_i\}_{C_i \in \mathsf{Corrupt}}$. The resulting simulated $\mathsf{view}^{\mathsf{Virt}}$ contains, in particular, the messages $\{\mathsf{msg}_i^{(j)}\}_{C_i \notin \mathsf{Corrupt}}$ received by each corrupt virtual server $j \in \mathsf{Corrupt}_S^{\mathsf{Virt}}$ from honest clients C_i, and all (pre-Decode) values $\mathsf{output}^{(1)}, \ldots, \mathsf{output}^{(k)}$.

For $j \in [k]$, Sim_{2r} simulates the output share $\mathsf{output}_{1-b}^{(j)}$ as follows. Sample λ random bits to serve as the bits $(\hat{c}^{(t)})_{t \in [\lambda]}$ of the jth key that $\mathsf{Pred}_b^{(j)}$ depends on (if $z_b^{(j)} = \mathsf{Pred}_b^{(j)} \neq \top$). If $j \notin \mathsf{Corrupt}_S^{\mathsf{Virt}}$, or if $\mathsf{Pred}_b^{(j)}(\mathsf{msg}^{(j)}, (\hat{c}^{(t)})_{t \in [\lambda]}) = 0$, then $\mathsf{output}_{1-b}^{(j)} = \mathsf{output}^{(j)} - \mathsf{output}_b^{(j)}$. Otherwise, $\mathsf{output}_{1-b}^{(j)} = \perp$. □

Theorem 4.9 is an application of the above, obtained by using the virtual client-server protocol of [IK00] for evaluating low-degree polynomials.

Theorem 4.8 (MPC for low-degree polynomials [IK00]). *For any $t, m, d \in \mathbb{N}$ there is a 2-round, m-client, k-server, perfectly t-robust protocol with* no *server-server interaction, for the class of degree-d polynomials over $\mathbb{F}_2$, where $k = O(dt)$. When evaluating a vector of ℓ polynomials on n inputs, the computation of each server can be implemented by a circuit of size $\mathsf{poly}(n, \ell, k)$ and depth $O(\log(n + \ell + k))$.*

Theorem 4.9 (Succinct 2-server protocol for low-degree polynomials). *Assuming PKI setup and DDH, there is a succinct 2-round 2-server client-server protocol for evaluating degree-d polynomials, for any constant d.*

Our final result follows from generic transformations using low-degree randomized encodings [AIK05].

Corollary 4.10 (2-server protocol for circuits). *Assuming PKI setup and DDH, there is a (non-succinct) 2-round 2-server client-server protocol for circuits.*

"Constant rate" via algebraic geometric codes. We note that we can modify the 2-round protocol for constant-degree polynomials by replacing Theorem 4.8 with a virtual client-server protocol based on algebraic geometric codes, as in Theorem 4.11 below.

Theorem 4.11 (Constant-rate MPC for finite functions). *[CC06, IKOS09] Let f be a finite function mapping m client inputs to m client outputs. For any positive integer t there is a 2-round, m-client, k-server, perfectly t-robust protocol for evaluating t instances of f, where $k = O(t)$. Each client in this protocol sends $O(1)$ bits to each server and each server sends $O(1)$ bits to each client (independently of t).*

In doing so, we obtain the following "constant rate" oblivious transfer protocol.

Theorem 4.12 (Constant-rate bit-OT via AG codes). *Assuming PKI setup and DDH, there is a 2-message protocol realizing n instances of bit-OT with $O(n) + \mathsf{poly}(\lambda)$ bits of communication.*

Succinct 2-round MPC for NC^1. Note that while the above solutions yield 2 rounds of communication, the amount of information communicated is greater than the program size. In what follows, we describe a more complex solution achieving *succinct* 2-round secure computation for the class of NC^1 programs.

The following theorem is used for succinct protocols for NC^1. This is the only case in which we need to rely on an MPC protocol that involves server-to-server communication.

Theorem 4.13 (MPC for NC^1). *Suppose there is a PRF in NC^1. Then for any positive integers t, m there is a constant-rounds, m-client, k-server, computationally t-robust protocol for formulas, where $k = O(t)$. When evaluating a boolean formula of size S on n input bits, each client sends to each server a message of length $O(n \log k + \lambda)$ in the first round, and each server sends a single bit in the final round. The servers' next message function* $\mathsf{NextMsg}$ *for each round is computable by a formula of size $\mathsf{poly}(S, k, m)$.*

Proof. (sketch) We rely on the fact that every formula f of size S admits a degree-3 randomized encoding $\hat{f}$ of size $\mathsf{poly}(S)$, where decoding can also be done by a formula of size $\mathsf{poly}(S)$. An encoding of this type can be realized using an information-theoretic variant of Yao's garbled circuit construction (cf. [IK02]). The protocol proceeds as follows. In the first round, each client secret-shares its input between the servers, and sends a PRF key to each server. Each server i expands the PRF keys received from each client into polynomial-length random strings, and uses their XOR as its private randomness r_i. In the next two rounds, the servers run a perfectly t-robust k-party MPC protocol for degree-3 polynomials to compute a randomized encoding (of length $\mathsf{poly}(S)$) of the output of f. The randomness of the encoding is taken from the random inputs r_i. Finally, the servers locally decode the output y of f from the output of the encoding $\hat{f}$, and send y to the clients. Robustness is inherited from the robustness of the protocol for degree-3 polynomials. □

We do not know whether Theorem 4.13 can be generalized to the case of branching programs. While we do have polynomial-size degree-3 randomized encodings for branching programs [IK00], decoding requires the computation of a determinant or similar functions. Computations of this type are conjectured to be unrealizable by polynomial-size deterministic branching programs.

Towards obtaining *succinct* 2-round protocols for NC^1, we need to rely on the protocol of Theorem 4.13 as the virtual MPC protocol. The server-to-server communication in this protocol (unlike all previous protocols) requires the servers to homomorphically perform *joint* computation on the views of different virtual servers. This means that the initial messages sent to these servers should be homomorphically secret-shared together. Since all these views are shared using the same ElGamal secret key c, leakage caused by share conversion may depend not only on information that is local to one server but also on bits of this global c.

As before, this dependence is easy to eliminate via further randomization of the bit decomposition of c, ensuring that any small subset of these bits is statistically independent of c. This suffices for jointly simulating the bits leaked by failure of the share conversion.

Using this approach we get the following theorem.

Theorem 4.14 (Succinct protocols for NC^1)**.** *Assuming PKI setup and DDH, there is a succinct 2-round 2-server client-server protocol (alternatively, a 2-message 2PC protocol with one-sided output) for boolean formulas.*

4.3 From 2 to k Servers

As the final step, we compile the 2-round m-client 2-server protocol into a 2-round m-client k*-server* protocol, for any constant $k \in O(1)$. This is achieved by iteratively emulating the role of one server by two servers via the original 2-server protocol. A similar notion of party emulation has appeared within many contexts in the literature (e.g., [Bra84, HM00]). In each step of this process, the next-message-function computed by the emulated server is realized by using a 2-round client-server protocol involving the m clients and the 2 emulating servers. This increases the number of servers by 1, while still maintaining security as long as only a strict subset of the servers are corrupted. The communication and computation complexity of the protocol increase by a factor of $\mathsf{poly}(\lambda)$ in each such step. Repeating $k-1$ times, we get the following.

Theorem 4.15 (2-round k-server client-server protocol)**.** *Assume PKI setup and DDH. Then for any constant $k \geq 2$ there is a 2-round k-server client-server protocol (alternatively, a 2-round k-party MPC protocol) for circuits.*

5 Optimizing Communication

In the previous section, we eliminated the inverse polynomial error and leakage of HSS by using secret-sharing of the inputs and applying virtual client-server MPC protocols to compute on these shares. In this section we describe a simpler alternative approach that has better asymptotic and concrete communication complexity (and better computational complexity as well) at the cost of requiring an additional round of interaction. In contrast to the previous approach, the current approach applies only to the case of 2PC and does not apply to the more general case of client-server MPC.

The high level idea is as follows. Denote the two parties by P_0, P_1 and assume that the functionality f delivers an output only to P_1. We rely on an asymmetric Las-Vegas HSS (see Definition 2.4) where the output of Eval is guaranteed to be correct (i.e., the two output shares add up to the correct output) unless P_1 outputs $\perp$, where the latter occurs with at most δ probability. The idea is to have P_1 use $\binom{m}{m-k}$-bit-oblivious-transfer (denoted by $\binom{m}{-k}$-OT) in order to block itself from the k output shares of P_0 that correspond to the positions in which it outputs $\perp$. Note that the $m-k$ selected output shares can be simulated given the correct output and the view of P_1, and thus they do not leak any additional information about the input. To make up for the k lost output bits, we use an erasure code to encode the output. Since we can make the number of erasures small, we only need to introduce a small amount of redundancy to the output.

Punctured OT. A key observation is that by setting the error parameter δ to be sufficiently small, we can ensure that the $\binom{m}{-k}$-OT parameters are such that k is much smaller than m. We refer to OT in this parameter regime as *punctured OT* and show how to implement it very efficiently by using a *puncturable PRF*.

A puncturable PRF [SW14] is a standard PRF family F_K equipped with a puncturing algorithm $\mathsf{Puncture}$ that given a set of points $X = \{x_, \dots, x_k\} \subseteq \{0,1\}^d$ produces an evaluation key K_X that allows an evaluation of the PRF on all inputs *except* those in X. Moreover, the PRF values on the inputs in X should be indistinguishable from random given K_X. See Section 2.3 for a formal definition. As was shown in [BW13, BGI14, KPTZ13], the GGM construction [GGM86] of PRFs from a length-doubling PRG can be used to obtain a puncturable PRF for $X = \{x_1, \dots, x_k\} \subseteq \{0,1\}^d$ with key size $|K_X| = O(\lambda k d)$. The evaluation of F at all points given K or at all non-punctured point given K_X requires $O(2^d)$ invocations of a PRG $G : \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$. The circuit size required for generating K_X given a λ-bit K and X is $kd \cdot poly(\lambda)$.

