More Efficient Constant-Round Multi-Party Computation from BMR and SHE

Yehuda Lindell[1], Nigel P. Smart[2], and Eduardo Soria-Vazquez[2]

[1] Bar-Ilan University
[2] University of Bristol

Abstract. We present a multi-party computation protocol in the case of dishonest majority which has very low round complexity. Our protocol sits philosophically between Gentry's Fully Homomorphic Encryption based protocol and the SPDZ-BMR protocol of Lindell et al (CRYPTO 2015). Our protocol avoids various inefficiencies of the previous two protocols. Compared to Gentry's protocol we only require Somewhat Homomorphic Encryption (SHE). Whilst in comparison to the SPDZ-BMR protocol we require only a quadratic complexity in the number of players (as opposed to cubic), we have fewer rounds, and we require less proofs of correctness of ciphertexts. In addition we present a variant of our protocol which trades the depth of the required SHE scheme for more homomorphic multiplications.

1 Introduction

Secure multiparty computation: In the setting of secure multiparty computation (MPC), a set of mutually distrusting parties wish to compute a joint function of their private inputs. Secure computation has been studied has been studied since the 1980s, and it has been shown that any functionality can be securely computed, even in the presence of a dishonest majority [28,14]. Classically, two main types of adversaries have been considered: *passive* (or semi-honest) adversaries follow the protocol specification but try to learn more than allowed from the transcript, and *active* (or malicious) adversaries who run any arbitrary strategy in an attempt to breach security.

Efficient MPC: In the last decade, significant effort has been placed on making secure computation efficient, both theoretically (with asymptotic efficiency) and practically. Both in theory and in practice the round complexity of MPC protocols is of interest. The theoretical interest is obvious, but it is in practice that probably the most effect can be felt. It is well known from practical experiments that often round complexity has more of an effect on the performance of MPC systems than communication complexity. This is especially true in networks with high latency (e.g., when the participating parties are on opposite sides of the world), where protocols with many rounds perform very poorly. In practice, one also finds that constants matter considerably.

Most of the research effort on making secure computation practically efficient has focused on the case of two parties [28,25,24]. The most progress has

been with protocols based on Yao's garbled circuits [28]. Extraordinary efficiency has been achieved for both passive adversaries [4,1] and active adversaries [19,20,27,18,16,22,23]. In contrast, the case of multiple parties is way behind. When considering protocols with many rounds, the protocol of GMW can be used for passive adversaries (see [14] with an implementation in [7]) and the protocols of SPDZ and TinyOT can be used for active adversaries [10,9,17]. However, as mentioned above, these protocols have inherent inefficiency based on the fact that the number of rounds in the protocol is linear in the *depth* of the circuit that the parties compute. In contrast to the impressive progress made in this area for the case of two parties, very little is known for multiple parties.

The focus of this paper: In this paper, we focus on the construction of a concretely efficient MPC protocol which requires a constant number of rounds. Our protocol is based on the BMR approach [3]. This approach consists of constructing a two phase protocol. In the first phase the parties use a generic MPC protocol to construct a "garbled" version of the function being computed. Then, in a constant round evaluation phase the garbled function is evaluated. The first garbling phase works in a gate-by-gate manner, and so by processing all gates in one go we obtain a constant round protocol. Indeed this first garbling phase evaluates, via generic MPC, a circuit of constant depth.

In recent work [21] an efficient variant of the BMR protocol is used which utilizes the SPDZ [10] generic MPC protocol in the first garbling phase. In addition the authors introduce other optimizations which make the entire protocol actively secure for very little additional overhead. The SPDZ protocol itself uses a two phase approach, in the first phase, which utilizes Somewhat Homomorphic Encryption (SHE), correlated randomness is produced. Then in the second phase this correlated randomness is used to evaluate the desired functionality (which in this case is the BMR garbling). Thus overall this protocol, which we dub SPDZ-BMR, consists of three phases; a phase using SHE, a phase doing generic MPC via the SPDZ online phase, and the final BMR circuit evaluation phase.

There is another approach to constant round MPC, which utilizes Fully Homomorphic Encryption (FHE), namely Gentry's MPC protocol [13]. In this protocol the parties simply input their data using the encryption of the underlying FHE scheme, the parties evaluate the function locally using FHE, and then perform a distributed decryption (which requires $R_{\mathsf{Out}} = 2$ rounds of interaction with current FHE schemes). This protocol is essentially optimal in terms of the number of rounds of communication, but it suffers from a number of drawbacks. The major drawback is that it requires FHE, which is itself a prohibitively expensive operation (and currently not practical). In addition, it is not immediately clear how to make the protocol actively secure without incurring significant additional costs. We outline in this paper how to address this latter problem, as a by-product of the analysis of our main protocol.

Our Contributions: Returning to the BMR based approach we note that *any* MPC protocol could be used for the BMR garbling phase, as long as it can be made actively secure within the specific context of the BMR protocol. In

particular we could utilize Gentry's FHE-based MPC protocol (using only a SHE scheme) to perform the first stage of the BMR protocol; a protocol idea which we shall denote by SHE-BMR. The main observation as to why this is possible is that, as we have mentioned, the depth of the circuit computing the BMR garbled circuit is itself constant (and, in particular, independent of the depth of the circuit computing the function itself). This is due to the fact that in the BMR approach all garbled gates are computed in parallel; thus, the depth of the circuit computing the entire garbled circuit equals the depth of the circuit required to compute a single garble gate. We therefore conclude that *somewhat* homomorphic encryption suffices, with the depth being that sufficient to compute a single garbled gate.

A number of problems arise with this idea, which we address in this paper. First, can we make the resulting protocol actively secure for little additional cost? Second, is the required depth of the SHE scheme sufficiently small to make the scheme somewhat practical? Recall the SPDZ-BMR protocol only requires the underlying SHE scheme to support circuits of multiplicative depth one, and increasing the depth increases the cost of the SHE itself. Third, is the resulting round complexity of the scheme significantly less than that of the SPDZ-BMR protocol? Note that we can only expect a constant factor improvement, but such constants matter in practice. Fourth, can we save on any additional costs of the SPDZ-BMR protocol?

Since we use Gentry's FHE-based protocol (or an SHE version of it), we now outline two key challenges with using Gentry's FHE based protocol, which also apply to our protocol. When entering data we require an actively secure protocol to encrypt the FHE data, in particular we need to guarantee to the receiving parties that each encryption is well formed. The standard technique to do this is to also transmit a zero-knowledge proof of the correctness of encryption. A method to do this is given in [9, Appendix F], or [2, Section 3.2]. This is costly, and in practice rather inefficient. We call this protocol ID, and the associated round cost by R_{ID}. In addition if we need to make further *input dependent inputs*, then this round cost will multiply. Thus we also need to introduce a sub-protocol with round cost $R_{\mathsf{Input}+} = 1$, which enables us to place all the zero-knowledge proofs for proving correctness of input into a pre-processing phase.

The second problem with Gentry's protocol is that we need to ensure that the distributed decryption is also actively secure; in the sense that the malicious parties cannot get an honest party to accept an incorrect result. To our knowledge this has not yet been treated in the literature, and we present a sub-protocol for performing this. We denote this sub-protocol by $\mathsf{Out}+$ and its associated round cost by $R_{\mathsf{Out}+}$. Importantly, our sub-protocol has round complexity $R_{\mathsf{Out}+} = 2$.

We present a variant of our protocol which reduces the depth of the required SHE scheme, at the expense of requiring each party to input a larger amount of data. Interestingly, the main aim of the design in the SPDZ-BMR protocol was to reduce the number of multiplications needed (since each multiplication required generating a multiplication tuple for SPDZ, and this was the main cost). In contrast, when using SHE directly, additional multiplications are not expensive

as long as they are carried out in parallel. Stated differently, the main concern is the *depth* of the circuit computing the BMR garbled circuit, and not necessarily its size. Of course, for concrete efficiency, one must try to minimize both, as reducing one slightly while greatly increasing the other would not be beneficial. In order to achieve this reduction in the depth of the circuit computing the BMR circuit, we utilize an observation that when computing the garbled circuit it suffices to obtain either the PRF key on the output wire or its additive inverse. This is due to the fact that we can actually take the PRF key to be the *square* of the value obtained in the garbled gate, which is the same whether k or $-k$ is obtained. This allows us to combine the generation of the indicator-bits and the key-vector generation together. The additional flexibility of being able to output either the key or its additive inverse allows us to reduce the required SHE depth by one; in particular, from a depth of *four* to a depth of *three*.

In summary, we actually obtain two distinct protocols π_b where $b \in \{0, 1\}$; for which $b = 0$ means applying our basic variant protocol and $b = 1$ means applying the modified variant with a reduced depth cost. In some sense we can think of our basic SHE-BMR protocol as the same as the SPDZ-BMR protocol of [21], but it "cuts out the middle man" of producing multiplication triples, and the interaction needed to evaluate the garbling via the online phase of SPDZ. Indeed almost all of our basic protocol is identical to that described in [21]. However, naively applying SHE to the protocol from [21] results in a protocol that is neither efficient nor secure. For example, naively applying Gentry's MPC protocol to the garbling stage would result in needing an SHE scheme which supports a depth logarithmic in the number of parties n; whereas we would rather utilize a SHE scheme with constant depth. Thus we need to carefully design the FHE based MPC protocol to realise the BMR garbled circuit.

By utilizing the actively secure input and output routines in Gentry's protocol we also obtain an actively secure variant of Gentry's FHE based protocol which we denote by G_a. This is in addition to the original passively secure FHE based protocol of Gentry which we denote by G_p.

Comparison: By way of comparison we outline in Figure 1 differences between the variants of our protocol, and those of Gentry and SPDZ-BMR. We let n denote the number of parties, W and G denote the number of wires and gates in the binary circuit respectively, and W_{in} the number of input wires and W_{out} the number of output wires. To ease counting of rounds we consider a secure broadcast to be a single round operation (in the case of a dishonest majority, where parties may abort, a simple two-round echo-broadcast protocol suffices in any case [15]). We will see later that $R_{\mathsf{Out+}} = 2$, and $R_{\mathsf{ID}} = 3$. In the table the various functions T_1, T_2, T_3 describing the number of executions of ID are

$$
\begin{aligned}
T_1 &= 16 \cdot G \cdot n^3 + (8 \cdot G + 4 \cdot W) \cdot n^2 + 9 \cdot W \cdot n + 156 \cdot G \cdot n, \\
T_2 &= 4 \cdot G \cdot n^2 + (3 \cdot W + 1) \cdot n, \\
T_3 &= (4 \cdot G + 2 \cdot W) \cdot n^2 + (W + 1) \cdot n.
\end{aligned}
$$

If we compare the SPDZ-BMR protocol with our protocol variants π_0 and π_1 we see that the major difference in computational cost is the number of invocations of the protocol ID. The difference between SPDZ-BMR and π_0 is equal to $T_1 - T_2 = 16 \cdot G \cdot n^3 + 4 \cdot (G + W) \cdot n^2 + (6 \cdot W - 1) \cdot n + 156 \cdot G \cdot n$ invocations. To be very concrete, for 9 parties, a circuit of size 10,000 gates and wires, the number of ID invocations equals 141,210,000 in SPDZ-BMR versus 3,510,009 in SHE-BMR-π_0 versus 4,950,009 in SHE-BMR-π_1. Thus, π_0 is *one fortieth* of the cost of SPDZ-BMR, and π_1 is *one twenty-eighth* of the cost of SPDZ-BMR. This gap widens further as the number of parties grows, with the difference for 25 parties being a factor of 100 for π_0 and 70 for π_1. We remark, however, that even for just 3 parties, protocols π_0 and π_1 are already one twenty-third and one eighteenth of the cost, respectively.

