Low Cost Constant Round MPC Combining BMR and Oblivious Transfer

Carmit Hazay* Peter Scholl† Eduardo Soria-Vazquez‡

这两个性质是重点：主动安全、常数轮

Abstract

In this work, we present two new actively secure, constant round multi-party computation (MPC) protocols with security against all-but-one corruptions. Our protocols both start with an actively secure MPC protocol, which may have linear round complexity in the depth of the circuit, and compile it into a constant round protocol based on garbled circuits, with very low overhead.

1. Our first protocol takes a generic approach using any secret-sharing-based MPC protocol for binary circuits, and a correlated oblivious transfer functionality. (备注：BGW and OT)
2. Our second protocol builds on secret-sharing-based MPC with information-theoretic MACs. This approach is less flexible, being based on a specific form of MPC, but requires no additional oblivious transfers to compute the garbled circuit. SPDZ

In both approaches, the underlying secret-sharing-based protocol is only used for *one actively secure* $\mathbb{F}_2$ *multiplication per AND gate*. An interesting consequence of this is that, with current techniques, constant round MPC for binary circuits is not much more expensive than practical, non-constant round protocols.

We demonstrate the practicality of our second protocol with an implementation, and perform experiments with up to 9 parties securely computing the AES and SHA-256 circuits. Our running times improve upon the best possible performance with previous protocols in this setting by 60 times.

*Bar-Ilan University, Israel. Email: carmit.hazay@biu.ac.il

†Aarhus University, Denmark. Work done whilst at University of Bristol, UK. Email: peter.scholl@cs.au.dk

‡Aarhus University, Denmark. Work done whilst at University of Bristol, UK. Email: eduardo@cs.au.dk

Contents

1 Introduction **3**
- 1.1 Our Contributions . . . 3
- 1.2 Technical Overview . . . 5

2 Preliminaries **9**
- 2.1 Circular 2-Correlation Robust PRF . . . 9
- 2.2 Almost-1-Universal Linear Hashing . . . 10
- 2.3 Security Model . . . 11
- 2.4 Commitment Functionality . . . 12
- 2.5 Coin-Tossing Functionality . . . 12
- 2.6 Correlated Oblivious Transfer . . . 12
- 2.7 Functionality for Secret-Sharing-Based MPC . . . 13
- 2.8 BMR Garbling . . . 13

3 Generic Protocol for Multi-Party Garbling **14**
- 3.1 The Preprocessing Functionality . . . 14
- 3.2 Protocol Overview . . . 14
- 3.3 Bit/String Multiplications . . . 17
- 3.4 Consistency Check . . . 18
- 3.5 Security Proof . . . 22

4 More Efficient Garbling with Multi-Party TinyOT **25**
- 4.1 Secret-Shared MAC Representation . . . 26
- 4.2 MAC-Based MPC Functionality . . . 27
- 4.3 Garbling with $\mathcal{F}_{\text{n-TinyOT}}$ . . . 27

5 The Online Phase **29**
- 5.1 The Online Phase with $\mathcal{F}^{\text{KQ}}_{\text{Prepocessing}}$ . . . 36

6 Performance **37**
- 6.1 Implementation . . . 38
- 6.2 Communication Complexity Analysis . . . 39

A Protocol for GMW-Style MPC for Binary Circuits **45**
- A.1 Why the Need for Key Queries? . . . 46
- A.2 Security . . . 49
- A.3 Parameters . . . 51
- A.4 Communication Complexity . . . 51
- A.5 Round Complexity . . . 52
- A.6 Realizing General Secure Computation . . . 52

1 Introduction

Secure multi-party computation (MPC) protocols allow a group of n parties to compute some function f on the parties' private inputs, while preserving a number of security properties such as *privacy* and *correctness*. The former property implies data confidentiality, namely, nothing leaks from the protocol execution but the computed output. The latter requirement implies that the protocol enforces the integrity of the computations made by the parties, namely, honest parties learn the correct output. Modern, practical MPC protocols typically fall into two main categories: those based on secret sharing [GMW87, RBO89, BOGW88, DN07, IPS09, DPSZ12], and those based on garbled circuits [Yao86, BMR90, LP07, KS08, LP09, LP11, CKMZ14, MRZ15]. When it comes to choosing a protocol, many different factors need to be taken into account, such as the function being evaluated, the latency and bandwidth of the network and the adversary model.

Secret-sharing-based protocols tend to have lower communication requirements in terms of bandwidth, but require a large number of rounds of communication, which increases with the circuit depth of the function. In this approach, the parties first secret-share their inputs and then evaluate the circuit gate by gate while preserving privacy and correctness. In low-latency networks, they can have an extremely fast online evaluation stage, but the round complexity makes them much less suited to high-latency networks, when the parties may be far apart.

Garbled circuits, introduced in Yao's protocol [Yao86], are the core behind all practical, constant round protocols for secure computation. In the two-party setting, one of the parties "encrypts" the circuit being evaluated, whereas the other party privately evaluates it. Garbled-circuit-based protocols have recently become much more efficient, and currently give the most practical approach for actively secure computation of binary circuits [RR16, NST17, WRK17a]. With more than two parties, the situation is more complex, as the garbled circuit must be computed by all parties in a distributed manner using general MPC. This approach comes from the 'BMR' protocol by Beaver, Micali and Rogaway [BMR90], which gives a constant round protocol because all gates can be garbled in parallel, and general MPC is only needed for a circuit of constant depth.

The multi-party BMR paradigm has received much less attention than two-party protocols. The original BMR construction uses generic zero-knowledge techniques for proving correct computation of the outputs of a pseudorandom generator, so is impractical. A different protocol, but only for three parties, was designed by Choi et al. [CKMZ14] in the dishonest majority setting. More practical, actively secure protocols for any number of parties are the recent works of Lindell et al. [LPSY15, LSS16], which use somewhat homomorphic encryption (SHE) or generic MPC to garble a circuit. Ben-Efraim et al. [BLO16] recently presented and implemented an efficient multi-party garbling protocol based on oblivious transfer, but with only semi-honest security. Concurrently to this work, Wang et al. [WRK17b] introduced protocols based on authenticated garbling and a preprocessing phase based on 'TinyOT', a secret-sharing-based protocol for binary circuits [NNOB12].

1.1 Our Contributions

In this work, we present two practical, actively secure, constant round multi-party protocols in the presence of up to $n-1$ out of n corruptions. Our protocols follow the BMR approach, where we use an existing, non-constant-round MPC protocol to generate a garbled circuit in a distributed manner. After creating the garbled circuit, there is an online phase where the parties obtain the output of the computation; this part is similar to previous works, and much more efficient than the garbling phase. In that context, we present two new protocols for securely generating the garbled circuit:

1. A generic approach using any actively secure secret-sharing-based MPC protocol for binary circuits, and a correlated oblivious transfer functionality.

2. A specialized protocol which uses secret-sharing-based MPC with information-theoretic message authentication codes (MACs), such as TinyOT [NNOB12, FKOS15]. This approach is less flexible, but requires no additional correlated OTs to compute the garbled circuit.

The main novelty in both of these approaches is that the underlying secret-sharing-based protocol is only used for *one actively secure $\mathbb{F}_2$ multiplication per AND gate*. Thus, the preprocessing cost of constant round MPC is brought down to almost the same as the best, non-constant-round protocol.

Generic Protocol. This method has a total communication complexity of $O(M + |C|\kappa n^2)$ bits to evaluate a binary circuit with $|C|$ AND gates, where κ is the computational security parameter and M is the cost of the underlying secret-sharing-based protocol, denoted Π. Protocol Π has to evaluate a circuit consisting of a layer of linear operations, followed by $|C|$ parallel AND gates, and another linear layer before the outputs.

The additional $O(|C|\kappa n^2)$ cost comes from the correlated oblivious transfers (OTs), of which we need $O(n^2)$ per AND gate, and sending the garbled circuit in the online phase. With OT extension techniques for amortizing the cost of OT, each OT requires $O(\kappa)$ communication, and the main computational cost of our protocol is cheap symmetric cryptographic primitives, after a one-time setup phase of seed OTs between every pair of parties.

An asymptotically good approach to realize Π is the IPS compiler [IPS08] that can be instantiated with semi-honest GMW [GMW87] as the inner protocol and a suitable honest-majority outer protocol [DI06]. By Theorem 2, Section 5 from [IPS08], for any constant number of parties $n \geq 2$, the functionality can be computed with communication complexity $O(|C|)$ plus low order terms that depend on a statistical parameter s, the circuit's depth and $\log |C|$. As in [IPS08], this extends to the case of a non-constant number of parties, in which case the communication complexity grows by an additional factor of $|C|\mathsf{poly}(n)$.

An interesting possibility for instantiating Π in practice would be to use an MPC protocol optimized for SIMD binary circuits such as MiniMAC [DZ13] or committed MPC [FPY18], which can further exploit the fact that all the AND gates can be computed in parallel. Currently, the preprocessing phases of these protocols are less practical than the TinyOT approach, but this may be an interesting future direction.

Protocol Using TinyOT. TinyOT is currently the most practical approach to secret-sharing-based MPC on binary circuits, so using this gives a highly efficient protocol for constant-round secure computation. Our TinyOT-based construction avoids the need for any additional OTs, and the cost is essentially the same as that of TinyOT. However, the protocol is worse asymptotically, since TinyOT costs either $O(|C|B\kappa n^2)$ (using [WRK17a, WRK17b]), or $O(|C|B^2\kappa n^2)$ (with [FKOS15]). The parameter B is in practice around 3–5 for large enough circuits, or asymptotically $B = O(1 + s/\log |C|)$ for statistical security parameter s.

Technical Highlights. The main technical difference between our approaches and previous work, is that in our preprocessing phase the parties produce an *unauthenticated* additively shared garbled circuit, which is opened in the online phase. This means the adversary can introduce an arbitrary additive error into the garbled circuit, and even pick the error *adaptively*, which requires a new analysis and proof of the online phase. Prior constructions only allowed a more restrictive type of error that was independent of the garbled circuit, and the protocols were much more expensive as they created authenticated sharings of the garbled circuit.

Secondly, we devise a new consistency check to enforce correctness of inputs to correlated OT, which is based on very efficient linear operations similar to recent advances in homomorphic commitments [CDD+16]. This check, combined with our improved error analysis for the online phase, allows the secret-shared garbled circuit to be created without authenticating any of the parties' keys or PRF values, which removes a significant cost from previous works.

Implementation. We demonstrate the practicality of our TinyOT-based protocol with an implementation, and perform experiments with up to 9 parties securely computing the AES and SHA-256 circuits. In a 1Gbps LAN setting, we can securely compute the AES circuit with 9 parties in just 620ms. This improves upon the best possible performance that would be attainable using [LPSY15] by around 60 times. The details of our implementation can be found in Section 6.

1.1.1 Comparison with Other Approaches

Table 1 shows how the communication complexity of our work compares with other actively secure, constant-round protocols. As mentioned earlier, most previous constructions express the garbling function as an arithmetic circuit over a large finite field. In these protocols, garbling even a single AND gate requires computing $O(n)$ multiplications over a large field with SHE or MPC. This means they scale at least cubically in the number of parties. In contrast, our protocol only requires one $\mathbb{F}_2$ multiplication per AND gate, so scales with $O(n^2)$. Previous SHE-based protocols also require zero-knowledge proofs of plaintext knowledge of SHE ciphertexts, which in practice are very costly. Note that the recent MASCOT protocol [KOS16] for secure computation of arithmetic circuits could also be used in [LPSY15], instead of SHE, but this still has very high communication costs. We denote by MASCOT-BMR-FX an optimized variant of [LPSY15], modified to use free-XOR as in our protocol, with multiplications in $\mathbb{F}_{2^\kappa}$ done using MASCOT. Finally, the recent concurrent work by Wang et al. [WRK17b] is based on an optimized variant of TinyOT, with comparable performance to our approach as well as an implementation; see Section 1.2.1 for a detailed comparison.

None of these previous works have reported implementations at the time of writing, but our implementation of the TinyOT-based protocol improves upon the best times that would be achievable with SPDZ-BMR and MASCOT by up to 60x. This is because our protocol has lower communication costs than [LPSY15] (by at least 2 orders of magnitude) and the main computational costs are from standard symmetric primitives, so far cheaper than using SHE.

Overall, our protocols significantly narrow the gap between the cost of constant-round and many-round MPC protocols for binary circuits. More specifically, this implies that, with current techniques, constant round MPC for binary circuits is not much more expensive than practical, non-constant round protocols. Additionally, both of our protocols have potential for future improvement by optimizing existing non-constant round protocols: a practical implementation of MiniMAC [DZ13] would lead to a very efficient approach with our generic protocol, whilst any future improvements to multi-party TinyOT would directly give a similar improvement to our second protocol.

1.2 Technical Overview

Background on BMR. We first give a brief summary of multi-party garbled circuit constructions based on BMR, focusing on the semi-honest protocol by Ben-Afraim et al. [BLO16], which we build upon.

Recall that in a traditional two-party garbled circuit, for every wire w of the circuit the garbler picks two random keys $k_{w,0}, k_{w,1}$. Let E be some two-key encryption algorithm. An AND gate is then garbled by

Protocol	Based on	Free XOR	Comm. per Garbled Gate
SPDZ-BMR [LPSY15]	SHE + ZKPoPK	✗	$O(n^4\kappa)$
SHE-BMR [LSS16]	SHE (depth 4) + ZKPoPK	✗	$O(n^3\kappa)$
MASCOT-BMR-FX [KOS16]	OT	✓	$O(n^3\kappa^2)$
This work §3	OT + [IPS08]	✓	$O(n^2\kappa + \mathsf{poly}(n))$
This work §4	TinyOT	✓	$O(n^2B^2\kappa)$
[WRK17b] (concurrent)	Optimized TinyOT	✓	$O(n^2B\kappa)$

Table 1: Comparison of actively secure, constant round MPC protocols. $B = O(1+s/\log|C|)$ is a cut-and-choose parameter, which in practice is between 3–5. Our second protocol can also be based upon optimized TinyOT to obtain the same complexity as [WRK17b].

computing and randomly permuting the four following ciphertexts:

$$E_{k_{u,0},k_{v,0}}(k_{w,0}), \quad E_{k_{u,0},k_{v,1}}(k_{w,0})$$
$$E_{k_{u,1},k_{v,0}}(k_{w,0}), \quad E_{k_{u,1},k_{v,1}}(k_{w,1}).$$

In the n-party version with BMR, the high-level idea is that instead of having separate parties who garble and evaluate the circuit, all parties will generate the garbled circuit *in a distributed manner*. Crucially, this is done in such a way that no strict subset of the n parties knows all the keys or the way in which the ciphertexts were permuted. After doing this in a preprocessing phase, there is an online phase where the garbled circuit and the appropriate input wire keys are revealed to all parties, who perform the evaluation locally.

To generate the garbled circuit efficiently, the basic scheme is modified so that each wire key $k_{w,b}$, for $b \in \{0,1\}$, is the concatenation of n separate keys $k^1_{w,b}, \ldots, k^n_{w,b}$. Each party P_i samples the key $k^i_{w,b}$. We then use the encryption scheme

$$E_{k_{u,a},k_{v,b}}(k^j_{w,c}) := \left(\bigoplus_{i=1}^{n} F_{k^i_{u,a},k^i_{v,b}}(g\|j)\right) \oplus k^j_{w,c}$$

where g is the index of the gate being garbled, and F is a pseudorandom function that takes two random keys (and is pseudorandom as long as one key remains private). Note that we need $4n$ such encryptions per gate, so the garbled circuit is n times larger than the two-party case.

We also use the free-XOR optimization [KS08], where the keys are picked so that every pair of keys for wire w satisfies $k_{w,1} = k_{w,0} \oplus R$, for a fixed offset $R = (R^1, \ldots, R^n)$ where R^i is known only to party P_i. This allows garbling XOR gates at no cost, so we only need to deal with AND gates.

Putting these together, in the construction of Ben-Efraim et al. [BLO16], a garbling of the g-th AND gate with input wires u, v and output wire w, consists of the $4n$ values, for $j = 1, \ldots, n$ and $a, b \in \{0,1\}$:

$$\tilde{g}^j_{a,b} = \left(\bigoplus_{i=1}^{n} F_{k^i_{u,a},k^i_{v,b}}(g\|j)\right) \oplus k^j_{w,0} \tag{1}$$
$$\oplus \left(R^j((\lambda_u \oplus a)(\lambda_v \oplus b) \oplus \lambda_w)\right), \quad (a,b) \in \{0,1\}^2$$

The values $\lambda_u, \lambda_v, \lambda_w \in \{0,1\}$ are called the *wire masks*, which are secret-shared random bits that serve to permute the ciphertexts.

Notice that since P_i knows the keys $k^i_{u,a}, k^i_{v,b}$, the PRF values $F_{k^j_{u,a},k^j_{v,b}}(\cdot)$ can be provided as input by party P_j to an MPC protocol for computing (1). This avoids the need to evaluate the PRF within MPC, and the protocol only needs to securely evaluate a simple, degree-3 function. As shown in [BLO16], this can be done with semi-honest security with just a few oblivious transfers, and in constant rounds.

To obtain active security, the original BMR protocol [BMR90] requires each party P_j to prove in zero-knowledge that the inputs to the secure computation were correctly computed as outputs of the PRF. The so-called 'SPDZ-BMR' protocol [LPSY15] showed that these are not actually needed to achieve active security with abort, and it suffices to compute a functionality which allows corrupt parties to choose their own PRF values as inputs to the computation. This works, because when evaluating the garbled circuit in the online phase, each party P_j can check that the decryption of the j-th entry in every garbled gate gives one of the keys $k^j_{w,0}$, $k^j_{w,1}$ and this check would overwhelmingly fail if any PRF value was incorrect. It further implies that the adversary cannot flip the value transmitted through some wire as that would require guessing a key. With this optimization, the main cost of SPDZ-BMR is the $O(n)$ actively secure MPC multiplications, which were used to compute an authenticated secret sharing of (1) using the SPDZ protocol based on information-theoretic MACs.

Our Approach. In our work, we simplify SPDZ-BMR further, observing that it is enough to compute an *unauthenticated* additive secret sharing of the garbled circuit. Opening the garbled circuit with unauthenticated shares allows a corrupt party to introduce further errors into the garbling by changing their share, even *after learning the correct garbled circuit*, since we may have a rushing adversary. Nevertheless, we prove that the BMR online phase remains secure when this type of error is allowed, and it would only lead to an abort.

The major benefit of this optimization is that we no longer need to authenticate the PRF values and keys in equation (1) with SPDZ-style MACs (as in [LPSY15]) or SHE ciphertexts (as in [LSS16]), and this greatly simplifies the preprocessing stage. We demonstrate this with two different instantiations of the preprocessing, which significantly improve upon previous works.

Preprocessing with General Bit-MPC (Section 3). Our first instantiation uses any actively secure MPC protocol for binary circuits, which we capture by a functionality $\mathcal{F}_{\text{BitMPC}}$, combined with oblivious transfer. We start with the observation in [BLO16] that the multiplications we need to compute securely are either between two secret-shared bits, or a secret-shared bit and a fixed, secret string. So, we do not need the full power of an MPC protocol for arithmetic circuit evaluation over $\mathbb{F}_{2^\kappa}$ or $\mathbb{F}_p$ (for large p), as used in previous works.

To compute shares of the product of wire masks $\lambda_u \cdot \lambda_v$, we use the functionality $\mathcal{F}_{\text{BitMPC}}$. This is only needed for computing *one secure AND per garbled AND gate*, since all 4 bit products in $\tilde{g}^j_{a,b}$ can be computed as linear combinations of $\lambda_u \cdot \lambda_v$, λ_u and λ_v. We then need to multiply the resulting secret-shared bits by each string R^j, known to P_j, which we do using actively secure correlated OT between P_j and every other party, where P_j inputs R^j as the (fixed) OT correlation and P_i inputs a share of one of the λ bits. This step costs $O(n^2)$ OTs per gate. Compared with the previous SPDZ-BMR approach, which used $O(n)$ SPDZ multiplications over a large field to compute these products, for a total cost in $O(n^3)$, we get significant savings.

However, so far this is not actively secure: we have no guarantee that the λ shares input to the OTs are consistent with the shares input to $\mathcal{F}_{\text{BitMPC}}$, and also a corrupt P_j could input different R^j strings to the

OTs with different parties. To prevent this we devise a new consistency check, where we apply a random, universal linear hash function to the OT outputs, before masking and opening them. We then apply the same linear function to the bits that were input to $\mathcal{F}_{\text{BitMPC}}$, open this, and check that both sets of hashes are consistent.

Finally, after performing the bit/string products and checking consistency, each party has enough information to locally compute an additive share of the whole garbled circuit. One small technical detail is that to prove security of the preprocessing phase, we then need to rerandomize the shares by adding pseudorandom shares of zero.

Preprocessing with TinyOT (Section 4). The second method is similar to the first, but *completely removes* the need for the bit/string multiplications and consistency check, by using a 'TinyOT'-style protocol [FKOS15, BLN+15] based on pairwise information-theoretic MACs to instantiate $\mathcal{F}_{\text{BitMPC}}$. We exploit the specific structure of TinyOT MACs on the shared λ values to compute the bit/string products with no interaction, provided each party's MAC key is chosen to be the same string R^i used in the garbling. This reduces costs since we do not need any additional OTs, or the consistency check.

The version based on TinyOT is our most practical constant-round protocol, although it is plausible that the first, more general approach could be more efficient in the future. For instance, if an alternative way to instantiate $\mathcal{F}_{\text{BitMPC}}$ becomes much more practical than TinyOT, then this could be plugged into the first approach, instead of using TinyOT.

1.2.1 Concurrent Work

Two recent works by Wang, Ranellucci and Katz introduced constant round, two-party [WRK17a] and multi-party [WRK17b] protocols based on *authenticated garbling*, with a preprocessing phase that can be instantiated based on TinyOT. Our work is conceptually quite similar, since both involve generating a garbled circuit in a distributed manner using TinyOT. The main difference seems to be that our protocol is symmetric, since all parties evaluate the same garbled circuit. With authenticated garbling, the garbled circuit is only evaluated by one party. This makes the garbled circuit slightly smaller, since there are $n-1$ sets of keys instead of n, but the online phase requires at least one more round of interaction (if all parties learn the output). The works of Wang et al. also contain concrete and asymptotic improvements to the two-party and multi-party TinyOT protocols, which improve upon our protocol in Appendix A by a factor of $O(s/\log|C|)$ where s is a statistical parameter. These improvements can be directly plugged into our second garbling protocol, to obtain a protocol with similar concrete efficiency.

We remark that the two-party protocol in [WRK17a] inspired our use of TinyOT MACs to perform the bit/string multiplications in our protocol from Section 4. The rest of our work is independent. Another difference is that our protocol from Section 3 is more generic, since $\mathcal{F}_{\text{BitMPC}}$ can be implemented with *any* secret-sharing-based bit-MPC protocol, whereas the multi-party paper [WRK17b] is only based on TinyOT. For a more detailed comparison of concrete efficiency, see Section 6.

1.2.2 Subsequent Work

Subsequently to our work, Aly et al. [AOR+19] have extended our protocol to support reactive computations, where the inputs and state of one secure computation can later be reused in another. Boyle et al. [BCG+19a, BCG+19b] presented new methods of performing oblivious transfer with greatly reduced communication, called silent OT extension, which can be applied to our protocols. With this approach, the random, correlated

OTs needed in our protocol cost as little as 0.1 bits of communication [BCG+19b], instead of 168 bits as in our implementation, at the cost of increased local computation and relying on the LPN assumption. Using this for the TinyOT protocol described in Appendix A, the communication cost of the TinyOT preprocessing drops by around 1–2 orders of magnitude, and we estimate would in fact have less communication than the opening of the garbled circuit in the online phase.

In other works, the underlying TinyOT protocol has been optimized in both the two-party [KRRW18] and multi-party [YWZ19] settings, with improvements that reduce communication and computation. Note, however, that using silent OT in these protocols (as well as the one from [WRK17b]) would have less of an impact than with the version of TinyOT we use, since these protocols additionally require sending several 128-bit hash values, which would become the communication bottleneck.

