Breaking the Circuit Size Barrier for Secure Computation Under DDH

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. Under the Decisional Diffie-Hellman (DDH) assumption, we present a 2-out-of-2 secret sharing scheme that supports a compact evaluation of branching programs on the shares. More concretely, there is an evaluation algorithm Eval with a single bit of output, such that if an input $w \in \{0,1\}^n$ is shared into (w^0, w^1), then for any deterministic branching program P of size S we have that $\mathsf{Eval}(P, w^0) \oplus \mathsf{Eval}(P, w^1) = P(w)$ except with at most δ failure probability. The running time of the sharing algorithm is polynomial in n and the security parameter λ, and that of Eval is polynomial in S, λ, and $1/\delta$. This applies as a special case to boolean formulas of size S or boolean circuits of depth $\log S$. We also present a public-key variant that enables homomorphic computation on inputs contributed by multiple clients.

The above result implies the following DDH-based applications:

- A secure 2-party computation protocol for evaluating any branching program of size S, where the communication complexity is linear in the input size and only the running time grows with S.
- A secure 2-party computation protocol for evaluating any layered boolean circuit of size S and m outputs with communication complexity $O(S/\log S) + m \cdot \mathsf{poly}(\lambda)$.
- A 2-party *function secret sharing* scheme, as defined by Boyle et al. (Eurocrypt 2015), for general branching programs (with inverse polynomial error probability).
- A 1-round 2-server *private information retrieval* scheme supporting general searches expressed by branching programs.

Prior to our work, similar results could only be achieved using fully homomorphic encryption. We hope that our approach will lead to more practical alternatives to known fully homomorphic encryption schemes in the context of low-communication secure computation.

1 Introduction

In this paper we introduce a simple new technique for low-communication secure computation that can be based on the Decisional Diffie-Hellman (DDH) assumption and avoids the use of fully homomorphic encryption. We start with some relevant background.

Since the seminal feasibility results of the 1980s [36, 22, 3, 8], a major challenge in the area of secure computation has been to break the "circuit size barrier." This barrier refers to the fact that all classical techniques for secure computation required a larger amount of communication than the size of a boolean circuit representing the function to be computed, even when the circuit is much bigger than the inputs. The circuit size barrier applied not only to general circuits, but also to useful restricted classes of circuits such as boolean formulas (namely, circuits with fan-out 1) or branching programs (a stronger computational model capturing non-uniform logarithmic-space computations). Moreover, the same barrier applied also to secure computation protocols that can rely on a trusted source of correlated randomness, provided that this correlated randomness needs to be *reusable*.

The first significant progress has been made in the context of *private information retrieval* (PIR), where it was shown that for the bit-selection function $f(x, i) = x_i$ it is possible to break the circuit size barrier either in the multi-server model [11, 9], where a client holds i and two or more servers hold x, or in the two-party model [26] under standard cryptographic assumptions. However, progress on extending this to other useful computations has been slow, with several partial results [10, 16, 29, 4, 24] that do not even cover very simple types of circuits such as general DNF or CNF formulas, let alone more expressive ones such as general formulas or branching programs.[4]

All this has changed with Gentry's breakthrough on fully homomorphic encryption (FHE) [32, 18]. FHE enables local computations on encrypted inputs, thus providing a general-purpose solution to the problem of low-communication secure computation. On the down side, even the best known implementations of FHE [23] are still quite slow. Moreover, while there has been significant progress on basing the feasibility of FHE on more standard or different assumptions [34, 7, 20], the set of cryptographic assumptions on which FHE can be based is still very narrow, and in particular it does not include any of the "traditional" assumptions that were known in the 20th century.

1.1 Our Contribution

Our new approach was inspired by the recent work on *function secret sharing* (FSS) [6]. A (2-party) FSS scheme for a function class $\mathcal{F}$ allows a client to split (a representation of) $f \in \mathcal{F}$ into succinctly described functions f_0 and f_1 such that for any input x we have that $f(x) = f_0(x) + f_1(x)$ (over some Abelian group), but each f_b hides f.

The notion of FSS was originally motivated by applications to m-server PIR and related problems. FSS schemes for simple classes of functions such as point functions were constructed from one-way functions in [21,

[4] In the homomorphic encryption for branching programs from [24] (see also [25]), the size of the encrypted output must grow with the *length* of the branching program. When simulating a boolean formula by a branching program, the length of the branching program is typically comparable to the formula size.

6]. However, a result from [6] shows that 2-party FSS for richer circuit classes, from AC^0 and beyond, would imply (together with a mild additional assumption) breaking the circuit size barrier for similar classes.
The idea is that by encrypting the inputs and applying FSS to the function f' that first decrypts the inputs and then computes f, the parties can shift the bulk of the work required for securely evaluating f to local evaluations of f'_0 and f'_1. Thus, breaking the circuit size barrier reduces to securely distributing the generation of f'_0 and f'_1 from f and the secret decryption keys, which can be done using standard secure computation protocols and reused for an arbitrary number of future computations. This was viewed in [6] as a negative result, providing evidence against the likelihood of basing powerful forms of FSS on assumptions that are not known to imply FHE.
We turn the tables by constructing FSS schemes for branching programs under DDH, which implies low-communication secure 2-party computation protocols under DDH.

Homomorphic secret sharing. For the purpose of presenting our results, it is more convenient to consider a dual version of FSS that can also be viewed as a form of "homomorphic secret sharing," or alternatively a variant of threshold FHE [18, 1]. Concretely, a client wants to split a secret input $w \in \{0,1\}^n$ into a pair of shares (w^0, w^1), each of which is sent to a different server. Any strict subset of the shares should computationally hide w. Each server, holding (a representation of) a function $f \in \mathcal{F}$, can apply an evaluation algorithm to compute $y_b = \mathsf{Eval}(f, w^b)$, so that $y_0 + y_1 = f(w)$. Note that this is precisely the original notion of FSS with the roles of the function and input reversed.[5]
Cast in the this language, our main technical contribution is such a homomorphic secret sharing scheme, based on DDH, with output group $\mathbb{Z}_2$ (or any other $\mathbb{Z}_p$), and the class $\mathcal{F}$ of functions represented by deterministic[6] branching programs. The scheme only satisfies a relaxed form of the above correctness requirement: for every input w and branching program P, the probability of producing local outputs that add up to the correct output $P(w)$ is upper bounded by a parameter $\delta > 0$ which affects the running time of Eval. This probability is over the randomness of the sharing. The running time of the sharing algorithm is $n \cdot \mathsf{poly}(\lambda)$, where λ is a security parameter. The running time of Eval is polynomial in S, λ, and $1/\delta$.
We would like to stress that branching programs are quite powerful and capture many useful real-life computations. In particular, a branching program of size S can simulate any boolean formula of size S or boolean circuit of depth $\log_2 S$, and polynomial-size branching programs can sim-

[5] While one can always switch between the notions by changing the definition of $\mathcal{F}$, for classes $\mathcal{F}$ that contain *universal functions* [**?**,**?**] the switch can be done with polynomial overhead without changing $\mathcal{F}$. This will be the case for all function classes considered in this work.

[6] In fact, our construction can handle a larger class of *arithmetic* branching programs over the integers, but correctness only holds as long as all integers involved in intermediate computations are bounded by some fixed polynomial.

ulate any computation in the complexity classes NC^1 or (non-uniform) deterministic log-space.
We also present a public key variant of the homomorphic secret sharing scheme. This variant can be viewed as a threshold homomorphic encryption scheme with secret evaluation keys and additive reconstruction. That is, there is a key generation algorithm that outputs a single public key and a pair of secret evaluation keys. Given the public key, an arbitrary number of clients can encrypt their inputs. Each server, given the public ciphertexts and its secret evaluation key, can locally compute an additive share of the output.
The above results imply the following applications, all based on the DDH assumption alone.

Succinct secure computation of branching programs. The general transformation from FSS to secure two-party computation described above can be used to obtain succinct two-party protocols for securely evaluating branching programs with reusable preprocessing. However, the public-key variant of our construction implies simpler and more efficient protocols. The high level approach is similar to that of other low-communication secure protocols from different flavors of FHE [18, 1, 28], except for requiring secret homomorphic evaluation keys and an additional error-correction sub-protocol. For a two-party functionality with a total of n input bits and $m = m(n)$ output bits, where each output can be computed by a polynomial-size branching program (alternatively, logarithmic space Turing Machine or NC^1 circuit), the protocol can be implemented with a constant number of rounds and $n + m \cdot \mathsf{poly}(\lambda)$ bits of communication, where λ is a security parameter. To reduce the $n \cdot \mathsf{poly}(\lambda)$ cost of a bit-by-bit encryption of the inputs, the protocol employs a hybrid homomorphic encryption technique from [19].

Breaking the circuit size barrier for "well structured" circuits. In the case of evaluating a layered circuit C, we can make the total communication slightly sublinear in the circuit size by partitioning the computation into segments of logarithmic depth and homomorphically computing additive shares of the outputs of each segment given additive shares of the inputs. Concretely, to evaluate a layered circuit of size S with m output bits, the protocol requires $O(S/\log S) + m \cdot \mathsf{poly}(\lambda)$ bits of communication. We employ error-correcting codes with encoding and decoding in NC^1 to ensure that errors introduced by the computation of a segment are corrected before propagating to the next segment.

Function secret sharing. Using a universal branching program we can reverse the roles of P and w in the above homomorphic secret sharing scheme, obtaining a polynomial-time 2-party FSS scheme for branching programs. Unlike the main definition of FSS from [6] here we can only satisfy a relaxed notion that allows an inverse polynomial error probability. However, the error probability can be made negligible in the context of natural applications. An m-party FSS scheme for *circuits* was recently obtained by Dodis et al. [14] under the Learning with Errors (LWE) assumption, by making use of multi-key FHE [27, 12, 28]. Our construction gives the first FSS scheme that applies to a rich class of functions and does not rely on FHE.

Private information retrieval. Following the application of FSS to PIR from [6] with a simple repetition-based error-correction procedure, a consequence of the above result is a 1-round 2-server (computational) PIR scheme in which a client can privately search a database consisting of N documents for the existence of a document satisfying a predicate P, where P is expressed as a branching program applied to the document. For instance, any deterministic finite automaton can be succinctly expressed by such a branching program. The length of the query sent to each server is polynomial in the size of the branching program and a computational security parameter, whereas the length of the answer is a statistical security parameter times $\log N$.

1.2 Overview of Techniques

We now describe the main ideas behind our construction. It will be convenient to use the homomorphic secret sharing view: a client would like to share an input w between 2 servers so that the servers, on input P, can locally compute additive shares of $P(w)$.

Let $\mathbb{G}$ be a DDH group of prime order q with generator g. Our construction employs three simple ideas. The first is that a combination of a threshold version of ElGamal and linear secret sharing allows the servers to locally multiply an encrypted input x with a linearly secret-shared value y such that the result $z = xy$ is shared multiplicatively between the servers; namely the servers end up with elements $z_i \in \mathbb{G}$ such that the product of the z_i is g^z. This idea alone is already useful, as it gives an $(m-1)$-private m-server protocol for computing any degree-2 polynomial P with small integer coefficients held by the servers on a vector w of small integers held by the client, where the communication complexity in each direction is essentially optimal.

