Breaking the Circuit Size Barrier for Secure Computation Under DDH[*]

Elette Boyle
IDC Herzliya
elette.boyle@idc.ac.il

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

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

September 2, 2016

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 or formula 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 leveled boolean circuits of size S with communication complexity $O(S/\log S)$.
- 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.

Keywords: Secure computation, fully homomorphic encryption, function secret sharing, private information retrieval

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.

[*] A preliminary version of this paper appears in [11].

Since the seminal feasibility results of the 1980s [58, 35, 6, 14], 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 [17, 15], where a client holds i and two or more servers hold x, or in the two-party model [42] under standard cryptographic assumptions. However, progress on extending this to other useful computations has been slow, with several partial results [16, 25, 47, 8, 38] 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.[1]

All this has changed with Gentry's breakthrough on fully homomorphic encryption (FHE) [52, 28]. 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 [36, 23] are still quite slow. Moreover, while there has been significant progress on basing the feasibility of FHE on more standard or different assumptions [55, 12, 30], 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) [10]. 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 multi-server PIR and related problems. FSS schemes for simple classes of functions such as point functions were constructed from one-way functions in [31, 10]. However, a result from [10] 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 [10] 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.

[1] In the homomorphic encryption for branching programs from [38] (see also [41]), 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.

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 [28, 4]. 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. Each individual share 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.[2]

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[3] 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 do not add up to the correct output $P(w)$ is upper bounded by an error parameter $\delta > 0$ which affects the running time of Eval. This probability is over the randomness of the sharing.[4] 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 simulate 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 [28, 4, 46], 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 [29]. Security against

[2] One can always switch between the notions by changing the class $\mathcal{F}$. However, for classes $\mathcal{F}$ that contain *universal functions* [54, 19] and are additionally closed under restrictions, 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.

[3] 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. This type of branching programs captures the complexity class ReachFewL, which contains only few natural problems not known to be computable in deterministic logspace [43, 27]. However, even this limited type of arithmetic branching programs can be useful for improving the asymptotic and concrete efficiency of homomorphic arithmetic computations.

[4] We also realize a stronger variant in which Eval gets an additional input id, and errors in multiple executions of Eval are essentially independent as long as the id's are different.

malicious parties can be achieved in a generic way via a communication-preserving GMW-style compiler [35, 47]. Such a compiler can be based on the existence of collision-resistant hash functions, which again follows from DDH.

Breaking the circuit size barrier for "well structured" circuits. In the case of evaluating general (possibly deep) boolean circuits, we can make the total communication slightly sublinear in the circuit size by breaking the computation into segments of logarithmic depth and homomorphically computing additive shares of the outputs of each segment given additive shares of the inputs. For instance, we can evaluate a *leveled* circuit of size S using $O(S/\log S)$ bits of communication (ignoring lower order additive terms; see Section 4 for a more precise statement). 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. More generally, we can evaluate any "circuit over branching programs" (i.e., a circuit whose gates can evaluate an arbitrary branching program on their inputs) with total communication complexity which is linear in the number of gates and with total computational complexity which is polynomial in the size of the branching programs.

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 [10] 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. [22] under the Learning with Errors (LWE) assumption, by making use of multi-key FHE [45, 18, 46]. 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 [10] 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 secretly 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 natural threshold version of ElGamal [21] and linear secret sharing allows the servers to locally multiply an encrypted input w with an additively secret-shared value x, such that the result $z = wx$ is shared between the servers in the following way: each server b has a group element g^{z_b} such that the product of the two group element 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.

To see how this step is possible, consider a simplified version of the world, where (instead of requiring ElGamal) it holds that g^w is a secure encryption of w. In this world, we can secret share input w by giving both servers a copy of the encryption g^w. Then, given an additive secret sharing $x_0, \ldots, x_{m-1}$ of another value x, the servers can generate a multiplicative sharing of wx, by each computing $(g^w)^{x_i}$. Indeed, $\prod_i g^{wx_i} = g^{wx}$. Extending this idea to ElGamal (as, alas, g^w is not a secure encryption) can be done via comparable "linear algebra in the exponent" given additive shares of x as well as for cx, where c is the ElGamal secret key.

What seems to stop us at degree-2 polynomials is the fact that g^z is now shared *multiplicatively* rather than z being shared 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 perhaps 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 $g_0 = g \cdot g_1$ 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_0 is one step ahead. Conversion is done by picking a pseudo-random δ-sparse[5] subset $\mathbb{G}' \subset \mathbb{G}$ 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 $1/\delta$ steps and if it is not found in $O((1/\delta)\log(1/\delta))$ 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 end up with integers z_0, z_1 such that $z_1 - z_0 = z$, yielding the desired linear sharing of z.

Once we have a linear sharing of z, we can freely add it to 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. As noted previously, each such multiplication requires additive shares not only of z, but also of cz for the ElGamal key c. Therefore, the servers must compute multiplicative shares of g^{cz} and convert them to additive shares of cz.

The third idea is that the client can assist the conversion by also 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. [9] 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, whereas c (and so cz) is a large number. This is handled by providing an encryption of each input w multiplied by each *bit* c_i of the secret key, and applying a linear combination whose coefficients are powers of 2 to the linear shares of the products $c_i wx$ to obtain additive shares of cz for $z = wx$.

These ideas allow the servers to compute a restricted type of "straight-line programs" on the client's input, consisting of a sequence of instructions that either: load an input into memory, add the values of two memory locations, multiply a memory location by an input, or output the value of a memory location modulo some integer $\beta \geq 2$. (Note that we cannot multiply two memory locations, which would allow evaluation of arbitrary circuits.) Such programs, referred to as restricted multiplication straight-line (RMS) 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

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

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 [20], has already been observed in the related contexts of information-theoretic PIR and locally decodable codes [59, 24, 5]. 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 [56], where a set of "distinguished points" is used to synchronize two different processors.

1.3 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 primitives we implement and the assumptions we rely on. We refer the reader to Appendix A for standard definitions and facts about branching programs.

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. We also associate with every program P a *size* parameter, which may be different from the bit-length of the string P. We will sometimes consider programs defined over a larger (non-binary) alphabet; the above definition extends naturally to this case. Finally, we will sometimes allow $P(w)$ to return a special $\perp$ symbol, specifying that the output is undefined. The following "correctness" requirements are automatically satisfied for such a choice of P and w.

While our constructions natively support programs that perform arithmetic computations over (small) integers, the following definitions consider binary inputs and outputs for simplicity. In the arithmetic version of our constructions, described in Section 3, programs include an additional parameter that specifies an upper bound on the size of intermediate values. In the case of evaluating standard (binary) branching programs, this bound can be fixed to 1.

2.1 Homomorphic Secret Sharing

We now define our basic notion of homomorphic secret sharing. We consider the case of 2-out-of-2 secret sharing, where an algorithm Share is used to split a secret $w = (w_1, \ldots, w_n) \in \{0,1\}^n$ into two shares, such that each share computationally hides w. The homomorphic evaluation algorithm Eval is used to locally evaluate a program $P \in \mathcal{P}$ on the two shares, such that the two outputs of Eval add up to $P(w)$ modulo a positive integer β (where $\beta = 2$ by default), except with δ error probability. The running time of Eval is polynomial in the size of P and $1/\delta$.

One could alternatively consider a relaxed notion that only requires the output representation to be "compact," in the sense that its length is independent of the size of $\mathcal{P}$. With such an alternative representation, one could make δ negligible via independent repetition. We require an additive representation as above because it is crucial for some of the applications we consider and is achieved by our constructions. See [10] for further discussion in the context of the dual notion of Function Secret Sharing.

Definition 2.1 (Homomorphic Secret Sharing)**.** A (2-party, $1/\mathsf{poly}$-error) *Homomorphic Secret Sharing (*HSS*)* scheme for a class of programs $\mathcal{P}$ consists of algorithms $(\mathsf{Share}, \mathsf{Eval})$ with the following syntax:

- $\mathsf{Share}(1^\lambda, (w_1, \ldots, w_n))$: On security parameter 1^λ and $(w_1, \ldots, w_n) \in \{0,1\}^n$, the sharing algorithm outputs a pair of shares $(\mathsf{share}_0, \mathsf{share}_1)$. We assume that the input length n is included in each share.
- $\mathsf{Eval}(b, \mathsf{share}, P, \delta, \beta)$: On input party index $b \in \{0,1\}$, share share (which also specifies an input length n), a program $P \in \mathcal{P}$ with n input bits and m output bits, an error bound $\delta > 0$ and integer $\beta \geq 2$, the homomorphic evaluation algorithm 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 algorithm Share is a PPT algorithm, whereas Eval can run in time polynomial in its input length and in $1/\delta$.
The algorithms $(\mathsf{Share}, \mathsf{Eval})$ should satisfy the following correctness and security requirements:

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

$$\Pr[(\mathsf{share}_0, \mathsf{share}_1) \leftarrow \mathsf{Share}(1^\lambda, (w_1, \ldots, w_n)); y_b \leftarrow \mathsf{Eval}(b, \mathsf{share}_b, P, \delta, \beta),\ b = 0, 1 : \\ y_0 + y_1 = P(w_1, \ldots, w_n)] \geq 1 - \delta - \nu(\lambda),$$

  where addition of y_0 and y_1 is carried out modulo β.
- **Security:** Each share keeps the input semantically secure. Formally, for any $b \in \{0,1\}$, pair of polynomial-length input sequences $v^1, v^2, \ldots$ and $w^1, w^2, \ldots$ such that $|v^i| = |w^i|$, and nonuniform polynomial-time distinguisher $\mathcal{A}$, there is a negligible function ν, such that for every positive integer λ we have $|\Pr[\mathcal{A}(V_\lambda^b) = 1] - \Pr[\mathcal{A}(W_\lambda^b) = 1]| \leq \nu(\lambda)$, where V_λ^b (resp., W_λ^b) is obtained by letting $(\mathsf{share}_0, \mathsf{share}_1) \leftarrow \mathsf{Share}(1^\lambda, v^\lambda)$ (resp., $(\mathsf{share}_0, \mathsf{share}_1) \leftarrow \mathsf{Share}(1^\lambda, w^\lambda)$) and outputting share_b.

2.2 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. Moreover, unlike standard homomorphic encryption schemes (and similarly to the above notion of HSS), the "encrypted" output is not represented by a single ciphertext, but rather by a pair of shares that add up to the output.

The corresponding security notion guarantees computational secrecy of an encrypted input given the public key, the ciphertext, and evaluation key of any single server. In a setting consisting of two servers and an arbitrary number of clients, the above security notion implies that inputs contributed by a set of uncorrupted clients remain secure even if one of the two servers colludes with all the remaining clients.

Definition 2.2 (Distributed-Evaluation Homomorphic Encryption)**.** A (2-party, $1/\mathsf{poly}$-error) *Distributed-Evaluation Homomorphic Encryption (DEHE)* for a class of programs $\mathcal{P}$ consists of algorithms $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval})$ with the following syntax:

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

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

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

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

  where addition of y_0 and y_1 is carried out modulo β.
- **Security:** For $b = 0, 1$, the two distribution ensembles $C_0(\lambda)$ and $C_1(\lambda)$ are computationally indistinguishable, where $C_w(\lambda)$ is obtained by letting $(\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \leftarrow \mathsf{Gen}(1^\lambda)$ and outputting $(\mathsf{pk}, \mathsf{ek}_b, \mathsf{Enc}(\mathsf{pk}, w))$.

2.3 Standard Cryptographic Definitions

We start by defining the DDH assumption on which our results rely.

Definition 2.3 (DDH)**.** We say that the Decisional Diffie-Hellman assumption (DDH) holds if there exists a PPT group generator $\mathcal{IG}$ with the following properties. The output of $\mathcal{IG}(1^\lambda)$ is a pair $(\mathbb{G}, g)$, where $\mathbb{G}$ describes a cyclic group of a prime order q (where we use multiplicative notation for the group operation) and g describes a group generator. We assume that q is included in the group description $\mathbb{G}$. We also assume the existence of an efficient algorithm that given $\mathbb{G}$ and descriptions of group elements h_1, h_2 outputs a description of $h_1 h_2$. Finally, we require that for every nonuniform polynomial-time algorithm $\mathcal{A}$ there is a negligible function ε such that

$$\begin{aligned} |\Pr[(\mathbb{G}, g) \leftarrow \mathcal{IG}(1^\lambda); (a, b) \leftarrow \mathbb{Z}_q^2 : \mathcal{A}(\mathbb{G}, g, g^a, g^b, g^{ab}) = 1] - \\ \Pr[(\mathbb{G}, g) \leftarrow \mathcal{IG}(1^\lambda); (a, b, c) \leftarrow \mathbb{Z}_q^3 : \mathcal{A}(\mathbb{G}, g, g^a, g^b, g^c) = 1]| \leq \varepsilon(\lambda) \end{aligned}$$

Public-key encryption and ElGamal encryption. A public-key encryption is defined by PPT algorithms $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$, where the key generation algorithm $\mathsf{Gen}(1^\lambda)$ outputs a key pair $(\mathsf{pk}, \mathsf{sk})$, the encryption algorithm $\mathsf{Enc}(m)$ outputs the encryption of a message m (where we view m as a non-negative integer), and the decryption algorithm $\mathsf{Dec}(\mathsf{sk}, c)$ outputs the decryption of a ciphertext c. We refer the reader to [32] for the standard security definition.

The ElGamal encryption scheme, whose security is based on the DDH assumption, is defined as follows. Let $\mathcal{IG}$ be as promised in Definition 2.3. The key generation algorithm $\mathsf{Gen}_{\mathsf{ElGamal}}(1^\lambda)$ lets $(\mathbb{G}, g) \leftarrow \mathcal{IG}(1^\lambda)$, picks a random $c \in \mathbb{Z}_q$ (recall that $q = |\mathbb{G}|$) and outputs $(\mathsf{pk}, \mathsf{sk})$ for $\mathsf{pk} = (\mathbb{G}, g, g^c)$ and $\mathsf{sk} = c$. The encryption algorithm is defined by $\mathsf{Enc}_{\mathsf{ElGamal}}(\mathsf{pk} = g^c, m) = (g^r, g^m \cdot g^{cr})$ and the decryption algorithm $\mathsf{Dec}_{\mathsf{ElGamal}}(\mathsf{sk}, \mathsf{ct} = (a, b))$ outputs m such that $g^m = b/a^c$ (we assume that $0 \leq m \leq \mathsf{poly}(\lambda)$ so that m can be found in brute-force). We will omit the group $\mathbb{G}$ and the generator g when they are clear from the context.

A more efficient variant of our construction requires 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. We present a simplified (weaker) version of the more general notion of key-dependent security as introduced by Black *et al.* [7].

Definition 2.4 (Circular Security)**.** We say that a public-key encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ with secret key length $\ell(\lambda)$ and message space containing $\{0, 1\}$ is *circular secure* if there exists a negligible function $\nu(\lambda)$ for which the following holds for every nonuniform polynomial-time $\mathcal{A}$:

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

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

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

We remark that circular security implies standard semantic security.

Pseudorandom functions. We will rely on pseudorandom functions [34] mapping (representations of) DDH group elements to bit-strings. We write $\phi \leftarrow \mathsf{PRFGen}(1^\lambda)$ to denote the choice of a key defining a PRF ϕ whose (polynomial-length) domain and range will be specified.

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, programs from this class can simulate deterministic branching programs with constant overhead.

Definition 3.1 (RMS programs)**.** The class of *Restricted Multiplication Straight-line (RMS)* programs consists of a magnitude bound 1^M and an arbitrary sequence of the four following instructions, each with a unique identifier id:

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

If at any step of execution the size of a memory value exceeds the bound M, the output of the program on the corresponding input is defined to be $\perp$. We define the *size* of an RMS program P as the number of its instructions.

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

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

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," bounded by a parameter M (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 2.2)—based on DDH, with ciphertext size $O(\ell)$ group elements per input (where ℓ is 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 evaluation algorithms is a local share conversion procedure 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 procedure DistributedDLog, which receives as input a group element $h \in \mathbb{G}$ and outputs an integer i. Loosely speaking, DistributedDLog outputs the distance on the cycle generated by $g \in \mathbb{G}$ between h and the first $h' \in \mathbb{G}$ such that a pseudo-random function (PRF) outputs 0 on z. Once the PRF is fixed, DistributedDLog is a deterministic procedure and consequently two invocations of the procedure with the same element h result in the same output i. Two invocations of the procedure on inputs h and $h \cdot g^x$ for a small x result, with good probability (over the initial choice of PRF seed), in outputs i and $i - x$ for some $i \in \mathbb{Z}$. Therefore, the DistributedDLog procedure converts a difference of small x in the cycle generated by g in $\mathbb{G}$ to the same difference over $\mathbb{Z}$.

The detailed description of $\mathsf{DistributedDLog}_{\mathbb{G},g}$ is given in Algorithm 1. The procedure is hard-wired with a description of a cyclic group $\mathbb{G}$ and a generator $g \in \mathbb{G}$. $\mathsf{DistributedDLog}_{\mathbb{G},g}$ receives as input a group element $h \in \mathbb{G}$, an allowable error probability δ, a maximum difference bound $M \in \mathbb{N}$ (where the correctness property described above will hold only for integers $0 \leq x < M$), and a pseudo-random function $\phi : \mathbb{G} \to \{0,1\}^{\lceil \log(2M/\delta) \rceil}$. The value $T = [2M \log_e(2/\delta)]/\delta$ corresponds to the bound on how many steps the procedure will take along $\mathbb{G}$ without reaching a "special" point before giving up. In the following proposition, we analyze the correctness of DistributedDLog in a simplified setting where ϕ is a truly random function. (The use of PRF will be incorporated in the subsequent subsections).

