Low Cost Constant Round MPC Combining BMR and Oblivious Transfer

Carmit Hazay[1](✉), Peter Scholl[2], and Eduardo Soria-Vazquez[3]

[1] Bar-Ilan University, Ramat Gan, Israel
carmit.hazay@biu.ac.il
[2] Aarhus University, Aarhus, Denmark
peter.scholl@cs.au.dk
[3] University of Bristol, Bristol, UK
eduardo.soria-vazquez@bristol.ac.uk

Abstract. In this work, we present two new universally composable, actively secure, constant round multi-party protocols for generating BMR garbled circuits with free-XOR and reduced costs.

1. Our first protocol takes a generic approach using any secret-sharing based MPC protocol for binary circuits, and a correlated oblivious transfer functionality.
2. Our specialized protocol uses secret-sharing based MPC with information-theoretic MACs. This approach is less general, but requires no additional correlated OTs to compute the garbled circuit.

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

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

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 [5,13,15,18,22,35], and

P. Scholl—Work done whilst at University of Bristol, UK.

T. Takagi and T. Peyrin (Eds.): ASIACRYPT 2017, Part I, LNCS 10624, pp. 598–628, 2017.
https://doi.org/10.1007/978-3-319-70694-8_21

those based on garbled circuits [2,11,25–28,32,39]. 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 [5,15,18] 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 this approach the parties first secret-share their inputs and then evaluate the circuit gate by gate while preserving privacy and correctness. In low-latency networks, they can have an extremely fast online evaluation stage, but the round complexity makes them much less suited to high-latency networks, when the parties may be far apart.

Garbled circuits, introduced in Yao's protocol [39], are the core behind all practical, constant round protocols for secure computation. In the two-party setting, one of the parties "encrypts" the circuit being evaluated, whereas the other party privately evaluates it. Garbled circuit-based protocols have recently become much more efficient, and currently give the most practical approach for actively secure computation of binary circuits [34,37]. 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 from [2]. 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. A different protocol, but only for three parties, was designed by Choi et al. [11] in the dishonest majority setting. More practical, actively secure protocols for any number of parties are the recent works of Lindell et al. [29,31], which use somewhat homomorphic encryption (SHE) or generic MPC to garble a circuit. Ben-Efraim et al. [4] recently presented and implemented an efficient multi-party garbling protocol based on oblivious transfer, but with only semi-honest security. Very recently, Katz et al. introduced in [23] protocols based on authenticated garbling, with a preprocessing phase that can be instantiated based on TinyOT [33].

1.1 Our Contributions

In this work, we present a practical, actively secure, constant round multi-party protocol for generating BMR garbled circuits with free-XOR in the presence of up to $n-1$ out of n corruptions. As in prior constructions, our approach has two phases: a 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 and incurs no cost to achieve active security, the focus of recent works was on optimizing the preprocessing complexity, where the main bottleneck is with respect to garbling

AND gates. In that context, we present two new constant-round protocols for securely generating the garbled circuit:

1. A generic approach using any secret-sharing based MPC protocol for binary circuits, and a correlated oblivious transfer functionality.
2. A specialized protocol which uses secret-sharing based MPC with information-theoretic MACs, such as TinyOT [17,33]. This approach is less general, but requires no additional correlated OTs to compute the garbled circuit.

In both approaches, the underlying secret-sharing based protocol is only used for *one secure* $\mathbb{F}_2$ *multiplication per AND gate.*

In the first, more general method, every pair of parties needs to run one correlated OT per AND gate, which costs $O(\kappa)$ communication for security parameter κ. Combining this with the overhead induced by the correlated OTs in our protocol, we obtain total complexity $O(|C|\kappa n^2)$, assuming only symmetric primitives and $O(\kappa)$ seed OTs between every pair of parties. This gives an overall communication cost of $O(M + |C|\kappa n^2)$ to evaluate a circuit C, where M is the cost of evaluating $|C|$ AND gates in the secret-sharing based protocol, Π. To realize Π, we can define a functionality with multiplication depth 1 that computes all the AND gates in parallel (these multiplications can be computed in parallel as they are independent of the parties' inputs). Furthermore, the [21] compiler can be instantiated with semi-honest [18] as the inner protocol and [12] as the outer protocol. By Theorem 2, Sect. 5 from [21], for some constant number of parties $m \geq 2$, the functionality can be computed with communication complexity $O(|C|)$ plus low order terms that depend on a statistical parameter s, the circuit's depth and $\log |C|$. As in [21], this extends to the case of a non-constant number of parties n, in which case the communication complexity grows by an additional factor of $|C|\mathsf{poly}(n)$.

Another interesting candidate for instantiating Π would be to use an MPC protocol optimized for SIMD binary circuits such as MiniMAC [16]. This is because in our construction, all the AND gates can be computed in parallel. Currently, the only known preprocessing methods [17] for MiniMAC are not practical, but this seems to be an interesting future direction to explore.

TinyOT is currently the most practical approach to secret-sharing based MPC on binary circuits, so the second method leads to a highly practical protocol for constant-round secure computation. The complexity is essentially the same as TinyOT, as here we do not require any additional OTs. However, the protocol is less general and has worse asymptotic communication complexity, since TinyOT costs either $O(|C|B\kappa n^2)$ (with 2 parties or the recent protocol of [38]), or $O(|C|B^2\kappa n^2)$ (with [17]), where $B = O(1 + s/\log |C|)$ (and in practice is between 3–5), and s is the statistical security parameter.

Our constructions employ 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 [29] so that it obtains a vector of additive errors from the adversary to be applied to 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. This requires a new analysis and proof of the online phase.

Secondly, we devise a new consistency check to enforce correctness of inputs to correlated OT, which is based on very efficient linear operations similar to recent advances in homomorphic commitments [9]. This check, combined with our improved error analysis for the online phase, allows the garbled circuit to be created without authenticating any of the parties' keys or PRF values, which removes a significant cost from previous works (saving a factor of $\Omega(n)$).

