Low Cost Constant Round MPC Combining BMR and Oblivious-Transfer

Carmit Hazay[*] Peter Scholl[†] Eduardo Soria-Vazquez[‡]

March 1, 2017

Abstract

In this work, we present a new universally composable, actively secure, constant round multi-party protocol for generating BMR garbled circuits with free-XOR and reduced costs. Specifically, the cost of garbling an AND gate is *essentially the same* as securely evaluating one AND gate using any non-constant-round, GMW-style multi-party computation (MPC) protocol, with a small additive communication overhead that is linear in the security parameter. We further introduce a second realization of the preprocessing phase that is less generic but more efficient, by building on optimized multi-party 'TinyOT'-style protocols based on information-theoretic message authentication codes.

An interesting consequence of our work is that, with current techniques, constant round MPC for binary circuits is not much more expensive than practical, non-constant round protocols. We estimate that the concrete communication cost of our preprocessing protocol improves upon previous works by up to *three orders of magnitude*, and with low computational complexity. We also improve asymptotically by reducing the overall communication complexity from $O(n^3)$ to $O(n^2)$, and our construction is even highly competitive in the two-party setting.

[*]Bar-Ilan University, Israel. Email: carmit.hazay@biu.ac.il.
[†]University of Bristol, UK. Email: peter.scholl@bristol.ac.uk.
[‡]University of Bristol, UK. Email: eduardo.soria-vazquez@bristol.ac.uk.

Contents

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], 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 such as [GMW87] and [BOGW88] tend to have lower communication requirements in terms of bandwidth, but require a large number of rounds of communication, which increases with the complexity of the function. 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, such as in Yao's protocol [Yao86], are the core behind all practical, constant round protocols for secure computation. In the two-party setting, 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]. 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 another (non-constant-round) MPC protocol, as in the BMR protocol [BMR90]. This still leads to a low depth circuit, hence a constant round protocol overall, because all gates can be garbled in parallel. We note that this paradigm has received very little attention, compared with two-party protocols. The original BMR construction uses generic zero-knowledge techniques for proving correct computation of PRG values, so is impractical; the only practical, actively secure protocols of this kind are the recent works of Lindell et al. [LPSY15, LSS16], which use somewhat homomorphic encryption (SHE) or generic MPC. 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.

1.1 Our Contributions

In this work, we present a new universal composable actively secure, constant round multi-party protocol for generating BMR garbled circuits with free-XOR in the presence of dishonest majority. As in prior constructions, our protocol is comprised of two phases: an preprocessing phase where the garbled circuit is mutually generated by all parties, and an online phase where the parties obtain the output of the computation. While the online phase is typically efficient, the focus of recent works was on optimizing the preprocessing complexity and simplifying its view, where the main bottleneck is with respect to garbling the AND gates. In that context, our protocol has a cost for garbling an AND gate that *is essentially the same as securely evaluating one AND gate using any non-constant-round, GMW-style MPC protocol*, with a small additive communication overhead that is linear in the security parameter. The additional costs in our main protocol come from standard, efficient primitives such as oblivious transfer extensions and universal hashing.

Our construction employs several very appealing features. For a start, we embed into the modeling of the preprocessing functionality, which computes the garbled circuit, an additive error introduced into the garbling by the adversary. Concretely, we extend the functionality from [LPSY15] so that it obtains a vector of additive errors from the adversary to be applied to on each garbled gate, which captures the fact that the adversary may submit inconsistent keys and pseudorandom function (PRF) values. We further strengthen this by allowing the adversary to pick the error *adaptively* after seeing the garbled circuit (in prior constructions this error is independent of the garbling) and allowing corrupt parties to choose their own PRF keys, possibly not at random. These modifications allow us to obtain a much more efficient preprocessing protocol.

Secondly, we present two constructions for realizing the preprocessing functionality: one based on any generic MPC protocol, and one exploiting specific features of practical secret-sharing based protocols. Both protocols proceed in three main stages where firstly, the parties locally sample all of their keys and shares of wire masks for the garbled circuit. In the second stage, the parties compute additive shares of the garbled circuit and finally, the parties rerandomize their shares by distributing a fresh, random secret-sharing of each share to the other parties, via private channels. This ensures that the shares do not leak any information on the PRF values.

Our second realization of the preprocessing phase is less generic but more efficient, by building on optimized multi-party 'TinyOT'-style protocols such as [BLN$^+$15, FKOS15] based on information-theoretic message authentication codes (MACs). An important challenge introduced by these protocols is due to the fact that the adversary is given an oracle, which can be used to attempt to guess the MAC keys (which, in turn, function as the parties' global differences to the garbling). In order to handle this leakage, we modify our preprocessing functionality and allow the adversary to submit key queries. We then extend our proof of the online phase and demonstrate that it holds for this functionality as well.

1.1.1 Comparison with Other Approaches

As mentioned earlier, the only previous constructions of practical, actively secure, constant round MPC use SHE, and express the garbling function as an arithmetic circuit over a large finite field. In these protocols, garbling even a single AND gate requires computing several multiplications over a large field with SHE. This requires the parties to perform 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. In contrast, our protocol only requires *one bit multiplication per AND gate*, and a small number of extra OTs to create the garbled gate.

We estimate that this improves upon previous works by around *three orders of magnitude* in terms of total communication cost, and the main computational costs are from standard symmetric primitives, so far cheaper than SHE. Even in the two-party setting, our protocol is highly competitive: after opening the garbled circuit, the online phase requires just one round of interaction, with lower per-party communication than all previous protocols.

Overall, our protocol significantly narrows the gap between the cost of constant-round and many-round MPC protocols for binary circuits, and also has great potential for improvement, by optimizing multi-party TinyOT. 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. Also, any efficiency improvements to non-constant round protocols such as multi-party TinyOT would directly give a similar improvement to our protocol.

In Table 1 we sketch how our work compares with previous protocols making use of BMR. In all protocols, we approximate the main overall cost with the number of finite field multiplications needed in the underlying MPC protocol used to create the garbled circuit. In [LPSY15], this can be based on the SPDZ protocol [DPSZ12] using SHE, or more generally any MPC protocol for arithmetic circuits modulo $p \approx 2^\kappa$. In [LSS16], a dedicated protocol based on SHE was given, but this still requires expensive zero-knowledge

Protocol	Underlying Primitive	Free XOR	Main Cost per Garbled Gate
SPDZ-BMR [LPSY15]	MPC	✗	$4n+5$ mult. in $\mathbb{F}_p$[1]
SHE-BMR [LSS16]	SHE (depth 4)	✗	$8n+5$ SHE mult. + $O(n^2)$ ZKPoPKs
MASCOT-BMR-FX	OT	✓	$3n+1$ mult. in $\mathbb{F}_{2^\kappa}$
This work	OT	✓	1 mult. in $\mathbb{F}_2$

Table 1: Comparison with previous actively secure, constant round MPC protocols.

proofs of plaintext knowledge (ZKPoPK) and homomorphic computations. 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 the OT-based MASCOT protocol [KOS16].

1.2 Technical Overview

Our protocol is based on the recent free-XOR variant of BMR garbling used for semi-honest MPC in [BLO16]. In that scheme, a garbling of the g-th AND gate with input wires u, v and output wire w, consists of the $4n$ values (where n is the number of parties):

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

Here, F is a double-key PRF, $R^j \in \{0,1\}^\kappa$ is a fixed correlation string for free-XOR known to party P_j, and the keys $k^j_{u,a}, k^j_{v,b} \in \{0,1\}^\kappa$ are also known to P_j. Furthermore, the wire masks $\lambda_u, \lambda_v, \lambda_w \in \{0,1\}$ are random, additively secret-shared bits known by no single party.