What seems to stop us at degree-2 polynomials is the fact that z is now shared *multiplicatively* rather than linearly, so the servers cannot multiply z by a new input encrypted by the client. Moreover, converting multiplicative shares to additive shares seems impossible without the help of the client, due to the intractability of computing discrete logarithms in $\mathbb{G}$. The second, and most surprising, idea is that if we allow for an inverse polynomial error probability, and *assuming there are only* $m = 2$ *servers*, the servers can convert multiplicative shares of g^z into linear shares of z without any interaction. For simplicity, suppose $z \in \{0, 1\}$. Taking the inverse of the second server's share, the servers now hold group elements g_0, g_1 such that $g_0 = g_1$ if $z = 0$ and $g_1 = g \cdot g_0$ if $z = 1$. Viewing the action of multiplication by g as a cycle over $\mathbb{Z}_q$, the elements g_0, g_1 are either in identical positions, or g_1 is one step ahead. Conversion is done by picking a pseudo-random δ-sparse[7] subset $\mathbb{G}'$ of $\mathbb{G}$ and having each server $b \in \{0, 1\}$ locally find the minimal integer $z_b \geq 0$ such that $g_b \cdot g^{z_b} \in \mathbb{G}'$. The first such z_b is expected to be found in roughly $1/\delta$

[7] Ideally, such a sparse subset would include each $g \in \mathbb{G}$ independently with probability δ. To emulate this efficiently we include each $g \in \mathbb{G}$ in $\mathbb{G}'$ if $\phi(g) = 0^{\lceil \log 1/\delta \rceil}$, where ϕ is a pseudorandom function.

steps and if it is not found in $(1/\delta)\log(1/\delta)$ steps, we set $z_b = 0$. The key observation is that except with $O(\delta)$ probability, both searches will find the same point in $\mathbb{G}'$ and the servers will end up with integers z_1, z_2 such that $z_1 - z_2 = z$, yielding the desired linear sharing of z.

Once we have a linear sharing of z, we can freely add it with other values that have a similar linear representation. We cannot hope to multiply two linearly shared values, but only to multiply them with another encrypted input. However, in order to perform such a multiplication, we need to convert z from the canonical "subtractive" representation to a different linear representation which is compatible with the ElGamal secret key.

The third idea is that since the coefficients of both linear representations are known to the client, the client can assist the conversion by providing an encryption of each input w multiplied by the secret key. This introduces two problems: the first is that semantic security may break down given a circular encryption of the secret key, which we handle either by assuming circular-security of ElGamal or (with some loss of efficiency) by using the circular-secure variant of Boneh et al. [5] instead of standard ElGamal. A more basic problem is that for the conversion to produce correct results with high probability, the secrets must be small integers. This is handled by providing an encryption of each input x multiplied by each *bit* of the secret key, and applying a linear combination whose coefficients are powers of 2 to the linear shares of the products of x and the bits of the key.

These ideas allow the servers to compute a restricted type of "straight-line programs" on the client's input. Such a program consists of a sequence of instructions, where each instruction can either load an input into memory, add the values of two memory locations, or multiply a memory location by an input. (Note that we cannot multiply two memory locations; this would allow evaluation of arbitrary circuits.) Such programs can emulate any branching program of size S by a sequence of $O(S)$ instructions.

It is instructive to note that the only limit on the number of instructions performed by the servers is the accumulation of error probabilities. This is analogous to the accumulation of noise in FHE schemes. However, the mechanisms for coping with errors are very different: in the context of known FHE schemes the simplest way of coping with noise is by using larger ciphertexts, whereas here we can reduce the error probability by simply increasing the *running time* of the servers, without affecting the ciphertext size or the complexity of encryption and decryption at all. We can also further trade running time for succinctness: the share size in our basic construction can be reduced by replacing the binary representation of the secret key with a representation over a larger basis, which leads to a higher homomorphic evaluation time.

The surprising power of local share conversions, initially studied in [13], has already been observed in the related contexts of information-theoretic PIR and locally decodable codes [37, 15, 2]. However, the type of share conversion employed here is very different in nature, as it is inherently tied to efficient computation rather than information.

Interestingly, our share conversion technique has resemblance to a cryptanalytic technique introduced by van Oorschot and Weiner for the purpose of parallel collision finding [35], where a set of "distinguished points" is used to synchronize two different processors.

Future Directions. This work gives rise to many natural open questions and future research directions. Can one bootstrap from branching programs to general circuits without relying on FHE? Can similar results be obtained for more than 2 parties? Can similar results be based on other assumptions that are not known to imply FHE? Can the dependence on the error parameter δ be eliminated or improved? To what extent can our protocols be optimized for practical use?
We hope that our approach will lead to faster solutions for some practical use-cases of FHE.

2 Preliminaries

In this section we define the main primitives we rely on and other necessary definitions.

Function representations. We capture a function representation (such as a circuit, formula, or branching program) by an infinite collection $\mathcal{P}$ of bit strings P (called "programs"), each specifying an input length n and an output length m, together with an efficient algorithm $\mathsf{Evaluate}$, such that $y \leftarrow \mathsf{Evaluate}(P, w)$ (denoted by shorthand notation "$P(w)$"), for any input $w \in \{0,1\}^n$, defines the output of P on w.

Homomorphic secret sharing. A (2-party) *Homomorphic Secret Sharing (HSS)* for a class of programs $\mathcal{P}$ consists of algorithms $(\mathsf{Share}, \mathsf{Eval})$, where $\mathsf{Share}(1^\lambda, (w_1, \ldots, w_n))$ splits the input w into a pair of shares $(\mathsf{share}_0, \mathsf{share}_1)$, and $\mathsf{Eval}(b, \mathsf{share}, P, \delta, \beta)$ homomorphically evaluates P on share, where the correct output is additively shared over $\mathbb{Z}_\beta$ except with error probability δ. When β is omitted it is understood to be $\beta = 2$. We allow Eval to run in time polynomial in its input length and in $1/\delta$ and require that each share_b output by Share keeps w semantically secure.

Public-Key Variant. We further consider a stronger variant of the homomorphic secret sharing primitive that supports homomorphic computations on inputs contributed by different clients. In fact, what we achieve is stronger: there is a single public key that can be used to encrypt inputs as in a standard public-key encryption scheme. However, similar to the original notion of homomorphic secret sharing (and in contrast to standard homomorphic encryption schemes), homomorphic computations on encrypted inputs are done in a distributed way and require two separate (secret) evaluation keys. As before, we require the reconstruction of the output to be additive.
The corresponding security notion guarantees "semantic"-style secrecy of an encrypted value, given only the evaluation key of a single server. In a setting consisting of two servers and an arbitrary number of clients, the above security notion implies that inputs contributed by a set of uncorrupted clients remain secure even if one of the two servers colludes with all the remaining clients.

Definition 1 (Distributed-Evaluation Homomorphic Encryption). *A (2-party)* Distributed-Evaluation Homomorphic Encryption (DEHE) *for a class of programs $\mathcal{P}$ consists of algorithms* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval})$ *with the following syntax:*

- $\mathsf{Gen}(1^\lambda)$*: On input a security parameter 1^λ, the key generation algorithm outputs a public key* pk *and a pair of evaluation keys* $(\mathsf{ek}_0, \mathsf{ek}_1)$.
- $\mathsf{Enc}(\mathsf{pk}, w)$*: On a public key* pk *and a secret input value $w \in \{0,1\}$, the encryption algorithm outputs a ciphertext* ct.
- $\mathsf{Eval}(b, \mathsf{ek}, (\mathsf{ct}_1, \ldots, \mathsf{ct}_n), P, \delta, \beta)$*: On input party index $b \in \{0,1\}$, an evaluation key* ek*, vector of n ciphertexts, a program $P \in \mathcal{P}$ with n input bits and m output bits, error bound $\delta > 0$, and an integer $\beta \geq 2$, the homomorphic evaluation algorithm outputs $y_b \in \mathbb{Z}_\beta^m$, constituting party b's share of an output $y \in \{0,1\}^m$. When β is omitted it is understood to be $\beta = 2$.*

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

- **Correctness:** *There exists a negligible function ν such that for every positive integer λ, input $(w_1, \ldots, w_n) \in \{0,1\}^n$, program $P \in \mathcal{P}$ with input length n, error bound $\delta > 0$, and integer $\beta \geq 2$,*

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

  where addition of y_0 and y_1 is carried out modulo β.
- **Security:** *The two distribution ensembles $C_0(\lambda)$ and $C_1(\lambda)$ are computationally indistinguishable, where $C_w(\lambda)$ is obtained by letting* $(\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \leftarrow \mathsf{Gen}(1^\lambda)$ *and outputting* $(\mathsf{pk}, \mathsf{ek}_b, \mathsf{Enc}(\mathsf{pk}, w))$.

2.1 Miscellaneous

Definition 2 (DDH and Circular Security). *Let $\mathcal{G} = \{\mathbb{G}_\rho\}$ be a set of finite cyclic groups, where $|\mathbb{G}_\rho| = q$ and ρ ranges over an infinite index set. We use multiplicative notation for the group operation and use $g \in \mathbb{G}_\rho$ to denote a generator of $\mathbb{G}_\rho$. Assume that there exists an algorithm running in polynomial time in $\log q$ that computes the group operation of $\mathbb{G}_\rho$. Assume further that there exists a PPT instance generator algorithm $\mathcal{IG}$ that on input 1^λ outputs an index ρ which determines the group $\mathbb{G}_\rho$ and a generator $g \in \mathbb{G}_\rho$. We say that the Decisional Diffie-Hellman assumption (DDH) is satisfied on $\mathcal{G}$ if $\mathcal{IG}(1^\lambda) = (\rho, g)$ and for every non-uniform PPT algorithm $\mathcal{A}$ and every three random exponents $a, b, c \in \{0, \ldots, q-1\}$ we have that*

$$|Pr[\mathcal{A}(\rho, g^a, g^b, g^{ab}) = 1] - Pr[\mathcal{A}(\rho, g^a, g^b, g^c) = 1]| < \varepsilon(\lambda),$$

for a negligible function ε. We will sometimes write $(\mathbb{G}, g, q) \leftarrow \mathcal{IG}(1^\lambda)$.

A more efficient variant of our construction will require a circular security assumption on the underlying bit encryption scheme, in which an efficient adversary cannot distinguish encryptions of the bits of the secret key from encryptions of 0.

Definition 3 (Circular Security). *We say that a public-key encryption scheme* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *with key length* $\ell(\lambda)$ *and message space containing* $\{0,1\}$ *is* circular secure *if there exists a negligible function* $\nu(\lambda)$ *for which the following holds for every non-uniform PPT* $\mathcal{A}$*:*

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

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

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

3 Homomorphic Secret Sharing for Branching Programs

In this section, we present constructions of homomorphic secret sharing schemes that enable non-interactive evaluation of a certain class of programs, known as restricted-multiplication straight-line programs. In particular, this class will include deterministic branching programs.

Definition 4 (RMS programs). *The class of* Restricted Multiplication Straight-line (RMS) *programs consists of an arbitrary sequence of the four following instructions, each with a unique identifier* id*:*
- *Load an input into memory:* $(\mathsf{id}, \hat{y}_j \leftarrow \hat{w}_i)$.
- *Add values in memory:* $(\mathsf{id}, \hat{y}_k \leftarrow \hat{y}_i + \hat{y}_j)$.
- *Multiply value in memory by an input value:* $(\mathsf{id}, \hat{y}_k \leftarrow \hat{w}_i \cdot \hat{y}_j)$.
- *Output value from memory, as element of* $\mathbb{Z}_\beta$*:* $(\mathsf{id}, \beta, \hat{O}_j \leftarrow \hat{y}_i)$.

