Dishonest Majority Multi-Party Computation for Binary Circuits

Enrique Larraia, Emmanuela Orsini, and Nigel P. Smart

Dept. Computer Science, University of Bristol, United Kingdom
Enrique.LarraiadeVega@bristol.ac.uk,Emmanuela.Orsini@bristol.ac.uk,
nigel@cs.bris.ac.uk

Abstract. We extend the Tiny-OT two party protocol of Nielsen et al (CRYPTO 2012) to the case of n parties in the dishonest majority setting. This is done by presenting a novel way of transferring pairwise authentications into global authentications. As a by product we obtain a more efficient manner of producing globally authenticated shares, in the random oracle model, which in turn leads to a more efficient two party protocol than that of Nielsen et al.

1 Introduction

In recent years actively secure MPC has moved from a theoretical subject into one which is becoming more practical. In the variants of multi-party computation which are based on secret sharing the major performance improvement has come from the technique of authenticating the shared data and/or the shares themselves using information theoretic message authentication codes (MACs). This idea has been used in a number of works: In the case of two-party MPC for binary circuits in [13], for n-party dishonest majority MPC for arithmetic circuits over a "largish" finite field [4,7], and for n-party dishonest majority MPC over binary circuits [8]. All of these protocols are in the pre-processing model, in which the parties first engage in a function and input independent offline phase. The offline phase produces various pieces of data, often Beaver style [3] "multiplication triples", which are then consumed in the online phase when the function is determined and evaluated.

In the case of the protocol of [13], called Tiny-OT in what follows, the authors use the technique of applying information theoretic MACs to the oblivious transfer (OT) based GMW protocol [10] in the two party setting. In this protocol the offline phase consists of producing a set of pre-processed random OTs which have been authenticated. The offline phase is then executed efficiently using a variant of the OT extension protocol of [12]. For a detailed discussion on OT extension see [2,12,13]. In this work we shall take OT extension as a given sub-procedure.

One can think of the Tiny-OT protocol as applying the authentication technique of [4] to the two party, binary circuit case, with a pre-processing which is based on OT as opposed to semi-homomorphic encryption. For two party protocols over binary circuits practical experiments show that Tiny-OT far out-performs other protocols, such as those based on Yao's garbled circuit technique. This is because of the performance of the offline phase of the Tiny-OT protocol. Thus a natural question is to ask, whether one can extend the Tiny-OT protocol to the n-party setting for binary circuits.

Results and Techniques. In this paper we mainly address ourselves to the above question, i.e. how can we generalize the two-party protocol from [13] to the n-party setting?

We first describe what are the key technical difficulties we need to overcome. The Tiny-OT protocol at its heart has a method for authenticating random bits via pairwise MACs, which itself

is based on an efficient protocol for OT-extension. In [13] this protocol is called aBit. Our aim is to use this efficient two-party process as a black-box. Unfortunately, if we extend this procedure naively to the three party case, we would obtain (for example) that parties P_1 and P_2 could execute the protocol so that P_1 obtains a random bit and a MAC, whilst P_2 obtains a key for the MAC used to authenticate the random bit. However, party P_3 obtains no authentication on the random bit obtained by P_1, nor does it obtain any information as to the MAC or the key.

To overcome this difficulty, we present a protocol in which we fix an unknown global random key and where each party holds a share of this key. Then by executing the pairwise aBit protocol, we are able to obtain a secret shared value, as well as a shared MAC, by all n-parties. This resulting MAC is identical to the MAC used in the SPDZ protocol from [6]. This allows us to obtain authenticated random shares, and in addition to permit parties to enter their inputs into the MPC protocol.

The online phase will then follow similarly to [6], if we can realize a protocol to produce "multiplication triples". In [13] one can obtain such triples by utilizing a complex method to produce authenticated random OTs and authenticated random ANDs (called aOTs and aANDs)[1]. We notice that our method for obtaining authenticated bits also enables us to obtain a form of authenticated OTs in a relatively trivial manner, and such authenticated OTs can be used directly to implement a multiplication gate in the online phase.

Our contribution is twofold. First, we generalize the two-party Tiny-OT protocol to the n-party setting, using a novel technique for authentication of secret shared bits, and completely new offline and online phases. Thus we are able to dispense with the protocols to generate aOTs and aANDs from [13], obtaining a simple and efficient online protocol. Second, and as a by product, we obtain a more efficient protocol than the original Tiny-OT protocol, in the two party setting when one measures efficiency in terms of the number of aBit's needed per multiplication gate. The security of our protocols are proven in the standard universal composability (UC) framework [5] against a malicious adversary and static corruption of parties. The definitional properties of an MPC protocol are implicit in this framework: output indistinguishability of the ideal and the real process gives *correctness*, and the fact that any information gathered by a real adversary is obtainable by an ideal adversary gives *privacy*. Although not explicitely stated, we work with the random oracle model, as we need to implement commitments to check the correctness of the MACs, more precisely, we work with programmable random oracles. See the Appendix of [6] for details.

Related Work. For the case of n party protocols, where $n > 2$, there are three main techniques using such MACs. In [4] each share of a given secret is authenticated by pairwise MACs, i.e. if party P_i holds a share a_i, then it will also hold a MAC $M_{i,j}$ for every $j \neq i$, and party P_j will hold a key $K_{i,j}$. Then, when the value a_i is made public, party P_i also reveals the $n-1$ MAC values, that are then checked by other parties using their private keys $K_{i,j}$. Note that each pair of parties holds a separate key/MAC for each share value. In [7] the authors obtain a more efficient online protocol by replacing the MACs from [4] with global MACs which authenticate the shared values a, as opposed to the shares themselves. The authentication is also done with respect to a fixed global MAC key (and not pairwise and data dependent). This method was improved in [6], where it is shown how to verify these global MACs without revealing the secret global key. In [8] the authors adapt the technique from [7] for the case of small finite fields, in a way which allows one to authenticate multiple field elements at the same time, without requiring multiple MACs.

[1] In fact the paper [13] does not produce such multiplication triples, but they follow immediately from the presentation in the paper and would result in a more efficient online phase than that described in [13]

This is performed using a novel application of ideas from coding theory, and results in a reduced overhead for the online phase.

Future Directions. We end this introduction by describing two possible extensions to our work. Firstly, each bit in our protocol is authenticated by an element in a finite field $\mathbb{F}_{2^\kappa}$. Whilst such values are never transmitted in our online phase due to our $\mathsf{MACCheck}$ protocol, they do provide an overhead in the computation. In [8] the authors show how to reduce this overhead using coding theory techniques. It would be interesting to see how such techniques could be applied to our protocol, and what advantage if any they would bring.

Secondly, our protocol requires $n \cdot (n-1)/2$ executions of the aBit protocol from [13]. Each pairwise invocation requires the execution of an OT-extension protocol, and hence we require $O(n^2)$ such OT-channels. In [11], in the context of traditional MPC protocols, the authors present techniques and situations in which the number of OT-channels can be reduced to $O(n)$. It would be interesting to see how such techniques could be applied in practice to the protocol described in this paper.

2 Notation

In this section we settle the notation used throughout the paper. We use κ to denote the security parameter. We let $\mathsf{negl}(\kappa)$ denote some unspecified function $f(\kappa)$, such that $f = o(\kappa^{-c})$ for every fixed constant c, saying that such a function is *negligible* in κ. We say that a probability is *overwhelming* in κ if it is $1 - \mathsf{negl}(\kappa)$.

We consider the sets $\{0,1\}$ and $\mathbb{F}_2^\kappa$ endowed with the structure of the fields $\mathbb{F}_2$ and $\mathbb{F}_{2^\kappa}$, respectively. Let $\mathbb{F} = \mathbb{F}_{2^\kappa}$, we will denote elements in $\mathbb{F}$ with greek letters and elements in $\mathbb{F}_2$ with roman letters.

We will additively secret share bits and elements in $\mathbb{F}$, among a set of parties $\mathcal{P} = \{P_1, \ldots, P_n\}$, and sometimes abuse notation identifying subsets $\mathcal{I} \subseteq \{1, \ldots, n\}$ with the subset of parties indexed by $i \in \mathcal{I}$. We write $\langle a \rangle^{\mathcal{I}}$ if a is shared amongst the set $\mathcal{I} = \{i_1, \ldots, i_t\}$ with party P_{i_j} holding a value a_{i_j}, such that $\sum_{i_j \in \mathcal{I}} a_{i_j} = a$. Also, if an element $x \in \mathbb{F}_2$ (resp. $\beta \in \mathbb{F}$) is additively shared among *all* parties we write $\langle x \rangle$ (resp. $\langle \beta \rangle$). We adopt the convention that if $a \in \mathbb{F}_2$ (resp. $\beta \in \mathbb{F}$) then the shares $a_i \in \mathbb{F}_2$ (resp. $\beta_i \in \mathbb{F}$).

(Linear) arithmetic on the $\langle \cdot \rangle^{\mathcal{I}}$ sharings can be performed as follows. Given two sharings $\langle x \rangle^{\mathcal{I}_x} = \{x_{i_j}\}_{i_j \in \mathcal{I}_x}$ and $\langle y \rangle^{\mathcal{I}_y} = \{y_{i_j}\}_{i_j \in \mathcal{I}_y}$ we can compute the following linear operations

$$\begin{aligned} a \cdot \langle x \rangle^{\mathcal{I}_x} &= \{a \cdot x_{i_j}\}_{i_j \in \mathcal{I}_x}, \\ a + \langle x \rangle^{\mathcal{I}_x} &= \{a + x_{i_1}\} \cup \{x_{i_j}\}_{i_j \in \mathcal{I}_x \setminus \{i_1\}}, \\ \langle x \rangle^{\mathcal{I}_x} + \langle y \rangle^{\mathcal{I}_y} &= \langle x + y \rangle^{\mathcal{I}_x \cup \mathcal{I}_y} \\ &= \{x_{i_j}\}_{i_j \in \mathcal{I}_x \setminus \mathcal{I}_y} \cup \{y_{i_j}\}_{i_j \in \mathcal{I}_y \setminus \mathcal{I}_x} \cup \{x_{i_j} + y_{i_j}\}_{i_j \in \mathcal{I}_x \cap \mathcal{I}_y}. \end{aligned}$$

Our protocols will make use of pseudo-random functions, which we will denote by $\mathsf{PRF}_s^{X,t}(\cdot)$ where for a key s and input $m \in \{0,1\}^*$ the pseudo-random function is defined by $\mathsf{PRF}_s^{X,t}(m) \in X^t$, where X is some set and t is a non-negative integer.

Authentication of Secret Shared Values. As described in the introduction the literature gives two ways to authenticate a secret globally held by a system of parties, one is to authenticate the shares of each party, as in [4], the other is to authenticate the secret itself, as in [7]. In addition we can also have authentication in a pairwise manner, as in [4,13], or in a global manner, as in [7]. Both combinations of these variants can be applied, but each implies important practical differences, e.g., the total amount of data each party needs to store and how checking of the MACs is performed. In this work we will use a combination of different techniques, indeed the main technical trick is a method to pass from the technique used in [13] to the technique used in [7].

Our main technique for authentication of secret shared bits is applied by placing an *information theoretic tag* (MAC) on the shared bit x. The authenticating key is a random line in $\mathbb{F}$, and the MAC on x is its corresponding line point, thus, the linear equation $\mu_\delta(x) = \nu_\delta(x) + x \cdot \delta$ holds, for some $\mu_\delta(x), \nu_\delta(x), \delta \in \mathbb{F}$. We will use these lines in various operations[2], for various values of δ. In particular, there will be a special value of δ, which we denote by α and assume to be $\langle \alpha \rangle^{\mathcal{P}}$ shared, which represents the *global* key for our online MPC protocol. This will be the same key for every bit that needs to be authenticated. It will turn out that for the key α we always have $\nu_\alpha(x) = 0$. By abuse of notation we will sometimes refer to a general δ also as a *global* key, and then the corresponding $\nu_\delta(x)$, is called the *local* key.