Proposition 3.2. *Let $\mathbb{G}$ be a cyclic group of order q. Let $\delta > 0$, $M \in \mathbb{N}$ with $M, T < q$. Let $\mathcal{R}$ be the family of* all *functions $\mathcal{R} = \{\phi : \mathbb{G} \to \{0,1\}^{\lceil \log(2M/\delta) \rceil}\}$. Then, for any $h \in \mathbb{G}$ and $x \leq M$, it holds that*

$$\Pr\left[\phi \leftarrow \mathcal{R} : \mathsf{DistributedDLog}_{\mathbb{G},g}(h, \delta, M, \phi) - \mathsf{DistributedDLog}_{\mathbb{G},g}(h \cdot g^x, \delta, M, \phi) = x\right] \geq 1 - \delta.$$

Proof. Let $\mathsf{Zone}_1 = \{0, \dots, x-1\}$ and $\mathsf{Zone}_2 = \{x, \dots, T\}$. The zones are defined such that $i \in \mathsf{Zone}_1$ when the element $h \cdot g^i$ lies "between" h and $h \cdot g^x$, and $i \in \mathsf{Zone}_2$ when the element $h \cdot g^i$ lies outside ("to the right") of this window but within the range that will be reached by the $\mathsf{DistributedDLog}$ procedure before the timeout condition T. For simplicity of notation, we will denote $\mathbf{0} := 0^{\lceil \log(2M/\delta) \rceil}$.

Given a choice of $\phi \leftarrow \mathcal{R}$, we are in one of three possible cases:

1. $\phi(h \cdot g^i) = \mathbf{0}$ for some $i \in \mathsf{Zone}_1$.
2. $\phi(h \cdot g^i) = \mathbf{0}$ for no $i \in \mathsf{Zone}_1$, and for some $i \in \mathsf{Zone}_2$.
3. $\phi(h \cdot g^i) = \mathbf{0}$ for no $i \in \mathsf{Zone}_1 \cup \mathsf{Zone}_2$.

The second case is the good case for us, where both executions of $\mathsf{DistributedDLog}$ (on input h and $h \cdot g^x$) will output the respective distances to the *same* distinguished point $h \cdot g^i$ with $\phi(h \cdot g^i) = \mathbf{0}$, and hence the respective outputs will be precisely i and $i - x$. By showing that the second case occurs with probability at least $1 - \delta$ over the choice of $\phi \leftarrow \mathcal{R}$ we complete the proof. Since cases 1, 2, and 3 partition the probability space, it suffices to bound the probability of cases 1 and 3 by δ.

Case 1: Over the choice of $\phi \leftarrow \mathcal{R}$, the probability that $\phi(h \cdot g^i) = \mathbf{0}$ for some $i \in \mathsf{Zone}_1$ is at most $1 - (1 - \delta/(2M))^M$ since $|\mathsf{Zone}_1| = M$ and $q > T > M$ (so that $h \cdot g^i \neq h \cdot g^{i'}$ for $i \neq i' \in \mathsf{Zone}_1$). By induction, this value is at most $1 - (1 - M\delta/(2M)) = \delta/2$.

Case 3: Similarly, we have that

$$\begin{aligned} \Pr\left[\phi \leftarrow \mathcal{R} : \phi(h \cdot g^i) \neq \mathbf{0} \ \forall i \in \mathsf{Zone}_1 \cup \mathsf{Zone}_2\right] &\leq \left(1 - \frac{\delta}{2M}\right)^{[2M \log_e(2/\delta)]/\delta} \\ &< e^{-\log_e 2/\delta} = \frac{\delta}{2}. \end{aligned}$$

□

Note that executing $\mathsf{DistributedDLog}$ requires $O(T) = O(M \log(1/\delta)/\delta)$ group operations.

3.2 Homomorphic Secret Sharing

We now construct a simple version of the homomorphic secret sharing scheme, using the procedure $\mathsf{DistributedDLog}$ from the previous section as a sub-routine. The resulting scheme will be a "secret-key" version. Further, the security of the scheme will rely on the assumption that ElGamal encryption is circular secure. These restrictions will be removed in the following subsections.

Consider a DDH group $(\mathbb{G}, g) \leftarrow \mathcal{IG}(1^\lambda)$ of prime order q (see Definition 2.3), and $\ell = \lceil \log_2 q \rceil$. We will use $c = c^{(\ell)}, \dots, c^{(1)}$ to denote bits of an element $c \in \mathbb{Z}_q$ (i.e., $c = \sum_{i=1}^{\ell} 2^{i-1} c^{(i)}$).

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. Namely,

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

Input values w will be initially "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 $\mathbb{Z}_q$ additive secret shares: $\langle y \rangle$ itself, and $\langle cy \rangle$—namely, a secret sharing of the product of y with the

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

Items in which *both* parties receive same value.

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

Items in which each party receives a separate share.

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

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

- $\mathsf{MultShares}\Big(\llbracket x \rrbracket_c, \langle y \rangle, \langle cy \rangle\Big) \to \langle\!\langle xy \rangle\!\rangle$.
  1. Denote $\llbracket x \rrbracket_c = (h_1, h_2) \in \mathbb{G}^2$.
  2. Compute $\langle\!\langle xy \rangle\!\rangle = h_2^{\langle y \rangle} h_1^{-\langle cy \rangle}$.
- $\mathsf{ConvertShares}(b, \langle\!\langle x \rangle\!\rangle, \mathsf{id}, \delta, M) \to \langle x \rangle$, with party identifier $b \in \{0,1\}$, execution identifier id, error parameter δ and max size bound M.
  1. Denote by $\phi' : \mathbb{G} \to \{0,1\}^{\lceil \log(2M/\delta) \rceil}$ the appropriate prefix output of $\phi(\mathsf{id}, \cdot)$.
  2. Let x_b denote the present party b's share of $\langle\!\langle x \rangle\!\rangle$.
     If $b = 1$, then replace $x_b \leftarrow x_b^{-1}$ // i.e., convert so that $x_0/x_1 = g^x$ (not $x_0 x_1 = g^x$)
  3. Let $i_b \leftarrow \mathsf{DistributedDLog}_{\mathbb{G},g}(\langle\!\langle x \rangle\!\rangle, \delta, M, \phi')$.
  4. If $b = 0$, output $-i_0 \pmod q \in \mathbb{Z}_q$. If $b = 1$, output $i_1 \pmod q$. // Additive shares

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

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 also stored as secret shares in this fashion.

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

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 $\mathsf{DistributedDLog}$. Note that our use of "multiplicative" shares $\langle\!\langle z \rangle\!\rangle$ refers here to a sharing of g^z as opposed to z itself (see Figure 1).

Remark 3.3 (Valid vs Random)**.** We emphasize that a "valid" encoding (e.g., $\llbracket x \rrbracket_c, \langle x \rangle$, 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 c), and outputs a level-3 (multiplicative secret sharing) encoding of the product xy. The second pairing procedure, $\mathsf{ConvertShares}$, 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 memory 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.

Remark 3.4 (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 all additive secret shares). 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 $x \in \mathbb{Z}$.

Remark 3.5 (Additive versus "Subtractive" Sharing)**.** We describe our HSS scheme using standard additive (respectively, multiplicative) secret sharing, where $x_0+x_1 = x$ (resp., $h_0 \cdot h_1 = g^x$). However, one can obtain a slightly more efficient version by implementing the scheme with *asymmetric* "subtractive" (resp., "division") secret sharing, where $x_0 - x_1 = x$ (resp., $h_0/h_1 = g^x$). For example, the latter approach removes the need for the multiplicative $\mathbb{Z}_q$-inversion in Step 2 of $\mathsf{ConvertShares}$ together with the additive $\mathbb{Z}_q$-inversion in Step 4.

Before we present the complete homomorphic secret sharing scheme, we first analyze these individual pairing procedures.

Claim 3.6 ($\mathsf{MultShares}$)**.** *Let $x, y \in \mathbb{Z}$ with $0 \le x, y, xy < q$, and $c \in \mathbb{Z}_q$. 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 $xy \in \mathbb{Z}$.*

Proof. Denote the ciphertext $[\![x]\!]_c = (h_1, h_2) \in \mathbb{G}^2$, and shares $\langle y\rangle = (y_0, y_1) \in \mathbb{Z}_q^2$, $\langle cy\rangle = (z_0, z_1) \in \mathbb{Z}_q^2$. Recall that the output of each party $b \in \{0,1\}$ upon execution of $\mathsf{MultShares}$ is the share $\mathsf{output}_b = h_2^{y_b} h_1^{-z_b} \in \mathbb{G}$. Now,

$$\begin{aligned}(\mathsf{output}_0)(\mathsf{output}_1) &= \left(h_2^{y_0}h_1^{-z_0}\right)\left(h_2^{y_1}h_1^{-z_1}\right) = h_2^{y_0+y_1}h_1^{-(z_0+z_1)}\\ &= h_2^{y}h_1^{-cy} \text{ From definition of the additive shares}\\ &= (h_2h_1^{-c})^y\\ &= (g^x)^y \text{ From definition of level-1 CT encoding } (h_1,h_2)\end{aligned}$$

That is, the resulting shares $\mathsf{output}_0, \mathsf{output}_1$ of $\mathsf{MultShares}$ satisfy precisely the multiplicative reconstruction property $\mathsf{output}_0 \cdot \mathsf{output}_1 = g^{xy}$ of a *level-3 encoding* of $xy \mod q$, and thus (since $0 \le xy < q$), of $xy \in \mathbb{Z}$ itself. □

We next analyze the correctness of $\mathsf{ConvertShares}$. Here, a slight subtlety arises regarding the PRF used to define the "δ-sparse" subset of $\mathbb{G}$ within the $\mathsf{DistributedDLog}$ sub-routine. The reason is that the share values on which we run $\mathsf{ConvertShares}$ are the results of partial computations and previous $\mathsf{ConvertShares}$ executions, meaning in particular that they *depend* on the choice of the sampled PRF ϕ. However, this is not an issue due to two reasons: (1) this dependence is efficiently computable given oracle access to the PRF *outputs* (as specified in the algorithm $\mathsf{DistributedDLog}$), and (2) we will explicitly ensure the PRF is never used on the same input twice, by use of unique identifier prefixes id. In such case, the PRF will still act as a random function in each $\mathsf{ConvertShares}$ invocation, and yield the required share conversion correctness guarantee.

The following claim states that for any pair of multiplicative shares $\langle\!\langle x \rangle\!\rangle = (h, h^{-1}g^x)$ generated efficiently as a function of previous calls to $\mathsf{ConvertShares}$ corresponding to *previous* instructions (i.e., with arbitrary identifiers $\mathsf{id}' \neq \mathsf{id}$ where id is the current identifier), the algorithm $\mathsf{ConvertShares}$ will correctly yield *additive* secret shares of x, except with error probability δ.

Claim 3.7 ($\mathsf{ConvertShares}$)**.** *For every nonuniform polynomial-time $\mathcal{A}$ and polynomial p there is a negligible function ν, such that for every $\lambda \in \mathbb{N}$, $\mathsf{id} \in \{0,1\}^\lambda$, $\delta > 0$, and $M \in \mathbb{N}$, where $M, 1/\delta \leq p(\lambda)$, we have*

$$\Pr[\phi \leftarrow \mathsf{PRFGen}(1^\lambda); (h,x) \leftarrow \mathcal{A}^{\mathsf{ConvertShares}(\cdot,\cdot,\mathsf{id}',\cdot,\cdot)}(1^\lambda) : x \in \{0,1,\ldots,M\} \wedge \\ \mathsf{ConvertShares}(0,h,\mathsf{id},\delta,M) + \mathsf{ConvertShares}(1,h^{-1}g^x,\mathsf{id},\delta,M) \neq x] < \delta + \nu(\lambda), \quad (1)$$

where $\mathsf{ConvertShares}$ is executed with PRF ϕ, and where $\mathsf{ConvertShares}(\cdot,\cdot,\mathsf{id}',\cdot,\cdot)$ denotes oracle access for any input with identifier $\mathsf{id}' \neq \mathsf{id}$.

Proof. We first prove the claim assuming ϕ is a truly random function, and then show that a significant deviation would contradict the pseudorandomness of the PRF.

Note that the computation and output of $\mathsf{ConvertShares}(\cdot,\cdot,\mathsf{id}',\cdot,\cdot)$ depends only on the outputs of the PRF ϕ on inputs of the form $(\mathsf{id}',\cdot)$ with prefix id'. If ϕ were a random function, then even conditioned on *all* evaluations of ϕ on inputs of the form $(\mathsf{id}',\cdot)$ for $\mathsf{id}' \neq \mathsf{id}$, the values of ϕ on inputs with the special prefix id are still uniformly distributed. In particular, they are uniformly distributed conditioned on the choice of h and x. This means, by Proposition 3.2, that if $0 \leq x < M$, then with probability at least $1-\delta$ over the choice of ϕ on the relevant inputs (with prefix id),

$$\begin{aligned}\mathsf{ConvertShares}(0,h,\mathsf{id},\delta,M) &+ \mathsf{ConvertShares}(1,h^{-1}g^x,\mathsf{id},\delta,M) \\ &= -\mathsf{DistributedDLog}(h,\delta,M,\phi) + \mathsf{DistributedDLog}(hg^{-x},\delta,M,\phi) \\ &= -(-x) = x.\end{aligned}$$

Now, we claim this condition must hold with probability at least $1-\delta-\nu(\lambda)$ for some negligible function ν when ϕ is instead chosen as a pseudorandom function (with security parameter λ). Suppose there exists a nonuniform polynomial-time $\mathcal{A}$, polynomial p, id sequence $\{\mathsf{id}_\lambda\}_\lambda$ with $\mathsf{id}_\lambda \in \{0,1\}^\lambda$, function δ with $\delta(\lambda) \geq 1/p(\lambda)$ for every $\lambda \in \mathbb{N}$, and function M with integer output with $0 \leq M(\lambda) \leq p(\lambda)$ for every $\lambda \in \mathbb{N}$ such that for every λ the probability expression in Equation (1) is at least $\delta(\lambda)+\epsilon(\lambda)$ for some $\epsilon(\lambda)$. Consider the following PRF adversary $\mathcal{B}$. First, $\mathcal{B}$ runs the algorithm $\mathcal{A}$, answering $\mathcal{A}$'s $\mathsf{ConvertShares}$ queries on inputs of the form $(\cdot,\cdot,\mathsf{id}',\cdot,\cdot)$ by forwarding the underlying PRF calls on input $(\mathsf{id}',\cdot)$ to his PRF challenge oracle. (Note that the evaluation of $\mathsf{ConvertShares}$ depends only on *outputs* of the PRF, and that the running time is polynomial in λ by the bound on M and δ.) As a result, $\mathcal{B}$ receives from $\mathcal{A}$ a pair (h,x). $\mathcal{B}$ then internally evaluates $\mathsf{ConvertShares}(0,h,\mathsf{id},\delta,M)+\mathsf{ConvertShares}(1,h^{-1}g^x,\mathsf{id},\delta,M)$, making the appropriate PRF queries $(\mathsf{id},\cdot)$ to the challenge oracle. If the resulting value is x, $\mathcal{B}$ outputs "random," otherwise $\mathcal{B}$ outputs "pseudorandom." By construction, the advantage of $\mathcal{B}$ in the PRF distinguishing game is at least $\epsilon(\lambda)$, and thus by the security of the PRF, $\epsilon(\lambda)$ must be negligible. □

We present our homomorphic secret sharing scheme Share in Figure 2, and the corresponding homomorphic operations on shares Eval in Figure 3.

We remark that our construction obtains a generalization of the notion of HSS, allowing multiple outputs of the program from possibly different groups $\mathbb{Z}_\beta$ (as specified by the program description P).

Homomorphic Secret Sharing Scheme - $\mathsf{Share}(1^\lambda, w_1, \ldots, w_n)$
Let $\mathsf{AdditiveShare}_q(x)$ for $x \in \mathbb{Z}_q$ return a random pair $\langle x \rangle = (x_0, x_1) \in \mathbb{Z}_q^2$ subject to $x_0 + x_1 = x$. (For $x \in \mathbb{Z}$, this is done for $x \pmod q \in \mathbb{Z}_q$).

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

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

Figure 2: Share generation procedure Share for secret sharing an input w via the homomorphic secret sharing scheme.

Theorem 3.8 (Homomorphic Secret Sharing for RMS)**.** *Assume that ElGamal is circular secure (as per Definition 2.4). Then the scheme* ($\mathsf{Share}, \mathsf{Eval}$) *as specified in Figures 2 and 3 is a secure homomorphic secret sharing scheme for the class of RMS programs.*

In particular, (as per Claim A.2), for the special case of RMS programs with magnitude bound $M = 1$, this implies a construction of HSS for deterministic branching programs.

Corollary 3.9 (HSS for Branching Programs)**.** *Assume that ElGamal is circular secure. Then there exists a secure homomorphic secret sharing scheme for the class of deterministic branching programs.*

Theorem 3.8 follows from the next two correctness and security lemmas.

Lemma 3.10 (Correctness of Eval)**.** *For every polynomial p there exists a negligible ν such that for every positive integer $\lambda \in \mathbb{N}$, input $(w_1, \ldots, w_n) \in \mathbb{Z}^n$, error bound $\delta > 0$, RMS program P (as in Definition 3.1) of input length n, size S, and magnitude bound M, and where $1/\delta, S, M \leq p(\lambda)$, we have*

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

Proof. Let p be a polynomial. We first address the probability of error due to execution of the share conversion procedure $\mathsf{ConvertShares}$. Observe:

- Each call to $\mathsf{ConvertShares}$ within Eval takes place with a *unique* identifier (id, t) (where id is the instruction id, and within a given instruction we use distinct values $t \in \{0, 1, \ldots, \ell\}$).

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

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

Parse share as in Figure 2. Parse P as in Definition 3.1, as a magnitude bound 1^M and sequence of instructions. Take $\delta' = \delta/((\ell+1)MS)$.

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

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

Instruction $(\mathsf{id}, \hat{y}_k \leftarrow \hat{y}_i + \hat{y}_j)$:
    1: Compute $\langle y_k \rangle \leftarrow \langle y_i \rangle + \langle y_j \rangle$, directly on the additive shares (over $\mathbb{Z}_q$).
    2: Compute $\langle cy_k \rangle \leftarrow \langle cy_i \rangle + \langle cy_j \rangle$, directly 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 $c^{(t)} w_i$ in the place of $w_i$
    5:     Compute $\langle\!\langle c^{(t)} w_i y_j \rangle\!\rangle = \mathsf{MultShares}([\![c^{(t)} w_i]\!]_c, \langle y_j \rangle, \langle cy_j \rangle)$.
    6:     Execute $\langle c^{(t)} w_i y_j \rangle = \mathsf{ConvertShares}(b, \langle\!\langle c^{(t)} w_i y_j \rangle\!\rangle, (\mathsf{id}, t), \delta', M, \phi)$.
    7: end for
    8: Compute $\langle cw_i y_j \rangle = \sum_{t=1}^{\ell} 2^{t-1} \langle c^{(t)} w_i y_j \rangle$.
    9: Set $\langle y_k \rangle \leftarrow \langle w_i y_j \rangle$ (from Step 3) and $\langle cy_k \rangle \leftarrow \langle cw_i y_j \rangle$.

