High Performance Multi-Party Computation for Binary Circuits Based on Oblivious Transfer

Sai Sheshank Burra[4], Enrique Larraia[5], Jesper Buus Nielsen[1], Peter Sebastian Nordholt[2], Claudio Orlandi[1], Emmanuela Orsini[3], Peter Scholl[1], and Nigel P. Smart[3]

[1] Aarhus University, Denmark,
[2] Chainalysis Inc, Denmark,
[3] imec-COSIC, KU Leuven, Leuven, Belgium.
[4] Indian Institute of Technology Guwahati, India
[5] Scytl, Spain. ⋆⋆

elarraia@gmail.com,
jbn@cs.au.dk,
pnordholt@chainalysis.com,
orlandi@cs.au.dk,
emmanuela.orsini@kuleuven.be,
peter.scholl@cs.au.dk,
nigel.smart@kuleuven.be.

Abstract We present a unified view of the two-party and multi-party computation protocols based on oblivious transfer first outlined in Nielsen *et al.* (CRYPTO 2012) and Larraia et al. (CRYPTO 2014). We present a number of modifications and improvements to these earlier presentations, as well as full proofs of the entire protocol. Improvements include a unified pre-processing and online MAC methodology, mechanisms to pass between different MAC variants, and fixing a minor bug in the protocol of Larraia *et al.* in relation to a selective failure attack. It also fixes a minor bug in Nielsen *et al.* resulting from using Jensen's inequality in the wrong direction in an analysis.

⋆⋆ The work in this paper was carried out when Larraia as at the University of Bristol, United Kingdom and Royal Holloway, University of London, United Kingdom; Orsini, Scholl and Smart were at the University of Bristol, United Kingdom; Nordholt was at the Alexandra Institute, Denmark.

Contents

1 Introduction

Secure two-party computation (2PC), introduced by Yao [Yao82], allows two parties to jointly compute any function of their inputs in such a way that 1) the output of the computation is correct and 2) the inputs are kept private. There is a natural generalization to many parties called multi-party computation (MPC) in what follows. The simplest protocols are secure only if the participants are *semi-honest* (they follow the protocol but try to learn more than they should by looking at their transcript of the protocol). A more realistic security definition considers *malicious adversaries*, that can arbitrarily deviate from the protocol. Recently a number of efforts to implement maliciously secure MPC in practice have been reported on.

In the two party case almost all of these are based on Yao's garbled circuit technique. A main advantage of Yao's garbled circuits is that it is primarily based on symmetric primitives and has low round complexity. It uses one OT per input bit, but then uses only a few calls to a symmetric primitive, for example a hash function or block cipher evaluation, per gate in the circuit to be evaluated. Malicious security is obtained by elaborate cut-and-choose protocols which mean the basic semi-honest Yao protocol needs to be run many times. The other approaches are heavy on public-key primitives which are typically orders of magnitude slower than symmetric primitives.

For more than two parties the standard technique is to use secret sharing of the computed values. The resulting protocols have high round complexity, but usually require very little symmetric machinery in their function evaluation phases. In the variants of MPC which are based on secret sharing the major performance improvement over the last few years 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: For n-party dishonest majority MPC for arithmetic circuits over a "largish" finite field [BDOZ11, DPSZ12], and for n-party dishonest majority MPC over binary circuits [DZ13]. 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 [Bea95] "multiplication triples", which are then consumed in the online phase when the function is determined and evaluated.

The idea of OT extension dates from the work of Beaver [Bea96] in 1996, however it was not until 2003 until Ishai *et al.* introduced an *efficient* manner to extend OTs [IKNP03]. The protocol of Ishai *et al.* allows one to turn κ seed OTs based on public-key crypto into any polynomial $\ell = \text{poly}(\kappa)$ number of OTs using only $O(\ell)$ invocations of a cryptographic hash function. For big enough ℓ the cost of the κ seed OTs is amortized away and OT extension essentially turns OT into a symmetric primitive in terms of its computational complexity. Since the basic approach of basing 2PC on OT in [GMW87] is efficient in terms of consumption of OTs and communication, this gives the hope that OT-based 2PC too could be practical, and that it can be extended to the multi-party case.

In [NNOB12] and [LOS14] the first practically efficient protocols in the two and multi-party case for MPC based on oblivious transfer were given, both of which utilize the secret sharing based, pre-processing and MAC based authentication of [BDOZ11, DPSZ12, DZ13]. The current paper presents the work in [NNOB12] and [LOS14] in a unified manner, we present a number of improvements to these protocols (in both exposition and protocols), as well as correcting a number of errors. In particular in this paper:

- We unify the pre-processing and online phases of both [NNOB12] and [LOS14]. In particular instead of the various different pre-processed data from [NNOB12] (i.e. aANDs, aOTs etc) and the generalized OT quadruples from [LOS14], we utilize standard Beaver multiplication triples in both the 2PC and the MPC cases.
- We unify the authenticating via MACs in the online protocols for 2PC and MPC so as to use SPDZ like MACs. This enables the efficient MAC check functionality from [DKL$^+$13] to be utilized; thus enabling efficient reactive functionalities.
- We unify the different secret sharing notations between the two papers; and provide mechanisms to pass from one sharing type to another.
- We correct a bug in the pre-processing phase of the protocol from [LOS14], which allowed a selective failure attack. This is done by providing a more elaborate sacrificing and checking step in the pre-processing; plus a more careful attention to the underlying functionalities.

– The methodologies and proofs of the extension from a small number of seed OTs to a large number of OTs presented in [NNOB12], is also extended and improved.
– Overall we provide full proofs and analysis, compared to the shorter "abstracts" provided in [NNOB12] and [LOS14].

We call both our 2PC and MPC variants Tiny-OT, as they are based on a method of producing a large number of authenticated bits via oblivious transfer, from a small number of seed OTs.

1.1 Impact

This current paper was originally submitted to Journal of Cryptology in 2015, in the intervening six years the impact of the protocols in this paper (and by extension the papers [NNOB12] and [LOS14]) has been transformative in the practical MPC community. We list a number of the applications here.

In the case of MPC based on linear secret sharing the impact has been considerable. For the case of 'large' characteristic finite fields the work in this paper forms the basis of the MASCOT protocol [KOS16] for performing OT-based pre-processing (instead of Somewhat Homomorphic Encryption based pre-processing) for the SPDZ family of protocols [DPSZ12]. For fields of characteristic two the work in this paper was applied in the same direction in [FKOS15]. For MPC over rings, such as $\mathbb{Z}_{2^k}$, the SPDZ2k protocol [CDE+18] also utilized Tiny-OT based pre-processing.

The application of the Tiny-OT protocols is not only seen in protocols based on secret sharing, BMR based protocols have also, in the intervening years, had a major performance improvement using techniques arising out of the Tiny-OT protocol. For example, at the time of writing (2021), the most efficient, practical, n-party BMR style protocols are HSS [HSS17] and WRK [WRK17b]; both of which are based on Tiny-OT for the underlying protocol to produce the shared garbling. Tiny-OT was also applied in the Tiny-Keys extensions to the above protocols [HOSS18a, HOSS18b]. Tiny-OT also forms the basis of the two party protocol [WRK17a].

1.2 Overview

Our starting point is the efficient passive-secure OT extension protocol of [IKNP03] and passive-secure 2PC of [GMW87]. In order to obtain active security and preserve the high practical efficiency of these protocols we chose to develop substantially different techniques, differentiating from other works that were only interested in *asymptotic* efficiency [HIKN08, Nie07, IPS08].

We introduce a new technical idea to the area of extending OTs efficiently, which allows us to dramatically improve the practical efficiency of active-secure OT extension. Our protocol has the same asymptotic complexity as the previously best protocol in [HIKN08], but it is only a small factor slower than the passive-secure protocol in [IKNP03].

In addition, we present techniques which allow us to relate the outputs and inputs of OTs in a larger construction, via the use of information theoretic tags. This can be seen as a new flavor of committed OT that only requires symmetric cryptography. In combination with our first contribution, our protocols show how to efficiently extend committed OT. Our protocols assume the existence of OT and are secure in the random oracle model. The question on the *asymptotic* computational overhead of cryptography was (essentially) settled in [IKOS08].

On the other hand, there is a growing interest in understanding the *practical* overhead of secure computation, and several works have perfected and implemented 2PC protocols based on Yao's garbled circuits [MNPS04, BDNP08, LPS08, KS08, NO09, PSSW09, HKS+10, MK10, LP11, asS11, HEK+11], protocols based on homomorphic encryption [IPS09, DO10, JMN10, BDOZ11] and protocols based on OT [IPS08, LOP11, CHK+12]. For the MPC variant a number of implementations have been presented based on secret sharing [DPSZ12, DKL+13, DGKN09, SIM, BLW08].

For the case of MPC protocols, where $n > 2$, there are three main techniques of using MACs to authenticate the shares in an MPC protocol. In [BDOZ11] 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 [DPSZ12], called the SPDZ protocol hereafter, the authors obtain a more efficient online protocol by replacing the MACs from [BDOZ11] 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 [DKL+13], where it is shown how to verify these global MACs without revealing the secret global key. In [DZ13] the authors adapt the technique from [DPSZ12] 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. This is performed using a novel application of ideas from coding theory, and results in a reduced overhead for the online phase.

One can think of the 2PC Tiny-OT protocol from [NNOB12] as applying the MAC based authentication technique of [BDOZ11] to the two party, binary circuit case, with a pre-processing which is based on OT as opposed to semi-homomorphic encryption. Whilst the authentication in [LOS14] is based on the SPDZ MAC'ing method. Part of our unification is to utilize the MACs from the SPDZ protocol in both the 2PC and the MPC variants; and thus also the secret sharing methodology from [LOS14, DPSZ12] to the 2PC variant.

We start from a classic textbook protocol for 2PC [Gol04, Sec. 7.3]. In this protocol, Alice holds secret shares x_A, y_A and Bob holds secret shares x_B, y_B of some bits x, y s.t. $x_A \oplus x_B = x$ and $y_A \oplus y_B = y$. Alice and Bob want to compute secret shares of $z = g(x, y)$ where g is some Boolean gate, for instance the AND gate: Alice and Bob need to compute a random sharing z_A, z_B of

$$z = x \cdot y = (x_A \cdot y_A) \oplus (x_A \cdot y_B) \oplus (x_B \cdot y_A) \oplus (x_B \cdot y_B).$$

The parties can compute the AND of their local shares ($x_A y_A$ and $x_B y_B$), while they can use oblivious transfer (OT) to compute the cross products ($x_A y_B$ and $x_B y_A$). Now the parties can iterate for the next layer of the circuit, up to the end where they will reconstruct the output values by revealing their shares.

This protocol is secure against a semi-honest adversary: assuming the OT protocol to be secure, Alice and Bob learn nothing about the intermediate values of the computation. It is easy to see that if a large circuit is evaluated, then the protocol is not secure against a malicious adversary: any of the two parties could replace values on any of the internal wires, leading to a possibly incorrect output and/or leakage of information.

Our main contribution is a new way to find a particular committed OT-like primitive which allows both a very efficient generation and a very efficient use: while previous results based on committed OT require hundreds of *exponentiations* per gate, our cost per gate is in the order of hundreds of *hash functions*. To the best of our knowledge, we present the first practical approach to extending a few seed OTs into a large number of committed OT-like primitives. Of more specific technical contributions, the main is that we manage to do all the proofs efficiently, thanks also to the preprocessing nature of our protocol: We obtain active security paying only a constant overhead over the passive-secure protocol in [IKNP03].

At the heart of both our 2PC and MPC protocols is a method, called aBit, for authenticating random bits via pairwise MACs, which itself is based on our efficient protocol for OT-extension. Our aim is to use this efficient two-party process as a black-box to produce our unified 2PC and MPC variants. In the case of 2PC a local computation enables us to transform from the authenticated random bits with pairwise MACs to our globally authenticated secret shared values. Unfortunately, if we extend this procedure naively to the three party case and beyond, 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 [DKL+13]. This allows us to obtain authenticated random shares in the multi-party case.

The online phase will then follow similarly to [DKL+13], if we can realize a protocol to produce "multiplication triples". Thus we then present protocols to produce such Beaver triples from the aBits. This is done in the 2PC case by utilizing a complex method to produce authenticated random OTs and authenticated random ANDs (called aOTs and aANDs). In the MPC case we are able to utilize our basic extended OTs to produce the multiplication triples; which is less efficient when specialised to the 2PC case than our specialised protocol. In the generation of aANDs and aOTs, we replace cut-and-choose with efficient, slightly leaky proofs and then use a combiner to get rid of the leakage: When we preprocess for ℓ gates and combine B leaky objects to get each unleaky object, the probability of leaking is $(2\ell)^{-B} = 2^{-\log_2(\ell)(B-1)}$. As an example, if we preprocess for 2^{20} gates with an overhead of $B = 6$, then we get leakage probability 2^{-100}.

1.3 Paper Overview

In Section 2 we set up the various authentication methods for the secret sharing schemes used in this paper. In Section 3 we define some of the basic primitives which we will use throughout the paper. Then in Section 4 we present the general 2PC/MPC protocol, and describe the pre-processing functionality we utilize. We show that the online 2PC/MPC functionality can be realised assuming a secure implementation of the pre-processing functionality. Then in Section 5 we present our main technical results on how to extend a number of seed OTs to a large number of pairwise authenticated random OTs. Utilizing these components we are then able to discuss our two pre-processing implementations. We first discuss the general MPC variant in Section 6, and then turn to an optimized variant in the case of two parties in Section 7.

2 Authenticated Secret Sharing

Most of our protocols will be processing binary data, i.e. elements in the finite field $\mathbb{F}_2$. However, to ensure security we will often require additional elements in extensions of $\mathbb{F}_2$, which we shall denote $\mathbb{F}$. The exact degree of $\mathbb{F}$ will depend on the precise context, if the degree is not clear from the context we will write $\mathbb{F}_2^m$. To aid exposition we will freely move between thinking of elements in $\mathbb{F}$ as finite field elements or as bit-strings of a fixed length; thus we identify $\mathbb{F}_2^m$ with F_{2^m} via some implicit public basis.

2.1 Basic Secret Sharing

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 $\bigoplus_{i_j \in \mathcal{I}} a_{i_j} = a$. 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 also lie in the same field, i.e. $a_i \in \mathbb{F}_2$ (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 locally

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

2.2 Authenticating Secret-Shared Values

Our main technique for authentication of secret shared bits is applied by placing an *information theoretic tag* (MAC) on the shared bit x. There are two ways to authenticate a secret globally held by a system of parties, one is to authenticate the shares of each party, as in [BDOZ11], the other is to authenticate the secret itself, as in [DPSZ12]. In addition we can also have authentication in a pairwise manner, as in [BDOZ11], or in a global manner, as in [DPSZ12]. 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 [BDOZ11] to the technique used in [DPSZ12]. The use of SPDZ style MACs in our main protocol, as opposed to BDOZ or NNOB style MACs allows our protocol to evaluate reactive functionalities without needing to re-run the pre-processing phase.

The authenticating key is a random line in $\mathbb{F}$, and the MAC on $x \in \mathbb{F}_2$ is its corresponding line point, thus, the linear equation $\mu_\delta(x) = \nu_\delta(x) \oplus (x \cdot \delta)$ holds, for some $\mu_\delta(x), \nu_\delta(x), \delta \in \mathbb{F}$. We will use these lines in various operations[6], for various values of δ (lying in finite fields of different degrees). 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. The degree of the field containing α, and hence the size of the bitstring representing α, will be equal to the statistical security parameter of our protocol σ.

The key α will be the same key for every bit that needs to be authenticated in the main MPC protocol. 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:

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

where we have that $x \in \mathbb{F}_2$ 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

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

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 $\nu_j = 0$ [7]. This simplification means that a $[\![x]\!]$ sharing is given by two sharings $\big(\langle x \rangle^{\mathcal{P}}, \langle \mu \rangle^{\mathcal{P}}\big)$. Notice that the $[\![\cdot]\!]$-representation of a bit x implies that x is both *authenticated* with the global key α and that it is $\langle \cdot \rangle$-shared, i.e. its value is actually unknown to the parties. 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 $\mu_\alpha(x)$. In [DPSZ12] the data is shared via our $[\![x]\!]$ notation, except that the MAC key value ν is 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 [DKL+13], bar the differences in the underlying finite fields.

With the above notation the MAC'd secret sharing scheme of [BDOZ11] 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, i.e. $x = \bigoplus x_i$, and the authentication is performed via $[x_i]^i_{\alpha_j,j}$ sharings, i.e. we are using two sharing schemes simultaneously. In [NNOB12], in the two party setting, the authors create sharings $[x_1]^1_{\Delta_1,2}$ of a sharing of a bit x_1 held by party P_1, and authenticated to party P_2, and a sharing $[x_2]^2_{\Delta_2,1}$ of a sharing of a bit x_2 held by party P_2 and authenticated to party P_1. These are the characteristic two variant of the sharings proposed in the multi-party case in [BDOZ11]. Their two party protocol then works on a pair of these sharings $[x_1|x_2]$. We let $\wr x \int$ denote a

[6] 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.

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

sharing of a value $x = x_1 \oplus x_2$ in the form $[x_1|x_2]$, i.e. where x_1 is shared via $[x_1]^1_{\Delta_1,2}$ and x_2 is shared by $[x_2]^2_{\Delta_2,1}$. The following procedure allows us to locally transform from a $\wr x \int$ sharing to a $[\![x]\!]$ sharing for the key $\Delta = \Delta_1 \oplus \Delta_2$:

- Party P_1 holds x_1, Δ_1, $\mu_{\Delta_2}(x_1)$ and $\nu_{\Delta_1}(x_2)$; whilst party P_2 holds x_2, Δ_2, $\mu_{\Delta_1}(x_2)$ and $\nu_{\Delta_2}(x_1)$, where we have $\mu_{\Delta_2}(x_1) = \nu_{\Delta_2}(x_1) \oplus (x_1 \cdot \Delta_2)$ and $\mu_{\Delta_1}(x_2) = \nu_{\Delta_1}(x_2) \oplus (x_2 \cdot \Delta_1)$.
- Party P_1 sets $\mu_1 = \nu_{\Delta_1}(x_2) \oplus \mu_{\Delta_2}(x_1) \oplus (x_1 \cdot \Delta_1)$.
- Party P_2 sets $\mu_2 = \nu_{\Delta_2}(x_1) \oplus \mu_{\Delta_1}(x_2) \oplus (x_2 \cdot \Delta_2)$.
- Party P_1's sharing is then (x_1, μ_1), whilst party P_2's sharing is (x_2, μ_2).

We then have as required

$$\begin{aligned}
\mu_1 \oplus \mu_2 &= (\nu_{\Delta_1}(x_2) \oplus \mu_{\Delta_2}(x_1) \oplus (x_1 \cdot \Delta_1)) \oplus (\nu_{\Delta_2}(x_1) \oplus \mu_{\Delta_1}(x_2) \oplus (x_2 \cdot \Delta_2)), \\
&= (\nu_{\Delta_1}(x_2) \oplus \nu_{\Delta_2}(x_1) \oplus (x_1 \cdot \Delta_2) \oplus (x_1 \cdot \Delta_1)) \oplus (\nu_{\Delta_2}(x_1) \oplus \nu_{\Delta_1}(x_2) \oplus (x_2 \cdot \Delta_1) \oplus (x_2 \cdot \Delta_2)), \\
&= (x_1 \cdot (\Delta_2 \oplus \Delta_1)) \oplus (x_2 \cdot (\Delta_1 \oplus \Delta_2)), \\
&= (x_1 \oplus x_2) \cdot \Delta = x \cdot \Delta.
\end{aligned}$$

Thus we can pass in the two-party case from the sharing methodology used in [BDOZ11], to our more general method, with no need for interaction. Extending this ability to transform [BDOZ11] style pairwise sharings to our $[\![\cdot]\!]$ sharing in the multi-party case will be one of the subjects of Section 6.

2.3 Arithmetic on $[\![x]\!]$ Shared Values.

Given two representations

$$[x]^{\mathcal{I}_x}_{\delta,\mathcal{J}_x} = \big(\langle x\rangle^{\mathcal{I}_x}, \langle \mu_\delta(x)\rangle^{\mathcal{I}_x}, \langle \nu_\delta(x)\rangle^{\mathcal{J}_x}\big) \text{ and } [y]^{\mathcal{I}_y}_{\delta,\mathcal{J}_y} = \big(\langle y\rangle^{\mathcal{I}_y}, \langle \mu_\delta(y)\rangle^{\mathcal{I}_y}, \langle \nu_\delta(y)\rangle^{\mathcal{J}_y}\big),$$

under same the δ, the parties can locally compute

$$[x \oplus y]^{\mathcal{I}_x \cup \mathcal{I}_y}_{\delta,\mathcal{J}_x \cup \mathcal{J}_y} = \big(\langle x\rangle^{\mathcal{I}_x} \oplus \langle y\rangle^{\mathcal{I}_y}, \langle \mu_\delta(x)\rangle^{\mathcal{I}_x} \oplus \langle \mu_\delta(y)\rangle^{\mathcal{I}_y}, \langle \nu_\delta(x)\rangle^{\mathcal{J}_x} \oplus \langle \nu_\delta(y)\rangle^{\mathcal{J}_y}\big)$$

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:

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

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

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 Basic Primitives

Through out this paper we will repeatedly use a common set of simple primitives to implement our protocols. In this section we present these basic primitives and discuss their computational complexity so that we can later estimate the complexity of protocols implemented using these primitives.

3.1 Hash Functions

We use a hash function $H : \{0,1\}^* \to \{0,1\}^\psi$. We sometimes use H to mask a message, as in $H(x) \oplus M$. If $|M| \neq \psi$, we will abuse the notation and use $H(x) \oplus M$ to denote $\mathrm{prg}(H(x)) \oplus M$, where prg is a pseudo-random generator $\mathrm{prg} : \{0,1\}^\psi \to \{0,1\}^{|M|}$. We model the function H as a random oracle (RO). We typically use H for two tasks: 1) As a prg to extend a string $s \in \{0,1\}^\psi$ to some longer string $s' \in \{0,1\}^\ell$ for some $\ell = \mathrm{poly}(\psi)$. 2) To hash a long string $s \in \{0,1\}^\ell$ to a short string for some $s' \in \{0,1\}^\psi$. When we attempt to sketch the complexity of these tasks we count both tasks as ℓ/ψ calls to the hash function H. We will also use a collision-resistant hash function $G : \{0,1\}^{2\psi} \to \{0,1\}^\psi$.

As in other works on MPC protocols whose focus is efficiency [KS08, HEK+11, DKL+13, NNOB12, LOS14, PSSW09], we are content with a proof in the random oracle model. What is the exact assumption on the hash function that we need for our protocol to be secure, as well as whether this can be implemented under standard cryptographic assumptions is an interesting theoretical question, see [AHI11, CKKZ12].

3.2 Oblivious Transfer

We use a two party oblivious transfer (OT) functionality: we denote by $\mathcal{F}_{\mathrm{OT}}$ the regular $\binom{2}{1}$-OT functionality, i.e. P_1 inputs two *messages* M_0 and M_1 to $\mathcal{F}_{\mathrm{OT}}$ and P_2 inputs a *choice bit* b. Nothing is output to P_1 and M_b output to P_2. We use the notation $\mathcal{F}_{\mathrm{OT}}(\ell, \tau)$ for the OT functionality that provides ℓ OTs with messages in $\{0,1\}^\tau$. When the length of messages and/or amount of OTs are clear from context we may drop these parameters.

The Functionality $\mathcal{F}_{\mathrm{OT}}(\ell, \tau)$

OT:

1. The functionality waits for input $(M_0^i, M_1^i)_{i\in[\ell]} \in \{0,1\}^{2\tau\ell}$ from P_1 and $(b_i)_{i\in[\ell]} \in \{0,1\}^\ell$ from P_2.
2. The functionality outputs $(M_{b_i}^i)_{i\in[\ell]}$ to P_2 and nothing to P_1.

Figure 1 The Oblivious Transfer Functionality

We will often make use of a randomized version of this functionality. The randomized version of the $\mathcal{F}_{\mathrm{OT}}$ functionality samples M_0, M_1 and b uniformly at random and then outputs (M_0, M_1) to P_1 and (b, M_b) to P_2. When considering the randomized version of OT we will let corrupted parties choose their own random values. Say P_2 above was corrupted, he then first gets to input M_b and b, the functionality then samples $M_{b\oplus 1}$ uniformly at random and outputs M_0 and M_1 to P_1. We denote the randomized version of OT by ROT and name the corresponding functionality $\mathcal{F}_{\mathrm{ROT}}$.

We will not consider a concrete implementation of the $\mathcal{F}_{\mathrm{OT}}$ or $\mathcal{F}_{\mathrm{ROT}}$ functionalities. When estimating concrete complexity of our protocols we will use $\mathcal{F}_{\mathrm{ROT}}(\ell, \psi)$ and $\mathcal{F}_{\mathrm{OT}}(\ell, \psi)$ as units of computational complexity, i.e., the functionality that does ℓ (random) OT's with messages of length ψ, the computational security parameter.

We note that we can easily implement $\mathcal{F}_{\mathrm{ROT}}(\ell, \tau)$ for any $\tau = \mathrm{poly}(\psi)$ using a prg and a $\mathcal{F}_{\mathrm{ROT}}(\ell, \psi)$ functionality: simply use $\mathcal{F}_{\mathrm{ROT}}(\ell, \psi)$ to transfer seeds of length ψ and then extend these seeds to length τ using the hash function H as described above, i.e., the cost of $\mathcal{F}_{\mathrm{ROT}}(\ell, \tau)$ becomes one call to $\mathcal{F}_{\mathrm{ROT}}(\ell, \psi)$ and τ/ψ calls to H.

3.3 Commitments

We use a general commitment functionality $\mathcal{F}_{\mathrm{COMM}}(\tau)$ for strings of length τ. We use this functionality both between two parties and between several parties in the case of a multiparty protocol. The functionality is described in Figure 2 for any number of parties. Note that we allow a corrupt party to refuse to open a commitment using the NoOpen command.

In the programmable RO model $\mathcal{F}_{\text{COMM}}$ can be implemented very efficiently using a few calls to a hash function as described formally in the protocol Π_{COMM} in Figure 3. To commit, the committer broadcasts a hash of the value v to be committed concatenated with a random bit string r_v. To open the commitment the committer broadcasts v, r_v to the other parties who check if the hash matches the commitment, i.e., each party uses H to hash a string of $\tau + \psi$ bits to ψ bits, so the total cost of commitment and opening is two broadcasts and $n(\tau/\psi + 1)$ calls H where n is number of parties involved in the protocol.

Below we prove that Π_{COMM} securely implements $\mathcal{F}_{\text{COMM}}$ in the programmable RO model.

The Functionality $\mathcal{F}_{\text{COMM}}(\tau)$

Commit:

On input $(\texttt{comm}, v, i, \tau_v)$ from P_i, where $v \in \{0,1\}^\tau \cup \{\perp\}$, and τ_v is a unique handle independent on v associated with the commitment, the functionality first stores (v, i, τ_v) and then outputs (i, τ_v) to all parties.

Open:

On input $(\texttt{open}, i, \tau_v)$ from P_i, where an entry (v, i, τ_v) is stored in the functionality, the functionality outputs (v, i, τ_v).

If a corrupt P_i inputs $(\texttt{NoOpen}, i, \tau_v)$, where an entry (v, i, τ_v) is stored in the functionality, the functionality outputs $(\perp, i, \tau_v)$ to all parties.

Figure 2 The Commitment Functionality

Protocol Π_{COMM}

Commit:

To commit to a value $v \in \{0,1\}^\tau$, P_i samples a uniformly random $r_v \in \{0,1\}^\psi$ and then defines the opening as $o_v = (i||v||r_v)$ and the commitment as $c_v = H(o_v)$. Finally, P_i broadcasts $(\texttt{comm}, c_v, i, \tau_v)$ to all parties, where τ_v is some unique handle chosen independt from v.

Open:

To open the previously committed value v P_i broadcasts $(\texttt{open}, o_v, i, \tau_v)$ to all parties. On receiving $(\texttt{open}, o_v, i, \tau_v)$ all parties P_j check if $H(o_v) = H(i||v||r_v) = c_v$. If so, P_j outputs (v, i, τ_v), otherwise P_j outputs $(\perp, i, \tau_v)$.

Figure 3 The Protocol Implementing $\mathcal{F}_{\text{COMM}}(\tau)$

Theorem 1. *The protocol Π_{COMM} described in Figure 3 securely implements $\mathcal{F}_{\text{COMM}}(\tau)$ in the programmable RO model.*

Proof. We consider the simulation in the two separate situations where the committer P_i is honest and corrupt respectively.