Our construction will support homomorphic evaluation of straight-line programs of this form over inputs $w_i \in \mathbb{Z}$, provided that all intermediate computation values in $\mathbb{Z}$ remain "small" (where the required runtime grows with this size bound). Our final result is a public-key variant—i.e., a homomorphic encryption scheme with distributed evaluation (as per Definition 1)—based on DDH, with ciphertext size $O(\ell)$ group elements per input (for ℓ the logarithm of the DDH group size), and where runtime for homomorphic evaluation of an RMS program of size S with intermediate computation values bounded by M is $\mathsf{poly}(\lambda, S, M, 1/\delta)$.
An important sub-procedure of our homomorphic share evaluation algorithms is a local share conversion algorithm $\mathsf{DistributedDLog}$, which intuitively converts a multiplicative secret sharing of g^x to an *additive* secret sharing of the value x, with inverse polynomial probability of error.

In the following subsections, we present: (1) The share conversion procedure DistributedDLog, (2) a simplified version of the homomorphic secret sharing scheme (in the secret-key setting), assuming circular security of ElGamal encryption, (3) the analogous public-key construction, and (4) the final public-key construction based on standard DDH.

3.1 Share Conversion Procedure

We now describe the local share conversion algorithm DistributedDLog, which receives as input a group element $h \in \mathbb{G}$ and outputs an integer w. Loosely speaking, DistributedDLog outputs the distance on the cycle generated by $g \in \mathbb{G}$ between h and the first $z \in \mathbb{G}$ such that a pseudo-random function outputs 0 on z. DistributedDLog is a deterministic algorithm and consequently two invocations of the algorithm with the same element h result in the same output w. Two invocations of the algorithm on inputs h and $h \cdot g^{\mu}$ for a small μ result, with good probability, in outputs w and $w - \mu$. Therefore, the DistributedDLog algorithm converts a difference of small μ in the cycle generated by g in $\mathbb{G}$ to the same difference over the integers.
The detailed description of $\mathsf{DistributedDLog}_{\mathbb{G},g}$ follows. The algorithm is hard-wired with ρ defining a group $\mathbb{G} = \mathbb{G}_\rho$ and a generator $g \in \mathbb{G}$. $\mathsf{DistributedDLog}_{\mathbb{G},g}$ receives as input $h \in \mathbb{G}$, an allowable error probability δ, maximum difference $\mu \in \mathbb{N}$, and a pseudo-random function $\phi : \mathbb{G} \to \{0,1\}^{\log(4\mu/\delta)}$. The difference μ specifies the maximum distance from h, along the cycle that g generates, of an element that is the input to a parallel invocation of $\mathsf{DistributedDLog}_{\mathbb{G},g}$ within a given application.

Algorithm 1 $\mathsf{DistributedDLog}_{\mathbb{G},g}(h, \delta, \mu, \phi)$

```
1: Set $z \leftarrow h$, $w \leftarrow 0$.
2: while ($\phi(z) \neq 0^{\log(4\mu/\delta)}$ and $w < \frac{4\mu \ln(4/\delta)}{\delta}$) do
3:     $z \leftarrow z \cdot g$, $w \leftarrow w + 1$.
4: end while
5: Output $w$.
```

Proposition 1. *Let λ be a security parameter, let $\mathcal{G} = \{\mathbb{G}_\rho\}$ be a set of finite cyclic groups and let the instance generator algorithm of the set running on input 1^λ return a group $\mathbb{G} = \mathbb{G}_\rho$ with generator g. Let $\delta > 0$, let $\mu \in \mathbb{N}$, let $\mathcal{F}^r$ be a family of PRF defined over $\mathcal{G}$ and let $\phi : \mathbb{G} \to \{0,1\}^{\log(4/\delta)}$ be randomly chosen from all the members of $\mathcal{F}^r$ with domain $\mathbb{G}$. Then, for any $h \in \mathbb{G}$ and $\mu' \leq \mu$ we have that*

$$\mathsf{DistributedDLog}_{\mathbb{G},g}(h, \delta, \mu, \phi) - \mathsf{DistributedDLog}_{\mathbb{G},g}(h \cdot g^{\mu'}, \delta, \mu, \phi) = \mu'$$

with probability greater than $1 - \delta$.

Proof. The values of ϕ on the sequence of elements traversed by the variable z in an execution of $\mathsf{DistributedDLog}_{\mathbb{G},g}(h,\delta,\mu,\phi)$ can be divided into three cases. The first case is that $\phi(h\cdot g^c)=0^{\log(4\mu/\delta)}$ for some c in the range $0\le c\le\mu-1$, the second case is that $\phi(h\cdot g^c)\neq 0^{\log(4\mu/\delta)}$ for $c=0,\dots,\mu-1$, but $\phi(h\cdot g^c)=0^{\log(4\mu/\delta)}$ for some c in the range $\mu\le c\le\frac{4\mu\ln(4/\delta)}{\delta}$ and the last case is that neither of the former occurs, i.e. $\phi(h\cdot g^c)\neq 0^{\log(4\mu/\delta)}$ for every $c=0,1,\dots,\frac{4\mu\ln(4/\delta)}{\delta}$.
In the second case, since $\phi(h\cdot g^c)\neq 0^{\log(4\mu/\delta)}$ for any $c=0,\dots,\mu-1$, the execution of $\mathsf{DistributedDLog}_{\mathbb{G},g}(h,\delta,\mu,\phi)$ returns the smallest $c,\mu\le c\le\frac{4\mu\ln(4/\delta)}{\delta}$ such that $\phi(h)=0^{\log(4\mu/\delta)}$. In $\mathsf{DistributedDLog}_{\mathbb{G},g}(h\cdot g^\mu,\delta,\mu,\phi)$ the variable z ranges over the elements $h\cdot g^\mu,\dots,h\cdot g^{c+\mu}$ and the return value is $c-\mu$. Therefore, $\mathsf{DistributedDLog}_{\mathbb{G},g}(h,\delta,\mu,\phi)-\mathsf{DistributedDLog}_{\mathbb{G},g}(h\cdot g^\mu,\delta,\mu,\phi)=\mu$. By showing that the second case occurs with probability at least $1-\delta$ we complete the proof.
If R is a random function then the probability that $R(h\cdot g^c)=0^{\log(4\mu/\delta)}$ for some $c,0\le c\le\mu-1$ is exactly $1-(1-\delta/4\mu)^\mu$, which by induction is at most $1-(1-\mu\delta/4\mu)=\delta/4$. In addition,

$$\begin{aligned}\Pr[\forall c\in\{0,\dots,\frac{4\mu\ln\frac{4}{\delta}}{\delta}\}\ R(hg^c)\neq 0^{\log(4\mu/\delta)}] &\le (1-\frac{\delta}{4\mu})^{\frac{4\mu\ln(4/\delta)}{\delta}}\\ &< e^{-\ln 4/\delta}=\frac{\delta}{4}\end{aligned}$$

Since ϕ is selected randomly from an appropriate family of pseudo-random functions any non-uniform PPT algorithm can distinguish between R and ϕ with negligible probability. Specifically, the probability of the first case is bound by $|\Pr[\exists c,0\le c\le\mu-1\ \phi(h\cdot g^c)=0^{\log(4/\delta)}]-\delta/4|<\varepsilon(\delta)$ (for a negligible function $\varepsilon(\delta)$), since otherwise an efficient non-uniform algorithm can distinguish between R and ϕ by testing their value on hard-coded element (h,δ,μ,ϕ) and returning 1 if the result is $0^{\log(4\mu/\delta)}$. Similarly, the probability of the third case is bound by $|\Pr[\forall c\in\{0,\dots,\frac{4\ln(4/\delta)}{\delta}\}\ \phi(hg^c)\neq 0^{\log(4/\delta)}]-\delta/4|<\varepsilon(\delta)$.
The probability of the second case is therefore at least $1-\frac{\delta}{2}-2\varepsilon(\delta)$. $\square$

3.2 Homomorphic Secret Sharing

We now construct a simple version of the homomorphic secret sharing scheme, using the procedure $\mathsf{DistributedDLog}$ as a sub-routine. The resulting scheme will be a "secret-key" version, in which parties must share secret key information in order to generate secret shared values that can be combined via homomorphic evaluation. Further, the security of the scheme will rely on the assumption that the ElGamal encryption scheme is circular secure. These two restrictions will be removed in the following subsections.
Consider a DDH group $\mathbb{G}$ of prime order q (with λ bits of security) with generator g, and $\ell=\lceil\log_2 q\rceil$. We will use $c=c^{(1)},\dots,c^{(\ell)}$ to denote bits of an element $c\in\mathbb{Z}_q$ (i.e., $c=\sum_{i=0}^{\ell-1}2^ic^{(i+1)}$).

Overview of construction. All values generated within the secret sharing and homomorphic evaluation sit within three "levels." We will maintain notation as in the top portion of Figure 1.

Level 1: ElGamal Ciphertexts $[\![w]\!]_c$.

Initial input values w will be "uploaded" into the homomorphic evaluation system by generating an ElGamal encryption $[\![w]\!]_c$ of the value w with respect to a common secret key c, as well as encryptions $[\![c^{(i)}w]\!]_c$ of each of the products $c^{(i)}w$ for the bits $c^{(i)}$ of the corresponding key c.

Level 2: Additive secret shares $\langle y \rangle$.

Each value y in memory of the RMS program will be maintained via two sets of additive secret shares: $\langle y \rangle$ itself, and $\langle cy \rangle$ secret sharing the product of y with the ElGamal secret key c of the system. We start with secret shares of this form for each input value (e.g., in the secret-key setting, these will be generated as part of the Share procedure). Then, after each emulated RMS instruction, we will maintain the invariant that each newly computed memory value is stored as secret shares in this fashion.

Level 3: Multiplicative secret shares $\langle\!\langle xy \rangle\!\rangle$.

Multiplicative secret shares appear only as intermediate values during the execution of homomorphic evaluation (of multiplication), and are then converted back to additive shares via DistributedDLog.

Notation. For $x \in \mathbb{Z}_q$ (or $x \in \mathbb{Z}$ with $x < q$).

Items in which *both* parties receive same value.

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

Items in which each party receives a separate share.

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

Pairing Operations.

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

- 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 $\langle\!\langle xy \rangle\!\rangle = h_2^{\langle y \rangle} h_1^{-\langle cy \rangle}$.
- ConvertShares$(b, \langle\!\langle x \rangle\!\rangle, \mathsf{id}, \delta', M) \to \langle x \rangle$, with party identifier $b \in \{0,1\}$, nonce id, error parameter δ' and max size bound M.
  1. Denote by $\phi' : \mathbb{G} \to \{0,1\}^{\lfloor \log(4M/\delta') \rfloor}$ the appropriate prefix output of $\phi(\mathsf{id}, \cdot)$.
  2. Let x_b denote the present party b's share of $\langle\!\langle x \rangle\!\rangle$.
     If $b = 1$, then replace $x_b \leftarrow x_b^{-1}$ // i.e., convert so that $x_0/x_1 = g^x$.
  3. Let $x'_b \leftarrow \mathsf{DistributedDLog}_{\mathbb{G},g}(\langle\!\langle x \rangle\!\rangle, \delta', M, \phi')$.
  4. If $b = 0$, output x'_0. If $b = 1$, output $-x'_1$. // Output additive shares

Fig. 1: Notation for components of the homomorphic secret sharing scheme, and pairing operations for transforming between different components.