The main idea behind BMR is to compute the garbling, except for the PRF values, with a general MPC protocol. The analysis of [LPSY15] showed that it is not necessary to prove in zero-knowledge that every party inputs the correct PRF values to the MPC protocol that computes the garbling. This is because when evaluating the garbled circuit, each party P_j can check that the decryption of the j-th entry in every garbled gate gives one of the keys $k^j_{w,b}$, 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 from it to guess a key.

Our garbling protocol proceeds by computing a random, unauthenticated, additive secret sharing of the garbled circuit. This differs from previous works [LPSY15, LSS16], which obtain *authenticated* (with MACs, or SHE ciphertexts) secret sharings of the *whole* of (1). Our protocol greatly reduces this complexity, since the PRF values and keys (on the first line of (1)) do not need to be authenticated. The main challenge, therefore, is to compute shares of the products on the second line of Equation (1). Similarly to [BLO16], a key observation that allows efficiency is the fact that these multiplications 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.

[1]If instantiated with SPDZ [DPSZ12], add $O(n^3)$ ZKPoPKs due to the use of SHE.

To compute the bit product $\lambda_u \cdot \lambda_v$, we can use any actively secure GMW-style MPC protocol for binary circuits, such as multi-party TinyOT [BLN+15]. This protocol is only needed for computing one secure AND per garbled AND gate, since all 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 the string R^j, known to P_j. We give two variants for computing this product, the first one being more general and the second more concretely efficient. The first solution performs the multiplication by running actively secure correlated OT between P_j and every other party, where P_j inputs R^j as the fixed OT correlation. The parties then run a consistency check by applying a universal linear hash function to the outputs, ensuring the correct inputs were provided to the OT. The second method exploits specific features of 'TinyOT'-style protocols based on information-theoretic MACs, and allows us to obtain the bit/string products directly from the MACs, with no interaction.

After creating shares of all these products, the parties can compute a share of the whole garbled circuit. These shares must then be rerandomized, before they can be broadcast this. We note that this rerandomization step was not included in the semi-honest protocol of [BLO16], but without it we cannot prove security of the preprocessing. Opening the garbled circuit in this way 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, as it would only lead to an abort. This significantly strengthens the result from [LPSY15], which only allowed corrupt parties to provide incorrect PRF values, and is an important factor that allows our preprocessing protocol to be so efficient.

We remark that the first, more general variant is still interesting not only due to a hypothetical future potential, but also because of some aspects of the BMR garbling that we do not currently exploit. One of these aspects is the fact that all of the products between bits are performed in parallel, and hence we could implement them with an MPC protocol optimized for SIMD binary circuits such as MiniMAC [DZ13].[2]

1.2.1 Concurrent Work

A recent work by Katz et al. [KRW17a] introduced a constant round, two-party protocol with a preprocessing phase that can be instantiated based on two-party TinyOT [NNOB12], or the [IPS08] compiler for better asymptotic efficiency. Our protocol is conceptually quite similar, since both involve generating a distributed garbled circuit using TinyOT. However, our approach naturally generalizes to multiple parties because we build on BMR, and this involves substantially different techniques to their Yao-based garbling. Moreover, our general protocol in Section 3 can also be based on [IPS08] to give greater asymptotic efficiency, and this in fact requires fewer multiplications than the method from [KRW17a] (which uses IPS to evaluate the TinyOT functionality). We note that the method of bit/string multiplication in our optimized protocol in Section 4 was inspired by [KRW17a], which also uses TinyOT MACs to produce shares of bit/string products. The rest of our work is independent.

A more recent follow-up work by the same authors [KRW17b] extends [KRW17a] to the multiparty setting, with similar results to ours. Their techniques appear different to ours, but we have not yet analysed this approach in detail and compared with our protocol.

[2]The only known preprocessing methods for MiniMAC are not currently competitive with TinyOT, however [FKOS15].

2 Preliminaries

We denote the 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 $\boldsymbol{c}$.

For 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 $x_i \cdot \boldsymbol{y}$. We use the following property.

Fact 2.1 *If $\boldsymbol{x} \in \mathbb{F}_2^n, \boldsymbol{y} \in \mathbb{F}_2^m$ and $\mathbf{M} \in \mathbb{F}_2^{m \times n}$ then*

$$\mathbf{M} \cdot (\boldsymbol{x} \otimes \boldsymbol{y}) = (\mathbf{M} \cdot \boldsymbol{x}) \otimes \boldsymbol{y}.$$

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 PPT machine $\mathcal{D}$ 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 Pseudorandom Functions

The BMR garbling technique from [LPSY15] is proven secure based on a pseudorandom function (PRF) with multiple keys, defined below.

Definition 2.2 *Let $F : \{0,1\}^n \times \{0,1\}^n \mapsto \{0,1\}^n$ be an efficient, length preserving, keyed function. F is a* pseudorandom function under multiple keys *if for all polynomial-time distinguishers $\mathcal{D}$, there exists a negligible function* negl *such that:*

$$\left|\Pr[\mathcal{D}^{F_{\bar{k}}(\cdot)}(1^\kappa) = 1] - \Pr[\mathcal{D}^{\bar{f}(\cdot)}(1^\kappa) = 1]\right| \leq \mathsf{negl}(\kappa).$$

where $F_{\bar{k}} = F_{k_1}, \ldots, F_{k_{m(n)}}$ are the pseudorandom function F keyed with polynomial number of randomly chosen keys $k_1, \ldots, k_{m(n)}$ and $\bar{f} = f_1, \ldots, f_{m(n)}$ are $m(n)$ random functions from $\{0,1\}^n \mapsto \{0,1\}^n$. The probability in both cases is taken over the randomness of $\mathcal{D}$ as well.

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.

2.2 Circular 2-Correlation Robust PRF

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 2.3 (Circular 2-correlation robust PRF) *A PRF F is* circular 2-correlation robust *if for any non-uniform polynomial-time 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 $\mathcal{O}(k_1, k_1, g, j, b_1, b_2, 0)$ and $\mathcal{O}(k_1, k_1, g, j, b_1, b_2, 1)$ 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.

2.3 Almost-1-Universal Linear Hashing

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

Definition 2.4 (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 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.

2.4 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 2.5 *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:*

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

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.

The 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: Ideal commitments

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

Upon receiving $(\texttt{rand}, S)$ from all parties, where S is any efficiently sampleable set, it samples $r \leftarrow S$ and outputs r to all parties.

Figure 2: Coin-tossing functionality

2.5 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 $r \leftarrow \{0,1\}^\kappa$.

2.6 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.

2.7 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

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

The functionality runs with parties $P_1, \dots, P_n$ and an adversary $\mathcal{A}$. The functionality maintains a dictionary, $\mathtt{Val} \leftarrow \{\}$, to keep track of values in $\mathbb{F}_2$.

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

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

Multiply: On input $(\mathtt{multiply}, \overline{\mathtt{id}}, \mathtt{id}_1, \mathtt{id}_2)$ from all parties, where $(\mathtt{id}_1, \mathtt{id}_2)$ are keys in $\mathtt{Val}$, compute $y \leftarrow \mathtt{Val}[\mathtt{id}_1] \cdot \mathtt{Val}[\mathtt{id}_2]$. 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 $\mathtt{Val}[\overline{\mathtt{id}}] \leftarrow y$.

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

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

2.8 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 $\mathcal{F}_{\mathrm{BitMPC}}$ in Figure 4. This functionality allows parties to provide private inputs, which are then stored and can be added or multiplied internally by $\mathcal{F}_{\mathrm{BitMPC}}$, 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 $\mathcal{F}_{\mathrm{BitMPC}}$, 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 $\mathcal{F}_{\mathrm{BitMPC}}$, respectively, and return the identifier associated with the result.