When P_i is honest, the simulator, upon receiving $(\texttt{comm}, i, \tau)$ from $\mathcal{F}_{\text{COMM}}$, simulates the commit phase of the protocol by picking a uniformly random c and broadcasting $(\texttt{comm}, c, i, \tau)$ to the corrupted parties. To simulate the opening phase the simulator first receives $(\texttt{open}, v, i, \tau)$ from $\mathcal{F}_{\text{COMM}}$ then samples a random $r_v \in \{0,1\}^\psi$ and programs the RO so that $H(i||v||r_v) = c$. Finally the simulator broadcasts $(\texttt{open}, o_v, i, \tau) = (\texttt{open}, i||v||r_v, i, \tau)$ to the corrupted parties. Note that the simulation is perfect unless the adversary has queried H on input $i||v||r_v$ before the simulator programs the RO. However, since this would require the adversary to guess the uniformly random string $r_v \in \{0,1\}^\psi$ this happens with at most negligible probability.

When P_i is corrupt, the simulator simulates broadcast by simply forwarding P_i's messages. The simulation of the commit phase starts by P_i broadcasting some $(\texttt{comm}, c^*, i, \tau)$ where c^* may or may not be obtained by the adversary querying H on some input $o^* = i||v^*||r^*$. In the former case the simulator can extract o^* from the call to H, in the latter the simulator picks $o^* = i||v^*||r^*$ with uniformly random $v^*||r^* \in \{0,1\}^{\tau+\psi}$. In either cases the simulator inputs $(\texttt{comm}, v^*, i, \tau)$ to $\mathcal{F}_{\text{COMM}}$. In the opening phase the corrupt P_i broadcasts $(\texttt{open}, o', i, \tau)$ for some $o' = i||v'||r'$. Now the simulator checks if $H(o') = c^*$, and if so inputs $(\texttt{open}, i, \tau)$ to $\mathcal{F}_{\text{COMM}}$, and otherwise inputs $(\texttt{NoOpen}, i, \tau)$ making the honest parties output (v^*, i, τ) or $(\perp, i, \tau)$ respectively. Since the simulator simply forwards the messages of P_i the only way to distinguish real protocol from the simulation is the output of the honest parties. Note then that in both real world and simulation the honest parties only output $(\perp, i, \tau)$ if $H(o') \neq c^*$. So the only hope to distinguish the real world from the simulation is if the corrupt party finds o' so that $v' \neq v^*$ and $H(o') = c^*$. In this case the honest parties output (v', i, τ) in the real world and (v^*, i, τ) in the simulation. However, this would require the adversary to either find a collision for H (in the case o^* could be extracted) or to find a preimage of H for the string c^* (in the case o^* was picked by the simulator). Either way by the properties of RO's this can happen with at most negligible probability, so the simulation is indistinguishable from the real world.

□

3.4 Equality Check

To simplify the exposition of our protocols we define a functionality $\mathcal{F}_{\text{EQ}}(\tau)$ which allows two parties to each input a string of length τ and check if the two strings are equal. In our protocols we will use $\mathcal{F}_{\text{EQ}}$ to detect cheating. Inequality will indicate that one party must have cheated and the protocol will be aborted, i.e., in our use of the primitive honest parties will never input unequal strings. For this reason we mainly need $\mathcal{F}_{\text{EQ}}$ to guarantee independence of inputs, i.e. we do not require the $\mathcal{F}_{\text{EQ}}$ functionality to protect the privacy of its inputs in case of inequality. Therefore, the functionality leaks both strings, which makes secure implementation easy using the $\mathcal{F}_{\text{COMM}}$ functionality. The $\mathcal{F}_{\text{EQ}}$ functionality is described formally in Figure 4.

The Functionality $\mathcal{F}_{\text{EQ}}(\tau)$

Honest Parties

1. The functionality waits for inputs $S_1, S_2 \in \{0,1\}^\tau$ from P_1 and P_2 respectively.
2. The functionality outputs to both parties $(\texttt{equal}, S_1, S_2)$ if $S_1 = S_2$ and $(\texttt{unequal}, S_1, S_2)$ otherwise.

Corrupt Parties

If P_1 is corrupted the functionality behaves as above, expect before it gives output to P_2 it waits for a special input from P_1. On input `deliver` the functionality outputs to P_2 as above. On input `abort` the functionality outputs `abort` to P_2 instead of the usual output.

Figure 4 The Equality Functionality

For completeness we give the simple protocol $\mathit{\Pi}_{\text{EQ}}$ in Figure 5 and prove that it securely implements $\mathcal{F}_{\text{EQ}}$ in the $\mathcal{F}_{\text{COMM}}$ hybrid model. In the protocol the two parties essentially just exchange and compare their inputs using a single invocation of $\mathcal{F}_{\text{COMM}}$ to ensure input independence, i.e., with this implementation the cost of $\mathcal{F}_{\text{EQ}}$ becomes simply $2(\tau/\psi)$ calls to H. Note that in Figure 4 we allow a corrupt P_1 to abort the protocol after she receives her output but before P_2 receives his output. In fact, this ability is implied since we want security with dishonest majority. In Figure 4 we only make this explicit as we will use it in the simulation of $\mathit{\Pi}_{\text{EQ}}$.

Theorem 2. *The protocol $\mathit{\Pi}_{EQ}$ securely implements $\mathcal{F}_{EQ}(\tau)$ in the $\mathcal{F}_{COMM}(\tau)$ hybrid model.*

Proof. The proof is straight forward for the case of no corrupted parties and both parties being corrupted.

In the case of corrupted P_1 the simulator extracts P_1's input S_1 from the call to $\mathcal{F}_{\text{COMM}}$ and forwards it to $\mathcal{F}_{\text{EQ}}$. When $\mathcal{F}_{\text{EQ}}$ leaks $(\cdot, S_1, S_2)$ the simulator forwards S_2 to P_1. Finally if P_1 refuses to open her commitment to S_1 (by inputting $(\texttt{NoOpen}, 1, \tau_{S_1})$ to $\mathcal{F}_{\text{COMM}}$) the simulator inputs `abort` to $\mathcal{F}_{\text{EQ}}$. If P_1 opens her commitment $\mathcal{S}$ inputs `deliver` to $\mathcal{F}_{\text{EQ}}$. Since $\mathcal{S}$ extracts the input of honest P_2 from $\mathcal{F}_{\text{EQ}}$ the message send to P_1 is identical to the real protocol. By the properties of $\mathcal{F}_{\text{COMM}}$ the output of honest P_2 is also identical to the real world, i.e., the simulation is perfect.

Protocol $\varPi_{\mathrm{EQ}}$

1. P_1 inputs $(\mathtt{comm}, S_1, 1, \tau_{S_1})$ to $\mathcal{F}_{\mathrm{COMM}}$.
2. On receiving $(1, \tau_{S_1})$ P_2 sends S_2 to P_1.
3. P_1 inputs $(\mathtt{open}, 1, \tau_{S_1})$ to $\mathcal{F}_{\mathrm{COMM}}$ and outputs $(\mathtt{equal}, S_1, S_2)$ if $S_1 = S_2$ and $(\mathtt{unequal}, S_1, S_2)$ otherwise.
4. If P_2 receives $(S_1, 1, \tau_{S_1})$ from $\mathcal{F}_{\mathrm{COMM}}$ he outputs $(\mathtt{equal}, S_1, S_2)$ if $S_1 = S_2$ and $(\mathtt{unequal}, S_1, S_2)$ otherwise. If P_2 receives $(\perp, 1, \tau_{s_1})$ from $\mathcal{F}_{\mathrm{COMM}}$ he outputs `abort`.

Figure 5 The Protocol Implementing $\mathcal{F}_{\mathrm{EQ}}(\psi)$

In case of corrupted P_2 the simulator simulates $\mathcal{F}_{\mathrm{COMM}}$ by simply sending a handle for S_1 to P_2. When P_2 sends his input S_2 the simulator forwards it to $\mathcal{F}_{\mathrm{EQ}}$. Since $\mathcal{F}_{\mathrm{EQ}}$ leaks S_1 the simulator can easily simulate the opening of $\mathcal{F}_{\mathrm{COMM}}$. Again the simulation is clearly perfect. $\square$

The relationship between the functionalitie $\mathcal{F}_{\mathrm{EQ}}$ and $\mathcal{F}_{\mathrm{COMM}}$ and the protocol $\varPi_{\mathrm{EQ}}$ are presented in Figure 6. We use such diagrams throughout this paper to explain how functionalities are related, and how protocols are used to implement the functionalities.

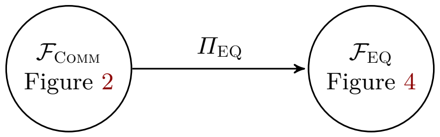


Figure 6 How $\mathcal{F}_{\mathrm{EQ}}$ is Built up From $\mathcal{F}_{\mathrm{COMM}}$

4 The 2PC and MPC Protocols in the Preprocessing Model

Our aim is to construct a protocol for arithmetic operations over $\mathbb{F}_2$. We work in the pre-processing model. In this model the circuit evaluation is divided in two phases. An offline phase produces correlated randomness which is almost independent to the circuit at hand[8]. To this end we use expensive primitives which typically need computational assumptions. In the online phase the actual evaluation of the circuit takes place. This phase is (essentially) information-theoretic, and hence very efficient, because we have “moved” all heavy operations into the offline phase. The only non-information theoretic primitives used in this phase are a small number of symmetric primitives used to perform checks on the MACs; which are amortized over possibly thousands of the main information theoretic steps. More precisely, we want to implement the ideal functionality given in Figure 7.

It models the arithmetic operations that are needed for the evaluation of the circuit. The relationship between this functionality (our main goal of the paper) and the sub-functionalities we will use are presented in Figure 8.

4.1 Ensuring Robust Computations

The basic idea to implement $\mathcal{F}_{\mathrm{ONLINE}}$ is to use precomputed correlated randomness. In this section we assume this data to be given by an offline functionality, in a secret-shared and authenticated fashion. Before we go into more details, we first explain how honest parties cope with dishonest behaviour.

At several points of the implementations of the online and offline functionalities we *partially* open values; recall this means that for a $[\![x]\!]$ sharing we open $\langle x\rangle$ but not the shares of $\mu_\alpha(x)$. Consequently, we need a mechanism to check that we are working with the correct values, or in other words, with (some linear

[8] During the offline phase only the size of the circuit is known

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

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

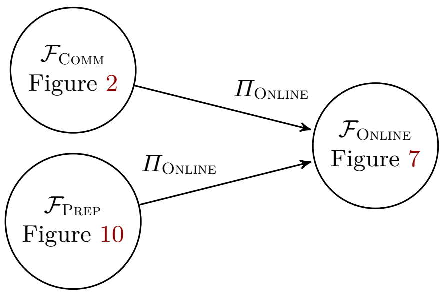
Input: On input $(input, P_i, varid, x)$ from P_i and $(input, P_i, varid, ?)$ from all other parties, with $varid$ a fresh identifier, the functionality stores $(varid, x)$.

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 $(varid_3, x \oplus y)$.

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 adversary. The functionality waits for an input from the adversary. If this input is Deliver then y is output to all parties. Otherwise it outputs $\varnothing$ to all parties.

Figure 7 Secure Function Evaluation



$\mathcal{F}_{\text{Comm}}$ Figure 2
Π_{Online}
$\mathcal{F}_{\text{Online}}$ Figure 7
Π_{Online}
$\mathcal{F}_{\text{Prep}}$ Figure 10

Figure 8 How $\mathcal{F}_{\text{Online}}$ is Built up From $\mathcal{F}_{\text{Comm}}$ and $\mathcal{F}_{\text{Prep}}$

transformation of) the values that possibly corrupted parties secret-shared in the first place . In [DKL+13] it was shown how to do this check on the (partially) opened values, and this is the one we also use here; see Figure 9 for details. The protocol, named MACCheck, utilizes a pseudorandom function to allow the parties agree on fresh-sampled random values. To agree on a fair seed for the PRF (i.e. a seed that is not known in advance) the parties use the ideal functionality $\mathcal{F}_{\text{Comm}}$ for commitments described in Figure 2.

We also emphasize that due to the linearity of the additive secret sharing and MACs, the check can be postponed to the end of the protocol. Note this protocol enables the checking of multiple sets of values *without* revealing the global secret key α, enabling support for reactive functionalities.

In order to understand the probability of an adversary being able to cheat during the execution of Figure 9, the authors in [DKL+13] used a security game approach, which in turn was an adaptation of the one in [DPSZ12]. For completeness, we state here both the protocol and the security game.