Implementation. We demonstrate the practicality of our TinyOT-based protocol with an implementation, and perform experiments with up to 9 parties securely computing the AES and SHA-256 circuits. In a 1 Gbps LAN setting, we can securely compute the AES circuit with 9 parties in just 620 ms. This improves upon the best possible performance that would be attainable using [29] by around 60 times. The details of our implementation can be found in Sect. 6.

Comparison with Other Approaches. Table 1 shows how the communication complexity of our work compares with other actively secure, constant-round protocols. As mentioned earlier, most previous constructions express the garbling function as an arithmetic circuit over a large finite field. In these protocols, garbling even a single AND gate requires computing $O(n)$ multiplications over a large field with SHE or MPC. This means they scale at least cubically in the number of parties. In constrast, our protocol only requires one $\mathbb{F}_2$ multiplication per AND gate, so scales with $O(n^2)$. Previous SHE-based protocols also require zero-knowledge proofs of plaintext knowledge of SHE ciphertexts, which in practice are very costly. Note that the recent MASCOT protocol [24] for secure computation of arithmetic circuits could also be used in [29], instead of SHE, but this still has very high communication costs. We denote by MASCOT-BMR-FX an optimized variant of [29], modified to use free-XOR as in our protocol, with multiplications in $\mathbb{F}_{2^\kappa}$ done using MASCOT. Finally, the recent concurrent work by Katz et al. [23] is based on an optimized variant of TinyOT, with comparable performance to our approach.

None of these previous works have reported implementations at the time of writing, but our implementation of the TinyOT-based protocol improves upon the best times that would be achievable with SPDZ-BMR and MASCOT by up to 60x. This is because our protocol has lower communication costs than [29] (by at least 2 orders of magnitude) and the main computational costs are from standard symmetric primitives, so far cheaper than using SHE.

Overall, our protocols significantly narrow the gap between the cost of constant-round and many-round MPC protocols for binary circuits. More specifically, this implies that, with current techniques, constant round MPC for binary circuits is not much more expensive than practical, non-constant round protocols. Additionally, both of our protocols have potential for future improvement by optimizing existing non-constant round protocols: a practical implementation of MiniMAC [16] would lead to a very efficient approach with our generic protocol, whilst any future improvements to multi-party TinyOT would directly give a similar improvement to our second protocol.

Table 1. Comparison of actively secure, constant round MPC protocols. $B = O(1 + s/\log|C|)$ is a cut-and-choose parameter, which in practice is between 3–5. Our second protocol can also be based upon optimized TinyOT to obtain the same complexity as [23].

Protocol	Based on	Free XOR	Comms. per garbled gate
SPDZ-BMR [29]	SHE + ZKPoPK	✗	$O(n^4\kappa)$
SHE-BMR [31]	SHE (depth 4) + ZKPoPK	✗	$O(n^3\kappa)$
MASCOT-BMR-FX	OT	✓	$O(n^3\kappa^2)$
This work Sect. 3	OT + [21]	✓	$O(n^2\kappa + \mathsf{poly}(n))$
This work Sect. 4	TinyOT	✓	$O(n^2B^2\kappa)$
[23] (concurrent)	Optimized TinyOT	✓	$O(n^2B\kappa)$

1.2 Technical Overview

Our protocol is based on the recent free-XOR variant of BMR garbling used for semi-honest MPC in [4]. 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):

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

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 [29] 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 [29, 31], which obtain *authenticated* (with MACs, or SHE ciphertexts) sharings of the entire garbled circuit. Our protocol greatly reduces this complexity, since the PRF values and keys (on the first line of Eq. (1)) do not need to be authenticated. The main challenge, therefore, is to compute shares of the products on the second line of (1). Similarly to [4], 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.

To compute the bit product $\lambda_u \cdot \lambda_v$, we can use any actively secure GMW-style MPC protocol for binary circuits. 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. In more details,

1. 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 and sacrificing a few OTs, ensuring the correct inputs were provided to the OT. This protocol is presented in Sect. 3.
2. The second method requires using a 'TinyOT'-style protocol [6,17] based on information-theoretic MACs, and allows us to compute the bit/string products directly from the MACs, provided each party's MAC key is chosen to be the same string R^i used in the garbling. This saves interaction since we do not need any additional OTs. This protocol is presented in Sect. 4.

After creating shares of all these products, the parties can compute shares of the whole garbled circuit. These shares must then be rerandomized, before they can be broadcast. 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 [29], which only allowed corrupt parties to provide incorrect PRF values, and is an important factor that allows our preprocessing protocol to be so efficient.

Concurrent Work. Two recent works by Katz, Ranellucci and Wang introduced constant round, two-party [38] and multi-party [23] protocols based on *authenticated garbling*, with a preprocessing phase that can be instantiated based on TinyOT. At the time of writing, their two-party paper also reports on an implementation, but the multi-party version does not. Our work is conceptually quite similar, since both involve generating a garbled circuit in a distributed manner using TinyOT. The main difference seems to be that our protocol is symmetric, since all parties evaluate the same garbled circuit. With authenticated garbling, the garbled circuit is only evaluated by one party. This makes the garbled circuit slightly smaller, since there are $n-1$ sets of keys instead of n, but the online phase requires at least one more round of interaction (if all parties learn the output). The works of Katz et al. also contain concrete and asymptotic improvements to the two-party and multi-party TinyOT protocols, which improves upon the TinyOT protocol we give in the full version of this paper [20] by a factor of $O(s/\log|C|)$, where s is a statistical parameter. These improvements can be directly plugged into our second garbling protocol. We remark that the two-party protocol in [38] inspired our use of TinyOT MACs to perform the bit/string multiplications in our protocol from Sect. 4. The rest of our work is independent.

Another difference is that our protocol from Sect. 3 is more generic, since $\mathcal{F}_{\text{BitMPC}}$ can be implemented with *any* secret-sharing based bit-MPC protocol, rather than just TinyOT. This can be instantiated with [21] to obtain a constant-round protocol with complexity $O(|C|(\kappa n^2 + \mathsf{poly}(n)))$ in the OT-hybrid model. The multi-party paper [23] does not have an analogous generic result.