Protocol	**Security**	**Rounds of Interaction**	**Depth of FHE/SHE**	**Number of ID Execs**
G_p	passive	$3 = 1 + R_{\mathsf{Out}}$	Depth of f	0
G_a	active	$5 = R_{\mathsf{ID}} + R_{\mathsf{Out}+}$	$1 +$ Depth of f	$n + W_{in}$
SPDZ-BMR	active	$16 = 13 + R_{\mathsf{ID}}$	1	T_1
π_0	active	$9 = R_{\mathsf{ID}} + 4 + R_{\mathsf{Out}+}$	4	T_2
π_1	active	$9 = R_{\mathsf{ID}} + 4 + R_{\mathsf{Out}+}$	3	T_3

Fig. 1: Comparison of Gentry's, the SPDZ-BMR and our protocol

On the downside we require an SHE scheme which will support depth three or four circuits, as opposed to the depth one circuits of the SPDZ-BMR protocol. The SHE scheme needs to support message spaces of $\mathbb{F}_p$, where $p > 2^\kappa$. We use [8], which gives potential parameter sizes for various SHE schemes supporting depth two and five, and run the experiments there to compare the parameters required for our specific depths here (depth 1 for SPDZ, depth 4 for protocol π_0 and depth 3 for protocol π_1). Specifically, assuming ciphertexts live in a ring R_q, then the dimension needs to go up by approximately a factor of 1.5 for depth-3 and a factor of 2 for depth-4, and the modulus by a factor of 1.6 for depth-3 and a factor of 2 for depth-4. Assuming standard DCRT representation of R_q elements, this equates to an increase in the ciphertext size by a factor of approximately 2.4 for depth-3, and by approximately a factor of 4 for depth-4. Furthermore, the performance penalty (cost of doing arithmetic) increases by a factor of approximately 3.6 for depth-3, and by a factor of 8 for depth-4. Factoring in this addditional cost, we have that when compared to SPDZ-BMR, the relative improvement in the computational cost in the above example becomes a factor of $40/8 = 5$ for π_0 and $28/3.6 = 7.7$ for π_1 for 9 parties, and a factor of $100/8 = 12.5$ for π_0 and $70/3.6 \approx 19.4$ for π_1 for 25 parties. Thus, both π_0 and π_1 significantly outperform BMR-SPDZ, and the depth reduction carried out in π_1 provides additional speedup (and reduction in bandwidth).

2 Background on MPC and FHE

As a warm up to our main protocol, and to introduce the aspects of the FHE functionality we shall be using in detail we first give an outline of Gentry's FHE based protocol to evaluate the generic MPC functionality.

2.1 The Generic MPC Functionality

The goal of all the protocols in this paper are to securely realise the functionality given in Figure 2. Namely we want protocols which allow n mutually distrusting parties, with a possibly dishonest majority, to evaluate the function $f(x_1, \ldots, x_n)$ on their joint inputs.

The General MPC Functionality: $\mathcal{F}_{\text{MPC}}$

The functionality is parametrized by a function $f(x_1, \ldots, x_n)$ which is input as a binary circuit C_f. The protocol consists of three externally exposed commands **Initialize**, **InputData**, and **Output** and one internal subroutine **Wait**.

Initialize: On input (*init*, C_f) from all parties, where C_f is a Boolean circuit, the functionality activates and stores C_f.

Wait: This waits on the adversary to return a *GO*/*NO-GO* decision. If the adversary returns *NO-GO* then the functionality aborts.

InputData: On input (*input*, P_i, varid, x_i) from P_i and (*input*, P_i, varid, ?) from all other parties, with varid a fresh identifier, the functionality stores (varid, x_i). The functionality then calls **Wait**.

Output: On input (*output*) from all parties, if (varid, x_i) is stored for each P_i, the functionality computes $y = f(x_1, \ldots, x_n)$ and outputs y to the adversary. The functionality then calls **Wait**. If **Wait** does not result in an abort, the functionality outputs y to all parties.

Fig. 2: The MPC Functionality: $\mathcal{F}_{\text{MPC}}$

2.2 A Basic FHE Functionality With Distributed Decryption

We first describe in Figure 3 a basic FHE functionality which contains a distributed decryption functionality. Two points need to be noted about the functionality: Firstly, the distributed decryption operation in **Output** can produce an incorrect result under the control of the adversary, but the "additive error" which is introduced by the adversary is introduced before the adversary learns the correct output. Secondly, the **InputData** routine is actively secure, and so a proof of correctness of its correct decryption is needed for each input ciphertext. The need for such an actively secure input routines is because we need to ensure that parties enter "valid" FHE/SHE encryptions, and that the simulator can "extract" the plaintext values.

A method to perform the required **InputData** operation is given in [9, Appendix F], or [2, Section 3.2]. The basic idea is to check a number of executions

of **InputData** at the same time. The protocol run in two phases, in the first phase a set of reference ciphertexts are produced and via cut-and-choose one subset is checked for correctness, whilst the other is permuted into buckets; one bucket for each value entered via **InputData**. In the second phase the input ciphertexts are checked for correctness by combining them homomorphically with the reference ciphertexts and opening the result. An analysis of the protocol from [9] indicates that it requires $R_{\mathsf{ID}} = 3$ rounds of communication: In the first round of the proof each party broadcasts the reference ciphertexts; in the next round the parties choose which ciphertexts to open, and in the third round the ciphertexts are opened and combined.[3] Thus, overall, three rounds suffice.

To be able to easily count the number of rounds of interaction, and the depth of the operations being performed, we provide a count of these operations in the functionality and/or protocol descriptions. For the functionalities where these are given this counting can be thought of as definitional, whereas for the protocols this counting is the induced counts given the underlying definitions of the basic operation costs from the functionalities.

In the following, we fix the notation $\langle varid \rangle$ to represent the result stored in the variable varid by the $\mathcal{F}_{\text{FHE}}/\mathcal{F}_{\text{SHE}}$ functionalities. In particular, we will use the arithmetic shorthands $\langle z \rangle = \langle x \rangle + \langle y \rangle$ and $\langle z \rangle = \langle x \rangle \cdot \langle y \rangle$ to represent the result of calling the **Add** and **Multiply** commands in the $\mathcal{F}_{\text{FHE}}/\mathcal{F}_{\text{SHE}}$ functionality, and we will slightly abuse those shorthands to denote subsequent additions or multiplications.

The description of **Output** in the case of a passively secure functionality is identical to the behaviour of the standard distributed decryption procedure for FHE schemes such as BGV, again see [9] for how the distributed decryption is performed. We shall provide a simple mechanism to provide active security for the Output command in the next section, which comes at the expense of increasing the required supported depth of the SHE scheme by one.

In the case of a passively secure variant of the FHE functionality, one would always have $e = 0$ in the **Output** routine. Furthermore, we would not need a proof of correctness of the input ciphertexts and so the number of rounds of interaction in the **InputData** routine would be $R_{\mathsf{ID}} = 1$.

2.3 Gentry's FHE-Based MPC Protcol

In [13] Gentry presents an MPC protocol which has optimal round complexity to implement $\mathcal{F}_{\text{MPC}}$. In the $\mathcal{F}_{\text{FHE}}$-hybrid model the protocol can be trivially described as follows: The parties enter their data using the **InputData** command of the FHE functionality, the required function is evaluated using the **Add** and **Multiply** commands (i.e. each party locally evaluates the function using the FHE operations). The **Add-scalar** command is computed by locally encrypting the scalar with fixed randomness (so that all parties have the same ciphertext) and then using the regular FHE **Add** command. Finally, the output is obtained using the **Output** command of the FHE functionality. For passively secure adversaries this gives us an "efficient" MPC protocol, assuming the FHE scheme

[3] Choosing at random which ciphertexts to open cannot be carried out in a single round. However, it is possible for all parties to commit to the randomness in previous rounds and only decrypt in this round.

The FHE Functionality: $\mathcal{F}_{\text{FHE}}/\mathcal{F}_{\text{SHE}}$

The functionality consists of externally exposed commands **Initialize**, **InputData**, **Add**, **Multiply** and **Output**, and one internal subroutine **Wait**.

Initialize: On input (*init*,p) from all parties, the functionality activates and stores p. All additions and multiplications below will be mod p.

Wait: This waits on the adversary to return a *GO*/*NO-GO* decision. If the adversary returns *NO-GO* then the functionality aborts.

InputData: On input (*input*,P_i,varid,x) from P_i and (*input*,P_i,varid,?) from all other parties, with varid a fresh identifier, the functionality stores (varid,x). The functionality then calls **Wait**.
Depth Cost: $D(x) = 0$.
Round Cost: R_{ID}.

Add: On command (*add,* varid_1*,* varid_2*,* varid_3) from all parties (if varid_1, varid_2 are present in memory and varid_3 is not), the functionality retrieves (varid_1, x), (varid_2, y) and stores ($\mathsf{varid}_3, x + y \mod p$).

Add-scalar: On command (*add-scalar, a,* varid_1*,* varid_2) from all parties (if varid_1 is present in memory and varid_2 is not), the functionality retrieves (varid_1, x) and stores ($\mathsf{varid}_2, a + x \mod p$).

Multiply: On command (*multiply,* varid_1*,* varid_2*,* varid_3*) from all parties (if* varid_1*,* varid_2 are present in memory and varid_3 is not), the functionality retrieves (varid_1,x), (varid_2,y) and stores (varid_3,$x \cdot y \mod p$).
In the case of the $\mathcal{F}_{\text{SHE}}$ version of this functionality only a limited depth of such commands can be performed; this depth is specified for the functionality.
Depth Cost: $D(\mathsf{varid}_3) = \max(D(\mathsf{varid}_1), D(\mathsf{varid}_2)) + 1$.

Output: On input (*output*,varid*,* i) from all honest parties (if varid is present in memory), and a value $e \in \mathbb{F}_p$ from the adversary, the functionality retrieves (varid, x), and if $i = 0$ it outputs (varid, x) to the adversary. The functionality then calls **Wait**. If **Wait** does not result in an abort, then the functionality outputs $x + e$ to all parties if $i = 0$, or it outputs $x + e$ only to party i if $i \neq 0$.
Round Cost: $R_{\mathsf{Out}} = 2$ (Basically commit to the distributed decryption and then open)

Fig. 3: The FHE/SHE Functionality: $\mathcal{F}_{\text{FHE}}/\mathcal{F}_{\text{SHE}}$

can actually evaluate the function. For active adversaries we then have to impose complex zero-knowledge proofs to ensure that the **InputData** command is performed correctly, and we need a way of securing the **Output** command (which we will come to later).

3 The SPDZ-BMR Protocol

We shall now overview the SPDZ-BMR protocol from [21]. Much of the details we cover here focus on the offline SHE-part of the SPDZ protocol and how it is used in the SPDZ-BMR protocol. Recall the SPDZ protocol makes use of two phases; one an offline phase which uses an SHE scheme (which for our purposes we model via the functionality $\mathcal{F}_{\text{SHE}}$ above restricted to functions of multiplicative depth one), and an online phase using (essentially) only information theoretic constructs. These two phases are used to create a shared garbled circuit which is then evaluated in a third phase in the SPDZ-BMR protocol.

First Phase Cost: The first phase of the SPDZ-BMR protocol requires an upper bound on the total number of parties n, internal wires W, gates G and input wires per party W_{in} of the circuit which will be evaluated. The phase then calls the offline phase of the SPDZ engine to produce $M = 13 \cdot G$ multiplication triples, $B = W$ shared random bits, $R = 2 \cdot W \cdot n$ shared random values and $I = 8 \cdot G \cdot n$ shared values for entering data per party.