Remark 1 (Valid vs Random). We emphasize that a "valid" encoding (e.g., $[\![x]\!]_c, \langle x \rangle$, or $\langle\!\langle x \rangle\!\rangle$) speaks *only* to the correctness of decoding, and does not imply that the encoding is a *random* such encoding (e.g., a randomly sampled ciphertext, or fresh secret shares).

The bottom portion of Figure 1 describes two pairing operations that constitute *cross-level* computations. The first, $\mathsf{MultShares}$, "multiplies" a level-1 encoding by a level-2 encoding. Namely, it takes as input a level-1 (ElGamal ciphertext) encoding of x under key c, and level-2 (additive secret sharing) encodings of y, and of cy (the product of y with the ElGamal secret key), and outputs a level-3 (multiplicative secret sharing) encoding of the product xy. The second, $\mathsf{ConvertShares}$, converts from a level-3 (multiplicative) encoding back down to a level-2 (additive) encoding, with some probability of error, as dictated by given parameters. Roughly, the intermediate values of homomorphic evaluation will be maintained in level-2 (additive) secret shared form. Any linear combination of such shares can be performed directly. Multiplication between a value in memory and an input value will be performed by performing the $\mathsf{MultShares}$ between the input value (encoded in level 1) and the relevant memory value (encoded in level 2). This will yield an encoding of the product, but in level 3 (i.e., as multiplicative shares). To return the computed product back to level 2, the parties will execute the pairing procedure $\mathsf{ConvertShares}$, which essentially runs the $\mathsf{DistributedDLog}$ procedure from the previous subsection.

Homomorphic Secret Sharing Scheme - $\mathsf{Share}_{\mathbb{G},g}(1^\lambda, w_1, \ldots, w_n)$
Inputs: 1^λ and input values $w_1, \ldots, w_n \in \mathbb{Z}$

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

Fig. 2: Share generation procedure $\mathsf{Share}_{\mathbb{G},g}$ for secret sharing an input w via the homomorphic secret sharing scheme.

Remark 2 (Variable Types). Note that the relevant values are nearly all elements of $\mathbb{G}$ (e.g., elements of ElGamal ciphertexts) or of $\mathbb{Z}_q$ (e.g., the values cy_i, as well as shares of $\mathsf{SubtShare}$). An important exception to this are the values w_i, y_i, which are interpreted as (small) *integers.* When necessary for computation, we will sometimes perform a type cast back and forth between $\mathbb{Z}$ and $\mathbb{Z}_q$, using the notation $(int)(x) \in \mathbb{Z}$ for $x \in \mathbb{Z}_q$, and $(x \mod q) \in \mathbb{Z}_q$ for $x \in \mathbb{Z}$.

A few calculations provides us with the following two claims on these pairing procedures (see full version of this work for details).

Claim ($\mathsf{MultShares}$*).* For all values $x, y \in \mathbb{Z}_q$ and any key $c \in \mathbb{Z}_q$, then on input a valid level-1 encoding $[\![x]\!]_c$ with respect to key c, and valid level-2 encodings $\langle y \rangle, \langle cy \rangle$, the output of $\mathsf{MultShares}([\![x]\!]_c, \langle y \rangle, \langle cy \rangle)$ is a valid *level-3* encoding $\langle\!\langle xy \rangle\!\rangle$ of the product $xy \in \mathbb{Z}_q$.

Claim ($\mathsf{ConvertShares}$*).* For every $\delta > 0, M \in \mathbb{N}$, every polynomial-time algorithm $\mathcal{A}$,

$$\Pr\left[\phi \leftarrow \mathsf{PRFGen}(1^\lambda); (x, \mu) \leftarrow \mathcal{A}^\phi(1^\lambda) : \mu < M, \mathsf{DistributedDLog}(\langle\!\langle x \rangle\!\rangle, \mathsf{id}, \delta', M)\right]$$

We present the secret sharing scheme Share in Figure 2, and the corresponding homomorphic operations on shares Eval in Figure 3. Note that we distinguish variables of the straight-line program from the actual values by using $\hat{y}_i$ as opposed to y_i, etc.
We remark that our combined construction obtains a generalization of the notion of HSS from Section 2 both extending beyond the Boolean setting to support arithmetic computations over small integers, and allowing multiple outputs of the program from possibly different groups $\mathbb{Z}_\beta$ (as specified by the program description P). The restricted case of the definition coincides with our construction with size bound M set to 1 and all program outputs in a fixed group $\mathbb{Z}_\beta$.

Theorem 1 (Homomorphic Secret Sharing). *Assume that ElGamal is circular secure (as per Definition 3). Then the scheme* $(\mathsf{Share}, \mathsf{Eval})$ *as specified in Figures 2, 3 is a secure homomorphic secret sharing scheme for the class of deterministic branching programs.*

Proof. We analyze the correctness and security of $(\mathsf{Share}, \mathsf{Eval})$ in the following claims.

Claim (Correctness of Eval*).* For every input $w_1, \ldots, w_n \in \mathbb{Z}_q$ and every Restricted Multiplication Straight-line program P (as in Definition 4) of size S for which *all* intermediate values $y_i \in \mathbb{Z}$ in the execution of P are bounded in size by M, then

$$\begin{aligned}\Pr[(\mathsf{share}_0, \mathsf{share}_1) \leftarrow \mathsf{Share}_{\mathbb{G},g}(1^\lambda, w_1, \ldots, w_n) :& \\ \mathsf{Eval}_{\mathbb{G},g}(\mathsf{share}_0, P, M, \delta) + \mathsf{Eval}_{\mathbb{G},g}(\mathsf{share}_1, P, M, \delta)& \\ = P(w_1, \ldots, w_n)] \geq 1 - \delta.&\end{aligned}$$

Proof. Fix input $w_1, \ldots, w_n \in \mathbb{Z}_q$ and allowable RMS program P (for which all intermediate values in the execution of P are bounded in size by M).
We first address the probability of error due to execution of the Share Conversion Procedure $\mathsf{DistributedDLog}$. Observe that each call to $\mathsf{DistributedDLog}$

Homomorphic Share Evaluation of RMS Programs - $\mathsf{Eval}_{\mathbb{G},g}(b, \mathsf{share}, P, M, \delta)$
Inputs: Homomorphic secret share value share, RMS program description P of size $\leq S$, plaintext size bound $M \in \mathbb{Z}$, error bound δ.
Take $\delta' = \delta/((\ell+1)MS)$.