In Appendix A we present a complete description of a protocol for implementing $\mathcal{F}_{\mathrm{BitMPC}}$. This is done by combining the multiplication triple generation protocol (over $\mathbb{F}_2$) from [FKOS15] with the method for providing inputs from [KOS16].

3 Generating the Garbled Circuit

3.1 BMR Garbling

The [BMR90] garbling technique by Beaver, Micali and Rogaway 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. The main task of generating the garbled circuit while supporting this property is to compute, for each AND gate g with input wires u, v and output wire w, the $4n$ values:

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

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.

This process involves garbling all the gates in parallel while maintaining consistency. More formally, our goal is to implement the $\mathcal{F}_{\text{Prepocessing}}$ functionality shown in Figure 5. For doing that, we need a way to multiply bits with bits and bits with strings securely. More concretely, these tasks are captured by functionalities $\mathcal{F}_{\text{BitMPC}}$ and $\mathcal{F}_{\text{COT}}$, respectively. A consistency check, implemented in $\Pi_{\text{Bit}\times\text{String}}$, ties together these two separate actively secure multiplications.

3.2 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 an additive error, chosen adaptively after seeing the garbled circuit, that is added by the functionality to each entry when the garbled circuit is opened.

3.3 Protocol Overview

Our preprocessing 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. Concretely, each party P_i samples 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:

$$\lambda_u \cdot R^j, \quad \lambda_v \cdot R^j, \quad \lambda_{uvw} \cdot R^j \in \mathbb{F}_2^\kappa \tag{3}$$

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 (3), 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 3.4 for details.

Using shares of the bit/string products, the parties can locally compute an unauthenticated additive share of the entire garbled circuit (steps 3d–4). First, for each of the four values $(a,b) \in \{0,1\}^2$, each party $P_i, i \neq j$ 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 $P_1, \dots, P_n$ and an adversary $\mathcal{A}$.

Garbling: On input $(\texttt{Garbling}, C_f)$ from all parties, where C_f is a boolean circuit, denote by W its set of wires and by G its set of AND gates. The functionality is defined 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 is an input wire:
    1. Sample $\lambda_w \leftarrow \{0,1\}$. If P_j, the party who provides input on that wire in the online phase, is corrupt, instead receive λ_w from $\mathcal{A}$.
    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 $i \in I$. Define $k^i_{w,1} = k^i_{w,0} \oplus R^i$ for all $i \in [n]$.
  - If w is the output of an AND gate:
    1. The functionality chooses and stores a random wire mask $\lambda_w \leftarrow \{0,1\}$.
    2. The functionality samples a key $k^j_{w,0} \leftarrow \{0,1\}^\kappa$, for each $j \notin I$, and receives corrupt parties' keys $k^i_{w,0}$ from $\mathcal{A}$, for $i \in I$. It sets $k^i_{w,1} = k^i_{w,0} \oplus R^i$, for $i \in [n]$.
  - If w is the output of a XOR gate, and u and v its input wires:
    1. The functionality stores $\lambda_w = \lambda_u \oplus \lambda_v$.
    2. For $i \in [n]$, the functionality 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 every AND gate $g \in G$, the functionality computes the $4n$ entries of the garbled version of g as:

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

  Set $\tilde{\mathbf{g}}_{a,b} = \tilde{g}^1_{a,b} \circ \ldots \circ \tilde{g}^n_{a,b} \quad (a,b) \in \{0,1\}^2$. 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, 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. $\langle\cdot\rangle^i$ denotes the i-th share of an authenticated bit and $(\cdot)^i$ the i-th share of a string.
Let $F : \{0,1\}^\kappa \times \{0,1\}^\kappa \times \{0,1\}^\kappa \to \{0,1\}^\kappa$ be a circular 2-correlation robust PRF.

Garbling:

1. Each party P_i samples a random key offset $R^i \leftarrow \mathbb{F}_{2^\kappa}$. Every ordered pair of parties (P_i, P_j) then calls **Initialize** on $\mathcal{F}_{\text{COT}}$, where P_i, the sender, inputs the correlation R^i.
2. **Generate wire masks and keys:** Passing through the wires of the circuit topologically, proceed as follows:
   - If w is a *circuit-input* wire, and P_j is the party whose input is associated with it:
     (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 $\mathcal{F}_{\text{BitMPC}}$.
   (b) Each P_i calls **Input** on $\mathcal{F}_{\text{BitMPC}}$ with randomly sampled bits $\hat{x}_1^i, \dots, \hat{x}_s^i$. For $\ell \in [s]$, the parties compute secret-shared mask $\langle\hat{x}_\ell\rangle = \sum_{i\in[n]}\langle\hat{x}_\ell^i\rangle$.
   (c) For every $j \in [n]$, the parties run the subprotocol $\Pi_{\text{Bit}\times\text{String}}$, where P_j inputs R^j and everyone inputs the $3|G| + s$ shared bits:
   $$(\langle\lambda_u\rangle, \langle\lambda_v\rangle, \langle\lambda_{uv}\rangle + \langle\lambda_w\rangle)_{(u,v,w)} \quad \text{and} \quad (\langle\hat{x}_1\rangle, \dots, \langle\hat{x}_s\rangle).$$
   where the (u, v, w) indices are taken over the input/output wires of each AND gate $g \in G$.
   (d) For each AND gate g, party P_i obtains from $\Pi_{\text{Bit}\times\text{String}}$ an additive share of the $3n$ values (each defined as one row of the matrix $\mathbf{Z}_j$ in this subprotocol):
   $$\lambda_u \cdot R^j, \quad \lambda_v \cdot R^j, \quad \lambda_{uvw} \cdot R^j, \qquad \text{for } j \in [n]$$
   where $\lambda_{uvw} := \lambda_{uv} + \lambda_w$. Each P_i then uses these to compute a share of
   $$\rho_{j,a,b} = \lambda_u \cdot R^j \oplus a \cdot \lambda_v \cdot R^j \oplus b \cdot \lambda_{uvw} \cdot R^j \oplus a \cdot b \cdot R^j$$
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, 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

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 additive shares of zero, $S^i_j \leftarrow \{0,1\}^{4n\kappa|G|}$, $j \in [n]$, such that $\bigoplus_j S^i_j = 0$. P_i sends S^i_j to P_j over a private channel, for $j \neq i$.[a]
2. Each P_j broadcasts $\tilde{C}^j \oplus \bigoplus_{i=1}^n S^i_j$, and XORs these together to reconstruct the garbled circuit, $\tilde{C}$.

[a]This step is independent of $\tilde{C}^i$, so can be merged with a previous round 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 $\rho_{j,a,b}$.

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.[3] Note that this extra rerandomization step needs $O(n^3)$ communication, whereas the rest of the protocol can be done in $O(n^2)$; in Section 4.4 we show how to reduce the cost of the entire preprocessing to $O(n^2)$ using random oracles.

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

3.4 Bit/String Multiplications

Our method for this is in the subrotocol $\Pi_{\text{Bit}\times\text{String}}$ (Figure 8). It proceeds in two stages: first the **Multiply** step creates the shared products, 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 $(a,b) \in \{0,1\}^2$:

$$R^j \cdot ((\lambda_u \oplus a) \cdot (\lambda_v \oplus b) \oplus \lambda_w) \tag{4}$$

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

[3]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.

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

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 $x \in \{\lambda_u, \lambda_v, \lambda_{uv}\}$.