The main cost of the SPDZ-BMR protocol is actually in computing this initial data; yet the paper [21] does not address this cost in much detail. Delving into the paper [9] we see that each of these operations requires parties to encrypt random data under the SHE scheme and to produce additive sharings of SHE encrypted data. This first operation is identical to our *input* command on the functionality $\mathcal{F}_{\text{SHE}}$. We delve into the costs of the operations in more detail:

- **Encrypting (Input) Data ID:** When a party produces an encryption we need to ensure that it is validly formed, so as to protect against active attackers. As remarked above this is done using a zero-knowledge proof of correctness. Whilst the computational costs of this can be amortized due to "packing" in the SHE scheme, it is a non-trivial cost per encryption. We shall denote the computational and round cost in what follows by C_{ID} and R_{ID} respectively, i.e. the computational and round cost of the actively secure *EncCommit* operation from [9].
- **Producing Random ReSharings:** Given a ciphertext encrypting a value m this procedure results in an additive sharing of m amongst the n parties. The computational cost of this procedure is dominated by the invocations of the ID protocol. Since each party needs to encrypt a random value, the computational cost $n \cdot C_{\mathsf{ID}}$ and the round complexity is $R_{\mathsf{ID}} + 1$. Again, the computational costs can be amortized due to the packing of the SHE scheme.
- **Producing Multiplication Triples:** To produce an unchecked triple this requires (per party) the encryption of two random values (of a_i and b_i in the triple $([a], [b], [c])$), plus four resharings (three of which can be done in

parallel, with the fourth only partially in parallel). To produce a checked triple, this needs to be done twice (in parallel), followed by a sacrificing step of one of the triples via a procedure (described in [9]) which requires another two rounds of interaction. Thus the total computational cost is dominated by $12 \cdot n \cdot C_{\mathsf{ID}}$; the round complexity is $R_{\mathsf{ID}} + 4$.

- **Producing Shared Random Bits:** To produce an unchecked random bit we require (per party) the encryption of one random value, one passively secure distributed decryption (requiring only one round of interaction), plus two resharings (which can be done in parallel). To produce a checked random bit, the above has to be combined with an unchecked multiplication triple in a sacrificing step which requires two rounds of interaction. Thus the total computational cost is dominated by $9 \cdot n \cdot C_{\mathsf{ID}}$; and the round complexity is $R_{\mathsf{ID}} + 4$.
- **Producing Shared Random Values:** This requires (per party) the encryption of one random value, and two resharings which can be done in parallel. Thus the total computational cost is $2 \cdot n \cdot C_{\mathsf{ID}}$, and the round complexity is $R_{\mathsf{ID}} + 1$.
- **Producing Input Data:** Per data item which needs to be input for each player this requires the encryption of one random value plus two resharings (which cannot be fully parallelised), as well as one additional round of interaction. Thus the total computational cost is dominated by $C_{\mathsf{ID}} + 2 \cdot n \cdot C_{\mathsf{ID}}$, and the round complexity is $R_{\mathsf{ID}} + 3$.

A major bottleneck in the protocol, for active security, is the cost of encrypting the random data required by the protocol. Combining the costs, using the various formulae above, we see that this cost is given by

$$\begin{aligned} T_{\mathsf{ID}} \cdot C_{\mathsf{ID}} &= 12 \cdot n \cdot C_{\mathsf{ID}} \cdot M + 9 \cdot n \cdot C_{\mathsf{ID}} \cdot B \\ &\qquad + 2 \cdot n \cdot C_{\mathsf{ID}} \cdot R + (1 + 2 \cdot n) \cdot n \cdot C_{\mathsf{ID}} \cdot I \\ &= (12 \cdot 13 \cdot G + 9 \cdot W + 4 \cdot W \cdot n + (1 + 2 \cdot n) \cdot n \cdot 8 \cdot G) \cdot n \cdot C_{\mathsf{ID}} \\ &= (16 \cdot G \cdot n^3 + (8 \cdot G + 4 \cdot W) \cdot n^2 + 9 \cdot W \cdot n + 156 \cdot G \cdot n) \cdot C_{\mathsf{ID}} \end{aligned}$$

which is *cubic* in the number of players. In our protocol the same amortization due to SHE packing can be achieved. Thus we do not pay further attention to the constant improvement in performance due to packing, as the same constant can be applied to our protocol.

The total round complexity of the SPDZ offline phase is the maximum round complexity of the various pre-processing operations in the SPDZ offline phase; namely $R_{\mathsf{ID}} + 4$. This holds since the transmission of *all* random encrypted values can occur in one round at the beginning of this phase. We stress that the depth of the SHE needed for SPDZ is just *one*, making it very efficient.

Second Phase Cost: A careful analysis of the rest of the garbling phase of the SPDZ-BMR protocol implies that it requires an additional six rounds of communication.[4]

[4] With reference to [21] this is one round in the preprocessing-I phase and the start of the preprocessing-II phase due to the Output commands, and three to evaluate the

Third Phase Cost: The online phase of the SPDZ-BMR protocol requires three rounds of interaction, one to open the secret shared values and two to verify the associated MACs.

Summary: In summary, the round complexity of SPDZ-BMR is $R_{\mathsf{ID}} + 10$ in the offline phase, and 3 in the online phase.

4 Extending the $\mathcal{F}_{\mathbf{FHE}}/\mathcal{F}_{\mathbf{SHE}}$ Functionalities

4.1 The Extended Functionality Definition

The first step in describing our new offline protocol for constructing the BMR circuit is to extend the functionalities $\mathcal{F}_{\text{FHE}}/\mathcal{F}_{\text{SHE}}$ to new functionalities $\mathcal{F}_{\text{FHE}^+}/\mathcal{F}_{\text{SHE}^+}$. In Figure 4 we present the $\mathcal{F}_{\text{FHE}^+}$ functionality; the definition of the $\mathcal{F}_{\text{SHE}^+}$ functionality is immediate. These new functionalities mimic the output possibilities of the SPDZ offline phase, which were exploited in [21]; by allowing the functionality to produce encryptions of random data and encryptions of random bits. In addition the functionalities provide a version of **Output**, which we call **Output+**, which does not allow the adversary to introduce an error value. There is in addition a new version of **InputData** called **InputData+** which will enable us to reduce the number of rounds of interaction in our main protocol. Functionally this does nothing different from **InputData** but it will be convenient to introduce a different name for a different implementation within our FHE functionality.

4.2 Securely Realising the Extended Functionality

In Figure 5 we give the protocol π_{FHE^+} for realising the $\mathcal{F}_{\text{FHE}^+}$ functionality in the $\mathcal{F}_{\text{FHE}}$-hybrid model.

Let us look at the **Output+** command in more detail (after first reading Figure 5). Suppose the adversary tries to make player P_j accept an incorrect value, by introducing errors into the calls to the weakly secure **Output** command from $\mathcal{F}_{\text{FHE}}$. The honest player P_j will receive $\mathsf{varid} + e_1$ instead of varid and $\mathsf{authvarid}_j + e_2$ instead of $\mathsf{authvarid}_j$, for some adversarially chosen values of e_1 and e_2. If player P_j is not to abort then these quantities must satisfy $\mathsf{authvarid}_j + e_2 = sk_j \cdot (\mathsf{varid} + e_1)$. Now since we know that $\mathsf{authvarid}_j = \mathsf{varid} \cdot sk_j$ then this implies that the adversary needs to select e_1 and e_2 such that $e_2 = sk_j \cdot e_1$, which it needs to do without having any knowledge of sk_j. Thus either the adversary needs to select $e_1 = e_2 = 0$, or he needs to guess the correct value of sk_j. This will happen with probability at most $1/p$, which is negligible.

The protocol which implements **InputData+** works by first running **InputData** with a random value, and then later providing the difference between the random value input and the real input. This enables *preprocessing* of the InputData procedure, thereby reducing the overall number of rounds.

---

required circuits in step 3 of preprocessing-II (since the circuits are of depth three, and hence require three rounds of computation), plus two to verify all the associated MAC values.

The Extended Functionality $\mathcal{F}_{\text{FHE}^+}$

This functionality runs the same **Initialize**, **Wait**, **InputData**, **Add**, **Multiply,** and **Output** commands as $\mathcal{F}_{\text{FHE}}$ of Figure 3. It additionally has the four following externally exposed commands:

Output+: On input (*output+*,varid, i) from all honest parties (if varid is present in memory), the functionality retrieves (varid, x), and if $i = 0$ it outputs (varid, x) to the adversary. The functionality then calls **Wait**, and only if **Wait** does not abort then it outputs x to all parties if $i = 0$, or it outputs x only to party i if $i \neq 0$.

InputData+: On input (*input+*,P_i,varid,x) from P_i and (*input+*,P_i,varid,?) from all other parties, with varid a fresh identifier, the functionality stores (varid,x). The functionality then calls **Wait**.

RandomElement: This command is executed on input (*randomelement*, varid) from all parties, with varid a fresh identifier. The functionality then selects uniformly at random $x \in \mathbb{F}_p$ and stores (varid,x).

RandomBit: This command is executed on input (*randombit*, varid) from all parties, with varid a fresh identifier. The functionality then selects uniformly at random $x \in \{0, 1\}$ and stores (varid,x).

Fig. 4: The Extended Functionality $\mathcal{F}_{\text{FHE}^+}$

The protocol which implements the **RandomElement** command generates an encrypted random value $\langle x \rangle$, unknown to any party as long as one of the parties honestly chooses his additional share x_i randomly.

The protocol which implements the **RandomBit** command is more elaborate, and borrows much from the equivalent operations in the SPDZ offline phase, see [9]. The basic idea is to generate an encrypted random value $\langle x \rangle$, unknown to any party. This value is then squared to obtain $\langle s \rangle$. The value of s is then publicly revealed and an arbitrary square root y is taken. As long as $s \neq 0$ (which happens with negligible probability due to the size of p) we then have that $\langle b \rangle = \langle x \rangle / y$ is an encryption of a value chosen uniformly from $\{-1, 1\}$. Since p is prime, with probability $1/2$ the square root taken will be equal to x and with probability $1/2$ it will be equal to $-x$. This encryption of a value in $\{-1, 1\}$ is turned into an encryption of a value in $\{0, 1\}$ by the final step, by computing the linear function $(\langle b \rangle + 1)/2$ However, unlike in SPDZ no sacrificing procedure is required as the **Output+** command is actively secure.

Theorem 1. *Protocol π_{FHE^+} securely computes $\mathcal{F}_{FHE^+}$ in the $\mathcal{F}_{FHE}$-hybrid model in the UC framework, in the presence of static, active adversaries corrupting any number of parties.*

Proof (sketch). By [5], it suffices to prove the security of Protocol π_{FHE^+} in the SUC (simple UC) framework. We will sketch the proof for each of the processes in the functionality separately. In the $\mathcal{F}_{\text{FHE}}$-hybrid model the security follows in a straightforward way utilizing the security of the commands in $\mathcal{F}_{\text{FHE}}$.

Protocol π_{FHE^+}

This protocol implements the functionality $\mathcal{F}_{\text{FHE}^+}$ in the $\mathcal{F}_{\text{FHE}}$-hybrid model.

Initialize: This performs the initialisation routine just as in the $\mathcal{F}_{\text{FHE}}$ functionality. However, in addition, each party executes **InputData** to obtain an encryption $\langle sk_i \rangle$ of a random MAC value sk_i known only to player P_i.