A protocol for $\binom{m}{-k}$-OT can be implemented using a puncturable PRF and any general-purpose 2PC protocol (e.g., Yao's protocol [Yao86, LP09]) in the following natural way.

- Sender's input: $s \in \{0,1\}^m$, where every $i \in [m]$ is represented by a d-bit string.
- Receiver's input: $X \subset [m]$ where $|X| = k$.
- Given primitives: a puncturable PRF $(F_K, \mathsf{Puncture})$, an ideal 2PC oracle Π.

1. Invoke Π on the randomized functionality that, on Receiver input X, delivers a random PRF key K to Sender and constrained PRF key K_X to Receiver.
2. Sender computes and sends $s' \in \{0,1\}^m$ where $s'_i = s_i \oplus F_K(i)$.
3. Receiver outputs $(i, s' \oplus F_{K_X}(i))$ for $i \in [m] \setminus X$.

Analysis. Correctness is straightforward. Security follows from the fact that the values of F_K on all inputs $i \in [m] \setminus X$ are pseudorandom given K_X. Thus, a simulator can simulate the receiver's view given the receiver's output by just running the protocol with an arbitrary s that is consistent with the output. Plugging in Yao's protocol[6] for implementing Π, we get the following theorem.

Theorem 5.1 (Punctured OT via puncturable PRF). *Suppose a $\binom{2}{1}$-OT protocol exists. Then there is a protocol for $\binom{m}{-k}$-OT with $m + k \cdot \log m \cdot \mathsf{poly}(\lambda)$ bits of communication, where the computational complexity consists of $O(m)$ invocations of a length-doubling PRG $G : \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$ and $\mathsf{poly}(\lambda)$ additional computation.*

[6]We do not attempt here to optimize the concrete efficiency of this secure computation. Given the current speed of secure 2PC protocols for AES, even a naive implementation is expected to be quite efficient.

We turn to describe our communication-efficient technique for eliminating the inverse polynomial error of HSS. In addition to punctured OT, our second ingredient is a simple randomized erasure correcting code.

Lemma 5.2 (Erasure correcting code)**.** *There is a randomized linear encoding function $C_r : \{0,1\}^m \to \{0,1\}^{m+m/\lambda}$ that can correct a $1/\lambda^2$ rate of random erasures with all but $m \cdot \mathsf{negl}(\lambda)$ probability.*

Proof. A message $x \in \{0,1\}^m$ is encoded by $(x, y_1, \ldots, y_{m/\lambda})$ where y_i is the parity of a random subset of $\lambda^2/2 - 1$ bits of x. By a Chernoff bound, except with $m \cdot \mathsf{negl}(\lambda)$ probability, every bit of x is involved in at least $\lambda/3$ sets, where every set (including the corresponding parity check) contains an erasure with at most $\frac{\lambda^2/2}{\lambda^2} = 1/2$ error probability. Hence, for any fixed x_i, the probability that all sets involving x_i contain an erasure is at most $2^{-\lambda/3}$. Hence, the probability that some x_i cannot be recovered is bounded by $m \cdot \mathsf{negl}(\lambda)$ as required. □

Finally, we combine punctured OT and erasure codes to give a succinct 2PC protocol for branching programs. This protocol avoids the use of virtual client-server MPC and can thus achieve better communication rate and computational complexity than its counterpart from Section 4.2.

The protocol is similar to the protocol for branching programs from [BGI16] (cf. Theorem 4.5 in full version), which evaluates m branching programs on inputs of total length n using $n + m \cdot \mathsf{poly}(\lambda)$ bits of communication, except for the following differences. First, instead of repeating each output bit λ times, the functionality is modified so that the outputs are encoded using the randomized erasure code of Lemma 5.2 (where a PRG is used to pick the randomness r with sublinear communication). Second, instead of applying a standard DEHE to compute shares of the output encoding, we use a (multi-evaluation) asymmetric Las Vegas variant in which P_1 outputs $\perp$ whenever there is a risk of error. We set the error parameter δ to be a sufficiently small $1/\mathsf{poly}(\lambda)$ so that: (1) except with $\mathsf{negl}(\lambda)$ probability, the number of $\perp$ outputs is bounded by $k = m/\lambda^2$, and (2) the communication complexity of $\binom{m'}{-k}$-OT, where $m' = m + m/\lambda$, is $m + o(m)$. Finally, P_1 uses punctured OT to retrieve the output shares of P_0 in the positions where it did not output $\perp$. Note that, by the definition of asymmetric Las Vegas HSS, the shares obtained from P_0 are determined by the shares of P_1 and the output (except with negligible probability), and hence they can be simulated given the output.

The above protocol gives rise to the following theorem.

Theorem 5.3 (Optimized 2PC for branching programs)**.** *Assuming DDH, there is a constant-round secure 2-party protocol for evaluating any sequence of m branching programs of size S on inputs (x_0, x_1) of total length n, using $n + (1 + o(1))m + \mathsf{poly}(\lambda)$ bits of communication and $\mathsf{poly}(\lambda) \cdot m \cdot S^2$ computation.*

As a corollary, we get the following near-optimal protocol for OT.

Corollary 5.4 (Constant-rate bit-OT)**.** *Assuming DDH, there is a constant-round secure 2-party protocol for evaluating n instances of bit-OT with $(4 + o(1))n + \mathsf{poly}(\lambda)$ bits of communication and $\mathsf{poly}(\lambda) \cdot n$ computation.*

Combining Corollary 5.4 with the GMW protocol for secure circuit evaluation using bit-OT [GMW87], we get the following corollary.

Corollary 5.5 (MPC for general circuits)**.** *Assuming DDH, there is a secure 2-party protocol for evaluating any circuit C of size S with $O(S) + \mathsf{poly}(\lambda)$ bits of communication.*

This should be compared with a similar protocol from the full version of [BGI16] (cf. Theorem 4.10) in which the communication complexity has an additional $(\mathit{depth} + \mathit{output}) \cdot \mathsf{poly}(\lambda)$ term.

6 Optimizing Computation

A bottleneck of the performance of the HSS scheme in [BGI16] and the schemes in this paper is the computation time of homomorphically evaluating RMS multiplications. The time required for the multiplication is almost entirely the result of $\ell + 1$ executions of ConvertShares and MultShares.

We present three optimizations of these procedures. The first optimizes the *worst case* asymptotic running time of the share conversion algorithm by a $\log(1/\delta)$ factor, but does not improve the *expected* running time. The second optimization, which is incompatible with the first, optimizes the concrete running time of the conversion. The third balances the computational complexity of ConvertShares and MultShares to reduce the overall running time of evaluating an RMS multiplication.

6.1 Improving Worst-Case Conversion Time via Min-Hashing

The first optimization we consider applies to the share conversion algorithm DistributedDLog from Figure 2. This algorithm uses a pseudo-random function to define a sparse set of distinguished group elements, and returns the minimal i such that $g \cdot h^i$ belongs to the distinguished set. This scheme has two disadvantages: first, while its *expected* run time is $O(M/\delta)$ (where M is an upper bound on the difference between inputs and δ is an upper bound on the error probability), its worst-case run time is bigger by a factor of $O(\log 1/\delta)$. This difference can be significant in a scenario of performing many share conversions in parallel, where the worst-case running time dominates. A second, and mainly theoretical, disadvantage is that the conversion is based on a cryptographic assumption instead of being unconditional.

We address both issues by using the alternative conversion procedure described in Figure 6. The high level idea is to apply a hash function ϕ to every group element $h \cdot g^i$ in a range $i = 0, \ldots, T-1$, and return the (first) value of i that minimizes the value of $\phi(h \cdot g^i)$. We argue that if ϕ has the property that the minimum of $\phi(h \cdot g^i)$ is as likely to occur for every choice of i, then the error probability δ is bounded by $2M/T$, making the *worst-case* run time $T = O(M/\delta)$. In general, we can rely on the following approximate version.

Definition 6.1 (min-wise independence)**.** [BCM98, Ind01] Let $\mathcal{H} = \{h_i : [N] \to [N]\}$ be a family of hash functions. We say that $\mathcal{H}$ is (k, ϵ)-*minwise independent* if for any $X \subset [N]$ where $|X| \leq k$ and $x \in [N] \setminus X$ we have

$$\Pr_{h \in \mathcal{H}}[h(x) \leq \min h(X)] \leq \frac{1}{|X| + 1}(1 + \epsilon).$$

Considering X to be the set of i's that both parties have in common, picking h from a min-wise independent family of hash functions ensures that the probability of the minimum being attained outside this set is roughly the ratio between the difference M and the overlap $|X|$.

Algorithm 2 $\mathsf{LVCon}_{b,\mathbb{G},g}(h,\delta,M,\phi)$

```
1: if $b = 0$ then
2:     Let $\mathsf{DangerZone} := \{h, hg, \ldots, hg^M\}$.
3:     if $\exists h' \in \mathsf{DangerZone}$ with $\phi(h') = 0^{\lceil \log(2M/\delta) \rceil}$ then
4:         Return $\perp$.
5:     end if
6: end if
7: Set $h' \leftarrow h$, $i \leftarrow 0$.
8: while $(\phi(h') \neq 0^{\lceil \log(2M/\delta) \rceil})$ do
9:     $h' \leftarrow h' \cdot g$, $i \leftarrow i + 1$.
10: end while
11: Return $i$.
```

Claim 6.2. *Let $0 \le z < M$ and let h_0, h_1 be such that $h_1 = h_0 \cdot g^z$. If ϕ is drawn at random from a (T,ϵ)-*minwise independent *family $\mathcal{H}$, then*

$$\Pr[\mathsf{DistributedDLog}'_{\mathbb{G},g}(h_1,\delta,M,\phi) - \mathsf{DistributedDLog}'_{\mathbb{G},g}(h_0,\delta,M,\phi) \neq z] \le \delta(1+\epsilon)$$

Proof. Consider the set $X_0 = \{h_0 \cdot g^i : 0 \le i < T\}$ and $X_1 = \{h_1 \cdot g^i : 0 \le i < T\}$. Let $X = X_0 \cap X_1$. An error can occur only if the first minimum value of ϕ on X_0 is attained at $X_0 \setminus X_1$ or the first minimum value of ϕ on X_1 is attained at $X_1 \setminus X_0$. Since the size of the difference between the two sets is bounded by $2M$ and the intersection size is $T - M$, the probability of an error is upper bounded by $(1+\epsilon)2M/(T-M) = \delta(1+\epsilon)$. □

The function ϕ can either be instantiated with a PRF, in which case $\epsilon \le \mathsf{negl}(\lambda)$ whenever $M \le \mathsf{poly}(\lambda)$, or unconditionally using a small family of min-wise independent hash functions constructed in [Ind01].