Our main concern with this method is that the adversary may input an incorrect share $\hat{x}^i = x^i \oplus 1$ when computing $Z = x \cdot R^j$, which would result in flipping $\rho_{j,a,b}$ by R^j to use the wrong key, giving an incorrect garbling. Once this consistency between $\langle x \rangle$ and the value used to compute Z is ensured, the adversary can only add an additive error e to her share of the *product* $x \cdot R^j$ rather than to the share of one of the multiplicands. Only with probability $2^{-\kappa}$ (if the adversary guesses the value of R^j and sets $e = R^j$) this will flip from the correct key $k^j_{w,c}$ of P_j to $k^j_{w,c\oplus 1}$. In all other cases, P_j will detect this cheating during the online phase, when she does not see one of her two valid keys as a result of the output of gate $\tilde{g}$.

3.4.1 Consistency Check

The parties verify that the correct shares of x and correlations R^j were used in the correlated OTs, by opening random linear combinations (over $\mathbb{F}_2$) of all the bit/string products, and checking correctness of the opened results. This is possible because the products are just a linear function of the fixed string R^j. 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 the wire masks to be multiplied, whilst $\hat{\boldsymbol{x}} \in \mathbb{F}_2^s$ is a vector of additional, random masking bits, which are used as a one-time pad to ensure that $\boldsymbol{c}_x$ does not leak information on $\boldsymbol{x}$.

To check correctness of a single shared matrix $\mathbf{Z}_j$ (as in Figure 8), each party P_i, for $i \neq j$, then commits to $\mathbf{H} \cdot \mathbf{Z}^i_j$, whilst P_j commits to $\mathbf{H} \cdot \mathbf{Z}^i_j + \boldsymbol{c}_x \otimes R^j$. 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 2.4). 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 $\mathcal{F}_{\text{COT}}$.

Bit/string multiplication subprotocol – $\Pi_{\mathrm{Bit}\times\mathrm{String}}$

Inputs: Each P_j inputs the private global difference string $R^j \in \mathbb{F}_2^\kappa$, which was generated in the main protocol. All parties input $3|G|$ authenticated, additively shared bits, $\langle x_1\rangle, \dots, \langle x_{3|G|}\rangle$, and s additional, random shared bits, $\langle \hat{x}_1\rangle, \dots, \langle \hat{x}_s\rangle$, to be used as masking values and discarded.

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 the sender, and P_i inputs the choice bits $\boldsymbol{x}^i = (x_1^i, \dots, x_{3|G|}^i, \hat{x}_1^i, \dots, \hat{x}_s^i)$.

   Store the $3|G| + s$ sets of outputs from the $\mathcal{F}_{\mathrm{COT}}$ instance with P_i and P_j into the rows of the matrices $\mathbf{T}_{i,j}$ and $\mathbf{Q}_{i,j}$, satisfying:

$$\mathbf{T}_{i,j} = \mathbf{Q}_{i,j} + \boldsymbol{x}^i \otimes R^j \in \mathbb{F}_2^{(3|G|+s)\times\kappa}.$$

2. Each P_i, for $i \neq j$, defines the matrix $\mathbf{Z}_j^i = \mathbf{T}_{i,j}$, and P_j defines $\mathbf{Z}_j^j = \sum_{i\neq j} \mathbf{Q}_{i,j} + \boldsymbol{x}^j \otimes R^j$.

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

Consistency Check: After multiplying, the parties check correctness of the above as follows:

1. Each P_i removes the last s rows from $\mathbf{Z}_j^i$ (for $j \in [n]$) and places these 'dummy' masking values in a matrix $\hat{\mathbf{Z}}_j^i \in \mathbb{F}_2^{s\times\kappa}$. Similarly, redefine $\boldsymbol{x}^i = (x_1^i, \dots, x_{3|G|}^i)$ and let $\hat{\boldsymbol{x}}^i = (\hat{x_1}^i, \dots, \hat{x_s}^i)$.
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, $\mathbf{H} \in \mathbb{F}_2^{s\times 3|G|}$.
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:

$$\mathbf{C}_j^i = \mathbf{H} \cdot \mathbf{Z}_j^i + \hat{\mathbf{Z}}_j^i, \quad \text{for } j \neq i, \text{ and} \quad \mathbf{C}_i^i = \mathbf{H} \cdot \mathbf{Z}_i^i + \hat{\mathbf{Z}}_i^i + \boldsymbol{c}_x \otimes R^i.$$

5. All parties open their commitments and check that, for each $j \in [n]$:

$$\sum_{i=1}^n \mathbf{C}_j^i = 0.$$

   If the check fails, the parties abort.
6. Each party P_i stores the matrices $\mathbf{Z}_1^i, \dots, \mathbf{Z}_n^i$.

Figure 8: Subprotocol for bit/string multiplication and checking consistency

Note that in both of these cases, we are only concerned when the other party in the $\mathcal{F}_{\mathrm{COT}}$ execution is honest, as if both parties are corrupt then $\mathcal{F}_{\mathrm{COT}}$ does not need to be simulated in the security proof.

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 $i \notin I$):

$$\begin{aligned}\Delta^{j,i} &= R^{j,i} + R^j \\ \delta_\ell^{j,i} &= x_\ell^{j,i} + x_\ell^j, \quad \ell \in [3|G|].\end{aligned}$$

Note that $\Delta^{j,i}$ is fixed in the initialization of $\mathcal{F}_{\mathrm{COT}}$, 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 $\mathcal{F}_{\mathrm{COT}}$ 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 $\Pi_{\mathrm{Bit}\times\mathrm{String}}$, we can put the $\mathcal{F}_{\mathrm{COT}}$ outputs for each party into the rows of a matrix, and express the above as:

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

where $\boldsymbol{\delta}^{j,i} = (\delta_1^{j,i}, \ldots, \delta_{3|G|}^{j,i})$, and the tensor product notation is defined in Section 2.

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

$$\mathbf{Z}_j = \sum_{i=1}^n \mathbf{Z}_j^i = \boldsymbol{x} \otimes R^j + R^j \cdot \underbrace{\sum_{i \in I} \boldsymbol{\delta}^{i,j}}_{=\boldsymbol{\delta}^j} + \sum_{i \notin I} x_i \cdot \Delta^{j,i}. \tag{5}$$

We now prove the following.

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

Proof: From (5), 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:

$$\begin{aligned}
0 &= \sum_{i\notin I}\mathbf{C}^i + \sum_{i\in I}\tilde{\mathbf{C}}^i + \mathbf{D}_I \\
\Leftrightarrow \mathbf{D}_I &= \sum_{i\notin I}\mathbf{C}^i + \sum_{i\in I}\tilde{\mathbf{C}}^i \\
&= \mathbf{H}(\sum_{i=1}^{n}\mathbf{Z}^i) + \sum_{i=1}^{n}\hat{\mathbf{Z}}^i + \boldsymbol{c}_x \otimes R^j \\
&= \mathbf{H}(\boldsymbol{x}\otimes R^j + \boldsymbol{\delta}^j \otimes R^j + \sum_{i\notin I}\boldsymbol{x}^i \otimes \Delta^{j,i}) + \hat{\mathbf{Z}} + \mathbf{H}(\boldsymbol{x})\otimes R^j + \hat{\boldsymbol{x}}\otimes R^j \qquad (6)\\
&= \mathbf{H}(\boldsymbol{\delta}^j \otimes R^j + \sum_{i\notin I}\boldsymbol{x}^i \otimes \Delta^{j,i}) + \hat{\mathbf{Z}} + \hat{\boldsymbol{x}}\otimes R^j \qquad (7)
\end{aligned}$$

where (6) holds because $\boldsymbol{c}_x = \mathbf{H}(\boldsymbol{x}) + \hat{\boldsymbol{x}}$, and (7) due to linearity of $\mathbf{H}$ and Fact 2.1.