Output+: On input (*output+*, varid, i) from all honest parties, if varid is present in memory, the following steps are executed.

1. If $i \neq 0$, party P_i computes $\mathsf{authvarid}_i = \langle \mathsf{varid} \rangle \cdot \langle sk_i \rangle$, else, each party P_j computes $\mathsf{authvarid}_j = \langle \mathsf{varid} \rangle \cdot \langle sk_j \rangle$.
2. The parties call $\mathcal{F}_{\text{FHE}}$ with the command (*output*, varid, i).
3. If $i \neq 0$, they call $\mathcal{F}_{\text{FHE}}$ with the command (*output,* $\mathsf{authvarid}_i$, i), else, they use command (*output,* $\mathsf{authvarid}_j$, j) for every $j \in [1, \ldots, n]$.
4. Any party P_j aborts if $\mathsf{authvarid}_j \neq \mathsf{varid} \cdot sk_j$.

Depth Needed: $D(\mathsf{varid}) + 1$.

Round Cost: 2 (since steps 2 and 3 can be performed in parallel).

InputData+: The first step of this command does not depend on the input, and so can be run in a pre-processing step if the number of values to be input per party are known in advance. Upon input $(input+, P_i, \mathsf{varid}, x)$ with $x \in \mathbb{F}_p$ for P_i and $(input+, P_i, \mathsf{varid}, ?)$ for all other parties:

1. Party P_i chooses a random $r_i \in \mathbb{F}_p$ (in the same field as x) and sends $(input, P_i, \mathsf{varid}\text{-}1, r_i)$ to Functionality $\mathcal{F}_{\text{FHE}}$.
2. All parties P_j with $j \neq i$ send $(input, P_i, \mathsf{varid}\text{-}1, ?)$ to Functionality $\mathcal{F}_{\text{FHE}}$.
3. Party P_i broadcasts $c_i = x_i - r_i \pmod p$ to all parties.
4. All parties send $(add\text{-}scalar, c_i, \mathsf{varid}\text{-}1, \mathsf{varid})$ to Functionality $\mathcal{F}_{\text{FHE}}$.

Depth Needed: $D(x_i) = D(c) = 0$.

Round Cost: $R_{\mathsf{ID}} + 1$. Although all R_{ID} rounds can be performed in parallel at the start of the protocol.

RandomElement:

1. For $i = 1, \ldots, n$, each P_i chooses a random $x_i \in \mathbb{F}_p$, and calls $\mathcal{F}_{\text{FHE}}$ with the command (*input*,P_i,x_i) from party P_i and (*input*,P_i,?) for the others.
2. Call **Add** as many times as needed to compute $\langle x \rangle = \langle x_1 \rangle + \cdots + \langle x_n \rangle$.

Depth Needed: $D(x_i) = \max\{D(x_i)\} = 0$.

Round Cost: R_{ID}.

RandomBit: This command requires a more elaborate implementation

1. For $i = 1, \ldots, n$, call $\mathcal{F}_{\text{FHE}}$ with the command (*input*,P_i,x_i) from party P_i and (*input*,P_i,?) for the rest of the parties.
2. Call **Add** as many times as needed to compute $\langle x \rangle = \langle x_1 \rangle + \cdots + \langle x_n \rangle$.
3. Call **Multiply** to compute $\langle s \rangle = \langle x \rangle \cdot \langle x \rangle$.
4. Call $\mathcal{F}_{\text{FHE}^+}$ on input (*output+*, s, 0) so all parties obtain s.
5. $y = \sqrt{s} \pmod p$, if $s = 0$ then restart the protocol.
6. $\langle b \rangle = \langle x \rangle / y$.
7. $\langle \mathsf{varid} \rangle = (\langle b \rangle + 1)/2$, note this last operation is a linear operation.

Depth Needed: $D(s) + 1 = 2$. Note this is the depth required, but the output encrypted bit has depth zero.

Round Cost: $R_{\mathsf{ID}} + 2$.

Fig. 5: Protocol π_{FHE^+}

Output+: The security of **Output+** relies on the security of the **InputData** and **Output** commands of $\mathcal{F}_{\text{FHE}}$. Namely, by the security of **InputData** we have that all sk_j values are secret, and by the security of **Output** the only change that $\mathcal{A}$ can make to the output is an additive difference e (fixed before the output is given). Thus, $\mathcal{A}$ can only change the output if it chooses additive differences e_1, e_2 with $e_1 \neq 0$ such that $(x + e_1) \cdot sk_j = x \cdot sk_j + e_2 \pmod{p}$, where x is the value output. This implies that $e_1 \cdot sk_j = e_2 \pmod{p}$. Since sk_j is secret, the adversary can cause this equality to hold with probability at most p.

We remark that the MAC key sk_j is only used for output values given to P_j. Thus, it always remains secret (even when used for many outputs).

The simulator for **Output+** works simply by simulating the **Output** interaction with $\mathcal{F}_{\text{FHE}}$ for all honest P_i. Regarding a corrupt P_j, the simulator receives the value x that is supposed to be output. Furthermore, the simulator receives the value sk_j from the **InputData** instruction, as well as any errors that are introduced in the **Output** calls by corrupted parties. Thus, the simulator can construct the exact value that $\mathcal{A}$ would receive in a real execution, as required.

InputData+: The only difference between **InputData+** and **InputData** is that **InputData+** can be run such that the actual input is only known to the party in the last round of the protocol. This is done in a straightforward way by using **InputData** to have a party input a random string, and then using that result to mask the real data (at the end). The simulator for this procedure therefore relies directly on the **InputData** procedure of $\mathcal{F}_{\text{FHE}}$ in a straightforward way. Namely, in the $\mathcal{F}_{\text{FHE}}$-hybrid model when the party P_i is corrupted, the simulator receives the value r_i that party P_i sends to **InputData**. Then, upon receiving c_i as broadcast by P_i, the simulator defines $x_i = c_i + r_i \pmod{p}$ and sends $(input+, P_i, \mathsf{varid}, x_i)$ to the ideal functionality as input. In the case that P_i is honest, the simulator chooses a random $c_i \in \mathbb{F}_p$ and simulates P_i broadcasting that value. Furthermore, it simulates the $(input, ...)$ and $(add\text{-}scalar, ...)$ interaction with $\mathcal{F}_{\text{FHE}}$.

The view of the adversary is identical in the simulated and real executions. In addition, since **InputData** is secure and c_i is broadcast and therefore the same for all parties, the protocol fully determines the input value $x_i = c_i + r_i pmodp$, as required.

RandomElement: This is a straightforward coin tossing protocol. The security is derived from the fact that $\mathcal{F}_{\text{FHE}}$ provides a secure **InputData** protocol that reveals no information about the input values. Thus, no party knows anything about the x-values input by the others. Formally, a simulator just simulates the message interaction with $\mathcal{F}_{\text{FHE}}$ for all of the $(input, P_i, x_i)$ and $(input, P_i, ?)$ messages. As long as at least one party is honest, the distribution over the value x defined is uniform, as required.

RandomBit: The first step of this protocol is to essentially run **RandomElement** in order to define a random shared value x. Then, the value $s = x^2$

(mod p) is output to all parties, and each takes the same square-root y of s. Assume that the square root taken is the one that is between 1 and $(p-1)/2$. Now, if $1 \leq x \leq \frac{p-1}{2}$, then $y = x$ and so $\langle b \rangle = \langle 1 \rangle$, and we have that $\langle \mathsf{varid} \rangle = \langle \frac{1+1}{2} \rangle = \langle 1 \rangle$. Else, if $\frac{p-1}{2} < x \leq p-1$ then $\langle b \rangle = \langle -1 \rangle$ and we have $\langle \mathsf{varid} \rangle = \langle \frac{-1+1}{2} \rangle = \langle 0 \rangle$. The security relies on the fact that the result is *fully determined* from the $(input, ...)$ messages sent in the beginning. Relying on the security of **InputData** and **Add/Multiply** in $\mathcal{F}_{\text{FHE}}$, and on the security of the **Output+** procedure, the value x is uniformly distributed and the value s that is output to all parties equals x^2 and no other value. All other steps are deterministic and thus this guarantees that the output is a uniformly distributed bit, as required.

Regarding simulation, the simulator simulates the calls to **InputData**, **Add** and **Multiply** as in the protocol. For the output, the simulator simply chooses a random s as the value received from **Output+**. The view of the parties is clearly identical to in a real execution.

This completes the proof sketch of the theorem. □

5 The First Variant of the SHE-BMR Protocol: π_0

In this section we outline our basic protocol, which follows much upon the lines of the SPDZ-BMR protocol. The modifications needed for a variant using only depth three will be left to Section 6. We divide our discussion into three subsections. In the first section we outline the offline functionality $\mathcal{F}_{\text{offline}}$ we require. This functionality produces a shared garbled circuit which computes the function amongst the players.

For each wire there are $2 \cdot n$ wire labels, corresponding to two labels for each party. The wire labels are held as encrypted key values $\langle k^i_{w,\beta} \rangle$, where encryption is under the SHE scheme, along with encrypted masking values $\langle \lambda_w \rangle$; where $1 \leq w \leq W$, $\beta \in \{0,1\}$ and $1 \leq i \leq n$. The garbled gates are held as a set of linear combinations of outputs from a suitable Pseudo-Random Function (PRF) which is keyed by the wire labels of all parties. These linear combinations are then used to one-time pad encrypt the output wire label, with the precise linear combination to be used in any given situation determined by the encrypted mask values. The output wire masking values are decrypted towards all parties, and the input wire masking values are decrypted towards the inputting party, but everything else remains held in encrypted form.

We then present the online protocol $\pi_{\text{MPC},0}$ which implements $\mathcal{F}_{\text{MPC}}$ in the $\mathcal{F}_{\text{offline}}$-hybrid model. This proceeds by first decrypting the input wire labels desired for each party via the distributed decryption functionality, and revealing the associated selector variables $\Lambda_w = \rho_w \oplus \lambda_w$, where ρ_w is the actual intended wire value. The parties are then able, for each gate, to determine which linear combination to apply (using the selector variables), and can then determine the output wire label using the given linear combination. From this they can determine the output selector variable and repeat the process for the next gate, and so on. Once all gates have been processed in this way the players have learnt

the selector variables Λ_w for the output wires, and so can compute the output wire values from $\Lambda_w \oplus \lambda_w$, where the value of λ_w, for the output wires, was revealed in the pre-processing phase.

At the end of this section we present the offline protocol itself $\pi_{\text{offline},0}$ which implements $\mathcal{F}_{\text{offline}}$ in the $\mathcal{F}_{\text{FHE}^+}$-hybrid model.

5.1 Functionality $\mathcal{F}_{\text{offline}}$ for the Offline Phase

We first present the offline functionality for our main MPC protocol. This is almost identical to the offline functionality for the SPDZ-BMR protocol of [21]. The main difference is that it is built on top of our $\mathcal{F}_{\text{FHE}^+}$ functionality from the previous section, as opposed to the SPDZ MPC protocol. In particular this means we have just a single pre-processing step as opposed to the two phases in [21], which are in turn inherited from the two phases of the SPDZ protocol. Our functionality is presented in Figure 6.

5.2 The SHE-BMR Protocol Specification $\pi_{\text{MPC},0}$

We can now give our protocol $\pi_{\text{MPC},0}$ described in Figure 7, which securely computes the functionality $\mathcal{F}_{\text{MPC}}$ described in Figure 2, in the $\mathcal{F}_{\text{offline}}$-hybrid model.