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 21. *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*

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

Universal composability. We prove security of our protocols in the universal composability (UC) framework [7] (see also [8] for a simplified version of UC).

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 [19]).

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

2.1 Circular 2-Correlation Robust PRF

The BMR garbling technique from [29] is proven secure based on a pseudorandom function (PRF) with multiple keys. However, since our scheme supports free-XOR, we need to adapt the definition of correlation robustness with circularity from [10] 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 21 (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 [10], some trivial queries must be ruled out. Specifically, the distinguisher is restricted as follows: (1) it is not allowed to make any query of the form $\mathcal{O}(k_1, k_2, g, j, 0, 0, b_3)$ (since it can compute $F_{k_1,k_2}(g \| j)$ on its own) and (2) it is not allowed to query both tuples $\mathcal{O}(k_1, k_2, g, j, b_1, b_2, 0)$ and $\mathcal{O}(k_1, k_2, g, j, b_1, b_2, 1)$ for any values k_1, k_2, g, j, b_1, b_2 (since that would allow it to trivially recover the global difference). We say that any distinguisher respecting these restrictions makes legal queries.

2.2 Almost-1-Universal Linear Hashing

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

Definition 22 (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 [9]). 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 [9], 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 [34]), described as follows (here we assume that s divides m, for simplicity):

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

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

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

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

2.3 Commitment Functionality

We require a UC commitment functionality $\mathcal{F}_{\mathrm{Commit}}$ (Fig. 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 $r \leftarrow \{0,1\}^\kappa$.

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.

Fig. 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.

Fig. 2. Coin-tossing functionality

2.4 Coin-Tossing Functionality

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

2.5 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 [33] for secure two-party computation constructs such a protocol, and a significantly optimized version of this is given in [34]. The communication cost is roughly $\kappa + s$ bits per OT. The ideal functionality is shown in Fig. 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 $i \in [m]$.

Fig. 3. Fixed correlation oblivious transfer functionality

2.6 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 Fig. 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.

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

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

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

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

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

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

Fig. 4. Functionality for GMW-style MPC for binary circuits

2.7 BMR Garbling

The [2] 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 [4,29,31], 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:

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

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

3 Generic Protocol for Multi-party Garbling

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

3.1 The Preprocessing Functionality

The preprocessing functionality, formalized in Fig. 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.

3.2 Protocol Overview

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

Concretely, each party P_i starts by sampling a global difference string $R^i \leftarrow \{0,1\}^\kappa$, and for each wire w which is an output wire of an AND gate, or an input wire, P_i also samples the keys $k^i_{w,0}$, $k^i_{w,1} = k^i_{w,0} \oplus R^i$ and an additive share of the wire mask, $\lambda^i_w \leftarrow \mathbb{F}_2$. As in [4], 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 [4]) 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.

The Preprocessing Functionality

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

Garbling: On input $(\texttt{Garbling}, C_f)$ from all parties, where C_f is a boolean circuit, denote by W its set of wires 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. Sample $\lambda_w \leftarrow \{0,1\}$.
    2. Sample a key $k^j_{w,0} \leftarrow \{0,1\}^\kappa$, for each $j \notin I$, and receive corrupt parties' keys $k^i_{w,0}$ from $\mathcal{A}$, for $i \in I$. Set $k^i_{w,1} = k^i_{w,0} \oplus R^i$, for $i \in [n]$.
  - If w is the output of a XOR gate, and u and v its input wires:
    1. Compute and store $\lambda_w = \lambda_u \oplus \lambda_v$.
    2. For $i \in [n]$, set $k^i_{w,0} = k^i_{u,0} \oplus k^i_{v,0}$ and $k^i_{w,1} = k^i_{w,0} \oplus R^i$.
- For every AND gate $g \in G$, the functionality computes the $4n$ entries of the garbled version of g as:

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

  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 $a, b \in \{0,1\}$.

Fig. 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 $F : \{0,1\}^{2\kappa} \times [|G|] \times [n] \to \{0,1\}^\kappa$ be a circular 2-correlation robust PRF, and $\mathcal{G} : \{0,1\}^\kappa \to \{0,1\}^{4n\kappa|G|}$ be a PRG.

Garbling:

1. Each party P_i samples a random key offset $R^i \leftarrow \mathbb{F}_2^\kappa$.
2. **Generate wire masks and keys:** For each wire w in topological order:
   - If w is a *circuit-input* wire from P_j:
     (a) P_j calls **Input** on $\mathcal{F}_{\text{BitMPC}}$ with a randomly sampled $\lambda_w \in \{0,1\}$ to obtain $\langle \lambda_w \rangle$. P_j defines the share $\lambda_w^j = \lambda_w$, every other P_i sets $\lambda_w^i = 0$.
     (b) Every P_i samples a key $k_{w,0}^i \leftarrow \{0,1\}^\kappa$ and sets $k_{w,1}^i = k_{w,0}^i \oplus R^i$.
   - If the wire w is the output of an AND gate:
     (a) Each P_i calls **Input** on $\mathcal{F}_{\text{BitMPC}}$ with a randomly sampled $\lambda_w^i \leftarrow \{0,1\}$. The parties then compute the secret-shared wire mask as $\langle \lambda_w \rangle = \sum_{i \in [n]} \langle \lambda_w^i \rangle$.
     (b) Every P_i samples a key $k_{w,0}^i \leftarrow \{0,1\}^\kappa$ and sets $k_{w,1}^i = k_{w,0}^i \oplus R^i$.
   - If the wire w is the output of a XOR gate:
     (a) The parties compute the mask on the output wire as $\langle \lambda_w \rangle = \langle \lambda_u \rangle + \langle \lambda_v \rangle$.
     (b) Every P_i sets $k_{w,0}^i = k_{u,0}^i \oplus k_{v,0}^i$ and $k_{w,1}^i = k_{w,0}^i \oplus R^i$.
3. **Secure product computations:**
   (a) For each AND gate $g \in G$, the parties compute $\langle \lambda_{uv} \rangle = \langle \lambda_u \rangle \cdot \langle \lambda_v \rangle$ by calling **Multiply** on $\mathcal{F}_{\text{BitMPC}}$.
   (b) Each P_i calls **Input** on $\mathcal{F}_{\text{BitMPC}}$ with randomly sampled bits $\hat{x}_1^i, \ldots, \hat{x}_s^i$. 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:
   $$(\langle \lambda_u \rangle, \langle \lambda_v \rangle, \langle \lambda_{uv} \rangle + \langle \lambda_w \rangle)_{(u,v,w)} \qquad \text{and} \qquad (\langle \hat{x}_1 \rangle, \ldots, \langle \hat{x}_s \rangle).$$
   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
   $$\rho_{j,a,b} = \lambda_{uvw} \cdot R^j \oplus a \cdot \lambda_v \cdot R^j \oplus b \cdot \lambda_u \cdot R^j \oplus a \cdot b \cdot R^j$$
4. **Garble gates:** For each AND gate $g \in G$, each $j \in [n]$, and the four combinations of $a, b \in \{0,1\}^2$, the parties compute shares of the j-th entry of the garbled gate $\tilde{g}_{a,b}$ as follows:
   - P_j sets $(\tilde{g}_{a,b}^j)^j = \rho_{j,a,b}^j \oplus F_{k_{u,a}^j, k_{v,b}^j}(g \| j) \oplus k_{w,0}^j$.
   - For every $i \neq j$, P_i sets $(\tilde{g}_{a,b}^j)^i = \rho_{j,a,b}^i \oplus F_{k_{u,a}^i, k_{v,b}^i}(g \| j)$.
5. **Reveal masks for output wires:** For every circuit-output-wire w, the parties call **Open** on $\mathcal{F}_{\text{BitMPC}}$ to reveal λ_w to all the parties.