Now, taking into account the fact that $\hat{\mathbf{Z}}$ is constructed the same way as $\mathbf{Z}$, and considering (5), 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 (7), the check passes if and only if:

$$\begin{aligned}
\mathbf{D}_I &= \mathbf{H}(\boldsymbol{\delta}^j \otimes R^j + \sum_{i\notin I}\boldsymbol{x}^i \otimes \Delta^{j,i}) + \hat{\boldsymbol{\delta}}^j \otimes R^j + \sum_{i\notin I}\hat{\boldsymbol{x}}^i \otimes \Delta^{j,i} \\
&= (\mathbf{H}(\boldsymbol{\delta}^j) + \hat{\boldsymbol{\delta}}^j)\otimes R^j + \sum_{i\notin I}(\mathbf{H}(\boldsymbol{x}^i) + \hat{\boldsymbol{x}}^i)\otimes \Delta^{j,i}. \qquad (8)
\end{aligned}$$

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

First consider the left-hand summation in (8), 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 $\mathcal{F}_{\mathrm{COT}}$, 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 2.4), it holds that

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

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 $\boldsymbol{\delta}^j$ is non-zero then the check passes with probability at most $\varepsilon + 2^{-\kappa}$.

Now suppose that $\boldsymbol{\delta}^j = \hat{\boldsymbol{\delta}}^j = 0$, so $\mathcal{A}$ only needs to guess the right-hand summation of (8) 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 uniformly random in the view of the adversary, this can only occur with probability 2^{-s}.

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 $\max(2^{-s}, \varepsilon + 2^{-\kappa})$. 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 uniformly random*. 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 uniformly random. 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 computational argument, 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 is missing from the protocol of [BLO16], which does not currently have a security proof. 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. 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, malicious 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 malicious, 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 $\mathcal{F}_{\text{COT}}.\texttt{Initialize}$ 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, $\mathcal{S}$, 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, 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 $\mathcal{F}_{\text{BitMPC}}.\texttt{Input}$ 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 $\mathcal{F}_{\text{BitMPC}}.\texttt{Multiply}$ 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 $\mathcal{F}_{\text{COT}}.\texttt{Extend}$ 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: In the real world, the first set of shares in the **Open Garbling** stage are uniformly random in the view of $\mathcal{Z}$, since $\mathcal{Z}$ sees at most $n-1$ of each set of n random shares. Similarly, the rerandomized shares sent in the final step are uniformly random, subject to summing up to the garbled circuit $\tilde{C}$, since $\mathcal{Z}$ never sees the resharings sent between honest parties. Therefore, the simulation of this stage is perfect, as $\mathcal{S}$ uses the correct garbled circuit output by $\mathcal{F}_{\mathrm{Prepocessing}}$, and the errors are defined according to the correct shares that should have been sent by $\mathcal{A}$. ∎

4 More Efficient Garbling with Multi-Party TinyOT

In this section we show how to optimize the general preprocessing protocol from the previous section, by replacing the generic $\mathcal{F}_{\mathrm{BitMPC}}$ functionality with a more specialized one. By exploiting specific features of 'TinyOT'-style protocols based on information-theoretic MACs, we completely avoid having to perform the bit/string multiplications and consistency checks in $\Pi_{\mathrm{Bit}\times\mathrm{String}}$. In addition, we show how to optimize the communication cost of the rerandomization step, in the random oracle model. 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 MAC-based protocol.

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. If the value x is not clear from context, we sometimes write $M_j^i(x)$ and $K_i^j(x)$ to denote the MAC or key on x, held by P_i.

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 $j \in [n]$:

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

■

We define addition of two shared values $[x], [y]$, to be straightforward addition of the components. 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: $(x^i, (M_1^i, K_1^i + c \cdot R^i), \{M_j^i, K_j^i\}_{j \in [n] \setminus \{1,i\}})$, 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 16.

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

Initialize: On receiving (`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.

Bit: On receiving $(\texttt{bit}, m)$ from all parties:

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

Triple: On receiving $(\texttt{triples}, m)$ from all parties:

1. Sample $a_i, b_i \leftarrow \mathbb{F}_2$ and compute $c_i = a_i \cdot b_i$, for $i \in [m]$.
2. For each $x \in \{a_i, b_i, c_i\}_{i \in [m]}$, 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 $\sum_i x^i = x$.
   (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 Preprocessing 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 similar to $\Pi_{\text{Preprocessing}}$ in Figure 6, so we only describe the differences, which are as follows.

1. The random key offset R^i is defined 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 shared mask $[\lambda_w]$ is obtained using the **Bit** command of $\mathcal{F}_{\text{n-TinyOT}}$, instead of using $\mathcal{F}_{\text{BitMPC}}$.

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

3. Every call to **Multiply** on $\mathcal{F}_{\text{BitMPC}}$, with inputs $\langle \lambda_u \rangle, \langle \lambda_v \rangle$, is replaced by a call to the subprotocol Π_{Mult} in Figure 16, with inputs $[\lambda_u], [\lambda_v]$.
4. Steps 3b–3d, which perform the bit/string multiplications, are replaced with the following:
   - For each AND gate $g \in G$ with wires (u, v, w), each party P_i uses their MACs of the shared values $[\lambda_u], [\lambda_v], [\lambda_{uv} + \lambda_w]$, to define, for each $j \in [n]$, the shares:
   $$M_j^i(\lambda_u),\ K_i^j(\lambda_u) \quad M_j^i(\lambda_v),\ K_i^j(\lambda_v) \quad M_j^i(\lambda_{uv} + \lambda_w),\ K_i^j(\lambda_{uv} + \lambda_w)$$
   which are shares of the bit/string products. These are then used to define shares of the garbled circuit.
5. The calls to **Open** on $\mathcal{F}_{\text{BitMPC}}$, to reveal the output wire masks λ_w, are replaced with a call to the subprotocol Π_{Open} in Figure 12. In addition, for every *circuit input wire* w for party P_i, at this point the parties run Π_{Open}^i (Figure 13) to open λ_w to P_i.

Apart from the bit/string products, another difference between this protocol and $\Pi_{\text{Preprocessing}}$ is that the parties do not choose their own wire masks for their input wires, but instead learn them via the private opening protocol, Π_{Open}^i. This change is not strictly 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.

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 standard techniques can be shown similarly to previous works, so we omit the proof here. 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 a **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 $\mathcal{F}_{\text{Prepocessing}}^{\text{KQ}}$.

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 $\mathcal{F}^{\mathrm{KQ}}_{\mathrm{Prepocessing}}$ from Figure 5 in the presence of a static, malicious adversary corrupting up to $n-1$ parties in the $\mathcal{F}_{\mathrm{n\text{-}TinyOT}}$-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.

4.4 Optimizing the Rerandomization Step

Asymptotically, the most expensive part of the preprocessing stage is generating the random shares of zero in the rerandomization step, because this requires $O(n^3)$ communication. We can reduce this to $O(n^2)$, in the random oracle model, by replacing **Open Garbling** with the following:

- Let $H : \{0,1\}^\kappa \to \{0,1\}^{4n\kappa|G|}$ be a random oracle.
- 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, for $j \neq i$.
- P_i computes the shares $S^i_i = \bigoplus_{i \neq j} H(s^i_j)$, and $S^j_i = H(s^j_i)$, for $j \neq i$.
- Each P_i, for $i = 2, \ldots, n$, sends $\tilde{C}^i \oplus \bigoplus_{j=1}^n S^j_i$ to P_1.
- P_1 reconstructs the garbled circuit, $\tilde{C}$, and broadcasts this.

Since every party no longer needs to broadcast an entire garbled circuit share, this reduces the overall cost of the preprocessing stage to $O(n^2)$.

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 $\mathcal{F}_{\mathrm{Prepocessing}}$-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 $\mathcal{F}_{\mathrm{Prepocessing}}$. We conclude with the following theorem.