6.2 Optimizing the Conversion

Consider Algorithm 2, which is a one-sided version of the share conversion in Algorithm 1. That is, only Party 0 may indicate a potential failure by outputting $\perp$, and unless failure is indicated the output is guaranteed to be correct. Algorithm 2 executes until a distinguished point is found, and we will therefore only be concerned with its *expected* running time.

A straightforward implementation of this algorithm for a group element $h \in \mathbb{G}$ requires computing the sequence $h, hg, \ldots, hg^i$ for a generator g, computing a pseudo-random function on each element and choosing the first distinguished point (or alternatively the minimal value). A natural strategy for this implementation is to choose the group $\mathbb{G}$ to be a group over elliptic curves, since the resulting ciphertexts would be relatively short and group operations would be relatively fast.

We explore a different implementation of the conversion step which tests whether a sub-sequence of elements $hg^i, \ldots, hg^{i+j}$ includes a distinguished point without explicitly computing each element in the sub-sequence. To achieve this idea we work over groups $\mathbb{Z}_p^*$ with special structure rather than over an EC group and terminate the conversion algorithm upon reaching a distinguished point. Therefore, the following optimizations are incompatible with reducing the worst-case running time via min-hashing as suggested in Section 6.1.

The first idea is to decide if an element $h' = hg^i \in \mathbb{G}$ is distinguished without using a PRF ϕ. We say that h' is distinguished if the binary representation of h' has d leading zeroes, where d is a parameter given to the conversion algorithm, i.e., $h' < 2^{\lceil \log p \rceil - d}$. (One should think of d as being roughly equal to $\log_2(1/\delta)$, where δ is the target failure probability for the conversion algorithm.)

The second idea is to consider pseudo-Mersenne primes, i.e. primes of the form $p = 2^n - \gamma$ for small γ, in which the element 2 generates a large sub-group. We refer to such primes as *conversion friendly.* A class of conversion-friendly primes which are relatively common are pseudo-Mersenne primes p which are safe, i.e. $p = 2q + 1$ for a prime q and which satisfy $p \equiv \pm 1 \bmod 8$. For such primes the sub-group $\mathbb{G}$ that includes all the quadratic residues modulo p is of size q. Since q is prime, every element in $\mathbb{G}$ generates the sub-group and one of these elements is 2 since $p \equiv \pm 1 \bmod 8$. Examples of useful conversion-friendly primes include $2^{1280} - 7243217$, $2^{1536} - 11510609$ and $2^{2048} - 1942289$.

The best currently known attack against DDH over pseudo-Mersenne primes is to use the Special Number Field Sieve (SNFS) [Pol88] to compute discrete logarithms modulo the prime. The current record for using the SNFS to compute discrete logarithms is 1024 bits by Fried et al. [FGHT17]. In the somewhat related task of factoring Mersenne primes, the current record is $2^{1199} - 1$ [KBL14]. To account for the speedup offered by SNFS, the bit-length of the special primes we use needs to be roughly 50% bigger than that of a general prime to provide a similar level of security. For instance, a 2048-bit special p is roughly comparable to a 1340-bit general p [FGHT17].

When working over conversion-friendly primes, computing $2h \bmod p$ given h can be done by shifting the bit representation of h one bit to the left, removing the most significant bit and adding γ to the result if the removed bit is 1. Therefore, computing the next element of the sequence $h, \ldots, hg^i$ involves little more than a comparison of the bit, an addition, and testing whether the d most significant bits of the result are zero.

We empirically validated that, in the groups $\mathbb{Z}_p^*$ we use, if $h \in \mathbb{G}$ is chosen randomly then the MSB sequence of hg^i behaves like a random bit-string with respect to the expected number of steps required to find a sequence of d zeros, which is roughly 2^{d+1} (it is precisely $2^{d+1} - 2$ [Nie73]). Note that it is easy to modify the conversion algorithm to use a random starting point h, since the two servers can shift their respective elements h_0, h_1 by the same (pseudo-)random element r, maintaining the difference between h_0 and h_1. Thus, to ensure conversion error δ, it suffices to use $d = \lceil \log_2(1/\delta) \rceil + 1$.

Further savings in the computation of the MSB sequence are possible by taking advantage of hardware architectures that enable fast multiplication of w-bit words. If $h = a_1 2^{n-w} + a_0$ for $0 \le a_0 < 2^{n-w}, 0 \le a_1 < 2^w$ then $2^w h \equiv a_0 2^w + a_1 \gamma \bmod p$. Note that if $\gamma << 2^w$ then computing $2^w h$ requires one multiplication of words and with high probability one addition of words.

It is possible to test if any of the $2w - d$ elements $h, 2h \bmod p, \ldots, 2^{2w-d-1} h \bmod p$ are distinguished by checking whether the $2w$ most significant bits of h include the substring 0^d. That can be done efficiently in a standard computer architecture by dividing the $2w$ bits into strips of length $d/2$ and checking whether any of the strips is $0^{d/2}$. If none of the strips are $0^{d/2}$ then the sequence $h, 2h, \ldots, 2^{w-d/2-1} h$ does not contain a distinguished point. In this case, after finding the longest 0 suffix of the first $2w$ bits of h, the next element to be examined is $2^w h$. If one of the strips is $0^{d/2}$ then the bits on both its sides are examined to check if the strip is part of a 0^d pattern, which determines a distinguished point.

To conclude, by using conversion-friendly groups and the zero-sequence heuristic, the implementation of the share conversion algorithm reduces to the following two tasks: (1) generating the

MSB sequence of the group elements hg^i, $i = 0, 1, 2, \ldots$; (2) finding the first occurrence of the pattern 0^d in a given bit-sequence. Both tasks can be implemented using less than a machine word operation per step i, for sufficiently large w.

Define the number of *conversion steps* that Algorithm 2 makes to be 0 if the algorithm returns $\perp$ and i if it returns i. An interesting property of the above algorithm for finding distinguished points is that it is almost independent of the size of the underlying group. Our implementation of the algorithm achieved an average rate of 1.6 billion conversion steps a second. The implementation ran on a commodity laptop, Dell Latitude 3550, with an Intel i7-5500 CPU based on the Broadwell architecture, running single-threaded at 2.4 GHz. The implementation used 32-bit words together with multiplications of two 32 bit operands into a 64 bit product.

Word size	Multiplications per step	Additions per step	Masking operations per step	No. of Conversion steps per second
32 bits	0.031	0.031	0.22	1.6 billion
w bits	$\frac{1}{w}$	$\frac{1}{w} + \frac{\gamma}{2^w}$	$\frac{2}{w}(\lceil \frac{w}{d} \rceil + \frac{d}{2^{d/2}})$	–

Table 1: Performance figures for the conversion step over a prime $p = 2^n - \gamma$ with d zero bits determining a distinguished point.

Table 1 counts the average number of multiplications, additions and bit level operations (bit-by-bit logical AND of words and shifts) for a conversion step in a 32-bit architecture and in a general w-bit architecture. The table makes it clear that a conversion step can require on average below a single machine word operation per step. Our actual implementation requires slightly more than one operation per step but that is possibly due to other system considerations such as memory access time.

The natural alternative to our algorithm, as previously noted, is to let $\mathbb{G}$ be a group over an elliptic curve, compute the sequence $h, g \cdot h, \ldots, g^i \cdot h$ by i point additions and test whether each one is a distinguished point. Even if one uses a good implementation for elliptic-curves that achieves four million point additions per second for 160-bit curves, this is still slower by 2-3 orders of magnitude compared to conversions over conversion-friendly primes.

6.3 Optimizing RMS Multiplication

An RMS multiplication involves multiplying an input w by a memory variable y. An input w is represented by a pair (g^a, g^{ac+w}) and ℓ pairs $(g^{a_t}, g^{a_t c + wc^{(t)}})$ for an ℓ-bit ElGamal private key $c = c^{(1)}, \ldots, c^{(\ell)}$. A memory variable y is represented by a (subtractive) sharing of y and of yc over the integers. The homomorphic evaluation of wy requires two modular exponentiations and a conversion procedure for each of these $\ell + 1$ pairs. Consider the following optimizations.

Short key. In a "short-key" version of ElGamal the key size is reduced to twice the value of a concrete security parameter [KK04]. Even for Pseudo-Mersenne primes the best known attacks on a short key c run in time $2^{c/2}$, e.g. giant-step baby-step, Pollard's Rho algorithm or Pollard's Lambda Algorithm. Therefore, choosing a 160-bit key is sufficient for 80 bits of security. In addition to using a short key, we also assume that (short-key) ElGamal encryption is circularly secure, namely it remains secure even when given the encryptions of the bits of the secret key.

Large key basis. Instead of representing the ElGamal secret key c by ℓ bits $c^{(1)}, \ldots, c^{(\ell)}$ the key can be represented in a larger basis B by using $\ell' = \lceil \ell / \log B \rceil$ digits $e^{(1)}, \ldots, e^{(\ell')}$. The impact on RMS evaluation is that an ElGamal ciphertext $(g^{a_t}, g^{a_t c + we^{(t)}})$ encrypts an integer $we^{(t)}$ in the range $0, \ldots, B-1$ and therefore the average running time of the conversion algorithm increases by a factor of B. On the other hand, the number of exponentiations, the required storage and the ciphertext size are all reduced by a factor of $\log_2 B$.