Paper Outline. The paper is structured as follows. In Section 2 we provide basic definitions and preliminaries. In Section 3 we describe our generic method for preprocessing the garbled circuit whereas in Section 4 and Appendix A we describe a concretely more efficient way to realize the same preprocessing functionality. Section 5 introduces the online MPC phase while calling the preprocessing functionality. Finally, our implementation results are discussed in Section 6.

2 Preliminaries

We denote the computational security parameter by κ. We say that a function $\mu : \mathbb{N} \to \mathbb{N}$ is *negligible* if for every positive polynomial $p(\cdot)$ and all sufficiently large κ it holds that $\mu(\kappa) < \frac{1}{p(\kappa)}$. We use the abbreviation PPT to denote probabilistic polynomial-time. We further denote by $a \leftarrow A$ the uniform sampling of a from a set A, and by $[d]$ the set of elements $(1, \ldots, d)$. We often view bit-strings in $\{0,1\}^k$ as vectors in $\mathbb{F}_2^k$, depending on the context, and denote exclusive-or by "$\oplus$" or "$+$". If $a, b \in \mathbb{F}_2$ then $a \cdot b$ denotes multiplication (or AND), and if $\boldsymbol{c} \in \mathbb{F}_2^\kappa$ then $a \cdot \boldsymbol{c} \in \mathbb{F}_2^\kappa$ denotes the product of a with every component of $\boldsymbol{c}$. We similarly use $\cdot$ to denote matrix or matrix-vector multiplication.

For column vectors $\boldsymbol{x} = (x_1, \ldots, x_n) \in \mathbb{F}_2^n$ and $\boldsymbol{y} \in \mathbb{F}_2^m$, the *tensor product* (or *outer product*) $\boldsymbol{x} \otimes \boldsymbol{y}$ is defined as the $n \times m$ matrix over $\mathbb{F}_2$ where the i-th row is $x_i \cdot \boldsymbol{y}$, or equivalently $\boldsymbol{x} \cdot \boldsymbol{y}^\top$.

We next specify the definition of computational indistinguishability.

Definition 2.1 *Let $X = \{X(a,\kappa)\}_{a\in\{0,1\}^*,\kappa\in\mathbb{N}}$ and $Y = \{Y(a,\kappa)\}_{a\in\{0,1\}^*,\kappa\in\mathbb{N}}$ be two distribution ensembles. We say that X and Y are* computationally indistinguishable*, denoted $X \overset{c}{\approx} Y$, if for every probabilistic polynomial time (PPT) distinguisher $\mathcal{D}$ and every $a \in \{0,1\}^*$, there exists a negligible function* negl *such that:*

$$\left|\Pr\left[\mathcal{D}(X(a,\kappa), a, 1^\kappa) = 1\right] - \Pr\left[\mathcal{D}(Y(a,\kappa), a, 1^\kappa) = 1\right]\right| < \mathsf{negl}(\kappa).$$

2.1 Circular 2-Correlation Robust PRF

The BMR garbling technique from [LPSY15] is proven secure based on a pseudorandom function (PRF) with multiple keys. When the keys are independently chosen then security with multiple keys is implied by the standard security PRF notion, by a simple hybrid argument. However, since our scheme supports free-XOR, we require assuming a stronger notion, discussed next. We adapt the definition of correlation robustness with circularity from [CKKZ12] given for hash functions to double-key PRFs. This definition captures the related key and circularity requirements induced by supporting the free-XOR technique. Formally, fix some function $F : \{0,1\}^n \times \{0,1\}^\kappa \times \{0,1\}^\kappa \mapsto \{0,1\}^\kappa$. We define an oracle Circ_R as follows:

- $\mathsf{Circ}_R(k_1, k_2, g, j, b_1, b_2, b_3)$ outputs $F_{k_1 \oplus b_1 R, k_2 \oplus b_2 R}(g \| j) \oplus b_3 R$.

The outcome of oracle Circ is compared with the a random string of the same length computed by an oracle Rand:

- $\mathsf{Rand}(k_1, k_2, g, j, b_1, b_2, b_3)$: if this input was queried before then return the answer given previously. Otherwise choose $u \leftarrow \{0,1\}^\kappa$ and return u.

Definition 2.2 (Circular 2-correlation robust PRF) *A PRF F is* circular 2-correlation robust *if for any PPT distinguisher $\mathcal{D}$ making legal queries to its oracle, there exists a negligible function* negl *such that:*

$$\left| \Pr[R \leftarrow \{0,1\}^\kappa; \mathcal{D}^{\mathsf{Circ}_R(\cdot)}(1^\kappa) = 1] - \Pr[\mathcal{D}^{\mathsf{Rand}(\cdot)}(1^\kappa) = 1] \right| \leq \mathsf{negl}(\kappa).$$

As in [CKKZ12], some trivial queries must be ruled out. Specifically, the distinguisher is restricted as follows: (1) it is not allowed to make any query of the form $\mathcal{O}(k_1, k_2, g, j, 0, 0, b_3)$ (since it can compute $F_{k_1,k_2}(g\|j)$ on its own) and (2) it is not allowed to query both tuples $\mathcal{O}(k_1, k_2, g, j, b_1, b_2, 0)$ and $\mathcal{O}(k_1, k_2, g, j, b_1, b_2, 1)$ for any values k_1, k_2, g, j, b_1, b_2 (since that would allow it to trivially recover the global difference). We say that any distinguisher respecting these restrictions makes legal queries.

2.2 Almost-1-Universal Linear Hashing

We use a family of almost-1-universal linear hash functions over $\mathbb{F}_2$, defined by:

Definition 2.3 (Almost-1-Universal Linear Hashing) *We say that a family $\mathcal{H}$ of linear functions $\mathbb{F}_2^m \to \mathbb{F}_2^s$ is ε-almost 1-universal, if it holds that for every non-zero $\boldsymbol{x} \in \mathbb{F}_2^m$ and for every $\boldsymbol{y} \in \mathbb{F}_2^s$:*

$$\Pr_{\mathbf{H} \leftarrow \mathcal{H}}[\mathbf{H}(\boldsymbol{x}) = \boldsymbol{y}] \leq \varepsilon$$

where $\mathbf{H}$ is chosen uniformly at random from the family $\mathcal{H}$. We will identify functions $\mathbf{H} \in \mathcal{H}$ with their $s \times m$ transformation matrix, and write $\mathbf{H}(\boldsymbol{x}) = \mathbf{H} \cdot \boldsymbol{x}$.

This definition is slightly stronger than a family of almost-universal linear hash functions (where the above need only hold for $\boldsymbol{y} = 0$, as in [CDD+16]). However, this is still much weaker than *2-universality* (or *pairwise independence*), which a linear family of hash functions cannot achieve, because $\mathbf{H}(0) = 0$ always. The two main properties affecting the efficiency of a family of hash functions are the *seed size*, which refers to the length of the description of a random function $\mathbf{H} \leftarrow \mathcal{H}$, and the *computational complexity* of evaluating the function. The simplest family of almost-1-universal hash functions is the set of all $s \times m$ matrices; however, this is not efficient as the seed size and complexity are both $O(m \cdot s)$. Recently, in [CDD+16], it was shown how to construct a family with seed size $O(s)$ and complexity $O(m)$, which is asymptotically optimal. A more practical construction is a polynomial hash based on GMAC (used also in [NST17]), described as follows (here we assume that s divides m, for simplicity):

- Sample a random seed $\alpha \leftarrow \mathbb{F}_{2^s}$.
- Define $\mathbf{H}_\alpha$ to be the function:

  $$\mathbf{H}_\alpha : \mathbb{F}_{2^s}^{m/s} \to \mathbb{F}_{2^s}, \quad \mathbf{H}_\alpha(x_1, \ldots, x_{m/s}) = \alpha \cdot x_1 + \alpha^2 \cdot x_2 + \cdots + \alpha^{m/s} \cdot x_{m/s}.$$

  Note that by viewing elements of $\mathbb{F}_{2^s}$ as vectors in $\mathbb{F}_2^s$, multiplication by a fixed field element $\alpha^i \in \mathbb{F}_{2^s}$ is linear over $\mathbb{F}_2$. Therefore, $\mathbf{H}_\alpha$ can be seen as an $\mathbb{F}_2$-linear map, represented by a unique matrix in $\mathbb{F}_2^{s \times m}$.

Here, the seed is short, but the computational complexity is $O(m \cdot s)$. However, in practice when $s = 128$ the finite field multiplications can be performed very efficiently in hardware on modern CPUs. Note that this gives a 1-universal family with $\varepsilon = \frac{m}{s} \cdot 2^{-s}$. This can be improved to 2^{-s} (i.e. perfect), at the cost of a larger seed, by using m/s distinct elements α_i, instead of powers of α.

2.3 Security Model

Universal composability. We prove security of our protocols in the universal composability (UC) framework [Can01] (see also [CCL15] for a simplified version of UC). This framework is based on the real/ideal paradigm, where all the entities (including the parties and the adversary) are modeled as interactive Turing machines. The goal of a protocol is defined by an *ideal functionality*, which can be seen as a trusted party sending the desired results to the parties. To prove security of a protocol, we aim to show that any adversary attacking the real protocol can be used to construct a corresponding ideal adversary, called the *simulator*, that runs in the ideal world, interacting only with the functionality $\mathcal{F}$ and the real adversary, such that the distributions of messages seen in the real world and ideal world executions are indistinguishable. The UC framework additionally defines a powerful entity called the *environment*, which is the interactive machine trying to distinguish the two worlds. The environment has total control over the adversary, and can choose the inputs, and see the outputs, of *all* parties.

We denote by $\mathbf{REAL}_{\pi,\mathcal{A},\mathcal{Z}}(1^\kappa, z)$ the output distribution of the environment $\mathcal{Z}$ in the real world execution of protocol π, with n parties and an adversary $\mathcal{A}$, where κ is the security parameter and z is the auxiliary input to $\mathcal{Z}$. The output distribution of $\mathcal{Z}$ in the ideal world is denoted by $\mathbf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(1^\kappa, z)$, where $\mathcal{F}$ is the ideal functionality to be realized and $\mathcal{S}$ is the simulator. Additionally, we denote the hybrid execution of a protocol π, which is given access to an ideal functionality $\mathcal{G}$, by $\mathbf{HYB}^{\mathcal{G}}_{\pi,\mathcal{A},\mathcal{Z}}(1^\kappa, z)$. This is defined similarly to the real execution, and is known as the $\mathcal{G}$-hybrid model. Security of a protocol is then defined as follows.

Definition 2.4 *A protocol π UC-securely computes an ideal functionality $\mathcal{F}$ in the $\mathcal{G}$-hybrid model if for any PPT adversary $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$ such that for any PPT environment $\mathcal{Z}$, it holds that:*

$$\mathbf{HYB}^{\mathcal{G}}_{\pi,\mathcal{A},\mathcal{Z}} \overset{\text{c}}{\approx} \mathbf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}.$$

The *composition theorem* provides security guarantees when protocols are composed together. This means that if ρ is a UC-secure protocol for $\mathcal{G}$, then the protocol π in the $\mathcal{G}$-hybrid model can be replaced by the composition $\pi \circ \rho$. Informally, the composition theorem then guarantees that $\mathbf{REAL}_{\pi\circ\rho,\mathcal{A},\mathcal{Z}}$ is indistinguishable from $\mathbf{HYB}^{\mathcal{G}}_{\pi,\mathcal{A},\mathcal{Z}}$.

Communication model. We assume all parties are connected via authenticated communication channels, as well as secure point-to-point channels and a broadcast channel. The default method of communication in our protocols is authenticated channels, unless otherwise specified. Note that in practice, these can all be implemented with standard techniques (in particular, for broadcast a simple 2-round protocol suffices, since we allow abort [GL05]).

Adversary model. The adversary model we consider is a static, active adversary who corrupts up to $n-1$ out of n parties. This means that the identities of the corrupted parties are fixed at the beginning of the protocol, and they may deviate arbitrarily from the protocol.

2.4 Commitment Functionality

We require a UC commitment functionality $\mathcal{F}_{\text{Commit}}$ (Figure 1). This can easily be implemented in the random oracle model by defining $\texttt{Commit}(x, P_i) = \mathsf{H}(x, i, r)$, where H is a random oracle and $r \leftarrow \{0,1\}^\kappa$.

Functionality $\mathcal{F}_{\text{Commit}}$

Commit: On input $(\texttt{Commit}, x, i, \tau_x)$ from P_i, store (x, i, τ_x) and output (i, τ_x) to all parties.

Open: On input $(\texttt{Open}, i, \tau_x)$ by P_i, output (x, i, τ_x) to all parties.

If instead $(\texttt{NoOpen}, i, \tau_x)$ is given by the adversary, and P_i is corrupt, the functionality outputs $(\perp, i, \tau_x)$ to all parties.

Figure 1: Commitments functionality.

2.5 Coin-Tossing Functionality

We use a standard coin-tossing functionality, $\mathcal{F}_{\text{Rand}}$ (Figure 2), which can be implemented with UC commitments to random values.

Functionality $\mathcal{F}_{\text{Rand}}$

Upon receiving $(\texttt{rand}, S)$ from all parties, where S is any efficiently sampleable set, sample $r \leftarrow S$, send r to $\mathcal{A}$ and wait for its input. If $\mathcal{A}$ inputs $\texttt{OK}$ then output r to all parties, otherwise output $\perp$.

Figure 2: Coin-tossing functionality.

2.6 Correlated Oblivious Transfer

In this work we use an actively secure protocol for oblivious transfer (OT) on correlated pairs of strings of the form $(a_i, a_i \oplus \Delta)$, where Δ is fixed for every OT. The TinyOT protocol [NNOB12] for secure two-party computation constructs such a protocol, and a significantly optimized version of this is given in [NST17]. The communication cost is roughly $\kappa + s$ bits per OT. The ideal functionality is shown in Figure 3.

Fixed Correlation OT Functionality $\mathcal{F}_{\text{COT}}$

Initialize: Upon receiving $(\texttt{init}, \Delta)$, where $\Delta \in \{0,1\}^\kappa$ from P_S and $(\texttt{init})$ from P_R, store Δ. Ignore any subsequent $(\texttt{init})$ commands.

Extend: Upon receiving $(\texttt{extend}, x_1, \ldots, x_m)$ from P_R, where $x_i \in \{0,1\}$, and $(\texttt{extend})$ from P_S, do the following:

- Sample $t_i \in \{0,1\}^\kappa$, for $i \in [m]$. If P_R is corrupted then wait for $\mathcal{A}$ to input t_i.
- Compute $q_i = t_i + x_i \cdot \Delta$, for $i \in [m]$.
- If P_S is corrupted then wait for $\mathcal{A}$ to input $q_i \in \{0,1\}^\kappa$ and recompute $t_i = q_i + x_i \cdot \Delta$.
- Output t_i to P_R and q_i to P_S, for $i \in [m]$.

Figure 3: Fixed correlation oblivious transfer functionality between a sender P_S and receiver P_R.

The Bit MPC Functionality - $\mathcal{F}_{\text{BitMPC}}$

The functionality runs with parties $P_1, \ldots, P_n$ and an adversary $\mathcal{A}$ that controls a subset of parties $I \subset [n]$. The functionality maintains a dictionary, $\texttt{Val} \leftarrow \{\}$, to keep track of values in $\mathbb{F}_2$.

Input: On receiving $(\texttt{Input}, \texttt{id}_1, \ldots, \texttt{id}_\ell, x_1, \ldots, x_\ell, P_j)$ from party P_j and $(\texttt{Input}, \texttt{id}_1, \ldots, \texttt{id}_\ell, P_j)$ from all other parties, where $x_i \in \mathbb{F}_2$ and $\texttt{id}_i$ are distinct, store $\texttt{Val}[\texttt{id}_i] \leftarrow x_i$ for $i \in [\ell]$.

Add: On input $(\mathsf{Add}, \overline{\texttt{id}}, \texttt{id}_1, \ldots, \texttt{id}_\ell)$ from all parties, where $(\texttt{id}_1, \ldots, \texttt{id}_\ell)$ are keys in $\texttt{Val}$, set $\texttt{Val}[\overline{\texttt{id}}] \leftarrow \sum_{i=1}^{\ell} \texttt{Val}[\texttt{id}_i]$.

Multiply: On input $(\texttt{multiply}, \overline{\texttt{id}}, \texttt{id}_1, \texttt{id}_2)$ from all parties, where $(\texttt{id}_1, \texttt{id}_2)$ are keys in $\texttt{Val}$, compute $y \leftarrow \texttt{Val}[\texttt{id}_1] \cdot \texttt{Val}[\texttt{id}_2]$. Receive shares $y^i \in \mathbb{F}_2$ from $\mathcal{A}$, for $i \in I$, then sample random honest parties' shares $y^j \in \mathbb{F}_2$, for $j \notin I$, such that $\sum_{i=1}^{n} y^i = y$. Send y^i to party P_i, for $i \in [n]$, and store the value $\texttt{Val}[\overline{\texttt{id}}] \leftarrow y$.

Open: On input $(\mathsf{Open}, \texttt{id})$ from all parties, where $\texttt{id}$ is a key in $\texttt{Val}$, send $x \leftarrow \texttt{Val}[\texttt{id}]$ to $\mathcal{A}$. Wait for an input from $\mathcal{A}$. If it inputs $\texttt{OK}$ then output x to all parties, otherwise output $\perp$ and terminate.

Figure 4: Functionality for GMW-style MPC for binary circuits.

2.7 Functionality for Secret-Sharing-Based MPC

We make use of a general, actively secure protocol for secret sharing-based MPC for binary circuits, which is modeled by the functionality $\mathcal{F}_{\text{BitMPC}}$ in Figure 4. This functionality allows parties to provide private inputs, which are then stored and can be added or multiplied internally by $\mathcal{F}_{\text{BitMPC}}$, and revealed if desired. Note that we also need the **Multiply** command to output a random additive secret sharing of the product to all parties; this essentially assumes that the underlying protocol is based on secret sharing.

We use the notation $\langle x \rangle$ to represent a secret-shared value x that is stored internally by $\mathcal{F}_{\text{BitMPC}}$, and define x^i to be party P_i's additive share of x (if it is known). We also define the $+$ and $\cdot$ operators on two shared values $\langle x \rangle, \langle y \rangle$ to call the **Add** and **Multiply** commands of $\mathcal{F}_{\text{BitMPC}}$, respectively, and return the identifier associated with the result.

2.8 BMR Garbling

The BMR garbling technique by Beaver, Micali and Rogaway [BMR90] involves garbling each gate separately using pseudorandom generators while ensuring consistency between the wires. This method was recently improved in a sequence of works [LPSY15, LSS16, BLO16], where the latter work further supports the free XOR property. Recall from our explanation in Section 1.2 that the main task of generating the garbled circuit is to compute, for each AND gate g with input wires u, v and output wire w, the $4n$ values:

$$\tilde{g}^j_{a,b} = \left(\bigoplus_{i=1}^{n} F_{k^i_{u,a}, k^i_{v,b}}(g \| j) \right) \oplus k^j_{w,0} \\ \oplus \left(R^j \cdot ((\lambda_u \oplus a) \cdot (\lambda_v \oplus b) \oplus \lambda_w) \right), \quad (a,b) \in \{0,1\}^2,\ j \in [n]$$

where the wire masks $\lambda_u, \lambda_v, \lambda_w \in \{0,1\}$ are secret-shared between all parties, while the PRF keys $k^j_{u,a}, k^j_{v,b}$ and the global difference string R^j are known only to party P_j.

3 Generic Protocol for Multi-Party Garbling

We now describe our generic method for creating the garbled circuit using any secret-sharing-based MPC protocol (modeled by $\mathcal{F}_{\text{BitMPC}}$) and the correlated OT functionality $\mathcal{F}_{\text{COT}}$. We first describe the functionality in Section 3.1 and the protocol in Section 3.2, and then analyse its security in Sections 3.4–3.5.

3.1 The Preprocessing Functionality

The preprocessing functionality, formalized in Figure 5, captures the generation of the garbled circuit as well as an error introduced by the adversary. The adversary is allowed to submit additive errors, chosen adaptively after seeing the garbled circuit, which are added by the functionality to each entry when the garbled circuit is opened.

3.2 Protocol Overview

The garbling protocol, shown in Figure 6, proceeds in three main stages. Firstly, the parties locally sample all of their keys and shares of wire masks for the garbled circuit. Secondly, the parties compute shares of the products of the wire masks and each party's global difference string; these are then used by each party to locally obtain a share of the entire garbled circuit. Finally, the bit masks for the output wires are opened to all parties. The opening of the garbled circuit is shown in Figure 7.

Concretely, each party P_i starts by sampling a global difference string $R^i \leftarrow \{0,1\}^\kappa$, and for each wire w which is an output wire of an AND gate, or an input wire, P_i also samples the keys $k^i_{w,0}$, $k^i_{w,1} = k^i_{w,0} \oplus R^i$ and an additive share of the wire mask, $\lambda^i_w \leftarrow \mathbb{F}_2$. As in [BLO16], we let P_i input the actual wire mask (instead of a share) for every input wire associated with P_i's input.

In step 3, the parties compute additive shares of the bit products $\lambda_{uv} = \lambda_u \cdot \lambda_v \in \mathbb{F}_2$, and then, for each $j \in [n]$, shares of:

$$\lambda_u \cdot R^j, \quad \lambda_v \cdot R^j, \quad \lambda_{uvw} \cdot R^j \in \mathbb{F}_2^\kappa \tag{2}$$

where $\lambda_{uvw} := \lambda_{uv} \oplus \lambda_w$, and u, v and w are the input and output wires of AND gate g. We note that (as observed in [BLO16]) only one bit/bit product and $3n$ bit/string products are necessary, even though each gate has $4n$ entries, due to correlations between the entries, as can be seen below.

We compute the bit multiplications using the $\mathcal{F}_{\text{BitMPC}}$ functionality on the bits that are already stored by $\mathcal{F}_{\text{BitMPC}}$. To compute the bit/string multiplications in (2), we use correlated OT, followed by a consistency check to verify that the parties provided the correct shares of λ_w and correlation R^i to each $\mathcal{F}_{\text{COT}}$ instance; see Section 3.3 for details.

Using shares of the bit/string products, the parties can locally compute an unauthenticated additive share of the entire garbled circuit (steps 3c–4). First, for each of the four values $(a, b) \in \{0,1\}^2$, each party P_i computes the share

$$\rho^i_{j,a,b} = \begin{cases} a \cdot (\lambda_v \cdot R^j)^i \oplus b \cdot (\lambda_u \cdot R^j)^i \oplus (\lambda_{uvw} \cdot R^j)^i & \text{if } i \neq j \\ a \cdot (\lambda_v \cdot R^j)^i \oplus b \cdot (\lambda_u \cdot R^j)^i \oplus (\lambda_{uvw} \cdot R^j)^i \oplus a \cdot b \cdot R^j & \text{if } i = j \end{cases}$$

These define additive shares of the values

$$\begin{aligned} \rho_{j,a,b} &= R^j \cdot (a \cdot \lambda_v \oplus b \cdot \lambda_u \oplus \lambda_{uvw} \oplus a \cdot b) \\ &= R^j \cdot ((\lambda_u \oplus a) \cdot (\lambda_v \oplus b) \oplus \lambda_w) \end{aligned}$$

The Preprocessing Functionality

Let F be a circular 2-correlation robust PRF. The functionality runs with parties $P_1, \ldots, P_n$ and an adversary $\mathcal{A}$, who corrupts a subset $I \subset [n]$ of parties.

