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

Elette Boyle[1], Niv Gilboa[2], and Yuval Ishai[3]

[1] IDC Herzliya, elette.boyle@idc.ac.il
[2] Ben Gurion University, gilboan@bgu.ac.il
[3] Technion and UCLA, yuvali@cs.technion.ac.il

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, our protocol can generate 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.

1 Introduction

Gentry's 2009 breakthrough on fully homomorphic encryption (FHE) [36, 18] 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 [32, 2, 33, 14].[4]

On the downside, despite impressive recent progress [22, 15, 13], 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. [8] 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 [8] 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* [7], where the roles of the function and the input are reversed, or a weaker version of *additive-spooky encryption* [14].

The main result of [8] 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 2.2. The HSS scheme of [8] 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. This difficulty was addressed in [8] by combining error-correcting codes with general-purpose secure two-party computation protocols for recovering the correct output from the encoding. This

[4] 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 [21, 34, 24].

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 [8] 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.

1.1 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 [8] 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 [4]) 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.[5] Prior to our work, such protocols were only known using different flavors of FHE [2, 33, 14] or indistinguishability obfuscation [17, 14]. (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 [33, 14], or even eliminated by relying on indistinguishability obfuscation [14].)

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 [8]). The protocol can be used to succinctly evaluate branching programs. Alternatively, it can be used to evaluate general circuits

[5] 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.*

(at the cost of compromising succinctness) by applying the HSS evaluation to a low-complexity randomized encoding of the circuit [38, 3, 1].

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.

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.[6] 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 [10, 23] 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-

[6] 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.

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, we 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 [27] or the Phi-hiding assumption [28]. A similar result to ours can also be obtained under LWE, via the HSS scheme implied by [14].

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 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 [37].

Improving computation complexity. We present several techniques for reducing the computational cost of the share conversion procedure from [8], improving the concrete efficiency of group-based protocols (both in [8] 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 [8], 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 [11, 25].

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 [8]. Combining all the optimizations, one can perform thousands of RMS multiplications per second with error probability that is small enough for performing shallow computations.

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

We give some necessary definitions and provide a high-level overview of the BGI construction of [8]. We refer the reader to the full version for further details.

2.1 Homomorphic Secret Sharing and DEHE

As in [8], 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 1 (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.*

HSS versus DEHE. We also consider a public-key variant of HSS, known as *distributed-evaluation homomorphic evaluation* (DEHE) [8]. This variant is described and explored in the full version of this work.

Multi-evaluation variant. For our applications of Las Vegas HSS it will sometimes not be enough to consider a single execution of Eval but rather a sequence of such executions following a single execution of Share. In such a case, we will need to assume that the events of outputting $\perp$ in different executions are indistinguishable from being independent. (This will allow us to apply a Chenoff-style bound when analyzing the total number of errors.) To simplify the terminology and notation, we implicitly assume by default that all instances of HSS we use are of the multi-evaluation variant.

2.2 BGI Construction [8]

The work of [8] 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 full version for formal RMS syntax). We provide here a high-level description of the [8] 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 $\beta \leq q$ 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 1 (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 [4], 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 [8] for a detailed formal treatment.

2.3 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 [12, 19] 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, $\mathsf{NextMsg}(i, \boldsymbol{m})$ is a next message function which determines the messages sent by Server i in in the current round given the messages $\boldsymbol{m}$ it received in previous rounds, and $\mathsf{Decode}(i, \boldsymbol{m})$ denote the output of Client i given the messages $\boldsymbol{m}$ 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 [8] 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.[7] 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 [8], 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.

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 formal definition of client-server HSS is deferred to the full version. A "multi-evaluation" version enables independent executions of Eval without re-executing Π.

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

[7] 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.

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 $c^{(t)}$.

We present the full construction and proof of client-server HSS in the full version. In fact, we achieve the stronger primitive of multi-evaluation client-server DEHE, which directly implies the former.

Remark 2 (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 [8], by either considering a leveled variant or replacing ElGamal with a circular-secure variant due to BHHO [4], as described in Remark 1. Our theorem statements implicitly apply this transformation directly.

Proposition 1 (Black-box client-server HSS/DEHE). *There exists a multi-evaluation client-server DEHE protocol (and thus also multi-evaluation client-server HSS) for branching programs that makes a black-box access to any DDH-hard group.*

3.1 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 $\mathbb{G}$.