Time-memory trade-off for fixed base exponentiation. Every exponentiation in evaluating RMS multiplications uses a base that is part of the encrypted input. For a given integer R, which we call the *tradeoff parameter*, a base h can be replaced by $\{h^{j \cdot 2^{iR}} \mid 0 \leq i < \ell/R,\ 0 < j < 2^R\}$ reducing exponentiation to a product of appropriate elements in this set. The number of modular multiplications without the optimization is more than ℓ (e.g. with a sliding-window algorithm) and is only $\lceil \ell/R \rceil - 1$ given the optimization. Therefore, as R grows the computation time decreases linearly with R while the required memory increases to $\lceil \ell/R \rceil \cdot (2^R - 1)$ group elements.

The following table summarizes various performance measures for a single binary RMS multiplication. It uses approximations of the analytic expressions described in the following paragraphs.

Parameter	Analytic expression
Failure probability	$\lceil \ell/\log B + 1 \rceil (B-1)/2^d$
Group operations	$\lceil \ell/\log B + 1 \rceil \frac{\ell+2d}{R}$
Expected conv. steps	$\lceil \ell/\log B + 1 \rceil 2^{d+1}$
Public key size (DEHE)	$2\lceil \ell/\log B \rceil + 5$
Share size per input (HSS)	$\lceil \ell/\log B + 2 \rceil$
Ciphertext size per input (DEHE)	$2\lceil \ell/\log B + 1 \rceil$
Preprocessing memory	$\lceil \ell/\log B + 1 \rceil (\ell + 2d)(2^R - 1)/R$

Table 2: Parameters of a single RMS multiplication of binary values as a function of ℓ (secret key bit length, $\ell = 160$ by default), B (basis size for representing secret key), R (modular exponentiation preprocessing parameter, $R = 1$ by default), and d (zero-sequence length for the conversion algorithm). All sizes are measured in group elements.

Concrete choice of parameters. We now give concrete analytical expressions that show some possible tradeoffs between the different parameters. For the following analysis, we let p be a conversion-friendly prime and $\mathbb{G}$ be the group of quadratic residues modulo p. As discussed above, we make the (empirically validated) assumption that the bit sequence defined by the MSB of hg^i, $i = 0, 1, 2, \ldots$, with a random starting point h, behaves like a random bit sequence as far as the expected first occurrence of the pattern 0^d goes.

We further let ℓ denote the bit-length of the secret key (typically $\ell = 160$), B denote the key basis (typically $B = 2, 4$ or 16) and R a tradeoff parameter that reduces the computational cost of fixed-basis exponentiations via precomputation, as discussed above (typically $R = 1$ or 8). We let α denote the number of conversion steps per second (roughly $1.6 \cdot 10^9$ in our implementation), and γ denote the number of multiplications modulo p per second (we use $\gamma = 10^6$ for 1536-bit primes). Realizing RMS multiplication of binary values with one-sided failure probability ϵ, as in Algorithm 2, it suffices to let the distinguished point pattern length be $d = \left\lceil \log \frac{(B-1)(\lceil \ell/\log B \rceil + 1)}{\epsilon} \right\rceil$. This setting of parameters gives rise to the following performance:

- The number of (binary) RMS multiplications per second is at least

$$\frac{\alpha\gamma}{\lceil\frac{\ell}{\log B}+1\rceil(\frac{\alpha(\ell+2d+3R)}{R}+\gamma\cdot 2^{d+1})}.$$

- The share size per input is $2\lceil\ell/\log B\rceil+3$ group elements for both HSS and DEHE variants with security in the standard model. The share size for the HSS variant is $\lceil\ell/\log B\rceil+2$ by using the heuristic compression of ElGamal ciphertexts suggested in [BGI16] (or alternatively proving security in the random oracle model).

- The extra memory per input used for speeding up fixed-base exponentiations is at most $S \triangleq \lceil\frac{\ell}{\log B}+1\rceil(\lceil\frac{\ell+d}{R}\rceil+\lceil\frac{d}{R}\rceil)(2^R-1)$ group elements if $R>1$ and is zero if $R=1$.

- The (offline) computation time per input is S/γ seconds.

In the following we provide the detailed calculation leading to the above expressions. The target error probability for the whole RMS operation is ϵ and therefore the error probability of each of the $\lceil\ell/\log B\rceil+1$ conversions must be no more than $\epsilon/(\lceil\ell/\log B\rceil+1)$. An error in the conversion occurs if a distinguished point exists between the two elements that are converted to integers. The maximum distance between these elements in the cycle over $\mathbb{G}$ is $B-1$ and therefore the probability of a random element being a distinguished point must be at most $\epsilon/(B-1)(\lceil\ell/\log B\rceil+1)$. As a consequence, the number of leading zeroes, d, which defines a distinguished point is $d=\lceil\log\frac{(B-1)(\lceil\ell/\log B\rceil+1)}{\epsilon}\rceil$.

The key is represented in base B by $c=(e^{(\lceil\ell/\log B\rceil)},\ldots,e^{(1)})$. The share size for an input w is determined by the representation of w as the subtractive shares of w and of wc over $\mathbb{Z}_q$ and by the $\lceil\ell/\log B\rceil+1$ pairs $(g^{a_t},g^{a_tc+we^{(t)}})\in\mathbb{G}^2$, $t=1,\ldots,\lceil\ell/\log B\rceil$ and $(g^a,g^{ac+we^{(t)}})\in\mathbb{G}^2$. The subtractive shares are elements in $\mathbb{Z}_q$ and their size in our implementation is less than the size of a single element in $\mathbb{G}$. Together with the $\ell+1$ pairs of elements in $\mathbb{G}$ there are at most $2(\lceil\ell/\log B\rceil+2)$ group elements altogether.

A possible optimization for the share size in the HSS variant is introduced in [BGI16] and suggests choosing the first element in each of the ElGamal pairs from a pseudo-random distribution. Then, the share includes a seed for a pseudo-random generator instead of the $\ell+1$ first elements of the pair. The result is that the share size is only $\lceil\ell/\log B\rceil+2$ at the cost of another heuristic security assumption (or alternatively security in the random oracle model). Note that this optimization does not work in the DEHE variant since a client does not know the ElGamal secret key.

A memory variable y is represented by subtractive shares of y and yc over the integers. The expected value of a share y_1 of y is the expected distance from a random element in $\mathbb{G}$ to a distinguished point, which is assumed to be 2^d. For similar reasons, since $c<2^\ell$, the expected value of a share $(yc)_1$ of yc is at most $2^{\ell+d}$. RMS multiplication of an input w by a memory variable y begins by computing $\lceil\ell/\log B\rceil+1$ elements of the type $(g^{a_t})^{(yc)_1}\cdot(g^{a_tc+we^{(t)}})^{y_1}$. Using the time-memory tradeoff optimization with parameter R for that computation requires calculating and storing the elements $\{(g^{a_t})^{j\cdot 2^{iR}}\mid 0\le i<(\ell+d)/R,\ 0<j<2^R\}$ and $\{(g^{a_tc+we^{(t)}})^{j\cdot 2^{iR}}\mid 0\le i<d/R,\ 0<j<2^R\}$. Therefore the total number of stored group elements is

$$\left\lceil\frac{\ell}{\log B}+1\right\rceil\Big(\left\lceil\frac{\ell+d}{R}\right\rceil+\left\lceil\frac{d}{R}\right\rceil\Big)(2^R-1)$$

and each stored element can be computed with a single modular multiplication.

To compute a bound on the number RMS multiplications per second we separate each RMS multiplication to a conversion phase and a modular multiplication phase and bound the rate at which these phases can be computed. In a single RMS multiplication there are $\lceil \ell / \log B \rceil + 1$ conversions. Taking into account the expected number of steps in a single conversion and the rate of conversion steps per second we have that the expected rate of conversion phases per second is $\frac{\alpha}{\lceil \ell / \log B + 1 \rceil 2^{d+1}}$. The number of modular multiplications in a modular multiplication phase is $\lceil \ell / \log B + 1 \rceil (\lceil d/R \rceil + \lceil (\ell + d)/R \rceil + 1)$. Therefore, the rate of modular-multiplication phases per second is at least $\frac{\gamma}{\lceil \ell / \log B + 1 \rceil ((\ell + 2d)/R + 3)}$. We compute the number of RMS multiplications per second by allocating a fraction η of a second to the conversion phases, a fraction of $1 - \eta$ to the multiplication phases and noting that

$$\frac{\eta \alpha}{\lceil \ell / \log B + 1 \rceil 2^{d+1}} = \frac{(1 - \eta)\gamma}{\lceil \ell / \log B + 1 \rceil ((\ell + 2d)/R + 3)}$$

and that the two quantities are equal to the rate of RMS multiplications. Solving for η we obtain the above expression.

In Table 3 we summarize the performance of the system for several typical parameters. We note that using by faster (but still inexpensive) hardware and parallelism, e.g. [Por15], one could realize thousands of RMS multiplications per second with failure probability that suffices for realizing simple but nontrivial computations.

Error	Base	Tradeoff param.	Length of dist. point	Share (bytes)	Memory (bytes)	RMS mult. per second
$\epsilon = 2^{-5}$	$B = 4$	$R = 1$	$d = 13$	21000	42000	62
	$B = 4$	$R = 8$	$d = 13$	21000	$127 \cdot 10^6$	338
	$B = 16$	$R = 1$	$d = 15$	10750	21500	104
	$B = 16$	$R = 8$	$d = 15$	10750	$64.2 \cdot 10^6$	360
$\epsilon = 2^{-10}$	$B = 4$	$R = 1$	$d = 18$	21000	42000	23.4
	$B = 4$	$R = 8$	$d = 18$	21000	$137.5 \cdot 10^6$	34.7
	$B = 16$	$R = 1$	$d = 20$	10750	21500	16.1
	$B = 16$	$R = 8$	$d = 20$	10750	$69.6 \cdot 10^6$	18.2
$\epsilon = 2^{-15}$	$B = 4$	$R = 1$	$d = 23$	21000	42000	1.15
	$B = 4$	$R = 8$	$d = 23$	21000	$137.5 \cdot 10^6$	1.17
	$B = 16$	$R = 1$	$d = 25$	10750	21500	0.57
	$B = 16$	$R = 8$	$d = 25$	10750	$75 \cdot 10^6$	0.58

Table 3: Performance of RMS multiplications with prime $2^{1536} - 11510609$, key length $\ell = 160$, $\alpha = 1.6 \cdot 10^9$ conversion steps per second (as measured on a standard Dell Latitude 3550 laptop, with an Intel i7-5500 CPU based on the Broadwell architecture, running single-threaded at 2.4 GHz), and assuming $\gamma = 10^6$ modular multiplications per second.