Game: Security Game for the MACCheck Protocol from Figure 9, 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 $a_1, \dots, a_t$ to the adversary.
2: The adversary sends back bits $b_1, \dots, b_t$.
3: The challenger generates random values $\chi_1, \dots, \chi_t \in \mathbb{F}$ and sends them to the adversary.
4: The adversary provides an error $\Delta \in \mathbb{F}$.
5: Set $b = \bigoplus_{i=1}^{t} \chi_i \cdot b_i$, and sharings $\langle\mu(a)\rangle = \bigoplus_{i=1}^{t} \chi_i \cdot \langle\mu(a_i)\rangle$, and $\langle\gamma\rangle = \langle\mu(a)\rangle \oplus b \cdot \langle\alpha\rangle$. The challenger checks that $\gamma = \Delta$.
```

Protocol Π_{MACCheck}

Let $\mathbb{F}$ be a binary finite field with $\log_2 |\mathbb{F}| = \sigma$.
The parties have a set of $[\![a_i]\!]$, sharings and public bits b_i, for $i = 1, \dots, t$, 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 $\big(\langle\alpha\rangle, \{b_i, \langle a_i\rangle, \langle\mu(a_i)\rangle\}_{i=1}^{t}\big)$, such that if the MAC values are correct then we have that $\mu(a_i) = b_i \cdot \alpha$, for all i.

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

1. Every party P_i samples a seed s_i and asks $\mathcal{F}_{\text{Comm}}$ to broadcast $\tau_i = \texttt{comm}(s_i)$.
2. Every party P_i calls $\mathcal{F}_{\text{Comm}}$ with $\texttt{open}(\tau_i)$ and all parties obtain s_j for all j.
3. Set $s = s_1 \oplus \dots \oplus s_n$.
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 $b = \bigoplus_{i=1}^{t} \chi_i \cdot b_i \in \mathbb{F}$.
6. The parties locally compute the sharings $\langle\mu(a)\rangle = \chi_1 \cdot \langle\mu(a_1)\rangle \oplus \dots \oplus \chi_t \cdot \langle\mu(a_t)\rangle$ and $\langle\sigma\rangle = \langle\mu(a)\rangle \oplus b \cdot \langle\alpha\rangle$.
7. Party i asks $\mathcal{F}_{\text{Comm}}$ to broadcast his share $\tau_i' = \texttt{comm}(\sigma_i)$.
8. Every party calls $\mathcal{F}_{\text{Comm}}$ with $\texttt{open}(\tau_i')$, and all parties obtain σ_j for all j.
9. If $\sigma_1 \oplus \dots \oplus \sigma_n \neq 0$, the parties output $\varnothing$ and abort, otherwise they accept all b_i as valid authenticated bits.

Figure 9 Method to Check MACs on Partially Opened Values

The adversary wins the game if there is an $i \in \{1, \dots, t\}$ 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.

Lemma 1 **(See [DKL+13]).** *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.*

4.2 Implementing the Online Phase for Binary Circuits

As we said earlier, we assume an ideal functionality that outputs correlated randomness to the parties. More concretely, the parties are given access to a pre-computed list of bit triples $\{[\![x]\!], [\![y]\!], [\![z]\!]\}$ such that $z = x \cdot y$ (called multiplication triples). This data will be produced utilizing the offline functionality described in Figure 10.

The parties use the data produced by $\mathcal{F}_{\text{Prep}}$ to evaluate each multiplication gate. To see how this is done, say we want to multiply two authenticated bits $[\![a]\!], [\![b]\!]$, the parties take an unused triple $\{[\![x]\!], [\![y]\!], [\![z]\!]\}$ off the list and do the following. They partially open $[\![x]\!] \oplus [\![a]\!]$ and $[\![y]\!] \oplus [\![b]\!]$ to obtain ε, ρ respectively. Then the parties locally compute the linear function $[\![c]\!] = [\![z]\!] \oplus (\varepsilon \cdot [\![y]\!]) \oplus (\rho \cdot [\![x]\!]) \oplus (\varepsilon \cdot \rho)$. Correctness is given by observing that $c = z \oplus ((x \oplus a) \cdot y) \oplus ((y \oplus b) \cdot x) \oplus ((x \oplus a) \cdot (y \oplus b)) = a \cdot b$, since $z = x \cdot y$. Our protocol to realise the functionality $\mathcal{F}_{\text{Online}}$ is then given by protocol Π_{Online} in Figure 11. And we then have the following theorem:

Theorem 3. *In the* $(\mathcal{F}_{\text{Comm}}, \mathcal{F}_{\text{Prep}})$*-hybrid model, the protocol* Π_{Online} *securely implements* $\mathcal{F}_{\text{Online}}$ *against any static adversary corrupting up to* $n-1$ *parties, assuming protocol* $\mathsf{MACCheck}$ *utilizes a secure pseudo-random function* $\mathsf{PRF}_s^{\mathbb{F},t}(\cdot)$.

Proof. 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 the Π_{Online} attached with $\mathcal{F}_{\text{Prep}}$ and $\mathcal{F}_{\text{Comm}}$, or with the simulator $\mathcal{S}$ and $\mathcal{F}_{\text{Online}}$. We start describing the behaviour of the simulator $\mathcal{S}$:

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

Let A be the set of indices of corrupt parties and $\mathcal{S}$ the ideal adversary.

Initialize:

On input (Init) from honest parties, the functionality samples random α_h for each $h \notin A$. It waits for the adversary to input corrupt shares $\{\alpha_c\}_{c \in A}$. If any $c \in A$ outputs `abort`, then the functionality aborts and returns the set of $c \in A$ which returned abort. Otherwise the functionality sets $\alpha = \alpha_1 \oplus \cdots \oplus \alpha_n$, and outputs α_h to honest P_h.

Share-to-Party:

On input (Share, i) from all the parties the functionality samples, at random, ℓ bits r^s. It proceeds as follows:

1. Generate $[\![r^s]\!] = (\langle r^s \rangle, \langle \mu^s \rangle, \langle \alpha \rangle)$, such that the bit share of P_i is r^s, and the bit share of P_j, $j \neq i$, is set to zero. The adversary specifies MAC shares of corrupt parties.
2. The adversary specifies bits $(e^s_h)_{h \notin A, s \leq \ell}$. The functionality then outputs $(\mu(r^s) \oplus (e^s_h \cdot \alpha_h))_{s \leq \ell}$ to honest P_h, and $(\mu(r^s))_{s \leq \ell}$ to corrupt P_c. Additionally, it output bits $(r^s)_{s \leq \ell}$ to P_i.

Thus, if we let $e^s = \bigoplus_{h \notin A} e^s_h \cdot \alpha_h$, then it holds that $\mu(r^s) = (r^s \cdot \alpha) \oplus e^s$, for each $s \leq \ell$.

MTriple:

On input ($\mathsf{MTriple}$) from all the parties, the functionality waits for the adversary to input `abort` or `continue` . If it is told to abort, it outputs the special symbol $\varnothing$ to all parties.

Otherwise, for $s = 1, \ldots, \ell$, it samples two random bits x^s, y^s, and sets $z^s = x^s \cdot y^s$. Then, for every bit $a^s \in \{z^s, x^s, y^s\}$ the functionality produces an authentication by calling $\mathsf{Authenticate}(a^s)$.

$\mathsf{Authenticate}(a)$:

Given a bit a, this sub-procedure produces an authentication $[\![a]\!] = (\langle a \rangle, \langle \mu(a) \rangle, \langle \alpha \rangle)$.

1. Set $\mu(a) = a \cdot \alpha$.
2. Generate input sharing $\langle a \rangle$ and MAC sharing $\langle \mu(a) \rangle$. (The adversary specifies bit shares $\{b_c\}_{c \in A}$, and MAC shares $\{\mu_c(a)\}_{c \in A}$.)

Output $(a_h, \mu_h(a))$ to P_h, for $h = 1, \ldots, n$.

Figure 10 Ideal Preprocessing

Protocol Π_{Online}

Initialize: The parties call Init on the $\mathcal{F}_{\text{Prep}}$ functionality to get the shares α_i of the global MAC key α. If $\mathcal{F}_{\text{Prep}}$ 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 b, each party P_i queries command (Share, i) of $\mathcal{F}_{\text{Prep}}$. They obtain an authenticated bit $[\![r]\!]$ only known to P_i. Party P_i broadcasts $\rho = b \oplus r$, and everyone sets $[\![r \oplus \rho]\!] = [\![b]\!]$

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

Multiply: On input $([\![a]\!], [\![b]\!])$, the parties execute either one of the folowing procedures

1. Call $\mathcal{F}_{\text{Prep}}$ on input ($\mathsf{MTriple}$), obtaining a random multiplication triple $\{[\![x]\!], [\![y]\!], [\![z]\!]\}$. The parties then perform:
   (a) The parties locally compute $[\![\varepsilon]\!] = [\![x]\!] \oplus [\![a]\!]$ and $[\![\rho]\!] = [\![y]\!] \oplus [\![b]\!]$.
   (b) The shares $[\![\varepsilon]\!]$ and $[\![\rho]\!]$ are partially opened.
   (c) The parties locally compute $[\![c]\!] = [\![z]\!] \oplus (\varepsilon \cdot [\![y]\!]) \oplus (\rho \cdot [\![x]\!]) \oplus (\varepsilon \cdot \rho)$.

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.

Figure 11 Secure Function Evaluation in the $\mathcal{F}_{\text{Comm}}, \mathcal{F}_{\text{Prep}}$-hybrid Model

– The simulation of the **Initialize** procedure is performed running a copy of $\mathcal{F}_{\text{Prep}}$ on query Init. All the data of the corrupted parties are known to the simulator. If $\mathcal{Z}$ inputs `abort` to the copy of $\mathcal{F}_{\text{Prep}}$, then

the simulator does the same to $\mathcal{F}_{\text{Online}}$ and forward the output of $\mathcal{F}_{\text{Online}}$ to $\mathcal{Z}$: If $\mathcal{F}_{\text{Online}}$ outputs `abort`, the simulator waits for input a set of corrupted parties from $\mathcal{Z}$ and forward it to $\mathcal{F}_{\text{Online}}$, 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 $\mathcal{F}_{\text{Prep}}$ sending back sharings $[x]^i_\alpha$ such that $i \in A$, 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 $\mathcal{F}_{\text{Online}}$ as inputs.
- The procedure **Add, Multiply** are performed according to the protocol and the simulator calls the respective procedure to $\mathcal{F}_{\text{Online}}$.
- In the **Output** step, the functionality $\mathcal{F}_{\text{Online}}$ 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 party and modify[9] 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 $\mathcal{F}_{\text{Online}}$.

All the steps of the protocol are perfectly simulated: during the initialization the simulator acts as $\mathcal{F}_{\text{Prep}}$; 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 Lemma 1). $\square$

Thus *all* that remains is to implement the $\mathcal{F}_{\text{Prep}}$ functionality. It is to this task that the remainder of this paper is devoted.

5 Extending Oblivious Transfer

In this section we show how we can produce a virtually unbounded number of OTs from a small number of seed OTs. The amortized work per produced OT is linear in ψ, the security parameter, or simply a few calls to a hash function. One can see this as the "core" of our technique to enable 2PC and MPC protocols. In later sections we will see how the functionalities created in this section can enable us to build the pre-processing functionality of the previous section. In fact we will concentrate on extending a few OTs to random OTs (ROTs), i.e. we will show how to generate a very large number of ROTs from a few OTs. As shown in [Bea95] one can easily obtain OT from precomputed ROTs, thus this is essentially without loss of generality.

The relationship between the various functionalities in this section is described in Figure 12. We note that all functionalities in this section are two party functionalities. To implement our OT-extension protocol we go via yet an other variant of OT we call Δ-ROT, inspired by an intermediate step of the highly efficient semi-honest OT-extender of [IKNP03]. A Δ-ROT is a variant of ROT where the senders messages M_0 and M_1 are correlated in such a way that $M_0 \oplus M_1 = \Delta$, for some constant Δ unknown to P_2, which we will call the *global key*. Given a functionality that provides Δ-ROTs it is very easy to implement ROT, thus the main work of this section goes into extending a few OTs to many Δ-ROTs efficiently and with malicious security.

Overview:

- As we shall see our construction goes via a number of leaky functionalities, i.e. functionalities where the adversary might mount an attack to gain some leakage. Therefore, we will start this section by introducing the idea of a *leakage agent* to abstract away the concrete attacks of the adversary.

[9] A little extra bookkeeping is needed at this point if two outputs are linearly related, this minor modification is left to the reader.

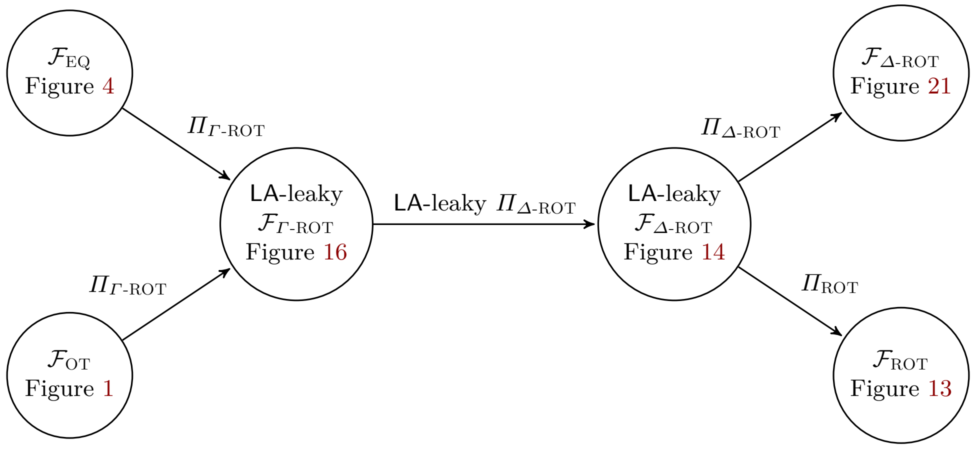

$\mathcal{F}_{\text{EQ}}$ Figure 4
$\Pi_{\Gamma\text{-ROT}}$
$\mathcal{F}_{\text{OT}}$ Figure 1
$\Pi_{\Gamma\text{-ROT}}$
LA-leaky $\mathcal{F}_{\Gamma\text{-ROT}}$ Figure 16
LA-leaky $\Pi_{\Delta\text{-ROT}}$
LA-leaky $\mathcal{F}_{\Delta\text{-ROT}}$ Figure 14
$\Pi_{\Delta\text{-ROT}}$
$\mathcal{F}_{\Delta\text{-ROT}}$ Figure 21
Π_{ROT}
$\mathcal{F}_{\text{ROT}}$ Figure 13

Figure 12 Relationship Between the OT Functionalities

- In Section 5.2 we implement $\mathcal{F}_{\text{ROT}}$ using a functionality providing a variant of Δ-ROT which are *leaky*. By leaky we mean that P_2 may learn a few bits of the global key Δ. We call this functionality a leaky $\mathcal{F}_{\Delta\text{-ROT}}$ functionality. The implementation simply uses a hash function (modeled as a random oracle) to break the correlation between messages from $\mathcal{F}_{\Delta\text{-ROT}}$.
- In Section 5.3 we then consider a functionality similar to the $\mathcal{F}_{\Delta\text{-ROT}}$ functionality but with reversed roles, i.e. with P_1 as the receiver and P_2 as sender. We call this functionality $\mathcal{F}_{\Gamma\text{-ROT}}$. We notice that if we relax this functionality, by allowing a few of P_1's choice bits to leak to P_2 the resulting *leaky* $\mathcal{F}_{\Gamma\text{-ROT}}$ functionality is essentially equivalent to the leaky $\mathcal{F}_{\Delta\text{-ROT}}$ functionality, i.e. the leakage on choice bits in $\mathcal{F}_{\Gamma\text{-ROT}}$ becomes leakage on Δ in $\mathcal{F}_{\Delta\text{-ROT}}$.
- In Section 5.4 we implement the leaky $\mathcal{F}_{\Gamma\text{-ROT}}$ functionality using a few OT's in the following way: P_1 inputs random choice bits c_i in an OT, and P_2 is supposed to input random messages $(N_0^i, N_0^i \oplus \Gamma)$ so that P_1 receives $N_{c_i}^i = N_0^i \oplus c_i\Gamma$. To ensure that P_2 is being honest and uses the same Γ in most OTs a check is performed, which restricts P_2 to cheat in at most a few OTs. We notice that what P_2 gains by using inconsistent Γ's in a few OTs is no more than learning a few of P_1's choice bits c_i, thus implementing the $\mathcal{F}_{\Gamma\text{-ROT}}$ functionality.
- While the leaky $\mathcal{F}_{\Delta\text{-ROT}}$ functionality is all we need for our OT-extension protocol, it is useful to have a non-leaky version of this functionality. The non-leaky version is used in sections Section 6 and Section 7 to implement the earlier $\mathcal{F}_{\text{PREP}}$ functionality. Therefore, in Section 5.7, using privacy amplification, we use the leaky $\mathcal{F}_{\Delta\text{-ROT}}$ functionality to implement a $\mathcal{F}_{\Delta\text{-ROT}}$ functionality where the Δ value is shorter than in the leaky functionality but fully secure.
- Finally in Section 5.8 we sketch a complexity analysis counting the symmetric primitives used in the protocol.

5.1 The Leakage Agent

As described above, our construction will go via a number of functionalities that are leaky in some way. The setting is that there is a secret string $\Delta \in_{\text{R}} \{0,1\}^\tau$ and an adversary P_1 who tries to guess Δ. Party P_1 can launch an attack which might leak some of the bits of Δ, but with some probability P_1's attack will be detected. In this section we introduce the idea of a *leakage agent*, in order to abstract away the concrete leakage attacks.

A leakage agent LA will be an interactive PPT algorithm that will interact with P_1 and then possibly specify what leakage to give to P_1: P_1 gets to first interact with LA in some way (specified by the interface of LA). Following this interaction we input $\Delta \in_{\text{R}} \{0,1\}^\tau$ to LA. Based on the interaction with P_1 LA computes some $S \subseteq [\tau]$ and $c \in \{0,1\}$ and outputs (S, c). Here S indicates the bits of Δ to be leaked to P_1 if the attack is undetected, and c whether or not the attack is detected ($c = 0$ indicating detection). This models that during P_1's attack, i.e. the interaction with LA, no bits of Δ leak, but after the attack the set of bits that does leak may depend on Δ it self.

We need a measure of how many bits a leakage agent LA leaks. We do this via a game against an unbounded adversary P_1.

$$\begin{array}{l}
\underline{\text{LeakageGame}(\mathsf{LA}, P_1, \tau)} \\
\Delta \in_{\mathrm{R}} \{0,1\}^{\tau} \\
P_1(\tau) \leftrightarrow \mathsf{LA}(\tau) \\
(S, c) \leftarrow \mathsf{LA}(\Delta) \\
(g_i)_{i \in \bar{S}} \leftarrow P_1(S, (\Delta_i)_{i \in S})
\end{array}$$

Where $\bar{S} = [\tau] \setminus S$ and $P_1(\tau) \leftrightarrow \mathsf{LA}(\tau)$ means P_1 and LA interact each given τ as input.

We say that P_1 wins if $c = 1$ and $(\Delta_i)_{i \in \bar{S}} = (g_i)_{i \in \bar{S}}$. Furthermore, we say that an adversary P_1 is *optimal* if she has the highest probability of any adversary of winning LeakageGame(LA, P_1, τ). If there were no leakage, i.e., $S = \emptyset$, then it is clear that an optimal P_1 wins the game with probability exactly $2^{-\tau}$. If P_1 is always given exactly s bits and is never detected, then it is clear that an optimal P_1 can win the game with probability exactly $2^{s-\tau}$. This motivates defining the number of bits leaked by LA to be $\text{leak}_{\mathsf{LA}} \overset{\text{def}}{=} \log_2(\text{success}_{\mathsf{LA}}) + \tau$, where $\text{success}_{\mathsf{LA}}$ is the probability that an optimal P_1 wins the leakage game. We say that LA is κ-secure if $\tau - \text{leak}_{\mathsf{LA}} \geq \kappa$ for all large enough τ, and if LA is κ-secure, then no P_1 can win the game with probability better than $2^{-\kappa}$ (asymptotically).

We now rewrite the definition of $\text{leak}_{\mathsf{LA}}$ to make it more workable. We denote by $(\mathsf{LA}, P_1)(\tau)$ the experiment sampling (S, c) as in LeakageGame(LA, P_1, τ). It is clear that an optimal P_1 can guess all Δ_i for $i \in \bar{S}$ with probability exactly $2^{|S|-\tau}$. This means that an optimal P_1 wins with probability

$$\sum_{s=0}^{\tau} \Pr\left((S, c) \leftarrow (\mathsf{LA}, P_1)(\tau) : |S| = s \wedge c = 1\right) 2^{s-\tau} \ .$$

To simplify this expression we define index variables $I_s, J_s \in \{0,1\}$ where I_s is 1 iff $c = 1$ and $|S| = s$ and J_s is 1 iff $|S| = s$ when $(S, c) \leftarrow (\mathsf{LA}, P_1)(\tau)$. Note that $I_s = cJ_s$ and that $\sum_s J_s 2^s = 2^{|S|}$. So, if we take expectation over $(S, c) \leftarrow (\mathsf{LA}, P_1)(\tau)$, then we get that

$$\sum_{s=0}^{\tau} \Pr\left((S, c) \leftarrow (\mathsf{LA}, P_1)(\tau) : |S| = s \wedge c = 1\right) 2^{s} = \sum_{s=0}^{\tau} \mathrm{E}\left[I_s\right] 2^s \ ,$$

where

$$\sum_{s=0}^{\tau} \mathrm{E}\left[I_s\right] 2^s = \mathrm{E}\left[\sum_{s=0}^{\tau} I_s 2^s\right] = \mathrm{E}\left[\sum_{s=0}^{\tau} cJ_s 2^s\right] = \mathrm{E}\left[c \cdot \sum_{s=0}^{\tau} J_s 2^s\right] = \mathrm{E}\left[c \cdot 2^{|S|}\right] \ .$$

Hence $\text{success}_{\mathsf{LA}} = \max_{P_1}(2^{-\tau}\, \mathrm{E}\left[c 2^{|S|}\right])$ and

$$\log_2(\text{success}_{\mathsf{LA}}) = -\tau + \log_2 \max_{P_1} \left(\mathrm{E}\left[c 2^{|S|}\right]\right) \ ,$$

which shows that

$$\text{leak}_{\mathsf{LA}} = \max_{P_1} \log_2 \left(\mathrm{E}\left[c 2^{|S|}\right]\right) \ .$$

5.2 $\mathcal{F}_{\mathrm{ROT}}$ from $\mathcal{F}_{\Delta\text{-ROT}}$

In this section we show how we implement the $\mathcal{F}_{\mathrm{ROT}}$ as described in Figure 13. Note that this functionality will be the result of our OT-extension.

The Functionality $\mathcal{F}_{\mathrm{ROT}}(\ell, \psi)$

Honest Parties

On input `start` from both P_1 and P_2 the functionality does the following.

1. The functionality samples $(X_0^i, X_1^i)_{i\in[\ell]} \in_{\mathrm{R}} \{0,1\}^{2\psi\ell}$, and outputs $(X_0^i, X_1^i)_{i\in[\ell]}$ to P_1.
2. The functionality samples $(b_i)_{i\in[\ell]} \in_{\mathrm{R}} \{0,1\}^{\ell}$ and outputs $(X_{b_i}^i, b_i)_{i\in[\ell]}$ to P_2.

Corrupt Parties

1. If P_1 is corrupt the functionality waits for her to input $(X_0^i, X_1^i)_{i\in[\ell]} \in \{0,1\}^{2\psi\ell}$. The functionality then outputs as above using these values.
2. If P_2 is corrupt the functionality waits for him to input $(X_{b_i}^i, b_i)_{i\in[\ell]} \in \{0,1\}^{\ell(\psi+1)}$. The functionality then outputs as above using these values.

Figure 13 The Random OT Functionality $\mathcal{F}_{\mathrm{ROT}}(\ell, \psi)$

To this end we introduce the notion of a *Δ-ROT*. In contrast to ROT, in Δ-ROT the sender P_1 first receives random *global key* Δ and then for each Δ-ROT P_1 receives only one random message M_0^i. The *other* message is defined to be $M_1^i = M_0^i \oplus \Delta$, i.e. P_2 receives random choice bit b_i and message $M_{b_i}^i = M_0^i \oplus b_i\Delta$. To implement ROT it will suffice for us to consider a *leaky* variant of Δ-ROT. Namely, a variant where P_2 may learn some bits of the global key Δ. We call the functionality that provides such leaky Δ-ROTs a LA-leaky $\mathcal{F}_{\Delta\text{-ROT}}$ functionality and describe it in detail Figure 14, where LA is a leakage agent as described above. As long as the leakage agent LA is ψ-secure such a functionality turns out to be enough to implement ROT.

The Functionality LA-leaky $\mathcal{F}_{\Delta\text{-ROT}}(\ell, \tau)$

Honest Parties

On input `start` from both P_1 and P_2 the functionality does the following.

1. The functionality samples $\Delta \in_{\mathrm{R}} \{0,1\}^{\tau}$ and outputs it to P_1.
2. For all $i \in [\ell]$ the functionality samples $b_i \in_{\mathrm{R}} \{0,1\}$ and $M_0^i \in_{\mathrm{R}} \{0,1\}^{\tau}$.
3. The functionality outputs $(M_0^i \oplus b_i\Delta, b_i)_{i\in[\ell]}$ to P_2 and $(M_0^i)_{i\in[\ell]}$ to P_1.

Corrupt Parties

1. If P_2 is corrupt the functionality runs LA to sample (S, c), with P_2 playing the role of the adversary and where the functionality inputs Δ to LA as its secret string. If $c = 0$ the functionality outputs `fail` to P_1 and terminates. Otherwise, the functionality outputs $(i, \Delta_i)_{i\in S}$ to P_2.
2. Furthermore, if P_2 is corrupt, the functionality waits to give output till it receives the message $(\hat{M}_{\hat{b}_i}^i, \hat{b}_i)_{i\in[\ell]}$ from P_2, where $\hat{M}_{\hat{b}_i}^i \in \{0,1\}^{\tau}$ and $\hat{b}_i \in \{0,1\}$. The functionality then sets $b_i = \hat{b}_i$ and $M_0^i = \hat{M}_{b_i}^i \oplus b_i\Delta$ and outputs as described above.
3. If P_1 is corrupt, the functionality waits to give output till it receives the message $(\hat{\Delta}, (\hat{M}_0^i)_{i\in[\ell]})$ from P_1, where $\hat{\Delta}, \hat{M}_0^i \in \{0,1\}^{\tau}$. The functionality then sets $\Delta = \hat{\Delta}$ and $M_0^i = \hat{M}_0^i$ and outputs as described above.

Figure 14 The LA-leaky $\mathcal{F}_{\Delta\text{-ROT}}(\ell, \tau)$ Functionality

To implement $\mathcal{F}_{\mathrm{ROT}}$ using LA-leaky $\mathcal{F}_{\Delta\text{-ROT}}$, we notice that the $\mathcal{F}_{\Delta\text{-ROT}}$ functionality resembles an intermediate step of the passive-secure OT extension protocol of [IKNP03]: $\mathcal{F}_{\Delta\text{-ROT}}$ is a random OT, where all the sender's messages are correlated, so that the XOR of the messages in any OT is a constant (the global key of the $\mathcal{F}_{\Delta\text{-ROT}}$). This correlation can be easily broken using the random oracle. This idea leads to the protocol for $\mathcal{F}_{\mathrm{ROT}}$ described in Figure 15.

Theorem 4. *Let ψ be the security parameter and LA be ψ-secure on τ bits. Then the protocol in Figure 15 securely implements $\mathcal{F}_{ROT}(\ell, \psi)$ in the LA-leaky $\mathcal{F}_{\Delta\text{-ROT}}(\ell, \tau)$-hybrid model. The work*[10]. *is $O(\tau\ell)$*

[10] counting hashing of τ bits as $O(\tau)$ work.

Protocol Π_{ROT}

1. P_1 and P_2 call a LA-leaky $\mathcal{F}_{\Delta\text{-ROT}}(\ell,\tau)$ functionality. The output to P_2 is $(M^i_{b_i}, b_i)_{i\in[\ell]}$. The output to P_1 is $(\Delta, (M^i_0)_{i\in[\ell]})$.
2. P_2 computes $Y^i = H(M^i_{b_i}) \in \{0,1\}^{\psi}$ and outputs $(Y^i, b_i)_{i\in[\ell]}$.
3. P_1 computes $X^i_0 = H(M^i_0) \in \{0,1\}^{\psi}$ and $X^i_1 = H(M^i_0 \oplus \Delta) \in \{0,1\}^{\psi}$ and outputs $(X^i_0, X^i_1)_{i\in[\ell]}$.

Figure 15 The Protocol for Reducing $\mathcal{F}_{\mathrm{ROT}}(\ell,\psi)$ to LA-leaky $\mathcal{F}_{\Delta\text{-ROT}}(\ell,\psi)$

Proof. Correctness is simple: we have that $M^i_{b_i} = M^i_0 \oplus b_i\Delta$, so $Y^i = H(M^i_{b_i}) = H(M^i_0 \oplus b_i\Delta) = X^i_{b_i}$. Clearly the protocol leaks no information on the b_i as there is no communication from P_2 to P_1. It is therefore sufficient to look at the case of a corrupt P_2. We are not going to give a simulation argument but just show that $X^i_{1\oplus b_i}$ is uniformly random to P_2 except with probability $\mathrm{poly}(\psi)2^{-\psi}$.

Since $X^i_{1\oplus b_i} = H(M^i_0 \oplus (1\oplus b_i)\Delta)$ and H is a random oracle, it is clear that $X^i_{1\oplus b_i}$ is uniformly random to P_2 until P_2 queries H on $Q = M^i_0 \oplus (1\oplus b_i)\Delta$. Since $M^i_{b_i} = M^i_0 \oplus b_i\Delta$ we have that $Q = M^i_0 \oplus (1\oplus b_i)\Delta$ would imply that $M^i_{b_i} \oplus Q = \Delta$. So, if we let P_2 query H, say, on $Q \oplus M^i_{b_i}$ each time it queries H on some Q, which would not change its asymptotic running time, then we have that all $X^i_{1\oplus b_i}$ are uniformly random to P_2 until it queries H on Δ. However this happens with probability at most $2^{-\psi}$ by the ψ-security of LA. Namely notice that all inputs to P_2 are independent of Δ except for what leakage he may get from LA. Thus, a P_2 that queries H on Δ with probability better than $2^{-\psi}$ would contradict the ψ-security of LA. ◻

5.3 Leaky $\mathcal{F}_{\Delta\text{-ROT}}$ From Leaky $\mathcal{F}_{\Gamma\text{-ROT}}$

We now consider a functionality similar to the leaky $\mathcal{F}_{\Delta\text{-ROT}}$ functionality, but reversed in the sense that P_1 plays the role of receiver and P_2 the role of sender and that the leakage is on the choice bits instead of the global key, i.e. P_2 receives a random global key Γ and random messages N^i_0, while P_1 receives random choice bits c_i and messages $N^i_{c_i} = N^i_0 \oplus c_i\Gamma$. If P_2 is corrupt he may learn some of the bits c_i. We call this functionality a leaky $\mathcal{F}_{\Gamma\text{-ROT}}$, and formally describe it in Figure 16.

The Functionality LA-leaky $\mathcal{F}_{\Gamma\text{-ROT}}(\tau,\ell)$

Honest Parties

On input `start` from both P_1 and P_2 the functionality does the following.

1. The functionality samples $\Gamma \in_{\mathrm{R}} \{0,1\}^{\ell}$ and outputs it to P_2.
2. For all $i \in [\tau]$ the functionality samples $c_i \in_{\mathrm{R}} \{0,1\}$ and $N^i_0 \in_{\mathrm{R}} \{0,1\}^{\tau}$.
3. The functionality outputs $(N^i_0 \oplus c_i\Gamma, c_i)_{i\in[\tau]}$ to P_1 and $(N^i_0)_{i\in[\tau]}$ to P_2.

Corrupt Parties

1. If P_2 is corrupt the functionality runs LA to sample (S, c), with P_2 playing the role of the adversary and where the functionality inputs $(c_i)_{i\in[\tau]}$ to LA as its secret string. If $c = 0$ the functionality outputs `fail` to P_2 and terminates. Otherwise, the functionality outputs $(i, c_i)_{i\in S}$ to P_2.