Theorem 1 (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.*

Remark 3 ($1/\mathsf{poly}$ 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 $1/\mathsf{poly}$ 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 1.

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) from $1/\mathsf{poly}$ 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 [8], as per Figure 1), 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

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. $\mathsf{DistributedDLog}_{\mathbb{G},g}(h, \delta, M, \phi)$

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

Fig. 1: Notation, pairing operations, and share conversion algorithm, as used in [8]. 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 discussion in full version.

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 $c = \mathsf{Recomp}((\hat{c}^{(t)})_{t \in [\ell']})$.[8]

The formal descriptions of $(\Pi_{\mathsf{1r}}, \mathsf{Eval}_{\mathsf{1r}})$ are given in Figures 2 and 3.

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

Lemma 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. We defer the proof to the full version. We remark that a crucial property for security is that any secret value owned by a client C_i is encrypted under a combination of keys that includes his *own* key, c_i (and distributed as a fresh encryption due to re-randomization). Because of this, semantic security holds for all honest-client values, by the key homomorphism properties of ElGamal.

Plugging in the client-server HSS $(\Pi_{\mathsf{1r}}, \mathsf{Eval}_{\mathsf{1r}})$ to the framework of Theorem 1, together with the round-savings-for-$1/\mathsf{poly}$ tradeoff described in Remark 3, we directly obtain the following proposition.

Proposition 2 (Succinct 2-server protocol with $1/\mathsf{poly}$ 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 $1/\mathsf{poly}$ 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.

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 $\mathsf{share}_b = \left(m, \phi, \left\{\langle c_i \rangle_b\right\}_{i\in[m]}, \left(\llbracket w_i \rrbracket_{c_i}, \{\llbracket \hat{c}_i^{(t)} w_i \rrbracket_{c_i}\}_{t\in[\ell']}, \{\llbracket \hat{c}_j^{(t)} w_i \rrbracket_{c_i+c_j}\}_{i\neq j\in[m], t\in[\ell']}\right)_{i\in[m]}\right)$.

Fig. 2: 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 2. Parse P as a magnitude bound 1^M and sequence of RMS instructions. Take $\delta' = \delta/((\ell'+1)MS)$. We describe here homomorphic evaluation of multiplication, and defer the other RMS operations to the full version.

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 1.
   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 1.
   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 ℓ' **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**

Fig. 3: 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.

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 $\boldsymbol{w}$ and randomness $\boldsymbol{r}$ such that the other party outputs $\perp$ if and only if $\mathsf{Pred}(\boldsymbol{w}, \boldsymbol{r}) = 1$. 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 2 (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 $(\mathsf{share}_0, \mathsf{share}_1) \leftarrow \Pi(\boldsymbol{w}; \boldsymbol{r}, R_0, R_1)$ where $\boldsymbol{w} = (w_1, \ldots, w_m)$ and $\boldsymbol{r} = (r_1, \ldots, r_m)$ 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 1 (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 $\boldsymbol{w}^\lambda = (w_1^\lambda, \ldots, w_m^\lambda) \in (\{0,1\}^{p(\lambda)})^m$, it holds that $\mathsf{view}(1^\lambda, \mathsf{Corrupt}, \boldsymbol{w}^\lambda) \stackrel{c}{\cong} \mathsf{Sim}(1^\lambda, \mathsf{Corrupt}, \{w_i\}_{C_i \in \mathsf{Corrupt}}, \{|w_i|\}_{C_i \notin \mathsf{Corrupt}})$.*
- **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 $b \in \{0,1\}$,*

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

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

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

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

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 $\mathsf{DistributedDLog}$ (called within $\mathsf{ConvertShares}$), 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 [8], the procedure $\mathsf{DistributedDLog}$ 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 $\mathsf{DistributedDLog}$ 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 $h, h \cdot g^z \in \mathbb{G}$, $\mathsf{DistributedDLog}$ yields an error in two cases:

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

Let $\mathsf{DangerZone} := \{h, hg^{(-1)^b}, \ldots, hg^{(-1)^b M}\}$.
Let $\mathsf{SimDangerZone} := \{hg^{-M+1}, \ldots, h, \ldots, hg^M\}$ and initialize $\mathsf{BadValues} \leftarrow \emptyset$.
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 $\mathsf{BadValues} \leftarrow \top$.
end if
if $\exists h' \in \mathsf{DangerZone}$ with $\phi(h') = 0^{\lceil \log(2M/\delta) \rceil}$ **then** Let $i = \perp$.
else
    Set $h' \leftarrow h$, $i \leftarrow 0$. Let $T = 2M\lambda/\delta$.
    while $(\phi(h') \neq 0^{\lceil \log(2M/\delta) \rceil}$ and $i < T)$ **do**
        $h' \leftarrow h' \cdot g$, $i \leftarrow i + 1$.
    end while
end if
Return $(i, \mathsf{BadValues})$.