Acknowledgements. We thank Antoine Joux for discussions, suggestions, and pointers that helped improve the results of Section 6. We also thank Geoffroy Couteau and the anonymous Eurocrypt reviewers for helpful comments.

The first author was supported by ISF grant 1861/16, AFOSR Award FA9550-17-1-0069, and ERC starting grant 307952. The second author was supported by ISF grant 1638/15, a grant by the BGU Cyber Center, the Israeli Ministry Of Science and Technology Cyber Program and by the European Union's Horizon 2020 ICT program (Mikelangelo project). The third 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. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense, the National Science Foundation, or the U.S. Government.

References

[AIK05] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. Computationally private randomizing polynomials and their applications. In *CCC*, pages 260–274, 2005.

[AJLA$^+$12] Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. Multiparty computation with low communication, computation and interaction via threshold FHE. In *EUROCRYPT*, pages 483–501, 2012.

[BCM98] Andrei Z. Broder, Moses Charikar, and Michael Mitzenmacher. A derandomization using min-wise independent permutations. In *RANDOM*, pages 15–24, 1998.

[BGI14] Elette Boyle, Shafi Goldwasser, and Ioana Ivan. Functional signatures and pseudorandom functions. In *PKC*, pages 501–519, 2014.

[BGI15] E. Boyle, N. Gilboa, and Y. Ishai. Function secret sharing. In *Advances in Cryptology - EUROCRYPT*, pages 337–367, 2015.

[BGI16] Elette Boyle, Niv Gilboa, and Yuval 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).

[BHHO08] Dan Boneh, Shai Halevi, Michael Hamburg, and Rafail Ostrovsky. Circular-secure encryption from decision diffie-hellman. In *CRYPTO 2008*, pages 108–125, 2008.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *STOC*, pages 503–513, 1990.

[Bra84] Gabriel Bracha. An asynchronous $[(n-1)/3]$-resilient consensus protocol. In *PODC*, pages 154–162, 1984.

[BRS02] John Black, Phillip Rogaway, and Thomas Shrimpton. Encryption-scheme security in the presence of key-dependent messages. In *SAC*, pages 62–75, 2002.

[BW13] Dan Boneh and Brent Waters. Constrained pseudorandom functions and their applications. In *ASIACRYPT*, pages 280–300, 2013.

[Can00] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, pages 143–202, 2000.

[CC06] Hao Chen and Ronald Cramer. Algebraic geometric secret sharing schemes and secure multi-party computations over small fields. In *CRYPTO*, pages 521–536, 2006.

[CGGI16] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *ASIACRYPT*, pages 3–33, 2016.

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

[DHRW16] Yevgeniy Dodis, Shai Halevi, Ron D. Rothblum, and Daniel Wichs. Spooky encryption and its applications. In *CRYPTO*, pages 93–122, 2016.

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

[FGHT17] Joshua Fried, Pierrick Gaudry, Nadia Heninger, and Emmanuel Thomé. A kilobit hidden SNFS discrete logarithm computation. In *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part I*, pages 202–231, 2017.

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

[GGHR14] Sanjam Garg, Craig Gentry, Shai Halevi, and Mariana Raykova. Two-round secure MPC from indistinguishability obfuscation. In *TCC*, pages 74–94, 2014.

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *J. ACM*, 33(4):792–807, 1986.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *STOC*, pages 218–229, 1987.

[Gol04] Oded Goldreich. *Foundations of Cryptography — Basic Applications*. Cambridge University Press, 2004.

[HK07] Omer Horvitz and Jonathan Katz. Universally-composable two-party computation in two rounds. In *CRYPTO*, pages 111–129, 2007.

[HM00] Martin Hirt and Ueli M. Maurer. Player simulation and general adversary structures in perfect multiparty computation. *J. Cryptology*, 13(1):31–60, 2000.

[HS15] Shai Halevi and Victor Shoup. Bootstrapping for HElib. In *EUROCRYPT*, pages 641–670, 2015.

[IK00] Yuval Ishai and Eyal Kushilevitz. Randomizing polynomials: A new representation with applications to round-efficient secure computation. In *FOCS*, pages 294–304, 2000.

[IK02] Yuval Ishai and Eyal Kushilevitz. Perfect constant-round secure computation via perfect randomizing polynomials. In *ICALP*, pages 244–256, 2002.

[IKOS08] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Cryptography with constant computational overhead. In *STOC*, pages 433–442, 2008.

[IKOS09] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge proofs from secure multiparty computation. *SIAM J. Comput.*, 39(3):1121–1152, 2009.

[Ind01] Piotr Indyk. A small approximately min-wise independent family of hash functions. *J. Algorithms*, 38(1):84–90, 2001.

[KBL14] Thorsten Kleinjung, Joppe W. Bos, and Arjen K. Lenstra. Mersenne factorization factory. In *ASIACRYPT*, pages 358–377, 2014.

[KK04] Takeshi Koshiba and Kaoru Kurosawa. Short exponent diffie-hellman problems. In *PKC*, pages 173–186, 2004.

[KPTZ13] Aggelos Kiayias, Stavros Papadopoulos, Nikos Triandopoulos, and Thomas Zacharias. Delegatable pseudorandom functions and applications. In *CCS*, pages 669–684, 2013.

[LP09] Yehuda Lindell and Benny Pinkas. A proof of security of Yao's protocol for two-party computation. *J. Cryptology*, 22(2):161–188, 2009.

[LTV12] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *STOC*, pages 1219–1234, 2012.

[MW16] Pratyay Mukherjee and Daniel Wichs. Two round multiparty computation via multi-key FHE. In *EUROCRYPT*, pages 735–763, 2016.

[Nie73] Peter Nielsen. On the expected duration of a search for a fixed pattern in random data (corresp.). *IEEE Transactions on Information Theory*, 19(5):702–704, 1973.

[NN01] Moni Naor and Kobbi Nissim. Communication preserving protocols for secure function evaluation. In *STOC*, pages 590–599, 2001.

[Pol88] John Pollard. Factoring with cubic integers. Unpublished Manuscript, 1988.

[Por15] Thomas Pornin. Optimizing MAKWA on GPU and CPU. Cryptology ePrint Archive, Report 2015/678, 2015.

[RAD78] Ronald L. Rivest, Len Adleman, and Michael L. Dertouzos. On data banks and privacy homomorphisms. In *Foundations of secure computation*, pages 169–179. 1978.

[SW14] Amit Sahai and Brent Waters. How to use indistinguishability obfuscation: deniable encryption, and more. In *STOC*, pages 475–484, 2014.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, pages 162–167, 1986.

A “BGI” HSS Scheme: from [BGI16]

For completeness, we present the complete BGI homomorphic secret sharing scheme construction. Figure 2 includes relevant notations, pairing operations, and the share conversion algorithm DistributedDLog. Figure 8 gives the corresponding algorithms Enc and Eval.

Remark A.1 (Additive vs Subtractive Sharing)**.** The HSS constructions of [BGI16] maintain values in a positive form of additive and multiplicative secret shares for notational convenience, but then must convert back and forth to their inverse (ie., subtractive or division shares) for computations. In the diagrams here, we directly represent in the negative versions, to remove the need for extra inversion computation steps.

Secret Sharing Notation. For small $x \in \mathbb{Z}$ (or $x \in \mathbb{Z}_q$ for the case of $\langle x \rangle$).

Items in which *both* parties receive same value.

- $[\![x]\!]_c = (h_1, h_2) \in \mathbb{G}^2$ for which $h_2/(h_1)^c = g^x$. I.e., ElGamal ciphertext of x w.r.t. key c.

Items in which each party receives a separate share.

- $\langle x \rangle$ = Additive secret shares $(x_1, x_2) \in \mathbb{Z}_q^2$ for which $x_1 - x_2 = x \in \mathbb{Z}_q$.
- $\langle\!\langle x \rangle\!\rangle$ = "Multiplicative" secret shares $(h_1, h_2) \in \mathbb{G}^2$ for which $h_1/h_2 = g^x \in \mathbb{G}$.

Pairing Operations.
Let $\phi : \{0,1\}^\lambda \times \mathbb{G} \to \{0,1\}^\ell$ be a given PRF.

- $\mathsf{MultShares}\Big([\![x]\!]_c, \langle y \rangle, \langle cy \rangle\Big) \to \langle\!\langle xy \rangle\!\rangle$.
  1. Denote $[\![x]\!]_c = (h_1, h_2) \in \mathbb{G}^2$.
  2. Compute $\langle\!\langle xy \rangle\!\rangle = h_2^{\langle y \rangle} h_1^{-\langle cy \rangle}$.
- $\mathsf{ConvertShares}(b, \langle\!\langle x \rangle\!\rangle, \mathsf{id}, \delta, M) \to \langle x \rangle$, with party identifier $b \in \{0,1\}$, execution identifier id, error parameter δ and max size bound M.
  1. Denote by $\phi' : \mathbb{G} \to \{0,1\}^{\lceil \log(2M/\delta) \rceil}$ the appropriate prefix output of $\phi(\mathsf{id}, \cdot)$.
  2. Let x_b denote the present party b's share of $\langle\!\langle x \rangle\!\rangle$.
  3. Output $i_b \leftarrow \mathsf{DistributedDLog}_{\mathbb{G},g}(x_b, \delta, M, \phi')$.