2. As in the Figure 14 any corrupt party is allowed to specify the value of its output.

Figure 16 The LA-leaky $\mathcal{F}_{\Gamma\text{-ROT}}(\tau,\ell)$ Functionality

The LA-leaky $\mathcal{F}_{\Gamma\text{-ROT}}(\tau,\ell)$ provides τ leaky Γ-ROTs with messages of length ℓ. It turns out that the leaky $\mathcal{F}_{\Gamma\text{-ROT}}$ functionality is actually equivalent to the leaky $\mathcal{F}_{\Delta\text{-ROT}}$ functionality, i.e. given one functionality simply renaming the outputs gives us the other, as demonstrated in Figure 17. We say this more formally in Theorem 5.

Theorem 5. *For all ℓ, τ and LA the LA-leaky $\mathcal{F}_{\Delta\text{-ROT}}(\ell,\tau)$ and $\mathcal{F}_{\Gamma\text{-ROT}}(\tau,\ell)$ functionalities are linear locally equivalent, i.e., can be implemented given the other in linear time without interaction.*

Protocol LA-leaky $\Pi_{\Delta\text{-ROT}}$

1. P_1 and P_2 invoke a LA-leaky $\mathcal{F}_{\Gamma\text{-ROT}}(\tau, \ell)$ functionality. P_1 learns $(N^i_{c_i}, c_i)_{i\in[\tau]}$ and P_2 learns $(\Gamma, (N^i_0)_{i\in[\tau]})$.
2. P_2 lets b_j be the j'th bit of Γ and $M^j_{b_j}$ the string consisting of the j'th bits from all the strings N^i_0, i.e. $M^j_{b_j} = N^1_{0,j}||N^2_{0,j}||\dots||N^\ell_{0,j}$.
3. P_1 lets Δ be the string consisting of all the bits c_i, i.e. $\Delta = c_1||c_2||\dots||c_\tau$, and lets M^j_0 be the string consisting of the j-th bits from all the strings $N^i_{c_i}$, i.e. $M^j_0 = N^1_{c_1,j}||N^2_{c_2,j}||\dots||N^\tau_{c_\tau,j}$. [a]

[a] It may be easier to think of this renaming in terms of bit matrices as in (1)

Figure 17 Protocol Reducing LA-leaky $\mathcal{F}_{\Delta\text{-ROT}}(\ell, \tau)$ to LA-leaky $\mathcal{F}_{\Gamma\text{-ROT}}(\tau, \ell)$

Proof. The first direction (reducing leaky $\mathcal{F}_{\Delta\text{-ROT}}$ to $\mathcal{F}_{\Gamma\text{-ROT}}$) is shown in Figure 17. The other direction ($\mathcal{F}_{\Gamma\text{-ROT}}$ to $\mathcal{F}_{\Delta\text{-ROT}}$) will follow by the fact that the renamings are reversible in linear time. One can easily verify that for all $j \in [\ell]$, $M^j_{b_j}$ is the correct message given choice bit b_j and the values M^j_0 and Δ, i.e. $M^j_{b_j} = M^j_0 \oplus b_j\Delta$. This is perhaps easiest seen by viewing the renaming described in Figure 17 in terms of bit matrices where the strings $N^i_c, \Gamma \in \{0,1\}^\ell$ are viewed as column vectors and strings $M^j_b, \Delta \in \{0,1\}^\tau$ are viewed as row vectors. Taking this view the following holds

$$\begin{aligned} \begin{pmatrix} | & & | \\ N^1_{c_1} & \dots & N^\tau_{c_\tau} \\ | & & | \end{pmatrix} &= \begin{pmatrix} | & & | \\ N^1_0 & \dots & N^\tau_0 \\ | & & | \end{pmatrix} \oplus \begin{pmatrix} | & & | \\ c_1\Gamma & \dots & c_\tau\Gamma \\ | & & | \end{pmatrix} \\ &= \begin{pmatrix} \text{—}\ M^1_{b_1}\ \text{—} \\ \vdots \\ \text{—}\ M^\ell_{b_\ell}\ \text{—} \end{pmatrix} \oplus \begin{pmatrix} \text{—}\ b_1\Delta\ \text{—} \\ \vdots \\ \text{—}\ b_\ell\Delta\ \text{—} \end{pmatrix} \\ &= \begin{pmatrix} \text{—}\ M^1_0\ \text{—} \\ \vdots \\ \text{—}\ M^\ell_0\ \text{—} \end{pmatrix} \in \{0,1\}^{\ell\times\tau}, \end{aligned} \tag{1}$$

where the first equality is by definition of $\mathcal{F}_{\Gamma\text{-ROT}}$ and the second is by design of the protocol in Figure 17.

It is easy to verify (as the protocol only consists of renamings) that leakage on the choice bits c_i is equivalent to leakage on the global key Δ under this transformation. Giving a simulation argument is then straight forward when LA is the same for both functionalities. ◻

Note that doing this simple renaming we turn a LA-leaky $\mathcal{F}_{\Gamma\text{-ROT}}(\ell, \tau)$ functionality into a LA-leaky $\mathcal{F}_{\Delta\text{-ROT}}(\tau, \ell)$ functionality. If we choose $\ell = \text{poly}(\psi)$ this means that we can turn τ leaky Γ-ROTs into a very larger number (ℓ) of leaky Δ-ROTs, i.e. implementing the leaky $\mathcal{F}_{\Gamma\text{-ROT}}$ functionality for a ψ-secure leakage agent LA on $\tau = O(\psi)$ bits, using only $\mathcal{F}_{\text{OT}}(\tau, \ell)$ we have our OT-extension protocol. In the following section we give such an implementation of the leaky $\mathcal{F}_{\Gamma\text{-ROT}}$ functionality.

5.4 A Protocol For leaky $\mathcal{F}_{\Gamma\text{-ROT}}$

In this section we show how to construct leaky $\mathcal{F}_{\Gamma\text{-ROT}}$ from $\mathcal{F}_{\text{OT}}$. The protocol ensures that most of the choice bits are kept secret. The main idea of the protocol, described in Figure 18, is the following: P_2 and P_1 run many OTs using an $\mathcal{F}_{\text{OT}}$ functionality. In the i'th OT P_2 inputs messages N^i_0 and N^i_1 and P_1 inputs choice bit c_i. An honest P_2 sets her messages so that $N^i_0 \oplus N^i_1 = \Gamma$ for all i. To test that P_2 used the same value for Γ in every OT the parties randomly partition the OTs into pairs. Say that one such pair consists of the i'th and j'th OT. P_1 then sends $d = c_i \oplus c_j$ to P_2 and computes $D = N^i_{c_i} \oplus N^i_{c_j}$. If P_2 is honest he can also compute D as $D = N^i_0 \oplus N^i_0 \oplus d\Gamma$. On the other hand if he used different values for Γ in the i'th and

j'th OT he can guess D with at most probability $\frac{1}{2}$, as we shall demonstrate below. Therefore, to test that P_2 behaved honestly the parties can use the $\mathcal{F}_{\mathrm{EQ}}$ functionality to test that they are both able to compute D. In case of inequality the protocol is aborted since this will indicate that one party is corrupt. Recall that the $\mathcal{F}_{\mathrm{EQ}}$ functionality leaks its inputs in case of inequality. However, as inequality will lead to the protocol being aborted before output is given and there is no private input, this does not course any insecurity. As P_1 reveals $c_i \oplus c_j$, the parties waste the j'th OT and only use the output of the i'th OT as output from the protocol—since c_j is uniformly random $c_i \oplus c_j$ leaks no information on c_i. Note that we cannot simply let P_2 reveal the D, as a malicious P_1 could send $d = 1 \oplus c_i \oplus c_j$: this would allow P_1 to learn both D and $D \oplus \Gamma$, thus leaking Γ. Using $\mathcal{F}_{\mathrm{EQ}}$ forces an P_1 who uses this attack to make the protocol abort unless she can guess a random message, which she can do only with negligible probability $2^{-\ell}$.

Protocol $\Pi_{\Gamma\text{-ROT}}$

1. P_2 samples $\Gamma \in_{\mathrm{R}} \{0,1\}^{\ell}$ and for $i = 1, \ldots, 2\tau$ samples $N_0^i \in_{\mathrm{R}} \{0,1\}^{\ell}$.
2. P_1 samples $(c_1, \ldots, c_{2\tau}) \in_{\mathrm{R}} \{0,1\}^{2\tau}$.
3. The parties run a $\mathcal{F}_{\mathrm{OT}}(2\tau, \ell)$ functionality, where for $i = 1, \ldots, 2\tau$ P_2 inputs messages N_0^i and $N_0^i \oplus \Gamma$. P_1 inputs choice bit c_i and receives $N_{c_i}^i = N_0^i \oplus c_i \Gamma$.
4. P_1 picks a uniformly random pairing π (a permutation $\pi : [2\tau] \to [2\tau]$ where $\forall i, \pi(\pi(i)) = i$ and $\forall i, \pi(i) \neq i$), and sends π to P_2. Given a pairing π, let $\mathcal{L}_\pi = \{i | i \leq \pi(i)\}$, i.e., for each pair, add the smallest index to $\mathcal{L}_\pi$.
5. For all τ indices $i \in \mathcal{L}_\pi$:
   (a) P_1 announces $d_i = c_i \oplus c_{\pi(i)}$.
   (b) P_1 computes $D_i = N_{c_i}^i \oplus N_{c_j}^j$ and P_2 computes $\hat{D}_i = N_0^i \oplus N_0^j \oplus d_i \Gamma$, where $j = \pi(i)$.

   The parties then compare the strings $(D_i)_{i \in \mathcal{L}_\pi}$ and $(\hat{D}_i)_{i \in \mathcal{L}_\pi}$ using $\mathcal{F}_{\mathrm{EQ}}(\tau\ell)$ and abort if they are different. Otherwise the protocol continues.
6. P_2 outputs $(\Gamma, (N_0^i)_{i \in \mathcal{L}_\pi})$ and P_1 outputs $(N_{c_i}^i, c_i)_{i \in \mathcal{L}_\pi}$.

Figure 18 The Protocol for Reducing $\mathcal{F}_{\Gamma\text{-ROT}}(\tau, \ell)$ to $\mathcal{F}_{\mathrm{OT}}(2\tau, \ell)$ and $\mathcal{F}_{\mathrm{EQ}}(\tau\ell)$.

Theorem 6 tell us that the protocol $\Pi_{\Gamma\text{-ROT}}$ described in Figure 18 securely implements the LA-leaky $\mathcal{F}_{\Gamma\text{-ROT}}(\tau, \ell)$ functionality. Proving Theorem 6 will take a lot of work and therefore, we push the proof to Section 5.6 and immediately present Corollary 1 summing up the results of this section so far.

Theorem 6. *Let $\kappa = 0.6\tau$, and let LA be a κ-secure leakage agent on τ bits defined below in the proof. The protocol in Figure 18 securely implements LA-leaky $\mathcal{F}_{\Gamma\text{-ROT}}(\tau, \ell)$ in the $(\mathcal{F}_{\mathrm{OT}}(2\tau, \ell), \mathcal{F}_{\mathrm{EQ}}(i\tau\ell))$-hybrid model. The communication is $O(\tau)$. The work is $O(\tau\ell)$.*

Corollary 1. *Let ψ denote the security parameter and let $\ell = \mathrm{poly}(\psi)$. Let $\alpha = \frac{10}{6}$. The functionality $\mathcal{F}_{\mathrm{ROT}}(\ell, \psi)$ can be reduced to $(\mathcal{F}_{\mathrm{OT}}(2\alpha\psi, \psi), \mathcal{F}_{\mathrm{EQ}}(\psi))$. The communication is $O(\ell\psi)$ and the work is $O(\psi\ell)$.*

Proof. Combining Theorem 4, 5 and 6 and setting $\tau = \alpha\psi$ we have that $\mathcal{F}_{\mathrm{ROT}}(\ell, \psi)$ can be reduced to $(\mathcal{F}_{\mathrm{OT}}(2\alpha\psi, \ell), \mathcal{F}_{\mathrm{EQ}}(\alpha\psi\ell))$ with communication $O(\ell\psi)$ and work $O(\psi\ell)$. For any polynomial ℓ, we can implement $\mathcal{F}_{\mathrm{OT}}(2\alpha\psi, \ell)$ given $\mathcal{F}_{\mathrm{OT}}(2\alpha\psi, \psi)$ and a pseudo-random generator prg : $\{0,1\}^{\psi} \to \{0,1\}^{\ell}$. Namely, seeds are sent using the OTs and the prg is used to one-time pad encrypt the messages. The communication is 2ℓ. If we use the RO to implement the pseudo-random generator and count the hashing of ψ bits as $O(\psi)$ work, then the work is $O(\ell\psi)$ (using ℓ/ψ calls to H to expand ψ bit seeds to ℓ bits). We can implement $\mathcal{F}_{\mathrm{EQ}}(\alpha\psi\ell)$ by comparing short hashes produced using the RO. The work is $O(\psi\ell)$ (using $\alpha\ell$ calls to H to hash the $\alpha\psi\ell$-bit string down to ψ-bits). □

Since the oracles $(\mathcal{F}_{\mathrm{OT}}(2\alpha\psi, \psi), \mathcal{F}_{\mathrm{EQ}}(\psi))$ are independent of ℓ, the cost of essentially any reasonable implementation of them can be amortized away by picking ℓ large enough.

Before we prove Theorem 6 we prove a technical lemma used in the proof of Theorem 6. We prefer to prove it up front as the proof is extensive but carries little intuition on the security of the protocol.

5.5 The Sock Game

Consider the following Sock Game. It is parametrized by an integer τ. It is played by two players called Color and Sampler, denoted by C and S.

The Moves First C makes a move by picking 2τ socks. The strategy consists of picking the colours of the socks. The colours of the socks will be non-negative integers. Let c_i denote the number of socks of colour i. We require that $c_0 \geq c_i$ for all i. Then Sampler makes a move, which is to pair up the socks into τ pairs. The pairing is done uniformly at random.

Scoring the Game Now for each pair of socks which do not have matching colours a coin is put on the table. Also for each pair of socks where both have colour 0 a coin is put on the table. Let c denote the number of coins on the table. Clearly $0 \leq c \leq \tau$. Now all the coins are tossed. If they all come out heads up, then C wins. Otherwise S wins, i.e., once the number of coins is fixed the winning probability is 2^{-c}. The goal of C is therefore to try to make a small number of coins under the restriction that colour 0 must be the majority colour. We will sometimes refer to colour 0 as *white*, thus capturing the idea that a pair of matching white socks is essentially as bad as a pair of mismatched colour.

Winning Probability We are interested in finding an upper bound on the winning probability of the best strategy for C. It is clear that the best winning probability can be obtained using a deterministic strategy by C, so we will focus on deterministic strategies. There are some easy cases. For $\tau = 1$ it is optimal to put two 0-socks. The winning probability is $\frac{1}{2}$. For $\tau = 2$ it is clearly optimal to put two 0-socks and two 1-socks. There will be two coins unless the 1-socks match up, which happens with probability $\frac{1}{3}$, so the winning probability is $\frac{1}{3}2^{-1} + (1 - \frac{1}{3})2^{-2} = \frac{1}{3}$.

Notation Note that since the pairing is uniformly random, the order in which C colours the socks does not affect the winning probability. Therefore, we will describe each strategy simply by the number of socks of each colour, i.e., by the sequence $\{c_i\}_{i=0}^{2\tau-1}$ and disregard the ordering. In our analysis we will consider strategies derived from other strategies by changing the color of a single sock, say, from colour j to colour k. Let $\sigma = \{c_i\}_{i=0}^{2\tau-1}$ be an arbitrary strategy with $c_j > 0$, we will denote by $\sigma_{k \leftarrow j} = \{c'_i\}_{i=0}^{2\tau-1}$ the strategy where $c'_j = c_j - 1$, $c'_k = c_k + 1$ and $c'_i = c_i$ otherwise. To conviniently extend this notation to the case of changing the colour of multiple socks we will write $\sigma_{j \leftarrow k, l \leftarrow m}$ rather than $(\sigma_{j \leftarrow k})_{l \leftarrow m}$. Additionally, for a given strategy σ we will denote by $W(\sigma)$ the event of C winning the game using σ.

We are going to prove the following theorem.

Theorem 7. *For any strategy σ we have $\Pr[W(\sigma)] \leq e^{1/2}2^{-0.64\tau}$.*[11]

We first prove a theorem saying that it is better for a non-white sock to be part of a larger colour than a smaller colour.

Lemma 2 (Recolour). *For any strategy $\sigma = \{c_i\}_{i=0}^{2\tau-1}$ where $c_0 > c_1 \geq c_2 > 0$ we have*

$$Pr[W(\sigma_{1 \leftarrow 2})] > \Pr[W(\sigma)] \ .$$

Proof. Let $\sigma^1 = \sigma_{1 \leftarrow 2}$. We want to prove that

$$\Pr[W(\sigma^1)] > \Pr[W(\sigma)] \ .$$

[11] Note that this bound is worse than the bound claimed in Theorem 4 in [NNOB12]. We here get 0.64τ-bit security instead of 0.75τ-bit security as claimed in [NNOB12]. The bound in [NNOB12] is most likely wrong, and the proof is certainly wrong. The mistakes occurs in Lemma 11 in the full version of [NNOB12] where Jensen's inequality was used in the wrong direction. The below Sock Game basically replaces Lemma 11 and Lemma 12 in the full version of [NNOB12].

Assume without loss of generality that $c_3 = 0$. This can be ensured by recolouring all socks of colour 3 to some other unused colour without changing the winning probability. Let $\sigma^3 = \sigma_{3\leftarrow 2}$. Since $c_1 \geq c_2$ we clearly have that

$$\Pr[W(\sigma^3_{1\leftarrow 3})] > \Pr[W(\sigma^3_{2\leftarrow 3})] \ .$$

Namely, it is better to colour the single sock of colour 3 into one of colour 1 than a sock of colour 2 as there are more socks of colour 1 than colour 2 in σ^3 and the only difference that the recolouring can make is that the sock previously of colour 3 will match its partner after the recolouring. Now notice that $\sigma^3_{1\leftarrow 3} = \sigma^1$ and $\sigma^3_{2\leftarrow 3} = \sigma$. □

We then prove the *Bleaching Lemma* which bounds the effect of modifying a strategy by taking one sock of the majority non-white colour and changing it to a white sock (i.e., bleaching the sock).

Lemma 3 (Bleaching). *For any strategy $\sigma = \{c_i\}_{i=0}^{2\tau-1}$ where $c_1 \geq c_2 \geq \cdots$ and $c_1 > 0$ we have*

$$\Pr[W(\sigma)] \leq (1 + \frac{c_1}{2\tau}) \Pr[W(\sigma_{0\leftarrow 1})].$$

Proof. Note that the condition that $c_1 \geq c_2 \geq \cdots$ is without loss of generality as we can always rename colours so that this is true.

Assume first that $c_1 > c_2$. In that case we can prove something stronger, namely:

$$\Pr[W(\sigma)] \leq \left(1 + \frac{c_1 - 1}{2\tau - 1}\right) \Pr[W(\sigma_{0\leftarrow 1})] \ . \tag{2}$$

This is stronger as $\frac{a-1}{b-1} < \frac{a}{b}$ when $a < b$. We first prove this case. Let B be the bleached sock. I.e., the sock that changes colour going from σ to $\sigma_{0\leftarrow 1}$. Let M be the event that B matches up with a 1-sock. In $\sigma_{0\leftarrow 1}$ the event M gives rise a coin as B is in a $(0, 1)$-pair. In σ the event M does not give rise to a coin as B is in a $(1, 1)$-pair. All other pairs have identical colours in the two strategies, and thus we have $\Pr[W(\sigma) \mid M] = 2\Pr[W(\sigma_{0\leftarrow 1}) \mid M]$.

Now let $\bar{M}$ denote the complement of M, i.e., the event that B does not match up with a 1-sock. If $\bar{M}$ occurs, the bleaching of B will have no effect, B's matching gives rise to a coin in both strategies, so we have $\Pr[W(\sigma) \mid \bar{M}] = \Pr[W(\sigma_{0\leftarrow 1}) \mid \bar{M}]$. Combining these observations we get that

$$\begin{aligned}
\Pr[W(\sigma)] &= \Pr[M]\Pr[W(\sigma) \mid M] + \Pr[\bar{M}]\Pr[W(\sigma) \mid \bar{M}] \\
&= \Pr[M] 2\Pr[W(\sigma_{0\leftarrow 1}) \mid M] + \Pr[\bar{M}]\Pr[W(\sigma_{0\leftarrow 1}) \mid \bar{M}] \\
&= \Pr[M]\Pr[W(\sigma_{0\leftarrow 1}) \mid M] + \Pr[\bar{M}]\Pr[W(\sigma_{0\leftarrow 1}) \mid \bar{M}] + \Pr[M]\Pr[W(\sigma_{0\leftarrow 1}) \mid M] \\
&= \Pr[W(\sigma_{0\leftarrow 1})] + \Pr[W(\sigma_{0\leftarrow 1}) \wedge M] \ .
\end{aligned}$$

It follows that

$$\begin{aligned}
\Pr[W(\sigma)]/\Pr[W(\sigma_{0\leftarrow 1})] &= 1 + \frac{\Pr[W(\sigma_{0\leftarrow 1}) \wedge M]}{\Pr[W(\sigma_{0\leftarrow 1})]} \\
&= 1 + \Pr[M \mid W(\sigma_{0\leftarrow 1})] \ ,
\end{aligned}$$

so

$$\Pr[W(\sigma)] = (1 + \Pr[M \mid W(\sigma_{0\leftarrow 1})])\Pr[W(\sigma_{0\leftarrow 1})] \ .$$

To prove (2) it is therefore sufficient to prove that $\Pr[M \mid W(\sigma_{0\leftarrow 1})] \leq \Pr[M]$, as $\Pr[M] \leq \frac{c_1 - 1}{2\tau - 1}$. By Bayes' law we have that $\Pr[M \mid W(\sigma_{0\leftarrow 1})] \leq \Pr[M]$ if and only if $\Pr[W(\sigma_{0\leftarrow 1}) \mid M] \leq \Pr[W(\sigma_{0\leftarrow 1})]$, which holds if $\Pr[W(\sigma_{0\leftarrow 1}) \mid M] \leq \Pr[W(\sigma_{0\leftarrow 1}) | \bar{M}]$. For all colours i let C_i denote the event that B pairs with a sock of colour i. Note that $C_1 = M$. It is therefore clearly enough to prove that

$$\Pr[W(\sigma_{0\leftarrow 1}) \mid C_1] \leq \Pr[W(\sigma_{0\leftarrow 1}) | C_i]$$

for all i. So we have to prove for $\sigma_{0\leftarrow 1}$ that if B is paired with a 1-sock, then this is bad for the winning probability compared to pairing with socks of any other colour. This makes sense as B is white in $\sigma_{0\leftarrow 1}$, so when it is paired with a 1-sock, it is stealing away a 1-sock which could otherwise have been paired with another 1-sock: recall that 1 is the most common non-white colour. We now give a more detailed proof.

Note that given C_i we can consider the uniformly random pairing of the socks as follows: First pair B with a random sock of colour i and then pair the $2\tau - 2$ remaining socks uniformly at random. Thus we can consider $\Pr[W(\sigma_{0\leftarrow 1}) \mid C_i]$ in terms of the reduced game with only $2\tau - 2$ socks. We denote by ω_i the strategy induced by $\sigma_{0\leftarrow 1}$ on the reduced game given C_i. I.e, ω_i is a strategy for the game with $2\tau - 2$ socks that first colours 2τ socks according to $\sigma_{0\leftarrow 1}$ and then removes B and a random i-sock and proceeds the game with the remaining socks. We then have $\Pr[W(\sigma_{0\leftarrow 1}) \mid C_i] = \frac{1}{2}\Pr[W(\omega_i)]$ (as we recall that the pairing of B we will introduce an extra coin in the full game compared to the reduced game).

So now what we want to prove is that for all i

$$\Pr[W(\omega_1)] \leq \Pr[W(\omega_i)]$$

To see this note that for all i we have $\omega_1 = (\omega_i)_{i\leftarrow 1}$. I.e., ω_1 and ω_i are identical except for one sock D that has colour i in ω_1 and 1 in ω_i. We then argue that it is always better to have D be of colour 1 rather than i. If i is 0, this is trivial. No matter how D pairs up, it is never worse for it to be 1 than 0, and sometimes better. If $i > 1$, then recall that we started from the condition on σ that $c_1 > c_i$. Therefore, for $\sigma_{0\leftarrow 1} = \{c'_i\}_{i=0}^{2\tau-1}$ we have $c'_1 \geq c'_i$. If $c'_1 = c'_i$, trivially it does not matter whether D is has colour 1 or i. If $c'_1 > c'_2$, it is strictly better to have D be a 1-sock. To see this consider the following way of pairing the socks: first we pair up D. Then we uniformly at random pair the remaining socks. The colours in the remaining pairs will be independent of the colour of the D. The only probability that changes is that the first pair is more likely to be a match when D is of colour 1. This proves (2).

Having proved the lemma for the case of $c_1 > c_2$, now assume the case that $c_1 = c_2$. Take any such strategy $\sigma = \{c_i\}_{i=0}^{2\tau-1}$ by C and consider the strategy $\sigma_{1\leftarrow 2} = \{c'_i\}_{i=0}^{2\tau-1}$. This might not be a legal strategy as we may have $c'_1 > c'_0$. But if we relax this condition on the game we can still evaluate the winning probability of the strategy. We then have that

$$\Pr[W(\sigma)] \leq \Pr[W(\sigma_{1\leftarrow 2})] \ , \tag{3}$$

as the recoloured sock is now part of a larger colour than before, which is obviously an advantage, as argued above.

Notice that the size of the largest non-0 colour in $\sigma_{1\leftarrow 2}$ is $c'_1 = c_1 + 1$. By applying first (3) and then (2) we get that

$$\Pr[W(\sigma)] \leq \Pr[W(\sigma_{1\leftarrow 2})] \leq \left(1 + \frac{(c_1+1)-1}{2\tau}\right)\Pr[W(\sigma_{1\leftarrow 2, 0\leftarrow 1})] \ .$$

Since $\sigma_{1\leftarrow 2,0\leftarrow 1} = \sigma_{0\leftarrow 2}$ and, since $\Pr[W(\sigma_{0\leftarrow 2})] = \Pr[W(\sigma_{0\leftarrow 1})]$ by the assumption $c_1 = c_2$, this proves the Bleaching Lemma.

Consider now a strategy σ as above where $c_1 > 2$. By applying the Bleaching Lemma twice we have that

$$\Pr[W(\sigma)] \leq \left(1 + \frac{c_1 - 1}{2\tau}\right)\left(1 + \frac{c_1}{2\tau}\right)\Pr[W(\sigma_{0\leftarrow 1, 0\leftarrow 1})] \ ,$$

by bleaching one sock at a time. [12] Similarly let σ_{c_1} be the strategy starting from σ, where we bleach $c_1 - 1$ of the 1-socks into 0-socks, leaving only a single 1-sock. But applying the Bleaching Lemma $c_1 - 1$ times we get that

$$\Pr[W(\sigma)] \leq \prod_{j=2}^{c_i}\left(1 + \frac{j}{2\tau}\right)\Pr[W(\sigma_{c_1})] \ .$$

[12] Note that the second time we apply the Bleaching Lemma we might no longer have the condition $c_1 \geq c_2 \geq \cdots$. We describe why this is not a problem below.

After this transformation we can rename the colours such that again it holds that $c_1 \geq c_2 \geq \cdots$.[13] If still $c_1 > 1$ we can apply the above iterative application of the Bleaching Lemma to bleach all but one of the new 1-socks. Doing this iteratively we will be left with a strategy σ_0, where $c_i < 2$ for all $i > 0$.

By the above argument we have that

$$\Pr[W(\sigma)] \leq \prod_{i=1}^{m} \prod_{j=2}^{c_i} \left(1 + \frac{j}{2\tau}\right) \Pr[W(\sigma_0)] \ .$$

We clearly also have that

$$\Pr[W(\sigma_0)] = 2^{-\tau}$$

as there will be τ coins on the table when using σ_0.

It follows from log being concave and an application of Jensen's inequality that $\prod_{i=1}^{N} a_i \leq (\frac{1}{N}\sum_{i=1}^{N} a_i)^N$ when all $a_i \geq 0$. From this we have that

$$\begin{aligned}\prod_{j=2}^{c_i} \left(1 + \frac{j}{2\tau}\right) &\leq \prod_{j=0}^{c_i} \left(1 + \frac{j}{2\tau}\right) \\ &\leq \left(1 + \frac{c_i}{4\tau}\right)^{c_i+1} \\ &= \left(1 + \frac{c_i}{4\tau}\right)\left(1 + \frac{c_i}{4\tau}\right)^{c_i} \ .\end{aligned}$$

All in all we have that

$$\Pr[W(\sigma)] \leq 2^{-\tau} \prod_{i=1}^{m} \left(1 + \frac{c_i}{4\tau}\right) \prod_{i=1}^{m} \left(1 + \frac{c_i}{4\tau}\right)^{c_i} \ .$$

We have that $1 + x \leq e^x$ for all real numbers x so if $\sum_i x_i \leq x$, then $\prod_i (1 + x_i) \leq e^x$. From this we get that

$$\Pr[W(\sigma)] \leq e^{1/2} 2^{-\tau} \prod_{i=1}^{m} \left(1 + \frac{c_i}{4\tau}\right)^{c_i} \ .$$

We move around and get that

$$\Pr[W(\sigma)] \leq e^{1/2} 2^{-\tau} \left(\prod_{i=1}^{m} \left(1 + \frac{c_i}{4\tau}\right)^{\frac{c_i}{4\tau}}\right)^{4\tau} \ .$$

We now use that $c_i \leq \tau$ and that when $x \in [0, \frac{1}{4}]$, then $(1 + x)^x \leq 1 + x^2$. We get that

$$\Pr[W(\sigma)] \leq e^{1/2} 2^{-\tau} \left(\prod_{i=1}^{m} \left(1 + \frac{c_i^2}{16\tau^2}\right)\right)^{4\tau} \ .$$

The remaining proof proceeds in four cases. First we observe that if one uses two colours only then the winning probability is at most $e^{1/2} 2^{0.64\tau}$. Then we show that if one uses more than five colours, then the winning probability is also at most $e^{1/2} 2^{0.64\tau}$. Then we prove the significantly more complicated case of strategies using three colours, showing that also for these the winning probability is also at most $e^{1/2} 2^{0.64\tau}$. Finally we generalise the proof from three to four colours.

[13] Note that we might have to apply this transformation more often to always have the condition $c_1 \geq c_2 \geq \cdots$. It is easy to see that keeping track of the original colour names and tallies c_i in these transformations we get the exact same bounds as in the above more sloppy argument.

Cases analysis: We now for all values of c_0 find the optimal way to set the other colours $c_1, c_2, \dots$ and then find the winning probability of the best way to set these values. In all cases it follows that once c_0 is fixed it is optimal to set f as large as possible while allowing that $c_1 = c_2 = \cdots = c_{f-1} = c_0$ and $c_f > 0$ and $c_{f+1} = 0$. This follows from the recolouring lemma. Note that it might be that $c_f < c_0$ if c_0 does not divide 2τ. We break the cases up according to what value f has. If $f = 1$ we say that we use two colours. If $f = 2$ we use three colours *et cetera*.

Two colours: If there are only two colours, then clearly $c_0 = c_1 = \tau$ is optimal. In that case

$$\Pr[W(\sigma)] \le e^{1/2}2^{-\tau}\left(1+\frac{\tau^2}{16\tau^2}\right)^{4\tau} = e^{1/2}\left(2^{-1}\left(1+\frac{1}{16}\right)^4\right)^{\tau} \le e^{1/2}2^{-0.64\tau} \ .$$

More than Four Colours: In general, if there are $m+1$ colours, then it is clearly optimal to use $c_0 = \cdots = c_{m-1}$ and then possibly have $c_m < c_{m-1}$. In that case, clearly $c_i \le \frac{2\tau}{m}$ for all i, so

$$\begin{aligned}\Pr[W(\sigma)] &\le e^{1/2}2^{-\tau}\left(\prod_{i=1}^{m}\left(1+\frac{c_i^2}{16\tau^2}\right)\right)^{4\tau} &(4)\\ &= e^{1/2}2^{-\tau}\left(\prod_{i=1}^{m}\left(1+\frac{4\tau^2}{16\tau^2m^2}\right)\right)^{4\tau} &(5)\\ &= e^{1/2}2^{-\tau}\left(1+\frac{1}{4m^2}\right)^{4\tau m} &(6)\\ &= e^{1/2}2^{-\alpha(m)\tau} \ , &(7)\end{aligned}$$

where

$$\alpha(m) = 1 - \log_2\left(\left(1+\frac{1}{4m^2}\right)^{4m}\right) \ .$$