We do not discuss the protocol in detail, since it is almost identical to the protocol from [21]. The only thing to really notice from the point of view of complexity is that the round complexity will be two more than the round complexity of the pre-processing phase, since the calls to **Output+** will consume two rounds in parallel. In addition the computational cost is mainly in the pre-processing step. The overal depth required of the SHE scheme will be one more than for the pre-processing phase, since we need to consume another level when executing **Output+**.

5.3 The $\pi_{\text{offline},0}$ Protocol

Protocol $\pi_{\text{offline},0}$ in Figure 8 implements $\mathcal{F}_{\text{offline}}$ in the $\mathcal{F}_{\text{FHE}^+}$-hybrid model.

For completeness, we show how to calculate the output indicators for functions $f_g = AND$ and $f_g = OR$ in Figure 9 as shown in [21]. Note that we consume a multiplicative depth of two for both operations.

- For $f_g = AND$, we compute $\langle t \rangle = \langle \lambda_a \rangle \cdot \langle \lambda_2 \rangle$ and then $\langle x_a \rangle = (\langle t \rangle - \langle \lambda_c \rangle)^2$, $\langle x_b \rangle = (\langle \lambda_a \rangle - \langle t \rangle - \langle \lambda_c \rangle)^2$, $\langle x_c \rangle = (\langle \lambda_b \rangle - \langle t \rangle - \langle \lambda_c \rangle)^2$, $\langle x_d \rangle = (1 - \langle \lambda_a \rangle - \langle \lambda_b \rangle + \langle t \rangle - \langle \lambda_c \rangle)^2$.
- For $f_g = OR$, we first compute $\langle t \rangle = \langle \lambda_a \rangle \oplus \langle \lambda_b \rangle = \langle \lambda_a \rangle + \langle \lambda_b \rangle - 2 \cdot \langle \lambda_a \rangle \cdot \langle \lambda_2 \rangle$, and then $\langle x_a \rangle = (\langle t \rangle - \langle \lambda_c \rangle)^2$, $\langle x_b \rangle = (1 - \langle \lambda_a \rangle - \langle \lambda_b \rangle + 2 \cdot \langle t \rangle - \langle \lambda_c \rangle)^2$, $\langle x_c \rangle = \langle x_b \rangle$, $\langle x_d \rangle = \langle x_a \rangle$.

5.4 Security

The security of our protocol follows from the proof of the security of the SPDZ-BMR protocol in [21]. Apart from the use of Gentry's MPC protocol, as opposed to the SPDZ protocol, (which is purely an implementation change) the only difference is that the InputData in SPDZ-BMR is generated in a way that

The Offline Functionality - $\mathcal{F}_{\text{offline}}$

This functionality runs the same **Initialize**, **Wait**, and **Output+** commands as $\mathcal{F}_{\text{FHE}^+}$. In addition it has the following command:

Preprocessing: On input $(\texttt{preprocessing}, C_f)$, for a circuit C_f with at most W wires and G gates, the functionality performs the following operations.

- For all wires $w \in [1, \ldots, W]$:
  - The functionality chooses and stores a random masking value $\langle \lambda_w \rangle$ where $\lambda_w \in \{0, 1\}$.
  - For every value $\beta \in \{0, 1\}$, each party P_i chooses and stores a random key $\langle k^i_{w,\beta} \rangle$, where $k^i_{w,\beta} \in \mathbb{F}_p$.
- For all wires w which are attached to party P_i the functionality decrypts $\langle \lambda_w \rangle$ to party P_i by running **Output+** as in functionality $\mathcal{F}_{\text{FHE}^+}$.
- For all output wires w the functionality decrypts $\langle \lambda_w \rangle$ to all parties by running **Output+** as in functionality $\mathcal{F}_{\text{FHE}^+}$.
- For every gate g with input wires $1 \leq a, b \leq W$ and output wire $1 \leq c \leq W$.
  - Party P_i provides the following values for $x \in \{a, b\}$ on the $4 \cdot G$ values:

$$F_{k^i_{x,0}}(0||1||g), \ldots, F_{k^i_{x,0}}(0||n||g), \qquad F_{k^i_{x,0}}(1||1||g), \ldots, F_{k^i_{x,0}}(1||n||g)$$
$$F_{k^i_{x,1}}(0||1||g), \ldots, F_{k^i_{x,1}}(0||n||g), \qquad F_{k^i_{x,1}}(1||1||g), \ldots, F_{k^i_{x,1}}(1||n||g)$$

    (In our protocols, the parties actually provide sums of pairs of these values; see Figure 9. This reduces the number of values input from 8 per-party per-gate to only 4 per-party per-gate.)
  - Define the selector variables

$$\chi_1 = \begin{cases} 0, & \text{If } f_g(\lambda_a, \lambda_b) = \lambda_c. \\ 1, & \text{Otherwise.} \end{cases} \qquad \chi_2 = \begin{cases} 0, & \text{if } f_g(\lambda_a, \bar{\lambda_b}) = \lambda_c. \\ 1, & \text{Otherwise.} \end{cases}$$
$$\chi_3 = \begin{cases} 0, & \text{If } f_g(\bar{\lambda_a}, \lambda_b) = \lambda_c. \\ 1, & \text{Otherwise.} \end{cases} \qquad \chi_4 = \begin{cases} 0, & \text{If } f_g(\bar{\lambda_a}, \bar{\lambda_b}) = \lambda_c. \\ 1, & \text{Otherwise.} \end{cases}$$

  - Set $\mathbf{A}_g = (A^1_g, \ldots, A^n_g)$, $\mathbf{B}_g = (B^1_g, \ldots, B^n_g)$, $\mathbf{C}_g = (C^1_g, \ldots, C^n_g)$, $\mathbf{D}_g = (D^1_g, \ldots, D^n_g)$ where for $1 \leq j \leq n$:

$$A^j_g = \left(\sum_{i=1}^{n} F_{k^i_{a,0}}(0||j||g) + F_{k^i_{b,0}}(0||j||g) \right) + k^j_{c,\chi_1}$$
$$B^j_g = \left(\sum_{i=1}^{n} F_{k^i_{a,0}}(1||j||g) + F_{k^i_{b,1}}(0||j||g) \right) + k^j_{c,\chi_2}$$
$$C^j_g = \left(\sum_{i=1}^{n} F_{k^i_{a,1}}(0||j||g) + F_{k^i_{b,0}}(1||j||g) \right) + k^j_{c,\chi_3}$$
$$D^j_g = \left(\sum_{i=1}^{n} F_{k^i_{a,1}}(1||j||g) + F_{k^i_{b,1}}(1||j||g) \right) + k^j_{c,\chi_4}$$

  - The functionality finally stores the values $\langle \mathbf{A}_g \rangle$, $\langle \mathbf{B}_g \rangle$, $\langle \mathbf{C}_g \rangle$, $\langle \mathbf{D}_g \rangle$.

Fig. 6: The Offline Functionality $\mathcal{F}_{\text{offline}}$

The MPC Protocol - $\pi_{\text{MPC},0}$

On input a circuit C_f representing the function f, the parties execute the following commands in sequence.

Preprocessing: This sub-task is performed as follows.

- Call **Initialize** on $\mathcal{F}_{\text{offline}}$ to initialize the FHE scheme.
- Call **Preprocessing** on $\mathcal{F}_{\text{offline}}$ with input C_f.

Online Computation: This sub-task is performed as follows.

- For all his input wires w, each party computes $\Lambda_w = \rho_w \oplus \lambda_w$, where λ_w was obtained in the preprocessing stage, and Λ_w is broadcast to all parties.
- Party i calls **Output+** to all parties on $\mathcal{F}_{\text{offline}}$ to decrypt the key $\langle k_w^i \rangle$ associated to Λ_w, for all his input wires w.
- The parties call **Output+** on $\mathcal{F}_{\text{offline}}$ to decrypt $\{\mathbf{A}_g\}$, $\{\mathbf{B}_g\}$, $\{\mathbf{C}_g\}$, and $\{\mathbf{D}_g\}$ for every gate g.
- Passing through the circuit topologically, the parties can now locally compute the following operations for each gate g. Let the gates input wires be labelled a and b, and the output wire be labelled c.
  - For $j = 1, \ldots, n$ compute k_c^j according to the following cases:
    $(\Lambda_a, \Lambda_b) = (0,0)$: Set $k_c^j = A_g^j - \left(\sum_{i=1}^n F_{k_a^i}(0||j||g) + F_{k_b^i}(0||j||g)\right)$.
    $(\Lambda_a, \Lambda_b) = (0,1)$: Set $k_c^j = B_g^j - \left(\sum_{i=1}^n F_{k_a^i}(1||j||g) + F_{k_b^i}(0||j||g)\right)$.
    $(\Lambda_a, \Lambda_b) = (1,0)$: Set $k_c^j = C_g^j - \left(\sum_{i=1}^n F_{k_a^i}(0||j||g) + F_{k_b^i}(1||j||g)\right)$.
    $(\Lambda_a, \Lambda_b) = (1,1)$: Set $k_c^j = D_g^j - \left(\sum_{i=1}^n F_{k_a^i}(1||j||g) + F_{k_b^i}(1||j||g)\right)$.
  - If $k_c^i \notin \{k_{c,0}^i, k_{c,1}^i\}$, then P_i outputs `abort`. Otherwise, it proceeds. If P_i aborts it notifies all other parties with that information. If P_i is notified that another party has aborted it aborts as well.
  - If $k_c^i = k_{c,0}^i$ then P_i sets $\Lambda_c = 0$; if $k_c^i = k_{c,1}^i$ then P_i sets $\Lambda_c = 1$.
  - The output of the gate is defined to be $(k_c^1, \ldots, k_c^n)$ and Λ_c.
- Assuming party P_i does not abort it will obtain Λ_w for every circuit-output wire w. The party can then recover the actual output value from $\rho_w = \Lambda_w \oplus \lambda_w$, where λ_w was obtained in the preprocessing stage.

Depth Needed: $D(Output + (\{\mathbf{A}_g\}, \{\mathbf{B}_g\}, \{\mathbf{C}_g\}, \{\mathbf{D}_g\})) = 3 + 1 = 4$.

Round Cost: If no party aborts, the round cost of the online stage is that of the first three steps, which can be done in parallel, giving a total round cost of two.

Fig. 7: The MPC Protocol - $\pi_{\text{MPC},0}$

The offline Protocol: $\pi_{\text{offline},0}$

The protocol runs the commands **Initialize**, **Wait**, and **Output+** by calling the equivalent commands on $\mathcal{F}_{\text{FHE}^+}$. Thus we only need to describe **Preprocessing**.