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

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

Fig. 7. Open garbling stage of the preprocessing protocol.

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 Sect. 3.3 for details.

Using shares of the bit/string products, the parties can locally compute an unauthenticated additive share of the entire garbled circuit (steps 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

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

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

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

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

3.3 Bit/String Multiplications

Our method for this is in the subrotocol $\mathit{\Pi}_{\text{Bit}\times\text{String}}$ (Fig. 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. Fig. 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:
$$\sum_{i=1}^{n} Z^i = \sum_{i \neq j} T_{i,j} + \sum_{i \neq j} Q_{i,j} + x^j \cdot R^j \quad = \sum_{i \neq j} (T_{i,j} + Q_{i,j}) + x^j \cdot R^j = x \cdot R^j$$
as required.

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

3.4 Consistency Check

We now show how the parties verify that the correct shares of x and correlations R^j were used in the correlated OTs, and analyse the security of this check. The parties first create $m + s$ bit/string products, where m is the number of products needed and s is a statistical security parameter, and then open random

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

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, $\langle x_1 \rangle, \ldots, \langle x_{3|G|} \rangle$, and s additional, random shared bits, $\langle \hat{x}_1 \rangle, \ldots, \langle \hat{x}_s \rangle$, to be used as masking values and discarded.

I: Init: Every ordered pair of parties (P_i, P_j) calls **Initialize** on $\mathcal{F}_{\mathrm{COT}}$, where P_j, the sender, inputs the global difference string R^j.

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

1. For every $i \neq j$, parties P_i and P_j call **Extend** on the $\mathcal{F}_{\mathrm{COT}}$ instance where P_j is sender, and P_i inputs the choice bits $\boldsymbol{x}^i = (x_1^i, \ldots, x_{3|G|}^i, \hat{x}_1^i, \ldots, \hat{x}_s^i)$.
   For each OT between (P_i, P_j), P_j receives $q \in \{0,1\}^{\kappa}$ and P_i receives $t \in \{0,1\}^{\kappa}$. P_i stores their $3|G| + s$ strings from this instance into the rows of a matrix $\mathbf{T}_{i,j}$, and P_j stores the corresponding outputs in $\mathbf{Q}_{i,j}$. These satisfy:
   $$\mathbf{T}_{i,j} = \mathbf{Q}_{i,j} + \boldsymbol{x}^i \otimes R^j \in \mathbb{F}_2^{(3|G|+s)\times\kappa}.$$
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 $j \in [n]$.

III: Consistency Check: 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 $\boldsymbol{x}^i = (x_1^i, \ldots, x_{3|G|}^i)$ and let $\hat{\boldsymbol{x}}^i = (\hat{x_1}^i, \ldots, \hat{x_s}^i)$.
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 $\mathbf{Z}_1^i, \ldots, \mathbf{Z}_n^i$.

Fig. 8. Subprotocol for bit/string multiplication and checking consistency

linear combinations (over $\mathbb{F}_2$) of all the products and check 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 m wire masks to be multiplied, whilst $\hat{\boldsymbol{x}} \in \mathbb{F}_2^s$ are the additional, random masking bits, used as a one-time pad to ensure that $\boldsymbol{c}_x$ does not leak information on $\boldsymbol{x}$.

To verify that a single shared matrix $\mathbf{Z}_j$ is equal to $\boldsymbol{x} \otimes R^j$ (as in Fig. 8), each party P_i, for $i \neq j$, then commits to $\mathbf{H} \cdot \mathbf{Z}_j^i$, whilst P_j commits to $\mathbf{H} \cdot \mathbf{Z}_j^i + \boldsymbol{c}_x \otimes R^j$. 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 22). Furthermore, it turns out that cancelling out any non-zero errors in the check requires either guessing an honest party's global difference R^j, or guessing the secret masking bits $\hat{\boldsymbol{x}}$.

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

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

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

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

$$\Delta^{j,i} = R^{j,i} + R^j$$

$$\delta_\ell^{j,i} = x_\ell^{j,i} + x_\ell^j, \quad \ell \in [3|G|].$$

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

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

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

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

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

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

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

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

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

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

Lemma 31. *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.*

The proof can be found in the full version of the paper [20].

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 $\mathcal{F}_{\mathrm{Prepocessing}}$ and the corrupt parties' shares.