One can easily verify that $\alpha(m) \ge 0.64$ for $m \ge 4$.

Three Colours: If there are three colours, then it is still optimal to use $c_1 = c_0$ and it will then be the case that $c_1 \ge c_2$ and $0 = c_3 = c_4 = \cdots$. From $\sum_i c_i = 2\tau$ it then follows that $c_2 = 2\tau - 2c_1$. From the general bound we get that

$$\Pr[W(\sigma)] \le e^{1/2}2^{-\tau}\left(\left(1+\frac{c_1^2}{16\tau^2}\right)\left(1+\frac{c_2^2}{16\tau^2}\right)\right)^{4\tau} \ ,$$

from which we conclude that

$$\Pr[W(\sigma)] \le e^{1/2}2^{-\tau}\left(\left(1+\frac{c_1^2}{16\tau^2}\right)\left(1+\frac{(2\tau-2c_1)^2}{16\tau^2}\right)\right)^{4\tau} \ .$$

If $c_2 = 0$, then $c_1 = 2\tau/2$. If $c_2 = c_1$, then $c_1 = 2\tau/3$. Write $c_1 = \gamma\tau$ for $2/3 \le \gamma \le 1$. We get that

$$\begin{aligned}\Pr[W(\sigma)] &\le e^{1/2}2^{-\tau}\left(\left(1+\frac{\gamma^2\tau^2}{16\tau^2}\right)\left(1+\frac{(2\tau-2\gamma\tau)^2}{16\tau^2}\right)\right)^{4\tau}\\ &\le e^{1/2}2^{-\tau}\left(\left(1+\frac{\gamma^2}{16}\right)\left(1+\frac{(2-2\gamma)^2}{16}\right)\right)^{4\tau} \ .\end{aligned}$$

It can be seen analytically that $\left(1+\frac{\gamma^2}{16}\right)\left(1+\frac{(2-2\gamma)^2}{16}\right)$ has a unique maximum in $\gamma \in [2/3, 1]$ at $\gamma = 1$ corresponding to using two colours only. To see this, note that the expression is equal to $(16+x^2)(20-8x+4x^2)$ up to a constant factor. The derivative of this is $8(2x^3-3x^2+21x-16)$, which has one real root at ≈ 0.80479. To the right of the root it is positive and to the left of the root it is negative. Therefore $(16+x^2)(20-8x+4x^2)$ has two maxima in $[2/3, 1]$, namely in $2/3$ and 1. It is larger in $x = 1$ than in $x = 2/3$.

Four Colours: If there are 4 colours, then it follows as above that it is optimal with $c_0 = c_1 = c_2$, that $c_3 = 2\tau - 3c_1$ and that

$$\Pr[W(\sigma)] \le e^{1/2} 2^{-\tau} \left(\left(1 + \frac{c_1^2}{16\tau^2}\right)^2 \left(1 + \frac{c_3^2}{16\tau^2}\right)\right)^{4\tau} ,$$

from which we conclude that

$$\Pr[W(\sigma)] \le e^{1/2} 2^{-\tau} \left(\left(1 + \frac{c_1^2}{16\tau^2}\right)^2 \left(1 + \frac{(2\tau - 3c_1)^2}{16\tau^2}\right)\right)^{4\tau} .$$

If $c_3 = 0$, then $c_1 = 2\tau/3$. If $c_3 = c_1$, then $c_1 = 2\tau/4$. Write $c_1 = \gamma\tau$ for $2/4 \le \gamma \le 2/3$. Then

$$\begin{aligned}\Pr[W(\sigma)] &\le e^{1/2} 2^{-\tau} \left(\left(1 + \frac{\gamma^2\tau^2}{16\tau^2}\right)^2 \left(1 + \frac{(2\tau - 3\gamma\tau)^2}{16\tau^2}\right)\right)^{4\tau} \\ &\le e^{1/2} 2^{-\tau} \left(\left(1 + \frac{\gamma^2}{16}\right)^2 \left(1 + \frac{(2 - 3\gamma)^2}{16}\right)\right)^{4\tau} .\end{aligned}$$

It is easy to see that on the internal $\gamma \in [1/2, 2/3]$ the expression

$$\left(1 + \frac{\gamma^2}{16}\right)^2 \left(1 + \frac{(2 - 3\gamma)^2}{16}\right)$$

is maximal in $\gamma = 2/3$, which corresponds to using only three colours. Since we already know it is better to use two colours than using three colours, it follows that it is better to use two colours than using four colours. □

5.6 Proof of Theorem 6

In this section we give the full proof of Theorem 6. The cases where no party is corrupt and where P_1 is corrupt is straight forward, so we will focus on the case of corrupt P_2. The proof goes via two intermediary functionalities, each modelling a different aspect of the intuition given in Section 5.4. Between these intermediate functionality we show linear reducibility. We will then construct a specific leakage agent LA so that the LA-leaky $\mathcal{F}_{\Gamma\text{-ROT}}$ is linearly reducible to the intermediate functionalities. Finally the proof concludes by showing that this leakage agent LA is κ-secure. Before we get into the proof we will present a convenient way to model the protocol of Figure 18.

Modeling The Protocol: It will be helpful to model the protocol in Figure 18 in terms of socks colored by P_2. More specifically with each pair of messages N_0^i and N_1^i input to $\mathcal{F}_{\mathrm{OT}}$ by P_2 we associate a *sock* i. If $N_0^i \oplus N_1^i = \Gamma_i$, then we associate with Γ_i a *color* in $[2\tau]$, say j, and say that P_2 colors sock i with the color j. Since there is 2τ message pair (N_0^i, N_1^i) in the protocol there can be at most 2τ socks of 2τ different colors. We will denote by col the *coloring function* $\mathrm{col} : [2\tau] \to [2\tau]$ that maps each sock to its color. An honest P_2 picks a single Γ and for all $i \in [2\tau]$ picks messages so that $N_0^i \oplus N_1^i = \Gamma_i = \Gamma$, i.e. an honest P_2 will color all socks with the same color, so in this case col is a constant function. A corrupt P_2, on the other hand, can use any coloring function col.

Let $\mathrm{col}_0, \dots, \mathrm{col}_{2\tau-1}$ be the possible colors, then for a coloring function col we will let C_i be the set of socks of color col_i, i.e. $C_i = \{j \in [2\tau] | \mathrm{col}(j) = \mathrm{col}_i\}$. Without loss of generality we will let col_0 be *most common color* of col, i.e. $|C_0| \ge |C_i|$ for all $i \in [2\tau]$. We will sometimes refer to the value Γ associated with col_0 as the *right value* and all other $\Gamma' \ne \Gamma$ as being *wrong*.

The pairing π used in the protocol defines pairs of socks. That is, each sock $j \in [2\tau]$ is paired with $\pi(j)$. If we let $\mathcal{L} = \mathcal{L}_\pi = \{i | i \leq \pi(i)\}$ as in Figure 18 we can view each $i \in \mathcal{L}$ as representing the pair of socks $(i, \pi(i))$. Furthermore, we will define the set $\mathcal{M} = \mathcal{M}_{\pi,\mathrm{col}} \subseteq \mathcal{L}$ to be the set representing *mismatched* pairs, i.e. $\mathcal{M} = \{i \in \mathcal{L} | \operatorname{col}(i) \neq \operatorname{col}(\pi(i))\}$. In the protocol $i \in \mathcal{M}$ corresponds to the situation where for the two messages pairs (N_0^i, N_1^i) and $(N_0^{\pi(i)}, N_1^{\pi(i)})$ we have

$$N_0^i \oplus N_1^i \neq N_0^{\pi(i)} \oplus N_1^{\pi(i)} \ .$$

It will also be useful to define the set $\mathcal{N} = \mathcal{N}_{\pi,\mathrm{col}}$ to be the subset of *matched* pairs in $\mathcal{L} \setminus \mathcal{M}$ which are not of color col_0, i.e. $\mathcal{N} = \{i \in \mathcal{L} \setminus \mathcal{M} | \operatorname{col}(i) \neq \mathrm{col}_0\}$. In the protocol $i \in \mathcal{N}$ corresponds to the situation where for the two messages pairs (N_0^i, N_1^i) and $(N_0^{\pi(i)}, N_1^{\pi(i)})$ we have

$$N_0^i \oplus N_1^i = N_0^{\pi(i)} \oplus N_1^{\pi(i)} = \Gamma' \ ,$$

but Γ' is not the value associated with col_0. In other words $\mathcal{M}$ and $\mathcal{N}$ are the pairs where P_2 deviated from the protocol. For each $i \in \mathcal{N}$ P_2 will not get caught, while for $i \in \mathcal{M}$ P_2 will get caught with probability $\frac{1}{2}$ as we shall show below.

Intermediate Functionality 1: We now present our first intermediate functionality $\mathcal{F}_{\mathrm{IB1}}$. This functionality captures the idea that a corrupt P_2 can only get away with using a few different values of Γ. To see this let $i \in S_\pi$ and $j = \pi(i)$ and note that if P_2 chose the two message pairs (N_0^i, N_1^i) and (N_0^j, N_1^j) so that

$$\Gamma_i = N_0^i \oplus N_1^i \neq N_0^{\pi(i)} \oplus N_1^{\pi(i)} = \Gamma_j \ ,$$

i.e. $i \in \mathcal{M}$, then P_1 computes

$$\begin{aligned} D_i = N_{c_i}^i \oplus N_{c_j}^j &= (N_0^i \oplus c_i\Gamma_i) \oplus (N_0^j \oplus c_j\Gamma_j) \\ &= (N_0^i \oplus N_0^j) \oplus (c_i \oplus c_j)\Gamma_j \oplus c_i(\Gamma_i \oplus \Gamma_j) \\ &= (N_0^i \oplus N_0^j) \oplus d_i\Gamma_j \oplus c_i(\Gamma_i \oplus \Gamma_j). \end{aligned}$$

Since $(\Gamma_i \oplus \Gamma_j) \neq 0^\ell$ and $c_i \oplus c_j$ is fixed by announcing d_i, guessing this D_i is equivalent to guessing c_i. As P_2 only knows $N_0^i, N_0^j, \Gamma_i, \Gamma_j$ and d_i, all of which are independent of c_i, she can guess c_i with probability at most $\frac{1}{2}$. If P_2 cheats and uses many different values of Γ, then with high probability the pairing π will be such that there are many pairs where $\Gamma_i \neq \Gamma_{\pi(i)}$, and P_2 will get caught with high probability. However, if she uses only few values of Γ she might pass the test.

This corresponds to saying that for P_2 to get away with using coloring function col, she must guess c_i for all $i \in \mathcal{M}$, i.e. P_2 gets away with using col with probability at most $2^{-|\mathcal{M}|}$. Therefore, she is not likely to get away with using a coloring function that colors the socks many different colors, because such a coloring will result in $|\mathcal{M}|$ being large with high probability.

Lemma 4. *The protocol in Figure 18 implements $\mathcal{F}_{IB1}$ in the $(\mathcal{F}_{OT}(2\tau, \ell)$, $\mathcal{F}_{EQ}(\tau\ell))$-hybrid model, i.e. $\mathcal{F}_{IB1}$ is linear reducible to $(\mathcal{F}_{OT}(2\tau, \ell), \mathcal{F}_{EQ}(\tau\ell))$.*

Proof. By observing P_2's inputs to the OTs, the simulator learns all (N_0^i, N_1^i). Let $L_i = N_0^i$ and $\Gamma_i = N_0^i \oplus N_1^i$. Let f be the number of distinct values in $(\Gamma_i)_{i=1}^{2\tau}$ and pick distinct $\Lambda_1, \ldots, \Lambda_f$ and $\mathrm{col} : [2\tau] \to [2\tau]$ so that $\Gamma_i = \Lambda_{\mathrm{col}(i)}$. For $i = f+1, \ldots, 2\tau$ pick the remaining $\Lambda_i \in \{0,1\}^\ell$ in any arbitrary way (these will not be used anyway). By construction

$$N_1^i = L_i \oplus \Gamma_i = L_i \oplus \Lambda_{\mathrm{col}(i)}.$$

Input $(\mathtt{colors}, \mathrm{col}, (\Lambda_i, L_i)_{i \in [2\tau]})$ to $\mathcal{F}_{\mathrm{IB1}}$ on behalf of P_2 and receive $(\mathtt{pairs}, \pi)$. Send π to P_2 as if coming from P_1 along with uniformly random $\{d_i\}_{i \in \mathcal{L}}$.

The Functionality $\mathcal{F}_{\text{IB1}}$

Honest Parties

As in the leaky $\mathcal{F}_{\Gamma\text{-ROT}}$ functionality.

Corrupt Parties

1. If P_1 is corrupt: As in the leaky $\mathcal{F}_{\Gamma\text{-ROT}}$ functionality.
2. (a) If P_2 is corrupt, the functionality waits to give output till P_2 inputs $(\texttt{colors}, \text{col}, (\Lambda_i, L_i)_{i\in[2\tau]})$, where $L_i, \Lambda_i \in \{0,1\}^\ell$ and col is a coloring function.
   (b) Then the functionality samples a uniformly random pairing $\pi : [2\tau] \to [2\tau]$ and outputs $(\texttt{pairs}, \pi)$ to P_2. Let $\mathcal{L} = \mathcal{L}_\pi$ and let $\mathcal{M} = \mathcal{M}_{\pi,col}$.
   (c) The functionality then waits for P_2 to input $(\texttt{guess}, (g_i)_{i\in\mathcal{M}})$.
   (d) The functionality samples $(c_i)_{i\in[2\tau]} \in_{\text{R}} \{0,1\}^{2\tau}$. Then the functionality lets $c = 1$ if $g_i = c_i$ for all $i \in \mathcal{M}$, otherwise it lets $c = 0$. If $c = 0$ the functionality outputs $\texttt{fail}$ to P_1 and terminates. Otherwise, for $i \in \mathcal{L}$ it computes $N^i_{c_i} = L_i \oplus c_i\Lambda_{\text{col}(i)}$ and outputs $(N^i_{c_i}, c_i)_{i\in\mathcal{L}}$ to P_1.

Figure 19 The First Intermediate Functionality $\mathcal{F}_{\text{IB1}}$

Then observe the inputs $\hat{D}_i$ from P_2 to the $\mathcal{F}_{\text{EQ}}$ functionality. The simulator must now pick the guesses g_i for $i \in \mathcal{M}$. Note that $i \in \mathcal{M}$ implies that $\Lambda_{\text{col}(i)} \neq \Lambda_{\text{col}(\pi(i))}$, which means that $\Gamma_i \neq \Gamma_{\pi(i)}$. We use this to pick g_i, as follows: after seeing d_i, P_2 knows that $c_i = d_i \oplus c_{\pi(i)}$. Hence an honest P_1 would input to the comparison the following value for D_i depending on c_i

$$D_i(c_i) = (L_i \oplus L_{\pi(i)}) \oplus d_i\Lambda_{\text{col}(\pi(i))} \oplus c_i(\Lambda_{\text{col}(i)} \oplus \Lambda_{\text{col}(\pi(i))}) \ .$$

As $\Lambda_{\text{col}(i)} \neq \Lambda_{\text{col}(\pi(i))}$ we have $D_i(0) \neq D_i(1)$. Thus if P_2's input to the $\mathcal{F}_{\text{EQ}}$ functionality $\hat{D}_i$ is equal to $D_i(0)$ (resp. $D_i(1)$), the simulator sets $g_i = 0$ (resp. $g_i = 1$). If the simulator is able to set all $(g_i)_{i\in[\tau]}$ in this manner, i.e. all $\hat{D}_i$ equal either $D_i(0)$ or $D_i(1)$, it inputs $(\texttt{guess}, (g_i)_{i\in[\tau]})$ to the $\mathcal{F}_{\text{IB1}}$ functionality. Otherwise, the simulator outputs $\texttt{fail}$ and aborts.

Notice that in the real world protocol, if $g_i = c_i$, then $D_i = D_i(g_i) = \hat{D}_i$ and P_2 passes the test. If $g_i \neq c_i$, then $D_i = D_i(1 \oplus g_i) \neq \hat{D}_i$ and P_2 fails the test. So, the protocol and the simulation fails on the same event. Note then that when the functionality does not fail, then it outputs

$$L_i \oplus c_i\Lambda_{\text{col}(i)} = N^i_0 \oplus c_i\Gamma_i = N^i_0 \oplus c_i(N^i_0 \oplus N^i_1) = N^i_{c_i},$$

exactly as the protocol. Hence the simulation is perfect. ◻

Intermediate functionality 2: The second intermediate functionality $\mathcal{F}_{\text{IB2}}$ in Figure 20 captures the idea that an adversary that gets away with using multiple values of Γ is equivalent to one that only uses a single value of Γ but instead learns some of P_1's choice bits c_i.

To see this first recall that we above showed that for P_2 to get away with using multiple values of Γ she must guess all c_i for $i \in \mathcal{M}$. Doing so confirms her guess on c_i, so if she passes the test she learns these c_i. Now we can let Γ be the value associated with the most common color col_0. Assume then that P_2 cheated and for some i (not necessarily in $\mathcal{M}$) used a message pair (N^i_0, N^i_1) where $N^i_0 \oplus N^i_1 = \Gamma' \neq \Gamma$. We can explain this as an honest run: If $c_i = 0$, the run is equivalent to P_2 having inputted $(N^i_0, N^i_0 \oplus \Gamma)$, as P_1 gets no information on the second message when $c_i = 0$ (by privacy of $\mathcal{F}_{\text{OT}}$). If $c_i = 1$, then the run is equivalent to having input message $(N^i_1 \oplus \Gamma, N^i_1)$ as P_1 gets no information on the first message when $c_i = 1$. So, any cheating strategy of P_2 can be simulated by letting her honestly use the same Γ in all message pairs and then let her try to guess the bits c_i for $i \in \mathcal{M}$. If her guess is incorrect, the deviation is reported to P_1. If her guess is correct, she is told so and the deviation is not reported to P_1.

Note that in order to make the simulation work using this idea the simulator must know c_i for all $i \in \mathcal{L}$ where $\text{col}(i) \neq \text{col}_0$ even if $i \notin \mathcal{M}$, i.e. it must know c_i for all $i \in \mathcal{M} \cup \mathcal{N}$. Thus to make the simulation work functionality $\mathcal{F}_{\text{IB2}}$ will leak the bits c_i $i \in \mathcal{N}$ "for free".

Lemma 5. *$\mathcal{F}_{IB2}$ is linear locally reducible to $\mathcal{F}_{IB1}$.*

The Functionality $\mathcal{F}_{\mathrm{IB2}}$

Honest Parties

As in the leaky $\mathcal{F}_{\Gamma\text{-ROT}}$ functionality.

Corrupt Parties

1. If P_1 is corrupt: As in the leaky $\mathcal{F}_{\Gamma\text{-ROT}}$ functionality.
2. If P_2 is corrupt:
   (a) The functionality waits to give output till P_2 inputs $(\texttt{colors}, \mathrm{col})$ where col is a coloring function.
   (b) The functionality samples a uniformly random pairing $\pi : [2\tau] \to [2\tau]$ and outputs $(\texttt{pairs}, \pi)$ to P_2. Let $\mathcal{L} = \mathcal{L}_\pi$, $\mathcal{M} = \mathcal{M}_{\pi,\mathrm{col}}$ and $\mathcal{N} = \mathcal{N}_{\pi,\mathrm{col}}$.
   (c) The functionality then waits for P_2 to input $(\texttt{guess}, (g_i)_{i\in\mathcal{M}})$.
   (d) The functionality samples $(c_i)_{i\in[2\tau]} \in_{\mathrm{R}} \{0,1\}^{2\tau}$. Then the functionality lets $c = 1$ if $g_i = c_i$ for all $i \in \mathcal{M}$, otherwise it lets $c = 0$. If $c = 0$ the functionality outputs $\texttt{fail}$ to P_1 and terminates. Otherwise, the functionality determines col_0 and for $i \in \mathcal{N}$, the functionality outputs (i, c_i) to P_2.
   (e) The functionality waits for P_2 to input $(\Gamma, (L_i)_{i\in[2\tau]})$ where $\Gamma, L_i \in \{0,1\}^\ell$.
   (f) For $i \in \mathcal{L}$ the functionality computes $N^i_{c_i} = L_i \oplus c_i\Gamma$ and outputs $(N^i_{c_i}, c_i)_{i\in\mathcal{L}}$ to P_1.

Figure 20 The Second Intermediate Functionality $\mathcal{F}_{\mathrm{IB2}}$

Proof. To implement $\mathcal{F}_{\mathrm{IB2}}$ simply calls $\mathcal{F}_{\mathrm{IB1}}$. Note that the simulator must simulate $\mathcal{F}_{\mathrm{IB2}}$ to the environment, and fully controls the $\mathcal{F}_{\mathrm{IB1}}$ towards the corrupt P_2. First the simulator observes the input values $(\texttt{colors}, \mathrm{col}, (\Lambda_i, L_i)_{i\in 2\tau})$ of P_2 to $\mathcal{F}_{\mathrm{IB1}}$ and inputs $(\texttt{colors}, \mathrm{col})$ to $\mathcal{F}_{\mathrm{IB2}}$. $\mathcal{F}_{\mathrm{IB2}}$ outputs $(\texttt{pairs}, \pi)$ and the simulator inputs $(\texttt{pairs}, \pi)$ to P_2 on behalf of $\mathcal{F}_{\mathrm{IB1}}$, and computes $\mathcal{M}$ as $\mathcal{F}_{\mathrm{IB1}}$ and $\mathcal{F}_{\mathrm{IB2}}$ would have done. Then the simulator observes the input $(\texttt{guess}, (g_i)_{i\in\mathcal{M}})$ from P_2 to $\mathcal{F}_{\mathrm{IB1}}$ and inputs $(\texttt{guess}, (g_i)_{i\in\mathcal{M}})$ to $\mathcal{F}_{\mathrm{IB2}}$. If $\mathcal{F}_{\mathrm{IB2}}$ outputs $\texttt{fail}$ to P_1 the simulation is over, and it is perfect as $\mathcal{F}_{\mathrm{IB1}}$ and $\mathcal{F}_{\mathrm{IB2}}$ fail based on the same event. If $\mathcal{F}_{\mathrm{IB2}}$ does not fail it determines col_0 and for $i \in \mathcal{M}$, if $\mathrm{col}(i) \neq \mathrm{col}_0$, the functionality outputs (i, c_i) to the simulator. Note that the simulator can also determine col_0 from col.

Now let $\Gamma = \Lambda_{\mathrm{col}_0}$ and for $i \in \mathcal{M} \cup \mathcal{N}$, if $\mathrm{col}(i) = \mathrm{col}_0$, let $L'_i = L_i$. Then for $i \in \mathcal{M} \cup \mathcal{N}$, if $\mathrm{col}(i) \neq \mathrm{col}_0$, let $L'_i = (L_i \oplus c_i\Lambda_{\mathrm{col}(i)}) \oplus c_i\Gamma$. Finally input $(\Gamma, (L'_i)_{i\in[2\tau]})$ to $\mathcal{F}_{\mathrm{IB2}}$. As a result $\mathcal{F}_{\mathrm{IB2}}$ will for $i \in \mathcal{L}$ where $\mathrm{col}(i) = \mathrm{col}_0$, output $L'_i \oplus c_i\Gamma = L_i \oplus c_i\Lambda_{\mathrm{col}_0}$, and for $i \in \mathcal{L}$ where $\mathrm{col}(i) \neq \mathrm{col}_0$ it will output $L'_i \oplus c_i\Gamma = L_i \oplus c_i\Lambda_{\mathrm{col}(i)}$. Hence $\mathcal{F}_{\mathrm{IB2}}$ gives exactly the outputs that $\mathcal{F}_{\mathrm{IB1}}$ would have given after interacting with P_2, giving a perfect simulation. ◻

The Leakage Agent: Finally we now specify a leakage agent LA so that the LA-leaky $\mathcal{F}_{\Gamma\text{-ROT}}$ functionality is equivalent to $\mathcal{F}_{\mathrm{IB2}}$. The interaction between LA and the adversary is as follows.

1. LA waits for the adversary to input $(\texttt{colors}, \mathrm{col})$ where col is a function $[2\tau] \to [2\tau]$.
2. Then LA outputs $(\texttt{pairs}, \pi)$ to the adversary, where $\pi : [2\tau] \to [2\tau]$ is a uniformly random pairing as defined above.
3. LA waits for the adversary to input $(\texttt{guess}, (g_i)_{i\in\mathcal{M}_{\pi,\mathrm{col}}})$.

After this interaction and on input $(c_i)_{i\in[\tau]} \in \{0,1\}^\tau$ the leakage agent LA does the following. Here we let $\Pi : \mathcal{L} \to [\tau]$ be an order preserving function (simply to map $\mathcal{L}$ onto $[\tau]$).

1. LA computes $\mathcal{M} = \mathcal{M}_{\pi,\mathrm{col}}$ and $\mathcal{N} = \mathcal{N}_{\pi,\mathrm{col}}$. If $\bigwedge_{i\in\mathcal{M}}(g_{\Pi(i)} = c_i)$ sets $c = 1$, otherwise it sets $c = 0$.
2. LA computes $S = \{j = \Pi(i) | i \in \mathcal{M} \cup \mathcal{N}\}$ and outputs (c, S).

Notice that when we define LA in this way $\mathcal{F}_{\mathrm{IB2}}$ and LA-leaky $\mathcal{F}_{\Gamma\text{-ROT}}$ is exactly the same. Therefore we trivially get the following.

Lemma 6. *For* LA *as defined above the* LA*-leaky* $\mathcal{F}_{\Gamma\text{-ROT}}(\tau, \ell)$ *functionality is linear locally equivalent to the* $\mathcal{F}_{\mathrm{IB2}}$ *functionality.*

And this leads us to the following theorem.

Theorem 8. *For* LA *as defined above* LA*-leaky* $\mathcal{F}_{\Gamma\text{-ROT}}(\tau, \ell)$ *is linear reducible to* $(\mathcal{F}_{\mathrm{OT}}(2\tau, \ell), \mathcal{F}_{\mathrm{EQ}}(\tau\ell))$.

Proof. This follows from Lemma 4, 5 and 6 and by the transitivity of linear local reducibility. ☐

What remains then to complete the proof of Theorem 6 is to prove that LA defined in this way is $\kappa = \frac{6}{10}\tau$-secure.

Lemma 7. *Let* P_2 *be an adversary playing in* $\mathrm{LeakageGame}(\mathsf{LA}, P_2, \tau)$*. Assume that during the interaction between* P_2 *and* LA*,* P_2 *inputs* $(\mathtt{colors}, \mathrm{col})$ *and* LA *outputs* $(\mathtt{pairs}, \pi)$*. Let* $\mathcal{L} = \mathcal{L}_\pi$*,* $\mathcal{M} = \mathcal{M}_{\mathrm{col},\pi}$ *and* $\mathcal{N} = \mathcal{N}_{\mathrm{col},\pi}$*. Then the success probability of* P_2 *is at most* $2^{-0.6\tau}$ *for all large enough* τ*.*

Proof. Let $\mathcal{L}' = \Pi(\mathcal{L})$, $\mathcal{M}' = \Pi(\mathcal{M})$ and $\mathcal{N}' = \Pi(\mathcal{N})$, i.e. $\mathcal{L}', \mathcal{M}'$ and $\mathcal{N}'$ are essentially the same sets as $\mathcal{L}, \mathcal{M}$ and $\mathcal{N}$ only mapped to interval $[\tau]$ instead of $[2\tau]$. In order for P_2 to win the game LA must set $c = 1$. For this to happen P_2 must guess the uniformly random bits $(c_i)_{i \in \mathcal{M}'}$. This happens with probability $2^{-|\mathcal{M}|}$. Furthermore, to win P_2 must also guess the bits $(c_i)_{i \in \mathcal{L}' \setminus (\mathcal{M}' \cup \mathcal{N}')}$. This happens with probability $2^{-|\mathcal{L} \setminus (\mathcal{M} \cup \mathcal{N})|}$. Since the sets $\mathcal{M}$ and $\mathcal{N}$ are disjoint, this gives a total success probability of $2^{-|\mathcal{L} \setminus \mathcal{N}|}$. Now notice that this is exactly the same probability as winning the sock game as $|\mathcal{L} \setminus \mathcal{N}|$ is exactly the number of matched white pairs and mismatched colored pairs (since $\mathcal{N}$ is the matched colored pairs). So by Theorem 7 we have that the success probability is upper bounded by $e^{1/2} 2^{-0.64\tau}$. Then use that $e^{1/2} 2^{-0.64\tau} \leq 2^{-0.64\tau + 0.73}$ and clearly $0.64\tau - 0.73 \geq 0.6\tau$ for large enough τ. ☐

Notice that $0.64\tau - 0.73 \geq 0.6\tau$ already for $\tau \geq 19$ so not much is lost to the asymptotic notion of security. We get 0.6τ-bit security already for $\tau = 19$.

5.7 $\mathcal{F}_{\Delta\text{-ROT}}$ From Leaky $\mathcal{F}_{\Delta\text{-ROT}}$

Finally we will give a functionality $\mathcal{F}_{\Delta\text{-ROT}}$ which does not leak any bits of the global key, i.e. the LA-leaky $\mathcal{F}_{\Delta\text{-ROT}}$ functionality where LA is the leakage agent that leaks nothing. This functionality will be useful in both Section 6 and Section 7, when we present the preprocessing needed for our MPC protocols. We present the non-leaky $\mathcal{F}_{\Delta\text{-ROT}}$ functionality in Figure 21.

The Functionality $\mathcal{F}_{\Delta\text{-ROT}}(\ell, \tau)$

Honest Parties

On input `start` from both P_1 and P_2 the functionality does the following.

1. The functionality samples $\Delta \in_{\mathrm{R}} \{0,1\}^\tau$ and outputs it to P_1.
2. For all $i \in [\ell]$ the functionality samples $b_i \in_{\mathrm{R}} \{0,1\}$ and $M_0^i \in_{\mathrm{R}} \{0,1\}^\tau$.
3. The functionality outputs $(M_0^i \oplus b_i\Delta, b_i)_{i \in [\ell]}$ to P_2 and $(M_0^i)_{i \in [\ell]}$ to P_1.

Corrupt Parties

1. If P_2 is corrupt, the functionality waits to give output till it receives the message $(\hat{M}^i_{\hat{b}_i}, \hat{b}_i)_{i \in [\ell]}$ from P_2, where $\hat{M}^i_{\hat{b}_i} \in \{0,1\}^\tau$ and $\hat{b}_i \in \{0,1\}$. The functionality then sets $b_i = \hat{b}_i$ and $M_0^i = \hat{M}^i_{\hat{b}_i} \oplus b_i\Delta$ and outputs as described above.
2. If P_1 is corrupt, the functionality waits to give output till it receives the message $(\hat{\Delta}, (\hat{M}_0^i)_{i \in [\ell]})$ from P_1, where $\hat{\Delta}, \hat{M}_0^i \in \{0,1\}^\tau$. The functionality then sets $\Delta = \hat{\Delta}$ and $M_0^i = \hat{M}_0^i$ and outputs as described above.

Figure 21 The $\mathcal{F}_{\Delta\text{-ROT}}(\ell, \tau)$ Functionality

In Figure 22 we describe a protocol which takes a leaky $\mathcal{F}_{\Delta\text{-ROT}}$ functionality, where 40% of the bits of the global key might leak, and amplifies it to the non-leaky $\mathcal{F}_{\Delta\text{-ROT}}$ functionality. We prove the following theorem.

Theorem 9. *Let* $\tau = \frac{55}{6}\psi$ *and* LA *be a* $\left(\frac{4}{10}\tau\right)$*-secure leakage agent on* τ *bits. The protocol in Figure 22 securely implements* $\mathcal{F}_{\Delta\text{-ROT}}(\ell, \psi)$ *in the* LA*-leaky* $\mathcal{F}_{\Delta\text{-ROT}}(\ell, \tau)$*-hybrid model with security parameter* ψ*. The communication is* $O(\psi^2)$ *and the work is* $O(\psi^2\ell)$*.*

Protocol $\Pi_{\Delta\text{-ROT}}$

1. The parties invoke a LA-leaky $\mathcal{F}_{\Delta\text{-ROT}}(\ell, \tau)$ with $\tau = \frac{55}{6}\psi$. The output to P_1 is $(\hat{M}^i_{b_i}, b_i)_{i\in[\ell]}$. The output to P_2 is $(\hat{\Delta}, (\hat{M}^i_0)_{i\in[\ell]})$.
2. P_2 samples $\mathbf{A} \in_{\mathrm{R}} \{0,1\}^{\psi\times\tau}$, a random binary matrix with ψ rows and τ columns, and sends $\mathbf{A}$ to P_1.
3. P_1 computes $M^i_{b_i} = \mathbf{A}\hat{M}^i_{b_i} \in \{0,1\}^{\psi}$ and outputs $(M^i_{b_i}, b_i)_{i\in[\ell]}$.
4. P_2 computes $\Delta = \mathbf{A}\hat{\Delta}$ and $M^i_0 = \mathbf{A}\hat{M}^i_0$ and outputs $(\Delta, (M^i_0)_{i\in[\ell]})$.

Figure 22 Protocol for Reducing $\mathcal{F}_{\Delta\text{-ROT}}(\ell, \psi)$ to LA-leaky $\mathcal{F}_{\Delta\text{-ROT}}(\ell, \tau)$.

Correctness of the protocol is straight forward: We have that $\hat{M}^i_{b_i} = \hat{M}^i_0 \oplus b_i\hat{\Delta}$, so

$$M^i_{b_i} = \mathbf{A}\hat{M}^i_{b_i} = \mathbf{A}M^i_0 \oplus b_i\mathbf{A}\hat{\Delta} = M^i_0 \oplus b_i\Delta.$$

In addition it is clear that the protocol leaks no information on the b_i's to P_2: there is only communication from P_2 to P_1. It is therefore sufficient to look at the case where P_1 is corrupt. To prove security against corrupt P_1 we will prove that Δ is uniformly random in the view of P_1 except with probability $2^{2-\psi}$.

When we say that Δ *is uniform to* P_1 we mean that Δ is uniformly random in $\{0,1\}^{\psi}$ and independent of the view of P_1. When we say *except with probability* $2^{2-\psi}$ we mean that there exists a failure event F for which it holds that:

1. F occurs with probability at most $2^{2-\psi}$.
2. When F does not occur, then Δ is uniform to P_1.

To prove this we consider the following experiment LeakExp. Assuming that LA is a leakage agent on τ bits which is κ-secure LeakExp corresponds to the leakage on $\hat{\Delta}$ a corrupt P_1 receives during the protocol.