Garbling: On input $(\texttt{Garbling}, C_f)$ from all parties, where C_f is a boolean circuit, denote by W its set of wires, W_{in_i} its set of input wires for party P_i, W_{out} its set of output wires and G its set of AND gates. Then the functionality does as follows:

- Sample a global difference $R^j \leftarrow \{0,1\}^\kappa$, for each $j \notin I$, and receive corrupt parties' strings $R^i \in \{0,1\}^\kappa$ from $\mathcal{A}$, for $i \in I$.
- Passing topologically through all the wires $w \in W$ of the circuit:
  - If $w \in W_{\mathsf{in}_i}$ for some i:
    1. Sample $\lambda_w \leftarrow \{0,1\}$. If P_i is corrupt, instead receive λ_w from $\mathcal{A}$.
    2. Sample a key $k^i_{w,0} \leftarrow \{0,1\}^\kappa$ if P_i is honest, and otherwise receive $k^i_{w,0}$ from $\mathcal{A}$. Define $k^i_{w,1} = k^i_{w,0} \oplus R^i$.
  - If w is the output of an AND gate:
    1. Sample $\lambda_w \leftarrow \{0,1\}$.
    2. Sample a key $k^j_{w,0} \leftarrow \{0,1\}^\kappa$, for each $j \notin I$, and receive corrupt parties' keys $k^i_{w,0}$ from $\mathcal{A}$, for $i \in I$. Set $k^i_{w,1} = k^i_{w,0} \oplus R^i$, for $i \in [n]$.
  - If w is the output of a XOR gate, and u and v its input wires:
    1. Compute and store $\lambda_w = \lambda_u \oplus \lambda_v$.
    2. For $i \in [n]$, set $k^i_{w,0} = k^i_{u,0} \oplus k^i_{v,0}$ and $k^i_{w,1} = k^i_{w,0} \oplus R^i$.
- For every AND gate $g \in G$, the functionality computes the $4n$ entries of the garbled version of g as:

$$\tilde{g}^j_{a,b} = \left(\bigoplus_{i=1}^{n} F_{k^i_{u,a}, k^i_{v,b}}(g \| j) \right) \oplus k^j_{w,0} \\ \oplus \left(R^j \cdot ((\lambda_u \oplus a) \cdot (\lambda_v \oplus b) \oplus \lambda_w) \right), \quad (a,b) \in \{0,1\}^2,\ j \in [n].$$

  For $(a,b) \in \{0,1\}^2$, set $\tilde{\mathbf{g}}_{a,b} = (\tilde{g}^1_{a,b}, \ldots, \tilde{g}^n_{a,b})$. The functionality stores the values $\tilde{\mathbf{g}}_{a,b}$.
- Wait for an input from $\mathcal{A}$. If it inputs $\texttt{OK}$ then output λ_w to all parties for each circuit-output wire $w \in W_{\mathsf{out}}$, and output to each P_i all the keys $\{k^i_{w,0}\}_{w \in W}$, and R^i. Otherwise, output $\perp$ and terminate.

Open Garbling: On receiving $(\texttt{OpenGarbling})$ from all parties, when the **Garbling** command has already run successfully, the functionality sends to $\mathcal{A}$ the values $\tilde{\mathbf{g}}_{a,b}$ for all $g \in G$ and waits for a reply.

- If $\mathcal{A}$ returns $\perp$ then the functionality aborts.
- Otherwise, the functionality receives $\texttt{OK}$ and an additive error $\boldsymbol{e} = \{e^{a,b}_g\}_{a,b \in \{0,1\}, g \in G}$ chosen by $\mathcal{A}$. Afterwards, it sends to all parties the garbled circuit $\tilde{\mathbf{g}}_{a,b} \oplus e^{a,b}_g$ for all $g \in G$ and $a, b \in \{0,1\}$.

Figure 5: The Preprocessing Functionality $\mathcal{F}_{\text{Prepocessing}}$.

The Preprocessing Protocol $\Pi_{\text{Preprocessing}}$ – Garbling Stage

Given a gate g, we denote by u (resp. v) its left (resp. right) input wire, and by w its output wire.
Let $F : \{0,1\}^{2\kappa} \times [|G|] \times [n] \to \{0,1\}^{\kappa}$ be a circular 2-correlation robust PRF.

Garbling:

1. Each party P_i samples a random global difference $R^i \leftarrow \mathbb{F}_2^{\kappa}$.
2. **Generate wire masks and keys:** Passing through the wires of the circuit topologically, proceed as follows:
   - If $w \in W_{\mathsf{in}_j}$ is a *circuit-input* wire for party P_j:
     (a) P_j calls **Input** on $\mathcal{F}_{\text{BitMPC}}$ with a randomly sampled $\lambda_w \in \{0,1\}$ to obtain $\langle \lambda_w \rangle$. P_j defines the share $\lambda_w^j = \lambda_w$, every other P_i sets $\lambda_w^i = 0$.
     (b) Every P_i samples a key $k_{w,0}^i \leftarrow \{0,1\}^{\kappa}$ and sets $k_{w,1}^i = k_{w,0}^i \oplus R^i$.
   - If the wire w is the output of an AND gate:
     (a) Each P_i calls **Input** on $\mathcal{F}_{\text{BitMPC}}$ with a randomly sampled $\lambda_w^i \leftarrow \{0,1\}$. The parties then compute the secret-shared wire mask as $\langle \lambda_w \rangle = \sum_{i \in [n]} \langle \lambda_w^i \rangle$.
     (b) Every P_i samples a key $k_{w,0}^i \leftarrow \{0,1\}^{\kappa}$ and sets $k_{w,1}^i = k_{w,0}^i \oplus R^i$.
   - If the wire w is the output of a XOR gate:
     (a) The parties compute the mask on the output wire as $\langle \lambda_w \rangle = \langle \lambda_u \rangle + \langle \lambda_v \rangle$.
     (b) Every P_i sets $k_{w,0}^i = k_{u,0}^i \oplus k_{v,0}^i$ and $k_{w,1}^i = k_{w,0}^i \oplus R^i$.
3. **Secure product computations:**
   (a) For each AND gate $g \in G$, the parties compute $\langle \lambda_{uv} \rangle = \langle \lambda_u \rangle \cdot \langle \lambda_v \rangle$ by calling **Multiply** on $\mathcal{F}_{\text{BitMPC}}$. Each P_i also receives an additive share λ_{uv}^i.
   (b) For every $j \in [n]$, the parties run the subprotocol $\Pi_{\text{Bit}\times\text{String}}^{3|G|}$, where P_j inputs R^j and everyone inputs the identifiers and additive shares of the $3|G|$ bits:
   $$\left(\langle \lambda_u \rangle, \langle \lambda_v \rangle, \langle \lambda_{uv} \rangle + \langle \lambda_w \rangle\right)_{(u,v,w)} .$$
   where the (u, v, w) indices are taken over the input/output wires of each AND gate $g \in G$.
   (c) For each AND gate g, party P_i obtains from $\Pi_{\text{Bit}\times\text{String}}$ an additive share of the $3n$ values:
   $$\lambda_u \cdot R^j, \quad \lambda_v \cdot R^j, \quad \lambda_{uvw} \cdot R^j, \qquad \text{for } j \in [n]$$
   where $\lambda_{uvw} := \lambda_{uv} + \lambda_w$. Each P_i then uses these to compute a share of
   $$\rho_{j,a,b} = \lambda_{uvw} \cdot R^j \oplus a \cdot \lambda_v \cdot R^j \oplus b \cdot \lambda_u \cdot R^j \oplus a \cdot b \cdot R^j$$
4. **Garble gates:** For each AND gate $g \in G$, each $j \in [n]$, and the four combinations of $a, b \in \{0,1\}^2$, the parties compute shares of the j-th entry of the garbled gate $\tilde{g}_{a,b}$ as follows:
   - P_j sets $(\tilde{g}_{a,b}^j)^j = \rho_{j,a,b}^j \oplus F_{k_{u,a}^j, k_{v,b}^j}(g \| j) \oplus k_{w,0}^j$.
   - For every $i \neq j$, P_i sets $(\tilde{g}_{a,b}^j)^i = \rho_{j,a,b}^i \oplus F_{k_{u,a}^i, k_{v,b}^i}(g \| j)$.
5. **Reveal masks for output wires:** For every circuit-output wire $w \in W_{\mathsf{out}}$, the parties call **Open** on $\mathcal{F}_{\text{BitMPC}}$ to reveal λ_w to all the parties.

Figure 6: The preprocessing protocol that realizes $\mathcal{F}_{\text{Prepocessing}}$ in the $\{\mathcal{F}_{\text{COT}}, \mathcal{F}_{\text{BitMPC}}, \mathcal{F}_{\text{Rand}}\, \mathcal{F}_{\text{Commit}}\}$-hybrid model.

The Preprocessing Protocol $\Pi_{\text{Preprocessing}}$ – Open Garbling Stage

Let $\mathsf{PRG} : \{0,1\}^\kappa \to \{0,1\}^{4n\kappa|G|}$ be a secure PRG.

Open Garbling: Let $\tilde{C}^i = ((\tilde{g}^j_{a,b})^i)_{j,a,b,g} \in \{0,1\}^{4n\kappa|G|}$ be P_i's share of the whole garbled circuit.

1. Each party P_i samples random seeds $s^i_j \leftarrow \{0,1\}^\kappa$, $j \neq i$. P_i sends s^i_j to P_j over a private channel.
2. P_i computes the shares $S^i_i = \bigoplus_{i \neq j} \mathsf{PRG}(s^i_j)$, and $S^j_i = \mathsf{PRG}(s^j_i)$, for $j \neq i$.[a]
3. Each P_i, for $i = 2, \ldots, n$, sends $\tilde{C}^i \oplus \bigoplus_{j=1}^n S^j_i$ to P_1.
4. P_1 reconstructs the garbled circuit, $\tilde{C}$, and broadcasts this.

[a] Steps 1 to 2 are independent of $\tilde{C}^i$, so can be merged with previous rounds in the **Garbling** stage.

Figure 7: Open Garbling stage of the preprocessing protocol.

Each party's share of the garbled circuit is then obtained by adding the appropriate PRF values and keys to the shares of each $\rho_{j,a,b}$. To conclude the **Garbling** stage, the parties reveal the masks for all output wires using $\mathcal{F}_{\text{BitMPC}}$, so that the outputs can be obtained in the online phase.

Before opening the garbled circuit, the parties must rerandomize their shares by distributing a fresh, random secret sharing of each share to the other parties, via private channels. This is needed so that the shares do not leak any information on the PRF values, so we can prove security. This may seem unnecessary, since the inclusion of the PRF values in the shares should randomize them sufficiently. However, we cannot prove this intuition, as the same PRF values are used to compute the garbled circuit that is output by the protocol, so they cannot also be used as a one-time pad.[1] In steps 1 to 2 of Figure 7, we show how to perform this extra rerandomization step with $O(n^2 \cdot \kappa)$ communication.

Finally, to reconstruct the garbled circuit, the parties sum up and broadcast the rerandomized shares and add them together to get $\tilde{g}^j_{a,b}$.

3.3 Bit/String Multiplications

Our method for this is in the subprotocol $\Pi_{\text{Bit}\times\text{String}}$ (Figure 8). It takes as input a global difference R^j from each P_j, and the additively shared wire masks that have been stored in $\mathcal{F}_{\text{BitMPC}}$. The protocol proceeds in two stages: first the **Multiply** step creates the shared products using correlated OT, and then the **Consistency Check** verifies that the correct inputs were used to create the products.

Recall that the task is for the parties to obtain an additive sharing of the products, for each $j \in [n]$ and $(a,b) \in \{0,1\}^2$:

$$R^j \cdot ((\lambda_u \oplus a) \cdot (\lambda_v \oplus b) \oplus \lambda_w) \tag{3}$$

where the string R^j is known only to P_j, and fixed for every gate. Denote by x one of the additively shared $\lambda_{(\cdot)}$ bits used in a single bit/string product and stored by $\mathcal{F}_{\text{BitMPC}}$. We obtain shares of $x \cdot R^j$ using actively secure correlated OT (cf. Figure 3), as follows:

1. For each $i \neq j$, parties P_i and P_j run a correlated OT, with choice bit x^i and correlation R^j. P_i obtains $T_{i,j}$ and P_j obtains $Q_{i,j}$ such that:

[1] Furthermore, the environment sees all of the PRF keys of the honest parties, since these are outputs of the protocol, which seems to rule out any kind of computational reduction in the security proof.

$$T_{i,j} = Q_{i,j} + x^i \cdot R^j.$$

2. Each P_i, for $i \neq j$, defines the share $Z^i = T_{i,j}$, and P_j defines $Z^j = \sum_{i \neq j} Q_{i,j} + x^j \cdot R^j$. Now we have:

$$\sum_{i=1}^{n} Z^i = \sum_{i \neq j} T_{i,j} + \sum_{i \neq j} Q_{i,j} + x^j \cdot R^j \quad = \sum_{i \neq j} (T_{i,j} + Q_{i,j}) + x^j \cdot R^j = x \cdot R^j$$

as required.

The above method is performed $3|G|$ times and for each P_j, to produce the shared bit/string products $x \cdot R^j$, for $x \in \{\lambda_u, \lambda_v, \lambda_{uv}\}$.

3.4 Consistency Check

We now show how the parties verify that the correct shares of x and correlations R^j were used in the correlated OTs, and analyse the security of this check. The parties first sample s extra random shared bits using $\mathcal{F}_{\text{BitMPC}}$, where s is a statistical security parameter, then create $m + s$ bit/string products, where m is the number of products needed, and open random linear combinations (over $\mathbb{F}_2$) of all the bits that are stored in $\mathcal{F}_{\text{BitMPC}}$. The parties then apply *the same* linear combination to the strings output from $\mathcal{F}_{\text{COT}}$, and use this to check correctness of the opened results.

In more detail, the parties first sample a random ε-almost 1-universal hash function $\mathbf{H} \leftarrow \mathbb{F}_2^{m \times s}$, and then open

$$\boldsymbol{c}_x = \mathbf{H} \cdot \boldsymbol{x} + \hat{\boldsymbol{x}}$$

using $\mathcal{F}_{\text{BitMPC}}$. Here, $\boldsymbol{x}$ is the vector of all m wire masks to be multiplied, whilst $\hat{\boldsymbol{x}} \in \mathbb{F}_2^s$ are the additional, random masking bits, used as a one-time pad to ensure that $\boldsymbol{c}_x$ does not leak information on $\boldsymbol{x}$.

To verify that a single shared matrix $\mathbf{Z}_j$ is equal to $\boldsymbol{x} \otimes R^j$ (as in Figure 8), each party P_i, for $i \neq j$, then commits to $\mathbf{H} \cdot \mathbf{Z}_j^i$, whilst P_j commits to $\mathbf{H} \cdot \mathbf{Z}_j^j + \boldsymbol{c}_x \otimes R^j$. The parties then open all commitments and check that these sum to zero, which should happen if the products were correct.

The intuition behind the check is that any errors present in the original bit/string products will remain when multiplied by $\mathbf{H}$, except with probability ε, by the almost-1-universal property (Definition 2.3). Furthermore, it turns out that cancelling out any non-zero errors in the check requires either guessing an honest party's global difference R^j, or guessing the secret masking bits $\hat{\boldsymbol{x}}$.

We formalize this, by first considering the exact deviations that are possible by a corrupt P_j in $\Pi_{\text{Bit} \times \text{String}}$. These are:

1. Provide inconsistent inputs R^j when acting as sender in the **Initialize** command of the $\mathcal{F}_{\text{COT}}$ instances with two different honest parties.

2. Input an incorrect share x^j when acting as receiver in the **Extend** command of $\mathcal{F}_{\text{COT}}$.

Note that in the first case, an inconsistency arises when P_j uses two different strings in different $\mathcal{F}_{\text{COT}}$ instances, whereas in the second case, an incorrect share x^j means that x^j is different to the share used previously by P_j in the main protocol. In both of these cases, we are only concerned when the other party in the $\mathcal{F}_{\text{COT}}$ execution is honest, as if both parties are corrupt then $\mathcal{F}_{\text{COT}}$ does not need to be simulated in the security proof.

Bit/string multiplication subprotocol – $\Pi^m_{\mathrm{Bit}\times\mathrm{String}}$

The protocol uses the functionalities $\mathcal{F}_{\mathrm{COT}}$ and $\mathcal{F}_{\mathrm{BitMPC}}$.
Inputs: Each party P_j inputs a string $R^j \in \mathbb{F}_2^\kappa$. As common input, all parties have the identifiers of m secret bits $\langle \boldsymbol{x} \rangle = (\langle x_1 \rangle, \ldots, \langle x_m \rangle)$ that are stored in $\mathcal{F}_{\mathrm{BitMPC}}$, for which each P_j also inputs additive shares $\boldsymbol{x}^j$.

I: Init:

1. Each P_i calls **Input** on $\mathcal{F}_{\mathrm{BitMPC}}$ with randomly sampled bits $\hat{\boldsymbol{x}}^i = (\hat{x}^i_1, \ldots, \hat{x}^i_s)$. Compute the secret mask $\langle \hat{\boldsymbol{x}} \rangle = \sum_{i \in [n]} \langle \hat{\boldsymbol{x}}^i \rangle$ using $\mathcal{F}_{\mathrm{BitMPC}}$.
2. Every ordered pair of parties (P_i, P_j) calls **Initialize** on $\mathcal{F}_{\mathrm{COT}}$, where P_j, the sender, inputs R^j.

II: Multiply: For each $j \in [n]$, the parties do as follows:

1. For every $i \neq j$, parties P_i and P_j call **Extend** on the $\mathcal{F}_{\mathrm{COT}}$ instance where P_j is sender, and P_i inputs the choice bits $(\boldsymbol{x}^i \| \hat{\boldsymbol{x}}^i)$.
   For each OT between (P_i, P_j), P_j receives $q \in \{0,1\}^\kappa$ and P_i receives $t \in \{0,1\}^\kappa$. P_i stores their $m + s$ strings from this instance into the rows of a matrix $\mathbf{T}_{i,j}$, and P_j stores the corresponding outputs in $\mathbf{Q}_{i,j}$. These satisfy:
   $$\mathbf{T}_{i,j} = \mathbf{Q}_{i,j} + (\boldsymbol{x}^i \| \hat{\boldsymbol{x}}^i) \otimes R^j \in \mathbb{F}_2^{(m+s)\times\kappa}.$$
2. Each P_i, for $i \neq j$, defines the matrix $\mathbf{Z}^i_j = \mathbf{T}_{i,j}$, and P_j defines $\mathbf{Z}^j_j = \sum_{i \neq j} \mathbf{Q}_{i,j} + (\boldsymbol{x}^j \| \hat{\boldsymbol{x}}^j) \otimes R^j$.

Now, it should hold that $\sum_{i=1}^n \mathbf{Z}^i_j = (\boldsymbol{x} \| \hat{\boldsymbol{x}}) \otimes R^j$, for each $j \in [n]$.

III: Consistency Check: The parties check correctness of the above as follows:

1. Each P_i removes the last s rows from $\mathbf{Z}^i_j$ (for $j \in [n]$) and places these 'dummy' masking values in a matrix $\hat{\mathbf{Z}}^i_j \in \mathbb{F}_2^{s\times\kappa}$.
2. The parties call $\mathcal{F}_{\mathrm{Rand}}$ (Figure 2) to sample a seed for a uniformly random, ε-almost 1-universal linear hash function, $\mathbf{H} \in \mathbb{F}_2^{s\times m}$.
3. All parties compute the vector:
   $$\langle \boldsymbol{c}_x \rangle = \mathbf{H} \cdot \langle \boldsymbol{x} \rangle \ + \langle \hat{\boldsymbol{x}} \rangle \in \mathbb{F}_2^s$$
   and open $\boldsymbol{c}_x$ using the **Open** command of $\mathcal{F}_{\mathrm{BitMPC}}$. If $\mathcal{F}_{\mathrm{BitMPC}}$ aborts, the parties abort.
4. Each party P_i calls **Commit** on $\mathcal{F}_{\mathrm{Commit}}$ (Figure 1) with input the n matrices:
   $$\mathbf{C}^i_j = \mathbf{H} \cdot \mathbf{Z}^i_j + \hat{\mathbf{Z}}^i_j, \quad \text{for } j \neq i, \text{ and } \quad \mathbf{C}^i_i = \mathbf{H} \cdot \mathbf{Z}^i_i + \hat{\mathbf{Z}}^i_i + \boldsymbol{c}_x \otimes R^i.$$
5. All parties open their commitments and check that, for each $j \in [n]$:
   $$\sum_{i=1}^n \mathbf{C}^i_j = 0.$$
   If any check fails, the parties abort.
6. Each party P_i outputs the mn rows in all the matrices $\mathbf{Z}^i_1, \ldots, \mathbf{Z}^i_n$.

Figure 8: Subprotocol for m bit/string multiplications with consistency check.

We model these two attacks by defining $R^{j,i}$ and $x^{j,i}$ to be the *actual* inputs used by a corrupt P_j in the above two cases, and then define the errors (for $j \in I$ and $i \notin I$):

$$\Delta^{j,i} = R^{j,i} + R^j$$
$$\delta_\ell^{j,i} = x_\ell^{j,i} + x_\ell^j, \quad \ell \in [m].$$

where R^j is defined to be the R^{j,i_0} with an arbitrary honest party P_{i_0}, and x^j is defined based on the shares used by P_j in the main protocol (either as input to or received from $\mathcal{F}_{\text{BitMPC}}$).

Note that $\Delta^{j,i}$ is fixed in the initialization of $\mathcal{F}_{\text{COT}}$, whilst $\delta_\ell^{j,i}$ may be different for every OT. Whenever P_i and P_j are both corrupt, or both honest, for convenience we define $\Delta^{j,i} = 0$ and $\delta^{j,i} = 0$.

This means that the outputs of $\mathcal{F}_{\text{COT}}$ with (P_i, P_j) then satisfy (omitting ℓ subscripts):

$$t_{i,j} = q_{i,j} + x^i \cdot R^j + \delta^{i,j} \cdot R^j + \Delta^{j,i} \cdot x^i$$

where $\delta^{i,j} \neq 0$ if P_i cheated, and $\Delta^{j,i} \neq 0$ if P_j cheated.

Now, as in step 1 of the first stage of $\Pi_{\text{Bit}\times\text{String}}$, we can put the $\mathcal{F}_{\text{COT}}$ outputs for each party into the rows of a matrix, and express the above as:

$$\mathbf{T}_{i,j} = \mathbf{Q}_{i,j} + \boldsymbol{x}^i \otimes R^j + \boldsymbol{\delta}^{i,j} \otimes R^j + \boldsymbol{x}^i \otimes \Delta^{j,i}$$

where $\boldsymbol{\delta}^{j,i} = (\delta_1^{j,i}, \ldots, \delta_m^{j,i})$, and the tensor product notation is defined in Section 2.

Accounting for these errors in the outputs of the **Multiply** step in $\Pi_{\text{Bit}\times\text{String}}$, we get:

$$\mathbf{Z}_j = \sum_{i=1}^{n} \mathbf{Z}_j^i = \boldsymbol{x} \otimes R^j + \underbrace{\sum_{i \in I} \boldsymbol{\delta}^{i,j}}_{=\boldsymbol{\delta}^j} \otimes R^j + \sum_{i \notin I} \boldsymbol{x}^i \otimes \Delta^{j,i}. \tag{4}$$

The following lemma shows if a party cheated, then to pass the check they must either guess all of the shares $\hat{\boldsymbol{x}}^i \in \mathbb{F}_2^s$ for some honest P_i, or guess P_i's global difference R^i (except with negligible probability over the choice of the ε-almost 1-universal hash function, $\mathbf{H}$).

Lemma 3.1 *If the check in $\Pi_{\text{Bit}\times\text{String}}$ passes, then except with probability at most $\varepsilon + 2^{-\kappa}$, all of the errors $\boldsymbol{\delta}^j, \Delta^{i,j}$ are zero.*

Proof: From (4), we have, for each $j \in [n]$:

$$\mathbf{Z}_j = \sum_{i=1}^{n} \mathbf{Z}_j^i = \boldsymbol{x} \otimes R^j + \boldsymbol{\delta}^j \otimes R^j + \sum_{i \notin I} \boldsymbol{x}^i \otimes \Delta^{j,i}.$$

Notice that steps 4–5 of the check in Figure 8 perform n individual checks on the matrices $\mathbf{Z}_1, \ldots, \mathbf{Z}_n$ in parallel. Fix j, and first consider the check for a single matrix $\mathbf{Z}_j$. From here, we omit the subscript j to simplify notation.