Firstly, by Lemma 31, 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 $\mathcal{F}_{\mathrm{COT}}$ 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, $(\tilde{g}_{a,b}^j)^i$, where P_j is corrupt. This is computed by adding some PRF value, v, to the $\mathcal{F}_{\mathrm{COT}}$ outputs where P_i was receiver and P_j was sender (step 2 of $\Pi_{\mathrm{Bit}\times\mathrm{String}}$). 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 reduction to the PRF, since all the PRF keys are included in the output to the environment. To avoid this issue, we need the rerandomization step using a PRG, and the additional assumption of secure point-to-point channels.

Theorem 31. *Protocol* $\Pi_{\text{Preprocessing}}$ *from Fig.* 6 *UC-securely computes* $\mathcal{F}_{\text{Prepocessing}}$ *from Fig.* 5 *in the presence of a static, active adversary corrupting up to* $n-1$ *parties in the* $\{\mathcal{F}_{\text{COT}}, \mathcal{F}_{\text{BitMPC}}, \mathcal{F}_{\text{Rand}}, \mathcal{F}_{\text{Commit}}\}$*-hybrid model.*

The proof can be found in the full version of the paper [20].

4 More Efficient Garbling with Multi-party TinyOT

We now describe a less general, but concretely more efficient, variant of the protocol in the previous section. We replace the generic $\mathcal{F}_{\text{BitMPC}}$ functionality with a more specialized one based on 'TinyOT'-style information-theoretic MACs. This is asymptotically worse, but more practical, than using [21] for $\mathcal{F}_{\text{BitMPC}}$. It also allows us to completely remove the bit/string multiplications and consistency checks in $\Pi_{\text{Bit}\times\text{String}}$, since we show that these can be obtained directly from the TinyOT MACs. This means the only cost in the protocol, apart from opening and evaluating the garbled circuit, is the single bit multiplication per AND gate in the underlying TinyOT-based protocol.

In the full version of this paper [20], we present a complete description of a suitable TinyOT-based protocol. This is done by combining the multiplication triple generation protocol (over $\mathbb{F}_2$) from [17] with a consistency check to enforce correct shared random bits, which is similar to the more general check from the previous section.

4.1 Secret-Shared MAC Representation

For $x \in \{0,1\}$ held by P_i, define the following two-party MAC representation, as used in 2-party TinyOT [33]:

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

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

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

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

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

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

Claim 41. *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:

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

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

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

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 [1].

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 the full version [20]. 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 the full version. In that section we also present a complete description of a variant on the multi-party TinyOT protocol, which can be used to implement this functionality.

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

Following from the observation in Claim 41, if each party P_j chooses the global difference string in $\Pi_{\text{Preprocessing}}$ to be the same R^j as in the MAC representation, then given $[\lambda]$, additive shares of the products $\lambda \cdot R^j$ can be obtained at no extra cost. Moreover, the shares are guaranteed to be correct, and the honest party's shares will be random (subject to the constraint that they sum to the correct value), since they come directly from the $\mathcal{F}_{\text{n-TinyOT}}$ functionality. This means there is no need to perform the consistency check, which greatly simplifies the protocol.

The rest of the protocol is mostly the same as $\Pi_{\text{Preprocessing}}$ in Fig. 6, using $\mathcal{F}_{\text{n-TinyOT}}$ with $[\cdot]$-sharings instead of $\mathcal{F}_{\text{BitMPC}}$ with $\langle\cdot\rangle$-sharings. One other small difference is that because $\mathcal{F}_{\text{n-TinyOT}}$ does not have a private input command, we instead sample $[\lambda_w]$ shares for input wires using random bits, and later use a private output protocol to open the relevant input wire masks to P_i. This change is not strictly necessary, but simplifies the protocol for implementing $\mathcal{F}_{\text{n-TinyOT}}$—if $\mathcal{F}_{\text{n-TinyOT}}$ also had an **Input** command for sharing private inputs based on n-Bracket, it would be much more complex to implement with the correct distribution of shares and MACs.

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

1. Each party obtains a random key offset R^i by calling the **Initialize** command of $\mathcal{F}_{\text{n-TinyOT}}$.
2. For every wire w which is an input wire, or the output wire of an AND gate, the parties obtain a shared mask $[\lambda_w]$ using the **Bit** command of $\mathcal{F}_{\text{n-TinyOT}}$.
3. All the wire keys $k^i_{w,0}, k^i_{w,1} = k^i_{w,0} \oplus R^i$ are defined by P_i the same way as in $\Pi_{\text{Preprocessing}}$.
4. For XOR gates, the output wire mask is computed as $[\lambda_w] = [\lambda_u] + [\lambda_v]$.
5. For each AND gate, the parties compute $[\lambda_{uv}] = [\lambda_u \cdot \lambda_v]$.
6. The parties then obtain shares of the garbled circuit as follows:
   - For each AND gate $g \in G$ with wires (u, v, w), the parties use Claim 41 with the shared values $[\lambda_u], [\lambda_v], [\lambda_{uv} + \lambda_w]$, to define, for each $j \in [n]$, shares of the bit/string products:
   $$\lambda_u \cdot R^j, \quad \lambda_v \cdot R^j, \quad (\lambda_{uv} + \lambda_w) \cdot R^j$$
   - These are then used to define shares of $\rho_{j,a,b}$ and the garbled circuit, as in the original protocol.
7. For every circuit-output-wire w, the parties run Π_{Open} to reveal λ_w to all the parties.
8. For every *circuit input wire* w corresponding to party P_i's input, the parties run Π^i_{Open} to open λ_w to P_i.

The only interaction introduced in the new protocol is in the multiply and opening protocols, which were abstracted away by $\mathcal{F}_{\text{BitMPC}}$ in the previous protocol. Simulating and proving security of these techniques is straightforward, due to the correctness and randomness of the multiplication triples and MACs produced by $\mathcal{F}_{\text{n-TinyOT}}$. One important detail is the **Key Queries** command of the $\mathcal{F}_{\text{n-TinyOT}}$ functionality, which allows the adversary to try to guess an honest party's global MAC key share, R^i, and learn if the guess is correct. To allow the proof to go through, we modify $\mathcal{F}_{\text{Prepocessing}}$ to also have 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 $\mathcal{F}^{\text{KQ}}_{\text{Prepocessing}}$.