Distinguishing between parties, say $\mathcal{I}$, that can reconstruct bits (together with the line point), and those parties, say $\mathcal{J}$, that can reconstruct the line gives a natural generalization of both ways to authenticate, and it also allows to move easily from one to another. We write $[x]^{\mathcal{I}}_{\delta,\mathcal{J}}$ if there exist $\mu_\delta(x), \nu_\delta(x) \in \mathbb{F}$ such that:

$$\mu_\delta(x) = \nu_\delta(x) + x \cdot \delta,$$

where we have that x and $\mu_\delta(x)$ are $\langle \cdot \rangle^{\mathcal{I}}$ shared, and $\nu_\delta(x)$ and δ are $\langle \cdot \rangle^{\mathcal{J}}$ shared, i.e. there are values x_i, μ_i, and ν_j, δ_j, such that

$$x = \sum_{i \in \mathcal{I}} x_i, \qquad \mu_\delta(x) = \sum_{i \in \mathcal{I}} \mu_i, \qquad \nu_\delta(x) = \sum_{j \in \mathcal{J}} \nu_j, \qquad \delta = \sum_{j \in \mathcal{J}} \delta_j.$$

Notice that $\mu_\delta(x)$ and $\nu_\delta(x)$ depend on δ and x: we can fix δ and so obtain *key-consistent* representations of bits, or we can fix x and obtain different *key-dependant* representations for the same bit x. To ease the reading, we drop the sub-index $\mathcal{J}$ if $\mathcal{J} = \mathcal{P}$, and, also, the dependence on δ and x when it is clear from the context. We note that in the case of $\mathcal{I}_x = \mathcal{J}_x$ then we can assume $\nu_j = 0$.

When we take the fixed global key α and we have $\mathcal{I}_x = \mathcal{J}_x = \mathcal{P}$, we simplify notation and write $[\![x]\!] = [x]^{\mathcal{P}}_{\alpha,\mathcal{P}}$. By our comment above we can, in this situation, set $\nu_j = 0$ [3], this means that a $[\![x]\!]$ sharing is given by two sharings $\left(\langle x \rangle^{\mathcal{P}}, \langle \mu \rangle^{\mathcal{P}}\right)$. Notice that the $[\![\cdot]\!]$-representation of a bit x implies that x is *authenticated* with the global key α and that it is $\langle \cdot \rangle$-shared, i.e. its value is actually unknown to the parties.

This notation does not quite align with the previous secret sharing schemes used in the literature, but it is useful for our purposes. For example, with this notation the MAC scheme of [4] is one where each data element x is shared via $[x_i]^i_{\alpha_j,j}$ sharings. Thus the data is shared via a $\langle x \rangle$ sharing and the authentication is performed via $[x_i]^i_{\alpha_j,j}$ sharings, i.e. we are using two sharing schemes simultaneously. In [7] the data is shared via our $[\![x]\!]$ notation, except that the MAC key value ν is

[2] For example, we will also use lines to generate OT-tuples, i.e. quadruples of authenticated bits which satisfy the algebraic equation for a random OT.

[3] Otherwise one can subtract ν_j from μ_j, before setting ν_j to zero.

set equal to $\nu = \nu'/\alpha$, where ν' being a *public value*, as opposed to a shared value. Our $[\![x]\!]$ sharing is however identical to that used in [6], bar the differences in the underlying finite fields.

Looking ahead we say that a bit $[\![x]\!]$ is *partially opened* if $\langle x \rangle$ is opened, i.e. the parties reveal the shares of x, but not the shares of the MAC value $\mu_\alpha(x)$.

Arithmetic on $[\![x]\!]$ Shared Values. Given two representations $[x]^{\mathcal{I}_x}_{\delta,\mathcal{J}_x} = \big(\langle x \rangle^{\mathcal{I}_x}, \langle \mu_\delta(x) \rangle^{\mathcal{I}_x}, \langle \nu_\delta(x) \rangle^{\mathcal{J}_x}\big)$ and $[y]^{\mathcal{I}_y}_{\delta,\mathcal{J}_y} = \big(\langle y \rangle^{\mathcal{I}_y}, \langle \mu_\delta(y) \rangle^{\mathcal{I}_y}, \langle \nu_\delta(y) \rangle^{\mathcal{J}_y}\big)$, under same the δ, the parties can locally compute $[x + y]^{\mathcal{I}_x \cup \mathcal{I}_y}_{\delta,\mathcal{J}_x \cup \mathcal{J}_y}$ as $\big(\langle x \rangle^{\mathcal{I}_x} + \langle y \rangle^{\mathcal{I}_y}, \langle \mu_\delta(x) \rangle^{\mathcal{I}_x} + \langle \mu_\delta(y) \rangle^{\mathcal{I}_y}, \langle \nu_\delta(x) \rangle^{\mathcal{J}_x} + \langle \nu_\delta(y) \rangle^{\mathcal{J}_y}\big)$ using the arithmetic on $\langle \cdot \rangle^{\mathcal{I}}$ sharings above.

Let $[\![x]\!] = \big(\langle x \rangle, \langle \mu(x) \rangle\big)$ and $[\![y]\!] = \big(\langle y \rangle, \langle \mu(y) \rangle\big)$ be two different authenticated bits. Since our sharings are linear, as well as the MACs, it is easy to see that the parties can locally perform linear operations:

$$\begin{aligned} [\![x]\!] + [\![y]\!] &= \big(\langle x \rangle + \langle y \rangle,\ \langle \mu(x) \rangle + \langle \mu(y) \rangle\big) = [\![x + y]\!] \\ a \cdot [\![x]\!] &= \big(a \cdot \langle x \rangle,\ a \cdot \langle \mu(x) \rangle\big) = [\![a \cdot x]\!], \\ a + [\![x]\!] &= \big(a + \langle x \rangle, \langle \mu(a + x) \rangle\big) = [\![a + x]\!]. \end{aligned}$$

where $\langle \mu(a + x) \rangle$ is the sharing obtained by each party $i \in \mathcal{P}$ holding the value $\alpha_i \cdot a + \mu_i(x)$.

This means that the only remaining question to enable MPC on $[\![\cdot]\!]$-shared values is how to perform multiplication and how to generate the $[\![\cdot]\!]$-shared values in the first place. Note, that a party P_i that wishes to enter a value into the MPC computation is wanting to obtain a $[x]^i_{\alpha,\mathcal{P}}$ sharing of its input value x, and that this is a $[\![x]\!]$-representation if we set $x_i = x$ and $x_j = 0$ for $j \neq i$.

3 MPC Protocol for Binary Circuit

We start presenting a high level view of the protocols that allow us to perform multi-party computation for binary circuits. We assume synchronous communication and authentic point-to-point channels.

Our protocol is in the pre-processing model in which we allow a function (and input) independent pre-processing, or offline, phase which produces correlated randomness. This enables a lightweight online phase, that does not need public-key machinery. In the following sections we will describe a protocol, Π_{Online}, implementing the actual function evaluation in the $(\mathcal{F}_{\mathsf{Comm}}, \mathcal{F}_{\mathsf{Prep}})$-hybrid model; a protocol, Π_{Prep}, implementing the offline phase in the $(\mathcal{F}_{\mathsf{Comm}}, \mathcal{F}_{\mathsf{Bootstrap}})$-hybrid model; and a novel way to authenticate bits to more than two parties, which takes as starting point the aBit command of [13], and which we model with the $\mathcal{F}_{\mathsf{Bootstrap}}$ functionality.

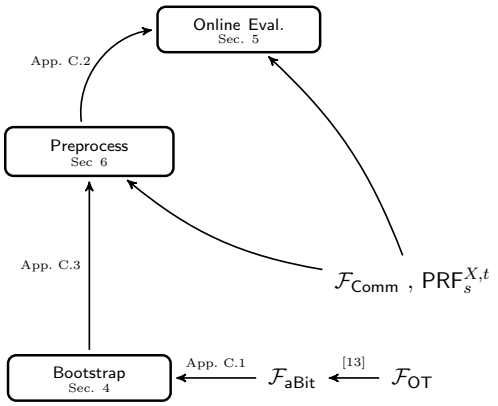

Figure 1 Overview of Protocols Enabling MPC

The online phase implements the standard functionality $\mathcal{F}_{\mathsf{Online}}$ (see Appendix C.2) It is based on the $[\![\cdot]\!]$-representation of bits described in Section 2, and it is very similar to the online phase of other MPC protocols [6,7,8,13]. We compute a function represented as a binary circuit, where private inputs are additively shared among the parties, and correctness is guaranteed by using additive secret sharings of linear MACs with global secret key α. For simplicity we assume one single input for each party and one public output. The online protocol, presented in Section 5, uses the linearity of the $[\![\cdot]\!]$-sharings to perform additions and scalar multiplications locally. For general multiplications we need utilize data produced during the offline phase, in particular the output of the GaOT (Global authenticated OT) command of Section 6. Refer to Figure 2 for a complete description of the functionality for preprocessing data. The aforementioned command GaOT builds upon $\Pi_{\mathsf{Bootstrap}}$ protocol, described in Section 4, to generate random authenticated OTs and, as we noted above, we skip the less efficient procedures of [13].

The Functionality $\mathcal{F}_{\mathsf{Prep}}$

Let A be the set of indices of corrupt parties.

Initialize: On input (Init) from honest parties, the functionality samples random α_i for each $i \notin A$. It waits for the environment to input corrupt shares $\{\alpha_j\}_{j\in A}$ If any $j \in A$ outputs abort, then the functionality aborts and returns the set of $j \in A$ which returned abort. Otherwise the functionality sets $\alpha = \alpha_1 + \cdots + \alpha_n$, and outputs α_k to honest P_k.

Share: On input (i, x, Share) from party P_i, and (i, Share) from all other parties. The functionality produces an authentication $[\![x]\!] = (\langle x\rangle, \langle\mu\rangle)$. It sets $x_j = 0$ if $j \neq i$. Also, the MAC might be shifted by a value Δ_H, i.e. $\mu = x \cdot \alpha + \Delta_H$, where Δ_H is an $\mathbb{F}_2$-linear combination of $\{\alpha_k\}_{k\notin A}$ not known to the environment. It proceeds as follows:

- Set $\mu = x \cdot \alpha$. If $i \in A$, the environment specifies x.
- Wait for the environment to specify MAC shares $\{\mu_j\}_{j\in A}$, and generate $\langle\mu\rangle$ where the portion of honest shares is consistent with the adversarial shares, but otherwise random.
- Set $x_k = 0$ if $k \neq i$, $k \notin A$. If the environment inputs $\mathsf{shift\text{-}P_k}$ set $\mu_k = \mu_k + \alpha_k$.
- Output $\{x_k, \mu_k\}$ to honest P_k

GaOT: On input (GaOT) from the parties, the functionality waits for the environment to input "Abort" or "Continue". If it is told to abort, it outputs the special symbol $\varnothing$ to all parties.
Otherwise it samples three random bits e, x_0, x_1, and sets $z = x_e$. Then, for every bit $y \in \{e, z, x_0, x_1\}$ the functionality produces an authentication $[\![y]\!] = (\langle y\rangle, \langle\mu(y)\rangle)$, but let the environment to specify shares for corrupt P_j. It proceeds as follows:

- Set $\mu(y) = y \cdot \alpha$.
- Wait for the environment to input bit shares $\{y_j\}_{j\in A}$, and MAC shares $\{\mu_j\}_{j\in A}$, and creates sharings $\langle y\rangle$, $\langle\mu\rangle$ where the portion of honest shares is consistent with adversarial shares.
- Output $\{y_k, \mu_k\}$ to honest P_k.

Figure 2 Ideal Preprocessing

Notice that, as in [6,7,8,13], during the online computation of the circuit we do not know if we are working with the correct values, since we do not check the MACs of partially opened values during the computation. This check is postponed to the end of the protocol, where we call the MACCheck procedure as in [6] (see Protocol 10). Note this procedure enables the checking of multiple sets of values partially opened during the computation without revealing the global secret key α, thus our MPC protocol can implement reactive functionalities.

The MAC checking protocol is called in both the offline and the online phases, it requires access to an ideal functionality for commitments $\mathcal{F}_{\mathsf{Comm}}$ in the random oracle model (see Appendix B), and it is not intended to implement any functionality. Also, note that the algebraic correctness of

the output of the GaOT command in the offline phase is checked in the offline phase and not in the online phase.

4 From Tiny-OT aBit's to $[\![\cdot]\!]$-Sharings

At the heart of our MPC protocol is a method to translate from the two party aBits produced by the offline phase of the Tiny-OT protocol in [13], to the $[\![\cdot]\!]$-sharings under some global shared key α from Section 2. We note that the protocol to produce aBit's is the only sub-protocol from [13] which we use in this paper, and thus the more complex protocols in [13] for producing aOT's and aAND's we discard. We first deal with the underlying two party sub-protocols, and then we use these to define our multi-party protocols.