Let $(\mathbf{C}^*)^i$, for $i \in I$, be the values committed to by corrupt parties in step 4, and define

$$\tilde{\mathbf{C}}^i = \begin{cases} \mathbf{H} \cdot \mathbf{Z}^i + \hat{\mathbf{Z}}^i, & \text{if } i \neq j \\ \mathbf{H} \cdot \mathbf{Z}^i + \hat{\mathbf{Z}}^i + \boldsymbol{c}_x \otimes R^i, & \text{if } i = j \end{cases}$$

to be the value which a corrupt P_i *should have* committed to.

Denote the *difference* between what a corrupt P_i actually committed to, and what they should have committed to, by:

$$\mathbf{D}^i = (\mathbf{C}^*)^i + \tilde{\mathbf{C}}^i \in \mathbb{F}_2^{s \times \kappa}.$$

Also, define the sum of the differences $\mathbf{D}_I = \sum_{i \in I} \mathbf{D}^i$. To pass the consistency check, it must hold that:

$$\begin{aligned}
0 &= \sum_{i \notin I} \mathbf{C}^i + \sum_{i \in I} \tilde{\mathbf{C}}^i + \mathbf{D}_I \\
\Leftrightarrow \mathbf{D}_I &= \sum_{i \notin I} \mathbf{C}^i + \sum_{i \in I} \tilde{\mathbf{C}}^i \\
&= \mathbf{H}(\sum_{i=1}^{n} \mathbf{Z}^i) + \sum_{i=1}^{n} \hat{\mathbf{Z}}^i + \boldsymbol{c}_x \otimes R^j \\
&= \mathbf{H}(\boldsymbol{x} \otimes R^j + \boldsymbol{\delta}^j \otimes R^j + \sum_{i \notin I} \boldsymbol{x}^i \otimes \Delta^{j,i}) + \hat{\mathbf{Z}} + \mathbf{H}(\boldsymbol{x}) \otimes R^j + \hat{\boldsymbol{x}} \otimes R^j \qquad (5) \\
&= \mathbf{H}(\boldsymbol{\delta}^j \otimes R^j + \sum_{i \notin I} \boldsymbol{x}^i \otimes \Delta^{j,i}) + \hat{\mathbf{Z}} + \hat{\boldsymbol{x}} \otimes R^j \qquad (6)
\end{aligned}$$

where (5) holds because $\boldsymbol{c}_x = \mathbf{H}(\boldsymbol{x}) + \hat{\boldsymbol{x}}$, and (6) due to the linearity of $\mathbf{H}$ and the tensor product.

Now, taking into account the fact that $\hat{\mathbf{Z}}$ is constructed the same way as $\mathbf{Z}$, and considering (4), there exist some adversarially chosen errors $\hat{\boldsymbol{\delta}}^j \in \mathbb{F}_2^s$ such that:

$$\hat{\mathbf{Z}} = \hat{\boldsymbol{x}} \otimes R^j + \hat{\boldsymbol{\delta}}^j \otimes R^j + \sum_{i \notin I} \hat{\boldsymbol{x}}^i \otimes \Delta^{j,i}.$$

Plugging this into equation (6), the check passes if and only if:

$$\begin{aligned}
\mathbf{D}_I &= \mathbf{H}(\boldsymbol{\delta}^j \otimes R^j + \sum_{i \notin I} \boldsymbol{x}^i \otimes \Delta^{j,i}) + \hat{\boldsymbol{\delta}}^j \otimes R^j + \sum_{i \notin I} \hat{\boldsymbol{x}}^i \otimes \Delta^{j,i} \\
&= (\mathbf{H}(\boldsymbol{\delta}^j) + \hat{\boldsymbol{\delta}}^j) \otimes R^j + \sum_{i \notin I} (\mathbf{H}(\boldsymbol{x}^i) + \hat{\boldsymbol{x}}^i) \otimes \Delta^{j,i}. \qquad (7)
\end{aligned}$$

We now show that the probability of this holding is negligible, if any errors are non-zero.

First consider the left-hand summation in (7), supposing that at least one of $\boldsymbol{\delta}^j, \hat{\boldsymbol{\delta}}^j$ is non-zero. Recall that $\boldsymbol{\delta}^j$ and $\hat{\boldsymbol{\delta}}^j$ are fixed by the adversary's inputs to $\mathcal{F}_{\text{COT}}$, so are independent of the random choice of the hash function $\mathbf{H}$. Therefore, by the ε-almost 1-universal property of the family of linear hash functions (Definition 2.3), it holds that

$$\Pr_{\mathbf{H}}[\mathbf{H}(\boldsymbol{\delta}^j) + \hat{\boldsymbol{\delta}}^j = 0] \leq \varepsilon.$$

So except with probability ε, $\mathcal{A}$ will have to guess R^j to construct $\mathbf{D}_I$ to pass the check, since R^j is independent of the right-hand summation. By a union bound, therefore, if at least one of $\boldsymbol{\delta}^j$ or $\hat{\boldsymbol{\delta}}^j$ is non-zero then the check passes with probability at most $\varepsilon + 2^{-\kappa}$.

On the other hand, suppose that $\boldsymbol{\delta}^j = \hat{\boldsymbol{\delta}}^j = 0$, so $\mathcal{A}$ only needs to guess the right-hand summation of (7) to pass the check. If the error $\Delta^{j,i} \neq 0$ for some $i \notin I$ then the adversary must successfully guess

$\mathbf{H}(\boldsymbol{x}^i) + \hat{\boldsymbol{x}}^i$ to be able to pass the check. Since $\hat{\boldsymbol{x}}^i$ is uniform in the view of the adversary, this can only occur with probability 2^{-s}. Also, note that $2^{-s} \leq \varepsilon$, since the hash function $\mathbf{H}$ outputs s bits.

In conclusion, the probability of passing the j-th check when any of the errors $\boldsymbol{\delta}^j, \hat{\boldsymbol{\delta}}^j$ or $\Delta^{j,i}$ are non-zero, is no more than $\varepsilon + 2^{-\kappa}$. Since the adversary must pass all n checks to prevent an abort, this also gives an upper bound for the overall success probability. ■

3.5 Security Proof

We now give some intuition behind the security of the whole protocol. In the proof, the strategy of the simulator is to run an internal copy of the protocol, using dummy, random values for the honest parties' keys and wire mask shares. All communication with the adversary is simulated by computing the correct messages according to the protocol and the dummy honest shares, until the final output stage. In the output stage, we switch to fresh, random honest parties' shares, consistent with the garbled circuit received from $\mathcal{F}_{\text{Prepocessing}}$ and the corrupt parties' shares.

Firstly, by Lemma 3.1, it holds that in the real execution, if the adversary introduced any non-zero errors then the consistency check fails with overwhelming probability. The same is true in the ideal execution; note that the errors are still well-defined in this case because the simulator can compute them by comparing all inputs received to $\mathcal{F}_{\text{COT}}$ with the inputs the adversary should have used, based on its random tape. This implies that the probability of passing the check is the same in both worlds. Also, if the check fails then both executions abort, and it is straightforward to see that the two views are indistinguishable because no outputs are sent to honest parties (hence, also the environment).

It remains to show that the two views are indistinguishable when the consistency check passes, and the environment sees the outputs of all honest parties, as well as the view of the adversary during the protocol. The main point of interest here is the output stage. We observe that, without the final rerandomization step, the honest parties' shares of the garbled circuit would *not be uniform*. Specifically, consider an honest P_i's share, $(\tilde{g}^j_{a,b})^i$, where P_j is corrupt. This is computed by adding some PRF value, v, to the $\mathcal{F}_{\text{COT}}$ outputs where P_i was receiver and P_j was sender (step 2 of $\Pi_{\text{Bit}\times\text{String}}$). Since P_j knows both strings in each OT, there are only two possibilities for P_i's output (depending on the choice bit), so this is not uniform. It might be tempting to argue that v is a random PRF output, so serves as a one-time pad, but this proof attempt fails because v is also used to compute the final garbled circuit. In fact, it seems difficult to rely on any reduction to the PRF, since all the PRF keys are included in the output to the environment.

To avoid this issue, we need the rerandomization step and the additional assumption of secure point-to-point channels. Note that this was originally missing from the protocol of [BLO16]. Rerandomization ensures that the honest shares can be simulated with random values which, together with the corrupt shares, sum up to the correct garbled circuit. To gain on efficiency we implement this step using a PRG. We proceed with the complete proof.

Theorem 3.1 *Protocol $\Pi_{\text{Preprocessing}}$ from Figure 6 UC-securely computes $\mathcal{F}_{\text{Prepocessing}}$ from Figure 5 in the presence of a static, active adversary corrupting up to $n-1$ parties in the $\{\mathcal{F}_{\text{COT}}, \mathcal{F}_{\text{BitMPC}}, \mathcal{F}_{\text{Rand}}, \mathcal{F}_{\text{Commit}}\}$-hybrid model.*

Proof: Let $\mathcal{A}$ denote a PPT adversary corrupting a strict subset $I \subsetneq [n]$ of parties. As part of the proof, we will construct a simulator $\mathcal{S}$ that plays the roles of the honest parties on arbitrary inputs, as well as the roles of functionalities $\{\mathcal{F}_{\text{COT}}, \mathcal{F}_{\text{BitMPC}}, \mathcal{F}_{\text{Rand}}, \mathcal{F}_{\text{Commit}}\}$ and interacts with $\mathcal{A}$. We assume w.l.o.g. that $\mathcal{A}$ is a deterministic adversary, which receives as additional input a random tape that determines its internal coin

tosses. Nevertheless, since $\mathcal{A}$ is active, it may still ignore the random tape, and use its own (possibly biased) random values instead.

The simulator begins by initializing $\mathcal{A}$ with the inputs from the environment, $\mathcal{Z}$, and a uniform string r as its random tape. During the simulation, we will use r to compute values that $\mathcal{A}$ *should* (but might not) use during the protocol. We can now define the rest of $\mathcal{S}$ as follows:

Garbling:

1. The simulator emulates $\mathcal{F}_{\text{COT}}$.`Initialize` as follows:
   - For a honest party $P_i, i \notin I$, $\mathcal{S}$ samples $R^i \in \{0,1\}^\kappa$.
   - For a corrupted party $P_j, j \in I$, $\mathcal{S}$ computes R^j from that party's random tape. Additionally, for each pair of parties involving that party, (P_j, P_i) where $i \notin I$, $\mathcal{S}$ receives by $\mathcal{A}$ values $R^{j,i} \in \{0,1\}^\kappa$ and stores $\Delta^{j,i} = R^{j,i} + R^j$. If $\forall i \notin I, \Delta^{j,i} = d$, then $\mathcal{S}$ modifies its stored values by setting $R^j \leftarrow R^j + d$ and $\Delta^{j,i} \leftarrow 0, \forall i \notin I$.
2. GENERATE WIRE MASKS AND KEYS: Passing through each wire w of the circuit topologically, the simulator $\mathcal{S}$ proceeds as follows.
   - It emulates $\mathcal{F}_{\text{BitMPC}}$ and defines the wire masks:
     - If w is a *circuit-input* wire, P_i is the party whose input is associated with it and $i \in I$, then $\mathcal{S}$ receives λ_w by $\mathcal{A}$. If $i \notin I$, then $\mathcal{S}$ chooses λ_w.
     - If w is the output of an AND gate, $\mathcal{S}$ samples $\lambda_w \leftarrow \mathbb{F}_2$ and receives from the adversary λ_w^i for every $i \in I$. Then, $\mathcal{S}$ samples random λ_w^j, $j \notin I$, such that $\lambda_w = \sum_{\ell \in [n]} \lambda_w^\ell$.
     - If w is the output of a XOR gate with input wires u and v, $\mathcal{S}$, it sets $\lambda_w = \lambda_u + \lambda_v$.
   - It defines the PRF keys:
     - If w is not the output of a XOR gate, $\mathcal{S}$ computes $k_{w,0}^i \in \{0,1\}^\kappa$ for $i \in [n]$. For corrupted parties $i \in I$, $\mathcal{S}$ reads this off P_i's random tape, whereas for honest parties it samples a random key.
     - If w is the output of a XOR gate with input wires u and v, for $i \in [n]$, it sets $k_{w,0}^i = k_{u,0}^i \oplus k_{v,0}^i$ and $k_{w,1}^i = k_{w,0}^i \oplus R^i$ for $i \in [n]$.
   - Finally, it sends to $\mathcal{F}_{\text{Prepocessing}}$ the global difference R^i and the keys $k_{w,0}^i$, for each $i \in I$ and each w that is an output wire of an AND gate, as well as the wire masks λ_w (for each w that is an input wire of a corrupt party).
3. SECURE PRODUCT COMPUTATIONS: The simulator $\mathcal{S}$ emulates $\mathcal{F}_{\text{BitMPC}}$.`Input` receiving $\hat{\boldsymbol{x}}^i = (\hat{x}_1^i, \ldots, \hat{x}_s^i)$ for every $i \in I$, and also samples honest parties' shares $\hat{\boldsymbol{x}}^j$, for $j \notin I$. For each AND gate $g \in G$, it emulates $\mathcal{F}_{\text{BitMPC}}$.`Multiply` by receiving shares $(\lambda_{uv})^i$ from $\mathcal{A}$ for $i \in I$ and setting random $(\lambda_{uv})^j$ for $j \notin I$ such that $\sum_{\ell \in [n]} (\lambda_{uv})^\ell = \lambda_u \cdot \lambda_v$. For the $\Pi_{\text{Bit} \times \text{String}}$ subprotocol:
   - Multiply: $\mathcal{S}$ emulates $\mathcal{F}_{\text{COT}}$.`Extend` between each pair of parties P_i and P_j. If both parties are honest, or if both are corrupted, the simulation is trivial. Hereafter, we focus on the cases where exactly one party of each pair is corrupted:
     - When P_i is a corrupted sender, $\mathcal{S}$ receives a (possibly) different $q_i \in \{0,1\}^\kappa$ from $\mathcal{A}$ for each of the $3|G| + s$ calls.
     - When P_i is a corrupted receiver, $\mathcal{S}$ receives $\left(\lambda_u^i + \delta_{\lambda_u}^{i,j}, \lambda_v^i + \delta_{\lambda_v}^{i,j}, \lambda_{uv}^i + \lambda_w^i + \delta_{\lambda_{uv}+\lambda_w}^{i,j}\right)_{(u,v,w)}$ for each AND gate, plus the values $\hat{x}_1^i + \delta_{\hat{x}_1}^{i,j}, \ldots, \hat{x}_s^i + \delta_{\hat{x}_s}^{i,j}$. For each of the $3|G| + s$ previous inputs, it also receives a (possibly) different $t_i \in \{0,1\}^\kappa$ from $\mathcal{A}$.

Note that the errors $\delta^{i,j}$ and the t_i, q_i values received from $\mathcal{A}$ are stored by $\mathcal{S}$, whilst the λ values and shares were fixed in the previous stage.

- Consistency Check:
  2. $\mathcal{S}$ emulates $\mathcal{F}_{\text{Rand}}$ and sends a seed for a uniformly random ε-almost 1-universal linear hash function $\mathbf{H} \in \mathbb{F}_2^{s \times 3|G|}$ to $\mathcal{A}$.
  3. $\mathcal{S}$ emulates $\mathcal{F}_{\text{BitMPC}}.\texttt{Open}$ and sends $\boldsymbol{c}_x = \mathbf{H} \cdot \boldsymbol{x} + \hat{\boldsymbol{x}} \in \mathbb{F}_2^s$ to $\mathcal{A}$, where $\boldsymbol{x}$ is the vector of $3|G|$ values $(\lambda_u, \lambda_v, \lambda_{uvw})_{(u,v,w)}$ from the previous stage, and $\hat{\boldsymbol{x}} = \sum_{i \in [n]} \hat{\boldsymbol{x}}^i$, received just before the **Multiply** step. If $\mathcal{A}$ does not send back `OK` to $\mathcal{S}$, then $\mathcal{S}$ sends $\perp$ to $\mathcal{F}_{\text{Prepocessing}}$ and aborts.

4-5. Emulating $\mathcal{F}_{\text{Commit}}$, $\mathcal{S}$ receives $\mathbf{C}_\ell^i$ from $\mathcal{A}$ for all $\ell \in [n]$ and $i \in I$. It then computes $\mathbf{C}_\ell^j$ for $j \notin I$, as each honest party would, and completes the emulation of $\mathcal{F}_{\text{Commit}}$ by sending these to $\mathcal{A}$. If $\sum_{i \in I} \mathbf{C}_\ell^i + \sum_{j \notin I} \mathbf{C}_\ell^j \neq 0$, for any $\ell \in [n]$, $\mathcal{S}$ sends $\perp$ to $\mathcal{F}_{\text{Prepocessing}}$ and terminates.

4. GARBLE GATES: For every AND gate $g \in G$, the simulator $\mathcal{S}$ computes and stores corrupt parties' shares of the garbled circuit, $\tilde{C}^i$, for $i \in I$, as the adversary should do according to the protocol. Note that $\mathcal{S}$ has all the necessary values to do so from the messages it previously received and the knowledge of $\mathcal{A}$'s random tape. Namely, the simulator knows the PRF keys, the global differences, and the t_i and q_i values received in the $\mathcal{F}_{\text{COT}}$ calls.
5. REVEAL MASKS FOR OUTPUT WIRES: $\mathcal{S}$ emulates $\mathcal{F}_{\text{BitMPC}}.\texttt{Open}$ and for every circuit-output wire w, it calls $\mathcal{F}_{\text{Prepocessing}}$ to get the wire mask λ_w and forward it to $\mathcal{A}$. If $\mathcal{A}$ does not send back `OK` to $\mathcal{S}$, then $\mathcal{S}$ sends $\perp$ to $\mathcal{F}_{\text{Prepocessing}}$ and terminates.

Open Garbling:

6. For each $i \in I$, $\mathcal{S}$ samples random shares $\{S_j^i\}_{j \notin I}$ and sends these to $\mathcal{A}$.
7. $\mathcal{S}$ then calls $\mathcal{F}_{\text{Prepocessing}}$ to receive the garbled circuit $\tilde{C}$. Using the corrupted parties' shares $\tilde{C}^i$, $i \in I$, received previously, $\mathcal{S}$ generates random honest parties' shares $\tilde{C}^j$, $j \notin I$, subject to the constraint that $\sum_{\ell \in [n]} \tilde{C}^\ell = \tilde{C}$. Once this is done, $\mathcal{S}$ forwards the honest shares of the garbled circuit to $\mathcal{A}$. If $\mathcal{A}$ does not respond with `OK` then $\mathcal{S}$ sends $\perp$ to $\mathcal{F}_{\text{Prepocessing}}$ and terminates. Otherwise, it receives from the adversary `OK` and shares $\hat{C}^i$ for $i \in I$. Finally, $\mathcal{S}$ computes the error $E = \sum_{i \in I} (\tilde{C}^i + \hat{C}^i)$ and sends this to $\mathcal{F}_{\text{Prepocessing}}$.

INDISTINGUISHABILITY: We will first show that, during the **Garbling** phase, the environment $\mathcal{Z}$ cannot distinguish between an interaction with $\mathcal{S}$ and $\mathcal{F}_{\text{Prepocessing}}$ and an interaction with the real adversary $\mathcal{A}$ and $\Pi_{\text{Preprocessing}}$. We then argue that the garbled circuit, and the honest parties' shares of it, are also identically distributed in both worlds.

Garbling phase indistinguishability: Let's look at the **Garbling** command. In both worlds and for every AND gate, the honest parties's shares for the masks λ_w, and for the products λ_{uv} are uniformly random additive shares, whereas the corrupted parties shares' are chosen by $\mathcal{A}$. Every other step up to the execution of the $\Pi_{\text{Bit} \times \text{String}}$ subprotocol provides no output to the parties, and hence $\mathcal{Z}$ has exactly the same view in both worlds up to that point.

In the **Multiply** step of $\Pi_{\text{Bit} \times \text{String}}$, $\mathcal{S}$ only receives values from $\mathcal{A}$, so no further information is added to his view here. Note that since the corrupted parties' inputs to $\mathcal{F}_{\text{COT}}$ are received by $\mathcal{S}$, all the errors $\Delta^{i,j}, \boldsymbol{\delta}^j = \sum_{i \in I} \boldsymbol{\delta}^{i,j}$ are well-defined in the simulation.

Next, consider the **Consistency Check** step. If any of the errors are non-zero, then from Lemma 3.1, we know that in both worlds the check fails with overwhelming probability. In this case, no outputs are sent to the honest parties, and (recalling that there are no inputs from honest parties) indistinguishability is trivial since the simulator just behaved as honest parties would until this point.

We now assume that all of the errors $\Delta^{i,j}, \boldsymbol{\delta}^j = \sum_{i \in I} \boldsymbol{\delta}^{i,j}$ are zero, and so the check passes. The values $\mathbf{H}$ and $\boldsymbol{c}_x = \mathbf{H}\boldsymbol{x} + \hat{\boldsymbol{x}}$ seen by $\mathcal{A}$ are uniformly random in both worlds, since the masking values $\hat{\boldsymbol{x}}$ are uniformly random and never seen by the environment. The distribution of the committed and opened values, $\mathbf{C}^i_j$, is more subtle, however. First, consider the case when there is exactly one honest party. In this case, in both worlds the values $\mathbf{C}^j_i$, for honest P_i, are a deterministic function of the adversary's behaviour, since they should satisfy $\sum_{i=1}^{n} \mathbf{C}^i_j = 0$. Therefore, these values are identically distributed in both worlds.

Now, suppose there is more than one honest party. The values $\mathbf{C}^i_i$ are computed based on the $\mathbf{Z}^i_i$ values, which are the sum of outputs from $\mathcal{F}_{\mathrm{COT}}$ with every other party. This means for every honest P_i, $\mathbf{C}^i_i$ is uniformly random, since it includes an $\mathcal{F}_{\mathrm{COT}}$ output with one other honest party. On the other hand, the values $\mathbf{C}^i_j$, where $j \in I$ and $i \notin I$, only come from a single $\mathcal{F}_{\mathrm{COT}}$ instance between P_i and P_j, and P_i's output from $\mathcal{F}_{\mathrm{COT}}$ in this case is not random in the view of $\mathcal{A}$. It actually should satisfy:

$$\mathbf{C}^i_j = \mathbf{H} \cdot \mathbf{Z}^i_j + \hat{\mathbf{Z}}^i_j = \mathbf{H} \cdot \mathbf{Q}_{i,j} + \hat{\mathbf{Q}}_{i,j} + (\mathbf{H} \cdot \boldsymbol{x}^i + \hat{\boldsymbol{x}}^i) \otimes R^j$$

where $\mathbf{Q}_{i,j}, \hat{\mathbf{Q}}_{i,j}$ are the $\mathcal{F}_{\mathrm{COT}}$ outputs of corrupt P_j, and R^j is also known to P_j. This means for each row of $\mathbf{C}^i_j$, in the view of $\mathcal{A}$ there are only two possibilities, depending on one bit from $(\mathbf{H} \cdot \boldsymbol{x}^i + \hat{\boldsymbol{x}}^i)$. Since the shares $\hat{\boldsymbol{x}}^i$ are uniformly random and never seen by the environment, $\mathbf{H} \cdot \boldsymbol{x}^i + \hat{\boldsymbol{x}}^i$ is also uniformly random in both worlds, subject to the constraint that these sum to $\boldsymbol{c}_x$ (which was opened previously). We conclude that the $\mathbf{C}^j_i$ values are identically distributed.

After $\Pi_{\mathrm{Bit} \times \mathrm{String}}$, $\mathcal{Z}$ remains unable to distinguish in the garbling phase. First, the **Garble Gates** step of $\Pi_{\mathrm{Preprocessing}}$ requires no communication. Finally, regarding **Reveal masks for output wires**, the revealed wire masks are random bits in both worlds.