Theorem 5.1 *Let f be an n-party functionality $\{0,1\}^{n\kappa} \mapsto \{0,1\}^\kappa$ and assume that F is a PRF. Then Protocol Π_{MPC} from Figure 11, UC-securely computes f in the presence of a static malicious adversary corrupting up to $n-1$ parties in the $\mathcal{F}_{\mathrm{Prepocessing}}$-hybrid.*

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 slightly different than the proof in [LPSY15] as in our case the adversary may choose its additive error as a function of the garbled circuit). Then condition on the event flip not occurring, we prove that the two executions are computationally indistinguishable via a

The MPC Protocol - Π_{MPC}

On input a circuit C_f representing the function f and $\rho = (\rho_1, \dots, \rho_n)$ where ρ_i is party's P_i input, the parties execute the following commands in sequence.

Preprocessing: This sub-task is performed as follows.

- Call **Garbling** on $\mathcal{F}_{\text{Prepocessing}}$ with input C_f.
- Each party P_i obtains the λ_w wire masks for every output wire and every wire associated with their input, and all the keys $\{k^i_{w,0}\}_{w \in W}$ and R^i.

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

- For all his input wires w, each party computes $\Lambda_w = \rho_w \oplus \lambda_w$, where ρ_w is that party's input to C_f, and λ_w was obtained in the preprocessing stage. Then, the public value Λ_w is broadcast to all parties.
- For all his input wires w, party P_i broadcasts the key k^i_w associated to Λ_w.
- The parties call **Open Garbling** on $\mathcal{F}_{\text{Prepocessing}}$ to reconstruct $\tilde{g}^j_{a,b}$ for every gate g and values a, b.
- 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 a and b be the respective public values on the input wires.
  1. If g is a XOR gate, set the public value on the output wire to be $c = a + b$. In addition, for every $j \in [n]$, each party computes $k^j_{w,c} = k^j_{u,a} \oplus k^j_{v,b}$.
  2. If g is an AND gate , then each party computes, for all $j \in [n]$:
  $$k^j_{w,c} = \tilde{g}^j_{a,b} \oplus \left(\bigoplus_{i=1}^{n} F^2_{k^i_{u,a}, k^i_{v,b}}(g \| j) \right)$$
  3. If $k^i_{w,c} \notin \{k^i_{w,0}, k^i_{w,1} = k^i_{w,0} \oplus R^i\}$, then P_i outputs `abort`. Otherwise, it proceeds. If P_i aborts it notifies all other parties with that information. If P_i is notified that another party has aborted it aborts as well.
  4. If $k^i_{w,c} = k^i_{w,0}$ then P_i sets $c = 0$; if $k^i_{w,c} = k^i_{w,1}$ then P_i sets $c = 1$.
  5. The output of the gate is defined to be $(k^1_{w,c}, \dots, k^n_{w,c})$ and the public value c.
- Assuming no party aborts, everyone will obtain a public value c_w for every circuit-output wire w. The party can then recover the actual output value from $\rho_w = c_w \oplus \lambda_w$, where λ_w was obtained in the preprocessing stage.

Figure 11: The MPC Protocol - Π_{MPC}

reduction to the correlated robustness 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 $I \subset [n]$, then we prove that there exists PPT simulator $\mathcal{S}$ with access to an ideal functionality $\mathcal{F}$ that implements f, that simulates the adversary's view. Denoting the set of honest parties by $\bar{I}$, our simulator $\mathcal{S}$ is defined below.

The description of the simulation.

1. INITIALIZATION. Upon receiving the adversary's input $(1^\kappa, I, \vec{x}_I)$, $\mathcal{S}$ incorporates $\mathcal{A}$ and internally emulates an execution of the honest parties running Π_{MPC} with the adversary $\mathcal{A}$:

2. PROCESSING. $\mathcal{S}$ emulates the preprocessing phase of functionality $\mathcal{F}_{\text{Prepocessing}}$, obtaining the adversary 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.

3. GARBLING. Let W' denote the set of input wires that are associated with the adversary's input. Then upon receiving the input $(\texttt{Garbling}, C_f)$ from the adversary the simulator emulates the garbling phase as follows.

   - Upon receiving the global differences $\{R^i\}_{i \in I}$ from the adversary the simulator records this set.
   - For every input wire $w \in W'$ that is associated to the adversary's input, the simulator obtains from the adversary a random masking value $\lambda_w \in \{0,1\}$ and an input key $k^i_{w,0} \in \{0,1\}^\kappa$.
   - For every wire $w \in W$ that is the output of an AND gate and $i \in I$, the simulator chooses a random $\lambda_w \in \{0,1\}$ and records it. Moreover, the simulator obtains from the adversary a key $k^i_{w,0}$ and records the pair $(k^i_{w,0}, k^i_{w,1} = k^i_{w,0} \oplus R^i)$.
   - Upon receiving an `OK` command from the adversary, the simulator forwards it a random masking value $\lambda_w \in \{0,1\}$, for every output wire $w \in W$.

4. ONLINE COMPUTATION. In the online computation the simulator honestly generates the public values $\{\Lambda_w\}_{w \in W''}$ and the input keys $\{k^i_{w,\Lambda_w}\}_{i \in \bar{I}, w \in W''}$ that are associated with the honest parties' input wire set W'', and broadcasts these to the adversary.

   It then obtains the adversary's public values $\{\Lambda_w\}_{w \in W'}$ as well as its input keys $\{\hat{k}^i_w\}_{i \in I, w \in W'}$ (which may be different to the keys received in the garbling phase), and defines the adversary's input as follows.

   - INPUT EXTRACTION. For each input wire $w \in W'$, the simulator computes $\rho_w = \Lambda_w \oplus \lambda_w$ and fixes the adversary's input $\{\vec{x}_I\}$ to be the concatenation of these bits. $\mathcal{S}$ sends this input to the trusted party computing f, receiving the output $\boldsymbol{y} = (y_1, \ldots, y_m)$. Note that $\mathcal{A}$ may still provide inconsistent input keys with ρ which we view as providing incorrect PRF values for these wires.

5. SIMULATED GARBLED CIRCUIT GENERATION. Upon receiving the adversary's $(\texttt{OpenGarbling})$ message on $\mathcal{F}_{\text{Prepocessing}}$ the simulator completes the generation of the garbled circuit as follows.

   - It first generates the honest parties' keys $\{k^i_{w,\Lambda_w}\}_{i \in \bar{I}, w \in W}$ associated with every internal wire $w \in W$ that is an output of an AND gate. Note that for the honest parties the simulator generates a single key per wire.

- Next, the simulator chooses a random $\Lambda_w \leftarrow \{0,1\}$ for the public value on every internal wire $w \in W$ that is an output of an AND gate, except for the circuit output wires. For the t-th output wire, $\mathcal{S}$ defines $\Lambda_w = \lambda_w \oplus y_t$ (recall that the masking values for the output wires are already fixed at this point, so the public values must be consistent with the output $\boldsymbol{y} = y_1, \dots y_m$).
- For every XOR gate with input wires u and 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 $\Lambda_w = \Lambda_u \oplus \Lambda_v$.
- Active path generation. In the next 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. More formally, for every AND gate g that is not an output gate, $\mathcal{S}$ honestly generates the entry in row (Λ_u, Λ_v), where Λ_u (resp. Λ_v) is the public value associated to the left (resp. right) input wire to g. Namely, the simulator computes

$$\tilde{g}^j_{\Lambda_u,\Lambda_v} = \left(\bigoplus_{i=1}^{n} F^2_{k^i_{u,\Lambda_u}, k^i_{v,\Lambda_v}}(g \| j) \right) \oplus k^j_{w,\Lambda_w}$$

fixing $\tilde{\mathbf{g}}_{\Lambda_u,\Lambda_v} = \tilde{g}^1_{\Lambda_u,\Lambda_v} \circ \ldots \circ \tilde{g}^n_{\Lambda_u,\Lambda_v}$. 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.