4.1 Two-party $[\cdot]$-representations.

Thus throughout we assume access to an ideal functionality $\mathcal{F}_{\mathsf{aBit}}$, given in Figure 3, that produces a substantially unbounded number of (oblivious) authenticated *random* bits for two parties, under some *randomly* chosen key δ_j known by one of the parties. This functionality can be implemented assuming a functionality $\mathcal{F}_{\mathsf{OT}}$ and using OT-extension techniques as in [13]. For ease of exposition we present the functionality as returning single bits for single requests. In practice the functionality is implemented via OT-extension and so one is able to obtain *many* aBits on each invocation of the functionality, for a given value of δ_j. Adapting our protocols to deal with multiple aBit production for a single random fixed δ_j chosen by the functionality is left to the reader[4].

The Functionality $\mathcal{F}_{\mathsf{aBit}}$

Authenticated Bit(P_i, P_j): This functionality selects a random $\delta_j \in \mathbb{F}$ and a random bit r, and returns a sharing $[r]^i_{\delta_j,j}$.

- On input (aBit, i, j) from honest P_i and P_j, the functionality samples a random δ_j and a random sharing $[r]^i_{\delta_j,j} = (r, \mu_i, \nu_j)$, such that $\mu_i = \nu_j + r \cdot \delta_j$. It then outputs $\{r, \mu_i\}$ to P_i and $\{\delta_j, \nu_j\}$ to P_j.
- If P_i is corrupted, the functionality waits for the environment to input the pair $\{r, \mu_i\}$ and it sets $\nu_j = \mu_i + r \cdot \delta_j$ for some randomly chosen δ_j, and $\{\delta_j, \nu_j\}$ is returned to party P_j.
- If P_j is corrupted, the functionality waits for the environment to input the pair $\{\delta_j, \nu_j\}$, r is selected at random and μ_i is set to be $\nu_j - r \cdot \delta_j$. The pair $\{r, \mu_i\}$ is returned to party P_i.

Figure 3 Two-party Bit Authentication [13]

Using the protocol $\Pi_{\mathsf{2\text{-}Share}}$, described in Protocol 4, we can obtain a "two-party" representation $[r]^i_{\delta_j,j}$ of a random bit known to P_i, under the key *chosen* by P_j. This extension is needed because we need to adapt the aBit command to the multi-party case. For example, if two parties, P_i and P_j, run the command (aBit, i, j), they obtain a random $[r]^i_{\delta'_j,j}$, with respect to δ'_j; when P_j calls (aBit, k, j) with a different party $P_k, k \neq j$, then they obtain a random $[s]^k_{\tilde{\delta}_j,j}$, with a different $\tilde{\delta}_j$. Thus allowing the parties to select their own values of δ_j means that we can obtain key-consistent $[\cdot]$-representations, in which each party P_j uses the same fixed δ_j. The security of the protocol

[4] Note, that in this situation we (say) produce $1,000,000$ aBits per invocation with a fixed random value of δ_j, then on the next invocation we obtain another $1,000,000$ aBits but with a new random δ_j value. This is not explicit in the ideal functionality description of aBit presented in [13], but is implied by their protocol.

The Subprotocol $\Pi_{\text{2-Share}}$

2Share$(i, j; \delta_j)$**:** On input $(\text{2-Share}, i, j, \delta_j)$, where P_j has $\delta_j \in \mathbb{F}$ as input, this command produces a $[r]^i_{\delta_j,j}$ sharing of a random bit r.

1. P_i and P_j call $\mathcal{F}_{\text{aBit}}$ on input (aBit, i, j): The box samples a random δ'_j and then produces

$$[r]^i_{\delta'_j,j} = (r, \mu'_i, \nu_j),$$

   such that $\mu'_i = \nu_j + r \cdot \delta'_j$, and outputs $\{r, \mu'_i\}$ to P_i and $\{\delta'_j, \nu_j\}$ to P_j.
2. P_j computes $\sigma_j = \delta_j + \delta'_j$ and sends σ_j to party P_i.
3. P_i sets $\mu_i = \mu'_i + r \cdot \sigma_j = \nu_j + r \cdot \delta_j$.

Protocol 4 Switching to Fixed δ-shares

$\Pi_{\text{2-Share}}$ follows from the security of the original aBit in [13]: intuitively the changes required to obtain a consistent $[\cdot]$-representation do not compromise security, because δ_j is one-time-padded with the random δ'_j produced by $\mathcal{F}_{\text{aBit}}$. See C.1 for details. Notice that the command 2-Share takes δ_j as the input of P_j. In particular the value δ_j may not be used to authenticate bits. Thus we could use the protocol $\Pi_{\text{2-Share}}$ to obtain a sharing of the *scalar product* $r \cdot \delta_j$, where P_i obtains the random bit r, and the other party decides what field element $\delta_j \in \mathbb{F}$ gets multiplied in. Then party P_i obtains the result μ_i masked by a one-time pad value ν_j known only to P_j. This application of the subprotocol $\Pi_{\text{2-Share}}$ is going to be crucial in our method to obtain authenticated OT's in our pre-processing phase. As a consequence we *do not always see* δ_j as an authentication key.

4.2 Multiparty $[\cdot]$-representation

The Functionality $\mathcal{F}_{\text{Bootstrap}}$

Let A be the indices of corrupt parties.

Initialize: On input (Init) from honest parties, the functionality activates and waits for the environment to input a set of shares $\{\delta_j\}_{j \in A}$. It samples random $\delta \in \mathbb{F}$ and prepares sharing $\langle \delta \rangle$, where the portions of honest shares are consistent with the adversarial shares, but otherwise random. If any $j \in A$ outputs abort, then the functionality aborts and returns the set of $j \in A$ which returned abort, otherwise it continues.

Share: On input (i, x, Share) from party P_i, and (i, Share) from all other parties. The functionality produces a representation $[x]^i_\delta = (\langle x \rangle^i, \langle \mu \rangle^i, \langle \nu \rangle^{\mathcal{P}})$, except that ν might be shifted by a value Δ_H, i.e. $\mu = x \cdot \delta + \nu + \Delta_H$, where Δ_H is an $\mathbb{F}_2$-linear combination of $\{\delta_k\}_{k \notin A}$, which is not known to the environment. It proceeds as follows:

- It samples random $\mu \in \mathbb{F}$. If $i \in A$ waits for the environment to input $\{\mu, x\}$.
- The functionality sets $\nu = x \cdot \delta + \mu$.
- The functionality waits for the environment to input shares $\{\nu_j\}_{j \in A}$, and prepares sharing $\langle \nu \rangle^{\mathcal{P}}$ consistent with the adversarial shares. The portion of honest shares are otherwise random.
- If the environment inputs shift-P_k, the functionality sets $\nu_k = \nu_k + \delta_k$, $k \notin A$.
- It outputs $\{\nu_k, \delta_k\}$ to honest P_k.

Figure 5 Ideal Generation of $[\cdot]^i_{\delta,\mathcal{P}}$-representations

Here we show how to generalize the $\Pi_{\text{2-Share}}$ protocol in order to obtain an n-party representation $[x]^i_\delta$ of a bit x chosen by P_i. This is what the functionality $\mathcal{F}_{\text{Bootstrap}}$ models in Figure 5. It bootstraps from a two party authentication to a multi-party authentication of the shared bit. As before for $\Pi_{\text{2-Share}}$, we can see the outputs of $\mathcal{F}_{\text{Bootstrap}}$ as the shares of scalar products $x \cdot \delta$, where one party P_i chooses the scalar (bit) x, but now the field element δ is unknown and additively shared among

all the parties. An interesting feature of this functionality is that the adversary can only influence *honest* outputs in a small way, that we model with the $\mathsf{shift\text{-}P_k}$ flag. Additionally, we can not prevent corrupt parties from outputting what they wish, this is reflected on the fact that the functionality leaves their outputs undefined. The main difference between this functionality and the equivalent in the SPDZ protocol [7], is that in [7] the functionality takes as input an offset known to the adversary who adjusts his shares to obtain an invalid MAC value by this linear amount. We do not model this in our functionality, instead we allow the adversary to choose his shares arbitrarily (which obtains the same effect). However, in our protocol the adversary can also introduce an unknown (to the adversary) error into the MAC values. In particular the adversary can decide whether to shift honest shares, but he cannot choose the shifting, namely, an element on the $\mathbb{F}_2$-span of secrets δ_k of honest parties P_k. Later, we manage to determine whether there are any errors (both adversarially known and unknown ones) using an *information-theoretic* $\mathsf{MACCheck}$ procedure that we borrow from [6]. See Appendix B for details.

The protocol $\Pi_{\mathsf{Bootstrap}}$, described in Protocol 6, realizes the ideal functionality $\mathcal{F}_{\mathsf{Bootstrap}}$ in a hybrid model in which we are given access to $\mathcal{F}_{\mathsf{aBit}}$. It permits to obtain $[x]^i_\delta$ and it is implemented by sending to each $P_j, j \neq i$, a mask of x using the random bits given by $\mathbf{2\text{-}Share}(i, j; \delta_j)$ as paddings, and then allowing P_j to adjust his share to the right value. In total the protocol needs to execute $n-1$ aBit per scalar product.

The Protocol $\Pi_{\mathsf{Bootstrap}}$

Initialize: Each party P_i samples a random δ_i. Define $\delta = \delta_1 + \cdots + \delta_n$.
Share: On input (i, x, Share) from P_i and (i, Share) from all other parties, do:

1. For each $j \neq i$, call $\Pi_{\mathsf{2\text{-}Share}}$ with $(\mathsf{2\text{-}Share}, i, j, \delta_j)$. Party P_i obtains $\{r_{i,j}, \mu_{i,j}\}_{j\neq i}$ whilst party P_j obtains $\nu_{i,j}$, such that $\mu_{i,j} = \nu_{i,j} + r_{i,j} \cdot \delta_j$.
2. Party P_i samples ϵ at random and sets $\mu_i = \epsilon + \sum_{j\neq i} \mu_{i,j}$ and $\nu_i = \epsilon + x \cdot \delta_i$.
3. Party P_i sends $d_j = x + r_{i,j}$ to party P_j for all $j \neq i$.
4. For $j \neq i$, P_j sets $\nu_j = \nu_{i,j} + d_j \cdot \delta_j$.
5. Output $\{\mu_i, \nu_i\}$ to P_i and $\{\nu_j\}$ to party P_j, for $j \neq i$. The system now has $[x]^i_\delta$.

Protocol 6 Transforming Two-party Representations onto $[\cdot]^i_{\delta,\mathcal{P}}$-representations

Lemma 1. *In the $\mathcal{F}_{\mathsf{aBit}}$-hybrid model, the protocol $\Pi_{\mathsf{Bootstrap}}$ implements $\mathcal{F}_{\mathsf{Bootstrap}}$ with perfect security against any static adversary corrupting up to $n-1$ parties.*

Proof. See Appendix C.1.

5 The Online Phase

In this section we present the protocol Π_{Online}, described in Protocol 7, which implements the online functionality in the $(\mathcal{F}_{\mathsf{Comm}}, \mathcal{F}_{\mathsf{Prep}})$-hybrid model. The basic idea behind our online phase is to use the set of GaOTs output in the offline phase to evaluate each multiplication gate. To see how this is done, consider that we want to multiply two authenticated bits $[\![a]\!], [\![b]\!]$. The parties take a GaOT tuple $\{[\![e]\!], [\![z]\!], [\![x_0]\!], [\![x_1]\!]\}$ off the pre-computed list. Recall we have for such tuples $z = x_e$. It is then relatively straightforward to compute authenticated shares of $[\![c]\!]$, where $c = a \cdot b$, as follows: First, the parties partially open $[\![f]\!] = [\![b]\!] + [\![e]\!]$ and $[\![g]\!] = [\![x_0]\!] + [\![x_1]\!] + [\![a]\!]$, and then set $[\![c]\!] = [\![x_0]\!] + f \cdot [\![a]\!] + g \cdot [\![e]\!] + [\![z]\!]$. To see why this is correct, note that since, $x_e + x_0 + e \cdot (x_0 + x_1) = 0$, we have $c = x_0 + (b + e) \cdot a + (x_0 + x_1 + a) \cdot e + z = a \cdot b$.