Open Garbling phase indistinguishability: The seeds sent in the first step of the **Open Garbling** stage are uniformly random in both executions. We claim that the rerandomized shares of the garbled circuit in the real execution are computationally indistinguishable from the simulated random shares. This holds because, (1) The simulated shares seen by the adversary are uniformly random, subject to the constraint that all shares sum up to the same garbled circuit, and (2) In the real world, every pair of honest parties masks their shares with outputs from the PRG $\mathcal{G}$, using a unique seed that is not seen by the environment. By a standard hybrid argument, we can therefore reduce indistinguishability of the shares to security of the PRG, by successively replacing each PRG output sent between two honest parties with a random string. ■

4 More Efficient Garbling with Multi-Party TinyOT

We now describe a less general, but concretely more efficient, variant of the protocol in the previous section. We replace the generic $\mathcal{F}_{\mathrm{BitMPC}}$ functionality with a more specialized one based on 'TinyOT'-style information-theoretic MACs. This is asymptotically worse, but more practical, than using [IPS08] for $\mathcal{F}_{\mathrm{BitMPC}}$. It also allows us to completely remove the bit/string multiplications and consistency checks in $\Pi_{\mathrm{Bit} \times \mathrm{String}}$, since we show that these can be obtained directly from the TinyOT MACs. This means the only cost in the protocol, apart from opening and evaluating the garbled circuit, is the single bit multiplication per AND gate in the underlying TinyOT-based protocol.

In Appendix A we present a complete description of a suitable TinyOT-based protocol. This is done by combining the multiplication triple generation protocol (over $\mathbb{F}_2$) from [FKOS15] with a consistency check to enforce correct shared random bits, which is similar to the more general check from the previous section.

4.1 Secret-Shared MAC Representation

For $x \in \{0,1\}$ held by P_i, define the following two-party MAC representation, as used in 2-party TinyOT [NNOB12]:

$$[x]_{i,j} = (x, M_j^i, K_i^j), \qquad M_j^i = K_i^j + x \cdot R^j$$

where P_i holds x and a MAC M_j^i, and P_j holds a local MAC key K_i^j, as well as the fixed, global MAC key R^j.

Similarly, we define the n-party representation of an additively shared value $x = x^1 + \cdots + x^n$:

$$[x] = (x^i, \{M_j^i, K_j^i\}_{j \neq i})_{i \in [n]}, \quad M_j^i = K_i^j + x^i \cdot R^j$$

where each party P_i holds the $n-1$ MACs M_j^i on x^i, as well as the keys K_j^i on each x^j, for $j \neq i$, and a global key R^i. Note that this is equivalent to every pair (P_i, P_j) holding a representation $[x^i]_{i,j}$.

The key observation for this section, is that a sharing $[x]$ can be used to directly compute shares of all the products $x \cdot R^j$, as in the following claim.

Claim 4.1 *Given a representation $[x]$, the parties can locally compute additive shares of $x \cdot R^j$, for each $j \in [n]$.*

Proof: Write $[x] = (x^i, \{M_j^i, K_j^i\}_{j \neq i})_{i \in [n]}$. Each party P_i defines the n shares:

$$Z_i^i = x^i \cdot R^i + \sum_{j \neq i} K_j^i \quad \text{and} \quad Z_j^i = M_j^i, \quad \text{for each } j \neq i$$

We then have, for each $j \in [n]$:

$$\sum_{i=1}^{n} Z_j^i = Z_j^j + \sum_{i \neq j} Z_j^i = (x^j \cdot R^j + \sum_{i \neq j} K_i^j) + \sum_{i \neq j} M_j^i = x^j \cdot R^j + \sum_{i \neq j} (M_j^i + K_i^j) = x^j \cdot R^j + \sum_{i \neq j} (x^i \cdot R^j) = x \cdot R^j.$$

■

Note that the additive shares of $x \cdot R^j$ are not authenticated in any way. Next, we define addition of two shared values $[x], [y]$, to be straightforward addition of the components. Finally, we define addition of $[x]$ with a public constant $c \in \mathbb{F}_2$ by:

- P_1 stores: $(x^1 + c, \{M_j^1, K_j^1\}_{j \neq 1})$
- P_i stores: $(x^i, (M_1^i, K_1^i + c \cdot R^i), \{M_j^i, K_j^i\}_{j \in [n] \setminus \{1,i\}})), \quad$ for $i \neq 1$

This results in a correct sharing of $[x + c]$.

We can create a sharing of the product of two shared values using a random multiplication triple $([x], [y], [z])$ such that $z = x \cdot y$ with Beaver's technique [Bea92], shown in Figure 18.

Functionality $\mathcal{F}_{\text{n-TinyOT}}$

Initialize: On receiving $(\texttt{init})$ from all parties, the functionality receives $R^i \in \{0,1\}^\kappa$, for $i \in I$, from the adversary, and then samples $R^i \leftarrow \{0,1\}^\kappa$, for $i \notin I$, and sends R^i to party P_i.

Prep: On receiving $(\texttt{Prep}, m, M)$ from all parties, generate m random bits as follows:

1. Receive corrupted parties' shares $b_\ell^i \in \mathbb{F}_2$ from $\mathcal{A}$, for $i \in I$.
2. Sample honest parties' shares, $b_\ell^i \leftarrow \mathbb{F}_2$, for $i \notin I$ and $\ell \in [m]$.
3. Run n-$\mathsf{Bracket}(b_\ell^1, \ldots, b_\ell^n)$, for every $\ell \in [m]$, so each party obtains the shares b_ℓ^i, as well as $n-1$ MACs on b_ℓ^i and a key on each b_ℓ^j, for $j \neq i$.

And M multiplication triples as follows:

1. Sample $a_\ell, b_\ell \leftarrow \mathbb{F}_2$ and compute $c_\ell = a_\ell \cdot b_\ell$, for $\ell \in [M]$.
2. For each $x \in \{a_\ell, b_\ell, c_\ell\}_{\ell \in [M]}$, authenticate x as follows:
   (a) Receive corrupted parties' shares $x^i \in \mathbb{F}_2$, for $i \in I$, from $\mathcal{A}$.
   (b) Sample honest parties' shares $x^i \leftarrow \mathbb{F}_2$, for $i \notin I$ subject to $\sum_{i=1}^n x^i = x$.
   (c) Run n-$\mathsf{Bracket}(x^1, \ldots, x^n)$, so the parties obtain $[x]$.

Key queries: On receiving (i, R') from $\mathcal{A}$, where $i \in [n]$, output 1 to $\mathcal{A}$ if $R^i = R'$. Otherwise, output 0 to $\mathcal{A}$.

Figure 9: Functionality for secure multi-party computation based on TinyOT.

4.2 MAC-Based MPC Functionality

The functionality $\mathcal{F}_{\text{n-TinyOT}}$, which we use in place of $\mathcal{F}_{\text{BitMPC}}$ for the optimized preprocessing, is shown in Figure 9. It produces authenticated sharings of random bits and multiplication triples. For both of these, $\mathcal{F}_{\text{n-TinyOT}}$ first receives corrupted parties' shares, MAC values and keys from the adversary, and then randomly samples consistent sharings and MACs for the honest parties.

Another important aspect of the functionality is the **Key Queries** command, which allows the adversary to try to guess the MAC key R^i of any party, and will be informed if the guess is correct. This is needed to allow the security proof to go through; we explain this in more detail in Appendix A. In that section we also present a complete description of a variant on the multi-party TinyOT protocol, which can be used to implement this functionality.

4.3 Garbling with $\mathcal{F}_{\text{n-TinyOT}}$

Following from the observation in Claim 4.1, if each party P_j chooses the global difference string in $\Pi_{\text{Preprocessing}}$ to be the same R^j as in the MAC representation, then given $[\lambda]$, additive shares of the products $\lambda \cdot R^j$ can be obtained at no extra cost. Moreover, the shares are guaranteed to be correct, and the honest party's shares will be random (subject to the constraint that they sum to the correct value), since they come directly from the $\mathcal{F}_{\text{n-TinyOT}}$ functionality. This means there is no need to perform the consistency check, which greatly simplifies the protocol.

The rest of the protocol is mostly the same as $\Pi_{\text{Preprocessing}}$ in Figure 6, using $\mathcal{F}_{\text{n-TinyOT}}$ with $[\cdot]$-sharings instead of $\mathcal{F}_{\text{BitMPC}}$ with $\langle\cdot\rangle$-sharings. One other small difference is that because $\mathcal{F}_{\text{n-TinyOT}}$ does not have a private input command, we instead sample $[\lambda_w]$ shares for input wires using random bits, and later use a private output protocol to open the relevant input wire masks to P_i. This change is not strictly

Macro n-Bracket

This subroutine of $\mathcal{F}_{\text{n-TinyOT}}$ uses the global MAC keys $R^1, \ldots, R^n$ stored by the functionality.

On input $(x^1, \ldots, x^n)$, authenticate the share $x^i \in \{0,1\}$, for each $i \in [n]$, as follows:

If P_i is corrupt: receive a MAC $M_j^i \in \mathbb{F}_2^\kappa$ from $\mathcal{A}$ and compute the key $K_i^j = M_j^i + x^i \cdot R^j$, for each $j \neq i$.

Otherwise:

1. Sample honest parties' keys $K_i^j \leftarrow \mathbb{F}_2^\kappa$, for $j \in [n] \setminus (I \cup \{i\})$.
2. Receive keys $K_i^j \in \mathbb{F}_2^\kappa$, for each $j \in I$, from $\mathcal{A}$.
3. Compute the MACs $M_j^i = K_i^j + x^i \cdot R^j$, for $j \in I$.

Finally, output $(x^i, \{M_j^i, K_j^i\}_{j \neq i})$ to party P_i, for $i \in [n]$.

Figure 10: Macro used by $\mathcal{F}_{\text{n-TinyOT}}$ to authenticate bits.

necessary, but simplifies the protocol for implementing $\mathcal{F}_{\text{n-TinyOT}}$ — if $\mathcal{F}_{\text{n-TinyOT}}$ also had an **Input** command for sharing private inputs based on n-Bracket, it would be much more complex to implement with the correct distribution of shares and MACs.

In more detail, the **Garbling** phase proceeds as follows.

1. Each party obtains a random global difference R^i by calling the **Initialize** command of $\mathcal{F}_{\text{n-TinyOT}}$.
2. For every wire w which is an input wire, or the output wire of an AND gate, the parties obtain a shared mask $[\lambda_w]$ using the **Bit** command of $\mathcal{F}_{\text{n-TinyOT}}$.
3. All the wire keys $k_{w,0}^i, k_{w,1}^i = k_{w,0}^i \oplus R^i$ are defined by P_i the same way as in $\Pi_{\text{Preprocessing}}$.
4. For XOR gates, the output wire mask is computed as $[\lambda_w] = [\lambda_u] + [\lambda_v]$.
5. For each AND gate, the parties compute $[\lambda_{uv}] = [\lambda_u \cdot \lambda_v]$ using the subprotocol Π_{Mult} in Figure 18.
6. The parties then obtain shares of the garbled circuit as follows:
   - For each AND gate $g \in G$ with wires (u, v, w), the parties use Claim 4.1 with the shared values $[\lambda_u], [\lambda_v], [\lambda_{uv} + \lambda_w]$, to define, for each $j \in [n]$, shares of the bit/string products:
   $$\lambda_u \cdot R^j, \quad \lambda_v \cdot R^j, \quad (\lambda_{uv} + \lambda_w) \cdot R^j$$
   - These are then used to define shares of $\rho_{j,a,b}$ and the garbled circuit, as in the original protocol.
7. For every circuit-output-wire w, the parties run Π_{Open} to reveal λ_w to all the parties.
8. For every *circuit input wire* w corresponding to party P_i's input, the parties run Π_{Open}^i (Figure 15) to open λ_w to P_i.

The only interaction introduced in the new protocol is in the multiply and opening protocols, which were abstracted away by $\mathcal{F}_{\text{BitMPC}}$ in the previous protocol. Simulating and proving security of these techniques is straightforward, due to the correctness and randomness of the multiplication triples and MACs produced

by $\mathcal{F}_{\text{n-TinyOT}}$. One important detail is the **Key Queries** command of the $\mathcal{F}_{\text{n-TinyOT}}$ functionality, which allows the adversary to try to guess an honest party's global MAC key share, R^i, and learn if the guess is correct. To allow the proof to go through, we modify $\mathcal{F}_{\text{Prepocessing}}$ to also have the same **Key Queries** command, so that the simulator can use this to respond to any key queries from the adversary. We denote this modified functionality by $\mathcal{F}_{\text{PrepKQ}}$.

The following theorem can be proven, similarly to the proof of Theorem 3.1 where we modify the preprocessing functionality to support key queries, and adjust the simulation as described above.

Theorem 4.1 *The modified protocol described above UC-securely computes $\mathcal{F}_{\text{PrepKQ}}$ from Figure 5 in the presence of a static, active adversary corrupting up to $n-1$ parties in the $\mathcal{F}_{\text{n-TinyOT}}$-hybrid model.*

Finally, in Section 5.1 we discuss how to extend the proof of the online phase, showing that allowing key queries in the preprocessing functionality does not affect security.

5 The Online Phase

Our final protocol, presented in Figure 11, implements the online phase where the parties reveal the garbled circuit's shares and evaluate it. Our protocol is presented in the $\mathcal{F}_{\text{Prepocessing}}$-hybrid model. Upon reconstructing the garbled circuit and obtaining all input keys, the process of evaluation is similar to that of [Yao86], except here all parties run the evaluation algorithm, which involves each party computing n^2 PRF values per gate. During evaluation, the parties only see the randomly masked wire values and cannot determine the actual wire values. Upon completion, the parties compute the actual output using the output wire masks revealed from $\mathcal{F}_{\text{Prepocessing}}$. We conclude with the following theorem.

Theorem 5.1 *Let f be an n-party functionality $\{0,1\}^{n\cdot n_I} \mapsto \{0,1\}^{n_O}$ where n_I, n_O are the respective input/output sizes per party, and assume that F is a circular 2-correlation robust PRF. Then Protocol Π_{MPC} from Figure 11, universally composable computes f in the presence of a static, active adversary corrupting up to $n-1$ parties in the $\mathcal{F}_{\text{Prepocessing}}$-hybrid model.*

Proof overview. Our proof follows by first demonstrating that the adversary's view is computationally indistinguishable in both real and simulated executions. To be concrete, we consider an event for which the adversary successfully causes the bit transferred through some wire to be flipped and prove that this event can only occur with negligible probability (our proof is different to the proof in [LPSY15] as in our case the adversary may choose its additive error as a function of the garbled circuit). Then, conditioned on the event flip not occurring, we prove that the two executions are computationally indistinguishable via a reduction to the correlation robust PRF, inducing a garbled circuit that is indistinguishable. The complete proof is found below.

Proof: Let $\mathcal{A}$ be a PPT adversary corrupting a subset of parties $I \subset [n]$, and let $\bar{I}$ denote the set of honest parties. We prove that there exists a PPT simulator $\mathcal{S}$ with access to an ideal functionality $\mathcal{F}$ that implements f, that simulates the adversary's view.

In the simulator below, we let $W_\wedge \subset W$ denote the set of output wires of AND gates in the circuit C_f.

Description of the simulator $\mathcal{S}$.

1. INITIALIZATION. $\mathcal{S}$ incorporates the adversary $\mathcal{A}$, who controls the set of corrupt parties I, and internally emulates an execution of the honest parties running Π_{MPC} with $\mathcal{A}$:

The MPC Protocol - Π_{MPC}

Inputs: A circuit C_f computing the function f, which consists of XOR and AND gates. Let W be the set of all wires in C_f, W_{in_i} be the set of input wires for party P_i, and W_{out} be the set of output wires. Each party P_i has input $\{x_w\}_{w \in W_{\mathsf{in}_i}}$.
The parties execute the following commands in sequence.

Preprocessing: This sub-task is performed as follows.

- Call **Garbling** on $\mathcal{F}_{\mathrm{Prepocessing}}$ with input C_f.
- Each party P_i obtains the λ_w wire masks for every wire $w \in W_{\mathsf{in}_i} \cup W_{\mathsf{out}}$, and the keys $\{k^i_{w,0}\}_{w \in W}$ and R^i.

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

- For its input wires $w \in W_{\mathsf{in}_i}$, party P_i computes $\Lambda_w = x_w \oplus \lambda_w$, and broadcasts the public value Λ_w to all parties.
- Each party P_i broadcasts the keys k^i_{w,Λ_w}, for all input wires $w \in \{W_{\mathsf{in}_j}\}_{j \in [n]}$.
- The parties call **Open Garbling** on $\mathcal{F}_{\mathrm{Prepocessing}}$ to reconstruct $\tilde{g}^j_{a,b}$ for every gate g, $(a,b) \in \{0,1\}^2$ and $j \in [n]$.
- 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 u and v, and the output wire be labelled w. Let Λ_a and Λ_b be the respective public values on the input wires.
  1. If g is an XOR gate, set the public value on the output wire to be $\Lambda_c = \Lambda_a \oplus \Lambda_b$. In addition, for every $j \in [n]$, each party computes $k^j_{w,\Lambda_c} = k^j_{u,\Lambda_a} \oplus k^j_{v,\Lambda_b}$.
  2. If g is an AND gate then each party computes, for all $j \in [n]$:
  $$k^j_{w,\Lambda_c} = \tilde{g}^j_{\Lambda_a,\Lambda_b} \oplus \left(\bigoplus_{i=1}^{n} F_{k^i_{u,\Lambda_a}, k^i_{v,\Lambda_b}}(g \| j) \right)$$
  3. If $k^i_{w,\Lambda_c} \notin \{k^i_{w,0}, k^i_{w,1} = k^i_{w,0} \oplus R^i\}$, then P_i outputs `abort`. Otherwise, it proceeds.
  4. If $k^i_{w,\Lambda_c} = k^i_{w,0}$ then P_i sets $\Lambda_c = 0$; if $k^i_{w,\Lambda_c} = k^i_{w,1}$ then P_i sets $\Lambda_c = 1$.
  5. The output of the gate is defined to be $(k^1_{w,\Lambda_c}, \ldots, k^n_{w,c})$ and the public value Λ_c.
- Assuming no party aborts, everyone obtains a public value Λ_w, for all $w \in W_{\mathsf{out}}$. The parties can then recover the actual outputs from $y_w = \Lambda_w \oplus \lambda_w$, where λ_w was obtained in the preprocessing stage.

Figure 11: The MPC Protocol - Π_{MPC}.

2. GARBLING. $\mathcal{S}$ emulates the preprocessing phase of functionality $\mathcal{F}_{\text{Prepocessing}}$ on input $(\texttt{init}, C_f)$ where C_f is a Boolean circuit that computes f with a set of wires W and a set G of AND gates.

   Upon receiving the input $(\texttt{Garbling}, C_f)$ from the adversary the simulator emulates the garbling phase as follows.

   - For each $i \in I$, $\mathcal{S}$ receives a global difference $R^i \in \{0,1\}^\kappa$ from $\mathcal{A}$.
   - For every $w \in \{W_{\mathsf{in}_i}\}_{i \in I}$, $\mathcal{S}$ receives from $\mathcal{A}$ an input wire key $k^i_{w,0} \in \{0,1\}^\kappa$ and a wire mask $\lambda_w \in \{0,1\}$.
   - For each $w \in W_\wedge$, $\mathcal{S}$ receives a set of keys $\{k^i_{w,0}\}_{i \in I}$.
   - For each $w \in W_\wedge \cap W_{\mathsf{out}}$, $\mathcal{S}$ samples a random mask $\lambda_w \in \{0,1\}$. Then, in topological order, for each subsequent XOR gate with wires (u, v, w), let $\lambda_w = \lambda_u \oplus \lambda_v$.
   - Upon receiving a message $\texttt{OK}$ from $\mathcal{A}$, send λ_w to $\mathcal{A}$ for each circuit output wire w.

3. ONLINE COMPUTATION. The simulator interacts with $\mathcal{A}$ in the online phase and generates a simulated garbled circuit, as follows.

   - For each $w \in \{W_{\mathsf{in}_i}\}_{i \in \bar{I}}$, $\mathcal{S}$ samples a random public value $\Lambda_w \in \{0,1\}$ and sends this to $\mathcal{A}$.
   - For each $w \in \{W_{\mathsf{in}_i}\}_{i \in I}$, $\mathcal{S}$ receives from $\mathcal{A}$ a public value Λ_w and computes $x_w = \Lambda_w \oplus \lambda_w$. $\mathcal{S}$ sends the extracted inputs $\{x_w\}_{i \in I, w \in W_{\mathsf{in}_i}}$ to the ideal functionality computing f, receiving the output $\boldsymbol{y} = \{y_w\}_{w \in W_{\mathsf{out}}}$.
   - $\mathcal{S}$ samples at random the honest parties' input wire keys $\{k^i_{w,\Lambda_w}\}_{i \in \bar{I}, w \in W_{\mathsf{in}_i}}$, and sends these to $\mathcal{A}$. It receives back the adversary's input keys $\{\hat{k}^i_{w,\Lambda_w}\}_{i \in I, w \in W_{\mathsf{in}_i}}$ (which may be different to the keys received in the garbling phase).
   - $\mathcal{S}$ now generates public values for the remaining wires of the circuit. Let $W_{\mathsf{final}\text{-}\wedge} \subset W_\wedge$ denote the output wires of AND gates g where no successor of g is an AND gate (that is, those in the last AND layer of the circuit).
     - For each $w \in W_{\mathsf{out}}$, let $\Lambda_w = y_w \oplus \lambda_w$.
     - For each $w \in W_\wedge \setminus W_{\mathsf{final}\text{-}\wedge}$, sample $\Lambda_w \in \{0,1\}$.
     - For all wires $w \in W_{\mathsf{final}\text{-}\wedge} \setminus W_{\mathsf{out}}$, sample Λ_w at random subject to the constraint that these form a satisfying assignment to the circuit with outputs $\{\Lambda_w\}_{w \in W_{\mathsf{out}}}$.[2]
   - For every XOR gate with input wires (u, v) and output wire w, $\mathcal{S}$ sets $k^i_{w,0} = k^i_{u,0} \oplus k^i_{v,0}$ and $k^i_{w,1} = k^i_{w,0} \oplus R^i$ for all $i \in [n]$, and (if it was not previously computed) $\Lambda_w = \Lambda_u \oplus \Lambda_v$.
   - ACTIVE PATH GENERATION. In this step the simulator computes an active path of the garbled circuit, which corresponds to the sequence of keys that will be observed by the adversary.

     For every wire $w \in W_\wedge$, with input wires to that gate (u, v), $\mathcal{S}$ honestly generates the entry in row (Λ_u, Λ_v) by computing:

$$\tilde{g}^j_{\Lambda_u,\Lambda_v} = \left(\bigoplus_{i=1}^{n} F_{k^i_{u,\Lambda_u}, k^i_{v,\Lambda_v}}(g \| j) \right) \oplus k^j_{w,\Lambda_w}, \quad \text{for } j = 1, \ldots, n$$

[2] This is needed to ensure that, for instance, if an output wire w comes from the XOR of two previous AND gates with output wires (u, v), then the public values Λ_u, Λ_v are chosen to satisfy $\Lambda_w = \Lambda_u \oplus \Lambda_v$, as required.

fixing $\tilde{\mathbf{g}}_{\Lambda_u,\Lambda_v} = (\tilde{g}^1_{\Lambda_u,\Lambda_v}, \ldots, \tilde{g}^n_{\Lambda_u,\Lambda_v})$. The remaining three rows are sampled uniformly at random from $\{0,1\}^{n\kappa}$. Importantly, $\mathcal{S}$ never uses the inactive keys $k^i_{u,\bar{\Lambda}_u}, k^i_{v,\bar{\Lambda}_v}$ and $k^i_{w,\bar{\Lambda}_w}$ in order to generate the garbled circuit.