Preprocessing: This step proceeds as follows:

1. Call **Initialize** on the functionality $\mathcal{F}_{\text{FHE}^+}$ with input a prime $p > 2^k$.
2. **Generate wire masks:** For every circuit wire w we need to generate a random and hidden masking-values λ_w. Thus for *all* wires w the parties execute **RandomBit** of $\mathcal{F}_{\text{FHE}^+}$; the output is denoted by $\langle \lambda_w \rangle$.
   Depth Needed: $D(RandomBit) = 2$
   Round Cost: $R_{\mathsf{ID}} + 2$.
3. **Generate keys:** For every wire w, each party $i \in [1, \ldots, n]$ and for $\beta \in \{0, 1\}$, the parties execute the command **InputData** of the functionality $\mathcal{F}_{\text{FHE}^+}$ to obtain output $\langle k^i_{w,\beta} \rangle$; where player i learns $k^i_{w,\beta}$. For the vector of shares $\left(\langle k^i_{w,\beta} \rangle\right)_{i=1}^n$ we shall abuse the notation and denote it by $\langle \mathbf{k}_{w,\beta} \rangle$.
   Depth Needed: $D(k^i_{w,\beta}) = 0$.
   Round Cost: R_{ID}.
4. **Output masks for circuit-input-wires:** For all wires w which are attached to party P_i we execute the command **Output+** on the functionality $\mathcal{F}_{\text{FHE}^+}$ to decrypt $\langle \lambda_w \rangle$ to party i.
   Depth Needed: $\max(D(RandomBit), D(Output + (\lambda_w))) = \max(2, 1) = 2$.
   Round Cost: 2.
5. **Output masks for circuit-output-wires:** In order to reveal the real values of the circuit-output-wires it is required to reveal their masking values. That is, for every circuit-output-wire w, the parties execute the command **Output+** on the functionality $\mathcal{F}_{\text{FHE}^+}$ for the stored value $\langle \lambda_w \rangle$.
   Depth Needed: $\max(D(RandomBit), D(Output + (\lambda_w))) = \max(2, 1) = 2$.
   Round Cost: 2.
6. **Calculate garbled gates:** See Figure 9 for the details of this step.

We note that steps two and three can be run in parallel, and that steps four and five also can be run in parallel, but need to follow steps two. We also note that the calls to **InputData+** in the last step (detailed in Figure 9) need to be executed after step three. Hence, we have:
Total Depth Needed: 3.
Total Round Cost: $\max(R_{\mathsf{ID}} + 3, R_{\mathsf{ID}} + 4) = R_{\mathsf{ID}} + 4$.

Fig. 8: The offline Protocol: $\pi_{\text{offline},0}$

Calculated Garbed Gates Step of $\pi_{\text{offline},0}$

Calculate garbled gates: This step is operated for each gate g in the circuit in parallel. Specifically, let g be a gate whose input wires are a, b and output wire is c. Do as follows:

(a) **Calculate output indicators:** This step calculates four indicators $\langle x_a\rangle, \langle x_b\rangle, \langle x_c\rangle, \langle x_d\rangle$ whose values will be $\langle 0\rangle$ or $\langle 1\rangle$. Each indicator is determined by some quadratic function f_g on $\langle\lambda_a\rangle, \langle\lambda_b\rangle, \langle\lambda_c\rangle$, depending on the truth table of the gate. See Section 5.3 for details.

$$\begin{aligned}
\langle x_a\rangle &= (f_g(\langle\lambda_a\rangle, \langle\lambda_b\rangle) - \langle\lambda_c\rangle)^2\\
\langle x_b\rangle &= (f_g(\langle\lambda_a\rangle, (1 - \langle\lambda_b\rangle)) - \langle\lambda_c\rangle)^2\\
\langle x_c\rangle &= (f_g((1 - \langle\lambda_a\rangle), \langle\lambda_b\rangle) - \langle\lambda_c\rangle)^2\\
\langle x_d\rangle &= (f_g((1 - \langle\lambda_a\rangle), (1 - \langle\lambda_b\rangle)) - \langle\lambda_c\rangle)^2
\end{aligned}$$

Depth Needed: $D(x_*) = D(\lambda_*) + 2 = 2$.

(b) **Assign the correct vector:** The indicators are used to choose, for every garbled label, either $\mathbf{k}_{c,0}$ or $\mathbf{k}_{c,1}$, for $t = a, b, c, d$,

$$\langle \mathbf{v}_{c,x_t}\rangle = (1 - \langle x_t\rangle) \cdot \langle \mathbf{k}_{c,0}\rangle + \langle x_t\rangle \cdot \langle \mathbf{k}_{c,1}\rangle.$$

Depth Needed: $D(\mathbf{v}_{c,x_*}) = \max(D(x_*), D(\mathbf{k}_{c,*})) + 1 = 3$.

(c) **Calculate garbled labels:** Party i can now compute the $2 \cdot n$ PRF values $F_{k^i_{w,\beta}}(0||1||g), \ldots, F_{k^i_{w,\beta}}(0||n||g)$ and $F_{k^i_{w,\beta}}(1||1||g), \ldots, F_{k^i_{w,\beta}}(1||n||g)$, for each input wire w of gate G, and $\beta = 0, 1$.

$$\begin{aligned}
F^0_{k^i_{w,\beta}}(g) &= \left(F_{k^i_{w,\beta}}(0||1||g), \ldots, F_{k^i_{w,\beta}}(0||n||g)\right)\\
F^1_{k^i_{w,\beta}}(g) &= \left(F_{k^i_{w,\beta}}(1||1||g), \ldots, F_{k^i_{w,\beta}}(1||n||g)\right).
\end{aligned}$$

Then, they call $4 \cdot n \cdot G$ times the command **InputData+** on the functionality $\mathcal{F}_{\text{FHE}}$, so all the parties obtain the output:

$$\langle F^0_{k^i_{a,0}} + F^0_{k^i_{b,0}}\rangle, \quad \langle F^1_{k^i_{a,0}} + F^0_{k^i_{b,1}}\rangle, \quad \langle F^0_{k^i_{a,1}} + F^1_{k^i_{b,0}}\rangle, \quad \langle F^1_{k^i_{a,1}} + F^1_{k^i_{b,1}}\rangle.$$

All the parties now compute $\langle \mathbf{A}_g\rangle, \langle \mathbf{B}_g\rangle, \langle \mathbf{C}_g\rangle, \langle \mathbf{D}_g\rangle$ via

$$\begin{aligned}
\langle \mathbf{A}_g\rangle &= \langle \mathbf{v}_{c,x_a}\rangle + \sum_{i=1}^{n} \langle F^0_{k^i_{a,0}}(g) + F^0_{k^i_{b,0}}(g)\rangle\\
\langle \mathbf{B}_g\rangle &= \langle \mathbf{v}_{c,x_b}\rangle + \sum_{i=1}^{n} \langle F^1_{k^i_{a,0}}(g) + F^0_{k^i_{b,1}}(g)\rangle\\
\langle \mathbf{C}_g\rangle &= \langle \mathbf{v}_{c,x_c}\rangle + \sum_{i=1}^{n} \langle F^0_{k^i_{a,1}}(g) + F^1_{k^i_{b,0}}(g)\rangle\\
\langle \mathbf{D}_g\rangle &= \langle \mathbf{v}_{c,x_d}\rangle + \sum_{i=1}^{n} \langle F^1_{k^i_{a,1}}(g) + F^1_{k^i_{b,1}}(g)\rangle
\end{aligned}$$

Round Cost: $R_{ID} = R_{ID} + 1$, but the R_{ID} can be done in parallel before.
Depth Needed: $D(\mathbf{A}_g) = D(\mathbf{B}_g) = D(\mathbf{C}_g) = D(\mathbf{D}_g) = D(\mathbf{v}_{c,x_*}) = 3$.

Fig. 9: Calculate Garbled Gates Step of $\pi_{\text{offline},0}$

guarantees that it is random. For our basic protocol, this is not the case. However, there is nothing that forces the adversary to input the value it actually gets and security is preserved. In particular, the adversary can ignore the value it obtained and use a different one honestly, and no problem arises. So, it is no different from this case where the adversary can choose the value in InputData.

5.5 Analysis of Efficiency

Just as in our analysis of the SPDZ-BMR protocol, we wish to estimate the cost of the most expensive operations; which are the encryptions of input data and random input data.

- Each party calls **InputData** once during the **Initialize** phase of the extended FHE functionality.
- We perform W **RandomBit** operations, each of which consumes a C_{ID} per party.
- To create the encrypted PRF keys we require an additional $2 \cdot W$ invocations of C_{ID} per party.
- Finally to enter the garbled labels we require, $4 \cdot n^2 \cdot G$ invocations of the input data routine, which consists of $4 \cdot n \cdot G$ invocations of **InputData** per party.

Thus the cost of encrypting the data for the SHE-BMR protocol is

$$\Big(4 \cdot n^2 \cdot G + (3 \cdot W + 1) \cdot n\Big) \cdot C_{\mathsf{ID}},$$

which is *quadratic* in n as opposed to the *cubic* complexity of the SPDZ-BMR protocol.

6 A Modified SHE-BMR Protocol of Depth 3: π_1

6.1 Protocol π_1 Description

In this section we give a description of the protocol π_1, which requires only a multiplicative depth of three rather than four as in π_0. On the downside, it also requires additional $2 \cdot W \cdot n \cdot (n-1)$ calls to **InputData**. The new protocol π_1 is, in fact, just a variant of π_0, and for which set of parameters one would be preferred in practice over the other remains to be empirically tested.

Protocol π_0 securely computes the BMR garbled gates, as follows. For every gate the parties first compute the shares $\langle x_a \rangle, \langle x_b \rangle, \langle x_c \rangle, \langle x_d \rangle$ and then use these shares to compute the shares $\langle \mathbf{v}_{c,x_a} \rangle, \langle \mathbf{v}_{c,x_b} \rangle, \langle \mathbf{v}_{c,x_c} \rangle, \langle \mathbf{v}_{c,x_d} \rangle$ of the keys $\mathbf{k}_{c,0}$ or $\mathbf{k}_{c,1}$ on the output wire of the gate. Finally, these shares are masked by the pseudorandom values provided by all parties; see Figure 9. Considering how these equations are computed, we have that the $\langle x_* \rangle$ values require two multiplications and the $\langle \mathbf{v}_{c,x_*} \rangle$ require an additional multiplication. The final multiplication, making it depth-4, is needed for computing **Output+**. Thus, our aim is to compute the $\langle \mathbf{v}_{c,x_*} \rangle$ values directly, with just two multiplications instead of three.

In order to achieve this, we directly considered AND and XOR gates, and provide direct formulae for them. The main idea is that it actually suffices to compute shares of either the key $\mathbf{k}_{c,*}$ on the output wire or its opposite $-\mathbf{k}_{c,*}$ modulo p. The reason that this suffices is that the *square* of these values is the same. Thus, we have two versions of each key: the basic-key and the squared-key. The offline protocol works by the parties calling **RandomElement** in order to generate each *basic-key* and then squaring the result and revealing the *squared-key* to the appropriate party. Recall that in BMR, each party has one part of the key, and inputs it in the offline phase to generate the garbled gates. The parties then compute the shares of the *basic-keys* on the output wire of the gate (or their negative) and mask the result with the outputs of the PRF, computed using the revealed *squared-keys*. Observe that in the online phase, the *basic-key* is revealed (since this is what is masked) and the parties then square it in order to compute the PRF values to decrypt the next garbled gate.

The important property here is that since the *basic-key* is random and was *never revealed*, the parties have no idea if they received the basic-key or it's negative. This is crucial because this would leak information about the values on the wires (as we mentioned, we compute either the key or its negative, and this depends on the values on the wires).

This adds $2 \cdot W \cdot n \cdot (n-1)$ calls to **InputData** to generate the keys via calls to **RandomElement** to ensure that no party knows them in the offline phase.