Protocol Π_{Online}

Initialize: The parties call Init on the $\mathcal{F}_{\mathsf{Prep}}$ functionality to get the shares α_i of the global MAC key α. If $\mathcal{F}_{\mathsf{Prep}}$ aborts outputting a set of corrupted parties, then the protocol returns this subset of A. Otherwise the operations specified below are performed according to the circuit.

Input: To share his input bit x, P_i calls $\mathcal{F}_{\mathsf{Prep}}$ with input (i, x, Share) and party P_j for $i \neq j$ calls $\mathcal{F}_{\mathsf{Prep}}$ with input (i, Share). The parties obtain $[\![x]\!]$ where the x-share of P_j is set to zero if $j \neq i$.

Add: On input $([\![a]\!], [\![b]\!])$, the parties locally compute $[\![a+b]\!] = [\![a]\!] + [\![b]\!]$.

Multiply: On input $([\![a]\!], [\![b]\!])$, the parties call $\mathcal{F}_{\mathsf{Prep}}$ on input (GaOT), obtaining a random GaOT tuple $\{[\![e]\!], [\![z]\!], [\![x_0]\!], [\![x_1]\!]\}$. The parties then perform:

1. The parties locally compute $[\![f]\!] = [\![b]\!] + [\![e]\!]$ and $[\![g]\!] = [\![x_0]\!] + [\![x_1]\!] + [\![a]\!]$.
2. The shares $[\![f]\!]$ and $[\![g]\!]$ are partially opened.
3. The parties locally compute
$$[\![c]\!] = [\![x_0]\!] + f \cdot [\![a]\!] + g \cdot [\![e]\!] + [\![z]\!].$$

Output: This procedure is entered once the parties have finished the circuit evaluation, but still the final output $[\![y]\!]$ has not been opened.

1. The parties call the protocol Π_{MACCheck} on input of all the partially opened values so far. If it fails, they output $\varnothing$ and abort. $\varnothing$ represents the fact that the corrupted parties remain undetected in this case.
2. The parties partially open $[\![y]\!]$ and call Π_{MACCheck} on input y to verify its MAC. If the check fails, they output $\varnothing$ and abort, otherwise they accept y as a valid output.

Protocol 7 Secure Function Evaluation in the $\mathcal{F}_{\mathsf{Comm}}, \mathcal{F}_{\mathsf{Prep}}$-hybrid Model

Theorem 1. *In the $(\mathcal{F}_{\mathsf{Comm}}, \mathcal{F}_{\mathsf{Prep}})$-hybrid model, the protocol Π_{Online} securely implements $\mathcal{F}_{\mathsf{Online}}$ against any static adversary corrupting up to $n-1$ parties, assuming protocol* MACCheck *utilizes a secure pseudo-random function* $\mathsf{PRF}_s^{\mathbb{F},t}(\cdot)$.

Proof. See Appendix C.2.

6 The Offline Phase

Here we present our offline protocol Π_{Prep} (Protocol 8). The key part of this protocol is the GaOT command. In [13] the authors give a two-party protocol to enable one party, say A, to obtain two authenticated bits e, z, and the other party, say B, to obtain two authenticated secret bits x_0, x_1, such that $z = x_e$ and e, x_0 and x_1 are chosen at random. We generalize such a procedure to many parties and we obtain sharings $[\![e]\!]$, $[\![z]\!]$, $[\![x_0]\!]$, $[\![x_1]\!]$, subject to $z = x_e$. Notice that the values e, z, x_0, x_1 are not known so they can be used in the online phase to implement multiplication gates.

The idea behind the GaOT command it is to exploit the relation between "affine functions" and "selector functions", in which a bit e selects one of two elements (χ_0, χ_1) in $\mathbb{F}$. This connection was already noted in [1] on the context of garbling arithmetic circuits via randomized encodings. Thus, on one hand we have authentications, that are essentially evaluations of affine functions, and on the other we have OT quadruples, that can be seen as selectors. Seeing both as the same object means that a way to authenticate bits also gives us a way to generate OTs, and the other way around. The procedure is broken into three steps, **Share OT**, **Authenticate OT** and **Sacrifice OT**. We examine these three stages in turn. To produce bit quadruples (e, z, x_0, x_1), such that $z = x_e$, the parties will use a (secret) affine line in $\mathbb{F}$ parametrized by (ϑ, η). Note that with our functionality $\mathcal{F}_{\mathsf{Bootstrap}}$ we get $[e_i]^i_\eta$, where e_i is known to P_i, and an additive sharing $\langle \eta \rangle$ is held by the system. We denote this concrete execution of the functionality as $\mathcal{F}_{\mathsf{Bootstrap}}(\eta)$, since we shall use fresh copies of $\mathcal{F}_{\mathsf{Bootstrap}}$ to generate more OT quadruples and also for authentication purposes. Note, that η is not an input to the functionality but a shared random value produced when initialising the functionality. Now, performing n independent queries of Share command on this copy $\mathcal{F}_{\mathsf{Bootstrap}}(\eta)$,

The Protocol Π_{Prep}

Let A be the set of indices of corrupt parties.

Initialize: On input (Init) from honest parties and adversary, the system runs a copy of $\mathcal{F}_{\mathsf{Bootstrap}}$ which is denoted $\mathcal{F}_{\mathsf{Bootstrap}}(\alpha)$. Then it calls Init on $\mathcal{F}_{\mathsf{Bootstrap}}(\alpha)$. If $\mathcal{F}_{\mathsf{Bootstrap}}(\alpha)$ aborts, outputting a set of corrupted parties, then the protocol returns this subset of A and aborts. Otherwise, the values δ_i returned by $\mathcal{F}_{\mathsf{Bootstrap}}(\alpha)$ are labelled as α_i. Set $\alpha = \alpha_1 + \cdots + \alpha_n$, and output α_i to honest parties P_i.

Share: On input (i, x, Share) from party i and (j, Share) from all parties $j \neq i$. The protocol calls Share command of $\mathcal{F}_{\mathsf{Bootstrap}}(\alpha)$ to obtain $[x]^i_\alpha$, given by $\{\langle\mu\rangle^i, \langle\nu\rangle^{\mathcal{P}}\}$. Then, for $j \neq i$, party P_j sets his share of x to be zero, and $\mu_j(x) = \nu_j$. Party P_i sets $\mu_i(x) = \mu + \nu_i$. Thus, the parties obtain $[\![x]\!]$.

GaOT: On input (GaOT) from all P_i, execute the following sub-procedures:

Share OT. This generates sharings $(\langle e\rangle, \langle z\rangle, \langle x_0\rangle, \langle x_1\rangle)$ such that x_0, x_1 and e are random bits. If all parties are honest then it holds $z = x_e$.

1. The system runs a fresh copy of $\mathcal{F}_{\mathsf{Bootstrap}}$ on Init command getting an additive sharing $\langle\eta\rangle$ for some random $\eta \in \mathbb{F}$. Denote this copy as $\mathcal{F}_{\mathsf{Bootstrap}}(\eta)$.
2. Each party samples a random bit e_i. Define $e = e_1 + \cdots + e_n$.
3. For each $i = 1, \ldots, n$, the system calls $\mathcal{F}_{\mathsf{Bootstrap}}(\eta)$ on input (i, e_i, Share) from party P_i and input (i, Share) from any other P_j, to obtain $[e_i]^i_\eta$. That is, (in an honest execution) P_i gets $\zeta_i \in \mathbb{F}$, and the parties gets an additive sharing $\langle\vartheta_i\rangle$ of some unknown $\vartheta_i \in \mathbb{F}$, such that $\zeta_i = \vartheta_i + e_i \cdot \eta$. The parties compute $[e]^{\mathcal{P}}_\eta = [e_1]^1_\eta + \cdots + [e_n]^n_\eta$.
4. At this point of the protocol, the system holds sharings $\langle e\rangle$, $\langle\zeta\rangle$, $\langle\vartheta\rangle$, $\langle\eta\rangle$, so it can derive $\langle\chi_0\rangle = \langle\vartheta\rangle$, and $\langle\chi_1\rangle = \langle\vartheta\rangle + \langle\eta\rangle$. Note that (for an honest execution) $\zeta = \vartheta + e \cdot \eta$, or in other words $\zeta = \chi_e$.
5. Each party P_i sets z_i, $x_{0,i}$, $x_{1,i}$ to be the least significant bits of ζ_i, $\chi_{0,i}$, $\chi_{1,i}$ respectively, so as to obtain sharings $\langle z\rangle, \langle x_0\rangle$ and $\langle x_1\rangle$.

Authenticate OT. This step produces authentications on the bits previously computed.
For every bit $y \in \{e, z, x_0, x_1\}$ it does the following:

6. Call $\mathcal{F}_{\mathsf{Bootstrap}}(\alpha)$ on input (i, y_i, Share) from P_i and (j, Share) for party P_j to obtain $[y_i]^i_\alpha$.
7. Compute $[\![y]\!]$ by forming $\sum_{i\in\mathcal{P}}[y_i]^i_\alpha$, and then subtracting $\nu(y)$ from $\mu(y)$.

Sacrifice OT. This step checks that the authenticated OT-quadruples are correct. Let $[\![e]\!], [\![z]\!], [\![x_0]\!], [\![x_1]\!]$, be the quadruple to check, and κ a security parameter:

8. Every party P_i samples a seed s_i and asks $\mathcal{F}_{\mathsf{Comm}}$ to broadcast $\tau_i = \mathsf{Comm}(s_i)$.
9. Every party P_i calls $\mathcal{F}_{\mathsf{Comm}}$ with $\mathsf{Open}(\tau_i)$ and all parties obtain s_j for all j. Set $s = s_1 + \cdots + s_n$.
10. Parties sample a random vector $\mathbf{t} = \mathsf{PRF}^{\mathbb{F}_2,\kappa}_s(0) \in \mathbb{F}_2^\kappa$. Note all parties obtain the same vector as they have agreed on the seed s.
11. For $i = 1, \ldots, \kappa$, repeat the following:
    - Take one fresh quadruple $[\![e_i]\!], [\![z_i]\!], [\![x_{0,i}]\!], [\![x_{1,i}]\!]$, and partially open the values $p_i = t_i \cdot ([\![x_0]\!] + [\![x_1]\!]) + [\![x_{0,i}]\!] + [\![x_{1,i}]\!]$ and $q_i = [\![e]\!] + [\![e_i]\!]$.
    - Locally evaluate c_i such that

$$[\![c_i]\!] = t_i \cdot ([\![z]\!] + [\![x_0]\!]) + [\![z_i]\!] + [\![x_{0,i}]\!] + p_i \cdot [\![e]\!] + q_i \cdot ([\![x_{0,i}]\!] + [\![x_{1,i}]\!]),$$

    and check it partially opens to zero. If it does not, then abort.
12. The parties call Π_{MACCheck} on the values partially opened in step 11.
13. If no abort occurs, output $[\![e]\!], [\![z]\!], [\![x_0]\!], [\![x_1]\!]$ as a valid quadruple.

Protocol 8 Preprocessing: Input Sharing and Creation of OT Quadruples in the $\mathcal{F}_{\mathsf{Bootstrap}}$-hybrid Model

the parties can generate

$$[e]^{\mathcal{P}}_\eta = [e_1]^1_\eta + \cdots + [e_n]^n_\eta. \tag{1}$$

Thus, the system obtains two (secret) elements $\langle e\rangle$, $\langle\zeta\rangle$, such that $\zeta = \vartheta + e \cdot \eta$, for line $(\langle\vartheta\rangle, \langle\eta\rangle)$. Define $\chi_0 = \vartheta$ and $\chi_1 = \vartheta + \eta$, so it holds $\zeta = \chi_e$. The quadruple (e, z, x_0, x_1) is then given by the least significant bits of the corresponding field elements $(e, \zeta, \chi_0, \chi_1)$. This conclude the **Share OT** step.