Instruction $(\mathsf{id}, \beta, \hat{O}_j \leftarrow \hat{y}_i)$:
    1: If $b = 0$, let $\langle z \rangle \leftarrow \langle y_i \rangle$; otherwise, if $b = 1$, let $\langle z \rangle \leftarrow -\langle y_i \rangle$ be the $\mathbb{Z}_q$ additive inverse.
       // I.e., convert to subtractive secret sharing
    2: Shift $\langle z \rangle$ share by rerandomization offset: $\langle z \rangle \leftarrow \langle z \rangle + \phi(\mathsf{id}, g)$, over $\mathbb{Z}_q$.
       // Note that shifting both shares does not change the shared value in $\mathbb{Z}_q$
    3: Convert share from $\mathbb{Z}_q$ to $\mathbb{Z}_\beta$: $\langle O_j \rangle \leftarrow \langle z \rangle \mod \beta$.
    4: Output $\langle O_j \rangle$.
```

Figure 3: Procedures for performing homomorphic operations on secret shares. Note that we distinguish variables of the straight-line program from the actual values by using $\hat{y}_i$ as opposed to y_i, etc. Here, notation $\langle y \rangle$ is used to represent *this party's* share of the corresponding additive secret sharing. Evaluation maintains the invariant that each of the *additive secret shares* $\langle y_i \rangle$ encode the correct current computation value of $\hat{y}_i$.

- For each $\mathsf{ConvertShares}$ execution, the group elements h, h' input by the respective parties are an efficient function of previous executions of $\mathsf{ConvertShares}$, with different identifiers $(\mathsf{id}', t') \neq (\mathsf{id}, t)$.

Thus Claim 3.7 guarantees there is a negligible ν' such that for every positive integer $\lambda \in \mathbb{N}$, input $w_1, \ldots, w_n \in \mathbb{Z}$, error bound $\delta > 0$ with $1/\delta \leq p(\lambda)$, and RMS program P of size $S \leq p(\lambda)$ and magnitude bound $M \leq p(\lambda)$, that each individual $\mathsf{ConvertShares}$ execution with error parameter δ' results in a correct set of additive secret shares with probability at least $1 - \delta' - \nu'(\lambda)$ for negligible ν'.

Note that the homomorphic evaluation of program P performs at most $S(\ell+1)$ executions of the share conversion procedure, with error parameter $\delta' = \delta/(\ell+1)MS$. Therefore, by a union bound, this implies that as long as no intermediate computation value exceeds the specified magnitude bound M (in which case correctness holds trivially) *no* conversion errors will occur with probability $[\delta' + \nu'(\lambda)](\ell+1)MS = \delta + \nu(\lambda)$ for negligible ν.

Assume, then, that every Share Conversion execution returns without error; that is, that for every such (h, x) we have $\mathsf{ConvertShares}(0, h, \mathsf{id}, \delta, M) + \mathsf{ConvertShares}(1, h^{-1}g^x, \mathsf{id}, \delta, M) = x$. 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 3.6, the shares $\langle\!\langle w_i y_j \rangle\!\rangle$ computed via $\mathsf{MultShares}$ constitute a valid multiplicative sharing of $g^{w_i y_j}$ (a level-3 encoding, as per Figure 1). Since we are in the case where $\mathsf{ConvertShares}$ does not err, the resulting converted *additive* shares $\langle w_i y_j \rangle$ encode exactly the value $w_i y_j \mod q \in \mathbb{Z}$. Since $w_i y_j$ is an intermediate computation value y_k in the evaluation of P, we have $0 \leq w_i y_j \leq M < q$, so that $w_i y_j \mod q = w_i y_j \in \mathbb{Z}$. Thus, invariant (a) holds.

   Consider now $\langle cy_k \rangle$. From precisely the same argument as above (since we are in the case where $\mathsf{ConvertShares}$ does not err), we have for each $t \in [\ell]$ that the computed intermediate value $\langle c^{(t)} w_i y_j \rangle$ is an additive (level-2) encoding of the corresponding value $c^{(t)} w_i y_j \mod q \in \mathbb{Z}$. Since $c^{(t)} \in \{0, 1\}$ and $w_i y_j$ is an intermediate computation value in P we have $0 \leq c^{(t)} w_i y_j \leq M < q$, and so the corresponding shares (v_t^A, v_t^B) further satisfy $(int)(v_t^A - v_t^B) = c^{(t)} w_i y_j \in \mathbb{Z}$.

   Combining the respective values over $t \in [\ell]$, it holds $\left(\sum_{t\in[\ell]} 2^{t-1} v_t^A\right) + \left(\sum_{t\in[\ell]} 2^{t-1} v_t^B\right) = \sum_{t\in[\ell]} 2^{t-1}(v_t^A + v_t^B) = \sum_{t\in[\ell]} 2^{t-1} c^{(t)} \cdot w_i \cdot y_j = cw_i y_j \in \mathbb{Z}_q$. 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 $(int)(y_i^A + y_i^B) = y_i \in \mathbb{Z}$ (where addition is in $\mathbb{Z}_q$), and so $\langle z \rangle = (z^A, z^B)$ for which $(int)(z^A - z^B) = y_i \in \mathbb{Z}$.

   Consider the validity of the output value O_j. Since both parties apply the identical offset $\phi(\mathsf{id}, g) \in \mathbb{Z}_q$ to their shares, we have

   $$(int)\left[(z^A + \phi(\mathsf{id}, g)) - (z^B + \phi(\mathsf{id}, g))\right] = y_i.$$

   Since $0 \leq y_i < M$, then either

   $$(int)\left(z^A + \phi(\mathsf{id}, g)\right) - (int)\left(z^B + \phi(\mathsf{id}, g)\right) = y_i, \tag{2}$$

   or

   $$\begin{aligned} (int)\left(z^A + \phi(\mathsf{id}, g)\right) - (int)\left(z^B + \phi(\mathsf{id}, g)\right) &= y_i - q \qquad (3) \\ \text{and } (int)(z^B + \phi(\mathsf{id}, g)) &\geq q - M. \qquad (4) \end{aligned}$$

   In the case of Equation (2), it holds that

   $$\begin{aligned} (int)\left(z^A + \phi(\mathsf{id}, g)\right) \bmod \beta - (int)\left(z^B + \phi(\mathsf{id}, g)\right) \bmod \beta &= \\ y_i \pmod{\beta}, \end{aligned}$$

   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'.

   Note that the invariants (a), (b) are trivially maintained for future computations, as the shares $\langle y_i \rangle, \langle c y_i \rangle$ themselves are untouched.

□

Lemma 3.11 (Security of Share). *Suppose that ElGamal is a circular secure encryption scheme (as per Definition 2.4). 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, $\mathsf{HonestShare}(1^\lambda, w_1, \ldots, w_n, b)$ defined by

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

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

1. Sample $\mathsf{share}_b \leftarrow \mathsf{HonestShare}(1^\lambda, w_1, \ldots, w_n, b)$, and parse the resulting value as $\mathsf{share}_b = \left\{\phi, \left([\![w_i]\!]_c, \left\{[\![c^{(t)} w_i]\!]_c\right\}_{t \in [\ell]}, \langle w_i \rangle_b, \langle c w_i \rangle_b\right)_{i \in [n]}\right\}$.
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: Distribution $\mathsf{SimCircSec}(1^\lambda, w_1, \ldots, w_n, b)$, consisting of shares as in the previous distribution, but with the encryptions $\llbracket c^{(t)} w_i \rrbracket_c$ replaced by encryptions of 0:

1. Sample $\mathsf{share}_b \leftarrow \mathsf{SimAddShare}(1^\lambda, w_1, \ldots, w_n, b)$, and parse the output as $\mathsf{share}_b = \left\{\phi, \left(\llbracket w_i \rrbracket_c, \{\llbracket c^{(t)} w_i \rrbracket_c\}_{t\in[\ell]}, v_i, v_i'\right)_{i\in[n]}\right\}$.
2. For each $i \in [n]$, sample ℓ ElGamal encryptions of 0: $\llbracket 0_{i,t} \rrbracket_c \leftarrow \mathsf{Enc}_{\mathsf{ElGamal}}(e, 0)$, for $t \in [\ell]$.
3. Output $\left\{\phi, \left(\llbracket w_i \rrbracket_c, \{\llbracket 0_{i,t} \rrbracket_c\}_{t\in[\ell]}, v_i, v_i'\right)_{i\in[n]}\right\}$.

Hybrid 3: Fully simulated shares, as follows.
Distribution $\mathsf{SimShare}(1^\lambda, n, b)$:

1. Sample a PRF $\phi \leftarrow \mathsf{PRFGen}(1^\lambda)$ with input $\{0,1\}^\lambda \times \mathbb{G}$ and output $\{0,1\}^\ell$.
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: $\llbracket 0_{i,t} \rrbracket_c \leftarrow \mathsf{Enc}_{\mathsf{ElGamal}}(e, 0)$, for $t \in \{0, 1, \ldots, \ell\}$.
   (b) Sample two random secret shares $v_i, v_i' \leftarrow \mathbb{Z}_q$.
4. Output $\left\{\phi, \left(\llbracket 0_{i,0} \rrbracket_c, \{\llbracket 0_{i,t} \rrbracket_c\}_{t\in[\ell]}, v_i, v_i'\right)_{i\in[n]}\right\}$.

Lemma 3.11 follows from the following three claims:

Claim 3.12. *For any $w_1, \ldots, w_n \in \mathbb{Z}$ and $b \in \{0,1\}$ the distributions $\mathsf{HonestShare}(1^\lambda, w_1, \ldots, w_n, b)$ and $\mathsf{SimAddShare}(1^\lambda, w_1, \ldots, w_n, b)$ are identically distributed.*

Proof. Follows by the perfect security of the 2-out-of-2 additive secret sharing scheme. □

Claim 3.13. *Assume the circular security of ElGamal. Then for any $w_1, \ldots, w_n \in \mathbb{Z}$ and $b \in \{0,1\}$ the distributions $\mathsf{SimAddShare}(1^\lambda, w_1, \ldots, w_n, b)$ and $\mathsf{SimCircSec}(1^\lambda, w_1, \ldots, w_n, b)$ are computationally indistinguishable.*

Proof. Suppose to the contrary 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 $\mathsf{SimAddShare}(1^\lambda, w_1^*, \ldots, w_n^*, b^*)$, and for $\mathsf{chall} = 1$ as $\mathsf{SimCircSec}(1^\lambda, w_1, \ldots, w_n, b^*)$. Such an adversary can directly be used to break the circular security of the ElGamal encryption.

Indeed, consider an adversary $\mathcal{B}'$ in the circular security game (see Definition 2.4) 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 (including $\mathbb{G}, g$).
2. For every $i \in [n]$, generate an encryption of w_i^*: $\llbracket w_i^* \rrbracket_c \leftarrow \mathsf{Enc}(\mathsf{pk}, w_i^*)$.
3. For every $i \in [n]$ for which $w_i^* \neq 0$, query the Circular Security oracle $\mathcal{O}$, and receive a vector of ElGamal ciphertexts $(\llbracket x_{i,1} \rrbracket_c, \ldots, \llbracket x_{i,\ell} \rrbracket_c)$, 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 $\llbracket x_{i,t} w_i^* \rrbracket_c := (\llbracket x_{i,t} \rrbracket_c)^{w_i^*}$.

4. For every $i \in [n]$ for which $w_i^* = 0$, sample ℓ encryptions of 0: i.e., for each $t \in [\ell]$, take $\llbracket x_{i,t} \rrbracket_c \leftarrow \mathsf{Enc}_{\mathsf{ElGamal}}(\mathsf{pk}, 0)$.
5. Sample a PRF $\phi \leftarrow \mathsf{PRFGen}(1^\lambda)$ with input $\{0,1\}^\lambda \times \mathbb{G}$ and output $\{0,1\}^\ell$.
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 $w = 0$, it holds that $c^{(t)} w_i^* = 0$, and so the corresponding ciphertexts $\llbracket c^{(t)} w_i^* \rrbracket_c$ are encryptions of 0 in both distributions. Consider now $w_i^* \neq 0$. For any $0 < w < q$, 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 any $x \in \mathbb{Z}_q$, the distribution $\{(\llbracket x \rrbracket_c)^w : \llbracket x \rrbracket_c \leftarrow \mathsf{Enc}_{\mathsf{ElGamal}}(\mathsf{pk}, v)\} = \{((g^r)^w, (g^{rc+x})^w) : r_{i,t} \leftarrow \mathbb{Z}_q\}$ is identical to $\{(g^r, g^{rc+wx}) : r \leftarrow \mathbb{Z}_q\} = \{\llbracket wx \rrbracket_c \leftarrow \mathsf{Enc}_{\mathsf{ElGamal}}(\mathsf{pk}, wx)\}$. 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 $\mathsf{SimAddShare}(1^\lambda, w_1^*, \ldots, w_n^*, b^*)$, and if it provided encryptions of 0 then share_{b^*} is distributed precisely as $\mathsf{SimCircSec}(1^\lambda, w_1^*, \ldots, w_n^*, b^*)$. By the assumed circular security of ElGamal, the claim follows. □

Claim 3.14. *For any $w_1, \ldots, w_n \in \mathbb{Z}$ and $b \in \{0,1\}$ the distributions $\mathsf{SimCircSec}(1^\lambda, w_1, \ldots, w_n, b)$ and $\mathsf{SimShare}(1^\lambda, n, b)$ are computationally indistinguishable.*

Proof. Follows by the semantic security of ElGamal, together with a standard hybrid argument (over the n ciphertexts $\llbracket w_i \rrbracket_c$). In particular, the distribution $\mathsf{SimCircSec}(1^\lambda, w_1, \ldots, w_n, b)$ contains all simulatable information aside from the n encryptions $\llbracket w_i \rrbracket_c$. Thus, given a challenge ElGamal public key pk as well as challenge ciphertexts that encrypt either the true message w_i or encrypt 0, a semantic security adversary can generate an output that is distributed either as $\mathsf{SimCircSec}(1^\lambda, w_1, \ldots, w_n, b)$ or $\mathsf{SimShare}(1^\lambda, b)$, depending on the plaintext value underlying the challenge ciphertexts. □

Security of the scheme Share follows.

□

Complexity Analysis. The complexities of the various algorithms are:

- $\mathsf{Share}(1^\lambda, w_1, \ldots, w_n)$ performs $O(n\ell)$ exponentiations in $\mathbb{G}$. The resulting share size is comparable to $O(n\ell)$ group elements of $\mathbb{G}$, corresponding to $(\ell + 1)$ ElGamal ciphertexts (each a constant number of group elements $\mathbb{G}$) and two additive secret sharings (each an element of $\mathbb{Z}_q$) for each of the n inputs.
- $\mathsf{Eval}_{\mathbb{G},g}(\mathsf{share}, P, \delta)$ requires $O(\ell^2 S^2 M/\delta)$ group operations, where $S = |P|$ is the size of the evaluated program and M is its specified magnitude bound. This is because $O(\ell)$ calls to $\mathsf{DistributedDLog}$ with error parameter $\delta' \sim \delta/(\ell M S)$ take place *per step* (or rather, per multiplication step, which in worst case is all S) of the executed program, each of which requires $O(\ell S M/\delta)$ operations.

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 samples an ElGamal key pair, and outputs pk consisting of encryptions $[\![1]\!]_c, \{[\![c^{(t)}]\!]_c\}_{t\in[\ell]}$, and evaluation keys ek_b corresponding to additive secret shares of $\langle c\rangle$. In Enc, a user computes the necessary ciphertexts $[\![w]\!]_c$ and $\{[\![c^{(t)}w]\!]_c\}_{t\in[\ell]}$ for his input w by exponentiating the ciphertexts in pk componentwise by w (i.e., using 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 $[\![w]\!]_c, \{[\![c^{(t)}]\!]_c\}_{t\in[\ell]}$) 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. As before, we state the result for the special case of deterministic branching programs (but can support the wider class of RMS programs with magnitude bound $M > 1$).

Theorem 3.15 (DEHE)**.** *Assume that ElGamal is circular secure (as per Definition 2.4). 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 3.8, 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 3.8.

Security. For a fixed value of $b \in \{0,1\}$, consider the distribution $C_w := \{\mathsf{pk}, \mathsf{ek}_b, \mathsf{Enc}(\mathsf{pk}, w)\}$. We observe that C_w is in fact a subset of the information contained within a (*secret-key*) secret sharing of the input pair $v_1 = 1, v_2 = w$, using the scheme of the previous section.

Namely, as per Figure 2 in the previous section, $\mathsf{share}_b \leftarrow \mathsf{Share}(1^\lambda, v_1 = 1, v_2 = w)$ contains: group description $\mathbb{G}, g$, PRF ϕ, *fresh* ElGamal ciphertexts $[\![1]\!]_c, [\![w]\!]_c, \{[\![c^{(t)}]\!]_c, [\![c^{(t)}w]\!]_c\}_{t\in[\ell]}$ (rerandomized with respect to the published ciphertexts $[\![c^{(t)}]\!]_c$), and additive secret shares $\langle 1\rangle, \langle c\rangle, \langle w\rangle, \langle cw\rangle$. In comparison, the distribution C_w contains: $\mathsf{pk} = (\mathbb{G}, g, [\![1]\!]_c, \{[\![c^{(t)}]\!]_c\}_{t\in[\ell]})$, $\mathsf{ek}_b = (\mathsf{pk}, \langle 1\rangle, \langle c\rangle)$, and $\mathsf{Enc}(\mathsf{pk}, w) = ([\![w]\!]_c, \{[\![c^{(t)}w]\!]_c\}_{t\in[\ell]})$. Thus, by the security of the previous scheme (Theorem 3.8), based on DDH the distributions C_0 and C_1 are indistingiushable.

□

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 $w_1, \dots, w_n$ 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

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

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

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

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

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

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

1. Parse ek as in Gen above; interpret $\hat{1}$ as loaded into memory, via $\langle 1 \rangle, \langle c \rangle$ as given.
2. Parse P as a sequence of instructions (as in Definition 3.1).
3. For each instruction $(\mathsf{id}, \hat{y}_k \leftarrow \hat{y}_i + \hat{y}_j)$, $(\mathsf{id}, \hat{y}_k \leftarrow \hat{w}_i \cdot \hat{y}_j)$, or $(\mathsf{id}, \hat{O}_j \leftarrow \hat{y}_i)$, perform the corresponding sequence of operations as given in Figure 3.
4. For each instruction $(\mathsf{id}, \hat{y}_j \leftarrow \hat{w}_i)$, execute $(\mathsf{id}, \hat{y}_j \leftarrow \hat{w}_i \cdot \hat{1})$.

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

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 in the previous section, yielding a scheme that relies solely on DDH. As in the setting of fully homomorphic encryption, this can be achieved directly in exchange for a multiplicative blowup of the computation depth in the share size, by considering a *leveled* version of the scheme: i.e., replacing the circular encryptions of bits of c under key c by bits of the ith key c_i under the next key c_{i+1} for a depth-length sequence of keys. However, we now demonstrate an alternative approach, which does *not* require increasing the share size with respect to the size of computation.

Our new construction replaces ElGamal encryption with the ElGamal-like cryptosystem of Boneh, Halevi, Hamburg, and Ostrovsky (BHHO) [9], 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 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: Namely, multiplication by the message m in encryption is replaced with multiplication by g^m. (Note that since decryption of such scheme requires taking discrete log, efficient decryption will hold for a polynomial-size message space). Accordingly, we will also view the secret key directly as a bit string $\mathsf{sk}_{\mathsf{BHHO}} = (s_1, \ldots, s_\ell)$ in the place of group elements $(g^{s_1}, \ldots, g^{s_\ell})$ as is typically considered.

Definition 3.16 (BHHO Encryption [9])**.** The BHHO public-key encryption scheme for message space Msg is as follows:

- Key Generation. Sample $(\mathbb{G}, g) \leftarrow \mathcal{IG}(1^\lambda)$. Let $\ell := \lceil 3 \log_2 q \rceil$, where $q = |\mathbb{G}|$. 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 $h = (g_1^{s_1} \cdots g_\ell^{s_\ell})^{-1}$ and define the public and secret keys to be
$$\mathsf{pk}_{\mathsf{BHHO}} := (g_1, \ldots, g_\ell, h), \quad \mathsf{sk}_{\mathsf{BHHO}} = (s_1, \ldots, s_\ell).$$
- Encryption. To encrypt $m \in \mathsf{Msg}$, choose a random $r \leftarrow \mathbb{Z}_q$ and output the ciphertext $(g_1^r, \ldots, g_\ell^r, h^r \cdot g^m)$. (Recall g is a fixed generator of $\mathbb{G}$).
- Decryption. Let $(c_1, \ldots, c_\ell, d)$ be a ciphertext and $\mathsf{sk}_{\mathsf{BHHO}} = (s_1, \ldots, s_\ell)$ a secret key. Output $m \in \mathsf{Msg}$ for which $g^m = d \cdot (c_1^{s_1} \cdots c_\ell^{s_\ell})$.

In [9], Boneh *et al.* proved (in particular) that based on DDH, BHHO encryptions of the secret key bit vector $(s_1, \ldots, s_\ell)$ are indistinguishable from encryptions of the 0 vector $(0, \ldots, 0)$.

Theorem 3.17 (Circular Security of BHHO [9])**.** *Assuming DDH, the BBHO encryption scheme satisfies circular security, as per Definition 2.4.*

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

DDH-Based Notation and Pairing Operations
Let $s = (s_1, \dots, s_\ell) \in \{0,1\}^\ell$.

- **Notation:** $[\![x]\!]_s = (g'_1, \dots, g'_\ell, h') \in \mathbb{G}^{\ell+1}$ for which $g^x = h' \cdot \prod_{t\in[\ell]} (g'_t)^{s_t}$.
  That is, the new level-1 encoding $[\![x]\!]_s$ is a BHHO ciphertext of $x \in \mathbb{Z}_q$ w.r.t. secret key s.
- **Pairing:** $\mathsf{MultShares}\Big([\![x]\!]_s, \langle y\rangle, \{\langle s_t y\rangle\}_{t\in[\ell]}\Big) \to \langle\!\langle xy\rangle\!\rangle$.
  1. Denote $[\![x]\!]_s = (g'_1, \dots, g'_\ell, h') \in \mathbb{G}^{\ell+1}$.
  2. Compute $\langle\!\langle xy\rangle\!\rangle = (h')^{\langle y\rangle} \cdot \prod_{t\in[\ell]} (g'_i)^{\langle s_t y\rangle}$.
- **Pairing:** $([\![x]\!]_s)^y$, for ciphertext $[\![x]\!]_s \in \mathbb{G}^{\ell+1}$ and plaintext $y \in \mathbb{Z}_q$, given public key $e \in \mathbb{G}$.
  1. Denote $[\![x]\!]_s = (g'_1, \dots, g'_\ell, h') \in \mathbb{G}^{\ell+1}$.
  2. Sample a fresh encryption of 0 (for rerandomization): $(g''_1, \dots, g''_\ell, h'') \leftarrow \mathsf{Enc}_{\mathsf{BHHO}}(e, 0)$.
  3. Output $[\![xy]\!]_s := \big((g'_1)^y g''_1, \dots, (g'_\ell)^y g''_\ell, (h')^y h''\big) \in \mathbb{G}^{\ell+1}$.

Figure 5: Modified DDH-based notation and pairing operations, using BHHO [9] encryption.

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 $\langle s_i wx\rangle$ 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 $(s_1, \dots, s_\ell)$ 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 $([\![x]\!]_s)^y$ 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 $\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval}$. We now proceed to analyze the complexity, correctness, and security of the scheme.

Theorem 3.18 (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.*

Proof. Correctness. It suffices to prove the following correctness of the new pairing procedure.

Claim 3.19 (Correctness of DHH-Based $\mathsf{MultShares}$)**.** *For any $x, y \in \mathbb{Z}$ with $0 \le x, y, xy < q$, any valid BHHO ciphertext $[\![x]\!]_s$ with respect to secret key $s = (s_1, \dots, s_\ell) \in \{0,1\}^\ell$, and additive secret shares $\langle y\rangle, \{\langle s_t y\rangle\}_{t\in[\ell]}$, then the output of* $\mathsf{MultShares}([\![x]\!]_s, \langle y\rangle, \{\langle s_t y\rangle\}_{t\in[\ell]})$ *is a valid multiplicative sharing $\langle\!\langle xy\rangle\!\rangle$ of the product $xy \in \mathbb{Z}$.*

Proof. Denote the ciphertext $[\![x]\!]_s = (g'_1, g'_2, \dots, g'_\ell, h') \in \mathbb{G}^{\ell+1}$, and shares $\langle y\rangle = (y_0, y_1) \in \mathbb{Z}_q^2$, $\langle s_t y\rangle = (z_0^{(t)}, z_1^{(t)}) \in \mathbb{Z}_q^2$, for each $t \in [\ell]$. Recall that the output of each party $b \in \{0,1\}$ upon

DDH-Based Distributed-Evaluation HE: Gen and Enc Algorithms