```
LeakExp(LA, $P_1$)
$\hat{\Delta} \in_{\mathrm{R}} \{0,1\}^{\tau}$
$P_1(\tau) \leftrightarrow \mathsf{LA}(\tau)$
$(S, c) \leftarrow \mathsf{LA}(\hat{\Delta})$
$\mathbf{A} \in_{\mathrm{R}} \{0,1\}^{\psi\times\tau}$
Input $\mathbf{A}$ to $P_1$
Output $\Delta = \mathbf{A}\hat{\Delta}$
```

To show that Δ is uniform to P_1, we give the following three technical lemmas on on LeakExp. For a subset $S \subset [\tau]$ of the column indices, let $\mathbf{A}^S$ be the matrix where column j is equal to the j'th column of $\mathbf{A}$ if $j \in S$, and column j is the 0 vector if $j \notin S$. We say that we *blind out* column j with 0's if $j \notin S$. Similarly, for a column vector v we use the notation v^S to mean that we set all indices v_i where $i \notin S$ to be 0. Note that $\mathbf{A}v^S = \mathbf{A}^S v = \mathbf{A}^S v^S$.

Lemma 8. *Let S and $\mathbf{A}$ be the values sampled in* LeakExp(LA, P_1)*. If $\mathbf{A}^{\bar{S}}$ spans $\{0,1\}^{\psi}$, then Δ is uniform to P_1.*

Proof. We start by making two simple observations. First of all, if P_1 learns $\hat{\Delta}_j$ for $j \in S$, then it learns $\hat{\Delta}^S$,[14] so it knows $\mathbf{A}\hat{\Delta}^S = \mathbf{A}^S\hat{\Delta}$. The second observation is that $\mathbf{A}\hat{\Delta} = \mathbf{A}^S\hat{\Delta} + \mathbf{A}^{\bar{S}}\hat{\Delta}$, as $\mathbf{A} = \mathbf{A}^S + \mathbf{A}^{\bar{S}}$. The lemma follows directly from these observations and the premise: We have that $\mathbf{A}^{\bar{S}}\hat{\Delta}$ is uniformly random in $\{0,1\}^{\psi}$ when the columns of $\mathbf{A}^{\bar{S}}$ span $\{0,1\}^{\psi}$. Since $\mathbf{A}^{\bar{S}}\hat{\Delta} = \mathbf{A}\hat{\Delta}^{\bar{S}}$ and $\hat{\Delta}^{\bar{S}}$ is uniformly random and independent of the view of P_1 it follows that $\mathbf{A}^{\bar{S}}\hat{\Delta}$ is uniformly random and independent of the view of P_1. Since only $\mathbf{A}^S\hat{\Delta}$ is known by P_1 it follows that $\mathbf{A}^S\hat{\Delta} + \mathbf{A}^{\bar{S}}\hat{\Delta}$ is uniform to P_1. The proof concludes by using that $\Delta = \mathbf{A}^S\hat{\Delta} + \mathbf{A}^{\bar{S}}\hat{\Delta}$. □

[14] Here we are looking at the string $\hat{\Delta}$ as a column vector of bits.

Lemma 9. *Let $n = \frac{9}{2}\psi$, $\alpha = \frac{\tau}{n} = \frac{55}{27}$ and $\kappa = \frac{6}{10}\tau$ such that and LA is κ-secure. Let W be the event that $|S| \geq \tau - n$ and $c = 1$ (where S and c are sampled as in* $\mathrm{LeakExp}(\mathsf{LA}, P_1)$*). Then* $\Pr(W) \leq 2^{-\psi}$.

Proof. Without loss of generality we can assume that P_1 interacts optimally with LA, i.e., $\log_2(\mathrm{E}\left[c2^{|S|}\right]) = \mathrm{leak}_{\mathsf{LA}}$. Since LA is κ secure on τ bits, it follows that $\mathrm{leak}_{\mathsf{LA}} \leq \tau - \kappa = \frac{4}{10}\tau$. This gives that

$$\mathrm{E}\left[c2^{|S|}\right] \leq 2^{\frac{4}{10}\tau} , \tag{8}$$

which we use later. Now let $\bar{W}$ be the event that W does not happen. By the properties of conditional expected value we have that

$$\mathrm{E}\left[c2^{|S|}\right] = \Pr(W) \cdot \mathrm{E}\left[c2^{|S|}|W\right] + \Pr(\bar{W}) \cdot \mathrm{E}\left[c2^{|S|}|\bar{W}\right] .$$

When W happens, then $|S| \geq \tau - n = \alpha n - n = (\alpha - 1)n$ and $c = 1$, so $c2^{|S|} = 2^{|S|} \geq 2^{(\alpha-1)n}$. This gives that

$$\mathrm{E}\left[c2^{|S|}|W\right] \geq 2^{(\alpha-1)n} .$$

Hence

$$\mathrm{E}\left[c2^{|S|}\right] \geq \Pr(W)\, 2^{(\alpha-1)n} .$$

Combining with (8) we get that

$$\Pr(W) \leq 2^{\frac{4}{10}\tau-(\alpha-1)n} .$$

It is, therefore, sufficient to show that $\frac{4}{10}\tau - (\alpha - 1)n = -\psi$, which can be verified to be the case by definition of τ, α, n and ψ, as follows,

$$\frac{4}{10}\tau - (\alpha - 1)n = \frac{4}{10}\tau - \tau + n = n - \frac{6}{10}\tau = \frac{9}{2}\psi - \frac{6}{10}\frac{55}{6}\psi = -\psi .$$

$\square$

Lemma 10. *As above let $n = \frac{9}{2}\psi$. Furthermore, let $x_1, \ldots, x_n \in_{\mathrm{R}} \{0,1\}^{\psi}$. Then $x_1, \ldots, x_n$ spans $\{0,1\}^{\psi}$ except with probability $2^{1-\psi}$.*

Proof. Define random variables $Y_1, \ldots, Y_n$ where $Y_i = 0$ if $x_1, \ldots, x_{i-1}$ spans $\{0,1\}^{\psi}$ or the span of $x_1, \ldots, x_{i-1}$ does not include x_i. Let $Y_i = 1$ in all other cases. Note that if $x_1, \ldots, x_{i-1}$ spans $\{0,1\}^{\psi}$, then $\Pr(Y_i = 1) = 0 \leq \frac{1}{2}$ and that if $x_1, \ldots, x_{i-1}$ does not span $\{0,1\}^{\psi}$, then they span at most half of the vectors in $\{0,1\}^{\psi}$ and hence again $\Pr(Y_i = 1) \leq \frac{1}{2}$. This means that it holds for all Y_i that $\Pr(Y_i = 1) \leq \frac{1}{2}$ independently of the values of Y_j for $j \neq i$. This implies that if we let $Y = \sum_{i=1}^{n} Y_i$, then

$$\Pr(Y \geq \frac{1}{2}(a + n)) \leq 2e^{-a^2/2n} ,$$

using the random walk bound. Namely, let $X_i = 2Y_i - 1$. Then $X_i \in \{-1, 1\}$ and it holds for all i that $\Pr(X_i = 1) \leq \frac{1}{2}$ independently of the other X_j. If the X_i had been independent and $\Pr(X_i = 1) = \Pr(X_i = -1) = \frac{1}{2}$, and $X = \sum_{i=1}^{n} X_i$, then the random walk bound gives that

$$\Pr(X \geq a) \leq 2e^{-a^2/2n} .$$

Since we have that $\Pr(X_i = 1) \leq \frac{1}{2}$ independently of the other X_j, the upper bound applies also to our setting. Then use that $X = 2Y - n$.

If we let $a = \frac{5}{2}\psi$, then $\frac{1}{2}(a + n) = \frac{7}{2}\psi = n - \psi$, $2e^{-a^2/2n} = 2e^{-\left(\frac{5}{2}\psi\right)^2/2\frac{9}{2}\psi} = 2e^{-\frac{25}{36}\psi}$, and $e^{-\frac{25}{36}} < \frac{1}{2}$. It follows that $\Pr(Y \geq n - \psi) \leq 2^{1-\psi}$. When $Y \leq n - \psi$, then $Y_i = 0$ for at least ψ values of i. This is easily seen to imply that $x_1, \ldots, x_n$ contains at least ψ linear independent vectors.

$\square$

Given the lemmas above we are now ready to conclude the proof of Theorem 9 for corrupt P_1.

Proof. As mentioned above we will not give a simulation argument but just prove that Δ is uniformly random in the view of a corrupt P_1 except with probability $2^{2-\psi}$. Turning this argument into a simulation argument is straight forward.

Recall that W is the event that $|S| \geq \tau - n$ and $c = 1$. By Lemma 9 we have that $\Pr(W) \leq 2^{-n} \leq 2^{-\psi}$. For the rest of the analysis we assume that W does not happen, i.e., $|S| < \tau - n$ and hence $|\bar{S}| \geq \tau = \frac{9}{2}\psi$. Since $\mathbf{A}$ is picked uniformly at random and independent of S it follows that $\frac{9}{2}\psi$ of the columns in $\mathbf{A}^{\bar{S}}$ are uniformly random and independent. Hence, by Lemma 10, they span $\{0,1\}^\psi$ except with probability $2^{1-\psi}$. We let D be the event that they do not span. If we assume that D does not happen, then by Lemma 8 Δ is uniform to P_1, i.e., if the event $F = W \cup D$ does not happen, then Δ is uniform to P_1. Since

$$\Pr(F) \leq \Pr(W) + \Pr(D) \leq 2^{-\psi} + 2^{1-\psi} \leq 2^{2-\psi} ,$$

we have that Δ is uniform to P_1 with overwhelming probability. □

Similar to Corollary 1 we can derive the following corollary for the $\mathcal{F}_{\Delta\text{-ROT}}$ functionality.

Corollary 2. *Let ψ denote the security parameter and let $\ell = \text{poly}(\psi)$. The functionality $\mathcal{F}_{\Delta\text{-ROT}}(\ell, \psi)$ can be reduced to $(\mathcal{F}_{\text{OT}}(2\tau, \psi), \mathcal{F}_{\text{EQ}}(\psi))$. The communication is $O(\psi\ell + \psi^2)$ and the work is $O(\psi^2\ell)$.*

Proof. Combining Theorem 9, 5 and 6 we have that $\mathcal{F}_{\Delta\text{-ROT}}(\ell, \psi)$ can be reduced to $(\mathcal{F}_{\text{OT}}(\frac{110}{6}\psi, \ell), \mathcal{F}_{\text{EQ}}(\frac{55}{6}\psi\ell))$ with communication $O(\psi^2)$ and work $O(\psi^2\ell)$. For any polynomial ℓ, we can implement $\mathcal{F}_{\text{OT}}(\frac{110}{6}\psi, \ell)$ given $\mathcal{F}_{\text{OT}}(\frac{110}{6}\psi, \psi)$ and a pseudo-random generator $\text{prg} : \{0,1\}^\psi \to \{0,1\}^\ell$. Namely, seeds are sent using the OTs and the prg is used to one-time pad encrypt the messages. The communication is 2ℓ. If we use the RO to implement the pseudo-random generator and count the hashing of ψ bits as $O(\psi)$ work, then the work is $O(\ell\psi)$ (using ℓ/ψ calls to H to expand ψ bit seeds to ℓ bits). We can implement $\mathcal{F}_{\text{EQ}}(\frac{55}{6}\psi\ell)$ by comparing short hashes produced using the RO. The work is $O(\psi\ell)$ (using $\frac{55}{6}\ell$ calls to H to hash the $\frac{55}{6}\psi\ell$-bit string down to ψ-bits). □

5.8 Complexity Analysis

We sketch a complexity analysis of the protocol in terms of the number of calls to the hash function H. We will count the number of total calls made by both P_1 and P_2.

To implement $\mathcal{F}_{\text{ROT}}(\ell, \psi)$ from LA-leaky $\mathcal{F}_{\Delta\text{-ROT}}(\ell, \tau)$ for ψ-secure LA, in Section 5.2 we hash 3ℓ strings of length $\tau = \frac{10}{6}\psi$, which we count as $\frac{3 \cdot 10}{6}\ell = 5\ell$ calls to H.

In Corollary 1 we show that LA-leaky $\mathcal{F}_{\Delta\text{-ROT}}(\ell, \frac{10}{6}\psi)$ for ψ-secure LA, can be reduced to $(\mathcal{F}_{\text{OT}}(\frac{20}{6}\psi, \psi)$, $\mathcal{F}_{\text{EQ}}(\psi))$. The construction in the proof of Corollary 1 involves using H to expand 3 ψ-bits seeds to ℓ bits for each OT, once for each of the senders messages and once the recieved message. This gives $\frac{3 \cdot 20}{6}\ell = 10\ell$ calls to H. We also need to use H to hash a $\frac{10}{6}\psi\ell$-bit string to ψ-bits in order to perform an equality check. Both P_1 and P_2 must do this giving a total of $\frac{20}{6}\ell$ calls to H. Thus in total we get $(5 + 10 + \frac{20}{6})\ell \leq 19\ell$ calls to H, apart from the cost of $(\mathcal{F}_{\text{OT}}(\frac{20}{6}\psi, \psi), \mathcal{F}_{\text{EQ}}(\psi))$. As $\mathcal{F}_{\text{ROT}}(\ell, \psi)$ provides ℓ random OTs this means only 19 calls to H are required per ROT.

To implement the non-leaky version of $\mathcal{F}_{\Delta\text{-ROT}}$ as in Section 5.7 we save the 5ℓ calls to H involved in implementing $\mathcal{F}_{\text{ROT}}$ from $\mathcal{F}_{\Delta\text{-ROT}}$. But on the other hand we will need longer keys in the leaky-$\mathcal{F}_{\Delta\text{-ROT}}$ functionality used in the construction described in the proof of Corollary 2. The construction and therefore the complexity analysis is identical to the one above for Corollary 1, only now we need $\frac{110}{6}\psi$ base OTs and an equality check for strings of length $\frac{55}{6}\psi\ell$ bit. Thus for the $\mathcal{F}_{\Delta\text{-ROT}}$ functionality we use around $74 \cdot \ell$ calls to H or 74 calls to H per Δ-ROT.

6 Preprocessing: The Multi-Party Case

Let $\mathcal{P}$ be a set of n parties. In this section we consider three variants of oblivious transfer suitable for scenarios with $n \geq 2$. The variants generalize standard OT in the sense that the role of *sender* and *receiver* is not played by single entities, but by subsets of $\mathcal{P}$ of arbitrary cardinality. The organization of this section is as follows, see Figure 23 for details.

In Section 6.1 we introduce the first two variants of oblivious transfer, $\mathcal{F}_{\text{ssCOT}}$ and $\mathcal{F}_{\text{ssOT}}$, and reduce them to their two-party counterparts, namely $\mathcal{F}_{\Delta\text{-ROT}}$ and $\mathcal{F}_{\text{ROT}}$. For active adversaries, the reductions are non robust, but this will be enough for our purposes. In Section 6.2 we consider the third variant, called $\mathcal{F}_{\text{auROT}}$, and construct it from the (non-robust) $\mathcal{F}_{\text{ssCOT}}$ and $\mathcal{F}_{\text{ssOT}}$. The implementation of $\mathcal{F}_{\text{auROT}}$ turns out to be robust, even for active adversaries. In Section 6.3 we present the protocol implementing the preprocessing phase, i.e. the protocol realizing the functionality $\mathcal{F}_{\text{Prep}}$ of Figure 10. It will be instantiated in the $(\mathcal{F}_{\text{ssOT}}, \mathcal{F}_{\text{ssCOT}}, \mathcal{F}_{\text{Comm}})$-hybrid model. This means that we can preprocess assuming only the two functionalities $\mathcal{F}_{\text{OT}}$ and $\mathcal{F}_{\text{Comm}}$ of Figure 1 and Figure 2, respectively. Finally, in Section 6.4 we discuss the complexity when the construction is instantiated for two or more parties.

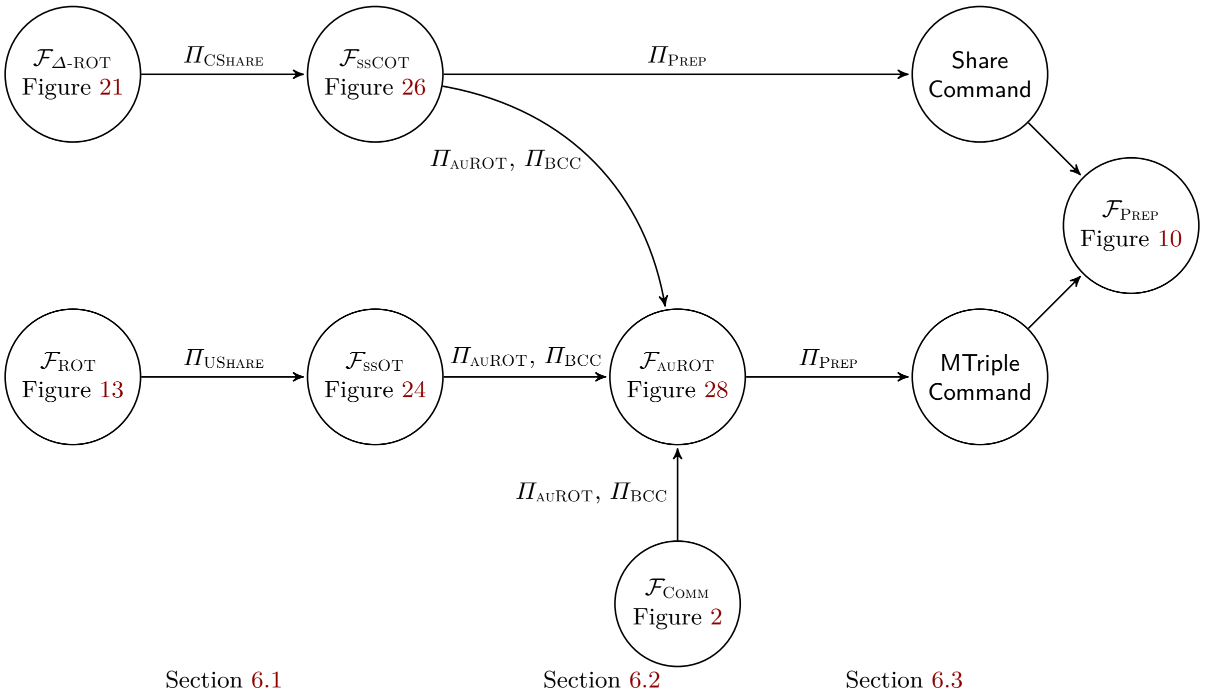

$\mathcal{F}_{\Delta\text{-ROT}}$ Figure 21
Π_{CShare}
$\mathcal{F}_{\text{ssCOT}}$ Figure 26
Π_{Prep}
Share Command
$\Pi_{\text{auROT}}, \Pi_{\text{BCC}}$
$\mathcal{F}_{\text{Prep}}$ Figure 10
$\mathcal{F}_{\text{ROT}}$ Figure 13
Π_{UShare}
$\mathcal{F}_{\text{ssOT}}$ Figure 24
$\Pi_{\text{auROT}}, \Pi_{\text{BCC}}$
$\mathcal{F}_{\text{auROT}}$ Figure 28
Π_{Prep}
MTriple Command
$\Pi_{\text{auROT}}, \Pi_{\text{BCC}}$
$\mathcal{F}_{\text{Comm}}$ Figure 2
Section 6.1
Section 6.2
Section 6.3

Figure 23 Relationship Between the Functionalities Implementing $\mathcal{F}_{\text{Prep}}$

6.1 Secret Shared OT

A secret shared oblivious transfer, $\text{ssOT}(a, b, \ell, \tau)$ is an n-party primitive defined as follows. For any subset $\mathcal{I} \subseteq \mathcal{P}$ of receivers, and any subset $\mathcal{J} \subseteq \mathcal{P}$ of senders, of size a and b respectively, the parties in $\mathcal{J}$ have ℓ pair of secrets vectors $(\mathbf{x}_s^{(0)}, \mathbf{x}_s^{(1)})_{s \leq \ell}$ in $\mathbb{F}_2^\tau$. The parties in $\mathcal{I}$ have a decision vector $\mathbf{b}$ in $\mathbb{F}_2^\ell$, such that:

- The set of receivers *obliviously* retrieves vector $\mathbf{u}_s = \mathbf{x}_s^{(0)} \oplus (b_s \cdot \mathbf{x}_s^{(1)})$ for each $s \leq \ell$, where the meaning of oblivious retrieving is the same as in a standard OT[15].

[15] More concretely, if $b_s = 0$ then $\mathbf{x}_s^{(1)}$ is uniformly distributed, over $\mathbb{F}_2^\tau$, in the *joint* view of the receivers $\mathcal{I} \setminus \mathcal{J}$, and if $b_s = 1$, both secrets vectors are uniformly distributed over the set of vectors that add up to $\mathbf{u}_s$. On the other hand, decision $\mathbf{b}$ is uniformly distributed, over $\mathbb{F}_2^\ell$, in the *joint* view of the senders $\mathcal{J} \setminus \mathcal{I}$.

- The decision $\mathbf{b}$, and the retrieved vectors $\mathbf{u}_s$ are additively secret shared between the receivers, i.e., parties in $\mathcal{I}$ have $\langle \mathbf{b} \rangle^{\mathcal{I}}$ and $\langle \mathbf{u}_s \rangle^{\mathcal{I}}$.
- The secret pairs $(\mathbf{x}_s^{(0)}, \mathbf{x}_s^{(1)})_{s \leq \ell}$ are additively secret-shared between the senders, i.e, parties in $\mathcal{J}$ have $\langle \mathbf{x}_s^{(c)} \rangle^{\mathcal{J}}$ for $s \leq \ell$, and $c = 0, 1$.

It is not difficult to see that one can obtain OT from ssOT; if sender P_1 wants to obliviously transmit one of the two secrets $(\mathbf{x}^{(0)}, \mathbf{x}^{(1)})$ in $\mathbb{F}_2^\tau$ to receiver P_2, the sender simply inputs $(\mathbf{x}^{(0)}, \mathbf{x}^{(0)} \oplus \mathbf{x}^{(1)})$ to an $\mathrm{ssOT}(1, 1, 1, \tau)$. The other direction is not straightforward though, and requires communication between the parties.

First observe that for (some arbitrary and previously fixed subsets $\mathcal{I}, \mathcal{J} \subseteq \mathcal{P}$ of sizes a and b, respectively) the output of an $\mathrm{ssOT}(a, b, \ell, \tau)$ gives ℓ representations $([x_s]^{\mathcal{I}}_{\mathcal{J},\mathbf{d}})_{s \leq \ell}$ over $\mathbb{F}_2^\tau$, i.e., sharings $\{\langle x_s \rangle^{\mathcal{I}}, \langle \mathbf{u}_s \rangle^{\mathcal{I}}, \langle \mathbf{v}_s \rangle^{\mathcal{J}}, \langle \mathbf{d}_s \rangle^{\mathcal{J}}\}$ such that $\mathbf{u} = \mathbf{v} \oplus (x \cdot \mathbf{d})$. We are ultimately interested in realizing the case $\mathcal{I} = \mathcal{J} = \mathcal{P}$; note here that to ease the description, the indices will be dropped when this is the case.

The idea is that of building ssOT using pairwise executions of $\mathcal{F}_{\mathrm{ROT}}$ (see Figure 13) and seeing the outputs as the shares that the parties hold. The obvious problem being that pairwise $\mathcal{F}_{\mathrm{ROT}}$s are not consistent across the pairs of parties. Thus, we need to give to each ordered pair (P_i, P_j) of senders and receivers, quadruples $(x^{i,j}, \mathbf{u}^{i,j}, \mathbf{v}^{i,j}, \mathbf{d}^{i,j})$, where bits, or vectors, with superindex (i, j) are obtained by P_i during an interaction with P_j, such that

$$\bigoplus_{(i,j)} \mathbf{u}^{i,j} = \bigoplus_{(i,j)} \mathbf{v}^{i,j} \oplus \left(\left(\bigoplus_{(i,j)} x^{i,j} \right) \cdot \left(\bigoplus_{(i,j)} \mathbf{d}^{i,j} \right) \right).$$

This is clearly true if the quadruples obtained from $\mathcal{F}_{\mathrm{ROT}}$ are correct, namely $\mathbf{u}^{i,j} = \mathbf{v}^{j,i} \oplus (x^{i,j} \cdot \mathbf{d}^{j,i})$, and each party holds the same share across all the pairs it belongs to, i.e. $x^{i,j} = x^{i,j'}$ and $\mathbf{d}^{i,j} = \mathbf{d}^{i,j'}$. One way to achieve this is to let the parties choose or "switch" to their own shares, during the pairwise executions.

Toy Example. In the three-party case, $n = 3$, suppose we want to realize $\mathrm{ssOT}(1, 3, 1, \tau)$; thus there is one receiver, say $\mathcal{I} = \{P_1\}$, three senders $\mathcal{J} = \{P_1, P_2, P_3\}$, and they want to transmit one single vector in $\mathbb{F}_2^\tau$. The parties proceed in three steps.

1. Party P_j inputs a share vector $\mathbf{d}^j$ in $\mathbb{F}_2^\tau$, for $j = 1, 2, 3$, and P_1 inputs a bit x.
2. P_1 runs $\mathcal{F}_{\mathrm{ROT}}$ with P_2 and with P_3. As a result they obtain ROT quadruples $(r^{1,2}, \mathbf{u}^{1,2}, \mathbf{v}_0^{2,1}, \mathbf{v}_1^{2,1})$, and $(r^{1,3}, \mathbf{u}^{1,3}, \mathbf{v}_0^{3,1}, \mathbf{v}_1^{3,1})$.
3. P_2 switches to his chosen share $\mathbf{d}^2$, setting $\mathbf{p}^{2,1} = \mathbf{v}_0^{2,1} \oplus \mathbf{v}_1^{2,1}$ and sending to P_1
$$\mathbf{m}^{1,2} = \mathbf{d}^2 \oplus \mathbf{p}^{2,1}.$$
Similarly, P_3 switches to his chosen share $\mathbf{d}^3$, setting $\mathbf{p}^{3,1} = \mathbf{v}_0^{3,1} \oplus \mathbf{v}_1^{3,1}$, and sending $\mathbf{m}^{1,3} = \mathbf{d}^3 \oplus \mathbf{p}^{3,1}$ to P_1. Then P_1 sets
$$\mathbf{u} = \mathbf{u}^{1,2} \oplus \mathbf{u}^{1,3} \oplus \left(r^{1,2} \cdot \mathbf{m}^{1,2}\right) \oplus \left(r^{1,3} \cdot \mathbf{m}^{1,3}\right), \text{ and } \mathbf{v}^1 = x \cdot \mathbf{d}^1.$$
4. Next, P_1 switches to his bit x. He sends the XOR $c^{2,1} = x \oplus r^{1,2}$ to P_2 and $c^{3,1} = x \oplus r^{1,3}$ to P_3. Now P_2 and P_3 can adjust their shares accordingly, namely P_j, sets
$$\mathbf{v}^j = \mathbf{v}_0^{j,1} \oplus \left(c^{j,1} \cdot \mathbf{d}^j\right) \text{ for } j = 2, 3.$$

By the correctness of $\mathcal{F}_{\mathrm{ROT}}$, it is not difficult to see that it holds [16]

$$\mathbf{u} = \mathbf{v}^1 \oplus \mathbf{v}^2 \oplus \mathbf{v}^3 \oplus x \cdot (\mathbf{d}^1 \oplus \mathbf{d}^2 \oplus \mathbf{d}^3) \overset{\mathsf{def}}{=} \mathbf{v} \oplus (x \cdot \mathbf{d}),$$

where $\mathcal{P} = \{P_1,\ P_2, P_3\}$ has an additive sharing of vectors $\mathbf{v}$, $\mathbf{d}$, and P_1 has bit x, and vector $\mathbf{u}$. In other words, the three parties have representation $[x]^1_{\mathbf{d}}$. It is straightforward to apply the same ideas with arbitrary n, $\mathcal{I}$ and $\mathcal{J}$. Namely receivers in $\mathcal{I}$ and senders in $\mathcal{J}$ exchange their bits $c^{j,i}$ and vectors $\mathbf{m}^{i,j}$.

There is a subtlety in the example above. The first secret $\mathbf{v}$ is not chosen by the senders, but rather drawn from the uniform distribution, for our context this semi-randomized flavour is enough.

[16] Namely, $\mathcal{F}_{\mathrm{ROT}}$ guarantees that $\mathbf{u}^{1,j} = \mathbf{v}_0^{j,1} \oplus \left(r^{1,j} \cdot \mathbf{p}^{j,1}\right)$ for each $j = 2, 3$.

Passive Adversaries. For arbitrary n, receivers $\mathcal{I}$, and senders $\mathcal{J}$, the key point to argue security is that the bits x^i and the vectors $\mathbf{d}^j$, are one-time-padded with the random values $r^{i,j}$ and $\mathbf{p}^{j,i} = \mathbf{v}_0^{j,i} \oplus \mathbf{v}_1^{j,i}$ given by the pairwise $\mathcal{F}_{\mathrm{ROT}}$. Continuing with the toy example, if P_1 is honest, then bit x is uniformly distributed in the joint view of the senders P_2, P_3, and if at least one sender is honest, the secrets $\mathbf{v}$, $\mathbf{d}$ are uniformly distributed in the view of the receiver P_1. It is not difficult to turn this into a simulation argument to see that in the $\mathcal{F}_{\mathrm{ROT}}$ hybrid, ssOT is perfectly realized in the presence of semi-honest adversaries.

Non Robust ssOT. The technique we have just described is too strong to implement ssOT with fully malicious adversaries. We instead realize a relaxed version, where dishonest parties can choose to produce quadruples where the relation $\mathbf{u} = \mathbf{v} \oplus (x \cdot \mathbf{d})$ does not necessary holds. In Section 6.2 we will show how to use a non robust ssOT (together with a non robust ssCOT) to implement an authenticated secret shared ROT. To this end, as we will see, it is enough to restrict ourselves to ssOT with one receiver and n senders.

In the toy example, a malicious sender, say P_2, had no room to cheat, because from any mask vector $\mathbf{m}^{1,2}$ that he have may sent, a simulator can extract his chosen share $\mathbf{d}^2$, namely $\mathbf{d}^2 \overset{\mathsf{def}}{=} \mathbf{m}^{1,2} \oplus \mathbf{p}^{2,i}$, where $\mathbf{p}^{2,1}$ is given by $\mathcal{F}_{\mathrm{ROT}}$. Consequently, if there were at least one honest sender (as it certainly was the case since $\mathcal{J} = \mathcal{P}$), the vector $\mathbf{d}$ still remained uniform in the joint view of malicious senders, *and* the parties would hold $[x]^1_{\mathbf{d}}$, where the malicious shares $\mathbf{d}^j$ are extracted as explained. Also the other secret, vector $\mathbf{u}$ can be proved to be uniformly random using the privacy of $\mathcal{F}_{\mathrm{ROT}}$.

The situation is different if the malicious party was the receiver P_1. If he had sent two mask bits $c^{2,1}$, $c^{3,1}$ to P_2 and P_3 respectively, such that $c^{2,1} \oplus r^{1,2} \neq c^{3,1} \oplus r^{1,3}$ (where $r^{1,2}$, $r^{1,3}$ are given by $\mathcal{F}_{\mathrm{ROT}}$ to P_1) then there is no consistent bit x that P_1 was committing to via his masks. This effectively means that, a malicious receiver P_1 could have potentially shifted the representation $[x]^1_{\mathbf{d}}$ that the parties were after, by a vector living in the space spanned by the shares $\mathbf{d}^2$, $\mathbf{d}^3$ of honest P_2 and P_3. [17]