To add MACs to each bit of the quadruple that the parties just generated, the protocol uses the $\mathcal{F}_{\mathsf{Bootstrap}}(\alpha)$ instance to obtain a sharing $\langle\alpha\rangle$ of the global key. Each party can now authenticate his shares of (e, z, x_0, x_1) querying Share command and obtaining $[\![e]\!]$, $[\![z]\!]$, $[\![x_0]\!]$, $[\![x_1]\!]$. We emphasize

that the same α is used to authenticate all OT quadruples, thus $\mathcal{F}_{\mathsf{Bootstrap}}(\alpha)$ is fixed once and for all.

After the **Authenticate OT** step the parties have sharings $[\![e]\!]$, $[\![z]\!]$, $[\![x_0]\!], [\![x_1]\!]$, which could suffer from two possible errors induced by the corrupted parties: Firstly the algebraic equation $z = x_e$ may not hold, and second the MAC values may be inconsistent. For the latter problem we will check all the partially opened values using the MACCheck procedure at the end of the offline phase. For the former case we use the **Sacrifice OT** step. We use the same methodology as in [4,7,6], i.e. one quadruple is checked by "sacrificing" another quadruple. The idea involving sacrificing can be seen as follows: We associate to each pair of quadruples a polynomial $S(t)$ over the field of secrets ($\mathbb{F}_2$ in our case), which is the zero polynomial only if both quadruples are correct. Thus, proving correctness of quadruples is equivalent to proving that $S(t)$ is the zero polynomial. This is done by securely evaluating $S(t)$ on a random public challenge bit t via a combination of addition gates and two openings (plus one extra opening to check the evaluation), and then checking that the result of the evaluation partially opens to zero. In this way we would waste κ quadruples to check one quadruple, to get security of $2^{-\kappa}$; we refer the reader to Appendix A for a more efficient sacrifice procedure.

Theorem 2. *Let κ be the security parameter and $t \in \mathbb{N}$. In the $(\mathcal{F}_{\mathsf{Comm}}, \mathcal{F}_{\mathsf{Bootstrap}})$-hybrid model, the protocol Π_{Prep} securely implements $\mathcal{F}_{\mathsf{Prep}}$ with statistical security on κ against any static adversary corrupting up to $n-1$ parties, assuming the existence of $\mathsf{PRF}_s^{X,m}(\cdot)$ with domain $X = \mathbb{F}$ (resp. $\mathbb{F}_2$) and $m = t$ (resp. κ).*

Proof. See Appendix C.3

7 Efficiency Analysis

As it stands our protocol is not that efficient, mainly due to the naive sacrificing step performed in the offline phase so as to check the GaOTs for correctness. In Appendix A we present a much more efficient sacrifice step, which for reasonable parameters means that the ratio of required GaOT's for each used one can be between four and six. Let this ratio be denoted r.

We examine the cost of a multiplication in terms of the number of aBits required in the case of two parties. We notice that each GaOT requires us to consume ten aBits; we need to execute the **Share OT** step to determine e, z, x_0, x_1 (which requires one aBit consumption per player, i.e. two in total when $n = 2$); in addition each of these four bits needs to be authenticated in **Authenticate OT** in Protocol 8 (which again requires one aBit consumption per player, i.e. eight in total when $n = 2$). Since we need one checked GaOT to perform a secure multiplication, and we sacrifice $r - 1$ GaOT to obtain a checked one; this means we require $r \cdot 10$ aBits per secure multiplication in the two party case. Depending on the parameters we use for our sacrifice step in Appendix A, this equates to 40, 50 or 60 aBits per secure multiplication, setting $t = 2^{20}, 2^{14}, 2^{10}$, respectively, and an error probability of 2^{-40}.

We now compare this to the number of aBits needed in the Tiny-OT protocol [13]. In this protocol each secure multiplication requires two aBits, two aANDs and two aOTs. Assuming a bucket size T in the protocols to generate aANDs and aOTs; each aAND (resp. aOT) requires four LaANDs (resp LaOTs). Each LaAND requires four aBits and each LaOT requires three aBits. Thus the total number of aBits per secure multiplication is $2 \cdot (1 + T \cdot 4 + T \cdot 3) = 14 \cdot T + 2$. To achieve the same error probability of 2^{-40}, with same values of $t = 2^{20}, 2^{14}, 2^{10}$, they need $44, 58$ and 72

aBits, respectively. We see therefore that we can make our protocol (in the two party case) more efficient than the Tiny-OT protocol, when we measure efficiency in terms of the number of aBits consumed.

8 Acknowledgements

This work has been supported in part by ERC Advanced Grant ERC-2010-AdG-267188-CRIPTO, by EPSRC via grant EP/I03126X and by research sponsored by Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under agreement number FA8750-11-2-0079[5].

References

1. B. Applebaum, Y. Ishai, and E. Kushilevitz. How to garble arithmetic circuits. In R. Ostrovsky, editor, *FOCS*, pages 120–129. IEEE, 2011.
2. G. Asharov, Y. Lindell, T. Schneider, and M. Zohner. More efficient oblivious transfer and extensions for faster secure computation. In A.-R. Sadeghi, V. D. Gligor, and M. Yung, editors, *ACM Conference on Computer and Communications Security*, pages 535–548. ACM, 2013.
3. D. Beaver. Efficient multiparty protocols using circuit randomization. In J. Feigenbaum, editor, *CRYPTO*, volume 576 of *Lecture Notes in Computer Science*, pages 420–432. Springer, 1991.
4. R. Bendlin, I. Damgård, C. Orlandi, and S. Zakarias. Semi-homomorphic encryption and multiparty computation. In K. G. Paterson, editor, *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Science*, pages 169–188. Springer, 2011.
5. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145. IEEE Computer Society, 2001.
6. I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart. Practical covertly secure mpc for dishonest majority - or: Breaking the spdz limits. In J. Crampton, S. Jajodia, and K. Mayes, editors, *ESORICS*, volume 8134 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2013.
7. I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In Safavi-Naini and Canetti [15], pages 643–662.
8. I. Damgård and S. Zakarias. Constant-overhead secure computation of boolean circuits using preprocessing. In A. Sahai, editor, *TCC*, volume 7785 of *Lecture Notes in Computer Science*, pages 621–641. Springer, 2013.
9. T. K. Frederiksen, T. P. Jakobsen, J. B. Nielsen, P. S. Nordholt, and C. Orlandi. Minilego: Efficient secure two-party computation from general assumptions. In T. Johansson and P. Q. Nguyen, editors, *EUROCRYPT*, volume 7881 of *Lecture Notes in Computer Science*, pages 537–556. Springer, 2013.
10. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In A. V. Aho, editor, *STOC*, pages 218–229. ACM, 1987.
11. D. Harnik, Y. Ishai, and E. Kushilevitz. How many oblivious transfers are needed for secure multiparty computation? In A. Menezes, editor, *CRYPTO*, volume 4622 of *Lecture Notes in Computer Science*, pages 284–302. Springer, 2007.
12. Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In D. Boneh, editor, *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 145–161. Springer, 2003.
13. J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra. A new approach to practical active-secure two-party computation. In Safavi-Naini and Canetti [15], pages 681–700.
14. J. B. Nielsen and C. Orlandi. Lego for two-party secure computation. In *TCC*, pages 368–386, 2009.
15. R. Safavi-Naini and R. Canetti, editors. *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*. Springer, 2012.

[5] The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

A Batching the Sacrifice Step

This technique (an adaptation of a technique to be found originally in [14,6,9]) permits to check a batch of OT quadruples for algebraic correctness using a *smaller* number of "sacrificed" quadruples than the basic version we described in Section 6. Recall, the idea is to check that an authenticated OT-quadruple $\mathsf{GaOT}_i = (\llbracket e_i \rrbracket, \llbracket z_i \rrbracket, \llbracket x_i \rrbracket, \llbracket y_i \rrbracket)$ verifies the "multiplicative" relation $m_i = z_i + x_i + e_i \cdot (x_i + y_i) = 0$.

At a high level, Protocol 9 essentially consists of two different phases. Let $(\mathsf{GaOT}_1, \ldots, \mathsf{GaOT}_N)$ be a set of OT quadruples, in the first phase a fixed portion of these GaOTs are partially opened as in a classical cut-and-choose step. If any of the opened OT quadruples does not satisfy the multiplicative relation the protocol aborts. Otherwise it runs the second phase: the remaining GaOTs are permuted and uniformly distributed into t buckets of size T. Then, for each of the buckets, the protocol selects a BucketHead, i.e. the first (in the lex order) GaOT in the bucket (as in [9]), and uses the remaining GaOTs in the same bucket to check that BucketHead correctly satisfies the multiplicative relation.

We call CheckGaOTs the GaOTs used to check the BucketHead, and we denote them by $\mathsf{CheckGaOT} = (\llbracket \mathfrak{e} \rrbracket, \llbracket \mathfrak{z} \rrbracket, \llbracket \mathfrak{h} \rrbracket, \llbracket \mathfrak{g} \rrbracket)$, with $\mathfrak{z} = \mathfrak{h} + \mathfrak{e}(\mathfrak{h} + \mathfrak{g})$.

If any BucketHead does not pass the test, then we know that some parties are corrupted and the protocol aborts. If all the checks pass then we obtain t algebraically correct BucketHeads, i.e. t OT quadruples, with overwhelming probability.

Bucket Cut-and-Choose Protocol

Input : Let $N = (T + h) \cdot t$ be the number of input GaOTs and T the size of the buckets, with $T \geq 2$. We let $1 \leq h \leq T$ denote an additional parameter controlling how much cut-and-choose we perform.

Phase-I *Cut-And-Choose* :

1. Every P_i samples a seed s_i and asks $\mathcal{F}_{\mathsf{Comm}}$ to broadcast $\tau_i = \mathsf{Comm}(s_i)$.
2. Every party P_i calls $\mathcal{F}_{\mathsf{Comm}}$ with $\mathsf{Open}(\tau_i)$ and all parties obtain s_j for all j. Set $s = s_1 + \cdots + s_n$.
3. Using a $\mathsf{PRF}_s^{\mathbb{F}_2, N}$, parties sample a random vector $\mathbf{v} \in \mathbb{F}_2^N$, such that the number of its non-zero entries is $h \cdot t$ (i.e. the Hamming weight of $\mathbf{v}$ is $h \cdot t$).
4. Let $\mathcal{J}$ be the set of indices j such that $v_j \neq 0$, and, $\forall j \in \mathcal{J}$, the parties partially open GaOT_j and check that it satisfies the algebraic relation $z_j + x_j = e_j \cdot (x_j + y_j)$. If there exists an algebraically incorrect GaOT_j quadruple, then the protocol aborts.

Phase-II *Bucket-Sacrifice* :

5. Permute the unopened GaOTs according to a random permutation π on $T \cdot t$ indices, again using a PRF_s. Then renumber the permuted unopened GaOT_j, such that $j = 1, \ldots, T \cdot t$, and, for $i = 1, \ldots, t$, create the ith bucket as $\{\mathsf{GaOT}_j\}_{j=iT-T+1}^{iT}$.
6. Parties compute a $\mathsf{BucketHead}(i)$ for each $i = 1, \ldots, t$, i.e. return the first (in the lex order) element in the ith bucket.
7. For $i = 1, \ldots, t$, parties check that $\mathsf{BucketHead}(i) = \mathsf{GaOT}_i = (\llbracket e_i \rrbracket, \llbracket z_i \rrbracket, \llbracket x_i \rrbracket, \llbracket y_i \rrbracket)$ is correct using the other GaOTs in the bucket: For $j = iT - T + 2, \ldots, iT$ do
   - Set $\mathsf{CheckGaOT}_j = \mathsf{GaOT}_j = (\llbracket \mathfrak{e}_j \rrbracket, \llbracket \mathfrak{z}_j \rrbracket, \llbracket \mathfrak{h}_j \rrbracket, \llbracket \mathfrak{g}_j \rrbracket)$.
   - Parties open $\langle e_i + \mathfrak{e}_j \rangle$ and $\langle x_i + y_i + \mathfrak{h}_j + \mathfrak{g}_j \rangle$.
   - Parties locally compute

$$\llbracket c_{i,j} \rrbracket = \llbracket z_i + x_i \rrbracket + \llbracket \mathfrak{z}_j + \mathfrak{h}_j \rrbracket + (e_i + \mathfrak{e}_j) \llbracket \mathfrak{h}_j + \mathfrak{g}_j \rrbracket + (x_i + y_i + \mathfrak{h}_j + \mathfrak{g}_j) \llbracket e_i \rrbracket,$$

     and check it partially opens to zero.
   - If all checks go through output GaOT_i as valid quadruples; otherwise abort.
8. The parties execute the protocol Π_{MACCheck} to check all partially opened values.