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 [8], 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 DangerZone in SLVDistribDLog.
3. 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 DangerZone (and output $\perp$) if and only if he runs 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 the full version 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 $\boldsymbol{w}$. The choice of Pred incorporates the P dependency, and operates on input $\boldsymbol{w}$ as well as a subset of (at most λ values of) $\hat{c}^{(t)}$.

Proposition 3. *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 $\hat{c}^{(t)}$.*

Remark 4 (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 $\mathsf{SLVDistribDLog}$.

Secure 2-server Computation from Simulatable LV-HSS and Leakage Resilience We now combine the simulatable LV-HSS of Proposition 3, 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* $w_i^{(1)}, \ldots, w_i^{(k)}$ 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 [26]; 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 [38, 1].

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 [8]), 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 2 (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 3, 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 2.3). Then for any polynomial $m = m(\lambda)$, the protocol Π given in Construction 3 is a secure m-client 2-server protocol for general circuits that executes in* 2 rounds *in the PKI model.*

Construction 3 (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 3.*

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 defer the formal security proof to the full version and 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 3, 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 5 is an application of the above, obtained by using the virtual client-server protocol of [26] for evaluating low-degree polynomials. Our final result follows from generic transformations using low-degree randomized encodings [1].

Theorem 4 (MPC for low-degree polynomials [26]). *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 depth $O(\log(n + \ell + k))$.*

Theorem 5 (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.*

Corollary 1 (2-server protocol for circuits). *Assuming PKI setup and DDH, there is a (non-succinct) 2-round 2-server client-server protocol for circuits.*

Note that while this solution yields 2 rounds of communication, the amount of information communicated is greater than the program size. In the full version, we describe a more complex solution achieving *succinct* 2-round secure computation for the class of NC^1 programs.

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., [10, 23]). 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 6 (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 1) 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 [37] is a standard PRF family F_K equipped with a puncturing algorithm $\mathsf{Puncture}$ that given a set of points $X = \{x, \ldots, x_k\} \subseteq \{0,1\}^d$ 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 full version for a formal definition. As was shown in [5, 9, 29], the GGM construction [20] of PRFs from a length-doubling PRG can be used to obtain a puncturable PRF for $X = \{x_1, \ldots, x_k\} \subseteq \{0,1\}^d$ 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 [38, 31]) 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[9] for implementing Π, we get the following theorem.

Theorem 7 (Punctured OT via puncturable PRF). *Suppose a $\binom{2}{1}$-OT protocol exists. Then there is a protocol for $\binom{m}{m-k}$-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.*

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 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 [8] (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 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 $\binom{m'}{m'-k}$-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

[9] 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.

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 8 (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 2 (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 2 with the GMW protocol for secure circuit evaluation using bit-OT [21], we get the following corollary.

Corollary 3 (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 [8] (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 [8] 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 $2(\ell+1)$ executions of 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. The first and third of these optimizations (discussed in greater detail in the Introduction) are deferred to the full version of the paper.

6.1 Optimizing the Conversion

A straightforward implementation of the share conversion step in Figure 1 for a group element $h \in \mathbb{G}$ requires computing the sequence $h, hg, \ldots, hg^x$ 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 computing the sequence $h, hg, \ldots, hg^x$ in such groups is more efficient than in other DDH groups.

We explore an alternative implementation to 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 specific structure rather than over an EC group. In addition, this approach requires the distinguished point version of share conversion rather than the min-hash method (described in the full version).