6. The simulator hands the adversary the complete garbled circuit. In case the adversary aborts, the simulator sends *NO-GO* to the trusted party and aborts. Otherwise, the simulator obtains an additive error $\boldsymbol{e} = \{e^{a,b}_g\}_{a,b \in \{0,1\}, g \in G}$ and computes the modified garbled circuit as $\tilde{\mathbf{g}}_{\Lambda_u,\Lambda_v} + e^{\Lambda_u,\Lambda_v}_g$.

   Next, the simulator evaluates the modified circuit using the input wire keys $\{\hat{k}^i_{w,\Lambda_w}\}_{w \in W', i \in I}$ and $\{k^j_{w,\Lambda_w}\}_{w \in W'', j \in \bar{I}}$ and checks whether the honest parties would have aborted. Namely, for each gate, whether the evaluation reveals the honest parties' keys associated with Λ_w for w the output wire of some AND gate. If there exists a gate for which the evaluation yields the honest parties' keys associated with $\bar{\Lambda}_w$ then the simulator outputs fail and aborts. In case the evaluation reveals an invalid key then the simulator sends *NO-GO* to the trusted party and aborts. Otherwise, the simulator outputs whatever the adversary does.

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 consider an event Flip which occurs whenever there exists a gate g with an output wire w in the adversarial garbled circuit, where the ciphertext in the active gate entry encrypts the inactive key $k^i_{w,\bar{\Lambda}_w}$ for some $i \in \bar{I}$. That is, for Λ_u, Λ_v, the public values associated with the input wires of g, the (Λ_u, Λ_v)th entry of $\tilde{\mathbf{g}}_{a,b} + e^{a,b}_g$ encrypts the key $k^i_{w,\bar{\Lambda}_w}$. Note that this event implies that the adversary causes an honest party P_i to compute an incorrect value, as the bit value being transferred within the output wire w is now flipped with respect to P_i.

To prove that this event occurs with negligible probability at the most, we consider an execution $\widetilde{\mathbf{IDEAL}}$ that 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 $\tilde{\mathcal{S}}$, which is defined identically to $\mathcal{S}$ with the exception that $\tilde{\mathcal{S}}$ further picks the global differences $\{R^i\}_{i\in\bar{I}}$ and $\{k^j_{w,\bar{\Lambda}_w}\}_{j\in\bar{I},w\in W}$ (which are never defined by $\mathcal{S}$). Fix the additive error $\boldsymbol{e} = \{e^{a,b}_g\}_{a,b\in\{0,1\},g\in G}$ that is added to the simulated garbled circuit, and let $\tilde{\mathbf{g}}_{a,b} + e^{a,b}_g$ 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^i.

Lemma 5.1 *The probability that* Flip *occurs in* $\widetilde{\mathbf{IDEAL}}$ *is bounded by* $2^{-\kappa}$.

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. On the other hand, carrying out a successful attack via adding the terms $\boldsymbol{e} = \{e^{a,b}_g\}_{a,b\in\{0,1\},g\in G}$ requires flipping the bit that is transferred through some wire w with respect to all honest parties without getting caught. More specifically, the adversary must be able to replace the active row

$$\left(\bigoplus_{i=1}^{n} F^2_{k^i_{u,\Lambda_u},k^i_{v,\Lambda_v}}(g\|j)\right) \oplus k^j_{w,\Lambda_w}.$$

with the row

$$\left(\bigoplus_{i=1}^{n} F^2_{k^i_{u,\Lambda_u},k^i_{v,\Lambda_v}}(g\|j)\right) \oplus k^j_{w,\bar{\Lambda}_w}.$$

for some $j \in \bar{I}$. 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. $\square$

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:*

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

Proof: We begin by defining a slightly modified real execution $\widetilde{\mathbf{HYB}}$, where the creation of the garbled circuit is moved from the preprocessing stage to the online computation stage, after the parties have broadcast their masked inputs. Note that the garbled circuit is still computed according to $\mathcal{F}_{\text{Prepocessing}}$, 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. It is simple to verify that in the $\mathcal{F}_{\text{Prepocessing}}$-hybrid model, the adversary's views in $\widetilde{\mathbf{HYB}}$ and $\mathbf{HYB}$ are identical.

Our proof of the lemma follows by a reduction to the correlation robustness of the PRF F (cf. Definition 2.3). Assume by contradiction the existence of an environment $\mathcal{Z}$, an adversary $\mathcal{A}$ and a non-negligible function $p(\cdot)$ such that

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

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,

- $\mathcal{D}'$ internally invokes $\mathcal{Z}$ that fixes the honest parties' inputs $\boldsymbol{\rho}$.
- $\mathcal{D}'$ 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 the circuit that is an output wire of an AND gate and $i\in\bar{I}$, it samples a key k^i_w and a public value Λ_w. 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 that is the output of an AND gate g with input wires u and v and output wire w, 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
  $$\left(\bigoplus_{i=1}^{n}F^2_{k^i_{u,\Lambda_u},k^i_{v,\Lambda_v}}(g\|j)\right)\oplus k^j_{w,\Lambda_w}.$$
  - 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:
  $$\left(\bigoplus_{i\neq i_0}F^2_{k^i_{u,a},k^i_{v,b}}(g\|j)\right)\oplus h^j_{a,b}\oplus k^j_{w,\Lambda_w}$$

  - 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. $\square$

Finally, we demonstrate that the probability Flip occurs in $\mathbf{HYB}$ is negligible as well due to indistinguishability of executions. This concludes the proof as it demonstrates that with overwhelming probability the adversary is getting caught whenever cheating in the computation of the PRF values.

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 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 by 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 correlation robust 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 $\tilde{k}^i_w$ that is revealed during the execution with the actual active key k^i_w that was created by $\mathcal{D}$ in the garbling phase. For every gate g for which there exists a difference $\Delta^i_g = \tilde{k}^i_w \oplus k^i_w$ for all $i \in \bar{I}$, $\mathcal{D}$ sets $R^i_g = \Delta^i_g$.

3. For every gate g for which $\mathcal{D}$ recorded a global difference $R^{i_0}_g$ for party i_0, it defines the inactive key for the output wire $w \in W$ of this gate by $k^{i_0}_{w,\bar{\Lambda}_w} = k^{i_0}_{w,\Lambda_w} \oplus R^{i_0}_g$. 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 $(k^{i_0}_{w,\bar{\Lambda}_w}, k^{i_0}_v, g, j, \bar{a} \oplus \lambda_w, b \oplus \lambda_v, \ell_{a,b})$ where $k^{i_0}_v$ is the active key associated with the other input wire of P_{i_0}. $\mathcal{D}$ compares this outcome with the values it obtained from its oracle for the query $(k^{i_0}_{w,\Lambda_w}, k^{i_0}_v, g, j, a \oplus \lambda_w, b \oplus \lambda_v, \ell_{a,b})$. 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 Flip occurs with respect to i_0 than $\mathcal{D}$ can identify this event by extracting some inactive key and querying its oracle in this key. Therefore $\mathcal{D}$ outputs Circ with probability $\frac{1}{q(\kappa)\cdot n}$. On the other hand, due to the claim made in Lemma 5.2, Flip rarely occurs in $\mathbf{IDEAL}$ and thus $\mathcal{D}$ outputs Rand on the event of Flip only with negligible probability. This implies a non-negligible gap with respect to the event occurring in the two executions and concludes the proof. □ ■

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, denoted by $\mathcal{F}^{\mathrm{KQ}}_{\mathrm{Prepocessing}}$ and formally defined in Section 4.3.