Share Conversion Sub-Routine. $\mathsf{DistributedDLog}_{\mathbb{G},g}(h, \delta, M, \phi)$

```
1: Set $h' \leftarrow h$, $i \leftarrow 0$. Let $T := [2M \ln(2/\delta)]/\delta$.
2: while ($\phi(h') \neq 0^{\lceil \log(2M/\delta) \rceil}$ and $i < T$) do
3:     $h' \leftarrow h' \cdot g$, $i \leftarrow i + 1$.
4: end while
5: Output $i$.
```

Figure 2: Notation, pairing operations, and share conversion algorithm, as used in [BGI16]. For simplicity we describe the scheme with *subtractive* (and *division*) secret sharing instead of converting back and forth between additive and subtractive (resp., multiplicative and division) shares; see Remark A.1.

One-Round Client-Server HSS (using PKI): m-client secret sharing protocol Π_{1r}.
Global parameters: $\mathbb{G}, g, q$. Let ℓ' be the output size of Decomp.
Inputs: Each client C_i for $i \in [m]$ holds input $w_i \in \{0,1\}$.
Outputs: Each server S_b for $b \in \{0,1\}$ learns share_b of all inputs.

Public-Key Infrastructure: Each client C_i's public-key information consists of:

- An ElGamal private key $c_i \leftarrow \mathbb{Z}_q$, known exclusively by C_i.
- A public key $\mathsf{pk}_i = \left(e_i, (\llbracket \hat{c}_i^{(t)} \rrbracket_{c_i})_{t\in[\ell']}\right)$ consisting of:
  - ElGamal public key $e_i = g^{c_i}$.
  - For $t \in [\ell']$, an ElGamal ciphertext under key e_i of the t'th symbol of $(\hat{c}_i^{(t)})_{t\in[\ell']} \leftarrow \mathsf{Decomp}(c_i)$; i.e., $\llbracket \hat{c}_i^{(t)} \rrbracket_{c_i} \leftarrow \mathsf{Enc}_{\mathsf{ElGamal}}(e_i, \hat{c}_i^{(t)})$.

Client Round 1: Each client $i \in [m]$ performs the following:

1. Generate ciphertexts of "owned" data w_i and $\hat{c}_i^{(t)} w_i$ under self key e_i:
   (a) Encrypt input w_i: i.e., $\llbracket w_i \rrbracket_{c_i} \leftarrow \mathsf{Enc}_{\mathsf{ElGamal}}(e_i, w_i)$.
   (b) For each $t \in [\ell']$, encrypt $\hat{c}_i^{(t)} w_i$: i.e., $\llbracket \hat{c}_i^{(t)} w_i \rrbracket_{c_i} \leftarrow \mathsf{Enc}_{\mathsf{ElGamal}}(e_i, \hat{c}_i^{(t)} w_i)$.
2. Generate ciphertexts of "joint" data $\hat{c}_j^{(t)} w_i$ under *pairwise* keys $e_i e_j$ for $j \neq i$:
   For each client $j \in [m], j \neq i$ and key-bit $t \in [\ell']$, generate ciphertext of $\hat{c}_j^{(t)} w_i$ as follows:
   (a) Let e_j and $\llbracket \hat{c}_j^{(t)} \rrbracket_{c_j}$ denote the ElGamal public key and tth-key-bit ciphertext within the public key pk_j of client j.
   (b) Sample a fresh encryption of 0 under key $e_i e_j$; i.e., $(h_1^0, h_2^0) \leftarrow \mathsf{Enc}_{\mathsf{ElGamal}}(e_i e_j, 0)$.
   (c) Let $(h_1, h_2) = \llbracket \hat{c}_j^{(t)} \rrbracket_{c_j} \in \mathbb{G}^2$ within the public key of Client C_j.
   (d) Compute $\llbracket \hat{c}_j^{(t)} w_i \rrbracket_{c_i + c_j}$ as follows:
      i. Let $(h_1', h_2') = (h_1^{w_i}, h_2^{w_i}(h_1^{w_i})^{c_i})$. //Decrypts to $\hat{c}_j^{(t)} w_i$ with key $c_i + c_j$
      ii. Rerandomize using the ciphertext of 0. Namely, take $\llbracket \hat{c}_j^{(t)} w_i \rrbracket_{c_i+c_j} = (h_1' h_1^0, h_2' h_2^0)$.
3. Send all ciphertexts $\llbracket w_i \rrbracket_{c_i}, \{\llbracket \hat{c}_i^{(t)} w_i \rrbracket_{c_i}\}_{t\in[\ell']}, \{\llbracket \hat{c}_j^{(t)} w_i \rrbracket_{c_i+c_j}\}_{i \neq j \in [m], t \in [\ell']}$ to both servers.
4. Other items:
   (a) Produce an additive secret sharing $\langle c_i \rangle \leftarrow \mathsf{AdditiveShare}(c_i)$ of the key c_i. and send each resulting share $\langle c_i \rangle_b$ to the corresponding server b.
   (b) Sample a random string $r_i \leftarrow \{0,1\}^\lambda$ (for PRF seed) and send r_i to both servers.

Server Output: Each server $b \in \{0,1\}$ performs the following:

1. Take $r = \sum_{i\in[m]} r_i$. Let $\phi = \mathsf{PRFGen}(1^\lambda; r)$ be a PRF from $\{0,1\}^\lambda \times \mathbb{G} \to \{0,1\}^{\ell'}$.
2. Let $\mathsf{share}_b = \left(m, \phi, \left\{\langle c_i \rangle_b\right\}_{i\in[m]}, \left(\llbracket w_i \rrbracket_{c_i}, \{\llbracket \hat{c}_i^{(t)} w_i \rrbracket_{c_i}\}_{t\in[\ell']}, \{\llbracket \hat{c}_j^{(t)} w_i \rrbracket_{c_i+c_j}\}_{i \neq j \in [m], t\in[\ell']}\right)_{i\in[m]}\right)$.

Figure 3: One-round m-client 2-server HSS secret sharing protocol Π_{1r}. ($\mathsf{Decomp}, \mathsf{Recomp}$) refer to a decomposition procedure with low-magnitude shares and linear reconstruction (generalizing bit decomposition).

One-Round Client-Server Homomorphic Evaluation $\mathsf{Eval}_{\mathbb{G},g}(b, \mathsf{share}, P, \delta)$
Inputs: Party identifier $b \in \{0,1\}$, homomorphic secret share value share, RMS program description P of size $\leq S$, error bound δ.

Parse share as in Figure 3. Parse P as a magnitude bound 1^M and sequence of RMS instructions. Take $\delta' = \delta/((\ell'+1)MS)$.

```
Instruction $(\mathsf{id}, \hat{y}_j \leftarrow \hat{w}_\alpha)$:
    1: Let $\langle y_j \rangle \leftarrow \langle w_\alpha \rangle$ and $\langle c_\gamma y_j \rangle \leftarrow \langle c_\gamma w_\alpha \rangle$ for each $\gamma \in [m]$, where $\langle w_\alpha \rangle, \langle c_\gamma w_\alpha \rangle$ are as in $\mathsf{share}$.

Instruction $(\mathsf{id}, \hat{y}_k \leftarrow \hat{y}_i + \hat{y}_j)$:
    1: Compute $\langle y_k \rangle \leftarrow \langle y_i \rangle + \langle y_j \rangle$, directly on the additive shares (over $\mathbb{Z}_q$).
    2: Compute $\langle c_\gamma y_k \rangle \leftarrow \langle c_\gamma y_i \rangle + \langle c_\gamma y_j \rangle$ for each $\gamma \in [m]$, directly on additive shares (over $\mathbb{Z}_q$).

Instruction $(\mathsf{id}, \hat{y}_k \leftarrow \hat{w}_\alpha \cdot \hat{y}_j)$:
    1. Produce shares $\langle w_\alpha y_j \rangle$ (using $[\![w_\alpha]\!]_{c_\alpha}$ and $\langle y_j \rangle, \langle c_\alpha y_j \rangle$):
        1: Compute $\langle\!\langle w_\alpha y_j \rangle\!\rangle = \mathsf{MultShares}([\![w_\alpha]\!]_{c_\alpha}, \langle y_j \rangle, \langle c_\alpha y_j \rangle)$, as in Figure 2.
        2: Execute $\langle w_\alpha y_j \rangle = \mathsf{ConvertShares}(b, \langle\!\langle w_\alpha y_j \rangle\!\rangle, (\mathsf{id}, 0), \delta', M, \phi)$, as in Figure 2.
        3: Set $\langle y_k \rangle \leftarrow \langle w_\alpha y_j \rangle$.
    2. Produce shares $\langle c_\gamma w_\alpha y_j \rangle$ for each $\gamma \in [m]$: (using $[\![\hat{c}_\gamma^{(t)} w_\alpha]\!]_{c_\alpha + c_\gamma}$ and $\langle y_j \rangle, \langle c_\alpha y_j \rangle, \langle c_\gamma y_j \rangle$)
        1: for $\gamma = 1$ to $m$ do
        2:     if $\alpha = \gamma$ then define $c_{\alpha,\alpha} := c_\alpha$. Let $\langle c_{\alpha,\alpha} y_j \rangle = \langle c_\alpha y_j \rangle$.
        3:     else define $c_{\alpha,\gamma} := c_\alpha + c_\gamma$. Compute $\langle c_{\alpha,\gamma} y_j \rangle = \langle c_\alpha y_j \rangle + \langle c_\gamma y_j \rangle$.
        4:     end if
        5:     for $t = 1$ to $\ell'$ do
        6:         Compute $\langle\!\langle \hat{c}_\gamma^{(t)} w_\alpha y_j \rangle\!\rangle = \mathsf{MultShares}([\![\hat{c}_\gamma^{(t)} w_\alpha]\!]_{c_{\alpha,\gamma}}, \langle y_j \rangle, \langle c_{\alpha,\gamma} y_j \rangle)$.
        7:         Execute $\langle \hat{c}_\gamma^{(t)} w_\alpha y_j \rangle = \mathsf{ConvertShares}(b, \langle\!\langle \hat{c}_\gamma^{(t)} w_\alpha y_j \rangle\!\rangle, (\mathsf{id}, t), \delta', M, \phi)$.
        8:     end for
        9:     Compute $\langle c_\gamma w_\alpha y_j \rangle = \mathsf{Recomp}((\langle \hat{c}_\gamma^{(t)} w_\alpha y_j \rangle)_{t \in [\ell']})$.
        10:    Set $\langle c_\gamma y_k \rangle \leftarrow \langle c_\gamma w_\alpha y_j \rangle$.
        11: end for

Instruction $(\mathsf{id}, \beta, \hat{O}_j \leftarrow \hat{y}_i)$:
    1: Shift $\langle y_i \rangle$ share by rerandomization offset: $\langle y_i \rangle \leftarrow \langle y_i \rangle + \phi(\mathsf{id}, g)$, over $\mathbb{Z}_q$.
       // Note that shifting both shares does not change the shared value in $\mathbb{Z}_q$
    2: Convert share from $\mathbb{Z}_q$ to $\mathbb{Z}_\beta$: $\langle O_j \rangle \leftarrow \langle y_i \rangle \mod \beta$.
    3: Output $\langle O_j \rangle$.
```

Figure 4: One-round m-client 2-server homomorphic evaluation algorithm Eval. Evaluation maintains the invariant that for each memory value $\hat{y}_i$ the servers hold: (1) additive shares $\langle y_i \rangle$, and (2) m sets of additive shares $\langle c_\alpha y_i \rangle$, for the secret key c_α of *each* of the m clients. Here, i, j, k denote memory indices, $t \in [\ell']$ denotes an index of a key representation, and $\alpha, \gamma \in [m]$ denote client ids.