The following theorem can be proven, similarly to the proof of Theorem 31 where we modify the preprocessing functionality to support key queries, and adjust the simulation as described above.

Theorem 41. *The modified protocol described above UC-securely computes* $\mathcal{F}^{\mathrm{KQ}}_{\mathrm{Prepocessing}}$ *from Fig. 5 in the presence of a static, active adversary corrupting up to* $n-1$ *parties in the* $\mathcal{F}_{\text{n-TinyOT}}$*-hybrid model.*

5 The Online Phase

Our final protocol, presented in Fig. 9, 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 [39], 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 51. *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 Fig. 9, UC-securely computes* f *in the presence of a static, active 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 different to the proof in [29] as in our case the adversary may choose its additive error as a function of the garbled circuit). Then, conditioned on the event flip not occurring, we prove that the two executions are computationally indistinguishable via a reduction to the correlation robust PRF, inducing a garbled circuit that is indistinguishable. The complete proof can be found in the full version of the paper [20].

6 Performance

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

We have made a couple of tweaks to our protocol to simplify the implementation. We moved the **Open Garbling** stage to the preprocessing phase, instead of the online phase. This optimizes the online phase so that the amount of communication is independent of the size of the circuit. This change means that

The MPC Protocol - Π_{MPC}

On input a circuit C_f representing the function f and $\rho = (\rho_1, \ldots, \rho_n)$ 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 input wires w with input from P_i, party P_i computes $\Lambda_w = \rho_w \oplus \lambda_w$, where ρ_w is P_i's input to C_f, and λ_w was obtained in the preprocessing stage. Then, P_i broadcasts the public value Λ_w to all parties.
- For all input wires w, each 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 $j \in [n]$:
  $$k^j_{w,c} = \tilde{g}^j_{a,b} \oplus \left(\bigoplus_{i=1}^{n} F_{k^i_{u,a}, k^i_{v,b}}(g \| j) \right)$$
  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 $(k^1_{w,c}, \ldots, k^n_{w,c})$ 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.

Fig. 9. The MPC protocol - Π_{MPC}

our standard model security proof would no longer apply, but we could prove it secure using a random oracle instead of the circular-correlation robust PRF, similarly to [3,30]. Secondly, when not working in a modular fashion with a separate preprocessing functionality, the share rerandomization step in the output stage is not necessary to prove security of the entire protocol, so we omit this.

6.1 Implementation

We implemented our variant of the multi-party TinyOT protocol (given in the full version) using the `libOTe` library [36] for the fixed-correlation OTs and tested it for between 3 and 9 parties. We benchmarked the protocol over a 1 Gbps LAN on 5 servers with 2.3 GHz Intel Xeon CPUs with 20 cores. For the experiments with more than 5 parties, we had to run more than one party per machine; this should not make much difference in a LAN, as the number of threads being used was still fewer than the number of cores. As benchmarks, we measured the time for securely computing the circuits for AES (6800 AND gates) and SHA-256 (90825 AND gates).

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

Table 2. Runtimes in ms for AES and SHA-256 evalution with 9 parties

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

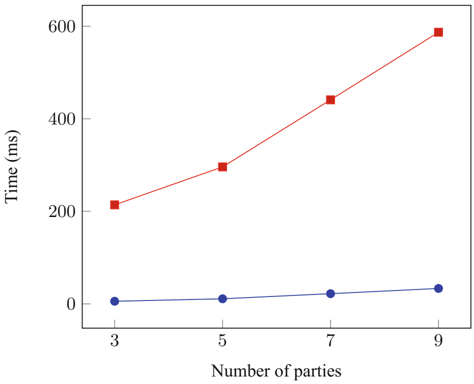

Fig. 10. AES performance (6800 AND gates).

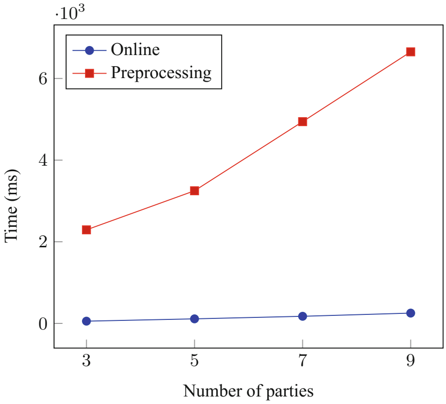

Fig. 11. SHA-256 performance (90825 AND gates).

Figures 10 and 11 show how the performance of AES and SHA-256 scales with different numbers of parties, in the amortized setting. Although the asymptotic complexity is quadratic, the runtimes grow relatively slowly as the number of parties increases. This is because in the preprocessing phase, the amount of data sent *per party* is actually linear. However, the super-linear trend is probably due to the limitations of the total network capacity, and the computational costs.

Comparison with other works. We calculated the cost of computing the SPDZ-BMR protocol [29] using [24] to derive estimates for creating the SPDZ triples (the main cost). Using MASCOT over $\mathbb{F}_{2^\kappa}$ with free-XOR, SPDZ-BMR requires $3n + 1$ multiplications per garbled AND gate. This gives an estimated cost of at least 14 s to evaluate AES, which is over 20x slower than our protocol.

The only other implementation of actively secure, constant-round, dishonest majority MPC is the concurrent work of [23], which presents implementation figures for up to 256 parties running on Amazon servers. Their runtimes with 9 parties in a LAN setting are around 200 ms for AES and 2200 ms for SHA-256, which is around 3 times faster than our results. However, their LAN setup has 10Gbps bandwidth, whereas we only tested on machines with 1Gbps bandwidth. Since the bottleneck in our implementation is mostly communication, it seems that our implementation could perform similar to or even faster than theirs in the same environment, despite our higher communication costs. However, it is not possible to make an accurate comparison without testing both implementations in the same environment.