```
$\mathsf{Gen}(1^\lambda)$:
  1. Sample a PRF $\phi \leftarrow \mathsf{PRFGen}(1^\lambda)$ with input $\{0,1\}^\lambda \times \mathbb{G}$ and output $\{0,1\}^\ell$.
  2. BHHO Key Setup:
     (a) Sample a DDH-hard group and generator $(\mathbb{G}, g) \leftarrow \mathcal{IG}(1^\lambda)$. Let $q = |\mathbb{G}|$.
     (b) Sample a BHHO secret key: $s \leftarrow \{0,1\}^\ell$, where $\ell := \lceil 3 \log q \rceil$.
         Let $e = (g_1, \ldots, g_\ell, h = (g_1^{s_1} \cdots g_\ell^{s_\ell})^{-1})$ be the BHHO public key for $s$.
  3. Sample BHHO encryptions:
     (a) The constant $1 \in \mathbb{Z}_q$: let $\llbracket 1 \rrbracket_c \leftarrow \mathsf{Enc}_{\mathsf{BHHO}}(e, 1)$.
     (b) The bits of the secret key $s$: $\forall i \in [\ell]$, let $\llbracket s_i \rrbracket_s \leftarrow \mathsf{Enc}_{\mathsf{BHHO}}(e, s_i)$.
  4. Sample 2-out-of-2 additive secret sharings:
     (a) The constant $1 \in \mathbb{Z}_q$: $\langle 1 \rangle \leftarrow \mathsf{AdditiveShare}_q(1)$. // (Included for notational convenience)
     (b) The bits of the secret key $s$: $\forall i \in [\ell]$, let $\langle s_i \rangle \leftarrow \mathsf{AdditiveShare}_q(s_i)$.
  5. Output $\mathsf{pk} = \big(\mathbb{G}, g, e, \phi, \llbracket 1 \rrbracket_s, \{\llbracket s_i \rrbracket_s\}_{i \in [\ell]}\big)$, $\mathsf{ek}_b = \big(\mathsf{pk}, \langle 1 \rangle, \{\langle s_i \rangle\}_{i \in [\ell]}\big)$.

$\mathsf{Enc}_{\mathbb{G},g}(\mathsf{pk}, w)$:
  1. Compute the following values:
     (a) BHHO encryption of $w \in \mathbb{Z}$: let $\llbracket w \rrbracket_s \leftarrow \mathsf{Enc}_{\mathsf{BHHO}}(e, w)$.
     (b) BHHO encryptions of $(w \cdot s_i) \in \mathbb{Z}$: i.e., for each $i \in [\ell]$, let $\llbracket s_i w \rrbracket_s = (\llbracket s_i \rrbracket_s)^w \in \mathbb{G}^{\ell+1}$,
         as in Figure 5 (note this includes ciphertext rerandomization).
  2. Output $(\llbracket w \rrbracket_s, \{\llbracket s_i w \rrbracket_s\}_{i \in [\ell]})$.
```

Figure 6: DDH-based homomorphic encryption with distributed evaluation, making use of the BHHO cryptosystem. (See notations in Figure 5).

DDH-Based Homomorphic Evaluation - $\mathsf{Eval}_{\mathbb{G},g}(\mathsf{ek}, b, \mathsf{share}, P, \delta)$
Inputs: Evaluation key ek, party identifier $b \in \{0, 1\}$, shared value share, RMS program description P of size $\leq S$, error bound δ.

Parse ek as in Figure 6, and interpret the value $\hat{1}$ as loaded into memory, via $\langle 1 \rangle, \{\langle s_t \rangle\}_{t \in [\ell]}$ as given.
Parse P as in Definition 3.1, as a magnitude bound 1^M and sequence of instructions.
Let $\delta' = \delta/((k+1)MS)$.

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

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

Instruction $(\mathsf{id}, \hat{y}_k \leftarrow \hat{y}_i + \hat{y}_j)$:
    Compute $\langle y_k \rangle \leftarrow \langle y_i \rangle + \langle y_j \rangle$, directly on the additive shares (over $\mathbb{Z}_q$).
    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 $(\mathsf{id}, \hat{y}_k \leftarrow \hat{w}_i \cdot \hat{y}_j)$:
    Let $[\![w_i]\!]_s$ and $\{[\![s_t w_i]\!]_s\}_{t \in [\ell]}$ be the BHHO ciphertexts associated with $w_i$, and let $\langle y_j \rangle$ and
      $\{\langle s_t y_j \rangle\}_{t \in [\ell]}$ be the additive secret shares associated with $y_j$.
    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.
    Execute Share Conversion: $\langle w_i y_j \rangle = \mathsf{ConvertShares}(b, \langle\!\langle w_i y_j \rangle\!\rangle, \delta', M, \phi)$.
    for $t = 1$ to $\ell$ do    // Repeat above process for each $s_t w_i$ in the place of $w_i$
        Compute $\langle\!\langle s_t w_i y_j \rangle\!\rangle = \mathsf{MultShares}([\![s_t w_i]\!]_s, \langle y_j \rangle, \{\langle s_t y_j \rangle\}_{t \in [\ell]})$.
        Execute Share Conversion: $\langle s_t w_i y_j \rangle = \mathsf{ConvertShares}(b, \langle\!\langle s_t w_i y_j \rangle\!\rangle, \delta', M, \phi)$.
    end for
    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 $(\mathsf{id}, \beta, \hat{O}_j \leftarrow \hat{y}_i)$:
    If $b = 0$, let $\langle z \rangle \leftarrow \langle y_i \rangle$; otherwise, if $b = 1$, let $\langle z \rangle \leftarrow -\langle y_i \rangle$ be the $\mathbb{Z}_q$ additive inverse.
      // I.e., convert to subtractive secret sharing
    Shift $\langle z \rangle$ share by rerandomization offset: $\langle z \rangle \leftarrow \langle z \rangle + \phi(\mathsf{id}, g)$, over $\mathbb{Z}_q$.
      // Note that shifting both shares does not change the shared value in $\mathbb{Z}_q$
    Convert share from $\mathbb{Z}_q$ to $\mathbb{Z}_\beta$: $\langle O_j \rangle \leftarrow \langle z \rangle \mod \beta$.
    Output $\langle O_j \rangle$.
```

Figure 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 additive secret sharing. Evaluation maintains the invariant that each of the additive secret shares $\langle y_i \rangle$ encode the correct current computation value of $\hat{y}_i$.

execution of $\mathsf{MultShares}$ is the share $\mathsf{output}_b = (h')^{y_b} \cdot \prod_{t \in [\ell]} (g'_t)^{z_b^{(t)}}$. Now,

$$\begin{aligned}
(\mathsf{output}_0)(\mathsf{output}_1) &= \left((h')^{y_0} \cdot \prod_{t \in [\ell]} (g'_t)^{z_0^{(t)}} \right) \left((h')^{y_1} \cdot \prod_{t \in [\ell]} (g'_t)^{z_1^{(t)}} \right) \\
&= (h')^{y_0 + y_1} \cdot \prod_{t \in [\ell]} (g'_t)^{z_0^{(t)} + z_1^{(t)}} \\
&= (h')^{y} \cdot \prod_{t \in [\ell]} (g'_t)^{s_t y} \quad \text{From definition of the secret shares} \\
&= \left(h' \cdot \prod_{t \in [\ell]} (g'_t)^{s_t} \right)^{y} \\
&= (g^x)^y \quad \text{From definition of level-1 encoding}
\end{aligned}$$

That is, the resulting shares $\mathsf{output}_0, \mathsf{output}_1$ of $\mathsf{MultShares}$ satisfy precisely the multiplicative reconstruction property $\mathsf{output}_0 \cdot \mathsf{output}_1 = g^{xy}$ of a *level-3 encoding* of $xy \mod q$, and thus (since $0 \leq xy < q$) of $xy \in \mathbb{Z}$, as desired. □

Security. By an identical reduction as in the proof of Theorems 3.8, 3.15, the security of $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval})$ holds if we assume circular security of the BHHO cryptosystem. However, from Theorem 3.17, this property is in fact *provably* true under the DDH assumption. The claim follows.

□

The complexity of the resulting scheme corresponds essentially to an extra factor of the security parameter λ as compared to the ElGamal scheme in the previous section (in both size and runtime), coming from the replacement of ElGamal ciphertexts (each consisting of 2 group elements) with BHHO ciphertexts (each consisting of $(\lambda + 1)$ group elements).

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 obtain similar asymptotic efficiency in the presence of malicious parties, one can apply a communication-efficient variant of the GMW compiler that relies on succinct interactive zero-knowledge arguments [35, 47]. Such a compiler can be based on collision-resistant hash functions, whose existence is implied by the DDH assumption.

The model. We refer the reader to [13, 33] for standard definitions of secure computation in the standalone model. Here we will mostly consider protocols in which two PPT parties, Party 0 and Party 1, wish to securely evaluate a given public "program" (e.g., branching program or boolean circuit) on their local inputs (x_0, x_1). We assume without loss of generality that the output is only revealed to Party 0.

In the beginning of the protocol, each party is given its local input and both parties receive common inputs 1^λ (a security parameter) and P (a program). In the end of the protocol, Party 0 should output $P(x_0, x_1)$ except with $\mathsf{negl}(\lambda)$ failure probability. This correctness requirement should hold for any sequence of programs and inputs of size $\mathsf{poly}(\lambda)$. The security requirement is that the view of each individual party in an honest execution of the protocol can be simulated (up to computational indistinguishability) given its input and output alone. That is, there is a PPT simulator SIM_0 such that for every polynomial-length input sequences P^i, x_0^i, x_1^i, the output of $\mathsf{SIM}_0(1^\lambda, P^\lambda, x_0^\lambda, P^\lambda(x_0^\lambda, x_1^\lambda))$ is computationally indistinguishable from the view of Party 0 in the protocol execution on common inputs $1^\lambda, P^\lambda$ and local inputs $(x_0^\lambda, x_1^\lambda)$. Similarly, there is another PPT simulator SIM_1 such that the output of $\mathsf{SIM}_1(1^\lambda, P^\lambda, x_1^\lambda)$ is computationally indistinguishable from the view of Party 1.

We rely on the following feasibility result for the class of programs P consisting of all boolean circuits:

Theorem 4.1 (Yao's protocol [58, 44])**.** *Suppose that a constant-round oblivious transfer protocol exists. Then there exists a constant-round secure 2-party protocol for evaluating any boolean circuit of size S using $S \cdot \mathsf{poly}(\lambda)$ bits of communication.*

We note that a 2-message oblivious transfer protocol can be based on the DDH assumption [48, 1]. Also, while we only defined secure computation for deterministic, single-output functionalities, a similar theorem applies to the more general case of randomized, two-output functionalities.