Parse share as in Figure 2. Parse P as a sequence of instructions (as in Definition 4); for each sequential instruction, perform the corresponding sequence of operations:

```
Instruction $(\mathsf{id}, \hat{y}_j \leftarrow \hat{w}_i)$:
    1: Let $\langle y_j \rangle \leftarrow \langle w_i \rangle$ and $\langle cy_j \rangle \leftarrow \langle cw_i \rangle$, where $\langle w_i \rangle, \langle cw_i \rangle$ are as in $\mathsf{share}$.

Instruction $(\mathsf{id}, \hat{y}_k \leftarrow \hat{y}_i + \hat{y}_j)$:
    1: Compute $\langle y_k \rangle \leftarrow \langle y_i \rangle + \langle y_j \rangle$, directly on the additive shares (over $\mathbb{Z}_q$).
    2: Compute $\langle cy_k \rangle \leftarrow \langle cy_i \rangle + \langle cy_j \rangle$, on the additive shares (over $\mathbb{Z}_q$).

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

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

Fig. 3: Procedures for performing homomorphic operations on secret shares. Here, notation $\langle y \rangle$ is used to represent *this party's* share of the corresponding subtractive secret shared pair. Evaluation maintains the invariant that each of the *additive secret shares* $\langle y_i \rangle$ encode the correct current computation value of $\hat{y}_i$.

takes place with a *unique* nonce value (id, t) where id is the instruction id (unique for each instruction) and for each distinct value $t \in \{0, 1, \ldots, \ell\}$. This means that while all calls share the same psuedorandom function ϕ (as sampled during the Share secret sharing procedure), each execution makes use of disjoint regions of the PRF domain, which by the pseudorandomness guarantee, is computationally indistinguishable from independently sampled PRFs. Now, observe that the homomorphic evaluation of program P performs at most $S(\ell + 1)$ executions of the Share Conversion Procedure (with error parameter $\delta' = \delta/(\ell + 1)MS$). From Proposition 1, for *any* pair of values h and $h' = h \cdot g^d$ with $d \leq M$, the probability (over the randomness of the PRF seed s) that the outputs $a, a' \in \mathbb{Z}$ of the respective parties do not satisfy $a' = a + d$ is bounded by $\delta' d$, which is in turn bounded by $\delta' M$; thus, by a union bound no such errors will occur with probability $\delta'(\ell + 1)MS = \delta$.

Assume, then, that every Share Conversion execution returns without error; that is, that for every such h, h' we have $\mathsf{DistributedDLog}_{\mathbb{G},g}(h, \delta', M, \phi) - \mathsf{DistributedDLog}_{\mathbb{G},g}(h', \delta', M, \phi) = \log_g h - \log_g h'$. We prove that following invariant is maintained at each step of homomorphic evaluation:

Invariant: For every memory item $\hat{y}_i$, let $y_i \in \mathbb{Z}$ denote the correct value that should be presently stored in memory. Then the shares $\langle y_i \rangle = (y_i^A, y_i^B) \in \mathbb{Z}_q^2$ and $\langle cy_i \rangle = (v^A, v^B) \in \mathbb{Z}_q^2$ held by the parties satisfy:

(a) $(int)(y_i^A + y_i^B) = y_i \in \mathbb{Z}$ (where addition is in $\mathbb{Z}_q$).

(b) $v^A + v^B = cy_i \in \mathbb{Z}_q$.

Note that the invariant holds vacuously at the start of Eval, as all memory locations are empty. Consider each instruction type of P.

1. Loading memory items $(\mathsf{id}, \hat{y}_j \leftarrow \hat{w}_i)$. By construction of Share, we have that $\langle w_i \rangle$ and $\langle cw_i \rangle$ are valid $\mathbb{Z}_q$ secret sharings of w_i and cw_i. This directly gives us invariant (b). Further, we have $\langle w_i \rangle = (w_i^A, w_i^B) \in \mathbb{Z}_q^2$ such that $w_i^A + w_i^B = w_i \mod q$. Since $w_i \in \mathbb{Z}$ is bounded by $0 \leq w_i \leq M < q$, this implies that $(int)(w_i^A + w_i^B) = w_i \in \mathbb{Z}$.
2. Linear combination $(\mathsf{id}, \hat{y}_k \leftarrow \hat{y}_i + \hat{y}_j)$. The additive secret sharing scheme is linearly homomorphic with respect to $\mathbb{Z}_q$ (implying invariant (b)). Recall we are guaranteed that no intermediate computation value (in particular, $y_k, y_i, y_j \in \mathbb{Z}$) will ever have magnitude greater than $M < q$. Thus, as above, the desired invariant (a) holds as well for $\langle y_k \rangle$.
3. Multiplication of memory value with input $(\mathsf{id}, \hat{y}_k \leftarrow \hat{w}_i \cdot \hat{y}_j)$. Consider the resulting shares $\langle y_k \rangle$ and $\langle cy_k \rangle$.
   By Claim 3.2, the shares $\langle\!\langle w_i y_j \rangle\!\rangle$ computed via $\mathsf{MultShares}$ constitute a valid level-3 sharing of the product $w_i y_j$ (as per Figure 1). Since we are in the case where $\mathsf{DistributedDLog}$ does not err, the resulting converted shares $\langle w_i y_j \rangle$ encode exactly the value $w_i y_j \in \mathbb{Z}_q$. Since $w_i y_j$ is an intermediate computation value y_k in the evaluation of P, we have $0 \leq w_i y_j \leq M < q$. Thus, invariant (a) holds.
   Consider now $\langle cy_k \rangle$. From precisely the same argument as above (since we are in the case where $\mathsf{DistributedDLog}$ does not err), we have for each $t \in [\ell]$ that the computed intermediate value $\langle c^{(t)} \cdot y_k \rangle$

is a level-2 encoding of the corresponding value $c^{(t)}y_k \in \mathbb{Z}_q$. Since y_k is an intermediate computation value in P we have $0 \leq y_k \leq M < q$, and so for $c^{(t)} \in \{0,1\}$, then $0 \leq c^{(t)} \cdot y_k \leq M < q$. Thus, it holds $(int)(v_t^A - v_t^B) = c^{(t)} \cdot w_i \cdot y_j \in \mathbb{Z}$.
Combining the respective values over $t \in [\ell]$, it holds $\big(\sum_{t\in[\ell]} 2^t v_t^A\big) - \big(-\sum_{t\in[\ell]} 2^t v_t^B\big) = -\sum_{t\in[\ell]} 2^t (v_t^A - v_t^B) = -\sum_{t\in[\ell]} 2^t c^{(t)} \cdot w_i \cdot y_j = -c \cdot w_i \cdot y_j \in \mathbb{Z}_q$. Therefore, invariant (b) holds.

4. Output value from memory $(\mathsf{id}, \beta, \hat{O}_j \leftarrow \hat{y}_i)$. The invariants (a),(b) hold inductively for the existing shares of y_i. In particular, $\langle y_i \rangle = (y_i^A, y_i^B) \in \mathbb{Z}_q^2$ for which $(int)(y_i^A - y_i^B) = y_i \in \mathbb{Z}$. Since both parties apply the identical offset $\phi(\mathsf{id}, g)$ to their shares, this property is preserved; i.e., invariant (a) holds. Invariant (b) is trivially maintained, as the shares of $\langle cy_i \rangle$ are untouched.
Now, consider the validity of the output value O_j. Since $0 \leq y_i < M$, then either

$$(int)\Big(y_i^A + \phi(\mathsf{id}, g)\Big) - (int)\Big(y_i^B + \phi(\mathsf{id}, g)\Big) = y_i, \qquad (1)$$

or

$$(int)\Big(y_i^A + \phi(\mathsf{id}, g)\Big) - (int)\Big(y_i^B + \phi(\mathsf{id}, g)\Big) = y_i - q \qquad (2)$$

$$\text{and } (int)(y_i^B + \phi(\mathsf{id}, g)) \geq q - M. \qquad (3)$$

In case (1), it holds that

$$(int)\Big(y_i^A + \phi(\mathsf{id}, g)\Big) \bmod \beta - (int)\Big(y_i^B + \phi(\mathsf{id}, g)\Big) \bmod \beta = y_i \pmod{\beta},$$

as desired; it remains to bound the probability of case (2). By the pseudo-randomness of ϕ, then over the random sampling of $\phi \leftarrow \mathsf{PRFGen}(1^\lambda)$, the probability of $(int)(x + \phi(\mathsf{id}, g)) \geq q - M$ for *any* of the S partial computation value shares x encountered during the course of execution is bounded by $S(M/q) + \nu(\lambda)$ for some negligible function λ. Note that this holds unconditionally, as it constitutes an efficient test for ϕ. Finally, since $q = |\mathbb{G}|$ and $\mathbb{G}$ is a DDH-hard group, then necessarily $q \in \lambda^{\omega(1)}$, implying for sufficiently large λ that this probability $S(M/q) + \nu(\lambda)$ is bounded above by our choice of allowable per-instruction error δ'.

Claim (Security of Share*).* Based on the assumption that ElGamal is a weakly circular secure encryption scheme (as per Definition 3), then Share is a computationally secure secret sharing scheme.

Proof. We prove that the distribution of a single party's share resulting from Share is computationally indistinguishable from a distribution that is *independent* of the shared values $w_1, \ldots, w_n$, via two hybrids.

Hybrid 0: Honest share distribution for party $b \in \{0,1\}$. That is, $\mathsf{HonestShare}_{\mathbb{G},g}(1^\lambda, w_1, \ldots, w_n, b)$ defined by

$$\{\mathsf{share}_b : (\mathsf{share}_0, \mathsf{share}_1) \leftarrow \mathsf{Share}_{\mathbb{G},g}(1^\lambda, w_1, \ldots, w_n)\}.$$

Hybrid 1: Distribution $\mathsf{SimShare}'_{\mathbb{G},g}(1^\lambda, w_1, \ldots, w_n, b)$, consisting of honest secret shares, but with additive secret shares replaced by uniform values:

1. Sample honest share $(\mathsf{share}_0, \mathsf{share}_1) \leftarrow \mathsf{Share}_{\mathbb{G},g}(1^\lambda, w_1, \ldots, w_n)$, and parse $\mathsf{share}_b = \left\{ \phi, \left([\![w_i]\!]_c, \left\{ [\![c^{(t)} w_i]\!]_c \right\}_{t\in[\ell]}, \langle w_i \rangle_b, \langle c w_i \rangle_b \right)_{i\in[n]} \right\}$.
2. Sample random shares $v_i, v'_i \leftarrow \mathbb{Z}_q \forall i \in [n]$ (replacing the shares $\langle w_i \rangle_b, \langle c w_i \rangle_b$), and output $\left\{ \phi, \left([\![w_i]\!]_c, \left\{ [\![c^{(t)} w_i]\!]_c \right\}_{t\in[\ell]}, v_i, v'_i \right)_{i\in[n]} \right\}$.

Hybrid 2: Fully simulated shares, as follows.
Distribution $\mathsf{SimShare}_{\mathbb{G},g}(1^\lambda, n, b)$:

1. Sample a PRF with input $\{0,1\}^n \times \mathbb{G}$ and output $\{0,1\}^\ell$: $\phi \leftarrow \mathsf{PRFGen}(1^\lambda)$.
2. Sample an ElGamal secret key $c \leftarrow \mathbb{Z}_q$, and compute the public key $e = g^c$.
3. For each input $i \in [n]$,
   (a) Sample $(\ell{+}1)$ ElGamal encryptions of 0: $[\![0_{i,t}]\!]_c \leftarrow \mathsf{Enc}_{\mathsf{ElGamal}}(e, 0)$, for $t \in \{0, 1, \ldots, \ell\}$.
   (b) Sample two random secret shares $v_i, v'_i \leftarrow \mathbb{Z}_q$ (as in the previous hybrid).
4. Output $\left\{ \phi, \left([\![0_{i,0}]\!]_c, \{[\![0_{i,t}]\!]_c\}_{t\in[\ell]}, v_i, v'_i \right)_{i\in[n]} \right\}$.

Observe that for any $w_1, \ldots, w_n \in \mathbb{Z}$ and $b \in \{0,1\}$ the distributions $\mathsf{HonestShare}_{\mathbb{G},g}(1^\lambda, w_1, \ldots, w_n, b)$ and $\mathsf{SimShare}'_{\mathbb{G},g}(1^\lambda, w_1, \ldots, w_n, b)$ are *identically* distributed, by the perfect security of the 2-out-of-2 secret sharing scheme. Now, suppose there exists $b^* \in \{0,1\}$, inputs $w_1^*, \ldots, w_n^* \in \mathbb{Z}$ each satisfying $0 \le w_i^* \le q$, a nonuniform polynomial-time adversary $\mathcal{A}^*$, and non-negligible function ϵ for which

$$\Pr\Big[\mathsf{chall} \leftarrow \{0,1\}; \mathsf{share}_{b^*} \leftarrow \mathsf{ChallengeGen}(1^\lambda, w_1^*, \ldots, w_n^*, b^*, \mathsf{chall}); \\ \mathsf{guess} \leftarrow \mathcal{A}(1^\lambda, \mathsf{share}_{b^*}) : \mathsf{guess} = \mathsf{chall}\Big] > \epsilon(\lambda),$$

where $\mathsf{ChallengeGen}(1^\lambda, w_1^*, \ldots, w_n^*, b^*, \mathsf{chall})$ is sampled for $\mathsf{chall} = 0$ as $\mathsf{SimShare}'_{\mathbb{G},g}(1^\lambda, w_1^*, \ldots, w_n^*, b^*)$, and for $\mathsf{chall} = 1$ as $\mathsf{SimShare}_{\mathbb{G},g}(1^\lambda, n, b^*)$. Such an adversary can directly be used to break the weak circular security of the ElGamal encryption.
Indeed, consider an adversary $\mathcal{B}'$ in the weak circular security game (see Definition 3) for ElGamal, with $b^*, w_1^*, \ldots, w_n^* \in \{0,1\}$ hardcoded, who proceeds as follows:

1. $\mathcal{B}$ receives a challenge ElGamal public key pk.
2. For every $i \in [n]$, generate an encryption of w_i^*: $[\![w_i^*]\!]_c \leftarrow \mathsf{Enc}(\mathsf{pk}, w_i^*)$.
3. For every $i \in [n]$ for which $w_i^* \neq 0$, query the Weak Circular Security oracle $\mathcal{O}$, and receive a vector of ElGamal ciphertexts $([\![x_{i,1}]\!]_c, \ldots, [\![x_{i,\ell}]\!]_c)$, which are either encryptions of $x_{i,t} = 0 \ \forall t \in [\ell]$, or encryptions of the bits $x_{i,t} = c^{(t)}$ of the secret key c.
   For each $t \in [\ell]$, take $[\![x_{i,t} w_i^*]\!]_c := ([\![x_{i,t}]\!]_c)^{w_i^*}$.
4. For every $i \in [n]$ for which $w_i^* = 0$, sample ℓ encryptions of 0: i.e., for each $t \in [\ell]$, take $[\![x_{i,t}]\!]_c \leftarrow \mathsf{Enc}_{\mathsf{ElGamal}}(\mathsf{pk}, 0)$.

5. Sample a PRF with input $\{0,1\}^n \times \mathbb{G}$ and output $\{0,1\}^\ell$: $\phi \leftarrow \mathsf{PRFGen}(1^\lambda)$.
6. Sample $2n$ random values for the "additive secret shares": i.e., $v_i \leftarrow \mathbb{Z}_q, v'_i \leftarrow \mathbb{Z}_q$ for each $i \in [n]$.
7. Let $\mathsf{share}_{b^*} := \left\{\phi, \left(\llbracket w_i^* \rrbracket_c, \{\llbracket x_{i,t} w_i^* \rrbracket_c\}_{t\in[\ell]}, v_i, v'_i\right)_{i\in[n]}\right\}$.
8. Output $\mathsf{guess} \leftarrow \mathcal{A}^*(1^\lambda, \mathsf{share}_{b^*})$.

Note that for any fixed $0 \le w < q$, the distribution $\{r \cdot w \in \mathbb{Z}_q : r \leftarrow \mathbb{Z}_q\}$ is identically distributed to $\{r \leftarrow \mathbb{Z}_q\}$ since $\mathbb{Z}_q$ is a field (and thus $w \neq 0$ has a multiplicative inverse). This means for both $v = 0, 1$, the distribution $\{(\llbracket v \rrbracket_c)^w : \llbracket v \rrbracket_c \leftarrow \mathsf{Enc}_{\mathsf{ElGamal}}(\mathsf{pk}, v)\} = \{(g^r)^w, (g^{r+v})^w) : r_{i,t} \leftarrow \mathbb{Z}_q\}$ is identical to $\{(g^r, g^{r+vw}) : r \leftarrow \mathbb{Z}_q\} = \{\llbracket vw \rrbracket_c \leftarrow \mathsf{Enc}_{\mathsf{ElGamal}}(\mathsf{pk}, vw)\}$. Therefore, if the circular security oracle $\mathcal{O}$ provided encryptions of the bits of the secret key c then share_{b^*} is distributed precisely as $\mathsf{SimShare}'_{\mathbb{G},g}(1^\lambda, w_1^*, \ldots, w_n^*, b^*)$, and if it provided encryptions of 0 then share_{b^*} is distributed precisely as $\mathsf{SimShare}_{\mathbb{G},g}(1^\lambda, n, b^*)$. Security of the secret sharing scheme follows.

3.3 Public-Key HSS for Branching Programs

In the construction of the previous section, secret shares of an input w consisted of ElGamal encryptions $\llbracket w \rrbracket_c, \{\llbracket c^{(t)} w \rrbracket_c\}_{t\in[\ell]}$ and additive secret shares $\langle w \rangle, \langle cw \rangle$, where c was a (freshly sampled) key for ElGamal. At face value, it would seem that one must know the value of the key c in order to generate these values—meaning, in turn, that homomorphic computation can only be performed on the data of a single user who generates the key c. In this section, we demonstrate that by leveraging the homomorphic properties of ElGamal encryption, we can in fact generate all required values for a secret sharing of w while maintaining security, given only "public key" information independent of the input w. That is, we obtain homomorphic encryption with distributed evaluation, as discussed in Section 2.

More formally, we now consider a separate procedure Gen for generating common setup information pk and secret evaluation keys $\mathsf{ek}_0, \mathsf{ek}_1$ (which we consider to be given to two servers). Given access to pk, a user can "upload" his input w to the system via Enc. Then, given their respective evaluation keys, two servers can perform non-interactive homomorphic computations on *all* users' inputs via Eval.

In our construction, the algorithm Gen will sample an ElGamal key pair, and will output pk consisting of encryptions $\llbracket 1 \rrbracket_c, \{\llbracket c^{(t)} \rrbracket_c\}_{t\in[\ell]}$ and evaluation keys ek_b corresponding to additive secret shares of $\langle c \rangle$. In Enc, a user computes the necessary ciphertexts $\llbracket w \rrbracket_c$ and $\{\llbracket c^{(t)} w \rrbracket_c\}_{t\in[\ell]}$ for his input w by simply exponentiating the ciphertexts in pk component-wise by w, (i.e., making use of multiplicative homomorphism of ElGamal). The final required values $\langle w \rangle, \langle cw \rangle$ can be obtained directly by the servers within Eval by performing the procedure for a homomorphic multiplication between the "input value" w (i.e., given $\llbracket w \rrbracket_c, \{\llbracket c^{(t)} \rrbracket_c\}_{t\in[\ell]}$) together with "memory value" 1 (i.e., given a trivial sharing $\langle 1 \rangle$ together with $\langle c \rangle$ from ek).

A formal description of the algorithms $\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval}$ is given in Figure 4.

Theorem 2 (DEHE). *Assume that ElGamal is circular secure (as per Definition 3). Then the scheme* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval})$ *as given in Figure 4 is a secure Distributed-Evaluation Homomorphic Encryption scheme for the class of deterministic branching programs.*

Proof. Correctness. To reduce to Theorem 1, it suffices to demonstrate: (1) the values $[\![w]\!]_c, \{[\![c^{(t)}w]\!]_c\}_{t\in[\ell]}$ as generated in Enc are valid level-1 encodings of w and $\{c^{(t)}w\}_{t\in[\ell]}$; and (2) the values $\langle w\rangle, \langle cw\rangle$ as generated in Eval are valid level-2 encodings of w and cw. Property (1): Recall $[\![w]\!]_c$ was obtained as (h_1^w, h_2^w) for (h_1, h_2) a valid level-1 encoding of 1. This means $h_2 h_1^{-c} = g^1$, which implies $(h_2^w)(h_2^w)^{-c} = g^w$, as desired. Same for each $[\![c^{(t)}w]\!]_c$. Property (2): Holds by the correctness of homomorphic RMS multiplication evaluation of Eval as per Theorem 1.
Security. Semantic security of the scheme follows as in Theorem 1, assuming circular security of ElGamal. Namely, the view of a server holding ek_b consists of information theoretically hiding secret shares, ciphertexts of values independent of the secret key, and vectors of ciphertexts encrypting the vector $(c^{(1)}w, \ldots, c^{(\ell)}w)$ for various inputs w. An adversary distinguishing between this view and one consisting of random share elements and ciphertexts of 0 can be used to break the circular security of ElGamal, precisely as in Theorem 1.

Comparing the complexity of the public-key scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval})$ to that of the secret-key scheme $(\mathsf{Share}, \mathsf{Eval})$ from the previous section, we see that the computation cost to the user for uploading inputs $w_1, \ldots, w_n$ via Enc is essentially equivalent to the cost of sharing the inputs via Share (exponentiating given ciphertexts by the respective inputs in one case, versus encrypting the values directly in the other), but the cost of each "load input" instruction $(\mathsf{id}, \hat{y}_j \leftarrow \hat{w}_i)$ within the homomorphic evaluation now incurs the cost of a multiplication step to generate additive secret shares $\langle w_i\rangle, \langle cw_i\rangle$ given only $\langle c\rangle$ and the uploaded ElGamal ciphertexts associated with w_i, as opposed to being essentially for free for the client to generate $\langle w_i\rangle, \langle cw_i\rangle$ when he knew the values of w_i, cw_i.

3.4 Removing the Circular Security Assumption

We now show how to remove the ElGamal circular security assumption in the construction of distributed-evaluation homomorphic encryption in the previous section, yielding a scheme that relies solely on DDH. Our new construction replaces ElGamal encryption with the ElGamal-like cryptosystem of Boneh, Halevi, Hamburg, and Ostrovsky (BHHO) [5], which is provably circular secure based on DDH. At a high level, BHHO ciphertexts possess an analogous structure of "linear algebra in the exponent," which allows us to mirror the same procedure we used with ElGamal for multiplicatively pairing a ciphertext with an additively shared value.
It will be convenient to consider a slightly modified version of the BHHO scheme, given below, in which the message space is a subset of the *exponent* space $\mathbb{Z}_q$ instead of the group $\mathbb{G}$ itself (i.e., the multiplication by

Distributed-Evaluation Homomorphic Encryption: $\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval}$

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

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

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

1. Parse pk as in Gen above.
2. Compute the following ElGamal ciphertexts:
   (a) Of $w \in \mathbb{Z}$: parse $[\![1]\!]_c = (h_1, h_2)$, and let $[\![w]\!]_c = (h_1^w, h_2^w) \in \mathbb{G}^2$.
   (b) Of $c^{(t)}w \in \mathbb{Z}$: i.e., for each $t \in [\ell]$, parse $[\![c^{(t)}]\!]_c = (h_1^{(t)}, h_2^{(t)})$ let $[\![c^{(t)}w]\!]_c = ((h_1^{(c)})^w, (h_2^{(c)})^w)$.
3. Output $([\![w]\!]_c, \{[\![c^{(t)}w]\!]_c\}_{t \in [\ell]})$.

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

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

Fig. 4: Construction of "public-key" variant of homomorphic secret sharing: i.e., homomorphic encryption with distributed evaluation.

message m in standard encryption is replaced by g^m). Since decryption of such scheme requires taking discrete log, efficient decryption will hold for a polynomial-size message space.

Definition 5 (BHHO Encryption [5]). *Let $\mathbb{G}$ be a group of prime order q and g a fixed generator of $\mathbb{G}$. The size of $\mathbb{G}$ is determined by a security parameter λ, in particular, $1/q$ is negligible in λ. The BHHO public-key encryption scheme for polynomial-size message space $\mathsf{Msg} \subset \mathbb{Z}_q$ is as follows:*

– *Key Generation. Let $\ell := \lceil 3 \log_2 q \rceil$. Choose random $g_1, \ldots, g_\ell \leftarrow \mathbb{G}$ and a random secret key vector $s = (s_1, \ldots, s_\ell) \leftarrow \{0,1\}^\ell$. Let $h = (g^{s_1} \cdots g_\ell^{s_\ell})^{-1}$ and define the public and secret keys to be*

$$\mathsf{pk}_{\mathsf{BHHO}} := (g_1, \ldots, g_\ell, h), \quad \mathsf{sk}_{\mathsf{BHHO}} = (g^{s_1}, \ldots, g^{s_\ell}).$$

– *Encryption. To encrypt $m \in \mathsf{Msg}$, choose a random $r \leftarrow \mathbb{Z}_q$ and output the ciphertext*

$$(g_1^r, \ldots, g_\ell^r, h^r \cdot g^m).$$

– *Decryption. Let $(c_1, \ldots, c_\ell, d)$ be a ciphertext and $\mathsf{sk}_{\mathsf{BHHO}} = (v_1, \ldots, v_\ell)$ a secret key. Do:*
  - *Decode the secret key: For $i = 1, \ldots, \ell$, set $s_i \leftarrow 0$ if $v_i = 1$ and $s_i \leftarrow 1$ otherwise.*
  - *Output $m \in \mathsf{Msg}$ for which $g^m = d \cdot (c_1^{s_1} \cdots c_\ell^{s_\ell})$.*

Theorem 3 (Circular Security of BHHO [5]). *Assuming DDH, the BBHO public-key encryption scheme satisfies circular security, as per Definition 3.*

In order to emulate the homomorphic evaluation procedure of the previous sections, there are two steps we must modify:
First, we must provide a means for pairing a BHHO ciphertext of an input w with additive secret sharings of a value x to obtain a multiplicative secret sharing of g^{wx}. For ElGamal this was done given $\langle x \rangle$ and $\langle cx \rangle$, and computing $h_2^{\langle x \rangle} h_1^{-\langle cx \rangle}$. Now, for BHHO, we can perform an analogous "partial decryption" procedure given shares $\langle x \rangle$ and $\{\langle s_i x \rangle\}_{i \in [\ell]}$, for the bits s_i of the BHHO secret key. The corresponding pairing computation is given as $\mathsf{MultShares}$ in Figure 5.
Once we obtain a multiplicative secret sharing of g^{wx}, we can perform the same share-conversion procedure $\mathsf{DistributedDLog}$ from the previous sections to return to an additive secret sharing of wx (with some error probability δ). But, to be able to perform a future pairing as above, we additionally must generate additive secret sharings $\langle wxs_i \rangle$ for each of the bits s_i of the secret key (analogous to generating $\langle cwx \rangle$ in the ElGamal case). Conveniently, this BHHO task is actually slightly simpler than that for ElGamal: whereas before we had to deal with the large size of the secret key $c \in \mathbb{Z}_q$ by operating on a bit decomposition of c and then reconstructing, here the secret key $(s_1, \ldots, s_\ell)$ is already interpreted as a binary vector. This means we can perform the multiplication steps directly without requiring the decomposition/reconstruction steps.

We remark that BHHO ciphertexts are multiplicatively homomorphic in the same fashion as ElGamal, which allows us to obtain a public-key variant of the secret sharing scheme precisely as in the previous section. The required procedure of modifying a ciphertext of some message x to one encrypting xy given y is explicitly described as $(\llbracket x \rrbracket_s)^y$ in Figure 5.

In Figure 5, we provide the modified notation and pairing procedures for this setting. The remaining notations $\langle x \rangle, \langle\!\langle x \rangle\!\rangle$ and pairing operation $\mathsf{ConvertShares}$ will remain as in the previous sections (Figure 1). Given these sub-procedures, we present in Figures 6 and 7 the corresponding algorithms $\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval}$.

DDH-Based Notation and Pairing Operations

Let $s = (s_1, \dots, s_\ell) \in \{0,1\}^\ell$.

- **Notation:** $\llbracket x \rrbracket_s = (g'_1, \dots, g'_\ell, h') \in \mathbb{G}^{\ell+1}$ for which $g^x = h' \cdot \prod_{t \in \ell} (g'_t)^{s_t}$.
  That is, BHHO ciphertext of x w.r.t. secret key s is the new level-1 encoding.
- **Pairing:** $\mathsf{MultShares}\Big(\llbracket x \rrbracket_s, \langle y \rangle, \{\langle s_t y \rangle\}_{t \in [\ell]}\Big) \to \langle\!\langle xy \rangle\!\rangle$.
  1. Denote $\llbracket x \rrbracket_s = (g'_1, \dots, g'_\ell, h') \in \mathbb{G}^{\ell+1}$.
  2. Compute $\langle\!\langle xy \rangle\!\rangle = (h')^{\langle y \rangle} \cdot \prod_{t \in [\ell]} (g'_i)^{\langle s_t y \rangle}$.
- **Pairing:** $(\llbracket x \rrbracket_s)^y$, for ciphertext $\llbracket x \rrbracket_s \in \mathbb{G}^{\ell+1}$ and plaintext $y \in \mathbb{Z}_q$.
  1. Denote $\llbracket x \rrbracket_s = (g'_1, \dots, g'_\ell, h') \in \mathbb{G}^{\ell+1}$.
  2. Output $\llbracket xy \rrbracket_s := \big((g'_1)^y, \dots, (g'_\ell)^y, (h')^y\big) \in \mathbb{G}^{\ell+1}$.

Fig. 5: Modified DDH-based notation and pairing operations, making use of BHHO encryption [5].

Theorem 4 (DEHE from DDH). *Assuming DDH, then the scheme* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval})$ *as given in Figures 6, 7 is a secure Distributed-Evaluation Homomorphic Encryption scheme for the class of deterministic branching programs.*

4 Applications

In this section we describe applications of our homomorphic secret sharing scheme and its public-key variant in the context of secure computation. We restrict the attention to security against semi-honest parties; to get similar asymptotic efficiency in the presence of malicious parties, one can apply general-purpose compilation techniques [22, 29]. For lack of space, formal protocol descriptions and security proofs are postponed to the full version.

4.1 Succinct Protocols for Branching Programs

Our protocols for branching programs can be based either on the weaker HSS primitive via the transformation from [6], or can be built more directly from the public-key variant. We present here the latter approach,