The first idea is to decide if an element $hg^i \in \mathbb{G}$ is distinguished without using a PRF ϕ. We say that an element h' is distinguished if the representation of h' has $d = \lceil \log(1/\delta) \rceil$ leading zeroes, i.e $h' < 2^{\lceil \log p \rceil - d}$. We conjecture that if $h \in \mathbb{G}$ is chosen randomly then the sequence $h, \ldots, hg^x$ has a distinguished point with essentially the same probability as that of the sequence $\phi(h), \ldots, \phi(hg^x)$. Observe that h can be chosen randomly since the two servers can shift their respective elements h_0, h_1 by a shared random element r maintaining the difference between the elements.

The second idea is to consider pseudo-Mersenne primes, i.e. primes of the form $p = 2^k - \gamma$ for small γ, in which the element 2 generates a large sub-group. We refer to such primes as *conversion friendly*. In this setting, $2h \bmod p$ can be computed 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 $h, \ldots, hg^x$ involves little more than a comparison of the bit, an addition, and testing whether the d most significant bits of the result are zero.

Further savings 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 $0 \leq a_0 < 2^{n-w}, 0 \leq a_1 < 2^w$ 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 w elements $h, 2h \bmod p, \ldots, 2^{w-1}h \bmod p$ are distinguished by checking whether the most significant $2w$ bits of h include the substring 0^d. That can be done efficiently in standard computer architectures 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 them are then the sequence $h, 2h, \ldots, 2^{w-1}h$ does not contain a distinguished point and the next element to be examined is $2^w h$. An interesting property of the algorithm is that it is almost independent of the size of the underlying group.

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 for such

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.

conversion-friendly primes one can use include $2^{1280} - 7243217$, $2^{1536} - 11510609$ and $2^{2048} - 1942289$.

Assessing the security of DDH over these primes is difficult due to the scarcity of published attacks. Theoretically, the best attack against DDH over pseudo-Mersenne primes is using the Special Number Field Sieve (SNFS) [35] to compute discrete logarithms modulo the prime. The SNFS has been used to factor Mersenne numbers, with the current record being $2^{1199} - 1$ [30]. To account for the speedup offered by SNFS, the bit-length of such special primes 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 [16].

Table 1 presents the average number of basic operations required for one conversion step, i.e. computing $2h \bmod p$ from h and checking whether h is distinguished, and the number of conversion steps per second. The figures in the first row of the table are based on an implementation on a commodity laptop (Dell Latitude 3550, with Intel i7-5500 CPU, running single-threaded at 2.4 GHz and with 8 GByte of RAM) and can be significantly improved given a dedicated hardware and software platform. The implementation used 32-bit words together with multiplications of two 32 bit operands into a 64 bit product. The second row is a general analysis for an architecture with w bit words. The basic operations which are measured in the table are word-sized multiplication, addition and bit level operations (bit-by-bit AND operations and shifts).

The table makes it clear that the conversion step requires on average well below a single instruction, e.g. 0.25 instructions per step in the example in the the first row. In the alternative approach for computing a conversion step, each such step includes a group operation over an elliptic curve. Based on [6] Table 3, the fastest elliptic curve multiplication by a scalar for a relatively small, 254-bit, curve requires 196,000 machine instructions (on a somewhat stronger machine than what we used). A multiplication requires on average $254 \cdot (3/2)$ group operations, which means that each group operation, and each conversion, requires at least 2000 times the number of instructions of a conversion step implemented via conversion-friendly primes.

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

First author supported by ISF grant 1861/16, AFOSR Award FA9550-17-1-0069, and ERC starting grant 307952. Second author 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). Third author 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

1. B. Applebaum, Y. Ishai, and E. Kushilevitz. Computationally private randomizing polynomials and their applications. In *CCC*, pages 260–274, 2005.
2. G. Asharov, A. Jain, A. López-Alt, E. Tromer, V. Vaikuntanathan, and D. Wichs. Multiparty computation with low communication, computation and interaction via threshold FHE. In *EUROCRYPT*, pages 483–501, 2012.
3. D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols (extended abstract). In *STOC*, pages 503–513, 1990.
4. D. Boneh, S. Halevi, M. Hamburg, and R. Ostrovsky. Circular-secure encryption from decision diffie-hellman. In *CRYPTO 2008*, pages 108–125, 2008.
5. D. Boneh and B. Waters. Constrained pseudorandom functions and their applications. In *ASIACRYPT*, pages 280–300, 2013.
6. J. W. Bos, C. Costello, P. Longa, and M. Naehrig. Selecting elliptic curves for cryptography: an efficiency and security analysis. *Journal of Cryptographic Engineering*, 6(4):259–286, 2016.
7. E. Boyle, N. Gilboa, and Y. Ishai. Function secret sharing. In *Advances in Cryptology - EUROCRYPT*, pages 337–367, 2015.
8. E. Boyle, N. Gilboa, and Y. Ishai. Breaking the circuit size barrier for secure computation under DDH. In *CRYPTO*, pages 509–539, 2016. Full version: IACR Cryptology ePrint Archive 2016: 585 (2016).
9. E. Boyle, S. Goldwasser, and I. Ivan. Functional signatures and pseudorandom functions. In *PKC*, pages 501–519, 2014.
10. G. Bracha. An asynchronous $[(n-1)/3]$-resilient consensus protocol. In *PODC*, pages 154–162, 1984.
11. A. Z. Broder, M. Charikar, and M. Mitzenmacher. A derandomization using min-wise independent permutations. In *RANDOM*, pages 15–24, 1998.
12. R. Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, pages 143–202, 2000.
13. I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *ASIACRYPT*, pages 3–33, 2016.