4.1 Succinct Protocols for Branching Programs

In this section we present secure protocols for branching programs with near-optimal communication complexity. We start with an informal overview and then proceed to a formal description.

4.1.1 Overview

Our protocols for branching programs can be based either on the weaker HSS primitive via the transformation from [10], or can be built more directly from the public-key variant $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval})$ as in Definition 2.2. We present here the latter approach, which gives rise to somewhat simpler protocols with better concrete efficiency. For simplicity, we start by restricting 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.

The simplest protocol proceeds as follows. The two parties run a general-purpose protocol (such as Yao's protocol) to jointly emulate the key generation Gen. In the end of this subprotocol, both parties hold a public key pk and each holds a secret evaluation key ek_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 Eval, for a statistical security parameter σ,[6] and outputting the majority value. However, this modification alone will not suffice, because the existence of errors within the homomorphic evaluation is *dependent* on the intermediate computation values, and as such the σ output bits may leak information about the inputs. Instead, the parties apply the σ instances of Eval locally (as before), and then jointly evaluate the reconstruction function (computing majority of XORs) using general-purpose secure computation. This ensures that

[6] 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.

only the correct output is revealed (and no further information) with negligible correctness and secrecy error.

The communication complexity of the above protocol is $n \cdot \mathsf{poly}(\lambda)$, where $n = |x_0| + |x_1|$ is the combined length of the two parties' inputs. This can be improved to $n + \mathsf{poly}(\lambda)$ by using the following hybrid encryption technique [29]. Let F_r be a pseudorandom function computable in NC^1, which can be based on DDH [49]. (Better concrete efficiency can be obtained by using a polynomial-stretch local pseudorandom generator [37, 2, 3].) Following the key generation phase, each party encrypts a random key r_b for F in the place of its input. Then, instead of encrypting the bits of x_b using Enc, Party b simply masks each bit i of its input using $F_{r_b}(i)$ and sends to the other party the encryption of r_b together with all of the masked bits. The value of program 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.

4.1.2 Technical details

Here and in the following, it will be useful to rely on a variant of DEHE which ensures that when running polynomially many instances of Eval, the events of error are indistinguishable from being independent. (We do not achieve full independence because all instances share a common PRF key used for share conversion.) To keep Eval deterministic, we include an additional input id where the independence condition is guaranteed as long as all values of id are distinct. We provide the formal definition below.

Definition 4.2 (Multi-evaluation DEHE)**.** A (2-party) *Multi-evaluation DEHE* scheme for a class of programs $\mathcal{P}$ consists of algorithms $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval})$ where Gen and Enc have the same syntax as a standard DEHE (see Definition 2.2) and Eval has an additional input id. The algorithms should satisfy the same security requirement as in Definition 2.2 and the following augmented correctness requirement.

Multi-evaluation correctness: For every polynomial m, s and nonuniform polynomial-time distinguisher $\mathcal{A}$ there is a negligible function ν such that the following holds. For every positive integer λ, input $w = (w_1, \ldots, w_n) \in \{0,1\}^n$, programs $P_1, \ldots, P_{m(\lambda)} \in \mathcal{P}$ of size $s(\lambda)$ with input length n, error bound $\delta > 0$, integer $\beta \geq 2$ and distinct identifiers $\mathsf{id}_1, \ldots, \mathsf{id}_{m(\lambda)} \in \{0,1\}^\lambda$, there are error probabilities $p_1, \ldots, p_{m(\lambda)} \leq \delta$ such that the advantage of $\mathcal{A}$ in distinguishing between the outputs of the following two experiments is at most $\nu(\lambda)$:

- **Experiment 1**: Output a bit sequence $\tau_1, \ldots, \tau_{m(\lambda)}$ where $\Pr[\tau_i = 1] = p_i$ and the τ_i are statistically independent.
- **Experiment 2**:
  - $(\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \leftarrow \mathsf{Gen}(1^\lambda)$;
  - $(\mathsf{ct}_1, \ldots, \mathsf{ct}_n) \leftarrow (\mathsf{Enc}(\mathsf{pk}, w_1), \ldots, \mathsf{Enc}(\mathsf{pk}, w_n))$;
  - For $i = 1, \ldots, m(\lambda)$: Let $y_b^i \leftarrow \mathsf{Eval}(\mathsf{id}_i, b, \mathsf{ek}_b, (\mathsf{ct}_1, \ldots, \mathsf{ct}_n), P_i, \delta, \beta)$, $b = 0, 1$; Output 0 if $y_0^i + y_1^i = P_i(w)$ (where addition is modulo β) and 1 otherwise.

Theorem 4.3 (Multi-evaluation DEHE from DDH)**.** *Under the DDH assumption, there exists a multi-evaluation DEHE scheme for the class $\mathcal{P}$ of branching programs.*

Proof sketch. A multi-evaluation DEHE can be obtained by applying the DEHE construction from Section 3 (Figures 6, 7), where the identifier id given as input to Eval is concatenated to the identifier used in $\mathsf{DistributedDLog}$ to ensure that every PRF invocation uses a distinct input. Multi-evaluation correctness follows by observing that if the PRF is replaced by a random function, then the error probabilities in different invocations of Eval are statistically independent (but each individual probability may depend on the input w and the program P_i). □

We note that one can similarly define a multi-evaluation variant of HSS; our HSS construction (which is slightly simpler than the DEHE construction) extends to the multi-evaluation case in a similar way.

We proceed to formally describe the basic version of the secure computation protocol, where each input bit is encrypted separately. Here and in the following, we will cast the protocol in a hybrid model that uses an ideal two-party oracle for performing "simple" secure computation tasks, for which we can afford to use standard general-purpose secure computation protocols. Using suitable composition theorems [13, 33], each oracle call can be emulated by an arbitrary secure computation protocol for the corresponding functionality. For our purposes, oracle calls can always be emulated using Yao's protocol (see Theorem 4.1).

Basic protocol for branching programs:

- Local inputs: For $b = 0, 1$, Party b holds input $x_b = (x_{b,1}, \ldots, x_{b,n_b})$.
- Common inputs: Security parameter 1^λ, a sequence of branching programs $P = (P_1, \ldots, P_m)$, where each P_i has $n = n_0 + n_1$ inputs.
- Output: Party 0 outputs $(P_1(x_0, x_1), \ldots, P_m(x_0, x_1))$.
- Given primitive: Multi-evaluation DEHE $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval})$ for branching programs.

1. Invoke an oracle to compute the following (randomized) two-party functionality: on input 1^λ, let $(\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \leftarrow \mathsf{Gen}(1^\lambda)$ and deliver $(\mathsf{pk}, \mathsf{ek}_0)$ to Party 0 and $(\mathsf{pk}, \mathsf{ek}_1)$ to Party 1;
2. For $b = 0, 1$ and $i = 1, \ldots, n_b$, Party b lets $\mathsf{ct}_{b,i} \leftarrow \mathsf{Enc}(\mathsf{pk}, x_{b,i})$ and sends $\mathsf{ct}_{b,i}$ to Party $1-b$;
3. For $b = 0, 1$, $j = 1, \ldots, m$ and $k = 1, \ldots, \lambda$, Party b lets $y_{b,j,k} \leftarrow \mathsf{Eval}(\mathsf{id} = (j, k), b, \mathsf{ek}_b, (\mathsf{ct}_{0,1}, \ldots, \mathsf{ct}_{0,n_0}, \mathsf{ct}_{1,1}, \ldots, \mathsf{ct}_{1,n_1}), P_j, \delta = 1/3, \beta = 2)$;
4. Invoke an oracle to compute the following two-party functionality:
On inputs $((y_{0,j,k})_{j \in [m], k \in [\lambda]}\ ,\ (y_{1,j,k})_{j \in [m], k \in [\lambda]})$, deliver to Party 0 the output $(y_1, \ldots, y_m)$ where $y_j = \mathsf{MAJ}(y_{0,j,1} \oplus y_{1,j,1}, \ldots, y_{0,j,\lambda} \oplus y_{1,j,\lambda})$.
5. Party 0 outputs $(y_1, \ldots, y_m)$.

Correctness. Since every output bit is computed λ times with error bound $\delta = 1/3$, the multi-evaluation correctness requirement of Definition 4.2 and a Chernoff bound guarantee that, except with $\mathsf{negl}(\lambda)$ probability, at most 0.4λ of the copies of each output bit are evaluated incorrectly. Thus, for $m \leq \mathsf{poly}(\lambda)$, all majority values will be correct except with $\mathsf{negl}(\lambda)$ probability.

Security. For each party b, the simulator SIM_b needs to simulate the entire view of Party b. Without loss of generality, this includes the random coins of Party b and two types of incoming messages: messages from the other party and messages from oracle calls. (Outgoing messages to the other party and oracles are determined by the above.) The simulator SIM_b can proceed by just running the protocol with the other input set to $0^{n_{1-b}}$ and outputting the view of Party b, where SIM_0 replaces the final oracle message by the actual output (which it gets as input for the simulation). The indistinguishability of the simulated view from the actual view follows from the semantic security of ciphertexts encrypted using Enc given pk and ek_b alone and, in the case of SIM_0, the correctness of the protocol (which guarantees that the simulated final oracle message is the same as in the real execution except with $\mathsf{negl}(\lambda)$ probability).

Applying Theorem 4.3, and using Yao's protocol (Theorem 4.1) to implement the oracle calls, we get the following:

Theorem 4.4 (Secure computation of branching programs: basic version)**.** *Under the DDH assumption, there exists a constant-round secure 2-party protocol for evaluating any sequence of m branching programs of total size S on inputs (x_0, x_1) of total length n, using $(n+m) \cdot \mathsf{poly}(\lambda)$ bits of communication and $\mathsf{poly}(\lambda, S)$ computation.*

We now present an asymptotic improvement that compresses the n input ciphertexts using a hybrid encryption technique.

Compressed-input protocol for branching programs:

- Local inputs: For $b = 0, 1$, Party b holds input $x_b = (x_{b,1}, \dots, x_{b,n_b})$.
- Common inputs: Security parameter 1^λ, a sequence of branching programs $P = (P_1, \dots, P_m)$, where each P_i has $n = n_0 + n_1$ inputs. We assume $n \le 2^\lambda$.
- Output: Party 0 outputs $(P_1(x_0, x_1), \dots, P_m(x_0, x_1))$.
- Given primitives:
  - Multi-evaluation DEHE $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval})$ for branching programs;
  - A pseudorandom function $F : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}$ such that $F_r(\alpha) := F(r, \alpha)$ is computed by a $\mathsf{poly}(\lambda)$-time computable branching program P_F.

1. Invoke an oracle to compute the following (randomized) two-party functionality: on input 1^λ, let $(\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \leftarrow \mathsf{Gen}(1^\lambda)$ and deliver $(\mathsf{pk}, \mathsf{ek}_0)$ to Party 0 and $(\mathsf{pk}, \mathsf{ek}_1)$ to Party 1;
2. For $b = 0, 1$, Party b picks a random PRF key $r_b \in \{0,1\}^\lambda$, lets $\mathsf{ct}_{b,i} \leftarrow \mathsf{Enc}(\mathsf{pk}, r_{b,i})$ for $i = 1, \dots, \lambda$, and sends $\mathsf{ct}_{b,i}$ to Party $1 - b$;
3. For $b = 0, 1$ and $i = 1, \dots, n_b$, Party b lets $x'_{b,i} \leftarrow x_{b,i} \oplus F_{r_b}(i)$ (where each index i is represented by a distinct λ-bit string) and sends $x'_{b,i}$ to Party $1 - b$;
4. For $b = 0, 1$ and $j = 1, \dots, m$, Party b lets P'_j be a branching program for the function $f(r_0, r_1) = P_j((x'_{b,i} \oplus F_{r_b}(i))_{b \in \{0,1\}, i \in [n_b]})$;
   // P'_j can be computed in time $\mathsf{poly}(\lambda, |P_j|)$ given λ, P_F, P_j and the n bits $x'_{b,i}$.
5. For $b = 0, 1$, $j = 1, \dots, m$ and $k = 1, \dots, \lambda$, Party b lets $y_{b,j,k} \leftarrow \mathsf{Eval}(\mathsf{id} = (j, k), b, \mathsf{ek}_b, (\mathsf{ct}_{0,1}, \dots, \mathsf{ct}_{0,\lambda}, \mathsf{ct}_{1,1}, \dots, \mathsf{ct}_{1,\lambda}), P'_j, \delta = 1/3, \beta = 2)$;
6. Invoke an oracle to compute the following two-party functionality:
   On inputs $((y_{0,j,k})_{j \in [m], k \in [\lambda]}\ ,\ (y_{1,j,k})_{j \in [m], k \in [\lambda]})$, deliver to Party 0 the output $(y_1, \dots, y_m)$ where $y_j = \mathsf{MAJ}(y_{0,j,1} \oplus y_{1,j,1}, \dots, y_{0,j,\lambda} \oplus y_{1,j,\lambda})$;
7. Party 0 outputs $(y_1, \dots, y_m)$.

Analysis. The correctness and security proofs are similar to those of the basic protocol described above. The only difference is that instead of directly encrypting their input bits, the parties encrypt bits of their secret PRF keys r_b, and use distinct PRF outputs to mask the inputs. This hybrid encryption scheme keeps the inputs of each party semantically secure.

Using Yao's protocol again to implement oracle calls, the communication complexity is dominated by the n bits used for communicating the masked inputs and the $m \cdot \mathsf{poly}(\lambda)$ bits of reconstructing the outputs. Thus, we get the following theorem.

Theorem 4.5 (Secure computation of branching programs: compressed version)**.** *Under the DDH assumption, there exists a constant-round secure 2-party protocol for evaluating any sequence of m branching programs of total size S on inputs (x_0, x_1) of total length n, using $n + m \cdot \mathsf{poly}(\lambda)$ bits of communication and $\mathsf{poly}(\lambda, S)$ computation.*

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

We now show how to use the succinct protocols for branching programs to reduce the communication complexity of securely evaluating deep boolean circuits that cannot be efficiently represented by branching programs.

4.2.1 Overview

Our general approach is to extend the standard model of boolean circuits to a much richer model of circuits in which a single gate can evaluate any branching program on its inputs. For such circuits, we can get a protocol in which the communication complexity is linear in the number of gates (regardless of the size of the branching program labeling each gate) and where only the computation complexity depends (polynomially) on the size of the branching programs. The main technical challenge in achieving this result is to cope with the errors introduced by the share conversion without increasing the asymptotic communication complexity.

Intuitively, one can think of a protocol as above as a general-purpose secure computation protocol in which every log-space (or NC^1) sub-computation is performed with communication cost which is comparable to the total length of the inputs and outputs, independently of the computation time or the circuit size of the sub-computation. Since useful circuits that arise naturally can typically be broken into large NC^1 sub-circuits (think for example of circuits obtained by compiling a C-program), this approach will typically make the communication complexity much smaller than the circuit size.

Below we show that the circuit size barrier can be broken not only for "natural" circuits but even for general classes of circuits. Consider a boolean circuit C of size S and depth D. Assume that the circuit is *leveled* in the sense that its S gates can be partitioned into $D + 1$ levels such that the gates from level i (except input gates) receive their inputs from gates of level $i - 1$.

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

The crucial observation is that each output of a new special level can be expressed as a circuit of depth $O(\log S)$ applied to values of the previous special level. The protocol will compute the values of the special levels 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 level i, each party encrypts his shares using the hybrid encryption scheme from Section 4.1, and the parties apply Eval on a function (computable by polynomial-size branching programs) that first reconstructs the value and then computes shares of the outputs of special level $i + 1$.

To avoid a multiplicative factor of σ in communication, we need to apply a more efficient error correction procedure for intermediate levels. 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 level. Such codes can be explicitly constructed, see, e.g., [53]; moreover, by using a Las-Vegas type variant of the share conversion algorithm (see Section 5) it suffices to correct *erasures*, in which case one can use any explicit asymptotically good *linear* code, such as [40].

The computation performed by Eval will start by reconstructing the noisy encoding of level i (using XOR), then apply a decoder to recover the actual values of level i, then compute the outputs of level $i + 1$, and then encode these outputs. Here it is crucial that a separate instance of Eval is applied for every bit of the output of level $i + 1$, which ensures that the errors are essentially independent. 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 a protocol for leveled circuits whose communication complexity is $O(S/\log S)$ bits, ignoring lower order additive terms.

4.2.2 Technical details

We start by describing a general result for "circuits over branching programs," namely circuits in which each gate can evaluate an arbitrary branching program on its inputs. We then derive a result for a class of "well-structured" (standard) circuits as a corollary.

Definition 4.6 (Circuit over branching programs)**.** A *circuit over branching programs* is a circuit C in which every non-input gate g_i (i.e., node with nonzero in-degree) is labeled by a branching program P_i. The function computed by C is defined naturally by applying the branching program labeling every gate to the values of the inputs of this gate. The *size* of C, denoted by $|\mathsf{C}|$, is the number of nodes, its *weight* is the total size of branching programs P_i labeling its nodes, and its *depth* is the length of the longest path from an input to an output. The depth of gate g_i is the length of the longest path from an input to g_i.

In order to present our error-correction method in a modular way, we define the following fault-tolerant version of circuits over branching programs. Informally, an η-fault-tolerate implementation of f computes the correct output of f with overwhelming probability even if the output of each gate g_i is flipped (independently) with probability $p_i \leq \eta$. Such an implementation may employ a trusted output decoder D that is not subject to faults.

Definition 4.7 (Fault tolerant implementation)**.** Let $f : \{0,1\}^n \to \{0,1\}^m$ and $\eta, \epsilon > 0$. An (η, ϵ)-fault-tolerant implementation of f (or a circuit C computing f) is a pair $(\mathsf{C}', \mathsf{D})$ with the following properties.

- **Syntax:** $\mathsf{C}' : \{0,1\}^n \to \{0,1\}^{m'}$ is a circuit over branching programs and $\mathsf{D} : \{0,1\}^{m'} \to \{0,1\}^m$ is a standard boolean circuit.
- **Fault tolerance:** For every input $x \in \{0,1\}^n$ and probabilities $0 \leq p_i \leq \eta$, where i ranges over all gates of C' except input gates, we have $\Pr[\mathsf{D}(\tilde{\mathsf{C}}'(x)) \neq f(x)] \leq \epsilon$, where $\tilde{\mathsf{C}}'$ is a probabilistic circuit obtained from C' by flipping the output of every gate g_i (independently) with probability p_i.