Thus, active corruptions produce faulty representations of the form $\mathbf{u} = \mathbf{v} \oplus (x \cdot \mathbf{d}) \oplus \bigoplus_{h \notin A} e^h \cdot \mathbf{d}^h$, where the e^h's are adversarial offsets, and A is the set of corrupted parties. In Figure 24 we present the ideal functionality that models a non-robust ssOT with one receiver and n senders. (With uniform first secret $\mathbf{v}$.)

In $\mathcal{F}_{\mathrm{ssOT}}$ (and later in $\mathcal{F}_{\mathrm{ssCOT}}$), we model this malicious behaviour by allowing the adversary to input vectors $\mathbf{x}_s = (x_s^1, \ldots, x_s^n) \in \mathbb{F}_2^n$, $s \leq \ell$, instead of bits. If $x_s^1 = \cdots = x_s^n$, it means that P_i inputs consistent bits to each P_j, $j \neq i$, and that $x_s = x_s^j$.

Realizing $\mathcal{F}_{ssOT}$. It is not difficult to formalize the ideas sketched in the toy example, to ℓ transfers of vectors of length τ. We present the protocol in Figure 25. Here let us recall what are the key points used in the extension: (1) the set of parties is divided into ordered pairs and each of these calls the $\mathcal{F}_{\mathrm{ROT}}$ functionality; (2) all parties (i.e. the senders) "switch" from the second secrets given by $\mathcal{F}_{\mathrm{ROT}}$ to their desired share of their second secrets; (3) the party designated to be the receiver, "switches" from the random bit decisions given by $\mathcal{F}_{\mathrm{ROT}}$ to his choice bits. To obtain ℓ representations $[x_s]^i_{\mathbf{d}}$ the extension makes n-1 calls to $\mathcal{F}_{\mathrm{ROT}}(\ell, \tau)$. We obtain the following lemma.

Lemma 11. *In the $\mathcal{F}_{ROT}(\ell, \tau)$-hybrid model, protocol $\Pi_{\mathrm{USHARE}}(1, n, \ell, \tau)$ of Figure 25 implements $\mathcal{F}_{ssOT}(1, n, \ell, \tau)$ of Figure 24, with perfect security against static adversaries corrupting up to n-1 parties.*

Proof. Let $\mathcal{Z}$ denote the environment and $\mathcal{S}$ the ideal world adversary. We assume *authenticated* communication between parties, that is, they are given access to a functionality $\mathcal{F}_{\mathrm{AT}}$, which on input (m, i, j) from P_i, it gives message τ to P_j and $\mathcal{S}$.

$\mathcal{S}$ starts invoking an internal copy of the real adversary $\mathcal{A}$ and setting dummy parties π_i for $i \in 1, \ldots, n$. It then runs an internal execution of Π_{USHARE} between $\mathcal{A}$ and the π_i's, where every incoming communication from $\mathcal{Z}$ is forwarded to $\mathcal{A}$, and any outgoing communication from $\mathcal{A}$ is forwarded to $\mathcal{Z}$. The description of $\mathcal{S}$ is as follows. (Below, index i denotes the party designated as the receiver, and A the indices corresponding

[17] Roughly, if P_1 deviated in the toy example, and we let e.g. $x = c^{2,1} \oplus r^{1,2}$, sending $(x \oplus 1) \oplus r^{1,3}$ to P_3 caused the parties to obtain quadruple $(x, \mathbf{u}, \mathbf{v}, \mathbf{d})$, s.t. $\mathbf{u} = \mathbf{v} \oplus (x \cdot \mathbf{d}) \oplus \mathbf{d}^3$. Here $\mathbf{d} = \mathbf{d}^1 \oplus \mathbf{d}^2 \oplus \mathbf{d}^3$, and $\mathbf{d}^j$ is known to P_j

Functionality $\mathcal{F}_{\text{ssOT}}(1, n, \ell, \tau)$

The functionality is parametrized by the number of transfers ℓ, and the length of the transmitted vectors τ. The functionality performs ℓ transfers from one party to n parties. The ideal adversary is denoted by $\mathcal{S}$.

Input On input $(\mathsf{ssOT}, i, \mathbf{x}, \mathbf{d}^i_1, \dots, \mathbf{d}^i_\ell)$ from party P_i, and $(\mathsf{ssOT}, i, \mathbf{d}^j_1, \dots, \mathbf{d}^j_\ell)$ from party $P_j \neq P_i$, where $\mathbf{x} = (\mathbf{x}_1, \dots \mathbf{x}_\ell) \in \mathbb{F}_2^{n \cdot \ell}$, and $\mathbf{d}^i_s, \mathbf{d}^j_s \in \mathbb{F}_2^\tau$, for $s \leq \ell$. The functionality sets $\mathbf{d}_s = \mathbf{d}^1_s \oplus \dots \oplus \mathbf{d}^n_s$, and creates representations $[x_s]^i_{\mathbf{d}_s}$ as follows.

Honest Parties

1. Sample random $\mathbf{v}_s \in \mathbb{F}_2^\tau$ and set $\mathbf{u}_s = \mathbf{v}_s \oplus (x_s \cdot \mathbf{d}_s)$.
2. Generate additive sharing $\langle \mathbf{v}_s \rangle = (\mathbf{v}^1_s, \dots, \mathbf{v}^n_s)$.

The functionality outputs $(\mathbf{u}_s, \mathbf{v}^i_s)_{s \leq \ell}$ to party P_i, and $(\mathbf{v}^j_s)_{s \leq \ell}$ to party P_j for $1 \leq j \neq i \leq n$.

Corrupt Parties

The functionality waits until $\mathcal{S}$ specifies the set of corrupted parties A and their inputs.
Then for each index s it does the following:

- If $\mathcal{S}$ specifies $(\mathsf{Honest\text{-}P}_i, \mathtt{deliver})$ (i.e. the receiver P_i is not corrupt), it generates ℓ representations $[x_s]^i_{\mathbf{d}_s}$ as in the honest case. It then outputs $(\mathbf{u}_s, \mathbf{v}^i_s)_{s \leq \ell}$ to P_i, and $(\mathbf{v}^h_s)_{s \leq \ell}$ to each honest sender $P_h \neq P_i$.
- If $\mathcal{S}$ specifies $(\mathsf{Corrupt\text{-}P}_i)$ (i.e. the receiver P_i is corrupt), and $\mathcal{S}$ additionally specifies ℓ tuples of offset bits $(e^h_s)_{h \notin A, s \leq \ell}$. The functionality generates ℓ representations $[x_s]^i_{\mathbf{d}_s}$ as in the honest case, and outputs $(\mathbf{v}^h_s \oplus (e^h_s \cdot \mathbf{d}^h_s))_{s \leq \ell}$ to honest P_h. Thus, if we let $\mathbf{e}^H_s = \bigoplus_{h \notin A} e^h_s \cdot \mathbf{d}^h$ it holds

$$\mathbf{u}_s = \mathbf{v}_s \oplus (x_s \cdot \mathbf{d}_s) \oplus \mathbf{e}^H_s$$

Figure 24 Non Robust Secret Shared OT with One Receiver and n Senders

Protocol $\Pi_{\text{UShare}}(1, n, \ell, \tau)$

The protocol is parametrized by the number of transfers ℓ, and the length of the transmitted vectors τ. (Below, bits, or vectors, with superindex (a, b) indicates that P_a has the bit, or the vector, after an interaction with P_b)

On input $(\mathsf{ssOT}, i, \mathbf{x}, \mathbf{d}^i_1, \dots \mathbf{d}^i_\ell)$ from party P_i, and $(\mathsf{ssOT}, i, \mathbf{d}^j_1, \dots, \mathbf{d}^j_\ell)$ from party P_j, $j \neq i$, where $\mathbf{x} \in \mathbb{F}_2^\ell$ and $\mathbf{d}^i_s, \mathbf{d}^j_s \in \mathbb{F}_2^\tau$, the parties do the following:

1. **Switch Secrets:** Each party P_j, such that $j \neq i$, interacts with P_i as follows.
   (a) Call $\mathcal{F}_{\text{ROT}}$ on input $(\mathtt{start}, i, j)$. This returns ℓ ROTs, i.e. $(\mathbf{v}^{j,i}_{s,0}, \mathbf{v}^{j,i}_{s,1})$ to P_j and $(\mathbf{u}^{i,j}_s, r^{i,j}_s)$ to P_i, for each $s = 1, \dots, \ell$, such that $\mathbf{u}^{i,j}_s = \mathbf{v}^{j,i}_{s, r^{i,j}_s}$.
   (b) P_j computes pads $\mathbf{p}^{j,i}_s = \mathbf{v}^{j,i}_0 \oplus \mathbf{v}^{j,i}_1$, and sends masks $\mathbf{m}^{i,j}_s = \mathbf{d}^j_s \oplus \mathbf{p}^{j,i}_s$ to P_i, for $s \leq \ell$.
   (c) P_j stores $\mathbf{v}^{j,i}_{s,0}$ in a list for later use.
2. P_i samples random vectors $\mathbf{f}_s \in \mathbb{F}_2^\tau$, for $s \leq \ell$, and then sets

$$\mathbf{u}_s = \mathbf{f}_s \oplus \left(\bigoplus_{j \neq i} \mathbf{u}^{i,j}_s \oplus (r^{i,j}_s \cdot \mathbf{m}^{i,j}_s) \right) \quad \text{and} \quad \mathbf{v}^i_s = \mathbf{f}_s \oplus (x_s \cdot \mathbf{d}^i_s).$$

3. **Switch decision vector:** P_i and P_j, $j \neq i$, interact as follows.
   (a) P_i sends mask bits $c^{j,i}_s = x_s \oplus r^{i,j}_s$ to P_j, for each $s \leq \ell$.
   (b) P_j recovers $\mathbf{v}^{j,i}_{s,0}$, and sets $\mathbf{v}^j_s = \mathbf{v}^{j,i}_{s,0} \oplus (c^{j,i}_s \cdot \mathbf{d}^j_s)$, $s = 1, \dots, \ell$.
4. P_i outputs $(\mathbf{u}_s, \mathbf{v}^i_s)_{s \leq \ell}$, and P_j outputs $(\mathbf{v}^j_s)_{s \leq \ell}$.

Figure 25 Bootstrapping from Two-party OT to n-party ssOT with One Receiver and n Senders

to the subset of corrupted parties).

Simulation for honest receiver P_i

- $\mathcal{S}$ initializes the inputs of honest dummy parties $(\pi_h)_{h\notin A}$ at random and receives inputs of corrupt parties from $\mathcal{A}$. $\mathcal{S}$ activates copies of $\mathcal{F}_{\text{AT}}$ and $\mathcal{F}_{\text{ROT}}$ that it uses in the internal execution of Π_{USHARE} with $\mathcal{A}$, i.e. $\mathcal{S}$ has access to all data provided by these two functionalities.
- For each pair of parties (P_i, P_j), it emulates $\mathcal{F}_{\text{ROT}}$ and sends $(\mathbf{v}^{c,i}_{s,0}, \mathbf{v}^{c,i}_{s,1})_{c\in A}$ to $\mathcal{A}$.
- In step 1b, for each $c \in A$, $\mathcal{S}$ internally receives masks $(\mathbf{m}^{i,c}_s)_{s\leq\ell}$ from $\mathcal{A}$, and sets $\bar{\mathbf{d}}^c_s = \mathbf{m}^{i,c}_s \oplus \mathbf{p}^{c,i}_s$, where $\mathbf{p}^{c,i}_s$ is specified by $\mathcal{F}_{\text{ROT}}$, i.e. $\mathbf{p}^{c,i}_s = \mathbf{v}^{c,i}_{s,0} \oplus \mathbf{v}^{c,i}_{s,1}$.
- The simulator externally notifies to $\mathcal{F}_{\text{ssOT}}$ that A is corrupt, and gives $(\bar{\mathbf{d}}^c_s)_{c\in A, s\leq\ell}$, as their inputs. Then externally sends $(\mathsf{Honest\text{-}P}_i, \mathtt{deliver})$ to $\mathcal{F}_{\text{ssOT}}$. It outputs whatever the adversary outputs and halts.

Simulation for corrupt receiver P_i

- $\mathcal{S}$ initializes the inputs of honest dummy parties $(\pi_h)_{h\notin A}$ at random, and receives inputs of corrupt parties from $\mathcal{A}$.
- Emulating $\mathcal{F}_{\text{ROT}}$ and $\mathcal{F}_{\text{AT}}$ it forwards $(r^{i,j}_s, \mathbf{u}^{i,j}_s)$ and $(\mathbf{v}^{i,c}_{s,0}, \mathbf{v}^{i,c}_{s,1})_{c\in A}$ to $\mathcal{A}$.
- In step 3a, and for each $h \notin A$, $\mathcal{S}$ receives mask bits $c^{h,i}_s$ from $\mathcal{A}$, and extracts the inputs of the corrupt parties by computing $x^{i,h}_s = c^{h,i}_s \oplus r^{i,h}_s$. Now the simulator checks that $x^{i,h}_s = x^{i,h'}_s$, $\forall h, h' \notin A$. If this is not the case, it computes the offset bit $(e^h_s)_{h\notin A}$.
- $\mathcal{S}$ externally sends $(\mathsf{Corrupt\text{-}P}_i)$ and $(e^h_s)_{h\notin A, s\leq\ell}$ to $\mathcal{F}_{\text{ssOT}}$, outputs what the adversary outputs and halts.

We now argue indistinguishability.

Case none of the parties are corrupted. We start showing that the environment does not distinguish the real process and the ideal process when no corruption occurs. First, in $\mathcal{F}_{\text{ssOT}}(1, n, \ell, \tau)$, party P_i inputs decision vector $\mathbf{x}$ in $\mathbb{F}_2^\ell$, and shares $(\mathbf{d}^i_1, \dots, \mathbf{d}^i_\ell)$ in $\mathbb{F}_2^\tau$, any other party P_j inputs shares $(\mathbf{d}^j_1, \dots, \mathbf{d}^j_\ell)$ in $\mathbb{F}_2^\tau$. They obtain ℓ representations $[x_s]^i_{\mathbf{d}_s}$, with $\mathbf{d}_s = \mathbf{d}^1_s \oplus \cdots \oplus \mathbf{d}^n_s$, i.e. sharings $(\langle \mathbf{u}_s\rangle^i, \langle \mathbf{v}_s\rangle^{\mathcal{P}})$ such that $\mathbf{u}_s = \mathbf{v}_s \oplus (x_s \cdot \mathbf{d}_s)$, with $\mathbf{v}_s$ drawn uniformly.

Expanding out the output of parties in Π_{USHARE}, we see that for each $s \leq \ell$ the following holds.

$$\begin{aligned}\mathbf{u}_s =& \mathbf{f}_s \oplus \bigoplus_{j\neq i} \mathbf{u}^{i,j}_s \oplus \bigoplus_{j\neq i} r^{i,j}_s \cdot \mathbf{m}^{i,j}_s \\ =& \mathbf{f}_s \oplus \bigoplus_{j\neq i} \mathbf{u}^{i,j}_s \oplus \bigoplus_{j\neq i} r^{i,j}_s \cdot (\mathbf{d}^j_s \oplus \mathbf{v}^{j,i}_{s,0} \oplus \mathbf{v}^{j,i}_{s,1}) \end{aligned} \tag{9}$$

where

$$\mathbf{u}^{i,j}_s = \bigoplus_{j\neq i} \mathbf{v}^{j,i}_{s,0} \oplus \bigoplus_{j\neq i} r^{i,j} \cdot (\mathbf{v}^{j,i}_{s,0} \oplus \mathbf{v}^{j,i}_{s,1}). \tag{10}$$

So, putting together (9) and (10), we get:

$$\mathbf{u}_s = \mathbf{f}_s + \bigoplus_{j\neq i} \mathbf{v}^{j,i}_{s,0} \oplus \bigoplus_{j\neq i} r^{i,j}_s \cdot \mathbf{d}^j_s.$$

Since

$$\mathbf{v}_s = \mathbf{v}^i_s \oplus \bigoplus_{j\neq i} \mathbf{v}^j_s = \mathbf{f}_s + x_s \cdot \mathbf{d}^i_s \oplus \bigoplus_{j\neq i} \mathbf{v}^{j,i}_{s,0} \oplus \bigoplus_{j\neq i} \mathbf{d}^j_s \cdot (x_s \oplus r^{i,j}_s),$$

we obtain $\mathbf{u}_s \oplus \mathbf{v}_s = x_s \cdot \mathbf{d}_s$. Since the share $\mathbf{v}^i_s$ is randomized via vector $\mathbf{f}_s$, and the shares $\mathbf{v}^j_s$, $j \neq i$, via $\mathcal{F}_{\text{ROT}}$, we conclude that the output in both processes are identically distributed. Second, in $\mathcal{Z}$'s view, the transcript in the real process, is not bind to the real inputs because $\mathcal{Z}$ has no access to the pads bits $r^{i,j}_s$ nor to the pad vectors $\mathbf{p}^{j,i}_s$, so the ideal transcript could also have been seen in the real process, with the same inputs, that $\mathcal{Z}$ gives. (We are implicitly reusing the simulator for the case P_i is honest and setting $A = \emptyset$.)

Case P_i is corrupted. In this case, step 3a is the only point in the real process (or in the ideal process) where $\mathcal{A}$ could have deviated and send inconsistent bit masks to honest parties. In the ideal process $\mathcal{S}$ can see this bad behaviour because it has access to both $c_s^{h,i}$ and $r_s^{i,h}$, $\forall h \notin A$ and it can correctly reconstruct vector $\mathbf{x}_s, s \leq \ell$. Thus, for each $h \notin A$, the simulator computes $(e_s^h)_{h \notin A, s \leq l}$, such that $x_s^h = x_s + e_s^h$, and forwards values e_s^h and x_s to the functionality. Therefore, the outputs are identically distributed in both processes. On the other hand, the honest part of the transcript that $\mathcal{Z}$ sees in either process, namely honest masks $\mathbf{m}^{h,i}$, is again not bind to the inputs vector $\mathbf{d}_s^h$ of honest parties, because $\mathcal{Z}$ has not access to pads $\mathbf{p}^{h,i}$, for $h \notin A$. This concludes the case P_i is corrupt.

Case P_i is honest. In this case, the only point where the adversary can deviate is in step 1b. Using a similar argument as in the previous case, in the ideal process $\mathcal{S}$ can recover the corrupt share $\mathbf{d}_s^c$ from the mask $\mathbf{m}_s^{i,c}$ sent to π_i by π_c, and tell $\mathcal{F}_{\text{ssOT}}$ which is the corrupt input that $\mathcal{A}$ chooses for P_c. In other words, the only attack that a real adversary can do when the receiver P_i is not corrupted, is to change the input of the corrupted parties, and this does not affect correctness. Ths concludes the case P_i is honest and the proof of the lemma. □

Secret Shared Correlated OT The second variant that we consider is the counterpart of Δ-ROT. A Secret Shared Correlated OT $(\text{ssCOT}(a, b, \ell, \tau))$ is like an $\text{ssOT}(a, b, \ell, \tau)$, the only difference is that the second secret in all the pairs are fixed to the same vector. Thus $\mathbf{x}_s^{(1)} = \mathbf{c}$ for each $s \leq \ell$, where $\mathbf{c}$ is the correlation vector.

One can use $\mathcal{F}_{\Delta\text{-ROT}}$ (see Figure 21), to implement $\mathcal{F}_{\text{ssCOT}}(1, n, \ell, \tau)$ where many correlated quadruples $([x_s]^1_{\mathbf{d}})_{s \leq \ell}$ are produced, using virtually the same techniques as for ssOT. The difference being that for *all* the transfers of the same batch, one defines the pad vector for the chosen share $\mathbf{d}^j$ of P_j as the correlation vector $\mathbf{c}^j$ that is obtained in a pairwise execution with the sender P_i. Observe that it requires less communication, as now the switch from random to chosen shares only needs to be done once per batch of quadruples.

For the sake of completeness we include the functionality and the protocol realizing it, in Figure 26 and Figure 27. They are very similar to the ones defined previously. (See Figure 26 and Figure 27.) Not surprisingly, we obtain the following lemma.

Lemma 12. *In the $\mathcal{F}_{\Delta\text{-ROT}}(\ell, \tau)$-hybrid model, protocol $\Pi_{\text{CShare}}(1, n, \ell, \tau)$ of Figure 27 implements the functionality $\mathcal{F}_{\text{ssCOT}}(1, n, \ell, \tau)$ of Figure 26, with perfect security against static adversaries corrupting up to n-1 parties.*

Proof. The proof is identical as the proof of Lemma 11. Correcntess follows from the correctness of $\mathcal{F}_{\Delta\text{-ROT}}$. The ideal and the ideal processes are indistinguishable also because the honest pads $\mathbf{c}^{h,i}$ are randomly distributed. (If P_i is honest the same goes for the pads $r_s^{i,c}$ used in the sth switch.) □

From One to Many Receivers Assuming access to an $\text{ssOT}(1, n, \ell, \tau)$ the parties can easily generate representations $[x_s] = \{\langle x_s \rangle, \langle \mathbf{u}_s \rangle, \langle \mathbf{v}_s \rangle, \langle \mathbf{d}_s \rangle\}_{s \leq \ell}$ over $\mathbb{F}_2^\tau$. Thus, after n executions of $\Pi_{\text{UShare}}(1, n, \ell, \tau)$ in the same input, where in the ith execution party P_i act as the receiver, the parties have obtained $[x_s^i]^i_{\mathbf{d}_s} = \{\langle x_s^i \rangle^i, \langle \mathbf{u}_{s,i} \rangle, \langle \mathbf{v}_{s,i} \rangle, \langle \mathbf{d}_s \rangle\}$, for $s \leq \ell$, adding up, they obtain:

$$[x_s^1]^1_{\mathbf{d}_s} \oplus \cdots \oplus [x_s^n]^n_{\mathbf{d}_s} = [x_s^1 \oplus \cdots \oplus x_s^n]_{\mathbf{d}_s} \overset{\text{def}}{=} [x_s]_{\mathbf{d}_s}.$$

The arithmetic defined in Section 2 ensures that if the parties use the same share $\mathbf{d}^j$ across the different executions, then the equality holds. Moreover, the secret shared bit x_s is uniform in the view of any $n-1$ parties. This is enough if one is considering semi honest adversaries. Observe that the same can be done to obtain correlated representations $[x_1]_{\mathbf{c}}, \ldots, [x_\ell]_{\mathbf{c}}$. (Using Π_{CShare} instead.)

Nevertheless the interesting case is when the parties are fully malicious. The effect of active adversaries in each call was quantified in Lemma 11. Namely, for an error vector $\mathbf{e}_R^H = \sum_{h \notin A} e_h \cdot \mathbf{d}^h$, with offsets bits e_h chosen by the adversary, and vector $\mathbf{d}^h$ only known to honest P_h, in the cth call the corrupted receiver

The Functionality $\mathcal{F}_{\text{ssCOT}}(1, n, \ell, \tau)$

The functionality is parametrized by the number of transfers ℓ, and the length of the transmitted vectors τ. The ideal adversary is denoted by $\mathcal{S}$.

Input On input $(\mathsf{ssCOT}, i, \mathbf{x}, \mathbf{d}^i)$ from party P_i, and $(\mathsf{ssCOT}, i, \mathbf{d}^j,)$ from party $P_j \neq P_i$, where $\mathbf{x} = (\mathbf{x}_1, \dots, \mathbf{x}_\ell) \in \mathbb{F}_2^{n \cdot \ell}$, and $\mathbf{d}^i, \mathbf{d}^j \in \mathbb{F}_2^\tau$, The functionality sets $\mathbf{d} = \mathbf{d}^1 \oplus \cdots \oplus \mathbf{d}^n$, and creates ℓ representations $[x_s]^i_{\mathbf{d}}$ as follows.

Honest Parties

1. Sample random $\mathbf{v}_s \in \mathbb{F}_2^\tau$ and set $\mathbf{u}_s = \mathbf{v}_s \oplus (x_s \cdot \mathbf{d})$.
2. Generate additive sharing $\langle \mathbf{v}_s \rangle = (\mathbf{v}_s^1, \dots, \mathbf{v}_s^n)$.

The functionality outputs $(\mathbf{u}_s, \mathbf{v}_s^i)_{s \leq \ell}$ to party P_i, and $(\mathbf{v}_s^j)_{s \leq \ell}$ to party P_j, for each $j \neq i$.

Corrupt Parties

The functionality waits until $\mathcal{S}$ specifies the set of corrupted parties A and their inputs. Then for each index s it does the following.

- If $\mathcal{S}$ specifies $(\mathsf{Honest\text{-}P}_i, \mathtt{deliver})$ (i.e. the receiver P_i is not corrupt), it generates ℓ representations $[x_s]^i_{\mathbf{d}}$ as in the honest case. It then outputs $(\mathbf{u}_s, \mathbf{v}_s^i)_{s \leq \ell}$ to P_i, and $(\mathbf{v}_s^h)_{s \leq \ell}$ to honest sender $P_h \neq P_i$.
- If $\mathcal{S}$ specifies $(\mathsf{Corrupt\text{-}P}_i)$ (i.e. the receiver P_i is corrupt), and $\mathcal{S}$ additionally specifies ℓ tuples of offset bits $(e_s^h)_{h \notin A, s \leq \ell}$. The functionality generates ℓ representations $[x_s]^i_{\mathbf{d}}$ as in the honest case, and outputs $(\mathbf{v}_s^h \oplus (e_s^h \cdot \mathbf{d}^h))_{s \leq \ell}$ to honest P_h. Thus, if we let $\mathbf{e}_s^H = \bigoplus_{h \notin A} e_s^h \cdot \mathbf{d}^h$ it holds

$$\mathbf{u}_s = \mathbf{v}_s \oplus (x_s \cdot \mathbf{d}) \oplus \mathbf{e}_s^H$$

Figure 26 Non Robust SS Correlated OT with One Receiver and n Senders

The Protocol $\Pi_{\text{CShare}}(1, n, \ell, \tau)$

The protocol is parametrized by the number of transfers ℓ, and the length of the transmitted vectors τ. (Below, bits, or vectors, with superindex (a, b) indicates that P_a has the bit, or the vector, after an interaction with P_b).//
On input $(\mathsf{ssOT}, i, \mathbf{x}, \mathbf{d}^i)$ from party P_i, and $(\mathsf{ssOT}, i, \mathbf{d}^j)$ from party $P_j \neq P_i$, where $\mathbf{x} \in \mathbb{F}_2^\ell$, and $\mathbf{d}^i, \mathbf{d}^j \in \mathbb{F}_2^\tau$, the parties do the following:

1. **Switch Secrets:** Each $P_j \neq P_i$ does the following with P_i
   (a) Call $\mathcal{F}_{\Delta\text{-ROT}}$ on input $(\mathtt{start}, i, j)$. The box returns ℓ Δ-ROT quadruples $(r_s^{i,j}, \mathbf{u}_s^{i,j}, \mathbf{v}_{s,0}^{j,i}, \mathbf{c}^{j,i})$, i.e. it holds $\mathbf{u}_s^{i,j} = \mathbf{v}_{s,0}^{j,i} \oplus (r_s^{i,j} \cdot \mathbf{c}^{j,i})$.
   (b) P_j sets pad $\mathbf{p}^{i,j} = \mathbf{c}^{j,i}$, and sends mask $\mathbf{m}^{i,j} = \mathbf{d}^j \oplus \mathbf{p}^{j,i}$ to P_i.
   (c) P_j stores $\mathbf{v}_{s,0}^{j,i}$ in a list for later use.
2. P_i samples random vectors $\mathbf{f}_s \in \mathbb{F}_2^\tau$, for $s \leq \ell$, and then sets $\mathbf{u}_s = \mathbf{f}_s \oplus \left(\bigoplus_{j \neq i} \mathbf{u}_s^{i,j} \oplus (r_s^{i,j} \cdot \mathbf{m}^{i,j}) \right)$, and $\mathbf{v}_s^i = \mathbf{f}_s \oplus (x_s \cdot \mathbf{d}^i)$.
3. **Switch decision vector:** P_i does the following with each other P_j
   (a) For $s \leq \ell$, P_i sends mask bits $c_s^{j,i} = x_s \oplus r_s^{i,j}$ to P_j.
   (b) For $s \leq \ell$, P_j recovers $\mathbf{v}_{s,0}^{j,i}$, and sets $\mathbf{v}_s^j = \mathbf{v}_{s,0}^{j,i} \oplus (c_s^{j,i} \cdot \mathbf{d}^j)$.
4. P_i outputs $(\mathbf{u}_s, \mathbf{v}_s^i)_{s \leq \ell}$, and P_j outputs $(\mathbf{v}_s^j)_{s \leq \ell}$.

Figure 27 Bootstrapping from Two-party OT to n-party ssCOT with One Receiver and n Senders

P_c can force the parties to create faulty representations $(x^c)^c_{\mathbf{d}} = \{\langle x^c \rangle^c, \langle \mathbf{u} \rangle^c, \langle \mathbf{u} \oplus \mathbf{e}_R^H \rangle, \langle \mathbf{d} \rangle\}$, where it holds $\mathbf{u} = \mathbf{v} \oplus (x^c \cdot \mathbf{d}) \oplus \mathbf{e}^H$. The other way a malicious adversary can influence the protocol is giving different inputs to the n calls of Π_{UShare} (i.e. now are the "senders" who are misbehaving). If we denote by $\mathbf{d} = \mathbf{d}^1 \oplus \cdots \oplus \mathbf{d}^n$ the vector obtained in the first call, then in the jth call, $2 \leq j \leq n$, the adversary, for an offset vector $\mathbf{f}_j$ chosen by him, can make the parties output $[x^j]^j_{\mathbf{d} \oplus \mathbf{f}_j}$, if P_j is honest, or faulty $(x^j)^j_{\mathbf{d} \oplus \mathbf{f}_j}$ if P_j is corrupt.