```
DDH-Based Distributed-Evaluation HE: Gen and Enc Algorithms
Gen($1^\lambda$):
 1. Sample a PRF with input $\{0,1\}^n \times \mathbb{G}$ and output $\{0,1\}^\ell$: $\phi \leftarrow \mathsf{PRFGen}(1^\lambda)$.
 2. BHHO Key Setup:
    (a) Sample a DDH-hard group and generator $(\mathbb{G}, g, q) \leftarrow \mathcal{IG}(1^\lambda)$.
    (b) Sample a BHHO secret key: $s \leftarrow \{0,1\}^\ell$.
 3. Sample BHHO encryptions:
    (a) The constant $1 \in \mathbb{Z}_q$: let $[\![1]\!]_c \leftarrow \mathsf{Enc}(e, 1)$.
    (b) The bits of the secret key $s$: $\forall i \in [\ell]$, let $[\![s_i]\!]_s \leftarrow \mathsf{Enc}(e, s_i)$.
 4. Sample 2-out-of-2 additive secret sharings:
    (a) The constant $1 \in \mathbb{Z}_q$: $\langle 1 \rangle \leftarrow \mathsf{AdditiveShare}(1)$. // Included for notational convenience
    (b) The bits of the secret key $s$: $\forall i \in [\ell]$, let $\langle s_i \rangle \leftarrow \mathsf{AdditiveShare}(s_i)$.
 5. Output $\mathsf{pk} = \big(\mathbb{G}, g, [\![1]\!]_s, \{[\![s_i]\!]_s\}_{i\in[\ell]}\big)$, $\mathsf{ek}_b = \big(\mathsf{pk}, \langle 1 \rangle, \{\langle s_i \rangle\}_{i\in[\ell]}\big)$.