Simulatable Las Vegas Homomorphic Evaluation $\mathsf{Eval}^{\mathsf{SLV}}_{\mathbb{G},g}(b, \mathsf{share}, P, \delta)$
Inputs: Party identifier $b \in \{0,1\}$, homomorphic secret share value share, RMS program description P of size $\leq S$, error bound δ.

Parse share as in Figure 3. Parse P as in Definition 2.1. Take $\delta' = \delta/((\ell+1)mMS)$. Initialize $\mathsf{LeakageInfo} \leftarrow 0$. For each instruction $(\mathsf{id}, \hat{y}_j \leftarrow \hat{w}_\alpha)$ or $(\mathsf{id}, \hat{y}_k \leftarrow \hat{y}_i + \hat{y}_j)$ in P, evaluate as in Figure 4. For each other instruction in P, perform the following:

Instruction $(\mathsf{id}, \hat{y}_k \leftarrow \hat{w}_\alpha \cdot \hat{y}_j)$:

1. Produce shares $\langle w_\alpha y_j \rangle$: (using $[\![w_\alpha]\!]_{c_\alpha}$ and $\langle y_j \rangle, \langle c_\alpha y_j \rangle$)

```
1: Compute the pairing $\langle\!\langle w_\alpha y_j \rangle\!\rangle = \mathsf{MultShares}^{\mathsf{SLV}}([\![w_\alpha]\!]_{c_\alpha}, \langle y_j \rangle, \langle c_\alpha y_j \rangle)$, as below.
2: Let $(\langle w_\alpha y_j \rangle, \mathsf{BadValues}_k) = \mathsf{ConvertShares}^{\mathsf{SLV}}(b, \langle\!\langle w_\alpha y_j \rangle\!\rangle, (\mathsf{id}, 0), \delta', M, \phi)$.
3: Set $\langle y_k \rangle \leftarrow \langle w_\alpha y_j \rangle$ and $\mathsf{LeakageInfo} \leftarrow \mathsf{LeakageInfo} \cup \{(k, \mathsf{BadValues}_k)\}$
```

2. Produce shares $\langle c_\gamma w_\alpha y_j \rangle$ for each $\gamma \in [m]$: (using $[\![\hat{c}_\gamma^{(t)} w_\alpha]\!]_{c_\alpha + c_\gamma}$ and $\langle y_j \rangle, \langle c_\alpha y_j \rangle, \langle c_\gamma y_j \rangle$)

```
1: for $\gamma = 1$ to $m$ do
2:     if $\alpha = \gamma$ then define $c_{\alpha,\alpha} := c_\alpha$. Let $\langle c_{\alpha,\alpha} y_j \rangle = \langle c_\alpha y_j \rangle$.
3:     else define $c_{\alpha,\gamma} := c_\alpha + c_\gamma$. Compute $\langle c_{\alpha,\gamma} y_j \rangle = \langle c_\alpha y_j \rangle + \langle c_\gamma y_j \rangle$.
4:     end if
5:     for $t = 1$ to $\ell$ do
6:         Compute $\langle\!\langle \hat{c}_\gamma^{(t)} w_\alpha y_j \rangle\!\rangle = \mathsf{MultShares}^{\mathsf{SLV}}([\![\hat{c}_\gamma^{(t)} w_\alpha]\!]_{c_{\alpha,\gamma}}, \langle y_j \rangle, \langle c_{\alpha,\gamma} y_j \rangle)$.
7:         Evaluate $(\langle \hat{c}_\gamma^{(t)} w_\alpha y_j \rangle, \mathsf{BadValues}_{k,\gamma,t}) =$
8:             $\mathsf{ConvertShares}^{\mathsf{SLV}}(b, \langle\!\langle \hat{c}_\gamma^{(t)} w_\alpha y_j \rangle\!\rangle, (\mathsf{id}, t), \delta', M, \phi)$.
9:     end for
10:    Compute $\langle c_\gamma w_\alpha y_j \rangle = \sum_{t=1}^{\ell} 2^{t-1} \langle \hat{c}_\gamma^{(t)} w_\alpha y_j \rangle$.
11:    Set $\langle c_\gamma y_k \rangle \leftarrow \langle c_\gamma w_\alpha y_j \rangle$.
12:    $\mathsf{LeakageInfo} \leftarrow \mathsf{LeakageInfo} \cup \left( \bigcup_{\gamma \in [m], t \in [\ell]} \{((k, \gamma, t), \mathsf{BadValues}_{k,\gamma,t})\} \right)$
13: end for
```

Instruction $(\mathsf{id}, \beta, \hat{O}_j \leftarrow \hat{y}_i)$:

```
1: Compute $\langle O_j \rangle$ as in Instruction $(\mathsf{id}, \beta, \hat{O}_j \leftarrow \hat{y}_i)$ of Figure 4.
2: If $\mathsf{LeakageInfo} = \emptyset$, Let $\mathsf{SimOutput} \leftarrow \top$.
3: Else, let $\mathsf{SimOutput}$ be predicate $\mathsf{Pred} = \bigvee (y_k \in \mathsf{BadValues}_k) \vee \bigvee (\hat{c}_\gamma^{(t)} y_k \in \mathsf{BadValues}_{k,\gamma,t})$
   corresponding to the pairs in $\mathsf{LeakageInfo}$.
4: Output $(\langle O_j \rangle, \mathsf{SimOutput})$.
```

Procedure $\mathsf{MultShares}^{\mathsf{SLV}}\big([\![x]\!]_c, \langle y \rangle, \langle cy \rangle\big) \to \langle\!\langle xy \rangle\!\rangle$:

1. If $\langle y \rangle = \perp$ or $\langle cy \rangle = \perp$, return $\perp$. Else, return $\mathsf{MultShares}\big([\![x]\!]_c, \langle y \rangle, \langle cy \rangle\big)$ as in Figure 2.

Procedure $\mathsf{ConvertShares}^{\mathsf{SLV}}(b, \langle\!\langle x \rangle\!\rangle, \mathsf{id}, \delta, M) \to (\langle x \rangle, \mathsf{BadValues})$:

1. If $\langle\!\langle x \rangle\!\rangle = \perp$, then return $\perp$.
2. Denote by $\phi' : \mathbb{G} \to \{0,1\}^{\lceil \log(2M/\delta) \rceil}$ the appropriate prefix output of $\phi(\mathsf{id}, \cdot)$.
3. Let x_b denote the present party b's share of $\langle\!\langle x \rangle\!\rangle$.
4. Return $(i_b, \mathsf{BadValues}) \leftarrow \mathsf{SLVDistribDLog}_{\mathbb{G},g}(x_b, \delta, M, \phi')$.

Figure 5: Simulatable (one-round, client-server) LV-HSS evaluation algorithm Eval, with corresponding sub-routines. $\phi : \{0,1\}^\lambda \times \mathbb{G} \to \{0,1\}^\ell$ denotes a given PRF.

Min-Based Share Conversion. $\mathsf{DistributedDLog}'_{\mathbb{G},g}(h, \delta, M, \phi)$

```
1: Set $h' \leftarrow h$, $i \leftarrow 0$, $\mathsf{min} = \infty$. Let $T := \lceil 2M/\delta \rceil + M$.
2: while $i < T$ do
3:     $y \leftarrow \phi(h')$
4:     if $y < \mathsf{min}$ then $i_{\mathsf{min}} \leftarrow i$, $\mathsf{min} \leftarrow y$
5:     end if
6:     $h' \leftarrow h' \cdot g$, $i \leftarrow i + 1$
7: end while
8: Output $i_{\mathsf{min}}$.
```

Figure 6: Alternative Min-based share conversion algorithm. The function ϕ can either be a PRF or a min-wise independent hash function.

BGI Homomorphic Secret Sharing Scheme - $\mathsf{Share}(1^\lambda, w)$

Let $\mathsf{AdditiveShare}(x)$ for $x \in \mathbb{Z}_q$ return a random pair $\langle x \rangle = (x_0, x_1) \in \mathbb{Z}_q^2$ subject to $x_0 - x_1 = x$. (For $x \in \mathbb{Z}$, this is done for $x \pmod q \in \mathbb{Z}_q$).