- The simulator hands the adversary the complete garbled circuit. In case the adversary aborts, the simulator sends $\perp$ to the ideal functionality and aborts. Otherwise, the simulator obtains an additive error $\boldsymbol{e} = \{e_g^{a,b}\}_{a,b\in\{0,1\},g\in G}$ and computes the modified garbled circuit as $\tilde{\mathbf{g}}_{\Lambda_u,\Lambda_v} + e_g^{\Lambda_u,\Lambda_v}$.

  Next, the simulator evaluates the modified circuit using the input wire keys $\{\hat{k}^i_{w,\Lambda_w}\}_{i\in I, w\in W_{\mathsf{in}_i}}$ and $\{k^j_{w,\Lambda_w}\}_{j\in\bar{I}, w\in W_{\mathsf{in}_j}}$ and checks whether the honest parties would have aborted. Namely, for each AND gate with output wire w, whether the evaluation reveals the honest parties' keys associated with Λ_w. If this check fails for any gate then the simulator outputs fail and aborts.

This concludes the description of the simulation. Note that the difference between the simulated and the real executions is regarding the way the garbled circuit is generated. More concretely, the simulated garbled circuit is only generated *after* the simulator extracts the adversary's input. Moreover, the simulated garbled gates include a single row that is properly produced, whereas the remaining three rows are picked at random. Let $\mathbf{HYB}^{\mathcal{F}_{\text{Prepocessing}}}_{\Pi_{\text{MPC}},\mathcal{A},\mathcal{Z}}(1^\kappa, z)$ denote the output distribution of the adversary $\mathcal{A}$ and honest parties in a real execution using Π_{MPC} with adversary $\mathcal{A}$. Moreover, let $\mathbf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(1^\kappa, z)$ denote the output distribution of $\mathcal{S}$ and the honest parties in an ideal execution.

We next define Flip to be the event that there exists an AND gate g and an honest party P_j, who, when evaluating the modified garbled circuit, (1) Does not abort; and (2) Obtains the incorrect key $k^j_{w,\bar{\Lambda}_w}$ for the output wire w of g. Note that this event implies that the adversary causes P_j to compute an incorrect value, as the bit value being transferred within the output wire w is now flipped with respect to P_j.

To prove that this event occurs with negligible probability, we consider an execution $\widetilde{\mathbf{IDEAL}}$ where a simulator $\tilde{\mathcal{S}}$ produces a view that is identical to the view produced by $\mathcal{S}$ in $\mathbf{IDEAL}$. Namely, the adversary's view is simulated exactly as in $\mathbf{IDEAL}$ by a simulator $\tilde{\mathcal{S}}$ with the exception that $\tilde{\mathcal{S}}$ further picks the global differences $\{R^j\}_{j\in\bar{I}}$ and computes inactive keys $k^j_{w,\bar{\Lambda}_w} = k^j_{w,\Lambda_w} \oplus R^j$ for $j \in \bar{I}, w \in W$ (which are never defined by $\mathcal{S}$). Moreover, the event for which the simulator outputs fail and aborts is modified as follows. Namely, the simulator aborts if there exists a gate for which the evaluation does not yield the key associated with Λ_w or with $\bar{\Lambda}_w$. Note that this event is well-defined since all the wire keys are chosen in this game. Then the difference between $\mathbf{IDEAL}$ and $\widetilde{\mathbf{IDEAL}}$ is the event that $\mathcal{S}$ aborts whereas $\tilde{\mathcal{S}}$ does not abort. Note that this event occurs when the adversary successfully flipped a wire value. Specifically, this will yield a valid evaluation in $\widetilde{\mathbf{IDEAL}}$ but not in $\mathbf{IDEAL}$. We next show that this event occurs with negligible probability.

Fix the additive error $\boldsymbol{e} = \{e_g^{a,b}\}_{a,b\in\{0,1\},g\in G}$ that is added to the simulated garbled circuit, and let $\tilde{\mathbf{g}}_{a,b} + e_g^{a,b}$ for all $g \in G$ and $a, b \in \{0,1\}$ denote the garbled circuit with the additive error. We prove that Flip only occurs with negligible probability in $\widetilde{\mathbf{IDEAL}}$ which implies that the statement also holds in $\mathbf{IDEAL}$. Intuitively, this is due to the fact that the adversary can only succeed in this attack by guessing correctly the global difference R^j of a honest party.

Lemma 5.1 *The probability that* Flip *occurs in* $\widetilde{\mathbf{IDEAL}}$ *is no more than* $2^{-\kappa}$.

Proof: Recall first that the simulated garbling in $\mathbf{IDEAL}$ involves only generating a single key k^j_w per wire and per honest party, which either corresponds to $k^j_{w,0}$ or $k^j_{w,1}$. Consequently, the simulator does not

even need to choose a global difference R^i in order to complete the garbling. Furthermore, the simulator in $\widetilde{\mathbf{IDEAL}}$ does generate these extra values, but never uses them; this means that the simulated garbled circuit given to $\mathcal{A}$ is completely independent of the honest parties' global differences.

Now, suppose Flip occurs with respect to party P_j, and let g be the first flipped AND gate (in some topological order), with input wires u, v and output wire w. Then, because P_j did not abort, the keys on wires u and v obtained by P_j must contain $k^j_{u,\Lambda_u}, k^j_{v,\Lambda_v}$. Note that it is possible that the corrupt parties' keys for these wires may be incorrect, so we denote these by $\hat{k}^i_u, \hat{k}^i_w$, for $i \in I$. Since gate g was flipped, the j-th entry of the active row of the garbled gate is

$$\hat{g}^j_{\Lambda_u,\Lambda_v} = \bigoplus_{i\in\bar{I}}^{n} \left(F_{k^i_{u,\Lambda_u},k^i_{v,\Lambda_v}}(g\|j)\right) \oplus \bigoplus_{i\in I}^{n} \left(F_{\hat{k}^i_u,\hat{k}^i_v}(g\|j)\right) \oplus k^j_{w,\bar{\Lambda}_w}.$$

To cause this to happen, the adversary needs to introduce an error into this entry of the original garbled gate $\tilde{\boldsymbol{g}}$, given by:

$$\begin{aligned}\Delta_g &:= \hat{g}^j_{\Lambda_u,\Lambda_v} \oplus \tilde{g}^j_{\Lambda_u,\Lambda_v} \\ &= \bigoplus_{i\in I}^{n} \left(F_{\hat{k}^i_u,\hat{k}^i_v}(g\|j) \oplus F_{k^i_{u,\Lambda_u},k^i_{v,\Lambda_v}}(g\|j)\right) \oplus R^j\end{aligned}$$

This boils down to correctly guessing R^j for the honest party P_j, which is bounded by $2^{-\kappa}$ as R^j is picked truly at random and independently of all other items in the execution. □

In the next step we prove that the ideal and real executions are indistinguishable, conditioned on the event Flip not occurring.

Lemma 5.2 *Conditioned on the event* $\overline{\mathsf{Flip}}$*, the following two distributions are computationally indistinguishable:*

- $\{\mathbf{HYB}^{\mathcal{F}_{\text{Prepocessing}}}_{\Pi_{\text{MPC}},\mathcal{A},\mathcal{Z}}(1^\kappa, z)\}_{\kappa\in\mathbb{N},z\in\{0,1\}^*}$
- $\{\widetilde{\mathbf{IDEAL}}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(1^\kappa, z)\}_{\kappa\in\mathbb{N},z\in\{0,1\}^*}$

Proof:

We begin by defining a slightly modified experiment $\widetilde{\mathbf{HYB}}$, which is identically distributed to the real execution $\mathbf{HYB}$, but differs in two ways. Firstly, we move the creation of the garbled circuit from the preprocessing stage to the online computation stage, after the parties have broadcast their masked inputs. Secondly, we modify the creation of the wire keys so that on receiving the parties' inputs $\{x_w\}_{w\in W_{\mathsf{in}_i}}$, we first evaluate the circuit C_f, computing the actual bit ℓ_w to be transferred through each $w \in W$, where W is the set of wires of C_f. We then choose two keys $k^i_{w,0}, k^i_{w,1}$ and a random bit λ^i_w for all $i \in \bar{I}$ and $w \in W$, and fix the active key for this wire to be $(k^1_{w,\ell_w\oplus\lambda_w}, \ldots, k^n_{w,\ell_w\oplus\lambda_w})$. The rest of this hybrid is identical to the real execution. Note that the garbled circuit is still computed according to $\mathcal{F}_{\text{Repocessing}}$, and the rest of the protocol is identical to $\mathbf{HYB}$, which induces the same view for the adversary. This hybrid execution is needed in order to construct a distinguisher for the correlation robustness assumption. Let $\widetilde{\mathbf{HYB}}^{\mathcal{F}_{\text{Prepocessing}}}_{\Pi_{\text{MPC}},\mathcal{A}}(1^\kappa, z)$ denote the output distribution of the adversary $\mathcal{A}$ and honest parties in this game.

Our proof of the lemma follows by a reduction to the correlation robustness of the PRF F (cf. Definition 2.2). Assume by contradiction the existence of an environment $\mathcal{Z}$, an adversary $\mathcal{A}$ and a non-negligible function $p(\cdot)$ such that

$$\left| \Pr[\mathcal{Z}(\widetilde{\mathbf{HYB}}^{\mathcal{F}_{\text{Prepocessing}}}_{\Pi_{\text{MPC}},\mathcal{A},\mathcal{Z}}(1^\kappa, z)) = 1] - \Pr[\mathcal{Z}(\widetilde{\mathbf{IDEAL}}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(1^\kappa, z)) = 1] \right| \geq \frac{1}{p(\kappa)}$$

for infinitely many κ's. We construct a distinguisher $\mathcal{D}'$ with access to an oracle $\mathcal{O}$ (that implements either Circ or Rand) that breaks the security of the correlation robustness assumption. Namely, we show that

$$\left| \Pr[R \leftarrow \{0,1\}^n; \mathcal{A}^{\mathsf{Circ}_R(\cdot)}(1^\kappa) = 1] - \Pr[\mathcal{A}^{\mathsf{Rand}(\cdot)}(1^\kappa) = 1] \right| \geq \frac{1}{p(\kappa)}.$$

Distinguisher $\mathcal{D}'$ receives the environment's input z and internally invokes $\mathcal{Z}$ and simulator $\mathcal{S}$, playing the role of functionality f. In more details,

- $\mathcal{D}'$ receives from $\mathcal{Z}$ the honest parties' inputs $\{x_w\}_{i \in \bar{I}, w \in W_{\mathsf{in}_i}}$.
- $\mathcal{D}'$ emulates the communication with the adversary (controlled by $\mathcal{Z}$) in the initialization, preprocessing and garbling steps as in the simulation with $\mathcal{S}$.
- For each wire u, let $\ell_u \in \{0,1\}$ be the actual value on wire u. Note that these values, as well as the output of the computation y, can be determined since $\mathcal{D}'$ knows the actual input of all parties to the circuit (where the adversary's input is extracted as in the simulation with $\mathcal{S}$).
- It next constructs the garbled circuit as follows. For each wire $w \in W_\wedge$, it samples the keys k^i_w, $i \in \bar{I}$ and computes the public value Λ_w as $\mathcal{S}$ would. Using the internal values ℓ_w, we can also compute the masks $\lambda_w = \ell_w \oplus \Lambda_w$.
- For each wire that is the output of an XOR gate with input wires u and v and output wire w, the distinguisher sets $k^i_w = k^i_u \oplus k^i_v$ for all $i \in [n]$, and $\Lambda_w = \Lambda_u \oplus \Lambda_v$.
- The distinguisher picks an honest party, say P_{i_0}, and samples global differences R^i for $i \in \bar{I} \setminus \{i_0\}$. For every $i \in \bar{I} \setminus \{i_0\}$ and $w \in W$, $\mathcal{D}'$ now has both keys $k^i_{w,\Lambda_w} = k^i_w$ and $k^i_{w,\overline{\Lambda}_w} = k^i_w \oplus R^i$.
- Finally, for each wire $w \in W_\wedge$, with input wires (u, v), the distinguisher computes four ciphertexts c_{00}, c_{01}, c_{10} and c_{11} as the garbled gate, that are generated as follows,
  - First, the j-th entry in the (Λ_u, Λ_v)-th row is computed as

$$\left(\bigoplus_{i=1}^{n} F_{k^i_{u,\Lambda_u}, k^i_{v,\Lambda_v}}(g \| j) \right) \oplus k^j_{w,\Lambda_w}.$$

  - Next, for for all $(a,b) \in \{0,1\}^2$ such that $(a,b) \neq (\Lambda_u, \Lambda_v)$ the distinguisher sets $\ell_{a,b} = 0$ if $g(a \oplus \lambda_u, b \oplus \lambda_v) = \ell_w$, and sets $\ell_{a,b} = 1$ otherwise. It then queries $h^j_{a,b} = \mathcal{O}(k^{i_0}_{u,\Lambda_u}, k^{i_0}_{v,\Lambda_v}, g, j, a \oplus \Lambda_u, b \oplus \Lambda_v, \ell_{a,b})$, and sets the j-th entry of row (a,b) in the garbled gate to be:

$$\left(\bigoplus_{i \neq i_0} F_{k^i_{u,a}, k^i_{v,b}}(g \| j) \right) \oplus h^j_{a,b} \oplus k^j_{w,\Lambda_w}$$

– For the output wires the distinguisher sets the public values as in the simulation.

- $\mathcal{D}'$ hands the adversary the complete description of the garbled circuit and concludes the execution as in the simulation with $\mathcal{S}$.
- $\mathcal{D}'$ outputs whatever $\mathcal{Z}$ does.

Note first that $\mathcal{D}'$ only makes legal queries to its oracle. Furthermore, if $\mathcal{O} = \mathsf{Circ}$ then the view of $\mathcal{A}$ is identically distributed to its view in the real execution of the protocol on the given inputs, whereas if $\mathcal{O} = \mathsf{Rand}$ then $\mathcal{A}$'s view is distributed identically to the output of the simulator described previously since the oracle's response is truly random in this case. This completes the proof. □

As a final stepping stone, we demonstrate that the probability Flip occurs in $\mathbf{HYB}$ is negligible too, due to the security assumptions on the PRF and our two previous Lemmas.

Lemma 5.3 *The probability that* Flip *occurs in* $\mathbf{HYB}$ *is bounded by* $2^{-\kappa} + \mathsf{negl}(\kappa)$ *for some negligible function* $\mathsf{negl}(\cdot)$.

Proof: Intuitively speaking, we prove that if Flip occurs in the real execution with a non-negligible probability, then we can leverage this distinguishing gap in order to break the circular 2-correlation robustness assumption. Namely, if this event occurs then it is possible to extract R^i and all pairs of inputs keys associated with every wire with respect to an honest party. Given all keys it is possible to recompute the garbled circuit and verify whether it was generated honestly or as in the simulation.

More formally, assume for a contradiction that

$$\Pr[\mathsf{Flip} \text{ occurs in } \mathbf{HYB}] \geq \frac{1}{q(\kappa)}$$

for some non-negligible function $q(\cdot)$ and infinitely many κ's. We construct a distinguisher $\mathcal{D}$ that breaks the security of the underlying PRF with non-negligible probability as follows.

1. Distinguisher $\mathcal{D}$ is identically defined as the distinguisher in the proof of Lemma 5.2, externally communicating with an oracle $\mathcal{Q}$ that either realizes the function Circ or Rand, while internally invoking $\mathcal{A}$. The only difference is that $\mathcal{D}$ chooses i_0 at random. This is due to the fact that the event Flip holds with respect to (at least) one honest party, whose identity is unknown.

2. Upon receiving the modified garbled circuit from $\mathcal{A}$, $\mathcal{D}$ evaluates the circuit on the parties' inputs and compares every active key $\tilde{k}_w^i$ that is revealed during the execution with the actual active key k_w^i that was created by $\mathcal{D}$ in the garbling phase. For every gate g for which there exists a difference $\Delta_g^i = \tilde{k}_w^i \oplus k_w^i$ for all $i \in \bar{I}$, $\mathcal{D}$ sets $R_g^i = \Delta_g^i$.

3. For every gate g for which $\mathcal{D}$ recorded a global difference $R_g^{i_0}$ for party i_0, it defines the inactive key for the output wire $w \in W$ of this gate by $k_{w,\bar{\Lambda}_w}^{i_0} = k_{w,\Lambda_w}^{i_0} \oplus R_g^{i_0}$. Next, for some gate g' for which wire w is an input wire (say associated with left input wire to g' w.l.o.g.), $\mathcal{D}$ queries its oracle on $(k_{w,\bar{\Lambda}_w}^{i_0}, k_v^{i_0}, g, j, \bar{a} \oplus \Lambda_w, b \oplus \Lambda_v, \ell_{a,b})$ where $k_v^{i_0}$ is the active key associated with the other input wire of g'. $\mathcal{D}$ compares this outcome with the values it obtained from its oracle for the query $(k_{w,\Lambda_w}^{i_0}, k_v^{i_0}, g, j, a \oplus \Lambda_w, b \oplus \Lambda_v, \ell_{a,b})$ during step 1. If equality holds, then $\mathcal{D}$ outputs Circ.

4. Upon concluding the execution so that $\mathcal{D}$ did not output Circ, it returns Rand.

Clearly, whenever $\mathcal{Q} = \mathsf{Circ}$ then the view of $\mathcal{A}$ is as in $\mathbf{HYB}$, so Flip occurs with probability at least $1/q(\kappa)$. The probability that it occurred with respect to party i_0 is at least $1/n$, therefore $\mathcal{D}$ outputs Circ with probability $\geq \frac{1}{q(\kappa)\cdot n}$.

On the other hand, as in the claim made in Lemma 5.2, when $\mathcal{Q} = \mathsf{Rand}$ then the adversary's view is as in $\mathbf{IDEAL}$, so the probability that Flip occurs is negligible, and thus $\mathcal{D}$ outputs Circ only with negligible probability. This implies a non-negligible distinguishing gap for the oracle $\mathcal{Q}$, which contradicts the circular 2-correlation robustness assumption on the PRF, and concludes the proof of this lemma. □

Informally, we have shown that, if the adversary introduces errors in the garbled circuit, this will be noticed with overwhelming probability and without causing any security harm. More formally, as the event Flip only happens with negligible probability in both the ideal and real executions, we have shown that:

$$\mathbf{HYB} \overset{c}{\approx} \widetilde{\mathbf{HYB}} \overset{c}{\approx} \widetilde{\mathbf{IDEAL}} \overset{c}{\approx} \mathbf{IDEAL}.$$

■

5.1 The Online Phase with $\mathcal{F}^{\mathrm{KQ}}_{\mathrm{Prepocessing}}$

In this section we now prove the following theorem, for the online protocol based on $\mathcal{F}_{\mathrm{Prepocessing}}$ with key queries. Recall that the functionality $\mathcal{F}^{\mathrm{KQ}}_{\mathrm{Prepocessing}}$ (defined at the end of Section 4.3) allows the adversary to additionally make a guess at an honest party's string R^i; the adversary learns if the guess is successful, and otherwise the functionality aborts.

Theorem 5.2 *Let f be an n-party functionality $\{0,1\}^{n\kappa} \mapsto \{0,1\}^{\kappa}$ and assume that F is a circular 2-correlation robust PRF. Then Protocol Π_{MPC} from Figure 11, UC-securely computes f in the presence of a static, active adversary corrupting up to $n-1$ parties in the $\mathcal{F}^{\mathrm{KQ}}_{\mathrm{Prepocessing}}$-hybrid model.*

In what follows, we discuss how to adapt the proof of Theorem 5.1 to support key queries.

Proof Sketch: We first modify the simulator specified in that proof. Namely, upon receiving the adversary's queries (i, R'), the simulator outputs 0. Intuitively, we claim that with overwhelming probability the adversary only sees zero responses to its key queries as it can only guess a global key with negligible probability. In the following, we formalize this intuition and discuss how to modify the proof of Theorem 5.1 by re-proving Lemma 5.1. Namely, we need to take into account the fact that the probability that the event Flip occurs also depends on the leakage obtained by the key queries made to the functionality.

We define a new hybrid game $\mathbf{H}$ where the simulator $\mathcal{S}_{\mathbf{H}}$ is defined identically to simulator $\mathcal{S}$ with the exception that $\mathcal{S}_{\mathbf{H}}$ knows the honest parties' inputs and further generates the inactive keys by picking global differences for the honest parties. Furthermore, for every key query (i, R') made by $\mathcal{A}$, $\mathcal{S}_{\mathbf{H}}$ verifies first whether $R' = R^i$ and aborts in case equality holds. Else, it replies with 0. Nevertheless, $\mathcal{S}_{\mathbf{H}}$ garbles the circuit the same way $\mathcal{S}$ does. Let Guess denote the event in $\mathbf{H}$ for which the adversary makes a key query R^i (meaning, it guesses the correct R^i value). We prove the following.

Lemma 5.4 *The probability that Flip occurs in $\mathbf{H}$ is no more than $(q+1)/2^{\kappa}$, where q is the number of key queries.*

Proof: We analyze the probability that the event Flip occurs.

$$\begin{aligned}\Pr(\mathsf{Flip}) &= \Pr(\mathsf{Flip}|\mathsf{Guess}) \cdot \Pr(\mathsf{Guess}) + \Pr(\mathsf{Flip}|\overline{\mathsf{Guess}}) \cdot \Pr(\overline{\mathsf{Guess}}) \\ &\leq \Pr(\mathsf{Guess}) + \Pr(\mathsf{Flip}|\overline{\mathsf{Guess}}) \leq q/2^{\kappa} + 2^{-\kappa}.\end{aligned}$$

□

This implies that the distributions induced within $\mathbf{IDEAL}$ and $\mathbf{H}$ are statistically close since the only difference is whenever event Guess occurs. We next claim that the proof of Lemma 5.2 holds with respect to $\mathbf{H}$ and $\mathbf{HYB}$.

Lemma 5.5 *Conditioned on the event* $\overline{\mathsf{Flip}}$*, the following two distributions are computationally indistinguishable:*

- $\{\mathbf{HYB}^{\mathcal{F}_{\text{Prepocessing}}}_{\Pi_{\text{MPC}},\mathcal{A}}(1^\kappa, z)\}_{\kappa\in\mathbb{N},z\in\{0,1\}^*}$
- $\{\mathbf{H}^{\mathcal{F}_{\text{Prepocessing}}}_{\Pi_{\text{MPC}},\mathcal{S}_{\mathbf{H}}}(1^\kappa, z)\}_{\kappa\in\mathbb{N},z\in\{0,1\}^*}$

Proof: This proof will have to incorporate the key queries as well. Namely, distinguisher $\mathcal{D}'$ first picks the identity of party i_0 at random. Then, whenever a key query (i_0, R^{i_0}) is made by $\mathcal{A}$, $\mathcal{D}'$ uses it to calculate the inactive keys of party P_{i_0} and checks whether this yields the garbling it obtained from its oracle. In case it does, $\mathcal{D}$ outputs Circ. Otherwise, if at the end of the execution no queries have resulted in a correct garbling, it outputs whatever $\mathcal{Z}$ does. □

Finally, we reprove Lemma 5.3 by demonstrating that if the success probability of Flip is non-negligibly higher in $\mathbf{HYB}$, then we can distinguish the two executions $\mathbf{HYB}$ and $\mathbf{H}$.

Lemma 5.6 *The probability that* Flip *occurs in* $\mathbf{HYB}$ *is bounded by* $2^{-\kappa} + \mathsf{negl}(\kappa)$ *for some negligible function* $\mathsf{negl}(\cdot)$.

Proof: This proof follows similarly to the proof of Lemma 5.3, except the reduction additionally needs to reply to the adversary's key queries. Namely, for each query (i, R^i) such that $i \neq i_0$, $\mathcal{D}$ can answer this query as it picked the global difference for that party. Moreover, for each query (i_0, R^{i_0}), $\mathcal{D}$ uses this query to fix a set of inactive keys for party i_0 and verifies whether this guess is correct by recomputing the garbled circuit and comparing it with the original garbled circuit. If equality holds then $\mathcal{D}$ responds with 1 to this query and outputs Circ. The rest of the proof follows identically. □ ■

6 Performance

In this section we present implementation results for our protocol from Section 4 for up to 9 parties. We also analyse the concrete communication complexity of the protocol and compare this with previous, state-of-the-art protocols in a similar setting.