Enc$_{\mathbb{G},g}$(pk, $w$):
 1. Compute the following values:
    (a) BHHO encryption of $w \in \mathbb{Z}$: let $[\![w]\!]_s = ([\![1]\!]_s)^w \in \mathbb{G}^{\ell+1}$.
    (b) BHHO encryptions of $(w \cdot s_i) \in \mathbb{Z}$: i.e., for each $i \in [\ell]$, let $[\![s_i w]\!]_s = ([\![s_i]\!]_s)^w$.
 2. Output $([\![w]\!]_s, \{[\![s_i w]\!]_s\}_{i\in[\ell]})$.
```

Fig. 6: DDH-based homomorphic encryption with distributed evaluation, making use of the BHHO cryptosystem.

which is more direct. For simplicity, we restrict the attention to the case of evaluating a *single* branching program P on inputs x_0, x_1 held by Party 0 and Party 1 respectively. This can be extended in a straightforward way to functions with m bits of output that are computed either by m separate branching programs or by a single RMS program.

The simplest protocol proceeds as follows. The two parties run a general-purpose protocol (such as Yao's protocol) to distribute the key generation Gen. In the end of this sub-protocol, both parties hold a public key pk and each holds a secret evaluation key sk_b. While this step may be expensive, its complexity depends (polynomially) only on the security parameter λ, and moreover the same key setup can be used for evaluating an arbitrary number of branching programs on an arbitrary number of inputs. In this basic version of the protocol, the key generation protocol is the only step that does not make a black-box use of the underlying DDH group.

Next, each party uses $\mathsf{Enc}(\mathsf{pk}, \cdot)$ to encrypt every bit of its input, and sends the encryptions to the other party. Finally, the two parties locally run Eval to generate additive (mod-2) shares of the output $P(x_0, x_1)$. If Eval had negligible error, the parties could simply exchange their shares of the output, since the share sent to Party b is determined by the output and the share computed by Party b.

The fact that Eval has a non-negligible error δ is problematic for two reasons. First, it poses a correctness problem. This can be fixed by setting δ to be a constant (say, $\delta = 1/4$), running σ independent instances of

DDH-Based Homomorphic Evaluation - $\mathsf{Eval}_{\mathbb{G},g}(\mathsf{ek}, b, \mathsf{share}, P, M, \delta)$
Inputs: Homomorphic secret share parameters params, shared value share, RMS program description P of size $\leq S$, plaintext size bound $M \in \mathbb{Z}$, error bound δ.
Take $\delta' = \delta/((k+1)MS)$.

Parse ek as in Figure 6, and interpret $\hat{1}$ as loaded into memory, via $\langle 1 \rangle, \{\langle s_i \rangle\}_{i \in [\ell]}$ as given. Parse P as a sequence of instructions (as in Definition 4); for each sequential instruction, perform the corresponding sequence of operations described below.

```
Instruction $(\mathsf{id}, \hat{y}_j \leftarrow \hat{w}_i)$:
  1: Execute the multiplication operation $(\mathsf{id}, \hat{y}_j \leftarrow \hat{w}_i \cdot \hat{1})$, as described below.

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

Instruction $(\mathsf{id}, \hat{y}_k \leftarrow \hat{w}_i \cdot \hat{y}_j)$:
  1: Let $[\![w_i]\!]_s$ and $\{[\![s_t w_i]\!]_s\}_{t \in [\ell]}$ be the BHHO ciphertexts associated with $w_i$, and
     $\langle y_j \rangle$ and $\{\langle s_t y_j \rangle\}_{t \in [\ell]}$ the additive secret shares associated with $y_j$.
  2: Compute $\langle\!\langle w_i y_j \rangle\!\rangle = \mathsf{MultShares}([\![w_i]\!]_s, \langle y_j \rangle, \{\langle s_t y_j \rangle\}_{t \in [\ell]})$, as in Figure 5.
  3: Execute Share Conversion: $\langle w_i y_j \rangle = \mathsf{ConvertShares}(b, \langle\!\langle w_i y_j \rangle\!\rangle, \delta', M, \phi)$.
     // $\mathsf{DistributedDLog}$ will yield subtractive shares of $w_i \cdot y_j$
  4: for $t = 1$ to $\ell$ do    // Repeat above process for each $s_t \cdot y_k$
  5:     Compute $\langle\!\langle s_t w_i y_j \rangle\!\rangle = \mathsf{MultShares}([\![s_t w_i]\!]_s, \langle y_j \rangle, \{\langle s_t y_j \rangle\}_{t \in [\ell]})$.
  6:     Execute Share Conversion: $\langle s_t w_i y_j \rangle = \mathsf{ConvertShares}(b, \langle\!\langle s_t w_i y_j \rangle\!\rangle, \delta', M, \phi)$.
  7: end for
  8: Let $\langle y_k \rangle \leftarrow \langle w_i y_j \rangle$ and $\langle s_t y_k \rangle \leftarrow \langle s_t w_i y_j \rangle$, for each $t \in [\ell]$.

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

Fig. 7: Procedures for performing homomorphic operations on secret shares. Here, notation $\langle y \rangle$ is used to represent *this party's* share of the corresponding subtractive secret shared pair. Evaluation maintains the invariant that each of the additive secret shares $\langle y_i \rangle$ encode the correct current computation value of $\hat{y}_i$.

Eval, for a statistical security parameter σ,[8] and outputting the majority value. This approach is also problematic, because each of the σ output bits may leak information about the inputs. Instead, the parties apply the σ instances of Eval locally, and distribute the reconstruction function (computing majority of XORs) using general-purpose secure computation. This ensures that only the correct output is revealed with negligible correctness and secrecy error.

The communication complexity of the above protocol is $n \cdot \mathsf{poly}(\lambda)$, where $n = |x_0| + |x_1|$. This can be improved to $n + \mathsf{poly}(\lambda)$ by using the following hybrid encryption techniques [19]. Let F_r be a pseudorandom function computable in NC^1, which can be based on DDH [30]. Following the key generation phase, each party encrypts a random key r_b for F. Then, instead of separately encrypting each bit of x_b, Party b masks every bit i of its input using $F_{r_b}(i)$ and sends to the other party all of the masked bits. The value of P on the inputs can now be expressed as the value of a (polynomially larger) publicly known branching program P' on the inputs r_0, r_1, where P' is determined by P and the masked inputs. The evaluation of P' is repeated σ times as before. This yields the following:

Theorem 5. *Under the DDH assumption, there exists a constant-round secure 2-party protocol for evaluating branching programs of size S on inputs (x_0, x_1) of total length n, using $n + \mathsf{poly}(\lambda)$ bits of communication.*

4.2 Breaking the Circuit size Barrier for "Well Structured" Circuits

We turn to the question of reducing the communication complexity of evaluating a deep boolean circuit C of size S and depth D. We assume for simplicity that the circuit is *layered* in the sense that its S gates can be partitioned into $D + 1$ layers such that the gates from layer i (except input gates) receive their inputs from gates of layer $i - 1$. This can be generalized to a broader class of "well-structured" circuits that captures most instances of circuits that arise naturally.