The AND gate. We now present the equations for computing an AND gate with input wires a, b and output wire c. In order to motivate these equations, we build the first equation for computing $\langle \mathbf{v}_{c,x_a} \rangle$, which is the share of the key output from the first ciphertext in the garbled gate. We denote the basic-keys (before being squared) on the output wire by $\tilde{\mathbf{k}}_{c,0}, \tilde{\mathbf{k}}_{c,1}$. We have:

$$\begin{aligned} \langle \mathbf{v}_{c,x_a} \rangle = (1 - \langle \lambda_a \rangle) \cdot \left(\langle \lambda_c \rangle \cdot \langle \tilde{\mathbf{k}}_{c,1} \rangle + (1 - \langle \lambda_c \rangle) \cdot \langle \tilde{\mathbf{k}}_{c,0} \rangle \right) \\ + \langle \lambda_a \rangle \cdot \left((\langle \lambda_b \rangle - \langle \lambda_c \rangle) \cdot \langle \tilde{\mathbf{k}}_{c,1} \rangle + (1 - \langle \lambda_b \rangle - \langle \lambda_c \rangle) \cdot \langle \tilde{\mathbf{k}}_{c,0} \rangle \right). \end{aligned} \tag{1}$$

This equation can be read as follows. Recall that if the indicator-bit λ_c on the output wire equals 1 then the roles of the 0-key and 1-key are reversed. Then, if the input-bit on wire a equals 0, then $\langle \lambda_a \rangle$ equals 0 and so the output is a function of the first row of the equation. Now, once a equals 0, the output equals 0 irrespective of b, since this is an AND gate. Thus, if the output indicator bit equals 0 then the output should be $\langle \tilde{\mathbf{k}}_{c,0} \rangle$; otherwise the output should be $\langle \tilde{\mathbf{k}}_{c,1} \rangle$. In contrast, if the input on wire a equals 1, then the output depends only on the second row of the equation (since $1 - \langle \lambda_a \rangle$ equals 0). The output in this case depends on b. If $b = 1$ and $c = 0$ or if $b = 0$ and $c = 1$ then the output should be $\langle \tilde{\mathbf{k}}_{c,1} \rangle$ (since in the first case $a = b = 1$ and the output is the 1-key, and in the second case the output should be the 0-key but $c = 1$ and so the roles are reversed). This is obtained by multiplying $\langle \tilde{\mathbf{k}}_{c,1} \rangle$ by $\langle \lambda_b \rangle - \langle \lambda_c \rangle$ which equals ± 1 in both of these cases (and 0 otherwise). Observe also that $\langle \tilde{\mathbf{k}}_{c,0} \rangle$ is multiplied in the second row by $1 - \langle \lambda_b \rangle - \langle \lambda_c \rangle$ which equals 0 in both of these cases that $b = 0, c = 1$ and $b = 1, c = 0$. In contrast, if $b = c = 0$ or $b = c = 1$ then the

output should be $\langle \tilde{\mathbf{k}}_{c,0} \rangle$ (since if $b = c = 0$ then the output is 0, and if $b = c = 1$ then the output is 1 but the 1-key is reversed). This holds using the same logic as above. The remaining three equations are computed similarly, as follows:

$$\begin{aligned}\langle \mathbf{v}_{c,x_b} \rangle = (1 - \langle \lambda_a \rangle) \cdot \Big(\langle \lambda_c \rangle \cdot \langle \tilde{\mathbf{k}}_{c,1} \rangle + (1 - \langle \lambda_c \rangle) \cdot \langle \tilde{\mathbf{k}}_{c,0} \rangle \Big) \\ + \langle \lambda_a \rangle \cdot \Big((\langle \lambda_b \rangle - \langle \lambda_c \rangle) \cdot \langle \tilde{\mathbf{k}}_{c,0} \rangle + (1 - \langle \lambda_b \rangle - \langle \lambda_c \rangle) \cdot \langle \tilde{\mathbf{k}}_{c,1} \rangle \Big)\end{aligned} \tag{2}$$

$$\begin{aligned}\langle \mathbf{v}_{c,x_c} \rangle = \langle \lambda_a \rangle \cdot \Big(\langle \lambda_c \rangle \cdot \langle \tilde{\mathbf{k}}_{c,1} \rangle + (1 - \langle \lambda_c \rangle) \cdot \langle \tilde{\mathbf{k}}_{c,0} \rangle \Big) \\ + (1 - \langle \lambda_a \rangle) \cdot \Big((\langle \lambda_b \rangle - \langle \lambda_c \rangle) \cdot \langle \tilde{\mathbf{k}}_{c,1} \rangle + (1 - \langle \lambda_b \rangle - \langle \lambda_c \rangle) \cdot \langle \tilde{\mathbf{k}}_{c,0} \rangle \Big)\end{aligned} \tag{3}$$

$$\begin{aligned}\langle \mathbf{v}_{c,x_d} \rangle = \langle \lambda_a \rangle \cdot \Big(\langle \lambda_c \rangle \cdot \langle \tilde{\mathbf{k}}_{c,1} \rangle + (1 - \langle \lambda_c \rangle) \cdot \langle \tilde{\mathbf{k}}_{c,0} \rangle \Big) \\ + (1 - \langle \lambda_a \rangle) \cdot \Big((\langle \lambda_b \rangle - \langle \lambda_c \rangle) \cdot \langle \tilde{\mathbf{k}}_{c,0} \rangle + (1 - \langle \lambda_b \rangle - \langle \lambda_c \rangle) \cdot \langle \tilde{\mathbf{k}}_{c,1} \rangle \Big)\end{aligned} \tag{4}$$

In order to prove correctness of these equations, we present the truth table of the outputs in Table 1. Observe that all values are correct, but sometimes the negative value of the basic-key is obtained.

λ_a	λ_b	λ_c	$\langle \mathbf{v}_{c,x_a} \rangle$	$\langle \mathbf{v}_{c,x_b} \rangle$	$\langle \mathbf{v}_{c,x_c} \rangle$	$\langle \mathbf{v}_{c,x_d} \rangle$
0	0	0	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$
0	0	1	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$	$\langle -\tilde{\mathbf{k}}_{c,1} \rangle$	$\langle -\tilde{\mathbf{k}}_{c,0} \rangle$
0	1	0	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$
0	1	1	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$	$\langle -\tilde{\mathbf{k}}_{c,0} \rangle$	$\langle -\tilde{\mathbf{k}}_{c,1} \rangle$
1	0	0	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$
1	0	1	$\langle -\tilde{\mathbf{k}}_{c,1} \rangle$	$\langle -\tilde{\mathbf{k}}_{c,0} \rangle$	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$
1	1	0	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$
1	1	1	$\langle -\tilde{\mathbf{k}}_{c,0} \rangle$	$\langle -\tilde{\mathbf{k}}_{c,1} \rangle$	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$

Table 1: The truth table of the vectors for an AND gate computed in Figure 10.

The XOR gate. We use a similar idea as above to compute the XOR gate. Intuitively, in a XOR gate, there are two cases: $\lambda_a = \lambda_b$ and $\lambda_a \neq \lambda_b$. Multiplying by $\lambda_a - \lambda_b$ gives ± 1 if $\lambda_a \neq \lambda_b$ and 0 if $\lambda_a = \lambda_b$. Furthermore, multiplying by $1 - \lambda_a - \lambda_b$ gives the exact reverse case; it equals 0 if $\lambda_a \neq \lambda_b$ and equals ± 1 if

$\lambda_a = \lambda_b$. This yields the following equations:

$$\langle \mathbf{v}_{c,x_a} \rangle = (\langle \lambda_a \rangle - \langle \lambda_b \rangle) \cdot \left(\langle \lambda_c \rangle \cdot \langle \tilde{\mathbf{k}}_{c,0} \rangle + (1 - \langle \lambda_c \rangle) \cdot \langle \tilde{\mathbf{k}}_{c,1} \rangle \right) \tag{5}$$
$$+ (1 - \langle \lambda_a \rangle - \langle \lambda_b \rangle) \cdot \left(\langle \lambda_c \rangle \cdot \langle \tilde{\mathbf{k}}_{c,1} \rangle + (1 - \langle \lambda_c \rangle) \cdot \langle \tilde{\mathbf{k}}_{c,0} \rangle \right)$$

$$\langle \mathbf{v}_{c,x_b} \rangle = (\langle \lambda_a \rangle - \langle \lambda_b \rangle) \cdot \left(\langle \lambda_c \rangle \cdot \langle \tilde{\mathbf{k}}_{c,1} \rangle + (1 - \langle \lambda_c \rangle) \cdot \langle \tilde{\mathbf{k}}_{c,0} \rangle \right) \tag{6}$$
$$+ (1 - \langle \lambda_a \rangle - \langle \lambda_b \rangle) \cdot \left(\langle \lambda_c \rangle \cdot \langle \tilde{\mathbf{k}}_{c,0} \rangle + (1 - \langle \lambda_c \rangle) \cdot \langle \tilde{\mathbf{k}}_{c,1} \rangle \right)$$

$$\langle \mathbf{v}_{c,x_c} \rangle = \langle \mathbf{v}_{c,x_b} \rangle \tag{7}$$

$$\langle \mathbf{v}_{c,x_d} \rangle = \langle \mathbf{v}_{c,x_a} \rangle \tag{8}$$

Observe that $\langle \mathbf{v}_{c,x_c} \rangle$ and $\langle \mathbf{v}_{c,x_d} \rangle$ need not be computed at all since $(1-a) \oplus b = a \oplus (1-b)$ and $(1-a) \oplus (1-b) = a \oplus b$. As above, we prove correctness via the truth table given in Table 2.

λ_a	λ_b	λ_c	$\langle \mathbf{v}_{c,x_a} \rangle$	$\langle \mathbf{v}_{c,x_b} \rangle$	$\langle \mathbf{v}_{c,x_c} \rangle$	$\langle \mathbf{v}_{c,x_d} \rangle$
0	0	0	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$
0	0	1	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$
0	1	0	$\langle -\tilde{\mathbf{k}}_{c,1} \rangle$	$\langle -\tilde{\mathbf{k}}_{c,0} \rangle$	$\langle -\tilde{\mathbf{k}}_{c,0} \rangle$	$\langle -\tilde{\mathbf{k}}_{c,1} \rangle$
0	1	1	$\langle -\tilde{\mathbf{k}}_{c,0} \rangle$	$\langle -\tilde{\mathbf{k}}_{c,1} \rangle$	$\langle -\tilde{\mathbf{k}}_{c,1} \rangle$	$\langle -\tilde{\mathbf{k}}_{c,0} \rangle$
1	0	0	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$
1	0	1	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$	$\langle \tilde{\mathbf{k}}_{c,1} \rangle$	$\langle \tilde{\mathbf{k}}_{c,0} \rangle$
1	1	0	$\langle -\tilde{\mathbf{k}}_{c,0} \rangle$	$\langle -\tilde{\mathbf{k}}_{c,1} \rangle$	$\langle -\tilde{\mathbf{k}}_{c,1} \rangle$	$\langle -\tilde{\mathbf{k}}_{c,0} \rangle$
1	1	1	$\langle -\tilde{\mathbf{k}}_{c,1} \rangle$	$\langle -\tilde{\mathbf{k}}_{c,0} \rangle$	$\langle -\tilde{\mathbf{k}}_{c,0} \rangle$	$\langle -\tilde{\mathbf{k}}_{c,1} \rangle$

Table 2: The truth table of the vectors for a XOR gate computed in Figure 10.

The modified protocol. In Figure 10 (part 1) we show the modifications required to the offline protocol in this case. The main differences are that the basic keys are generated using **RandomElement** (instead of just being chosen by each party), that the function actually outputs either the key $\langle \tilde{k}^i_{w,\beta} \rangle$ or $\langle -\tilde{k}^i_{w,\beta} \rangle$. Since the actual key used is $k^i_{w,\beta} = (\tilde{k}^i_{w,\beta})^2$, upon receiving some k in the online phase, the parties square it and use the result. This is the only modification to the online phase, as showed in Figure 10 (part 2), which is used to guarantee that it makes no difference whether they receive the key or its negative since the square is always the same.