We have made a couple of tweaks to our protocol to simplify the implementation. We moved the **Open Garbling** stage to the preprocessing phase, instead of the online phase. This optimizes the online phase so that the amount of communication is independent of the size of the circuit. This change means that our standard model security proof would no longer apply, but we could prove it secure using a random oracle instead of the circular-correlation robust PRF, similarly to [BHR12, LR15]. Secondly, when not working in a modular fashion with a separate preprocessing functionality, the share rerandomization step in the output stage is not necessary to prove security of the entire protocol, so we omit this.

	AES ($B = 5$)	AES ($B = 3$)	SHA-256 ($B = 4$)	SHA-256 ($B = 3$)
Prep.	1329	586.9	10443	6652
Online	35.34	33.30	260.58	252.8

Table 2: Runtimes in ms for AES and SHA-256 evaluation with 9 parties

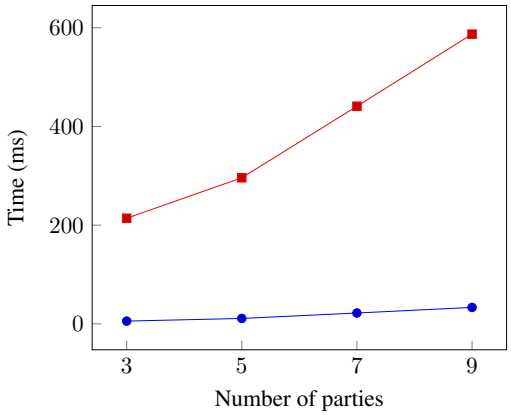

Figure 12: AES performance (6800 AND gates).

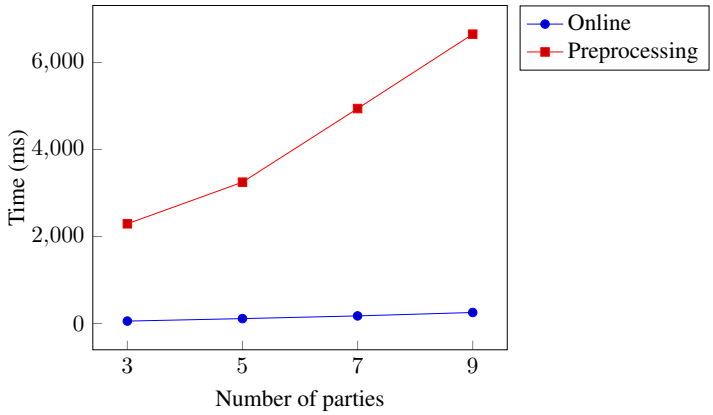

Figure 13: SHA-256 performance (90825 AND gates).

6.1 Implementation

We implemented our variant of the multi-party TinyOT protocol (Section A) using the `libOTe` library [Rin] for the fixed-correlation OTs. and tested it for between 3 and 9 parties. We benchmarked the protocol over a 1Gbps LAN on 5 servers with 2.3GHz Intel Xeon CPUs with 20 cores. For the experiments with more than 5 parties, we had to run more than one party per machine; this should not make much difference in a LAN, as the number of threads being used was still fewer than the number of cores. As benchmarks, we measured the time for securely computing the circuits for AES (6800 AND gates) and SHA-256 (90825 AND gates).

For the TinyOT bit and triple generation, every pair of parties needs two correlated OT instances running between them (one in each direction). We ran each OT instance in a separate thread with `libOTe`, so that each party uses $2(n-1)$ OT threads. This gave a small improvement ($\approx 6\%$) compared with running $n-1$ threads. We also considered a multiple execution setting, where many (possibly different) secure computations are evaluated. Provided the total number of AND gates in the circuits being evaluated is at least 2^{20}, this allows us to generate the TinyOT triples for all executions at once using a bucket size of $B = 3$, compared with $B = 5$ for one execution of AES or $B = 4$ for one execution of SHA-256. Since the protocol in Section A scales with B^2, this has a big impact on performance. The results for 9 parties, for the different choices of B, are shown in Table 2.

Figures 12–13 show how the performance of AES and SHA-256 scales with different numbers of parties, in the amortized setting. Although the asymptotic complexity is quadratic, the runtimes grow relatively slowly as the number of parties increases. This is because in the preprocessing phase, the amount of data sent *per party* is actually linear. However, the super-linear trend is probably due to the limitations of the total network capacity, and the computational costs.

Comparison with other works. We calculated the cost of computing the SPDZ-BMR protocol [LPSY15] using [KOS16] to derive estimates for creating the SPDZ triples (the main cost). Using MASCOT over $\mathbb{F}_{2^\kappa}$ with free-XOR, SPDZ-BMR requires $3n+1$ multiplications per garbled AND gate. This gives an estimated cost of at least 14 seconds to evaluate AES, which is over 20x slower than our protocol.

The only other implementation of actively secure, constant-round, dishonest majority MPC is the concurrent work of [WRK17b], which presents implementation figures for up to 256 parties running on Amazon servers. Their runtimes with 9 parties in a LAN setting are around 200ms for AES and 2200ms for SHA-256, which is around 3 times faster than our results. However, their LAN setup has 10Gbps bandwidth, whereas we only tested on machines with 1Gbps bandwidth. Since the bottleneck in our implementation is mostly communication, it seems that our implementation could perform similar to or even faster than theirs in the same environment, despite our higher communication costs. However, it is not possible to make an accurate comparison without testing both implementations in the same environment.

Our implementation does not scale well to a WAN environment in the cloud because we have not fully exploited the low round complexity of the protocol. However, the LAN results in our paper serve to demonstrate the practicality of the protocol and its low round complexity means that it will still be practical in a WAN setting, even more with some refactoring.

Compared with protocols based solely on secret-sharing, such as SPDZ and TinyOT, the advantage of our protocol is the low round complexity. We have not yet managed to benchmark our protocol in a WAN setting, but since our total round complexity is less than 20, it should perform reasonably fast. With secret-sharing, using e.g. TinyOT, evaluating the AES circuit requires at least 50 rounds in just the online phase (it can be done with 10 rounds [DNNR16], but this uses a special representation of the AES function, rather than a general circuit), whilst computing the SHA-256 circuit requires *4000 rounds*. In a network with 100ms delay between parties, the AES online time alone would be at least 4 seconds, whilst SHA-256 would take over *10 minutes* to securely compute in that setting. If our protocol is run in this setting, we should be able to compute both AES and SHA-256 in just a few seconds (assuming that latency rather than bandwidth is the bottleneck).

6.2 Communication Complexity Analysis

We now focus on analysing the concrete communication complexity of the optimized variant of our protocol and compare it with the state of the art in constant-round two-party and multi-party computation protocols. We have not implemented our protocol, but since the underlying computational primitives are very simple, the communication cost will be the overall bottleneck. As a benchmark, we estimate the cost of securely computing the AES circuit (6800 AND gates, 25124 XOR gates), where we assume that one party provides a 128-bit plaintext or ciphertext and the rest of them have an XOR sharing of a 128-bit AES key. This implies we have $128 \cdot n$ input wires and an additional layer of XOR gates in the circuit to add the key shares together. We consider a single set of 128 output wires, containing the final encrypted or decrypted message.

6.2.1 Complexity of Our TinyOT-Based Protocol

We now measure the exact communication cost of our optimized protocol based on TinyOT (in the random oracle model), in terms of number of bits sent over the network per party (multiply this by n for the overall complexity). We exclude one-time costs such as checking MACs, which can be done in a batch at the end, and initializing the base OTs. Consider a circuit with G AND gates, I input wires (in total) and O output wires. The costs of the different stages are as follows, with computational security parameter $\kappa = 128$ and statistical security parameter $s = 40$.

TinyOT Preprocessing. One triple and one random bit per AND gate, plus one random bit per input wire. From the analysis in Appendix A this gives:

$$(504B^2 + 168)(n-1)G + 168(n-1)I.$$

Garbling. Two bit openings for each AND gate (for the bit multiplication), one bit opening for every output wire and one private opening for every input wire, gives

$$2(n-1)G + (n-1)O + (n-1)I.$$

Open Garbling. The rerandomization step costs $\kappa(n-1)$ bits per party. Opening the garbled circuit can be done efficiently by each party sending their share to P_1, who broadcasts the result; this costs $4n\kappa G$ bits per party, giving a total of

$$4n\kappa G + \kappa(n-1).$$

Online. If party P_i has I_i input bits then the cost for P_i is $I_i + I \cdot (n-1) \cdot \kappa$ bits.

Note that for a single execution of AES we have $G = 6800, I = 128n$ and $O = 128$, which means for multi-party TinyOT we can choose $B = 4$, following the combinatorial analysis of [FLNW17].

6.2.2 Two Parties

In Table 3 we compare the cost of our protocol in the two-party case, with state-of-the-art secure two-party computation protocols. We instantiate our TinyOT-based preprocessing method with the optimized, two-party TinyOT protocol from [WRK17a], lowering the previous costs further. For consistency with the other two-party protocols, we divide the protocol costs into three phases: function-independent preprocessing, which only depends on the size of the circuit; function-dependent preprocessing, which depends on the exact structure of the circuit; and the online phase, which depends on the parties' inputs. As with the implementation, we move the garbled circuit opening to the function-dependent preprocessing, to simplify the online phase.

The online phase of the modified protocol is just two rounds of interaction, and has the lowest online cost of *any* actively secure two-party protocol.[3] The main cost of the function-dependent preprocessing is opening the garbled circuit, which requires each party to send 8κ bits per AND gate. This is slightly larger than the best Yao-based protocols, due to the need for a set of keys for every party in BMR.

In the batch setting, where many executions of the *same circuit* are needed, protocols such as [RR16] clearly still perform the best. However, if many circuits are required, but they may be different, or not known in advance, then our multi-party protocol is highly competitive with two-party protocols.

6.2.3 Comparison with Multi-Party Protocols

In Table 4 we compare our work with previous constant-round protocols suitable for any number of parties, again for evaluating the AES circuit. We do not present the communication complexity of the online phase as we expect it to be very similar in all of the protocols. We denote by MASCOT-BMR-FX an optimized

[3] If counting the *total* amount of data sent, in both directions, our online cost would be larger than [WRK17a], which is highly asymmetric. In practice, however, the latency depends on the largest amount of communication from any one party, which is why we measure in this way.

Protocol	# Executions	Function-indep. prep.	Function-dep. prep.	Online
[RR16]	32	–	3.75 MB	25.76 kB
	128	–	2.5 MB	21.31 kB
	1024	–	1.56 MB	16.95 kB
[NST17]	1	14.94 MB	227 kB	16.13 kB
	32	8.74 MB	227 kB	16.13 kB
	128	7.22 MB	227 kB	16.13 kB
	1024	6.42 MB	227 kB	16.13 kB
[WRK17a]	1	2.86 MB	570 kB	4.86 kB
	32	2.64 MB	570 kB	4.86 kB
	128	2.0 MB	570 kB	4.86 kB
	1024	2.0 MB	570 kB	4.86 kB
Ours + [WRK17a]	1	2.86 MB	872 kB	4.22 kB
	32	2.64 MB	872 kB	4.22 kB
	128	2.0 MB	872 kB	4.22 kB
	1024	2.0 MB	872 kB	4.22 kB

Table 3: Communication estimates for secure AES evaluation with our protocol and previous works in the two-party setting. Cost is the maximum amount of data sent by any one party, per execution.

variant of [LPSY15], modified to use free-XOR as in our protocol, with multiplications done using the OT-based MASCOT protocol [KOS16].

As in the previous section, we move the cost of opening the garbled circuit to the preprocessing phase for all of the presented protocols (again relying on random oracles). By applying this technique the online phase of our work is just two rounds, and has exactly the same complexity as the current most efficient *semi-honest* constant-round MPC protocol for any number of parties [BLO16], except we achieve active security. We see that with respect to other actively secure protocols, we improve the communication cost of the preprocessing by around 2–4 orders of magnitude. Moreover, our protocol scales much better with n, since the complexity is $O(n^2)$ instead of $O(n^3)$. The concurrent work of Katz et al. [WRK17b] requires around 3 times less communication than our protocol, which is due to their optimized version of the multi-party TinyOT protocol.

Acknowledgements

We are grateful to Moriya Farbstein and Lior Koskas for their valuable assistance with implementation and experiments. We also thank Yehuda Lindell for helpful feedback.

The first author was supported by the European Research Council under the ERC consolidators grant agreement No. 615172 (HIPS), and by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office. The second author was supported by the Defense Advanced Research Projects Agency (DARPA) and Space and Naval Warfare Systems Center, Pacific (SSC Pacific) under contract No. N66001-15-C-4070, and by the Danish

Protocol	Security	Function-indep. prep.		Function-dep. prep.	
		$n = 3$	$n = 10$	$n = 3$	$n = 10$
SPDZ-BMR	active	25.77 GB	328.94 GB	61.57 MB	846.73 MB
SPDZ-BMR	covert, pr. $\frac{1}{5}$	7.91 GB	100.98 GB	61.57 MB	846.73 MB
MASCOT-BMR-FX	active	3.83 GB	54.37 GB	12.19 MB	178.25 MB
[WRK17b]	active	4.8 MB	20.4 MB	1.3 MB	4.4 MB
Ours	active	14.01 MB	63.22 MB	1.31 MB	4.37 MB

Table 4: Comparison of the cost of our protocol with previous constant-round MPC protocols in a range of security models, for secure AES evaluation. Costs are the amount of data sent over the network per party.

Independent Research Council, Grant-ID DFF–6108-00169. The third author was supported by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 643161.

References

[ALSZ15] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer extensions with security for malicious adversaries. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 673–701. Springer, Heidelberg, April 2015.

[AOR+19] Abdelrahaman Aly, Emmanuela Orsini, Dragos Rotaru, Nigel P. Smart, and Tim Wood. Zaphod: Efficiently combining LSSS and garbled circuits in SCALE. In *WAHC '19: Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. ACM, 2019. https://eprint.iacr.org/2019/974.

[BCG+19a] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators: Silent OT extension and more. In *CRYPTO 2019*, 2019.

[BCG+19b] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. Efficient two-round OT extension and silent non-interactive secure computation. In *CCS*, pages 291–308, 2019.

[Bea92] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 420–432. Springer, Heidelberg, August 1992.

[BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 12*, pages 784–796. ACM Press, October 2012.

[BLN+15] Sai Sheshank Burra, Enrique Larraia, Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, Emmanuela Orsini, Peter Scholl, and Nigel P. Smart. High performance multi-party computation for binary circuits based on oblivious transfer. Cryptology ePrint Archive, Report 2015/472, 2015. http://eprint.iacr.org/2015/472.

[BLO16] Aner Ben-Efraim, Yehuda Lindell, and Eran Omri. Optimizing semi-honest secure multiparty computation for the internet. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 16*, pages 578–590. ACM Press, October 2016.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *22nd ACM STOC*, pages 503–513. ACM Press, May 1990.

[BOGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th ACM STOC*, pages 1–10. ACM Press, May 1988.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[CCL15] Ran Canetti, Asaf Cohen, and Yehuda Lindell. A simpler variant of universally composable security for standard multiparty computation. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 3–22. Springer, Heidelberg, August 2015.

[CDD+16] Ignacio Cascudo, Ivan Damgård, Bernardo David, Nico Döttling, and Jesper Buus Nielsen. Rate-1, linear time and additively homomorphic UC commitments. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part III*, volume 9816 of *LNCS*, pages 179–207. Springer, Heidelberg, August 2016.

[CKKZ12] Seung Geol Choi, Jonathan Katz, Ranjit Kumaresan, and Hong-Sheng Zhou. On the security of the "free-XOR" technique. In Ronald Cramer, editor, *TCC 2012*, volume 7194 of *LNCS*, pages 39–53. Springer, Heidelberg, March 2012.

[CKMZ14] Seung Geol Choi, Jonathan Katz, Alex J. Malozemoff, and Vassilis Zikas. Efficient three-party computation from cut-and-choose. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 513–530. Springer, Heidelberg, August 2014.

[DI06] Ivan Damgård and Yuval Ishai. Scalable secure multiparty computation. In Cynthia Dwork, editor, *CRYPTO 2006*, volume 4117 of *LNCS*, pages 501–520. Springer, Heidelberg, August 2006.

[DKL+13] 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 2013*, volume 8134 of *LNCS*, pages 1–18. Springer, Heidelberg, September 2013.

[DN07] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In Alfred Menezes, editor, *CRYPTO 2007*, volume 4622 of *LNCS*, pages 572–590. Springer, Heidelberg, August 2007.

[DNNR16] Ivan Damgård, Jesper Buus Nielsen, Michael Nielsen, and Samuel Ranellucci. Gate-scrambling revisited - or: The TinyTable protocol for 2-party secure computation. Cryptology ePrint Archive, Report 2016/695, 2016. http://eprint.iacr.org/2016/695.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Heidelberg, August 2012.

[DZ13] Ivan Damgård and Sarah Zakarias. Constant-overhead secure computation of Boolean circuits using preprocessing. In Amit Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 621–641. Springer, Heidelberg, March 2013.

[FKOS15] Tore Kasper Frederiksen, Marcel Keller, Emmanuela Orsini, and Peter Scholl. A unified approach to MPC with preprocessing using OT. In Tetsu Iwata and Jung Hee Cheon, editors, *ASIACRYPT 2015, Part I*, volume 9452 of *LNCS*, pages 711–735. Springer, Heidelberg, November / December 2015.

[FLNW17] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In *EUROCRYPT*, pages 225–255, 2017.

[FPY18] Tore Kasper Frederiksen, Benny Pinkas, and Avishay Yanai. Committed MPC - maliciously secure multiparty computation from homomorphic commitments. In *Public-Key Cryptography - PKC 2018 - 21st IACR International Conference on Practice and Theory of Public-Key Cryptography, Rio de Janeiro, Brazil, March 25-29, 2018, Proceedings, Part I*, pages 587–619, 2018.

[GL05] Shafi Goldwasser and Yehuda Lindell. Secure multi-party computation without agreement. *Journal of Cryptology*, 18(3):247–287, July 2005.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.

[IPS08] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer - efficiently. In David Wagner, editor, *CRYPTO 2008*, volume 5157 of *LNCS*, pages 572–591. Springer, Heidelberg, August 2008.

[IPS09] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Secure arithmetic computation with no honest majority. In Omer Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 294–314. Springer, Heidelberg, March 2009.

[KOS16] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 16*, pages 830–842. ACM Press, October 2016.

[KRRW18] Jonathan Katz, Samuel Ranellucci, Mike Rosulek, and Xiao Wang. Optimizing authenticated garbling for faster secure two-party computation. In *CRYPTO 2018*, 2018. https://eprint.iacr.org/2018/578.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP 2008, Part II*, volume 5126 of *LNCS*, pages 486–498. Springer, Heidelberg, July 2008.

[LP07] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In Moni Naor, editor, *EUROCRYPT 2007*, volume 4515 of *LNCS*, pages 52–78. Springer, Heidelberg, May 2007.

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

[LP11] Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. In Yuval Ishai, editor, *TCC 2011*, volume 6597 of *LNCS*, pages 329–346. Springer, Heidelberg, March 2011.

[LPSY15] Yehuda Lindell, Benny Pinkas, Nigel P. Smart, and Avishay Yanai. Efficient constant round multi-party computation combining BMR and SPDZ. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 319–338. Springer, Heidelberg, August 2015.

[LR15] 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, *ACM CCS 15*, pages 579–590. ACM Press, October 2015.

[LSS16] Yehuda Lindell, Nigel P. Smart, and Eduardo Soria-Vazquez. More efficient constant-round multi-party computation from BMR and SHE. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B, Part I*, volume 9985 of *LNCS*, pages 554–581. Springer, Heidelberg, October / November 2016.

[MRZ15] Payman Mohassel, Mike Rosulek, and Ye Zhang. Fast and secure three-party computation: The garbled circuit approach. In Indrajit Ray, Ninghui Li, and Christopher Kruegel:, editors, *ACM CCS 15*, pages 591–602. ACM Press, October 2015.

[NNOB12] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 681–700. Springer, Heidelberg, August 2012.

[NST17] Jesper Buus Nielsen, Thomas Schneider, and Roberto Trifiletti. Constant round maliciously secure 2pc with function-independent preprocessing using lego. In *24th NDSS Symposium*. The Internet Society, 2017. http://eprint.iacr.org/2016/1069.

[RBO89] Tal Rabin and Michael Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority (extended abstract). In *21st ACM STOC*, pages 73–85. ACM Press, May 1989.

[Rin] Peter Rindal. libOTe: an efficient, portable, and easy to use Oblivious Transfer Library. https://github.com/osu-crypto/libOTe.

[RR16] Peter Rindal and Mike Rosulek. Faster malicious 2-party secure computation with online/offline dual execution. In *USENIX*, pages 297–314, 2016.

[WRK17a] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. In *CCS*, pages 21–37, 2017.

[WRK17b] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In *CCS*, pages 39–56, 2017.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, October 1986.

[YWZ19] Kang Yang, Xiao Wang, and Jiang Zhang. More efficient MPC from improved triple generation and authenticated garbling. Cryptology ePrint Archive, Report 2019/1104, 2019. https://eprint.iacr.org/2019/1104.

A Protocol for GMW-Style MPC for Binary Circuits

Here we describe the full protocol for realizing the $\mathcal{F}_{\text{n-TinyOT}}$ functionality. It essentially consists of the bit triple generation protocol from [FKOS15], with some minor modifications, and a method for producing random shared bits with a consistency check that is similar to the bit/string check from Section 3.3.

We first recall the two-party and n-party MAC representations from Section 4:

$$[x^i]_{i,j} = (x^i, M_j^i, K_i^j)$$
$$[x] = (x^i, \{M_j^i, K_j^i\}_{j\neq i}))_{i\in[n]}, \quad M_j^i = K_i^j + x^i \cdot R^j$$

where in the two-party sharing $[x^i]_{i,j}$, P_i holds the share x^i and MAC M_j^i, whilst P_j holds the local key K_i^j and a fixed, global key R^j. In the n-party sharing, each party P_i holds $n-1$ MACs on x^i, as well as a key on x^j, for each $j \neq i$, and a global key R^i. Note that if P_i holds x^i and P_j holds the key R^j, a sharing $[x^i]_{i,j}$ can easily be created using one call to the correlated OT functionality (Figure 3), in which the correlation R^j is fixed by P_j in the initialization stage.

As required in the modified preprocessing protocol from Section 4, we need a method for opening $[x]$-shared values, both to all parties, and privately to a single party. These are straightforward, shown in Figure 14–15.

The main protocol, shown in Figure 16, consists of two main parts, for creating shared random bits, and for multiplication (AND) triples. Creating shared bits is straightforward, by using $\mathcal{F}_{\text{COT}}$ to MAC random bits, then opening a random linear combination of the MACs to ensure consistency.

To create shares of a random multiplication triple (x, y, z), each party first locally samples shares x^i, y^i, and then uses $\mathcal{F}_{\text{COT}}$ to authenticate these shares. The MAC on a share x^i is used to obtain a sharing of the product of x^i with a random bit $(u_\ell^{i,j} + v_\ell^{i,j})$ known to P_j (using a hash function), and then P_j converts this to a share of $x^i \cdot y^j$ by sending a correction bit in step 4a. This opens up an avenue for cheating, as

Subprotocol Π_{Open}

To open a shared value $[x]$ to all parties:

1. Each party P_i broadcasts its share x^i, and sends the MAC M_j^i to P_j, for $j \neq i$.
2. All parties compute $x = x^1 + \cdots + x^n$.
3. Each P_i has received MACs M_i^j, for $j \neq i$, and checks that

$$M_i^j = K_j^i + x^j \cdot R^i.$$

If any check fails, broadcast $\perp$ and abort.

Figure 14: Subprotocol for opening and checking MACs on n-party authenticated secret shares.