Given a layered circuit as above, we divide the layers into intervals of $\lceil \log S \rceil$ consecutive layers, and pick for every interval the layer that has the smallest number of gates (except for the input layer). Overall, we have at most $D/\log S$ "special" layers, whose total size is at most $S/\log S$. In addition, the output layer is considered the last special layer.

The crucial observation is that each output of a new special layer can be expressed as a circuit of depth $O(\log S)$ applied to values of the previous special layer. The protocol will compute the values of the special layers one at a time, by using the previous protocol for branching programs, except that the reconstruction protocol is only applied in the end. That is, given additive shares of special layer i, each party encrypts his shares and the parties apply Eval on a function (computable by polynomial-size

[8] Here we assume that the events of error in different instances of Eval are independent. This can be enforced by using a fresh set of pseudorandom values for each share conversion.

branching programs) that first reconstructs the value and then computes the outputs of special layer $i+1$.
To avoid a multiplicative factor of σ in communication, we need to apply a more efficient error correction procedure for intermediate layers. To this end, we apply an asymptotically good error-correcting code, with encoding and decoding in NC^1, for encoding the values of each special layer. (Many such codes are known to exist; see, e.g., [33]; moreover, by using a Las-Vegas type algorithm for the share conversion it suffices to correct *erasures.*) The computation performed by Eval will start by reconstructing the noisy encoding of layer i (using XOR), then apply a decoder to recover the actual values of layer i, then compute the outputs of layer $i+1$, and then encode these outputs. If the error probability δ of Eval is smaller than the relative error correction radius of the code, the error rate in the encoded output will be within the error-correction radius with overwhelming probability. Thus, we can use a general-purpose protocol for decoding the correct outputs from the shared noisy encoding. This approach yields the following theorem.

Theorem 6. *Under the DDH assumption, there exists a secure 2-party protocol for evaluating any layered boolean circuit of size S and output length m using $O(S/\log S) + m \cdot \mathsf{poly}(\lambda)$ bits of communication.*

4.3 Function Secret Sharing and Generalized PIR

Function Secret Sharing. As discussed in the Introduction, homomorphic secret sharing can be viewed as a "dual" notion of *function secret sharing*, as defined in [6]. In a homomorphic secret sharing scheme for a class of programs $\mathcal{P}$, given a share of a secret input w and a public *program* $P \in \mathcal{P}$, one can locally compute a share of $\Pi(w)$. In a function secret sharing (FSS) scheme for function class $\mathcal{F}$, given a share of a secret function (represented by a "program"') and a public *input* x, one can locally compute a share of $f(x)$. In particular, given a homomorphic secret sharing scheme supporting a class of programs $\mathcal{P}$ containing a *universal program* U, one can directly obtain a FSS scheme for $\mathcal{P}$, by secret sharing a *description* of the secret program $P \in \mathcal{P}$, and then shares of the evaluation of P on an input x can be obtained by homomorphically evaluating the universal program $U_x(\cdot)$ on the given shares. If for each program $P \in \mathcal{P}$ the homomorphic secret sharing scheme produces output error on the evaluation of P with probability δ (over the randomness of the secret sharing), then for each input x in the domain of f, the resulting FSS scheme will also yield an output error with probability δ. Thus, as a corollary of our homomorphic secret sharing scheme, we obtain a DDH-based FSS scheme for branching program with an arbitrary inverse polynomial error. The resulting FSS key size corresponds to the size of a homomorphic secret share of a description of the secret function: namely, a fixed polynomial in the size of the branching program S and security parameter λ.

Private Information Retrieval. A motivating application regime of function secret sharing (and thus our homomorphic secret sharing scheme) is that of 2-server private information retrieval (PIR) for expressive query classes [6]. As we demonstrate, such applications can be achieved with *negligible* error even when starting with FSS with inverse-polynomial error δ. Together with our construction of such δ-FSS, this gives us DDH-based 2-server PIR for queries expressed by branching programs. Useful examples include counting or retrieving matches that are specified by conjunction queries or fuzzy match predicates (e.g., requiring that a document contains at least a given threshold of keywords from a given list).

A (standard) FSS scheme for a program class $\mathcal{P}$ can be used to obtain secure 2-server PIR schemes for classes of queries related to $\mathcal{P}$, via three basic steps. For simplicity, we focus our treatment to querying the count of database entries satisfying a (secret) predicate $f \in \mathcal{F}$.[9] (1) The client generates FSS shares P_0, P_1 of the desired query P and sends one share to each server. (2) The servers locally compute, and reply with, the linear combination $\sum_{x \in DB} P_b(x)$ for database DB (where the output group of P, P_0, P_1 is $\mathbb{Z}_N$ for $N = |DB|$). (3) Then, leveraging the linearity of FSS reconstruction, the client can recover the desired output $\sum_{x \in DB} P(x) = \sum_{x \in DB} P_0(x) + \sum_{x \in DB} P_1(x)$. To extend this approach to δ-FSS, we execute several independent parallel instances of the δ-FSS scheme, and compute the *majority* of the resulting execution outputs. If the .

ACKNOWLEDGEMENTS. We thank an anonymous reviewer for pointing out the relevance of [35].

Research done in part while visiting the Simons Institute for the Theory of Computing, supported by the Simons Foundation and by the DIMACS/Simons Collaboration in Cryptography through NSF grant #CNS-1523467. Supported by ERC starting grant 259426.

Boyle was additionally supported by ISF grant 1709/14 and ERC starting grant 307952. Gilboa was additionally 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). Ishai was additionally supported by ISF grant 1709/14, BSF grant 2012378, a DARPA/ARL SAFEWARE award, NSF Frontier Award 1413955, NSF grants 1228984, 1136174, 1118096, and 1065276. 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 author 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. 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.

[9] Using sketching or coding techniques (e.g., [31, 17]), this approach can be extended to *recovery* of data entries satisfying a hidden predicate.

2. A. Beimel, Y. Ishai, E. Kushilevitz, and I. Orlov. Share conversion and private information retrieval. In *CCC*, pages 258–268, 2012.
3. M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *STOC*, pages 1–10, 1988.
4. D. Boneh, E. Goh, and K. Nissim. Evaluating 2-DNF formulas on ciphertexts. In *Proc. TCC 2005*, pages 325–341, 2005.
5. D. Boneh, S. Halevi, M. Hamburg, and R. Ostrovsky. Circular-secure encryption from decision diffie-hellman. In *Proc. CRYPTO 2008*, pages 108–125, 2008.
6. E. Boyle, N. Gilboa, and Y. Ishai. Function secret sharing. In *Advances in Cryptology - EUROCRYPT*, pages 337–367, 2015.
7. Z. Brakerski and V. Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE.
8. D. Chaum, C. Crépeau, and I. Damgård. Multiparty unconditionally secure protocols (extended abstract). In *STOC*, pages 11–19, 1988.
9. B. Chor and N. Gilboa. Computationally private information retrieval (extended abstract). In *Proceedings of 29th Annual ACM Symposium on the Theory of Computing*, pages 304–313, 1997.
10. B. Chor, N. Gilboa, and M. Naor. Private information retrieval by keywords. *IACR Cryptology ePrint Archive*, 1998:3, 1998.
11. B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, 1998.
12. M. Clear and C. McGoldrick. Multi-identity and multi-key leveled FHE from learning with errors. In *Proc. CRYPTO 2015*, pages 630–656, 2015.
13. R. Cramer, I. Damgård, and Y. Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In *Proc. TCC 2005*, pages 342–362, 2005.
14. Y. Dodis, S. Halevi, R. D. Rothblum, and D. Wichs. Spooky encryption and its applications. *IACR Cryptology ePrint Archive*, 2016:272, 2016. To appear in Crypto 2016.
15. K. Efremenko. 3-query locally decodable codes of subexponential length. In *Proc. STOC*, pages 39–44, 2009.
16. J. Feigenbaum, Y. Ishai, T. Malkin, K. Nissim, M. Strauss, and R. N. Wright. Secure multiparty computation of approximations. In *Proc. ICALP*, pages 927–938, 2001.
17. M. Finiasz and K. Ramchandran. Private stream search at the same communication cost as a regular search: Role of LDPC codes. In *Proc. ISIT*, pages 2556–2560, 2012.
18. C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178, 2009.
19. C. Gentry, J. Groth, Y. Ishai, C. Peikert, A. Sahai, and A. D. Smith. Using fully homomorphic hybrid encryption to minimize non-interative zero-knowledge proofs. *J. Cryptology*, 28(4):820–843, 2015.
20. C. Gentry, A. Sahai, and B. Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *CRYPTO (1)*, pages 75–92, 2013.

21. N. Gilboa and Y. Ishai. Distributed point functions and their applications. In *Proc. EUROCRYPT 14*, pages 640–658, 2014.
22. 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.
23. S. Halevi and V. Shoup. Bootstrapping for helib. In *Proc. EUROCRYPT 2015*, pages 641–670, 2015.
24. Y. Ishai and A. Paskin. Evaluating branching programs on encrypted data. In *Proc. TCC 2007*, pages 575–594, 2007.
25. A. Kiayias, N. Leonardos, H. Lipmaa, K. Pavlyk, and Q. Tang. Optimal rate private information retrieval from homomorphic encryption. *PoPETs*, 2015(2):222–243, 2015.
26. E. Kushilevitz and R. Ostrovsky. Replication is NOT needed: SINGLE database, computationally-private information retrieval. In *Proc. FOCS 97*, pages 364–373, 1997.
27. A. López-Alt, E. Tromer, and V. Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *Proc. STOC 2012*, pages 1219–1234, 2012.
28. P. Mukherjee and D. Wichs. Two round multiparty computation via multi-key FHE. In *Proc. EUROCRYPT 2016*, pages 735–763, 2016.
29. M. Naor and K. Nissim. Communication preserving protocols for secure function evaluation. In *In Proc. STOC*, pages 590–599, 2001.
30. M. Naor and O. Reingold. Number-theoretic constructions of efficient pseudo-random functions. In *Proc. FOCS*, pages 458–467, 1997.
31. R. Ostrovsky and W. Skeith III. Private searching on streaming data. In *Proc. CRYPTO 2005*, pages 223–240, 2005.
32. R. L. Rivest, L. Adleman, and M. L. Dertouzos. On data banks and privacy homomorphisms. In *Foundations of secure computation*, pages 169–179. Academic, New York, 1978.
33. D. A. Spielman. Linear-time encodable and decodable error-correcting codes. *IEEE Trans. Information Theory*, 42(6):1723–1731, 1996.
34. M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. In *Proc. EUROCRYPT 2010*, pages 24–43, 2010.
35. P. C. van Oorschot and M. J. Wiener. Parallel collision search with cryptanalytic applications. *J. Cryptology*, 12(1):1–28, 1999.
36. A. C.-C. Yao. How to generate and exchange secrets (extended abstract). In *Proc. FOCS*, pages 162–167, 1986.
37. S. Yekhanin. Towards 3-query locally decodable codes of subexponential length. In *Proc. STOC*, pages 266–274, 2007.