The Offline Protocol: $\pi_{\text{offline},1}$

This protocol is identical to the $\pi_{\text{offline},0}$ protocol given in Figure 8, except for the following changes:

3 **Generate keys** in Figure 8 is changed as follows:
   (a) For every wire w, bit value $\beta \in \{0,1\}$ and party $i \in [1,\ldots,n]$, the parties execute the command **RandomElement** of the functionality $\mathcal{F}_{\text{FHE}^+}$ to obtain output $\langle \tilde{k}^i_{w,\beta} \rangle$. We stress that nobody learns $\tilde{k}^i_{w,\beta}$. Let varid be the identifier of $\langle \tilde{k}^i_{w,\beta} \rangle$. In the following, we shall abuse the notation to denote $\langle \tilde{\mathbf{k}}_{w,\beta} \rangle = \left(\langle \tilde{k}^1_{w,\beta} \rangle, \ldots, \langle \tilde{k}^n_{w,\beta} \rangle \right)$.
   (b) The parties call $(\mathit{multiply}, \mathsf{varid}, \mathsf{varid}, \mathsf{varid2})$ where varid2 is a new identifier, in order to share a ciphertext $\langle k^i_{w,\beta} \rangle = \langle \tilde{k}^i_{w,\beta} \rangle^2$.
   (c) The parties call $\mathcal{F}_{\text{FHE}^+}$ on input $(\mathit{output}+, \mathsf{varid2}, i)$ for party P_i to obtain $k^i_{w,\beta}$.

   Depth Needed: $D(\mathit{Output} + (k^i_{w,\beta})) = 2$.
   Round Cost: $R_{\mathsf{ID}} + 2$.

4 **Calculate garbled gates** in Figure 9 is changed as follows:
   (a) The **calculate output indicators** and **assign the correct vector** phases are replaced by the following functions, that choose, for every garbled label, either $\tilde{\mathbf{k}}_{c,0}$, $-\tilde{\mathbf{k}}_{c,0}$, $\tilde{\mathbf{k}}_{c,1}$ or $-\tilde{\mathbf{k}}_{c,1}$.
      – For an AND gate, the parties compute shares of the keys on the output wires according to Equations (1)–(4).
      – For a XOR gate, the parties compute shares of the keys on the output wires according to Equations (5)–(8).

      Depth Needed: $D(\mathbf{v}_{c,x_a}) = D(\mathbf{v}_{c,x_b}) = D(\mathbf{v}_{c,x_c}) = D(\mathbf{v}_{c,x_d}) = 2$.

Total Round Cost: $\max(R_{\mathsf{ID}} + 3, R_{\mathsf{ID}} + 4) = R_{\mathsf{ID}} + 4$.
Total Depth Needed: 2.

The modified MPC Protocol - $\pi_{\text{MPC},1}$

This protocol is identical to the $\pi_{\text{MPC},1}$ protocol described in Figure 7, except for the four cases of the **Online Computation** sub-task, in which for $j = 1,\ldots,n$, the values k^j_c are now computed as follows:

$$\mathit{Case}\ (\Lambda_a,\Lambda_b) = (0,0) : \text{Compute } k^j_c = \left(A^j_g - \left(\textstyle\sum_{i=1}^n F_{k^i_a}(0||j||g) + F_{k^i_b}(0||j||g)\right)\right)^2.$$

$$\mathit{Case}\ (\Lambda_a,\Lambda_b) = (0,1) : \text{Compute } k^j_c = \left(B^j_g - \left(\textstyle\sum_{i=1}^n F_{k^i_a}(1||j||g) + F_{k^i_b}(0||j||g)\right)\right)^2.$$

$$\mathit{Case}\ (\Lambda_a,\Lambda_b) = (1,0) : \text{Compute } k^j_c = \left(C^j_g - \left(\textstyle\sum_{i=1}^n F_{k^i_a}(0||j||g) + F_{k^i_b}(1||j||g)\right)\right)^2.$$

$$\mathit{Case}\ (\Lambda_a,\Lambda_b) = (1,1) : \text{Compute } k^j_c = \left(D^j_g - \left(\textstyle\sum_{i=1}^n F_{k^i_a}(1||j||g) + F_{k^i_b}(1||j||g)\right)\right)^2.$$

Depth Needed: $D_{Out+} + D(\{\mathbf{A}_g\},\{\mathbf{B}_g\},\{\mathbf{C}_g\},\{\mathbf{D}_g\}) = 2 + 1 = 3$.

Fig. 10: The Modified Protocol π_1

6.2 Security Of the Modified Protocol

Observe that in the offline phase, the only difference is that the $\langle \mathbf{v}_{c,x_*} \rangle$ values contain the "tilde" version of the keys; more formally, the $\langle \mathbf{v}_{c,x_*} \rangle$ ciphertexts encrypt the *square root* of the keys, and not the keys themselves. Thus, in the online phase, the parties receive the square roots of the keys and need to square them before proceeding. The only issue that needs to be explained here is that the specific square root provided reveals no information. This needs to be justified because if an adversary could know that $-\tilde{k}$ is computed or $\tilde{k}$, then it would know some information about the masks $\lambda_a, \lambda_b, \lambda_c$. However, since the $\tilde{k}$ values are *uniformly distributed* in $\mathbb{F}_p$, and the keys themselves revealed in the offline phase are $k = \tilde{k}^2$, it follows that each of the two square roots of k are equally probable. Stated differently, given k, the distribution over $\tilde{k}$ and $-\tilde{k}$ is identical.

6.3 Analysis of Efficiency of the Modified Protocol

As we noted in the introduction, the two main sources of overhead that concern our MPC protocol are the number of rounds and the number of calls to the ID protocol. The former is not changed by our π_1 variant, but the latter does. To generate the keys in $\pi_{\text{offline},1}$, we now perform $2 \cdot W \cdot n^2$ calls to **InputData**, via calls to **RandomElement**. In π_0 we performed $2 \cdot W \cdot n$ calls to generate the keys, so overall we add $2 \cdot W \cdot n \cdot (n-1)$ calls to **InputData**. To analyse the number of homomorphic multiplications we go through each step of the protocol:

- **Generate keys step:** We perform $4 \cdot W \cdot n$ more multiplications (half of them to square the keys, the other half to **Output+** them).
- **Calculate garbled gates step:**
  1. For every AND gate, we used 13 multiplications in the first variant. Now, by careful rewriting of the equations, we can do this in 20.
  2. For every XOR gate, we used 7 multiplications in the first variant. Now we use 12.
  3. So, on average, we pass from 10 to 16 multiplications per gate, which is an increase of 60% per gate.

Thus, overall on average we perform $4 \cdot W \cdot n + 6 \cdot G$ more homomorphic multiplications. However, in practice each homomorphic multiplication will be more efficient since the overall depth of the SHE scheme can now be three rather than four.

Acknowledgements

The first author was supported in part by the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC consolidators grant agreement n. 615172 (HIPS). The second author was supported in part by ERC Advanced Grant ERC-2010-AdG-267188-CRIPTO and by EPSRC via grant EP/I03126X. The third author was supported in part

by the Marie Sklodowska-Curie ITN ECRYPT-NET (Project Reference 643161). All authors were also supported by an award from EPSRC (grant EP/M012824), from the Ministry of Science, Technology and Space, Israel, and the UK Research Initiative in Cyber Security.

References

1. Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 535–548. ACM, 2013.
2. Carsten Baum, Ivan Damgård, Tomas Toft, and Sarah Zakarias. Better preprocessing for secure multiparty computation. *IACR Cryptology ePrint Archive*, 2016:048, 2016.
3. Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols. In Harriet Ortiz, editor, *22nd STOC*, pages 503–513. ACM, 1990.
4. Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 478–492. IEEE Computer Society, 2013.
5. Ran Canetti, Asaf Cohen, and Yehuda Lindell. A simpler variant of universally composable security for standard multiparty computation. In Gennaro and Robshaw [12], pages 3–22.
6. Ran Canetti and Juan A. Garay, editors. *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*, volume 8043 of *Lecture Notes in Computer Science*. Springer, 2013.
7. Seung Geol Choi, Kyung-Wook Hwang, Jonathan Katz, Tal Malkin, and Dan Rubenstein. Secure multi-party computation of boolean circuits with applications to privacy in on-line marketplaces. In Orr Dunkelman, editor, *Topics in Cryptology - CT-RSA 2012 - The Cryptographers' Track at the RSA Conference 2012, San Francisco, CA, USA, February 27 - March 2, 2012. Proceedings*, volume 7178 of *Lecture Notes in Computer Science*, pages 416–432. Springer, 2012.
8. Anamaria Costache and Nigel P. Smart. Which ring based somewhat homomorphic encryption scheme is best? *IACR Cryptology ePrint Archive*, 2015:889, 2015.
9. Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS*, volume 8134 of *LNCS*, pages 1–18. Springer, 2013.
10. Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Safavi-Naini and Canetti [26], pages 643–662.
11. Juan A. Garay and Rosario Gennaro, editors. *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II*, volume 8617 of *Lecture Notes in Computer Science*. Springer, 2014.
12. Rosario Gennaro and Matthew Robshaw, editors. *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA,*

August 16-20, 2015, Proceedings, Part II, volume 9216 of *Lecture Notes in Computer Science*. Springer, 2015.
13. Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. crypto.stanford.edu/craig.
14. Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred V. Aho, editor, *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 218–229. ACM, 1987.
15. Shafi Goldwasser and Yehuda Lindell. Secure multi-party computation without agreement. *J. Cryptology*, 18(3):247–287, 2005.
16. Yan Huang, Jonathan Katz, and David Evans. Efficient secure two-party computation using symmetric cut-and-choose. In Canetti and Garay [6], pages 18–35.
17. Enrique Larraia, Emmanuela Orsini, and Nigel P. Smart. Dishonest majority multi-party computation for binary circuits. In Garay and Gennaro [11], pages 495–512.
18. Yehuda Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. In Canetti and Garay [6], pages 1–17.
19. Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In Moni Naor, editor, *Advances in Cryptology - EUROCRYPT 2007, 26th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Barcelona, Spain, May 20-24, 2007, Proceedings*, volume 4515 of *Lecture Notes in Computer Science*, pages 52–78. Springer, 2007.
20. Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. In Yuval Ishai, editor, *Theory of Cryptography - 8th Theory of Cryptography Conference, TCC 2011, Providence, RI, USA, March 28-30, 2011. Proceedings*, volume 6597 of *Lecture Notes in Computer Science*, pages 329–346. Springer, 2011.
21. Yehuda Lindell, Benny Pinkas, Nigel P. Smart, and Avishay Yanai. Efficient constant round multi-party computation combining BMR and SPDZ. In Gennaro and Robshaw [12], pages 319–338.
22. Yehuda Lindell and Ben Riva. Cut-and-choose yao-based secure computation in the online/offline and batch settings. In Garay and Gennaro [11], pages 476–494.
23. Yehuda Lindell and Ben Riva. Blazing fast 2pc in the offline/online setting with security for malicious adversaries. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 579–590. ACM, 2015.
24. Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Safavi-Naini and Canetti [26], pages 681–700.
25. Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. In Mitsuru Matsui, editor, *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings*, volume 5912 of *Lecture Notes in Computer Science*, pages 250–267. Springer, 2009.
26. Reihaneh Safavi-Naini and Ran Canetti, editors. *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*. Springer, 2012.

27. Abhi Shelat and Chih-Hao Shen. Two-output secure computation with malicious adversaries. In Kenneth G. Paterson, editor, *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings*, volume 6632 of *Lecture Notes in Computer Science*, pages 386–405. Springer, 2011.
28. Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*, pages 160–164. IEEE Computer Society, 1982.