We now present a general transformation that adds fault tolerance to any circuit over branching programs. The transformation incurs only a constant multiplicative overhead to the size (up to lower order additive terms) and a polynomial overhead to the weight.

Lemma 4.8 (Adding fault tolerance)**.** *There is $\eta > 0$ such that the following holds. For every $\epsilon > 0$ and circuit over branching programs $\mathsf{C} : \{0,1\}^n \to \{0,1\}^m$ of size S, weight W and depth D, there is an (η, ϵ)-fault-tolerant implementation $(\mathsf{C}', \mathsf{D})$ of the function computed by C, where C' has size $O(S + D\log(D/\epsilon) + m\log(m/\epsilon))$, weight $\mathsf{poly}(W, \log 1/\epsilon)$ and depth D, and D has size $O(m\log(m/\epsilon))$. Moreover C' and D can be computed in time $\mathsf{poly}(W, \log 1/\epsilon)$ given C, ϵ.*

Proof. Let $\mathsf{ENC} : \{0,1\}^k \to \{0,1\}^{ck}$ be an explicit, asymptotically good family of linear error-correcting codes with a $\mathsf{poly}(k)$-size $O(\log k)$-depth decoder DEC that can correct a θ-fraction of errors, for constants $c > 1$ and $0 < \theta < 1/4$ that are independent of the block length. Such codes are known to exist (e.g., [53]). By linearity, the encoding function ENC can also be computed by a $\mathsf{poly}(k)$-size $O(\log k)$-depth circuit.

Let $\eta = \theta/2$. Partition the gates of C into $D+1$ levels $L_0, \ldots, L_D$ where gates in L_j are of depth j. Let σ be the smallest integer such that the probability of at least $\theta\sigma$ errors in σ independent trials, where each error occurs with η probability, is at most $\epsilon/(2D)$. By a Chernoff bound, $\sigma = O(\log(D/\epsilon))$. We assume without loss of generality that each level L_j contains at least σ gates. Otherwise pad it with dummy gates, adding at most $O(D\log(D/\epsilon))$ to the size and weight of C.

By definition of depth, all inputs of a gate from L_j belong to levels L_h with $h < j$. The circuit C' is defined as follows. Similarly to C, it has $D+1$ levels $L'_0, \ldots, L'_D$. The input level L'_0 is identical to L_0. For any $1 \leq j \leq D$, we will maintain the invariant that the outputs of gates L'_j in C' encode the outputs $\mathbf{y}_j$ of gates L_j in C. Concretely, we have $|L'_j| = c \cdot |L_j|$ where the value of the i-th gate $g'_{j,i}$ in L'_j is computed as follows:

1. For every level $h < j$ on which level L_j in C depends, apply DEC to decode the values $\mathbf{y}_h$ of L_h from the values $\mathbf{y}'_h$ of L'_h;
2. Apply the branching programs labeling the nodes of L_j to compute the values $\mathbf{y}_j$ of L_j;
3. Let $g'_{j,i} \leftarrow \mathsf{ENC}_i(\mathbf{y}_j)$, where ENC_i is the i-th output bit of ENC.

In addition to gates $g'_{j,i}$ as above, for every output gate $o_{j,i}$ in L_j we add σ' output gates $o'_{j,i,\ell}$ to C', $1 \leq \ell \leq \sigma'$, where the value of each of these gates is obtained by applying only Steps 1 and 2 above and outputting the i-th bit of $\mathbf{y}_j$. That is, the output is encoded via a simple repetition code. The parameter σ' is chosen to be the smallest integer such that the probability of at least $\sigma'/2$ errors in σ' independent trials, where each error occurs with θ probability, is at most $\epsilon/(2m)$. Since $\theta < 1/2$, by a Chernoff bound we have $\sigma' = O(\log(m/\epsilon))$. This completes the description of C'. The output decoder D simply computes the majority value of the σ' copies of each output.

We turn to analyze the efficiency and fault tolerance of the above construction. By closure properties of branching programs (see Appendix A), the computation of each gate $g'_{j,i}$ or output gate $o'_{j,i,\ell}$ from its inputs can be done by a branching program of size $\mathsf{poly}(W, \log 1/\epsilon)$. Thus, C' can be implemented with size $S' = O(S + D\sigma + m\sigma') = O(S + D\log(D/\epsilon) + m\log(m/\epsilon))$, weight $\mathsf{poly}(W, \log 1/\epsilon)$, and depth D, as required. An implementation of D with the specified circuit complexity follows from the fact that majority has linear-size circuits.

To analyze the error probability in the presence of $\eta = \theta/2$ noise probability, note that the output is correct if the following two conditions hold: (1) for every level L'_j of C', less than a θ fraction of the gates are flipped; (2) for every output bit, less than half of its σ' copies are flipped. By the choice of σ we have that (1) occurs except with at most $D \cdot \epsilon/(2D) = \epsilon/2$ probability. By the choice of σ' we have that (2) occurs except with at most $m \cdot \epsilon/(2m) = \epsilon/2$ probability. It follows that the output is incorrect with at most ϵ probability, as required. □

We now use multi-evaluation DEHE for communication-efficient secure computation of fault-tolerant circuits over branching programs. The high level approach is similar to the compressed-input protocol for branching programs, except that every level L'_j is evaluated based on shares of the previous levels L_h, $h < j$. Once the outputs of C' are shared, the decoder D is implemented via general-purpose secure computation. We formally describe the protocol below.

Secure computation of fault-tolerant circuits over branching programs:

- Local inputs: For $b = 0, 1$, Party b holds input $x_b \in \{0,1\}^{n_b}$.
- Common inputs: Security parameter 1^λ, an $(\eta, 2^{-\lambda})$-fault tolerant implementation $(\mathsf{C}', \mathsf{D})$ (over branching programs) of $f : \{0,1\}^{n_0} \times \{0,1\}^{n_1} \to \{0,1\}^m$, where the gates of C' are partitioned into levels $L'_0, \ldots, L'_D$ according to their depth. We assume $|\mathsf{C}'| \leq 2^\lambda$.
- Output: Party 0 outputs $f(x_0, x_1)$.
- Given primitives:
  - Multi-evaluation DEHE $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval})$ for branching programs;
  - A pseudorandom function $F : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}$ such that $F_r(\alpha) \overset{\triangle}{=} F(r, \alpha)$ is computed by a $\mathsf{poly}(\lambda)$-time computable branching program P_F.

1. Invoke an oracle to compute the following (randomized) two-party functionality: on input 1^λ, let $(\mathsf{pk}, (\mathsf{ek}_0, \mathsf{ek}_1)) \leftarrow \mathsf{Gen}(1^\lambda)$ and deliver $(\mathsf{pk}, \mathsf{ek}_0)$ to Party 0 and $(\mathsf{pk}, \mathsf{ek}_1)$ to Party 1;
2. Party b picks a random PRF key $r_b \in \{0,1\}^\lambda$, lets $\mathsf{ct}_{b,i} \leftarrow \mathsf{Enc}(\mathsf{pk}, r_{b,i})$ for $i = 1, \ldots, \lambda$, and sends $\mathsf{ct}_{b,i}$ to Party $1 - b$;

3. Party 0 lets $y_0^0 \leftarrow (x_0, 0^{n_1})$ and Party 1 lets $y_1^0 \leftarrow (0^{n_0}, x_1)$;
   // y_b^h is Party b's share of the values of level h
4. For $h = 1, \ldots, D$: // generate shares of level h from shares of level $h-1$
   (a) Encrypt the shares y_b^{h-1}: For $b = 0, 1$ and $i = 1, \ldots, |L'_{h-1}|$, Party b lets $s_{b,i}^{h-1} \leftarrow y_{b,i}^{h-1} \oplus F_{r_b}(h-1, i)$ and sends $s_{b,i}^{h-1}$ to Party $1-b$;
   (b) Compute the shares y_b^h: For $b = 0, 1$ and $i = 1, \ldots, |L'_h|$, Party b lets P_i^h be a branching program computing the output of the i-th gate in L'_h from (r_0, r_1). This branching program is determined by the branching program labeling the gate and by the encrypted shares $s_{*,*}^{h'}$ for $h' < h$. Party b then lets $y_{b,i}^h \leftarrow \mathsf{Eval}(\mathsf{id} = (h, i), b, \mathsf{ek}_b, (\mathsf{ct}_{0,1}, \ldots, \mathsf{ct}_{0,\lambda}, \mathsf{ct}_{1,1}, \ldots, \mathsf{ct}_{1,\lambda}), P_i^h, \delta = \eta, \beta = 2)$;
5. Invoke an oracle to compute D on the shares of the output gates of C', delivering the output $y \in \{0,1\}^m$ to Party 0.
6. Party 0 outputs y.

Correctness. If Eval were perfect, the protocol would compute correct shares of every wire in the circuit. By the multi-evaluation correctness requirement of Definition 4.2, we can assume (with only a negligible difference in the error probability) that every invocation of Eval incurs an independent error probability $p_i \leq \eta$. It follows from the $(\eta, 2^{-\lambda})$-fault-tolerance of $(\mathsf{C}\,\mathsf{D})$ that the output is correct except with negligible probability.

Security. Consider the messages received by Party b. In Step 2, it gets a semantically secure encryption of a random secret PRF key r_{1-b} (semantic security holds even when given the information received from the oracle in Step 1). The rest of the view does not depend on the PRF keys but only on their encryptions and the PRF outputs. It follows that the view of a party is indistinguishable from its view in a hybrid game where the PRF outputs are replaced by the outputs of a random function. In this game, the message received in each iteration of Step 4a is uniformly random and can therefore be trivially simulated.

Combining the above protocol with our (multi-evaluation) DEHE from DDH, we get the following protocol for fault tolerant circuits over branching programs. Here the class of programs includes fault-all tolerant circuit implementations that are promised to realize some function f with a constant noise parameter $\eta > 0$ and error parameter $\epsilon = 2^{-\lambda}$.

Claim 4.9 (Protocol for fault tolerant circuits over branching programs)**.** *Let $0 < \eta < 1/2$ be a constant. Under the DDH assumption, there exists a secure 2-party protocol for evaluating any $(\eta, 2^{-\lambda})$-fault-tolerant implementation $(\mathsf{C}', \mathsf{D})$ (over branching programs) of some function f on inputs (x_0, x_1) of total length n with $|\mathsf{C}'| + |\mathsf{D}| \cdot \mathsf{poly}(\lambda)$ bits of communication, $\mathsf{poly}(\lambda, W)$ computation and $D + O(1)$ rounds, where D and W are the depth and weight of C', respectively.*

Combining Claim 4.9 with Lemma 4.8 we get the following theorem.

Theorem 4.10 (Protocol for circuits over branching programs)**.** *Under the DDH assumption, there exists a secure 2-party protocol for evaluating any circuit over branching programs $\mathsf{C} : \{0,1\}^n \to \{0,1\}^m$ of size S, weight W, and depth D with $O(S) + (D + m) \cdot \mathsf{poly}(\lambda)$ bits of communication, $\mathsf{poly}(\lambda, W)$ computation, and $D + O(1)$ rounds.*

We note that the above is nontrivial even for standard boolean circuits, since previous DDH-based protocols required $\mathsf{poly}(\lambda)$ bits of communication per gate. As we show below, for natural classes of "well structured" circuits we can use the extra power of circuits over branching programs to make the communication sublinear in the circuit size.

Definition 4.11 (Leveled circuit)**.** Let $C : \{0,1\}^n \to \{0,1\}^m$ be a boolean circuit. We say that C is *leveled* if its gates can be partitioned into levels $L_0, \ldots, L_D$ such that for every $1 \leq i \leq D$ and gate $g_i \in L_i$, the inputs of g_i are in L_{i-1}.

We show how to convert any leveled boolean circuit into a circuit over branching programs that has asymptotically smaller size and polynomial weight.

Lemma 4.12 (Compact circuits over branching programs from leveled circuits)**.** *Let $C : \{0,1\}^n \to \{0,1\}^m$ be a leveled boolean circuit of size S and depth D. Then there is a circuit over branching programs C of size $O(S/\log S) + n + m$, weight $\mathsf{poly}(S)$, and depth $O(D/\log S)$. Moreover, C can be computed in polynomial time from C.*

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

Since two special levels are at most $2 \log S$ far apart, each gate g in a special level can be expressed as a circuit of depth $2 \log S$ in the values of the previous special level (or the inputs for the first special level), and this computation can be performed by a $\mathsf{poly}(S)$-size branching program P_g. We can thus construct C as follows. The nodes include all input gates and all gates in the special levels. Every non-input gate g is labeled by the branching program P_g that computes its value from values of the previous special level (or inputs). The circuit C has size, weight, and depth as required. □

The requirement of being leveled can be relaxed in several ways. For instance, if the inputs of gates in level L_i come from a level L_j such that $i - j$ is bounded by a constant Δ, the proof of Lemma 4.12 can be generalized by picking a sub-interval of Δ consecutive special layers (instead of a single special layer) from each interval. More generally, for any set T of gates in C such that the length of any directed path in C which avoids T is bounded by ℓ, we can convert C into a circuit C over branching programs, where C has size $|T|$ and weight $O(|T| \cdot 2^{\ell})$.

Combining Lemma 4.12 with Theorem 4.10, we get the following theorem.

Theorem 4.13 (Sublinear-communication protocol for leveled circuits)**.** *Under the DDH assumption, there exists a secure 2-party protocol for evaluating any leveled boolean circuit $C : \{0,1\}^n \to \{0,1\}^m$ of size S and depth D using $O(S/\log S) + n + (D + m) \cdot \mathsf{poly}(\lambda)$ bits of communication, $\mathsf{poly}(\lambda, S)$ computation, and $O(D/\log S)$ rounds.*

4.3 Function Secret Sharing

As discussed in the Introduction, homomorphic secret sharing can be viewed as a "dual" notion of *function secret sharing* from [10]. In an HSS 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 $P(w)$. In a function secret sharing (FSS) scheme for a class of programs $\mathcal{P}$, given a share of a secret function represented by program $P \in \mathcal{P}$ and a public *input* x, one can locally compute a share of $P(x)$.

Below we define the notion of FSS, specialized to the case of two parties and additive reconstruction modulo β, as well as the notion of δ-FSS, a relaxation of FSS that allows for an inverse polynomial error probability.

Definition 4.14 (Function Secret Sharing)**.** A (2-party) *Function Secret Sharing (FSS)* scheme for a class of programs $\mathcal{P}$ consists of PPT algorithms $(\mathsf{Gen}, \mathsf{Eval})$ with the following syntax:

- $\mathsf{Gen}(1^\lambda, P)$: On security parameter 1^λ and program $P \in \mathcal{P}$, the key generation algorithm outputs a pair of keys (k_0, k_1). Each key is assumed to include the description of the input and output domains D_P, R_P of P, where we assume that $D_P = \{0,1\}^n$ and $R_P = \mathbb{Z}_\beta$ for some $\beta \geq 2$.
- $\mathsf{Eval}(b, k_b, x)$: On party index $b \in \{0,1\}$, key k_b (specifying input domain $D_P = \{0,1\}^n$ and output domain $R_P = \mathbb{Z}_\beta$), and input $x \in \{0,1\}^n$, the evaluation algorithm outputs a value $y_b \in \mathbb{Z}_\beta$, corresponding to Party b's share of $P(x)$.

The algorithms $(\mathsf{Gen}, \mathsf{Eval})$ should satisfy the following correctness and security requirements:

- **Correctness:** For every positive integer λ, program $P \in \mathcal{P}$ with input domain $\{0,1\}^n$ and output domain $\mathbb{Z}_\beta$, and input $x \in \{0,1\}^n$,

$$\Pr[(k_0, k_1) \leftarrow \mathsf{Gen}(1^\lambda, P); y_b \leftarrow \mathsf{Eval}(b, k_b, x),\ b = 0,1 : y_0 + y_1 = P(x)] = 1,$$

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

- **Security:** Each k_b keeps P semantically secure (except for revealing the input and output domains and the program size). Formally, for every $b \in \{0,1\}$, pair of polynomial-length program sequences $P^1, P^2, \dots$ and $Q^1, Q^2, \dots$ over $\mathcal{P}$ such that P^i and Q^i have the same size, input domain and output domain, and every nonuniform polynomial-time distinguisher $\mathcal{A}$, there is a negligible function ν, such that for every positive integer λ we have $|\Pr[\mathcal{A}(V^b_\lambda) = 1] - \Pr[\mathcal{A}(W^b_\lambda) = 1]| \le \nu(\lambda)$, where V^b_λ (resp., W^b_λ) is obtained by letting $(k_0, k_1) \leftarrow \mathsf{Gen}(1^\lambda, P^\lambda)$ (resp., $(k_0, k_1) \leftarrow \mathsf{Gen}(1^\lambda, Q^\lambda)$) and outputting k_b.

The relaxed notion of δ-FSS is defined similarly, except that Eval has an additional input δ and its running time is polynomial in the input length and $1/\delta$. The correctness requirement is relaxed to hold with at least $1 - \delta - \nu(\lambda)$ probability for a negligible function ν, provided that $|P|, 1/\delta \geq 1/\mathsf{poly}(\lambda)$. A *multi-evaluation* variant of δ-FSS is defined analogously to Definition 4.2.

We now show how to apply universal branching programs for obtaining a δ-FSS for branching programs from our HSS for branching programs.

Theorem 4.15 (δ-FSS for branching programs)**.** *Under the DDH assumption, there exists a δ-FSS scheme for the class of deterministic branching programs.*

Proof. By Theorem 3.18, the DDH assumption implies the existence of an HSS scheme $(\mathsf{Share}_{\mathsf{HSS}}, \mathsf{Eval}_{\mathsf{HSS}})$ for branching programs. Let $U_{S,n}(\hat{P}, x)$ denote a universal branching program for branching programs of size S over n inputs, where $\hat{P}$ denotes the encoding of P as an input for $U_{S,n}$. That is, for every branching program $P : \{0,1\}^n \to \{0,1\}$ of size S and $x \in \{0,1\}^n$ we have $U_{S,n}(\hat{P}, x) = P(x)$. Such a universal branching program can be generated in time $\mathsf{poly}(S, n)$ (see Corollary A.6). Note that the bit-length $|\hat{P}|$ of $\hat{P}$ is determined by S and n.