Compared with protocols based solely on secret-sharing, such as SPDZ and TinyOT, the advantage of our protocol is the low round complexity. We have not yet managed to benchmark our protocol in a WAN setting, but since our total round complexity is less than 20, it should perform reasonably fast. With secret-sharing, using e.g. TinyOT, evaluating the AES circuit requires at least 40 rounds in just the online phase (it can be done with 10 rounds [14], but this uses a special representation of the AES function, rather than a general circuit), whilst computing the SHA-256 circuit requires *4000 rounds*. In a network with 100 ms delay between parties, the AES online time alone would be at least 4 s, whilst SHA-256 would take over 10 min to securely compute in that setting. If our protocol is run in this setting, we should be able to compute both AES and SHA-256 in just a few seconds (assuming that latency rather than bandwidth is the bottleneck).

6.2 Communication Complexity Analysis

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

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

Protocol	# Executions	Function-indep. prep.	Function-dep. prep.	Online
[37]	32	–	3.75 MB	25.76 kB
	128	–	2.5 MB	21.31 kB
	1024	–	1.56 MB	16.95 kB
[34]	1	14.94 MB	227 kB	16.13 kB
	32	8.74 MB	227 kB	16.13 kB
	128	7.22 MB	227 kB	16.13 kB
	1024	6.42 MB	227 kB	16.13 kB
[38]	1	2.86 MB	570 kB	4.86 kB
	32	2.64 MB	570 kB	4.86 kB
	128	2.0 MB	570 kB	4.86 kB
	1024	2.0 MB	570 kB	4.86 kB
Ours + [38]	1	2.86 MB	872 kB	4.22 kB
	32	2.64 MB	872 kB	4.22 kB
	128	2.0 MB	872 kB	4.22 kB
	1024	2.0 MB	872 kB	4.22 kB

Two Parties. In Table 3 we compare the cost of our protocol in the two-party case, with state-of-the-art secure two-party computation protocols. We instantiate our TinyOT-based preprocessing method with the optimized, two-party TinyOT protocol from [38], lowering the previous costs further. For consistency with the other two-party protocols, we divide the protocol costs into three phases: function-independent preprocessing, which only depends on the size of the circuit; function-dependent preprocessing, which depends on the exact structure of the circuit; and the online phase, which depends on the parties' inputs. As with the implementation, we move the garbled circuit opening to the function-dependent preprocessing, to simplify the online phase.

The online phase of the modified protocol is just two rounds of interaction, and has the lowest online cost of *any* actively secure two-party protocol.[2] The

[2] If counting the *total* amount of data sent, in both directions, our online cost would be larger than [38], 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.

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 [37] 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.

Comparison with Multi-party Protocols. In Table 4 we compare our work with previous constant-round protocols suitable for any number of parties, again for evaluating the AES circuit. We do not present the communication complexity of the online phase as we expect it to be very similar in all of the protocols. We denote by MASCOT-BMR-FX an optimized variant of [29], modified to use free-XOR as in our protocol, with multiplications done using the OT-based MASCOT protocol [24].

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

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
[23]	Active	4.8 MB	20.4 MB	1.3 MB	4.4 MB
Ours	Active	14.01 MB	63.22 MB	1.31 MB	4.37 MB

As in the previous section, we move the cost of opening the garbled circuit to the preprocessing phase for all of the presented protocols (again relying on random oracles). By applying this technique the online phase of our work is just two rounds, and has exactly the same complexity as the current most efficient *semi-honest* constant-round MPC protocol for any number of parties [4], except we achieve active security. We see that with respect to other actively secure protocols, we improve the communication cost of the preprocessing by around 2–4 orders of magnitude. Moreover, our protocol scales much better with n, since the complexity is $O(n^2)$ instead of $O(n^3)$. The concurrent work of Katz et al. [23] requires around 3 times less communication than our protocol, which is due to their optimized version of the multi-party TinyOT protocol.

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

The first author was supported by the European Research Council under the ERC consolidators grant agreement No. 615172 (HIPS), and by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office. The second author was supported by the Defense Advanced Research Projects Agency (DARPA) and Space and Naval Warfare Systems Center, Pacific (SSC Pacific) under contract No. N66001-15-C-4070, and by the Danish Independent Research Council, Grant-ID DFF-6108-00169. The third author was supported by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 643161.

References

1. Beaver, D.: Efficient multiparty protocols using circuit randomization. In: Feigenbaum, J. (ed.) CRYPTO 1991. LNCS, vol. 576, pp. 420–432. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-46766-1_34
2. Beaver, D., Micali, S., Rogaway, P.: The round complexity of secure protocols (extended abstract). In: 22nd ACM STOC, pp. 503–513. ACM Press, May 1990
3. Bellare, M., Hoang, V.T., Rogaway, P.: Foundations of garbled circuits. In: Yu, T., Danezis, G., Gligor, V.D. (eds.) ACM CCS 12, pp. 784–796. ACM Press, October 2012
4. Ben-Efraim, A., Lindell, Y., Omri, E.: Optimizing semi-honest secure multiparty computation for the internet. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) ACM CCS 16, pp. 578–590. ACM Press, October 2016
5. Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In: 20th ACM STOC, pp. 1–10. ACM Press, May 1988
6. Burra, S.S., Larraia, E., Nielsen, J.B., Nordholt, P.S., Orlandi, C., Orsini, E., Scholl, P., Smart, N.P.: 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
7. Canetti, R.: Universally composable security: a new paradigm for cryptographic protocols. In: 42nd FOCS, pp. 136–145. IEEE Computer Society Press, October 2001
8. Canetti, R., Cohen, A., Lindell, Y.: A simpler variant of universally composable security for standard multiparty computation. In: Gennaro, R., Robshaw, M. (eds.) CRYPTO 2015. LNCS, vol. 9216, pp. 3–22. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48000-7_1
9. Cascudo, I., Damgård, I., David, B., Döttling, N., Nielsen, J.B.: Rate-1, linear time and additively homomorphic UC commitments. In: Robshaw, M., Katz, J. (eds.) CRYPTO 2016. LNCS, vol. 9816, pp. 179–207. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53015-3_7
10. Choi, S.G., Katz, J., Kumaresan, R., Zhou, H.-S.: On the security of the "Free-XOR" technique. In: Cramer, R. (ed.) TCC 2012. LNCS, vol. 7194, pp. 39–53. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28914-9_3
11. Choi, S.G., Katz, J., Malozemoff, A.J., Zikas, V.: Efficient three-party computation from cut-and-choose. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014. LNCS, vol. 8617, pp. 513–530. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44381-1_29