Protocol 9 Bucket Cut-and-Choose Protocol

Theorem 3. *For $T \geq \frac{\kappa+\log_2(t)}{\log_2(t)}$ the previous protocol provide t correct* GaOT*s with error probability $2^{-\kappa}$.*

Proof. It is easy to check that the protocol is correct and secure in the semi-honest model, i.e. if all the OT quadruples are honestly generated, according to the GaOT command in Π_{Prep}, then $c_i = 0, \forall i$.

The argument for active security is as follows. A badGaOT, i.e. a OT quadruple which does not satisfy the multiplicative relation, passes the test if and only if all the partially opened GaOTs in the cut-and-choose phase are correct and then it ends up in a bucket containing only badGaOTs. This is because if we combine two badGaOTs, say GaOT_i and GaOT_j, we obtain $c_{i,j} = m_i + m_j = 1+1 = 0$, and the test passes. We show that this happens with negligible probability with an appropriate choice of the parameters. We argue this in two steps: first we prove that when a bucket contains at least one goodGaOT (a OT satisfying the multiplicative relation) a badGaOT will be always detected, and then we bound the probability of having buckets containing only badGaOTs.

If parties misbehaved in any previous step yielding a $\mathsf{badGaOT}_i$, when we combine it with a $\mathsf{goodGaOT}_j$, then $c_{i,j} = m_i + m_j = 1$ and the check fails. Notice that the protocol always abort if there is a bucket with both bad and good GaOTs. More precisely the protocol checks the algebraic correctness of the BucketHeads, but indirectly also that of any other GaOTs (We use the BucketHead notation so that each GaOT is only once paired with a different GaOT).
Let

- PassICheck be the event that the protocol does not abort in the cut-and-choose step
- $m$badGaOT be the event that m GaOTs are bad. Note that we fix m here.
- NoMixedBucket the event that there are no buckets containing both goodGaOTs and badGaOTs.

We bound the probability that both PassICheck and NoMixedBucket occur. To do this we prove:

1. $\Pr[E_1] = \Pr[\mathsf{PassICheck} \wedge m\mathsf{badGaOT}] \leq (\frac{T}{T+h})^m$.
2. $\Pr[E_2] = \Pr[E_1 \wedge \mathsf{NoMixedBucket}] \leq 2^{(\log_2(t))(1-T)}$.

The first point is straightforward. First note that if $m > h \cdot t$ then $\Pr[\mathsf{PassICheck}] = 0$ and the protocol aborts; similarly, if $m < T$, then $\Pr[\mathsf{NoMixedBucket}] = 0$, so we can suppose $T \leq m \leq h \cdot t$ (in particular $m > 1$). Moreover as a bad BucketHead will be always detected if a bucket contains both good and bad GaOTs, we add the condition $m = k \cdot T$, $k = 1, \ldots, t$. In this way if m denotes the number of badGaOTs, and PassICheck is true, then the $h \cdot t$ GaOTs that are opened in the cut-and-choose step are sampled from the $N - m$ good GaOTs. It holds:

$$\begin{aligned}
\Pr[E_1] = & \binom{N-m}{h\cdot t} \cdot \binom{N}{h\cdot t}^{-1} = \binom{(T+h)\cdot t - m}{T\cdot t - m} \cdot \binom{(T+h)\cdot t}{T\cdot t}^{-1} = \\
= & \frac{((T+h)\cdot t - m)\cdots(h\cdot t+1)\cdot(Tt)!}{(T\cdot t + h\cdot t)\cdots(ht+1)(Tt-m)!} = \frac{(T\cdot t)\cdots(Tt-m+1)\cdot(Tt-m)!}{(Tt+ht)\cdots(Tt+ht-m+1)\cdot(Tt-m)!} \leq \\
\leq & \left(\frac{Tt}{Tt+ht}\right)^m = \left(\frac{T}{T+h}\right)^m.
\end{aligned}$$

Now we compute the probability of NoMixedBucket $\wedge E_1$. Recall that the cardinality of each of the t buckets is T and that we are assuming $m = k \cdot T$ bad GaOTs. It is easy to see that

$$\Pr[E_2] \leq \left(\frac{T}{T+h}\right)^{kT} \cdot \binom{t}{k} \cdot \binom{Tt}{k\cdot T}^{-1}. \tag{2}$$

This probability is maximized in $k = 1$. Intuitively we can see this as follows: the term $\binom{t}{k} \cdot \binom{Tt}{k \cdot T}^{-1}$ is symmetric with respect to the value $k = t/2$, as $\binom{t}{k} \cdot \binom{Tt}{k \cdot T}^{-1} = \binom{t}{t-k} \cdot \binom{Tt}{Tt-k \cdot T}^{-1}$, $k = 1, \ldots, t-1$, and it strictly decreases for $1 \leq k \leq t/2$, with minimum in $k = t/2$; the term $\left(\frac{T}{T+h}\right)^{kT}$ is less than 1 and it strictly decreases when k grows. So when we multiply the two terms we have that the above probability for values of k in $[1, \ldots, t/2[$ is bigger than the same probability for "symmetric" values in $]t/2, \ldots, t[$. Finally, for $k = t$, $\Pr[E_2]_{|k=t} = (\frac{T}{T+h})^{t \cdot T}$ and, for big values of t, this probability is less that $\Pr[E_2]_{|k=1}$.

By substituting the value $k = 1$ in (2) we get:

$$\begin{aligned}
\Pr[E_2] &= \left(\frac{T}{T+h}\right)^T \cdot t \cdot \binom{Tt}{T}^{-1} \\
&\leq \left(\frac{T}{T+h}\right)^T \cdot t^{(1-T)} = 2^{(\log_2(t))(1-T)+T(\log_2(T/(T+h))} \\
&\leq 2^{(\log_2(t))(1-T)}
\end{aligned}$$

Thus for $T \geq \frac{\kappa+\log_2(t)}{\log_2(t)}$ we obtain $\Pr[E_2] \leq 2^{-\kappa}$.

□

We can replace the **Sacrifice OT** step in Π_{Prep} with the above Bucket-Cut-and-Choose Protocol and, for an appropriate choice of the parameters, Theorem 2 (and relative proof) still holds.

Notice, how the value h has little effect on the final probability (we suppressed the effect in the statement of the Theorem since it is so low). This means we can take $h = 1$ to obtain the most efficient protocol, which means the amount of cut-and-choose performed is relatively low.

To measure the efficiency of this protocol we can consider the ratio $r = \frac{(T+h) \cdot t}{t} = T + h$: it measures the number of GaOTs that we need to produce one actively secure OT quadruple. Setting $h = 1$ and an error probability of 2^{-40}, we obtain Table 1 for different values of $t = 2^{10}, 2^{14}, 2^{20}$.

Table 1 Number of GaOTs we need to check t quadruples

r	$T = r - h$	t	$\frac{40+\log_2(t)}{\log_2(t)}$
4	3	2^{20}	3
5	4	2^{14}	3.85
6	5	2^{10}	5

B Information Theoretic Tags for Dishonest Majority

In the online phase, parties work with representations with *information-theoretic* message authentication codes. The key properties of the MACs is that are homomorphic, and hold enough entropy to convince an honest party that local computation has been done correctly. The homomorphic property allows us to postpone the check of the correctness in the MACs until the very end of the circuit evaluation (where the circuit can be the one implicitly used in the preprocessing or the target online circuit). In [6] it was shown how to do the check on partially open values whilst keeping secret the key, hence enabling support for reactive online evaluations, and this is the one we use.

Protocol Π_{MACCheck}

Usage: The parties have a set of $[\![a_i]\!]$, sharings and public bits b_i, for $i = 1, \ldots, t$, and they wish to check that $a_i = b_i$, i.e. they want to check whether the public values are consistent with the shared MACs held by the parties.
As input the system has sharings $\big(\langle\alpha\rangle, \{b_i, \langle a_i\rangle, \langle\mu(a_i)\rangle\}_{i=1}^{t}\big)$. If the MAC values are correct then we have that $\mu(a_i) = b_i \cdot \alpha$, for all i.

$\mathsf{MACCheck}(\{b_1, \ldots, b_t\})$**:**

1. Every party P_i samples a seed s_i and asks $\mathcal{F}_{\mathsf{Comm}}$ to broadcast $\tau_i = \mathsf{Comm}(s_i)$.
2. Every party P_i calls $\mathcal{F}_{\mathsf{Comm}}$ with $\mathsf{Open}(\tau_i)$ and all parties obtain s_j for all j.
3. Set $s = s_1 + \cdots + s_n$.
4. Parties sample a random vector $\boldsymbol{\chi} = \mathsf{PRF}_s^{\mathbb{F},t}(0) \in \mathbb{F}^t$; note all parties obtain the same vector as they have agreed on the seed s.
5. Each party computes the public value $b = \sum_{i=1}^{t} \chi_i \cdot b_i \in \mathbb{F}$.
6. The parties locally compute the sharings $\langle\mu(a)\rangle = \chi_1 \cdot \langle\mu(a_1)\rangle + \cdots + \chi_t \cdot \langle\mu(a_t)\rangle$ and $\langle\sigma\rangle = \langle\mu(a)\rangle - b \cdot \langle\alpha\rangle$.
7. Party i asks $\mathcal{F}_{\mathsf{Comm}}$ to broadcast his share $\tau_i' = \mathsf{Comm}(\sigma_i)$.
8. Every party calls $\mathcal{F}_{\mathsf{Comm}}$ with $\mathsf{Open}(\tau_i')$, and all parties obtain σ_j for all j.
9. If $\sigma_1 + \cdots + \sigma_n \neq 0$, the parties output $\varnothing$ and abort, otherwise they accept all b_i as valid authenticated bits.

Protocol 10 Method to Check MACs on Partially Opened Values

See Protocol 10 for details. The procedure utilizes an ideal functionality $\mathcal{F}_{\mathsf{Comm}}$ for commitments given in Figure 11. An implementation of $\mathcal{F}_{\mathsf{Comm}}$ in the random oracle model can be found in the Appendix of [6].

Game: Security of the MACCheck procedure assuming pseudorandom functions

1: The challenger samples random sharing $\langle\alpha\rangle \in \mathbb{F}$. It sets $\langle\mu(a_i)\rangle = a_i \cdot \langle\alpha\rangle$ and sends bits $a_1, \ldots, a_t$ to the adversary.
2: The adversary sends back bits $b_1, \ldots, b_t$.
3: The challenger generates random values $\chi_1, \ldots, \chi_t \in \mathbb{F}$ and sends them to the adversary.
4: The adversary provides an error $\Delta \in \mathbb{F}$.
5: Set $b = \sum_{i=1}^{t} \chi_i \cdot b_i$, and sharings $\langle\mu(a)\rangle = \sum_{i=0}^{t} \chi_i \cdot \langle\mu(a_i)\rangle$, and $\langle\sigma\rangle = \langle\mu(a)\rangle - b \cdot \langle\alpha\rangle$. The challenger checks that $\sigma = \Delta$.

In order to understand the probability of an adversary being able to cheat during the execution of Protocol 10, the authors in [6] used a security game approach, which in turn was an adaptation of the one in [7]. For completeness, we state here both the protocol and the security game.

The adversary wins the game if there is an $i \in \{1, \ldots, t\}$ for which $b_i \neq a_i$, and the check goes through. The second step in the game, where the adversary sends the b_i's, models the fact that corrupted parties can choose to lie about their shares of values opened on the execution of the parent protocol. The offset Δ models the fact that the adversary is allowed to introduce errors on the MACs. A formal proof of Theorem 4 can be found in the Appendix of [7,6].

Theorem 4 ([6]). *The protocol* $\mathsf{MACCheck}$ *is correct, i.e. it accepts if all the public values* b_i*, and the corresponding MACs are correctly computed. Moreover, it is sound, i.e. it rejects except with probability* $\frac{2}{|\mathbb{F}|}$ *in case at least one value, or MAC, is not correctly computed.*

C Security Proofs

C.1 Proof of the Bootstrap Step (Lemma 1)

We show that an environment $\mathcal{Z}$ corrupting up to $n-1$ parties, playing with $\Pi_{\mathsf{Bootstrap}}$ attached to $\mathcal{F}_{\mathsf{aBit}}$ or with the simulator $\mathcal{S}$ attached to $\mathcal{F}_{\mathsf{Bootstrap}}$, sees transcripts that are identically distributed.