We define a δ-FSS scheme $(\mathsf{Gen}_{\mathsf{FSS}}, \mathsf{Eval}_{\mathsf{FSS}})$ for branching programs as follows. To share a branching program $P : \{0,1\}^n \to \{0,1\}$ of size S, where the output is viewed as an element of $\mathbb{Z}_\beta$, the algorithm $\mathsf{Gen}_{\mathsf{FSS}}(1^\lambda, P)$ computes the encoding $\hat{P}$ required by $U_{S,n}$, lets $(\mathsf{share}_0, \mathsf{share}_1) \leftarrow \mathsf{Gen}_{\mathsf{HSS}}(1^\lambda, \hat{P})$, and outputs keys (k_0, k_1) where $k_b = (\mathsf{share}_b, \beta)$. The algorithm $\mathsf{Eval}_{\mathsf{FSS}}(b, k_b = (\mathsf{share}_b, \beta), x, \delta)$ returns $\mathsf{Eval}_{\mathsf{HSS}}(b, \mathsf{share}_b, P_x, \delta, \beta)$, where $P_x : \{0,1\}^{|\hat{P}|} \to \{0,1\}$ is the branching program obtained from $U_{S,n}$ by restricting the last n inputs to x (this can be done by just changing the edge labels of nodes that depend on the last n variables to the constants determined by x). Correctness is easy to verify. Security is inherited from the security of the HSS scheme, noting that if P and P' have the same input domain and the same size, then $|\hat{P}| = |\hat{P}'|$. □

A multi-evaluation δ-FSS can be obtained in the natural way by using a multi-evaluation HSS instead of standard HSS.

Finally, we note that δ-FSS can be used to realize a different relaxation of FSS, in which correctness is guaranteed with all but $\mathsf{negl}(\lambda)$ probability, but where reconstruction of the output from the (compact) output shares y_b need not be linear. This can be done by executing λ independent instances of a δ-FSS in parallel with, say, $\delta = 1/3$, and decoding via *majority* of the executions. In fact, using the multi-evaluation variant of δ-FSS, it suffices to execute Gen once and only use multiple executions of Eval.

4.4 Private Information Retrieval

One of the motivating applications of function secret sharing is that of multi-server private information retrieval (PIR) for expressive query classes [10]. Such a PIR protocol allows a client to search a remote database $DB = (x_i)_{i \in N}$, replicated among $k \leq 2$ servers, for entries satisfying a secret predicate P while hiding P from each individual server. Furthermore, the communication complexity is required to be sublinear in the size of DB. This rules out the trivial solution of having the entire database sent to the client. In fact, we will require the communication complexity to be polynomial in $\log N$ and the size of P.

As we demonstrate, such PIR protocols can be achieved with *negligible* error even when starting with δ-FSS that has inverse polynomial error. 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).

For simplicity, we restrict the attention to querying the *count* of database entries satisfying a secret predicate $P \in \mathcal{P}$.[7] We formalize the notion of generalized PIR in this setting, naturally extending standard definitions of PIR [17, 15, 42].

Definition 4.16 (Generalized PIR)**.** A (1-round, 2-server, computational) private information retrieval (PIR) scheme for a class of programs $\mathcal{P}$ is defined by a triple of PPT algorithms $(\mathsf{Q}, \mathsf{A}, \mathsf{R})$ with the following syntax:

- $\mathsf{Q}(1^\lambda, P)$: On security parameter 1^λ and query program $P : \{0,1\}^n \to \{0,1\}$, where $P \in \mathcal{P}$, the query generation algorithm outputs a pair of queries (q_0, q_1). Each query is assumed to include the parameters n, N.
- $\mathsf{A}(1^\lambda, b, q_b, DB)$: On security parameter 1^λ, server index $b \in \{0,1\}$, query q_b (specifying an input length n), and database $DB = (x_1, \ldots, x_N)$, the answer algorithm outputs an answer y_b.
- $\mathsf{R}(y_0, y_1)$: On answers (y_0, y_1), the reconstruction algorithm outputs an integer y.

The algorithms $(\mathsf{Q}, \mathsf{A}, \mathsf{R})$ should satisfy the following correctness and security requirements:

- **Correctness:** For every polynomial p there is a negligible function ν such that for every positive integer λ, query program $P : \{0,1\}^n \to \{0,1\}$ with $P \in \mathcal{P}$, and database $DB = (x_1, \ldots, x_N)$ with entries x_i of length n, where $n, N, |P| \leq p(\lambda)$, we have

$$\Pr[(q_0, q_1) \leftarrow \mathsf{Q}(1^\lambda, P); y_b \leftarrow \mathsf{A}(b, q_b, DB),\ b = 0, 1 : \mathsf{R}(y_0, y_1) = \sum_{i=1}^{N} P(x_i)] \geq 1 - \nu(\lambda).$$

- **Security:** Each query q_b keeps P semantically secure, except perhaps for leaking the input length and the size of P. This is formalized similarly to Definition 4.14.
- **Efficiency:** The total communication complexity, namely $|q_0| + |q_1| + |y_0| + |y_1|$, is bounded by $\mathsf{poly}(\lambda, n, \log N)$ bits.

We now show how to use δ-FSS to construct generalized PIR schemes. We describe the construction using a multi-evaluation variant of δ-FSS for better concrete efficiency; however, a standard δ-FSS suffices at the cost of increasing the communication complexity by at most a factor of λ.

[7] Using sketching or coding techniques (e.g., [50, 26]), this approach can be extended to *recovery* of data entries satisfying a hidden predicate.

Generalized PIR from δ-FSS:

- Given primitive: a multi-evaluation δ-FSS scheme $(\mathsf{Gen}, \mathsf{Eval})$ for a class of programs $\mathcal{P}$; we assume that for every $P : \{0,1\}^n \to \{0,1\}$, with $P \in \mathcal{P}$, and positive integer β, the program P_β obtained by viewing the output of P as an element of $\mathbb{Z}_\beta$ is also in $\mathcal{P}$ and has the same size as P.

1. $\mathsf{Q}(1^\lambda, P)$: let $(q_0, q_1) \leftarrow \mathsf{Gen}(1^\lambda, P_\beta)$ for $\beta = 2^\lambda$; output (q_0, q_1);
   // The input length n of P as well as β are specified in the queries.
2. $\mathsf{A}(1^\lambda, b, q_b, DB = (x_1, \ldots, x_N))$: For $j = 1, \ldots, \lambda$, let $y_{b,j} \leftarrow \sum_{i=1}^{N} \mathsf{Eval}(\mathsf{id} = j, b, q_b, x_i, \delta)$, where $\delta = 1/(3N)$ and summation is carried out modulo β. Output $y_b = (\beta, y_{b,1}, \ldots, y_{b,\lambda})$;
3. $\mathsf{R}(y_0, y_1)$: Output the most common value of $(y_{0,j} + y_{1,j})$ over $j \in [\lambda]$, where addition is modulo β (ties are broken arbitrarily).

Analysis. Since correctness only needs to hold for $N \leq \mathsf{poly}(\lambda)$, we can assume that $N < \beta = 2^\lambda$. By the correctness of the δ-FSS we have that $\Pr[\mathsf{Eval}(\mathsf{id} = j, 0, q_0, x_i, \delta) + \mathsf{Eval}(\mathsf{id} = j, 1, q_1, x_i, \delta) \neq P(x_i)] \leq 1/(3N) + \mathsf{negl}(\lambda)$ for every i, j. By a union bound, it follows that $\Pr[y_{0,j} + y_{1,j} \neq \sum_{i=1}^{N} P(x_i)] \leq 1/3 + \mathsf{negl}(\lambda)$ for $1 \leq j \leq \lambda$. Finally, it follows from the multi-evaluation correctness of the δ-FSS and a Chernoff bound that the majority of the sums $y_{0,j} + y_{1,j}$ are equal to the correct output $\sum_{i=1}^{N} P(x_i)$ except with $\mathsf{negl}(\lambda)$ probability. The security of the above PIR scheme follows directly from that of the underlying δ-FSS.

Combining the above protocol with Theorem 4.15, we get the following theorem.

Theorem 4.17 (Generalized PIR for branching programs)**.** *Under the DDH assumption, there exists a generalized 2-server PIR scheme for the class of deterministic branching programs.*

5 Examples and Optimizations

There are several possible optimizations and trade-offs between computation and communication in our HSS construction. We describe these optimizations and trade-offs and show how they can be applied in two concrete scenarios. We first consider a toy example of a client who computes the AND of n bits $x_1, \ldots, x_n$ by homomorphic secret sharing and then discuss a two-party computation of a formula. Throughout this section we assume the circular security of ElGamal encryption. Forgoing this assumption, similar results can be derived for BHHO encryption.

The communication complexity of our homomorphic secret sharing scheme is dominated by the $\ell + 1$ ElGamal ciphertexts, or $2(\ell + 1)$ group elements encoding each input bit (where $\ell = \lceil \log_2 q \rceil = \lceil \log_2 |\mathbb{G}| \rceil$ for DDH group $\mathbb{G}$). The computation is dominated by running $\mathsf{MultShares}$ and $\mathsf{ConvertShares}$ $\ell + 1$ times for each product of a memory variable and input variable.

$\mathsf{MultShares}$ consists of raising two group elements to the power of a secret share and adding the two results over the group. Given a sliding window implementation of exponentiation, raising a group element to a positive integer smaller than q requires less than $3\ell/2$ group operations and therefore $\mathsf{MultShares}$ requires at most 3ℓ group operations. The computation time of $\mathsf{ConvertShares}$ with target error δ' and maximum difference between two shares M is dominated by $\frac{4M \ln(4/\delta')}{\delta'}$ group operations and PRF evaluations.

Consider the following optimizations and trade-offs.

1. *Ciphertext description reduction.* The first optimization is heuristically secure (or alternatively, secure in the random oracle model) and uses a PRG $G : \{0,1\}^\ell \to \mathbb{G}^{\ell+1}$ to reduce the communication by almost half. Let $G(\sigma) = (g^r, g^{r_1}, \ldots, g^{r_\ell})$ for a seed $\sigma \in \{0,1\}^\ell$. To encode an input bit, instead of sending $\ell + 1$ complete ElGamal ciphertexts, a party will now send (a random) σ and the $\ell + 1$ group elements $(g^{rc+w_i}, g^{r_1 c + w_i c_1}, \ldots, g^{r_\ell c + w_i c_\ell})$,

corresponding to the *second terms* of the prescribed ciphertexts, using the outputs of $G(\sigma)$ implicitly as the first terms. Given this information, each party can locally generate the full $\ell+1$ ciphertexts, and compute as before.

2. *Modified key representation.* A trade-off reducing communication and increasing computation is possible by changing the representation of the key c from $\sum_{j=0}^{\ell-1} c_j 2^j$ in base 2 to $\sum_{j=0}^{\lceil \ell/b \rceil -1} c'_j B^j$ in base $B = 2^b$ for some $b > 1$. Communication complexity and the *number* of MultShares and ConvertShares operations are reduced by a factor of b, as encrypting the ℓ values $\{c_j w_i\}_{j \in [\ell]}$ encoding input w_i is replaced by encrypting the $\lceil \ell/b \rceil$ values $\{c_{j'} w_i\}_{j' \in [\ell']}$. However, in ConvertShares the possible difference M between the shares held by the two parties (equivalently, the size of encoded values) increases from 1 to $2^b - 1$, increasing the computation time by a factor of $2^b - 1$.

3. *Las-Vegas algorithm.* A Las-Vegas type algorithm for share conversion can be used to relax the target error probability and reduce the computation time. ConvertShares potentially induces an error in one of two situations (these are the two error cases in the proof of Proposition 3.2) which can both be identified by the second party. In the proposed optimization, the second party outputs a flag indicating failure in each of these cases. In applications that involve a client and two servers the client is sure that the result is correct if the second player does not return a failure. Given target error probabilities δ for the whole protocol and, e.g., 1/4 for a single execution, we require that the number of independent executions of the algorithm γ satisfies $(1/4)^\gamma < \delta$, or $\gamma > \frac{\ln 1/\delta}{\ln 4}$. Note that this optimization may reveal information on intermediate computation values, since errors are input dependent. However, this type of leakage is harmless for applications like PIR. On the other hand, for applications such as two-party computation, revealing intermediate values in this way is unacceptable.

4. *Breaking computation into chunks.* The final trade-off increases communication and decreases computation by breaking the computation into "chunks" and encrypting (and communicating) the input to each chunk separately. Loosely speaking, if the computation is split into ζ chunks, then the required communication increases by a factor of ζ, and computation is reduced by a factor of ζ because the quadratic overhead in computing n gates is reduced to ζ times computing a quadratic overhead in n/ζ gates. In general this method requires up to ζ communication rounds, but in certain applications (like PIR for NC^1 predicates) it does not require additional interaction.

HOMOMORPHIC n-BIT AND. In the first (toy) example application without applying any optimization, the communication complexity is dominated by $2(\ell+1)n$ group elements, to encode n bits. The operations are $n-1$ homomorphic evaluations of AND of bits, which amount to less than $n\ell$ applications of MultShares and ConvertShares or a total of less than $3n\ell^2 + \frac{4n^2\ell \ln 4n/\delta}{\delta}$ group operations and PRF evaluations.

In this example, communication is minimized by using the ciphertext reduction optimization and by representing c in base $B = 2^b$. Communication complexity is about $\frac{(\ell+1)n}{b}$ group elements and computation is dominated by $\frac{3n\ell^2}{b^2} + \frac{4Bn^2\ell \ln 4n/\delta}{b\delta}$ group operations and PRF evaluations.

Computation is minimized using the Las-Vegas algorithm and breaking into chunks. The communication complexity is increased by a factor of ζ for each of the $\lceil \frac{\ln 1/\delta}{\ln 4} \rceil$ invocations of the Las Vegas algorithm or $\lceil \frac{\ln 1/\delta}{\ln 4} \rceil \zeta n\ell$ group elements altogether. The computation requires at most

$$\left\lceil \frac{\ln 1/\delta}{\ln 4} \right\rceil \left(3n\ell^2 + \frac{16n^2\ell \ln(16n/\zeta)}{\zeta} \right)$$

group operations (and PRF evaluations) altogether.

2PC formula evaluation. The second example is a two-party computation of a formula ψ. This application requires the public-key variant of our protocol. The unoptimized version of this protocol is roughly similar in performance to the unoptimized version of homomorphic secret sharing. However, two of the optimizations, ciphertext reduction and the Las-Vegas algorithm do not apply in this case. Communication can be minimized by representing c in base B, reducing communication by $\log B$ and increasing computation by B compared to the unoptimized version. Computation can be minimized by breaking ψ into ζ chunks increasing communication by ζ and reducing computation by ζ compared to the unoptimized version.

Acknowledgements. We thank an anonymous reviewer for pointing out the relevance of [56] and Eric Allender for useful pointers regarding the power of the class ReachFewL.

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.

The first author was additionally supported by ISF grant 1709/14 and ERC starting grant 307952. The second author 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). The third author 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] W. Aiello, Y. Ishai, and O. Reingold. Priced oblivious transfer: How to sell digital goods. In *EUROCRYPT*, pages 119–135, 2001.

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

[3] B. Applebaum and S. Lovett. Algebraic attacks against random local functions and their countermeasures. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 1087–1100, 2016.

[4] 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.

[5] A. Beimel, Y. Ishai, E. Kushilevitz, and I. Orlov. Share conversion and private information retrieval. In *CCC*, pages 258–268, 2012.

[6] 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.

[7] J. Black, P. Rogaway, and T. Shrimpton. Encryption-scheme security in the presence of key-dependent messages. In *Selected Areas in Cryptography, 9th Annual International Workshop, SAC 2002, St. John's, Newfoundland, Canada, August 15-16, 2002. Revised Papers*, pages 62–75, 2002.

[8] D. Boneh, E. Goh, and K. Nissim. Evaluating 2-DNF formulas on ciphertexts. In *Proc. TCC 2005*, pages 325–341, 2005.

[9] D. Boneh, S. Halevi, M. Hamburg, and R. Ostrovsky. Circular-secure encryption from decision diffie-hellman. In *Proc. CRYPTO 2008*, pages 108–125, 2008.

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

[11] E. Boyle, N. Gilboa, and Y. Ishai. Breaking the circuit size barrier for secure computation under DDH. In *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part I*, pages 509–539, 2016.

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

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

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

[15] 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.

[16] B. Chor, N. Gilboa, and M. Naor. Private information retrieval by keywords. *IACR Cryptology ePrint Archive*, 1998:3, 1998.

[17] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, 1998.

[18] M. Clear and C. McGoldrick. Multi-identity and multi-key leveled FHE from learning with errors. In *Proc. CRYPTO 2015*, pages 630–656, 2015.

[19] S. A. Cook and H. J. Hoover. A depth-universal circuit. *SIAM J. Comput.*, 14(4):833–839, 1985.

[20] 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.

[21] Y. Desmedt and Y. Frankel. Threshold cryptosystems. In *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, pages 307–315, 1989.

[22] Y. Dodis, S. Halevi, R. D. Rothblum, and D. Wichs. Spooky encryption and its applications. In *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part III*, pages 93–122, 2016.

[23] L. Ducas and D. Micciancio. FHEW: bootstrapping homomorphic encryption in less than a second. In *EUROCRYPT 2015, Proceedings, Part I*, pages 617–640, 2015.

[24] K. Efremenko. 3-query locally decodable codes of subexponential length. In *Proc. STOC*, pages 39–44, 2009.

[25] 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.

[26] 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.

[27] B. Garvin, D. Stolee, R. Tewari, and N. V. Vinodchandran. ReachFewL = ReachUL. *Comput. Complex.*, 23(1):85–98, Mar. 2014.