Theorem 5.2 *Let f be an n-party functionality $\{0,1\}^{n\kappa} \mapsto \{0,1\}^{\kappa}$ and assume that F is a PRF. Then Protocol Π_{MPC} from Figure 11, UC-securely computes f in the presence of a static malicious 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:*

- $\{\mathbf{HYB}^{\mathcal{F}_{\mathrm{Prepocessing}}}_{\Pi_{\mathrm{MPC}},\mathcal{A}}(1^{\kappa}, z, I, \boldsymbol{\rho})\}_{\kappa\in\mathbb{N},z\in\{0,1\}^*,\boldsymbol{\rho}\in\{0,1\}^{n\kappa}}$

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

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, it outputs Rand. Otherwise, if at the end of the execution no queries have resulted in a correct garbling, output Rand. $\square$

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 in the sense that the reduction additionally needs to reply 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 $\mathcal{D}$ responses with 1 to this query. The rest of the proof follows identically. $\square$ ∎

6 Complexity Analysis

We now focus on analysing the concrete round and 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.

6.1 Cost 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:

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

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	226.86 kB	16.13 kB
	32	8.74 MB	226.86 kB	16.13 kB
	128	7.22 MB	226.86 kB	16.13 kB
	1024	6.42 MB	226.86 kB	16.13 kB
[KRW17a]	1	2.56 MB	464.3 kB	4.1 kB
	32	2.56 MB	464.3 kB	4.1 kB
	128	1.92 MB	464.3 kB	4.1 kB
	1024	1.92 MB	464.3 kB	4.1 kB
Ours + [KRW17a]	1	2.56 MB	872.1 kB	2.06 kB
	32	2.56 MB	872.1 kB	2.06 kB
	128	1.92 MB	872.1 kB	2.06 kB
	1024	1.92 MB	872.1 kB	2.06 kB

Table 2: 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.

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. Using random oracles (Section 4.4), 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 $(n-1)(\kappa+1) \cdot I_i$ 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 [FLNW16].

6.2 Comparison with Two-Party Protocols

In Table 2 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 [KRW17a], lowering the previous costs further. For consistency with the other two-party protocols, we divide the protocol costs into three phases: function-independent preprocessing,

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
Ours	active	14.01 MB	63.22 MB	1.31 MB	4.37 MB

Table 3: Comparison of the cost of our protocol with previous constant-round MPC protocols in a range of security models. Costs are the amount of data sent over the network per party.

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.

We have made a couple of tweaks to our protocol for this comparison. We count the cost of the **Open Garbling** stage in the function-dependent preprocessing, instead of the online phase; this change means that our 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 there are only two parties, our preprocessing protocol can be proven secure without the share rerandomization in the output stage (step 1). This is because an honest party's share of the garbled circuit is anyway uniquely defined by the corrupted party's share, and the garbled circuit itself, so there is no need for it to be randomized.

The online phase of the modified protocol is just one round of interaction, which involves each party sending only $I \cdot (\kappa + 1)$ bits, where I is the number of inputs for that party. This is the lowest online cost of *any* actively secure two-party protocol.[4] 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.

6.3 Comparison with Multi-Party Protocols

In Table 3 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 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 one round, 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

[4]If counting the *total* amount of data sent, in both directions, our costs would be similar to [KRW17a], 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.

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 $O(n^3)$.

Acknowledgements

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. 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.

[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 *ACM CCS 16*, pages 578–590. ACM Press, 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.

[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.

[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.

[FLNW16] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. Cryptology ePrint Archive, Report 2016/944, 2016. http://eprint.iacr.org/2016/944.

[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 *ACM CCS 16*, pages 830–842. ACM Press, 2016.

[KRW17a] Jonathan Katz, Samuel Ranellucci, and Xiao Wang. Authenticated garbling and communication-efficient, constant-round, secure two-party computation. *IACR Cryptology ePrint Archive*, 2017:30, 2017.

[KRW17b] Jonathan Katz, Samuel Ranellucci, and Xiao Wang. Authenticated garbling and efficient maliciously secure multi-party computation. *IACR Cryptology ePrint Archive*, 2017:189, 2017.

[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 *TCC 2016-B, Part I*, LNCS, pages 554–581. Springer, Heidelberg, 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.

[RR16] Peter Rindal and Mike Rosulek. Faster malicious 2-party secure computation with online/offline dual execution. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 297–314, Austin, TX, 2016. USENIX Association.

[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.

A Protocol for GMW-Style MPC for Binary Circuits

Here we describe the full protocol for realizing the $\mathcal{F}_{\text{n-TinyOT}}$ functionality. The protocol is not new, but given for the reader's interest as a complete description has not been previously published. It essentially consists of the bit triple generation protocol from [FKOS15], with some simplifications for clarity, and standard techniques for producing random shared bits and sharing private inputs. We omit the proof, as it is similar to previous works.

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

$$[x^i]_{i,j} = (x^i, M^i_j, K^j_i)$$
$$[x] = (x^i, \{M^i_j, K^i_j\}_{j \neq i})_{i \in [n]}, \quad M^i_j = K^j_i + x^i \cdot R^j$$

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 12: 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 $i \neq j$, 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 13: Subprotocol for private opening to one party

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 12–13.

The main protocol, shown in Figure 14, 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 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 15, 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]):

- 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 Parameters

Based on the analysis from previous works [FKOS15, KRW17a], 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.3 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.

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 $(\texttt{init})$.

Bits: To create m random shared bits:

1. Each party P_i samples $b_1^i, \dots, b_m^i \leftarrow \{0,1\}$.
2. Every ordered pair (P_i, P_j) calls $\mathcal{F}_{\text{COT}}$, where P_i is receiver and inputs $(\texttt{extend}, b_1^i, \dots, b_m^i)$.
3. Use the previous outputs to define sharings $[b_1], \dots, [b_m]$.
4. Run the *Consistency check* on $[b_1], \dots, [b_\ell]$ and output these sharings.

ANDs: 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 $(\texttt{extend}, x_1^i, \dots, x_{m'}^i)$.
3. P_i and P_j obtain their respective value of $[x_\ell^i]_{i,j} = (M_\ell^{i,j}, K_\ell^{j,i})$, such that $M_\ell^{i,j} = K_\ell^{j,i} + x^i \cdot R^j \in \mathbb{F}_2^\kappa$.
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 $w_\ell^{i,j} = \mathsf{H}(M_\ell^{i,j}) + x_\ell^i \cdot d = u_\ell^{j,i} + x_\ell^i \cdot y_\ell^i$
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.

Subprotocol *Consistency check*: To verify the shared values $[\boldsymbol{x}] = ([x_1], \dots, [x_m])$:

1. Each party P_i samples s random bits $r_1^i, \dots, r_s^i \leftarrow \{0,1\}$.
2. Every unordered pair (P_i, P_j) runs $\mathcal{F}_{\text{COT}}$ with command $\texttt{extend}$ to authenticate their bits, hence obtaining $[R] = ([r_1], \dots, [r_s])$.
3. Call $\mathcal{F}_{\text{Rand}}$ to obtain a seed for a ε-almost 1-universal linear hash function, $\mathbf{H} : \mathbb{F}_2^m \to \mathbb{F}_2^s$.
4. Locally compute
$$[\boldsymbol{c}] = \mathbf{H}([\boldsymbol{x}]) + [R].$$
5. Run Π_{Open} on each component of $[\boldsymbol{c}]$ to check correctness of the MACs. If any check fails, abort.

Figure 14: 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 15: Checking corretness 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 16: Subprotocol for multiplying secret shared values using a triple

A.4 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.5 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 $[x] = [b] + d$.