14. Y. Dodis, S. Halevi, R. D. Rothblum, and D. Wichs. Spooky encryption and its applications. In *CRYPTO*, pages 93–122, 2016.
15. L. Ducas and D. Micciancio. FHEW: bootstrapping homomorphic encryption in less than a second. In *EUROCRYPT*, pages 617–640, 2015.
16. J. Fried, P. Gaudry, N. Heninger, and E. Thomé. A kilobit hidden SNFS discrete logarithm computation. *IACR Cryptology ePrint Archive*, 2016:961, 2016.
17. S. Garg, C. Gentry, S. Halevi, and M. Raykova. Two-round secure MPC from indistinguishability obfuscation. In *TCC*, pages 74–94, 2014.
18. C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178, 2009.
19. O. Goldreich. *Foundations of Cryptography — Basic Applications*. Cambridge University Press, 2004.
20. O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *J. ACM*, 33(4):792–807, 1986.
21. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *STOC*, pages 218–229, 1987.
22. S. Halevi and V. Shoup. Bootstrapping for HElib. In *EUROCRYPT*, pages 641–670, 2015.
23. M. Hirt and U. M. Maurer. Player simulation and general adversary structures in perfect multiparty computation. *J. Cryptology*, 13(1):31–60, 2000.
24. O. Horvitz and J. Katz. Universally-composable two-party computation in two rounds. In *CRYPTO*, pages 111–129, 2007.
25. P. Indyk. A small approximately min-wise independent family of hash functions. *J. Algorithms*, 38(1):84–90, 2001.
26. Y. Ishai and E. Kushilevitz. Randomizing polynomials: A new representation with applications to round-efficient secure computation. In *FOCS*, pages 294–304, 2000.
27. Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. Cryptography with constant computational overhead. In *STOC*, pages 433–442, 2008.
28. Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. Zero-knowledge proofs from secure multiparty computation. *SIAM J. Comput.*, 39(3):1121–1152, 2009.
29. A. Kiayias, S. Papadopoulos, N. Triandopoulos, and T. Zacharias. Delegatable pseudorandom functions and applications. In *CCS*, pages 669–684, 2013.
30. T. Kleinjung, J. W. Bos, and A. K. Lenstra. Mersenne factorization factory. In *ASIACRYPT*, pages 358–377, 2014.
31. Y. Lindell and B. Pinkas. A proof of security of Yao's protocol for two-party computation. *J. Cryptology*, 22(2):161–188, 2009.
32. A. López-Alt, E. Tromer, and V. Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *STOC*, pages 1219–1234, 2012.
33. P. Mukherjee and D. Wichs. Two round multiparty computation via multi-key FHE. In *EUROCRYPT*, pages 735–763, 2016.
34. M. Naor and K. Nissim. Communication preserving protocols for secure function evaluation. In *STOC*, pages 590–599, 2001.
35. J. Pollard. Factoring with cubic integers. Unpublished Manuscript, 1988.
36. R. L. Rivest, L. Adleman, and M. L. Dertouzos. On data banks and privacy homomorphisms. In *Foundations of secure computation*, pages 169–179. 1978.
37. A. Sahai and B. Waters. How to use indistinguishability obfuscation: deniable encryption, and more. In *STOC*, pages 475–484, 2014.
38. A. C.-C. Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, pages 162–167, 1986.