12. Damgård, I., Ishai, Y.: Scalable secure multiparty computation. In: Dwork, C. (ed.) CRYPTO 2006. LNCS, vol. 4117, pp. 501–520. Springer, Heidelberg (2006). https://doi.org/10.1007/11818175_30
13. Damgård, I., Nielsen, J.B.: Scalable and unconditionally secure multiparty computation. In: Menezes, A. (ed.) CRYPTO 2007. LNCS, vol. 4622, pp. 572–590. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74143-5_32
14. Damgård, I., Nielsen, J.B., Nielsen, M., Ranellucci, S.: Gate-scrambling revisited - or: the TinyTable protocol for 2-party secure computation. Cryptology ePrint Archive, Report 2016/695 (2016). http://eprint.iacr.org/2016/695
15. Damgård, I., Pastro, V., Smart, N.P., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 643–662. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32009-5_38
16. Damgård, I., Zakarias, S.: Constant-overhead secure computation of boolean circuits using preprocessing. In: Sahai, A. (ed.) TCC 2013. LNCS, vol. 7785, pp. 621–641. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36594-2_35
17. Frederiksen, T.K., Keller, M., Orsini, E., Scholl, P.: A unified approach to MPC with preprocessing using OT. In: Iwata, T., Cheon, J.H. (eds.) ASIACRYPT 2015. LNCS, vol. 9452, pp. 711–735. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48797-6_29
18. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or a completeness theorem for protocols with honest majority. In: Aho, A. (ed.) 19th ACM STOC, pp. 218–229. ACM Press, May 1987
19. Goldwasser, S., Lindell, Y.: Secure multi-party computation without agreement. J. Cryptol. **18**(3), 247–287 (2005)
20. Hazay, C., Scholl, P., Soria-Vazquez, E.: Low cost constant round MPC combining BMR and oblivious transfer. IACR Cryptology ePrint Archive, 2017:214 (2017)
21. Ishai, Y., Prabhakaran, M., Sahai, A.: Founding cryptography on oblivious transfer – efficiently. In: Wagner, D. (ed.) CRYPTO 2008. LNCS, vol. 5157, pp. 572–591. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85174-5_32
22. Ishai, Y., Prabhakaran, M., Sahai, A.: Secure arithmetic computation with no honest majority. In: Reingold, O. (ed.) TCC 2009. LNCS, vol. 5444, pp. 294–314. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00457-5_18
23. Katz, J., Ranellucci, S., Wang, X.: Authenticated garbling and efficient maliciously secure multi-party computation. IACR Cryptology ePrint Archive, 2017:189 (2017)
24. Keller, M., Orsini, E., Scholl, P.: MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) ACM CCS 16, pp. 830–842. ACM Press, October 2016
25. Kolesnikov, V., Schneider, T.: Improved garbled circuit: free XOR gates and applications. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008. LNCS, vol. 5126, pp. 486–498. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70583-3_40
26. Lindell, Y., Pinkas, B.: An efficient protocol for secure two-party computation in the presence of malicious adversaries. In: Naor, M. (ed.) EUROCRYPT 2007. LNCS, vol. 4515, pp. 52–78. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72540-4_4
27. Lindell, Y., Pinkas, B.: A proof of security of Yao's protocol for two-party computation. J. Cryptol. **22**(2), 161–188 (2009)

28. Lindell, Y., Pinkas, B.: Secure two-party computation via cut-and-choose oblivious transfer. In: Ishai, Y. (ed.) TCC 2011. LNCS, vol. 6597, pp. 329–346. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19571-6_20
29. Lindell, Y., Pinkas, B., Smart, N.P., Yanai, A.: Efficient constant round multi-party computation combining BMR and SPDZ. In: Gennaro, R., Robshaw, M. (eds.) CRYPTO 2015. LNCS, vol. 9216, pp. 319–338. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48000-7_16
30. Lindell, Y., Riva, B.: Blazing fast 2PC in the offline/online setting with security for malicious adversaries. In: Ray, I., Li, N., Kruegel, C. (eds.) ACM CCS 15, pp. 579–590. ACM Press, October 2015
31. Lindell, Y., Smart, N.P., Soria-Vazquez, E.: More efficient constant-round multi-party computation from BMR and SHE. In: Hirt, M., Smith, A. (eds.) TCC 2016. LNCS, vol. 9985, pp. 554–581. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53641-4_21
32. Mohassel, P., Rosulek, M., Zhang, Y.: Fast and secure three-party computation: the garbled circuit approach. In: Ray, I., Li, N., Kruegel, C. (eds.) ACM CCS 15, pp. 591–602. ACM Press, October 2015
33. Nielsen, J.B., Nordholt, P.S., Orlandi, C., Burra, S.S.: A new approach to practical active-secure two-party computation. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 681–700. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32009-5_40
34. Nielsen, J.B., Schneider, T., Trifiletti, R.: 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
35. Rabin, T., Ben-Or, M.: Verifiable secret sharing and multiparty protocols with honest majority (extended abstract). In: 21st ACM STOC, pp. 73–85. ACM Press, May 1989
36. Rindal, P.: libOTe: an efficient, portable, and easy to use Oblivious Transfer Library. https://github.com/osu-crypto/libOTe
37. Rindal, P., Rosulek, M.: Faster malicious 2-party secure computation with online/offline dual execution. In: 25th USENIX Security Symposium (USENIX Security 16), pp. 297–314. USENIX Association, Austin (2016)
38. Wang, X., Ranellucci, S., Katz, J.: Authenticated garbling and communication-efficient, constant-round, secure two-party computation. IACR Cryptology ePrint Archive, 2017:30 (2017)
39. Yao, A.C.-C.: How to generate and exchange secrets (extended abstract). In: 27th FOCS, pp. 162–167. IEEE Computer Society Press, October 1986