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

[29] C. Gentry, J. Groth, Y. Ishai, C. Peikert, A. Sahai, and A. D. Smith. Using fully homomorphic hybrid encryption to minimize non-interactive zero-knowledge proofs. *J. Cryptology*, 28(4):820–843, 2015.

[30] 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.

[31] N. Gilboa and Y. Ishai. Distributed point functions and their applications. In *Proc. EUROCRYPT 14*, pages 640–658, 2014.

[32] O. Goldreich. *Foundations of Cryptography — Basic Tools.* Cambridge University Press, 2001.

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

[34] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *J. ACM*, 33(4):792–807, 1986.

[35] 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.

[36] S. Halevi and V. Shoup. Bootstrapping for helib. In *Proc. EUROCRYPT 2015*, pages 641–670, 2015.

[37] Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. Cryptography with constant computational overhead. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17-20, 2008*, pages 433–442, 2008.

[38] Y. Ishai and A. Paskin. Evaluating branching programs on encrypted data. In *Proc. TCC 2007*, pages 575–594, 2007.

[39] S. Jukna. *Boolean Function Complexity - Advances and Frontiers*, volume 27 of *Algorithms and combinatorics*. Springer, 2012.

[40] J. Justesen. Class of constructive asymptotically good algebraic codes. *IEEE Trans. Information Theory*, 18(5):652–656, 1972.

[41] 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.

[42] E. Kushilevitz and R. Ostrovsky. Replication is NOT needed: SINGLE database, computationally-private information retrieval. In *Proc. FOCS 97*, pages 364–373, 1997.

[43] K. Lange. An unambiguous class possessing a complete set. In *Proc. STACS 97*, pages 339–350, 1997.

[44] Y. Lindell and B. Pinkas. A proof of security of yao's protocol for two-party computation. *J. Cryptology*, 22(2):161–188, 2009.

[45] 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.

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

[47] M. Naor and K. Nissim. Communication preserving protocols for secure function evaluation. In *In Proc. STOC*, pages 590–599, 2001.

[48] M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *SODA*, pages 448–457, 2001.

[49] M. Naor and O. Reingold. Number-theoretic constructions of efficient pseudo-random functions. In *Proc. FOCS*, pages 458–467, 1997.

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

[51] N. Pippenger. Simultaneous resource bounds. In *Proc. FOCS 79*, pages 307–311, 1979.

[52] 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.

[53] D. A. Spielman. Linear-time encodable and decodable error-correcting codes. *IEEE Trans. Information Theory*, 42(6):1723–1731, 1996.

[54] L. G. Valiant. Universal circuits (preliminary report). In *Proc. STOC 1976*, pages 196–203, 1976.

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

[56] P. C. van Oorschot and M. J. Wiener. Parallel collision search with cryptanalytic applications. *J. Cryptology*, 12(1):1–28, 1999.

[57] I. Wegener. *Branching Programs and Binary Decision Diagrams.* Society for Industrial and Applied Mathematics, 2000.

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

[59] S. Yekhanin. Towards 3-query locally decodable codes of subexponential length. In *Proc. STOC*, pages 266–274, 2007.

A Branching programs

In this section we review some standard definitions of Branching Programs (BP) and prove several simple properties of these programs. We refer the interested reader to [57, 39] for a comprehensive study of branching programs, their various flavors and their computational power. Branching programs generalize several interesting computational models. Some very useful special cases of branching programs include finite automata, decision trees, and OBDDs.

We wish to show that RMS programs imply branching programs and therefore our results on RMS carry over to BP. Given the well known inclusion of NC^1 circuits in non-uniform log space and the equivalence of non-uniform log-space and BP we deduce that our HSS protocol applies to log space computation and NC^1 circuits.

We then present two additional results on BP. The first shows how to compose several branching programs that each compute a binary-valued function to compute the composed function. We then construct a universal branching program for branching programs of a given size. The universal program UBP accepts as input a branching program P and an input x for P and reaches an accepting state if and only if P accepts when it runs on input x.

We start by defining deterministic branching programs and their oblivious variant.

Definition A.1 (Branching program)**.** A deterministic branching program over the variables $x = (x_1, \ldots, x_n)$ with input domain $\{0, 1\}$ is defined by a tuple $P = (G = (V, E), v_0, T, \psi_v, \psi_e)$ where:

- G is a directed acyclic graph.
- $v_0 \in V$ is an initial node of in-degree 0. We assume without loss of generality that every $u \in V - \{v_0\}$ is reachable from v_0.
- $T = \{v_a, v_r\} \subseteq V$ is a set of two terminal nodes of out-degree 0. v_a is called the accept state and v_r is called the reject state.
- $\psi_v : V \setminus T \to [n]$ is a node labeling function assigning a variable index from $[n]$ to every non-terminal level.

- $\psi_e : E \to 2^{\{0,1\}}$ is an edge labeling function, such that every edge is mapped to a non-empty set, and for every node v the sets labeling its outgoing edges form a partition of $\{0,1\}$.

We say that P is a (leveled and) *oblivious* BP if V can be partitioned into $\ell+1$ disjoint levels $V = \bigcup_{i=0}^{\ell} V_i$, such that $V_0 = \{v_0\}$, $V_\ell = T$, and the node labeling function ψ_v assigns the same variable index to the nodes in each level. In this case we can write ψ_v as $\psi_v : \{V_0, V_1, \ldots, V_{\ell-1}\} \to [n]$. The *size* of a BP P, denoted by $|P|$, is the number of nodes. For an oblivious BP, we say that it has length ℓ and width w if the number of levels is $\ell+1$ and the maximum number of nodes in a level is w. Finally, a *canonical oblivious* BP with parameters n, w, d is an oblivious BP of width w and length $\ell = dn$ over n inputs, where $\psi_v(i) = i \mod n+1$ (i.e., the variables are read in the order $x_1, \ldots, x_n, x_1, \ldots, x_n, \ldots$ and each variable is read d times).

BP EVALUATION. For a program P and an input x, the successor of node u is the unique node u' such that $x_{\psi_v(u)} \in \psi_e(u, u')$. P accepts x if and only if the path from v_0 defined by the successor relation reaches v_a. This defines the function $f : \{0,1\}^n \to \{0,1\}$ computed by P, where $f(x) = 1$ iff P accepts x.

Every boolean formula of size S over AND, OR, and NOT gates has an equivalent branching program of size $S+3$ [57]. It follows that every NC^1 function can be computed by polynomial-size branching programs.

We show that a Restricted Multiplication Straight-line (RMS) program can efficiently simulate a branching program.

Claim A.2. *(BP to RMS) For any deterministic branching program P of size $|P|$ over input $x \in \{0,1\}^n$ there exists an (efficiently computable) RMS R of size $O(|P|)$ with magnitude bound $M = 1$ which outputs 1 if P accepts and outputs 0 otherwise.*

Proof. R has the same inputs $x_1, \ldots, x_n$ as P and uses a memory cell $\hat{y}_v$ for each node v in P. The first instructions of the program load 1 to $\hat{y}_{v_0}$ and load 0 to all other cells.

The order of the instructions of R follows a breadth-first search of the graph G underlying P. For every node u in the traversal and for $b \in \{0,1\}$ let $S_{u,b} \subseteq V_i$ denote all the nodes $v \in V$ that have an edge to u labeled b. Assume that node v is labeled by x_v. R simulates the transition to u by computing $\hat{y}_u \leftarrow \sum_{b=0}^{1} \sum_{v \in S_{u,b}} \hat{y}_v \cdot (bx_v + (1-b)(1-x_v))$, which uses the addition and multiply by input value instructions of RMS.

After the initialization of the program the only variable which has value 1 is $\hat{y}_{v_0}$, while all other variables have value 0. Assume that after R evaluates i inputs $x_{k_1}, \ldots, x_{k_i}$ the only variable that stores 1 in the program is $\hat{y}_v$ and that v is the current node in P after it evaluates $x_{k_1}, \ldots, x_{k_i}$. Assume further that there is an edge from v to u labeled by the value of x_k. Then, the computation that R runs ensures that $\hat{y}_u$ is loaded with the value 1 while all other variables are loaded with 0. This property propagates to all levels including the last implying that the value of $\hat{y}_{v_a}$ which R outputs is 1 if and only if P accepts.

Since there is a constant number of instructions in R for every edge in P the size of R is $O(|P|)$. □

We now show that branching programs can compute composed functions in an efficient manner.

Claim A.3. *Let $f : \{0,1\}^n \to \{0,1\}$ and $g_j : \{0,1\}^{n_j} \to \{0,1\}$, $j = 1, \ldots, n$. Suppose that $P, P_1, \ldots, P_n$ are branching programs that compute $f, g_1, \ldots, g_n$ respectively. Then there is a branching program $P(P_1, \ldots, P_n)$ that computes the composed function $f(g_1, \ldots, g_n)$ whose size is polynomial in $|P| + \sum_{i=1}^{n} |P_i|$.*

Proof. The program $P(P_1, \ldots, P_n)$ is defined over all the input variables for the component programs $P, P_1, \ldots, P_n$. The structure of $P(P_1, \ldots, P_n)$ is identical to P except that we replace any node labeled by $x_j, j = 1, \ldots, n$ by the branching program P_j in the following way.

Assume that in P a node v has an edge labeled 0 and an edge labeled 1 to nodes u_0, u_1 respectively and that u_b is labeled by x_{k_b} for $b = 0, 1$. In $P(P_1, \ldots, P_n)$ the node v is replaced by a copy of P_j except for the last level of P_j. Every edge in P_j to the accepting state is replaced by an edge to the initial state of the copy of P_{k_1} that replaces u_1. Similarly, every edge in P_j to the rejecting state is replaced by an edge to the initial state of the copy of P_{k_0} that replaces u_0.

It follows immediately that $P(P_1, \ldots, P_n)$ computes $f(g_1, \ldots, g_n)$ and its size is at most $|P| \cdot \max\{|P_1|, \ldots, |P_n|\}$. □

Towards constructing universal branching programs, we will need to convert a given branching program into a canonical oblivious form.

Claim A.4 (cf. [51])**.**

Theorem A.5. *For every w, n, d there exists a polynomial-time encoding mapping branching programs P to bit-strings $\hat{P}$ and a (*$\mathsf{poly}(w, n, d)$*-time computable) universal branching program* $\mathsf{UBP}_{w,n,d}$ *such that for every canonical oblivious P with parameters w, n, d and input $x \in \{0,1\}^n$ we have* $\mathsf{UBP}_{w,n,d}(\hat{P}, x) = P(x)$.

Proof. We first describe the high level idea for the encoding and the universal branching program. The encoding of a program P is a binary string which has 1 for each possible pair of nodes (in adjacent levels) that have an edge between them and 0 for any other pair of nodes. The universal branching program has a block of levels for each level V_i of P such that the first level in the block mirrors V_i and the last level in the block encodes for each node in V_i its children in V_{i+1}. The values of $\hat{P}$ move the evaluation from a node v in the first level of the block (which is identical to a node v in V_i evaluating the same input) to a node (v, u_0, u_1) in the last block, such that in V there is a 0 edge from v to u_0 and a 1 edge from v to u_1. The input to that last level is a variable from the input x which moves the evaluation to either u_0 or u_1 in the first level of the next block, again reflecting the evaluation of P. The details follow.

Let P be defined by a tuple $(G = (V, E), v_0, T, \psi_V, \psi_E)$ over input $x = (x_1, \ldots, x_n) \in \{0,1\}^n$ and let the leveling of V be $V = \bigcup_{i=0}^{\ell} V_i$ where $\ell = nd$. Denote the number of nodes in a level V_i by w_i.

The encoding $\hat{P}$ of P works as follows. For any node $v_j \in V_i$, and nodes $v'_{k_0}, v'_{k_1} \in V_{i+1}$, $0 \le i \le \ell - 1, 1 \le j, k_0, k_1 \le w$, let $\alpha_0 = (j, k_0), \alpha_1 = (j, k_1)$ and let $y_{\alpha_0}, y_{\alpha_1}$ be two additional input variables. For a given branching program P and for $b = 0, 1$ the variable y_{α_b} is assigned the value 1 if and only if there is an edge in P from v_j to v'_{k_b}, which the edge labeling function ψ_e marks b, i.e. $b \subseteq \psi_e(v, v'_{k_b})$. The encoding $\hat{P}$ is the sequence of values of all variables y_α.

We next define $\mathsf{UBP}_{n,w,d}$ to be an oblivious branching program in which there are $\ell + 1 = nd + 1$ blocks of levels $W_0, W_1, \ldots, W_\ell$. The number of levels in W_i, $0 \le i \le \ell - 1$ is $2w_i \cdot w_i + 1$ denoted by $U_0, \ldots, U_{2w_iw_{i+1}}$ and the number of levels in W_ℓ is one.

The single level of W_ℓ has the two terminal nodes v_a and v_r. Each of the other blocks of levels, W_i, $0 \le i \le \ell - 1$ simulates the operation of level V_i of an input BP P. U_0 is of width w and has a node u_j for every $v_j \in V_i$. $U_{w_iw_{i+1}}$ is of width w_iw_{i+1} and has a node u_α for every pair $\alpha = (j, k), v_j \in V_i, v'_k \in V_{i+1}$. $U_{2w_iw_{i+1}}$ is of width $w_iw_{i+1}^2$ and has a node u_β for every tuple $\beta = (j_j, k_0, k_1), v_j \in V_i, v'_{k_0}, v'_{k_1} \in V_{i+1}$.

We define the nodes of the levels of W_i inductively as follows. Let $U_0 = \{u_1, \ldots, u_{w_i}\}$. For $0 \le m < w_iw_{i+1}$ let $m = jw_{i+1} + k$, $0 \le j \le w_i, 1 \le k \le w_{i+1}$ and assume that $U_m = \{u_\alpha | \alpha = (c, d), 1 \le c \le j, 1 \le d \le k\} \bigcup \{u_{j+1}, \ldots, u_{w_i}\}$. Define $U'_m = \{u'_\alpha | u_\alpha \in U_m\} \bigcup \{u'_{j+1}, \ldots, u'_w\}$

and define the next level, U_{m+1} by:

$$U_{m+1} = \begin{cases} U'_m \bigcup \{u'_\alpha\} & \text{if } k+1 < w_{i+1},\ \alpha = (j, k+1) \\ U'_m \setminus \{u_j\} \bigcup \{u'_\alpha\} & \text{if } k+1 = w_{i+1},\ \alpha = (j, k+1) \end{cases}$$

Define the set of edges E'_m between U_m and U'_m by all the pairs of analogous nodes, i.e.

$$E'_m = \{(u_{j+1}, u'_{j+1}), \dots (u_{w_i}, u'_{w_i})\} \bigcup \{(u_\alpha, u'_\alpha) | u_\alpha \in U_m, u'_\alpha \in U'_m\}.$$

Define the set of edges E_{m+1} between U_m and U_{m+1} by

$$E_{m+1} = \begin{cases} E'_m \bigcup \{(u_j, u'_\alpha)\} & \text{if } k+1 < w_{i+1},\ \alpha = (j, k+1) \\ E'_m \setminus \{(u_j, u'_j)\} \bigcup \{(u_j, u'_\alpha)\} & \text{if } k+1 = w_{i+1},\ \alpha = (j, k+1) \end{cases}.$$

All edges in E_{m+1} are marked $0, 1$ except for the two edges (u_j, u'_j) and $(u_j, u'_{j,k+1})$ when $k+1 < w_{i+1}$. The two edges are marked $\{0\}$ and $\{1\}$ respectively in this case.

The construction of the first $w_i w_{i+1}+1$ levels of W_i expands each node u_j in U_0 to w_{i+1} nodes $u_{j,1}, \dots, u_{j,w_{i+1}}$ in $U_{w_i w_{i+1}}$. We repeat the same construction *in parallel* expanding each node u_α in $U_{w_i w_{i+1}}$ to $u_{\alpha,1}, \dots, u_{\alpha,w_{i+1}}$ in $U_{2w_i w_{i+1}}$ for every $\alpha = (j, k), 1 \le j \le w_i, 1 \le k \le w_{i+1}$.

The last component of the construction is linking the nodes of last level of block W_i, $\{u_\beta | \beta = (j, k_0, k_1), 1 \le j \le w_i, 1 \le k_0, k_1 \le w_{i+1}\}$ with the nodes of the first level of block W_{i+1}, $\{u_1, \dots, u_{w+1}\}$. For every such β there is an edge from u_β to u_{k_0} marked 0 and an edge from u_β to u_{k_1} marked 1.

Let P be an oblivious BP and $x \in \{0,1\}^n$. Then, formatting the pair (P, x) for input to UBP requires first computing the encoding $\hat{P}$ of P. For every block of levels W_i, the input to level U_m for $m < w_i w_{i+1}$, i.e. $m = j w_{i+1} + k$, $1 \le j \le w_i, 1 \le k \le w_{i+1}$ is y_{α_0} for $\alpha_0 = (j, k)$. The input to level U_m for $w_i w_{i+1} \le m < 2w_i w_{i+1}$, i.e. $m = w_i w_{i+1} + j w_{i+1} + k$, $1 \le j \le w_i, 1 \le k \le w_{i+1}$ is y_{α_1} for $\alpha_1 = (j, k)$. The input to level $2w_i w_{i+1}$ is x_i.

It follows that for every program P, every $i, 0 \le i < \ell$ and every $v_j \in V_i$ there is a path in W_i from $u_j \in U_0$ to $u_\beta \in U_{2w_i w_{i+1}+1}$, $\beta = (j, k_0, k_1)$ marked by the variables of $\hat{P}$ iff there are edges from v_j to v_{k_0} and v_{k_1} marked by 0 and 1 respectively. This in turn implies that $\mathsf{UBP}(\hat{P}, x) = P(x)$. □

Combining Theorem A.5 and Claim A.4, we get the following corollary.

Corollary A.6. *For every S, n there exists a polynomial-time encoding mapping branching programs P to bit-strings $\hat{P}$ and a ($\mathsf{poly}(S, n)$-time computable) universal branching program $\mathsf{UBP}_{S,n}$ such that for every branching program P of size S with n inputs and every $x \in \{0,1\}^n$ we have $\mathsf{UBP}_{S,n}(\hat{P}, x) = P(x)$.*