Subprotocol Π_{Open}^j

To open a shared value $[x]$ to only P_j:

1. Each party P_i, for $i \neq j$, privately sends its share x^i and MAC M_j^i to P_j.
2. P_j computes $x = x^1 + \cdots + x^n$, and checks that, for each $i \neq j$

$$M_j^i = K_i^j + x^i \cdot R^j.$$

If any check fails, broadcast $\perp$ and abort.

Figure 15: Subprotocol for private opening to one party.

P_j may send an incorrect correction value to some P_i. This could result in the triple being correct, or, if $x^i = 0$ the triple would still be correct but P_j would learn the bit x^i. These issues are addressed by the bucket-based cut-and-choose procedure in Figure 17, which first checks correctness by sacrificing triples, and then removes any potential leakage on the x values by combining several triples together. Note that the complex bucketing procedure is necessary for these steps, as opposed to simple pairwise checks, because with triples over $\mathbb{F}_2$, one pairwise check (or leakage combiner) can only guarantee correctness (or remove leakage) if *at least one* of the two triples is correct (or leakage-free). So, the cut-and-choose and bucketing procedure are done so that each bucket contains at least one good triple, with overwhelming probability.

A.1 Why the Need for Key Queries?

For completeness, we briefly explain why these are needed in $\mathcal{F}_{\text{n-TinyOT}}$, when using this protocol. After the protocol execution the environment learns the honest parties' outputs, which include MAC keys K_i and R. On the other hand, during the protocol the adversary sees values of the form (simplifying things slightly):

$$U = \mathsf{H}(K_i) + \mathsf{H}(K_i + R)$$

where H is modeled as a random oracle. In the security proof, U is simulated as a uniformly random value U, since the simulator, $\mathcal{S}$, does not know K_i or $K_i + R$. This means that if the environment later queries both K_i and $(K_i + R)$ to the random oracle then they could distinguish, as $\mathcal{S}$ would not be able to detect this, so the response would be inconsistent with the simulated U. However, with a **Key Query** command in the functionality, the simulator can detect this (based on the technique from [NNOB12]):

Protocol $\Pi_{\text{n-TinyOT}}$

Let $\mathsf{H} : \{0,1\}^\kappa \to \{0,1\}$ be a single-bit output hash function, modeled as a random oracle.

Initialize:

1. Each party P_i samples $R^i \leftarrow \{0,1\}^\kappa$.
2. Every ordered pair (P_i, P_j) calls $\mathcal{F}_{\text{COT}}$, where P_i sends $(\texttt{init}, R^i)$ and P_j sends $(\texttt{init})$.

Prep: To create m random shared bits $[b_1], \ldots, [b_m]$ do:

1. Each party P_i samples $m + \kappa$ random bits $b_1^i, \ldots, b_m^i, r_1^i, \ldots, r_\kappa^i \leftarrow \{0,1\}$.
2. Every ordered pair (P_i, P_j) calls $\mathcal{F}_{\text{COT}}$, where P_i is receiver and inputs $(\texttt{extend}, b_1^i, \ldots, b_m^i, r_1^i, \ldots, r_\kappa^i)$.
3. Use the previous outputs to define sharings $[b_1], \ldots, [b_m], [r_1], \ldots, [r_\kappa]$.
4. Check consistency of the $\mathcal{F}_{\text{COT}}$ inputs as follows:
   (a) Call $\mathcal{F}_{\text{Rand}}$ to obtain random field elements $\chi_1, \ldots, \chi_m \in \mathbb{F}_{2^\kappa}$
   (b) The parties locally compute (with arithmetic over $\mathbb{F}_{2^\kappa}$)
   $$[C] = \sum_{\ell=1}^{m} \chi_\ell \cdot [b_\ell] + \sum_{h=1}^{\kappa} X^{h-1} \cdot [r_h]$$
   (c) Each P_i now has a share $C^i \in \mathbb{F}_{2^\kappa}$, and the MACs and keys $(M_j^i, K_j^i)_{j \neq i}$ from $[C]$.
   (d) Each P_i rerandomizes C^i by privately sending fresh, random shares of zero to the other parties.
   (e) Broadcast $\bar{C}^i$ and reconstruct $c = \bar{C}^1 + \cdots + \bar{C}^n$.
   (f) Each party P_i defines and commits to the $n + 1$ values:
   $$C^i, \quad Z_j^i = M_j^i \quad (\text{for } j \neq i), \quad Z_i^i = \sum_{j \neq i} K_j^i + (C + C^i) \cdot R^i.$$
   (g) All parties open their commitments and check that, for each $j \in [n]$, $\sum_{i=1}^n Z_j^i = 0$. Additionally, each P_i checks that $Z_i^j = K_j^i + C^j \cdot R^i$. If any check fails, abort.

To create M AND triples, first create $m' = B^2 M + c$ triples as follows:

1. Each party P_i samples $x_\ell^i, y_\ell^i \leftarrow \mathbb{F}_2$ for $\ell \in [m']$
2. Every ordered pair (P_i, P_j) calls $\mathcal{F}_{\text{COT}}$, where P_i is receiver and inputs $(\texttt{extend}, x_1^i, \ldots, x_{m'}^i)$.
3. P_i and P_j obtain one value from $[x_\ell^i]_{i,j} = (M_\ell^{i,j}, K_\ell^{j,i})$, such that $M_\ell^{i,j} = K_\ell^{j,i} + x_\ell^i \cdot R^j \in \mathbb{F}_2^\kappa$.
4. For each $\ell \in [m']$ and each pair of parties (P_i, P_j):
   (a) P_j computes $u_\ell^{j,i} = \mathsf{H}(K_\ell^{j,i})$, $v_\ell^{j,i} = \mathsf{H}(K_\ell^{j,i} + R^j)$, and sends $d = u_\ell^{j,i} + v_\ell^{j,i} + y_\ell^j$ to P_i
   (b) P_i computes $w_\ell^{i,j} = \mathsf{H}(M_\ell^{i,j}) + x_\ell^i \cdot d = u_\ell^{j,i} + x_\ell^i \cdot y_\ell^j$
5. Each party P_i defines shares
   $$z_\ell^i = \sum_{j \neq i} (u_\ell^{i,j} + w_\ell^{i,j}) + x_\ell^i \cdot y_\ell^i$$
6. Every ordered pair (P_i, P_j) calls $\mathcal{F}_{\text{COT}}$, where P_i is receiver and inputs $(\texttt{extend}, \{y_\ell^i, z_\ell^i\}_{\ell \in [m']})$.
7. Use the above, and the previously obtained MACs on x_ℓ^i, to create sharings $[x_\ell], [y_\ell], [z_\ell]$.

Finally, run $\Pi_{\text{TripleBucketing}}$ on $([x_\ell], [y_\ell], z_\ell])_{\ell \in [m']}$, to output M correct and secure triples.

Figure 16: Protocol for TinyOT-style secure multi-party computation of binary circuits.

Subprotocol $\Pi_{\text{TripleBucketing}}$

The protocol takes as input $m' = B^2m + c$ triples, which may be incorrect and/or have leakage on the x component, and produces m triples which are guaranteed to be correct and leakage-free.
B determines the bucket size, whilst c determines the amount of cut-and-choose to be performed.

Input: Start with the shared triples $\{[x_i], [y_i], [z_i]\}_{i\in[m']}$.

I: Cut-and-choose: Using $\mathcal{F}_{\text{Rand}}$, the parties select at random c triples, which are opened with Π_{Open} and checked for correctness. If any triple is incorrect, abort.

II: Check correctness: The parties now have B^2m unopened triples.

1. Use $\mathcal{F}_{\text{Rand}}$ to sample a random permutation on $\{1, \ldots, B^2m\}$, and randomly assign the triples into mB buckets of size B, accordingly.
2. For each bucket, check correctness of the first triple in the bucket, say $[T] = ([x], [y], [z])$, by performing a pairwise sacrifice between $[T]$ and every other triple in the bucket. Concretely, to check correctness of $[T]$ by sacrificing $[T'] = ([x'], [y'], [z'])$:
   (a) Open $d = x + x'$ and $e = y + y'$ using Π_{Open}.
   (b) Compute $[f] = [z] + [z'] + d \cdot [y] + e \cdot [x] + d \cdot e$.
   (c) Open $[f]$ using Π_{Open} and check that $f = 0$.

III: Remove leakage: Taking the first triple in each bucket from the previous step, the parties are left with Bm triples. They remove any potential leakage on the $[x]$ bits of these as follows:

1. Place the triples into m buckets of size B.
2. For each bucket, combine all B triples into a single triple. Specifically, combine the first triple $([x], [y], [z])$ with $[T'] = ([x'], [y'], [z'])$, for every other triple T' in the bucket:
   (a) Open $d = y + y'$ using Π_{Open}.
   (b) Compute $[z''] = d \cdot [x'] + [z] + [z']$ and $[x''] = [x] + [x']$.
   (c) Output the triple $[x'']$, $[y]$, $[z'']$.

If all the checks and MAC checks passed, output the first triple from each of the m buckets in the final stage.

Figure 17: Checking correctness and removing leakage from triples with cut-and-choose.

Subprotocol Π_{Mult}

Given a multiplication triple $[a], [b], [c]$ and two shared values $[x], [y]$, the parties compute a sharing of $x \cdot y$ as follows:

1. Each party broadcasts $d^i = a^i + x^i$ and $e^i = b^i + y^i$.
2. Compute $d = \sum_i d^i$, $e = \sum_i e^i$, and run Π_{Open} to check the MACs on $[d]$ and $[e]$.
3. Output
$$\begin{aligned}[z] &= [c] + d \cdot [b] + e \cdot [a] + d \cdot e \\ &= [x \cdot y].\end{aligned}$$

Figure 18: Subprotocol for multiplying secret shared values using a triple.

- For each query Q, $\mathcal{S}$ looks up all previous queries Q_i, and sends $(Q + Q_i)$ to the **Key Query** of the functionality.
- If **Key Query** is successful then $\mathcal{S}$ knows that $Q + Q_i = R$, so can program the response $\mathsf{H}(Q)$ such that $\mathsf{H}(Q) + \mathsf{H}(Q_i) = U$, as required.

A.2 Security

In this section we formalize the security of the implementation of the **Prep** command of $\mathcal{F}_{\text{n-TinyOT}}$ in our $\Pi_{\text{n-TinyOT}}$ protocol. More concretely, we focus on the consistency check in the production of m random bits. This guarantees that the MAC keys are consistent, after which the triple generation protocol can be proven secure similarly to [FKOS15]. The exact deviations that are possible by a corrupt P_j in the bit generation are:

1. Provide inconsistent inputs R^j when acting as sender in the **Initialize** command of the $\mathcal{F}_{\text{COT}}$ instances with two different honest parties.
2. Input inconsistent shares $b^j_\ell, \ell \in [m]$ or $r^j_h, h \in [\kappa]$ when acting as receiver in the **Extend** command of $\mathcal{F}_{\text{COT}}$ with two different honest parties.

Note that in both of these cases, we are only concerned when the other party in the $\mathcal{F}_{\text{COT}}$ execution is honest, as if both parties are corrupt then $\mathcal{F}_{\text{COT}}$ does not need to be simulated in the security proof. We should also remark that preventing the first attack in the production of the random bits extends to preventing it everywhere else in the protocol, as the R^j values are fixed in the **Initialize** phase.

These two attacks are modelled by defining $R^{j,i}, b^{j,i}_\ell$ and $r^{j,i}_h$ to be the *actual* inputs used by a corrupt P_j in the above two cases. Without loss of generality, we pick a honest party P_{i_0} and fix $b^j_\ell = b^{j,i_0}_\ell, r^j_h = r^{j,i}_h, R^j = R^{j,i_0}$ to be the inputs by P_j that should be consistent with every other honest party. Let I be the set of corrupted parties. For each $j \in I$, we can resume the previous statements by defining the values:

$$\begin{aligned}
\Delta^{j,i_0} = 0, \quad \Delta^{j,i} = R^{j,i} + R^j, \quad i \notin (I \cup i_0) \\
\delta^{j,i_0}_\ell = 0, \quad \delta^{j,i}_\ell = b^{j,i}_\ell + b^j_\ell, \quad \ell \in [m], i \notin (I \cup i_0) \\
\hat{\delta}^{j,i_0}_\ell = 0, \quad \hat{\delta}^{j,i}_h = r^{j,i}_h + r^j_h, \quad h \in [\kappa], i \notin (I \cup i_0).
\end{aligned}$$

Note that $\Delta^{j,i}$ is fixed in the initialization of $\mathcal{F}_{\text{COT}}$, whilst $\delta^{j,i}_\ell$ may be different for every OT. Whenever P_i and P_j are both corrupt, or both honest, for convenience we define $\Delta^{j,i} = 0$ and $\delta^{j,i} = 0$. The above means that the outputs of $\mathcal{F}_{\text{COT}}$ with (P_i, P_j) then satisfy:

$$M^j_i(b^{j,i}_\ell) = K^i_j(b^{j,i}_\ell) + b^{j,i}_\ell \cdot R^{i,j}$$

or, equivalently:

$$M^j_i(b^j_\ell + \delta^{j,i}_\ell) = K^i_j(b^j_\ell + \delta^{j,i}_\ell) + (b^j_\ell + \delta^{j,i}_\ell) \cdot (R^i + \Delta^{i,j})$$

where $\delta^{j,i} \neq 0$ if P_j (the receiver) cheated, and $\Delta^{i,j} \neq 0$ if P_i (the sender) cheated. Remember from Section 4.1 that $M^j_i(b^j_\ell + \delta^{j,i}_\ell)$ represents the receiver's MAC on the value $b^j_\ell + \delta^{j,i}_\ell$ and $K^i_j(b^j_\ell + \delta^{j,i}_\ell)$ and represents the sender's MAC key on that same value.

We start by assuming that the corrupted party in the couple (P_i, P_j) running $\mathcal{F}_{\text{COT}}$ is the sender P_j, trying to have inconsistent correlations $R^{j,i}$ with different honest parties $P_i, i \notin I$. We prove the inconsistency impossible in the next claim:

Claim A.1 *If the **Prep** step of* $\Pi_{\text{n-TinyOT}}$ *succeeds then all the global keys* R^j *are consistent and well-defined, i.e.* $\Delta^{j,i} = 0$ *for every* $i, j \in [n]$.

Proof: We enumerate the possible deviations by the Adversary affecting the check $\sum_{i=1}^{n} Z_j^i = 0$ in Step 4g with which we want to catch inconsistent $R^{j,i}$ values to different honest parties. These possible disruptions are two:

In Step 4e, the parties broadcast $\bar{C}^i$ values, every corrupted $P_\ell, \ell \in I$ can send instead some adversarial value $\hat{C}^\ell$ such that $\sum_{j=1}^{n} \hat{C}^j = C + e$, where e is some additive error of the Adversary's choice. Finally, a similar active deviation is to commit to $\hat{Z}_j^\ell$ values, $\ell \in I$, in such a way that $\sum_{\ell \in I} \hat{Z}_j^\ell = \sum_{\ell \in I} Z_j^\ell + E^j$.

An active P_j trying to cheat has to pass the aforementioned mentioned check, which becomes:

$$0 = \sum_{i=1}^{n} \hat{Z}_j^i = E^j + Z_j^j + \sum_{i \neq j} Z_j^i = E^j + \left(\sum_{i \neq j} K_i^j(C^i) + (C + e + C^j) \cdot R^j \right) + \sum_{i \neq j} M_j^i(C^i) =$$

$$E^j + (C + e + C^j) \cdot R^j + \sum_{i \neq j} (K_i^j(C^i) + M_j^i(C^i)) = E^j + (C + e + C^j) \cdot R^j + \sum_{i \neq j} C^i \cdot R^{j,i} =$$

$$E^j + (C + e + C^j + \sum_{i \neq j} C^i) \cdot R^j + \sum_{i \neq j} C^i \cdot \Delta^{j,i} = E^j + e \cdot R^j + \sum_{i \neq j} C^i \cdot \Delta^{j,i}$$

As having inconsistent keys requires that there exists $i_0, i_1 \notin I$ such that $\Delta^{j,i_0} \neq \Delta^{j,i_1} \neq 0$, the attack would require the adversary to set $E^j + e \cdot R^j = C^{i_0} \cdot \Delta^{j,i_0} + C^{i_1} \cdot \Delta^{j,i_1}$. But this is negligible in κ, as the only information the adversary has about $C^{i_0}, C^{i_1} \in \mathbb{F}_{2^\kappa}$ at the time of committing to the values $\hat{Z}_j^\ell, \ell \in I$ is that they are two uniform additive shares of C, due to the rerandomization in Step 4d. ∎

Finally, we prove that a corrupted receiver P_j cannot input inconsistent values $b_\ell^{j,i}$ to different honest parties.

Claim A.2 *If the **Prep** step of* $\Pi_{\text{n-TinyOT}}$ *succeeds, every ordered pair* (P_i, P_j) *holds a secret sharing of* $b_\ell^j \cdot R^i$ *for every* $\ell \in [m]$. *In other words,* $\delta_\ell^{j,i} = 0$ *for every* i, j, ℓ.

Proof: For every ordered pair (P_i, P_j) we can define P_j's MAC on $[C^j]_{j,i}$ as

$$M_i^j(C^j) = \sum_{\ell=1}^{m} \chi_\ell \cdot M_i^j(b_\ell^{j,i}) + \sum_{h=1}^{\kappa} X^{h-1} \cdot M_i^j(r_h^{j,i})$$

and P_i's key on the same value as:

$$K_j^i(C^j) = \sum_{\ell=1}^{m} \chi_\ell \cdot K_j^i(b_\ell^{j,i}) + \sum_{h=1}^{\kappa} X^{h-1} \cdot K_j^i(r_h^{j,i})$$

In Step 4f of **Bits**, an adversarial P_j can also commit to incorrect MACs $\hat{Z}_i^j(c^j) = M_i^j(c^j) + E_i^j$ and $\hat{C}^j = C^j + e^j$. Nevertheless, in order to succeed an attack, the check $\hat{Z}_i^j = K_j^i(C^j) + \hat{C}^j \cdot R^i$ from Step 4g

would have to hold. This check implies the following:

$$\begin{aligned}
& M_i^j(C^j) + E_i^j = K_j^i(C^j) + (C^j + e^j) \cdot R^i \\
\Leftrightarrow\ & E_i^j + (C^j + e^j) \cdot R^i = M_i^j(C^j) + K_j^i(C^j) = (\sum_{\ell=1}^{m} \chi_\ell \cdot b_\ell^{j,i} + \sum_{h=1}^{\kappa} X^{h-1} \cdot r_h^{j,i}) \cdot R^i \\
& \Leftrightarrow E_i^j = \left(C^j + e^j + \sum_{\ell=1}^{m} \chi_\ell \cdot (b_\ell^j + \delta_\ell^{j,i}) + \sum_{h=1}^{\kappa} X^{h-1} \cdot (r_h^j + \hat{\delta}_h^{j,i}) \right) \cdot R^i \\
& = (e^j + \sum_{\ell=1}^{m} \chi_\ell \cdot \delta_\ell^{j,i} + \sum_{h=1}^{\kappa} X^{h-1} \cdot \hat{\delta}_h^{j,i}) \cdot R^i
\end{aligned}$$

An active P_j has then just two options to cheat P_i, both with only probability $2^{-\kappa}$ to succeed:

1. Setting $E_i^j = (e^j + \sum_{\ell=1}^{m} \chi_\ell \cdot \delta_\ell^{j,i} + \sum_{h=1}^{\kappa} X^{h-1} \cdot \hat{\delta}_h^{j,i}) \cdot R^i \neq 0$, which requires guessing the string $R^i \in \mathbb{F}_{2^\kappa}$ kept secret by the honest party P_i.

2. Setting $E_i^j = 0$ and $e_j = \sum_{\ell=1}^{m} \chi_\ell \cdot \delta_\ell^{j,i} + \sum_{h=1}^{\kappa} X^{h-1} \cdot \hat{\delta}_h^{j,i}$ for every $i \notin I$. As $\delta_\ell^{j,i_0} = \hat{\delta}_h^{j,i_0} = 0$, this implies that $e_j = 0$. Thus, for every $i \notin (I \cup i_0)$ it needs to hold that

$$0 = \sum_{\ell=1}^{m} \chi_\ell \cdot \delta_\ell^{j,i} + \sum_{h=1}^{\kappa} X^{h-1} \cdot \hat{\delta}_h^{j,i} = \sum_{h=1}^{\kappa} X^{h-1} \cdot (\hat{\delta}_h^{j,i} + \sum_{\ell=1}^{m} \delta_\ell^{j,i} \cdot \chi_{\ell,h}),$$

   where the $\chi_{\ell,h}$ values are defined in such a way that $\chi_\ell = \sum_{h=1}^{\kappa} X^{h-1} \cdot \chi_{\ell,h}$. This would need that, for every $h \in [\kappa]$:

$$\hat{\delta}_h^{j,i} = \sum_{\ell=1}^{m} \delta_\ell^{j,i} \cdot \chi_{\ell,h} \in \mathbb{F}_2,$$

   which can only happen with probability $1/2$ for each of them, as $\chi_{\ell,h} \in \mathbb{F}_2$ are uniformly random sampled field elements after the deviations $\hat{\delta}_h^{j,i}, \delta_\ell^{j,i}$ have been defined.

■

A.3 Parameters

Based on the analysis from previous works [FKOS15, FLNW17, WRK17b], if roughly 1 million triples are created at once then the buckets in the cut-and-choose stages can be of size $B = 3$, to guarantee security except with probability 2^{-40}. The additional cut-and-choose parameter c can be as low as 3, so is insignificant as we initially need $m' = B^2 m + c$ triples to produce m final triples.

A.4 Communication Complexity

Here we analyse the communication complexity of $\Pi_{\text{n-TinyOT}}$. The cost of creating one shared random bit is the same as one invocation of the `extend` command in $\mathcal{F}_{\text{COT}}$ between all pairs of parties, giving $n(n-1)(\kappa + s)$ bits (we ignore the consistency check, since this cost amortizes away when creating many bits).

The cost of one triple (not counting the bucketing stage), is 3 calls to $\mathcal{F}_{\text{COT}}$ between every pair of parties for authenticating shares of (x, y, z), plus sending one correction bit between every pair of parties, giving

$n(n-1)(3(\kappa+s)+1)$ bits. This is then multiplied by approximately B^2 to account for the bucketing. When creating a batch of at least a million triples (with $s = 40$), we can set $B = 3$, so the overall cost per party is around $(n-1)27 \cdot (\kappa+s)$ bits.

We remark that when checking a large number of MACs using Π_{Open} or Π^i_{Open}, the checks can be batched together, by first computing a random linear combination of all MACs, and checking the MAC on this, as in e.g. [DKL+13, KOS16]. This means that the cost of checking many MACs is roughly the cost of checking one, which is why we did not factor the MAC checks into the cost of the bucketing stage.

A.5 Round Complexity

Initializing the correlated OTs can be done with any 2-round OT protocol. Extending the correlated OTs using [NST17] and [ALSZ15] takes 3 rounds. Note that when authenticating random bits, the s additional bits in the consistency check can be created in parallel with the original m bits, giving an overall cost of 5 rounds for random bits.

The triple generation consists of one set of correlated OTs (2 + 3 rounds), plus 1 round, plus another round of correlated OTs (3 rounds). Then there are 2 rounds for $\mathcal{F}_{\text{Rand}}$ in the bucketing (which can all be done in parallel), one round for the openings in step 2a and one round for step 2c. The openings in step 2a can be merged with the previous round. This gives a total of 13 rounds.

A.6 Realizing General Secure Computation

The previous protocol can easily be used to implement a general secure computation functionality such as $\mathcal{F}_{\text{BitMPC}}$. The main feature missing is the ability for parties to provide inputs, since we only need to create random bits and triples for our application to garbled circuits. However, this is easy to do with a standard technique: if P_i wishes to secret-share an input x, the parties do as follows:

- Create a shared random bit $[b]$.
- Open b to P_i using Π^i_{Open}.
- P_i broadcasts $d = x - b$.
- All parties compute $[x] = [b] + d$.