The Functionality $\mathcal{F}_{\mathsf{Comm}}$

Commit: On input $(\mathsf{Comm}, v, i, \tau_v)$ by P_i or the adversary on his behalf (if P_i is corrupt), where v is either in a specific domain or $\perp$, it stores (v, i, τ_v) on a list and outputs (i, τ_v) to all parties and adversary.

Open: On input $(\mathsf{Open}, i, \tau_v)$ by P_i or the adversary on his behalf (if P_i is corrupt), the ideal functionality outputs (v, i, τ_v) to all parties and adversary. If $(\mathsf{NoOpen}, i, \tau_v)$ is given by the adversary, and P_i is corrupt, the functionality outputs $(\perp, i, \tau_v)$ to all parties.

Figure 11 Ideal Commitments

We assume *authenticated* communication between parties, that is, they are given access to a functionality $\mathcal{F}_{\mathsf{AT}}$, which on input (m, s, s') from P_s, it gives message m to $P_{s'}$ and also leak it to $\mathcal{Z}$. In a nutshell, the simulator runs a copy of $\Pi_{\mathsf{Bootstrap}}$ acting on behalf of honest parties. Let A be the set of indices of corrupted parties, parties in A are indexed with j, and parties not in A with k.

We start describing the behaviour of $\mathcal{S}$. The corruption is static, so we can distinguish the two cases:

a) P_i **is honest.**

1. In step 1, for $s \in \mathcal{P}$, $\mathcal{S}$ engages in a run of $\Pi_{\mathsf{2\text{-}Share}}(\mathsf{2\text{-}Share}, i, s, \delta_s)$ with $\mathcal{Z}$, acting on behalf of P_i and honest P_k: It sets an internal copy of $\mathcal{F}_{\mathsf{aBit}}$ to generate representations $[r_{i,j}]^i_{\delta'_j}$ on dummy bits $r_{i,j}$. It answers queries from $\mathcal{Z}$ by sending him $\{\nu_{i,j}, \delta'_j\}_{j\in A}$. $\mathcal{S}$ also gives random σ_k to $\mathcal{Z}$, for $k \notin A$, and gets back σ^*_j for $j \in A$ (acting as $\mathcal{F}_{\mathsf{AT}}$). It then sets $\delta^*_j = \sigma^*_j + \delta'_j$.
2. $\mathcal{S}$ sends $\{\delta^*_j\}_{j\in A}$ to $\mathcal{F}_{\mathsf{Bootstrap}}$ as part of **Initialize**.
3. In step 3, $\mathcal{S}$ acting as $\mathcal{F}_{\mathsf{AT}}$ gives random d_s to $\mathcal{Z}$, $\forall s \neq i$. Note that $\nu^*_j = \nu_{i,j} + d_j \cdot \delta^*_j$ is the purported share that corrupt P_j should come up with.
4. $\mathcal{S}$ sends $\{\nu^*_j\}_{j\in A}$ to $\mathcal{F}_{\mathsf{Bootstrap}}$.

b) P_i **is dishonest** ($\mathcal{Z}$ specifies input bit x).

1. In step 1, for $s \in \mathcal{P}$, $\mathcal{S}$ engages in a run of $\Pi_{\mathsf{2\text{-}Share}}(\mathsf{2\text{-}Share}, i, s, \delta_s)$ with $\mathcal{Z}$, acting on behalf of honest P_k. It sets an internal copy of $\mathcal{F}_{\mathsf{aBit}}$ to generate representations $[r_{i,s}]^i_{\delta'_s}$ on dummy bits $r_{i,s}$, for $s \neq i$. $\mathcal{S}$ answers queries from $\mathcal{Z}$ by sending him $\{r_{i,s}, \mu'_{i,s}\}_{s\in\mathcal{P}}$, and $\{\nu_{i,j}, \delta'_j\}_{j\in A}$. Acting as $\mathcal{F}_{\mathsf{AT}}$, $\mathcal{S}$ gives random σ_k to $\mathcal{Z}$ and it gets back σ^*_j. It then sets corrupt $\delta^*_j = \sigma^*_j + \delta'_j$. $\mathcal{S}$ also extracts $\nu^*_j = \nu_{i,j} + (x + r_{i,j}) \cdot \delta^*_j$, for $j \in A$, and $\mu_i = \sum_{s\neq i} \mu_{i,s}$ and $\nu^*_i = x \cdot \delta^*_i$.
2. $\mathcal{S}$ sends $\{\delta^*_j\}_{j\in A}$ to $\mathcal{F}_{\mathsf{Bootstrap}}$ as part of **Initialize**.
3. In step 3, $\mathcal{S}$ gets bits d^*_s for $s \neq i$ via $\mathcal{F}_{\mathsf{AT}}$, and for each $k \notin A$ sets the flag $\mathsf{shift\text{-}P_k}$ to true if $d^*_k \neq r_{i,k} + x$.
4. $\mathcal{S}$ sends $\{\mathsf{shift\text{-}P_k}\}_{k\notin A}$, $\{\nu^*_j\}_{j\in A}$, μ_i, x, to $\mathcal{F}_{\mathsf{Bootstrap}}$.

Case honest P_i. First, we show that $\Pi_{\mathsf{Bootstrap}}$ and $\mathcal{F}_{\mathsf{Bootstrap}}$ output identically distributed values if $\mathcal{Z}$ is honest-but-curious. In $\Pi_{\mathsf{Bootstrap}}$, the parties obtains a sharing $\langle\delta\rangle$, $\langle\nu\rangle$, and party P_i provides

input bit x and also obtains a field element μ. Then, we have

$$\begin{aligned}
\sum_{s\in\mathcal{P}} \nu_s + x\cdot\delta &= \big(\epsilon + x\cdot\delta_i\big) + \Big(\sum_{s\neq i}\big(\nu_{i,s} + d_s\cdot\delta_s\big)\Big) + x\cdot\delta,\\
&= \big(\epsilon + x\cdot\delta_i\big) + \Big(\sum_{s\neq i}\big((\mu_{i,s} + r_{i,s}\cdot\delta_s) + (x + r_{i,s})\cdot\delta_s\big)\Big) + x\cdot\delta,\\
&= \big(\epsilon + x\cdot\delta_i\big) + \Big(\sum_{s\neq i}\big(\mu_{i,s} + x\cdot\delta_s\big)\Big) + x\cdot\delta,\\
&= \epsilon + \sum_{s\neq i}\mu_{i,s},\\
&= \mu.
\end{aligned}$$

For what $\mathcal{Z}$ sees during the execution, either σ_k or d_s, leaked by $\mathcal{F}_{\mathsf{AT}}$, look random since they are paddings of δ_k and x with fresh pads δ'_k and $r_{i,s}$, given by $\mathcal{F}_{\mathsf{aBit}}$ to P_i. Now, denote by δ_H the sum of the portion of δ-shares that honest parties generated in **Initialize** of $\Pi_{\mathsf{Bootstrap}}$, and let $\delta^*_A = \sum_{j\in A}(\sigma^*_j + \delta'_j)$. That is, δ^*_A should match the sum of the corrupt portion of δ-shares generated in **Initialize**. Now, say P_i inputs bit x to $\Pi_{\mathsf{Bootstrap}}$, then, shares $\{\nu_k\}_{k\notin A}$ are such that $\sum_{k\notin A}\nu_k = \sum_{j\in A}\nu^*_j + x\cdot(\delta^*_A + \delta_H)$. In other words, honest ν_k is consistent with both, δ^*_A (that the adversary imposes via the σ^*_j's) and ν^*_j (that the adversary is suppose to derive from the bits d_j), and these shares are extracted by $\mathcal{S}$ in steps 1 and 3 respectively.

Case dishonest P_i. In this case, $\mathcal{S}$ sends random σ_k to $\mathcal{Z}$ on behalf of honest P_k. This is indistinguishable from what is sent in a real run, as P_k is using a padding given by $\mathcal{F}_{\mathsf{aBit}}$. For what $\Pi_{\mathsf{Bootstrap}}$ outputs to honest parties, we note again that, if $\mathcal{Z}$ gave correct d^*_k to $\mathcal{S}$ using $\mathcal{F}_{\mathsf{AT}}$, the sum of the honest portion of ν-shares is equal to $\sum_{j\in A}(\nu_{i,j} + (x + r_{i,j})\cdot\delta^*_j) + x\cdot\delta_i + \sum_{j\neq i}\mu_{i,j}$, which is extracted by $\mathcal{S}$ in step 1. And if $\mathcal{Z}$ does not send correct d^*_k, namely $d^*_k = x + r_{i,k} + 1$, it would cause honest P_k to compute shifted $\nu_k + \delta_k$, which is exactly what $\mathcal{S}$ tells to $\mathcal{F}_{\mathsf{Bootstrap}}$ to output in step 3. □

C.2 Functionality and Proof of the Online Phase (Theorem 1)

Functionality $\mathcal{F}_{\mathsf{Online}}$

Initialize: On input $(init)$ the functionality activates and waits for an input from the environment. Then it does the following: if it receives Abort, it waits for the environment to input a set of corrupted parties, outputs it to the parties, and aborts; otherwise it continues.

Input: On input $(input, P_i, varid, x)$ from P_i and $(input, P_i, varid, ?)$ from all other parties, with $varid$ a fresh identifier, the functionality stores $(varid, x)$.

Add: On command $(add, varid_1, varid_2, varid_3)$ from all parties (if $varid_1, varid_2$ are present in memory and $varid_3$ is not), the functionality retrieves $(varid_1, x)$, $(varid_2, y)$ and stores $(varid_3, x + y)$.

Multiply: On input $(multiply, varid_1, varid_2, varid_3)$ from all parties (if $varid_1, varid_2$ are present in memory and $varid_3$ is not), the functionality retrieves $(varid_1, x)$, $(varid_2, y)$ and stores $(varid_3, x\cdot y)$.

Output: On input $(output, varid)$ from all honest parties (if $varid$ is present in memory), the functionality retrieves $(varid, y)$ and outputs it to the environment. The functionality waits for an input from the environment. If this input is Deliver then y is output to all players. Otherwise it outputs $\varnothing$ is output to all players.

Figure 12 Secure Function Evaluation

We construct a simulator $\mathcal{S}$ such that an environment $\mathcal{Z}$ corrupting up to $n-1$ parties cannot distinguish whether it is playing with Π_{Online} attached with $\mathcal{F}_{\mathsf{Prep}}$ and $\mathcal{F}_{\mathsf{Comm}}$, or with the simulator $\mathcal{S}$ and $\mathcal{F}_{\mathsf{Online}}$. We start describing the behaviour of the simulator $\mathcal{S}$:

- The simulation of the **Initialize** procedure is performed running a copy of $\mathcal{F}_{\mathsf{Prep}}$ on query Init. All the data of the corrupted parties are known to the simulator. If $\mathcal{Z}$ inputs Abort to the copy of $\mathcal{F}_{\mathsf{Prep}}$, then the simulator does the same to $\mathcal{F}_{\mathsf{Online}}$ and forward the output of $\mathcal{F}_{\mathsf{Online}}$ to $\mathcal{Z}$: If $\mathcal{F}_{\mathsf{Online}}$ outputs Abort, the simulator waits for input a set of corrupted parties from $\mathcal{Z}$ and forward it to $\mathcal{F}_{\mathsf{Online}}$, and aborts; otherwise it uses the $\mathcal{Z}$'s inputs as preprocessed data.
- In the **Input** stage the simulator does the following. For the honest parties this step is run correctly with dummy inputs; it reads the inputs of corrupted parties specified by $\mathcal{Z}$. Then the simulator runs a copy of Share command of $\mathcal{F}_{\mathsf{Prep}}$ sending back sharings $[x]^i_\alpha$, for $i \in A$, where A is the set of corrupted parties. When $\mathcal{Z}$ writes the outputs corresponding to the corrupted parties, the simulator writes these values on the influence port of $\mathcal{F}_{\mathsf{Online}}$ as inputs.
- The procedure **Add, Multiply** are performed according to the protocol and the simulator calls the respective procedure to $\mathcal{F}_{\mathsf{Online}}$.
- In the **Output** step, the functionality $\mathcal{F}_{\mathsf{Online}}$ outputs y to the $\mathcal{S}$. Now the simulator has to provide shares of honest parties such that they are consistent with y. It knows an output value y' computed using the dummy inputs for the honest parties, so it can select a random honest player and modify its share adding $y - y'$ and modify the MAC adding $\alpha(y - y')$, which is possible for the simulator, since it knows α. After that the simulator opens y as in the protocol. If y passes the check, the simulator sends Deliver to $\mathcal{F}_{\mathsf{Online}}$.