Inputs: 1^λ and input $w = w_1, \ldots, w_n \in \mathbb{Z}$

- Sample a DDH-hard group $(\mathbb{G}, g) \leftarrow \mathcal{IG}(1^\lambda)$.
- Sample a PRF $\phi \leftarrow \mathsf{PRFGen}(1^\lambda)$ with input $\{0,1\}^\lambda \times \mathbb{G}$ and output $\{0,1\}^\ell$.
- Sample an ElGamal secret key: $c \leftarrow \mathbb{Z}_q$, where $q = |\mathbb{G}|$.
- For each input w_i, sample the following values:
  1. ElGamal encryptions:
     (a) of $w_i \in \mathbb{Z}$: let $[\![w_i]\!]_c \leftarrow \mathsf{Enc}_{\mathsf{ElGamal}}(g^c, w_i) \in \mathbb{G}^2$. $//g^c$ is ElGamal public key of c
     (b) of $(c^{(t)} w_i) \in \mathbb{Z}$: i.e., for each $t \in [\ell]$, let $[\![c^{(t)} w_i]\!]_c \leftarrow \mathsf{Enc}_{\mathsf{ElGamal}}(g^c, c^{(t)} w_i)$.
  2. Subtractive secret sharings:
     (a) of $w_i \in \mathbb{Z}$: let $\langle w_i \rangle \leftarrow \mathsf{AdditiveShare}(w_i)$.
     (b) of $cw_i \in \mathbb{Z}_q$: let $\langle cw_i \rangle \leftarrow \mathsf{AdditiveShare}(cw_i)$.
- For each $b \in \{0,1\}$, output $\mathsf{share}_b = \left\{ \phi, \left([\![w_i]\!]_c, \left\{ [\![c^{(t)} w_i]\!]_c \right\}_{t \in [\ell]}, \langle w_i \rangle_b, \langle cw_i \rangle_b \right)_{i \in [n]} \right\}$.

Figure 7: Share generation procedure Share for secret sharing an input w via the homomorphic secret sharing scheme. For simplicity we describe the scheme with *subtractive* secret sharing instead of additive as in BGI (see Remark A.1).

BGI Homomorphic Share Evaluation of RMS Programs - $\mathsf{Eval}_{\mathbb{G},g}(b, \mathsf{share}, P, \delta)$

Inputs: Party identifier $b \in \{0,1\}$, homomorphic secret share value share, RMS program description P of size $\leq S$, error bound δ.

Parse share as in Figure 7. Parse P as in Definition 2.1, as a magnitude bound 1^M and sequence of instructions. Take $\delta' = \delta/((\ell+1)MS)$.

For each sequential instruction in P, perform the corresponding sequence of operations:

Instruction $(\mathsf{id}, \hat{y}_j \leftarrow \hat{w}_i)$:

```
1: Let $\langle y_j \rangle \leftarrow \langle w_i \rangle$ and $\langle cy_j \rangle \leftarrow \langle cw_i \rangle$, where $\langle w_i \rangle, \langle cw_i \rangle$ are as in $\mathsf{share}$.
```

Instruction $(\mathsf{id}, \hat{y}_k \leftarrow \hat{y}_i + \hat{y}_j)$:

```
1: Compute $\langle y_k \rangle \leftarrow \langle y_i \rangle + \langle y_j \rangle$, directly on the subtractive shares (over $\mathbb{Z}_q$).
2: Compute $\langle cy_k \rangle \leftarrow \langle cy_i \rangle + \langle cy_j \rangle$, directly on the subtractive shares (over $\mathbb{Z}_q$).
```

Instruction $(\mathsf{id}, \hat{y}_k \leftarrow \hat{w}_i \cdot \hat{y}_j)$:

```
1: Let $[\![w_i]\!]_c$ and $\{[\![c^{(t)}w_i]\!]_c\}_{t\in[\ell]}$ be the ElGamal ciphertexts associated with $w_i$, and let $\langle y_j \rangle$
   and $\langle cy_j \rangle$ the subtractive secret shares associated with $y_j$.
2: Compute the pairing $\langle\!\langle w_i y_j \rangle\!\rangle = \mathsf{MultShares}([\![w_i]\!]_c, \langle y_j \rangle, \langle cy_j \rangle)$, as in Figure 2.
3: Execute Share Conversion: $\langle w_i y_j \rangle = \mathsf{ConvertShares}(b, \langle\!\langle w_i y_j \rangle\!\rangle, (\mathsf{id}, 0), \delta', M, \phi)$, as in Figure 2.
4: for $t = 1$ to $\ell$ do    // Repeat above process for each $c^{(t)}w_i$ in the place of $w_i$
5:     Compute $\langle\!\langle c^{(t)} w_i y_j \rangle\!\rangle = \mathsf{MultShares}([\![c^{(t)}w_i]\!]_c, \langle y_j \rangle, \langle cy_j \rangle)$.
6:     Execute $\langle c^{(t)} w_i y_j \rangle = \mathsf{ConvertShares}(b, \langle\!\langle c^{(t)} w_i y_j \rangle\!\rangle, (\mathsf{id}, t), \delta', M, \phi)$.
7: end for
8: Compute $\langle cw_i y_j \rangle = \sum_{t=1}^{\ell} 2^{t-1} \langle c^{(t)} w_i y_j \rangle$.
9: Set $\langle y_k \rangle \leftarrow \langle w_i y_j \rangle$ (from Step 3) and $\langle cy_k \rangle \leftarrow \langle cw_i y_j \rangle$.
```

Instruction $(\mathsf{id}, \beta, \hat{O}_j \leftarrow \hat{y}_i)$:

```
1: Shift $\langle z \rangle$ share by rerandomization offset: $\langle z \rangle \leftarrow \langle z \rangle + \phi(\mathsf{id}, g)$, over $\mathbb{Z}_q$.
   // Note that shifting both shares does not change the shared value in $\mathbb{Z}_q$
2: Convert share from $\mathbb{Z}_q$ to $\mathbb{Z}_\beta$: $\langle O_j \rangle \leftarrow \langle z \rangle \mod \beta$.
3: Output $\langle O_j \rangle$.
```

Figure 8: Procedures for performing homomorphic operations on secret shares. For simplicity we describe the scheme with *subtractive* and *division* secret sharing (see Remark A.1). Note that we distinguish variables of the straight-line program from the actual values by using $\hat{y}_i$ as opposed to y_i, etc. Here, notation $\langle y \rangle$ is used to represent *this party's* share of the corresponding subtractive secret sharing. Evaluation maintains the invariant that each of the *subtractive secret shares* $\langle y_i \rangle$ encode the correct current computation value of $\hat{y}_i$.

BGI Distributed-Evaluation Homomorphic Encryption (DEHE): $\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval}$.

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

1. Sample a PRF $\phi \leftarrow \mathsf{PRFGen}(1^\lambda)$ with input $\{0,1\}^\lambda \times \mathbb{G}$ and output $\{0,1\}^\ell$.
2. ElGamal key setup:
   (a) Sample a DDH-hard group and generator $(\mathbb{G}, g) \leftarrow \mathcal{IG}(1^\lambda)$.
   (b) Sample an ElGamal secret key: $c \leftarrow \mathbb{Z}_q$, where $q = |\mathbb{G}|$.
   Let $e = g^c$ be the corresponding ElGamal encryption key.
3. Sample ElGamal encryptions:
   (a) The constant $1 \in \mathbb{Z}_q$: let $[\![1]\!]_c \leftarrow \mathsf{Enc}_{\mathsf{ElGamal}}(e, 1)$.
   (b) The bits of the secret key c: $\forall t \in [\ell]$, let $[\![c^{(t)}]\!]_c \leftarrow \mathsf{Enc}_{\mathsf{ElGamal}}(e, c^{(t)})$.
4. Sample 2-out-of-2 additive secret sharings:
   (a) The constant $1 \in \mathbb{Z}_q$: $\langle 1 \rangle \leftarrow \mathsf{AdditiveShare}(1)$. // Included for notational convenience
   (b) The secret key c: let $\langle c \rangle \leftarrow \mathsf{AdditiveShare}(c)$.
5. Output $\mathsf{pk} = \big(\mathbb{G}, g, e, [\![1]\!]_c, \{[\![c^{(t)}]\!]_c\}_{t \in [\ell]}\big)$, $\mathsf{ek}_b = (\mathsf{pk}, \langle 1 \rangle, \langle c \rangle)$.

$\mathsf{Enc}_{\mathbb{G},g}(\mathsf{pk}, w)$:

1. Parse pk as in Gen above.
2. Compute the following ElGamal ciphertexts:
   (a) Of $w \in \mathbb{Z}$: let $[\![w]\!]_c \leftarrow \mathsf{Enc}_{\mathsf{ElGamal}}(e, w)$.
   (b) Of $c^{(t)}w \in \mathbb{Z}$: for each $t \in [\ell]$, parse $[\![c^{(t)}]\!]_c = (h_1^{(t)}, h_2^{(t)})$, sample a fresh encryption of 0 $(h'_1, h'_2) \leftarrow \mathsf{Enc}_{\mathsf{ElGamal}}(e, 0)$, and let $[\![c^{(t)}w]\!]_c = ((h_1^{(t)})^w \cdot h'_1, (h_2^{(t)})^w \cdot h'_2)$.
3. Output $([\![w]\!]_c, \{[\![c^{(t)}w]\!]_c\}_{t \in [\ell]})$.

$\mathsf{Eval}_{\mathbb{G},g}(b, \mathsf{ek}, \mathsf{ct}, P, \delta)$:

1. Parse ek as in Gen above; interpret $\hat{1}$ as loaded into memory, via $\langle 1 \rangle, \langle c \rangle$ as given.
2. Parse P as a sequence of instructions (as in Definition 2.1).
3. For each instruction $(\mathsf{id}, \hat{y}_k \leftarrow \hat{y}_i + \hat{y}_j)$, $(\mathsf{id}, \hat{y}_k \leftarrow \hat{w}_i \cdot \hat{y}_j)$, or $(\mathsf{id}, \hat{O}_j \leftarrow \hat{y}_i)$, perform the corresponding sequence of operations as given in Figure 8.
4. For each instruction $(\mathsf{id}, \hat{y}_j \leftarrow \hat{w}_i)$, execute $(\mathsf{id}, \hat{y}_j \leftarrow \hat{w}_i \cdot \hat{1})$.

Figure 9: BGI construction of homomorphic encryption with distributed evaluation (DEHE). For simplicity we describe the scheme with *subtractive* and *division* secret sharing (see Remark A.1).