All the steps of the protocol are perfectly simulated: during the initialization the simulator acts as $\mathcal{F}_{\mathsf{Prep}}$; addition does not involve communication, while multiplication implies partial opening: in the protocol, as well as in the simulation, this opening reveals uniform values. Also, MACs have the same distributions in both the protocol and the simulation.

Finally, in the output stage, $\mathcal{Z}$ can see y and the shares from honest parties, which are uniform and compatible with y and its MAC. Moreover it is a correct evaluation of the function on the inputs provided by the parties in the input stage. The same happens in the protocol with overwhelming probability, since the probability that a corrupted party is able to cheat in a MACCheck call is $2/|\mathbb{F}|$ (see Theorem 4). □

C.3 Proof of the Preprocessing (Theorem 2)

The description of the simulator, denoted by $\mathcal{S}$, is provided in Figure 13. Define $\mathcal{T}_{Real}$ to be the set of messages sent or received from corrupt parties together with the inputs and outputs of the parties, in an execution of Π_{Prep} with $\mathcal{F}_{\mathsf{Bootstrap}}$ and $\mathcal{F}_{\mathsf{Comm}}$. Likewise define $\mathcal{T}_{Ideal}$ for an execution of $\mathcal{F}_{\mathsf{Prep}}$with $\mathcal{S}$. To prove UC security, we see $\mathcal{Z}$ as a distinguisher between the two systems, and our aim is to show that

$$|\Pr[0 \leftarrow \mathcal{Z}(\mathcal{T}_{Real})] - \Pr[0 \leftarrow \mathcal{Z}(\mathcal{T}_{Ideal})] - \frac{1}{2}| \leq \mathsf{negl}(\kappa).$$

For this to hold, it is enough to show that $\mathcal{Z}$ receives as inputs transcripts $\mathcal{T}_{Real}$, $\mathcal{T}_{Ideal}$ that are statistically indistinguishable. We argue as follows.

First note that transcripts generated on calls to **Initialize** and **Share** in both executions, are *perfectly* indistinguishable, as they are nothing but calls to $\mathcal{F}_{\mathsf{Bootstrap}}$ in the real case, with identical behaviour of Share command in $\mathcal{F}_{\mathsf{Prep}}$, (and $\mathcal{S}$ only forwards queries to the $\mathcal{F}_{\mathsf{Prep}}$).

The Simulator of Π_{Prep}

The set of corrupt parties is denoted with A.

Initialize: $\mathcal{S}$ forwards to $\mathcal{F}_{\mathsf{Prep}}$ the query (Init) together with $\{\alpha_j\}_{j\in A}$ that $\mathcal{Z}$ does to $\mathcal{F}_{\mathsf{Bootstrap}}$. Then samples random $\alpha \in \mathbb{F}$, and a set of sharings $\{\alpha_k\}_{k\notin A}$ consistent with $\{\alpha_j\}_{j\in A}$ and α, but otherwise random. It stores the complete sharing for later use.

Share: $\mathcal{S}$ forwards to $\mathcal{F}_{\mathsf{Prep}}$ the query (i, Share) of $\mathcal{Z}$ to $\mathcal{F}_{\mathsf{Bootstrap}}$. $\mathcal{S}$ also gets flags $\{\mathsf{shift\text{-}P_k}\}_{k\notin A}$, and MAC shares $\{\mu_j\}_{j\in A}$ from $\mathcal{Z}$. If $i \in A$, $\mathcal{Z}$ specifies input bit x. $\mathcal{S}$ sends shift flags, MAC shares and (possibly) input x to $\mathcal{F}_{\mathsf{Prep}}$.

GaOT:

1. In steps 1 and 3, when $\mathcal{Z}$ thinks is querying $\mathcal{F}_{\mathsf{Bootstrap}}$, on commands Init and Share, respectively, $\mathcal{S}$ discards all the values received from $\mathcal{Z}$.
2. Steps 2, 4, 5 are local, and $\mathcal{S}$ does nothing.
3. Steps 6-7, are repeated four times, one for each symbol $y \in \{e, z, x_0, x_1\}$. In each invocation $\mathcal{S}$ does:
   - During the i-th query to Share command of $\mathcal{F}_{\mathsf{Bootstrap}}$, $\mathcal{S}$ receives from $\mathcal{Z}$ MAC shares $\{\nu_j(y_i) \in \mathbb{F}\}_{j\in A}$, and flags $\{\mathsf{shift\text{-}P_k}^{(i)}\}_{k\notin A}$; it also receives bit y_i, and $\mu_i(y_i) \in \mathbb{F}$, if $i \in A$.
   - After the n queries are done, $\mathcal{S}$ sets the data of each representation $[y_i]^i_\alpha$ corresponding to honest parties exactly as $\mathcal{F}_{\mathsf{Bootstrap}}$ would do. Thus, if $i \notin A$, $\mathcal{S}$ samples $y_i \in \mathbb{F}_2$, and $\mu_i(y_i) \in \mathbb{F}$ at random, otherwise uses $\mathcal{Z}$'s choice. It sets $\nu(y_i) = \mu_i(y_i) + y_i \cdot \alpha$ and prepares sharings $\langle \nu(y_i) \rangle$, where the honest shares $\nu_k(y_i)$ are consistent with $\mathcal{Z}$'s shares. Finally, $\mathcal{S}$ shifts honest share $\nu_k(y_i) = \nu'_k(y_i) + \alpha_k$ if $\mathsf{shift\text{-}}(P_k)^{(i)}$ is true. The honest data on the joint representation $[\![y]\!]$ is generated as one expects, where $y = \sum_{i\in\mathcal{P}} y_i$.
4. The above steps are repeated at least $\kappa + 1$ times, as in Π_{Prep}.
5. Steps 8-12 are performed as in Π_{Prep}, where $\mathcal{S}$ acts on behalf of honest parties using the dummy quadruples generated in the executions of step 3. It also answers queries from $\mathcal{Z}$ to Comm and Open commands of $\mathcal{F}_{\mathsf{Comm}}$. Openings on behalf of honest parties are set to random seed values.
6. If some iteration in the previous step result in abort, $\mathcal{S}$ inputs Abort to $\mathcal{F}_{\mathsf{Prep}}$. Otherwise, inputs Continue, and for each bit $y \in \{e, z, x_0, x_1\}$ of the checked quadruple, $\mathcal{S}$ discards the shift flags, and gives bit shares $\{y_j\}_{j\in A}$, and MAC shares $\{\mu_j(y)\}_{j\in A}$ derived in step 3, to $\mathcal{F}_{\mathsf{Prep}}$.

Figure 13 The Simulator of Π_{Prep}

We turn now to GaOT command. Let $\mathsf{OT_{out}} = \{[\![e]\!], [\![z]\!], [\![x_0]\!], [\![x_1]\!]\}$ be the quadruple that honest parties are hoping to output if no abort occurs. Define the "multiplicative relation" $m = z + x_0 + e \cdot (x_0 + x_1)$, and say that $\mathsf{OT_{out}}$ is bad if $m = 1$. Thus, bad quadruples are those that implement the multiplication gate incorrectly. Additionally, say that quadruple is noauth if $\mathcal{Z}$ sent to $\mathcal{F}_{\mathsf{Bootstrap}}(\alpha)$ flag $\mathsf{shift\text{-}P_k}$ set to true for at least one honest party P_k, during the execution of **AuthenticateOT**.

Indistinguishability of transcripts. First notice that $\mathcal{T}_{Real}$ and $\mathcal{T}_{Ideal}$ truncated up to the point where the parties output the quadruple are perfectly indistinguishable (steps 12 and 5 respectively): looking at Figure 13, we see that $\mathcal{S}$ sacrifices quadruples exactly as Π_{Prep}. More precisely, step 5 of $\mathcal{S}$ mimics steps 8-12 of Π_{Prep}. Moreover, $\mathcal{S}$ uses quadruples generated in step 3, and honest parties use quadruples generated in steps 6-7. These quadruples are identically distributed because $\mathcal{S}$ proceeds exactly as $\mathcal{F}_{\mathsf{Bootstrap}}$ does. Also, notice that in Π_{Prep} the output quadruples are those that parties choose to authenticate, and hence $\mathcal{S}$ skips the simulation of **ShareOT** (besides accepting $\mathcal{Z}$'s queries) since no outgoing communication from either $\mathcal{F}_{\mathsf{Bootstrap}}$ or party-to-party is done.

Output indistinguishability. If $\mathcal{Z}$ is honest-but-curious, then a run with Π_{Prep} outputs a quadruple that is neither bad nor noauth. This follows from the correctness of **ShareOT** and **AuthenticateOT** steps. Also, in step 3, $\mathcal{S}$ is able to extract the portion of shares of $\mathsf{OT_{out}}$ corresponding to corrupt parties, and give them to $\mathcal{F}_{\mathsf{Prep}}$. We therefore conclude that the outputs in both worlds are identically distributed. On the other hand, if $\mathcal{Z}$ misbehaves in an arbitrary way, it suffices to show the following to conclude the proof:

$$\mathsf{OT_{out}} \text{ is } \mathsf{bad} \vee \mathsf{noauth} \Rightarrow \Pi_{\mathsf{Prep}} \text{ outputs } \varnothing \text{ with probability } 1 - \mathsf{negl}(\kappa).$$

We argue as follows: the sacrifice step is run by the honest parties. Therein, in the ith iteration, a fresh check quadruple OT_i is taken and honest parties reveal a linear combination on their portion of the shares of $\mathsf{OT}_{\mathsf{out}}$ and OT_i, that open to p_i, q_i and c_i. If $\mathcal{Z}$ started with input shares that render an $\mathsf{OT}_{\mathsf{out}}$ that is noauth, or chooses to reveal something different, say wlog, the first opening gives wrong p_i^* Then he managed to either pass Π_{MACCheck} on the open values with p_i^* not authenticated, or he managed to authenticate p_i^* and feed it to Π_{MACCheck}. The former happens with probability $\frac{2}{|\mathbb{F}|}$ by Theorem 4 (assuming $\mathsf{PRF}_s^{\mathbb{F},t}(\cdot)$), and the latter is equivalent to have $\mathcal{Z}$ holding the field element $\mu_H + p_i^* \cdot \alpha_H = \sum_{k \notin A}(\mu_k(p_i^*) + p_i^* \cdot \alpha_k)$, and this happens with probability $\frac{1}{|\mathbb{F}|}$, since $\mu_H + p_i^* \cdot \alpha_H$ is only derivable from the private transcripts of honest parties (thus, $\mathcal{Z}$ must guess it). We conclude that, if Π_{MACCheck} passes, then $\mathcal{Z}$ misbehaves in the sacrifice step, or it inputs shares that render an $\mathsf{OT}_{\mathsf{out}}$ that is noauth, with probability bounded by $\frac{2}{|\mathbb{F}|} = 2^{-\kappa+1}$. Now, it is easy to see that if $\mathcal{Z}$ follows the sacrifice step, then we can write $c_i = m \cdot t_i + m'_i$, where m'_i is the multiplicative relation of OT_i. Therefore, if $\mathcal{Z}$ misbehaved in any previous step, yielding $\mathsf{bad}\ \mathsf{OT}_{\mathsf{out}}$, then $c_i = t_i + m'_i$. In this way if the sacrifice step passes, we can write $\mathbf{t} = \mathbf{m}'$, where $\mathbf{t}$ is the challenge vector. This vector is randomly sampled from $\mathbb{F}_2^\kappa$, assuming $\mathsf{PRF}_s^{\mathbb{F}_2,\kappa}(\cdot)$, thus the probability of having $\mathbf{t}$ fixed to $\mathbf{m}'$ is $2^{-\kappa}$.

Summing up, bad or noauth output quadruples will pass both tests with probability at most $2^{-\kappa+1}$. This concludes the proof of the theorem. □