Adding both ways of influencing, if we let $\mathbf{e}_S^H = \sum_{h \notin A} x^h \cdot \mathbf{f}_h$, after combining the output of the n calls, the adversary can force the parties to create (faulty) representations $(x)_{\mathbf{d}} = \{\langle x \rangle, \langle \mathbf{u} \oplus \mathbf{e}_S^H \rangle, \langle \mathbf{v} \oplus \mathbf{e}_R^H \rangle, \langle \mathbf{d} \rangle\}$, where it holds

$$\mathbf{u} = \mathbf{v} \oplus (x \cdot \mathbf{d}) \oplus \mathbf{e}_R^H \oplus \mathbf{e}_S^H.$$

The difference between both error vectors is that $\mathbf{e}_S^H$ depends on the value of the honest share bits x^h. This is not the case with $\mathbf{e}_R^H$, that takes always the same value regardless the values of the honest bits. Thus, in some respect, influencing as a "sender" is stronger than influencing as a "receiver".

6.2 Authenticated Multi Random OT

In addition to the above two variants we also consider a third variant of oblivious transfer, which is the multiparty and authenticated notion of a standard random OT. Recall what we mean by authenticated bit e: we write $[\![e]\!]_{\boldsymbol{\alpha}}$ when the parties in $\mathcal{P}$ have $[\![e]\!]_{\boldsymbol{\alpha}} = \{\langle e \rangle, \langle \mathbf{u} \rangle, \langle \boldsymbol{\alpha} \rangle\}$ such that $\mathbf{u}, \boldsymbol{\alpha}$ are in $\mathbb{F}_2^\tau$, and it holds $\mathbf{u} = e \cdot \boldsymbol{\alpha}$. (The vector $\boldsymbol{\alpha}$ is the global key, and its length τ will be seen as a security parameter.)

For a vector $\boldsymbol{\alpha}$ uniformly sampled in $\mathbb{F}_2^\tau$, an authenticated secret-shared random oblivious transfer is a quadruple $[\![x^{(0)}]\!]_{\boldsymbol{\alpha}}, [\![x^{(1)}]\!]_{\boldsymbol{\alpha}}, [\![e]\!]_{\boldsymbol{\alpha}}, [\![z]\!]_{\boldsymbol{\alpha}}$ held by the set of parties $\mathcal{P}$ with the following two properties:

Privacy. The triplet $(x^{(0)}, x^{(1)}, e)$ is uniformly distributed in $\mathbb{F}_2$ in the joint view of any $n-1$ parties.
Correctness. It holds that $z = x^{(e)}$.

The ideal functionality $\mathcal{F}_{\text{AUROT}}$ modeling creation of (authenticated) random OT quadruples is in Figure 28. The rough idea to implement $\mathcal{F}_{\text{AUROT}}$ is to call $\mathcal{F}_{\text{SSOT}}$ for the generation of the secret-shared quadruple and $\mathcal{F}_{\text{SSCOT}}$ to authenticate it. Hereafter we employ a cut-and-choose technique to check that the quadruples are indeed correct and make sure they are also private. The protocol implementing these ideas, given in Figure 29, is broken in three steps that we examine in turn.

*Generating additive secret-shared quadruples. (***Share***)* A random OT quadruple $(x^{(0)}, x^{(1)}, e, z)$ is created calling n times $\mathcal{F}_{\text{SSOT}}$. In the ith call the parties obtain a representation $[e^i]_{\mathbf{d}}^i$ where bit share e^i is known to P_i. Exploiting the linearity of additive secret-shared values they set $[e]_{\mathbf{d}} = [e^1]_{\mathbf{d}}^1 \oplus \cdots \oplus [e^n]_{\mathbf{d}}^n$. The parties now have quadruple $\{\langle \mathbf{v} \rangle, \langle \mathbf{d} \rangle, \langle e \rangle, \langle \mathbf{u} \rangle\}$, such that $\mathbf{u} = \mathbf{v} \oplus e \cdot \mathbf{d}$. In other words if we let $\mathbf{x}^{(0)} = \mathbf{v}$, $\mathbf{x}^{(1)} = \mathbf{v} \oplus \mathbf{d}$, and $\mathbf{z} = \mathbf{u}$, then it holds $\mathbf{z} = \mathbf{x}^{(e)}$. The ROT quadruple $(\langle x^{(0)} \rangle, \langle x^{(1)} \rangle, \langle e \rangle, \langle z \rangle)$ is then given by decision bit e and the first bits of vectors $\mathbf{x}^{(0)}$, $\mathbf{x}^{(1)}$, $\mathbf{z}$; the quadruple is indeed secret-shared. Recall nothing stop corrupt parties to input distinct values in two calls to the functionality. For example, they may give $\langle \mathbf{d} \oplus 1 \rangle$ in the first call and $\langle \mathbf{d} \rangle$ in subsequent calls; this means that the parties end up with n representations of the form $[e^1]_{\mathbf{d} \oplus 1}^1$ and $[e^j]_{\mathbf{d}}^j$ for $j \geq 2$. After adding them up they would hold a faulty representation $(e)_{\mathbf{d}} = \{\langle \mathbf{v} \rangle, \langle \mathbf{d} \rangle, \langle e \rangle, \langle \mathbf{u} \oplus e^1 \rangle\}$, where now it holds $\mathbf{u} = \mathbf{v} \oplus e \cdot \mathbf{d}$. The derived quadruple is correct iff the share of the decision bit e^1 is zero, opening the door to selective-failure attacks that we prevent later.

*Putting MACs to the quadruples. (***Authenticate***)* To add a MAC to each bit of the quadruple the parties now call $\mathcal{F}_{\text{SSCOT}}$ n times. The input share of the correlated vector that the functionality receives from P_j is the jth share of the MAC key $\boldsymbol{\alpha}^j$. Party P_j authenticates his share bits of the (secret) quadruple $(x^{(0)}, x^{(1)}, e, z)$ in the jth call. Using again the linearity of the representations the parties can easily derive $[\![x^{(0)}]\!]_{\boldsymbol{\alpha}}$, $[\![x^{(1)}]\!]_{\boldsymbol{\alpha}}$, $[\![e]\!]_{\boldsymbol{\alpha}}$, $[\![z]\!]_{\boldsymbol{\alpha}}$. Again observe that the parties can input to the functionality values different to the ones created in the previous step. This potentially raises more selective-failure attacks (see below).

*Testing semi-honest behaviour. (***Sacrifice***)* Here we want to guarantee the correctness and privacy of the authenticated quadruples. We use a technique also found in [NO09, DKL+13, FJN+13] that works on large batches of quadruples. We start with $\gamma = \ell(b^3+1)$ quadruples $Q_s = ([\![x_s^{(0)}]\!]_{\boldsymbol{\alpha}}, [\![x_s^{(1)}]\!]_{\boldsymbol{\alpha}}, [\![e_s]\!]_{\boldsymbol{\alpha}}, [\![z_s]\!]_{\boldsymbol{\alpha}})_{s \leq \gamma}$ created in the previous steps, where ℓ is the final number of output quadruples and b is a secondary parameter. The technique is divided in three phases. We have separated it out from the main protocol of Figure 29 due to the length of the description, it is detailed in Figure 30 but let us here sketch the ideas behind the scenes.

The Functionality $\mathcal{F}_{\text{AUROT}}(\ell, \tau)$

This functionality is parametrized by the number of output authenticated ROT quadruples ℓ, and the length of the MAC key τ.

Initialize

On input (Init) from all the parties, the functionality samples a random vector $\boldsymbol{\alpha}$ in $\mathbb{F}_2^\tau$. When $\mathcal{S}$ inputs the set of corrupted parties A, the functionality computes a sharing $\langle \boldsymbol{\alpha} \rangle = (\boldsymbol{\alpha}^1, \ldots, \boldsymbol{\alpha}^n)$, where the corrupt shares $(\boldsymbol{\alpha}^c)_{c \in A}$ are specified by $\mathcal{S}$, and outputs $\boldsymbol{\alpha}^h$ to honest P_h.

Authenticated Secret Shared ROT

On input (auROT) from all the parties, the functionality waits for $\mathcal{S}$ to input `abort` or `deliver`. If it is told to abort, it outputs the special symbol $\varnothing$ to honest parties. Otherwise for $s \leq \ell$ it samples three random bits $(x_s^{(0)}, x_s^{(1)}, e_s)$, and sets $z_s = x_s^{(e^s)}$. Then generates authentications $[\![x_s^{(0)}]\!]_{\boldsymbol{\alpha}}$, $[\![x_s^{(1)}]\!]_{\boldsymbol{\alpha}}$, $[\![e_s]\!]_{\boldsymbol{\alpha}}$, $[\![z_s]\!]_{\boldsymbol{\alpha}}$ calling sub-procedure $\mathsf{Authenticate}(y)$ for every bit $y \in \{x_s^{(0)}, x_s^{(1)}, e_s, z_s\}_{s \leq \ell}$.

$\mathsf{Authenticate}(y)$:

Given a bit y, this sub-procedure produces an authentication $[\![y]\!]_{\boldsymbol{\alpha}} = (\langle y \rangle, \langle \mathbf{m} \rangle, \langle \boldsymbol{\alpha} \rangle)$, where $\boldsymbol{\alpha}$ is the MAC key.

1. Set $\mathbf{m} = y \cdot \boldsymbol{\alpha}$.
2. Generate bit sharing $\langle y \rangle = (y^1, \ldots, y^n)$ in $\mathbb{F}_2$, and MAC sharing $\langle \mathbf{m} \rangle = (\mathbf{m}^1, \ldots, \mathbf{m}^n)$ in $\mathbb{F}_2^\tau$.

Output $(y^j, \mathbf{m}^j)$ to party $P_j, j = 1, \ldots, n$.

Corrupt Parties:

1. The functionality sets $\mathbf{m} = y \cdot \boldsymbol{\alpha}$.
2. $\mathcal{S}$ specifies bit shares $\{y^c\}_{c \in A}$, and MAC shares $\{\mathbf{m}^c\}_{c \in A}$. Then the functionality creates sharings $\langle y \rangle$, $\langle \mathbf{m} \rangle$ where the portion of honest shares is consistent with adversarial shares.

Output $(y^h, \mathbf{m}^h)$ to honest P_h.

Figure 28 Authenticated ROT Quadruples

- First we ensure that not all the quadruples we start with are incorrect. The parties partially open ℓ of them and everyone checks that the algebraic relationship that gives correctness holds in the bit shares. This corresponds to step 4 of Phase-I in Figure 30.
- Next we enter Phase-II to test correctness in the quadruples that have not been discarded. The array is randomly permuted and divided in buckets $\mathcal{B}_1, \ldots, \mathcal{B}_\ell$ each of size $b \ll \sigma$ (step 5). Inside each bucket $\mathcal{B}_s = (Q_{s,1}, \ldots, Q_{s,b})$ one quadruple is chosen, say the first one $Q_{s,1}$, and the others $b - 1$ quadruples are used to verify that indeed $Q_{s,1}$ is correct (step 6 of Phase-II). This detects incorrect quadruples with high probability. [18] Unfortunately while checking correctness we loose the privacy of $Q_{s,1}$. If dishonest parties are lucky enough they may be able to learn a certain predicate $\mathcal{L}(X_0, X_1, E, Z)$ of their choosing in one of the (verified) correct quadruples. (Below we give an explicit attack where dishonest parties learn the secret decision bit e.)
- In Phase-III we remove any potential leakage as follows. We see again the surviving quadruples of the previous phase as groups or buckets. For each bucket we combine the quadruples obtaining one new quadruple. This ensures that a fixed leakage predicate $\mathcal{L}$ of the resulting quadruple is *not* learnt (step 7 of Phase-III) followed by a local change in the syntax of the new quadruples (step 9); the change is such that if a quadruple Q is leaking predicate $\mathcal{L}'$ the locally swapped quadruple potentially leaks only the fixed predicate $\mathcal{L}$. Therefore combining once more we are able to remove any potential leakage in the output quadruples.
- Right before the parties output something they run protocol Π_{MACCHECK} over all partially opened values, forcing corrupt parties to stick to the inputs chosen in step **Authenticate** during the three phases.

[18] If the first quadruple of some bucket is incorrect the check passes with negligible probability in the size of MAC key vector σ

Protocol $\Pi_{\text{AUROT}}(\ell, m, b)$

It is parametrized by the number of output ROT quadruples ℓ, the length of the MAC key τ, and an additional parameter b, denoting the size of the buckets.

Share OT. This generates $\gamma = \ell \cdot (b^3 + 1)$ random quadruples $(\langle x_s^{(0)} \rangle, \langle x_s^{(1)} \rangle, \langle e_s \rangle, \langle z_s \rangle)$ such that $(x_s^{(0)}, x_s^{(1)}, e_s)$ are randomly distributed and $z_s = x_s^{(e_s)}$ for $s = 1, \dots, \gamma$.

1. Each party P_j samples a random vector $\mathbf{e}^j$, in $\mathbb{F}_2^\gamma$, and τ random vectors $\mathbf{d}_1^j, \dots, \mathbf{d}_\tau^j$ in $\mathbb{F}_2^\gamma$.
2. The parties call $\mathcal{F}_{\text{SSOT}}$ n times. In the ith call, P_i inputs vectors $(\mathbf{e}^i, (\mathbf{d}_s^i)_{s \leq \gamma})$, and P_j inputs $(\mathbf{d}_s^j)_{s \leq \gamma}$. They obtain representations $[e_s^i]_{\mathbf{d}_s}^i$ for $i \leq n$ (e_s^i is the sth bit of $\mathbf{e}^i$), that is, P_i gets $\mathbf{u}_s^i \in \mathbb{F}_2^\tau$, and all the parties get sharings $\langle \mathbf{v}_{s,i} \rangle$, such that $\mathbf{u}_s^i = \mathbf{v}_{s,i} \oplus (e_s^i \cdot \mathbf{d}_s)$. (Here $\mathbf{d}_s \stackrel{\text{def}}{=} \mathbf{d}_s^1 \oplus \dots \oplus \mathbf{d}_s^n$.)
   The parties locally compute $[e_s]_{\mathbf{d}} = [e_s^1]_{\mathbf{d}_s}^1 \oplus \dots \oplus [e_s^n]_{\mathbf{d}_s}^n$. (Here $e_s \stackrel{\text{def}}{=} e_s^1 \oplus \dots \oplus e_s^n$.)
3. At this point the parties have γ sharings $(\langle e_s \rangle, \langle \mathbf{u}_s \rangle, \langle \mathbf{v}_s \rangle, \langle \mathbf{d}_s \rangle)$. They set $\langle \bar{\mathbf{z}}_s \rangle = \langle \mathbf{u}_s \rangle$, $\langle \bar{\mathbf{x}}_s^{(0)} \rangle = \langle \mathbf{v}_s \rangle$, and $\langle \bar{\mathbf{x}}_s^{(1)} \rangle = \langle \mathbf{v}_s \rangle \oplus \langle \mathbf{d}_s \rangle$. Note that $\bar{\mathbf{z}}_s = \bar{\mathbf{x}}_s^{(e_s)}$, for $s \leq \gamma$.
4. Each party takes the first bits of their shares vectors. Thus, for each secret vector $\bar{\mathbf{y}}_s \in \{\bar{\mathbf{x}}_s^{(0)}, \bar{\mathbf{x}}_s^{(1)}, \bar{\mathbf{z}}_s\}$ in $\mathbb{F}_2^\tau$, party P_i sets $y_s^i = \text{LSB}(\bar{\mathbf{y}}_s^i)$. Now the parties have four secret shared vectors $(\mathbf{x}^{(0)}, \mathbf{x}^{(1)}, \mathbf{e}, \mathbf{z})$ in $\mathbb{F}_2^\gamma$, such that $z_s = x_s^{e_s}$, giving γ ROT quadruples.

Authenticate OT. This step produces authentications of the quadruples previously computed.
For every secret shared vector $\mathbf{y} \in \{\mathbf{x}^{(0)}, \mathbf{x}^{(1)}, \mathbf{e}, \mathbf{z}\}$ in $\mathbb{F}_2^\gamma$, the parties do the following:

5. All the parties call $\mathcal{F}_{\text{SSCOT}}$ n times. In the ith call, party P_i inputs $(\mathbf{y}^i, \boldsymbol{\alpha}^i)$ and P_j inputs his share of the MAC key $\boldsymbol{\alpha}^j$ (in $\mathbb{F}_2^\tau$). After each call, the functionality gives $[y_s^i]_{\boldsymbol{\alpha}}^i$ to the parties.
6. Compute $[\![y_s]\!]_{\boldsymbol{\alpha}}$ forming $\bigoplus_{i \in \mathcal{P}} [y_s^i]_{\boldsymbol{\alpha}}^i$, and then subtract $\mathbf{v}_s$ from $\mathbf{u}_s$. (Vectors $\mathbf{u}_s$ and $\mathbf{v}_s$ are embedded in $[y_s]_{\boldsymbol{\alpha}}$)

Sacrifice OT. The parties call subprocedure Π_{BCC} inputting the γ authenticated ROT quadruples (with possibly faulty quadruples), see Figure 30. Then the parties output what the subprocedure outputs.

Figure 29 Creating Authenticated ROT Quadruples in the $(\mathcal{F}_{\text{COMM}}, \mathcal{F}_{\text{SSOT}}, \mathcal{F}_{\text{SSCOT}})$ Hybrid Model

Correctness and Privacy of the Output Quadruples As we said, it is not enough to check the quadruples are correct because in doing so new information may be leaked. For example, the set of corrupt parties can mount the following selective-failure attack: for a target quadruple $Q = (\langle x^{(0)} \rangle, \langle x^{(1)} \rangle, \langle e \rangle, \langle z \rangle)$, generated as specified in step **Share** of Π_{AUROT}, they authenticate the flipped bit $x^{(1)} \oplus 1$ instead of $x^{(1)}$ in step **Authenticate**. If they are lucky enough so that Q is not opened in Phase-I and it ends up being the first of its bucket then Phase-II aborts iff the decision bit e is one; if it is zero the value z corresponds to the first secret bit $x^{(0)}$ and this what the parties check. Thus, if there is no complain when testing for correctness corrupt parties learn the value of the decision bit e, since the quadruples are randomly distributed this happens with probability one half. Combining quadruples after testing correctness prevents honest parties outputting quadruples that are not private.

The following two lemmas prove what we are claiming and they are used in the simulation. The first one shows that the output of Phase-II gives correct quadruples and the second that after combining several quadruples in Phase-III any potential leakage is removed.

Lemma 13 (Correctness). *Let σ the size of the MAC key vector, ℓ the total number of output quadruples in Π_{AUROT}, and b the size of the buckets used in step* **Sacrifice**. *Then if $(b-1)\log_2(\ell) \geq \sigma$ the output of Phase-II of subprotocol Π_{BCC} gives an incorrect quadruple with probability $p \leq 2^{-\sigma}$.*

Proof. Let ℓ be the final number of output quadruples of protocol Π_{AUROT} and $\gamma = \ell \cdot (b^3+1)$ be the number of input quadruples to step **Sacrifice**. In Phase-II the parties are randomly grouping $\ell \cdot b^3$quadruples in buckets of size b perform a check and then taking one quadruple per bucket. Let the ith bucket $\mathcal{B}_i = \{Q_{i,1}, \dots, Q_{i,b}\}$ and denote with $(Q_i)_{i \leq t}$ the quadruples that Phase-II outputs, where $t = \ell \cdot b^2$.

Now, suppose that dishonest parties enter with exactly r incorrect quadruples step **Sacrifice**, we want to upper bound the probability that at least one of the quadruples $(Q_i)_{i \leq t}$ is incorrect. Let $p(r, t, b)$ be this probability and consider the following events.

Subprotocol $\Pi_{\mathrm{BCC}}(\ell, m, b)$

It is parametrized by the number of output ROT quadruples ℓ, the length of the MAC key m, and the size of the buckets b.

Output : ℓ quadruples $Q_s = ([\![x_s^{(0)}]\!], [\![x_s^{(1)}]\!], [\![e_s]\!], [\![z_s]\!])_{s\leq\ell}$ such that

- $z_s = x_s^{(e_s)}$
- $(x_s^{(0)}, x_s^{(1)}, e_s)$ is randomly distributed in $\{0,1\}^3$

Phase-I: Cut-And-Choose

Input: Quadruples generated in steps **Share** and **Authenticate** of Π_{AuROT}. (There are $\ell(b^3+1)$ quadruples)

1. Party P_i samples a seed r^i and asks $\mathcal{F}_{\mathrm{COMM}}$ to broadcast $\tau^i = \mathtt{comm}(r^i)$.
2. Party P_i calls $\mathcal{F}_{\mathrm{COMM}}$ with $\mathtt{open}(\tau^i)$ and all parties obtain r^j for all j. They set $r = r^1 \oplus \cdots \oplus r^n$.
3. Using a $\mathsf{PRF}_r^{\mathbb{F}_2,m}$, parties sample a random vector $\mathbf{w} \in \mathbb{F}_2^m$, with weight ℓ.
   For $c = 0, 1$, let $\mathcal{J}_c \subseteq [\ell(b^3+1)]$ be the set of indices such that $w_j = c$.
4. For each $s \in \mathcal{J}_1$, the sth quadruple is partially opened. If $z_s \neq x_s^{(e_s)}$ the honest parties abort.

Phase-II: Sacrifice

Input: Quadruples indexed with $\mathcal{J}_0$. (There are ℓb^3 quadruples.)

5. Permute the quadruples using a random permutation π on ℓb^3 objects. (The permutation is chosen using again PRF_r.) After, split the quadruples into ℓb^2 buckets $\mathcal{B}_i$ of size b.
6. For each bucket the parties check the first quadruple in the bucket using the other $b-1$ quadruples. Thus, if $\mathcal{B}_s = (Q_{s,1}, \ldots, Q_{s,b})$, they repeat the following for $k = 2, \ldots, b$
   - Partially open $p_{s,k} = x_{s,1}^{(0)} \oplus x_{s,1}^{(1)} \oplus x_{s,k}^{(0)} \oplus x_{s,k}^{(1)}$, and $q_{s,k} = e_{s,1} \oplus e_{s,k}$
   - Compute

$$[\![c_{s,k}]\!] = [\![z_{s,1}]\!] \oplus [\![z_{s,k}]\!] \oplus [\![x_{s,1}^{(0)}]\!] \oplus [\![x_{s,k}^{(0)}]\!] \oplus (p_{s,k} \cdot [\![e_{s,1}]\!]) + q_{s,k} \cdot \left([\![x_{s,k}^{(0)}]\!] \oplus [\![x_{s,k}^{(1)}]\!] \right)$$

   Then partially open $c_{s,k}$, if it is not zero the honest parties abort.

Phase-III: Combine

Input: The first quadruples of each bucket of Phase II.(There are $\ell \cdot b^2$ quadruples.)

7. Split the quadruples into ℓb buckets $\mathcal{B}'_i$ of size b each. (No need to permute again.)
8. For each bucket combine its quadruples. This is done recursively, taking a combined quadruple and combining it with the next quadruple of the bucket. Combining two quadruples Q_1 and Q_2 into a third quadruple Q_3 is done as follows:
   (a) Partially open $f = x_1^{(0)} \oplus x_2^{(0)} \oplus x_1^{(1)} \oplus x_2^{(1)}$
   (b) Set $[\![x_3^{(0)}]\!] = [\![x_1^{(0)}]\!] \oplus [\![x_2^{(0)}]\!]$, $[\![x_3^{(1)}]\!] = [\![x_1^{(0)}]\!] \oplus [\![x_2^{(1)}]\!]$, $[\![e_3]\!] = [\![e_1]\!] \oplus [\![e_2]\!]$, $[\![z_3]\!] = [\![z_1]\!] \oplus [\![z_2]\!] \oplus (f \cdot [\![e_1]\!])$
9. The parties now have ℓb quadruples $Q_s = ([\![x_s^{(0)}]\!], [\![x_s^{(1)}]\!], [\![e_s]\!], [\![z_s]\!])_{s\leq \ell b}$. They swap the components of each Q_s. Thus, the swapped quadruple $\bar{Q}_s$ is defined as follows:
   $[\![\bar{x}_s^{(0)}]\!] = [\![x_s^{(0)}]\!]$, $[\![\bar{x}_s^{(1)}]\!] = [\![x_s^{(0)}]\!] \oplus [\![e_s]\!]$, $[\![\bar{e}_s]\!] = [\![x_s^{(0)}]\!] \oplus [\![x_s^{(1)}]\!]$, $[\![\bar{z}_s]\!] = [\![z_s]\!]$
10. Split the swapped quadruples into ℓ buckets $\mathcal{B}_i''$ of size b, and repeat step 8.

Output: The parties execute the protocol Π_{MACCHECK} to check all partially opened values. If no abort occurs, output the ℓ quadruples obtained after executing step 10.

Figure 30 Bucket Cut-and-Choose Protocol in the $\mathcal{F}_{\mathrm{COMM}}$-Hybrid Model

Event 1 (D_r): Subprotocol Π_{BCC} does not abort in Phase-I provided r quadruples are incorrect.

Event 2 ($E_{r,t,b}$): After randomly bucketing tb quadruples, of which r are incorrect, in buckets of size b there are no buckets containing both incorrect and correct quadruples (i.e Phase-II does not abort).

The check done in step 6 of Phase-II consists in (partially) open the sums $z_i \oplus z_{i,s} \oplus x_i^{e_i} \oplus x_{i,s}^{e_{i,s}}$ and see if they are zero for each $2 \leq s \leq b$ and all buckets $\mathcal{B}_i$. If some Q_i is incorrect the test goes through iff all the other quadruples in the ith bucket are also incorrect. In other words the parties reach the end of Phase-II starting with r incorrect quadruples if and only if the failure event $F_{r,t,b} = E_r \cap D_{r,t,b}$ happens. Thus

$$p(r,t,b) = Pr[F_{r,t,b}] = Pr[D_r] \cdot Pr[E_{r,t,b}] \tag{11}$$

where the last equality follows from the fact that aborting in Phase-I is independent of aborting in Phase-II. We do the following claim

Claim. $Pr[D_r] \le (\frac{b^3}{b^3+1})^r$.

Proof. If D_r is true then in Phase-I the ℓ quadruples that are partially opened are (randomly) sampled from the set of $\gamma - r$ correct ones, so for $c = b^3$ we can write

$$\begin{aligned}
\Pr[D_r] = & \binom{\gamma - r}{\ell} \cdot \binom{\gamma}{\ell}^{-1} = \binom{(c+1)\ell - r}{c\ell - r} \cdot \binom{(c+1)\ell}{c\ell}^{-1} \\
= & \frac{((c+1)\ell - r)\cdots(\ell+1)(c\ell)!}{(c\ell+\ell)\cdots(\ell+1)(c\ell-r)!} \\
= & \frac{1}{c\ell+\ell}\cdots\frac{1}{c\ell+\ell-r+1}\cdot\frac{c\ell+\ell-r}{c\ell+\ell-r}\cdots\frac{\ell+1}{\ell+1}\cdot\frac{(c\ell)!}{(c\ell-r)!} \\
= & \frac{(c\ell)\cdots(c\ell-r+1)}{(c\ell+\ell)\cdots(c\ell+\ell-r+1)}\cdot\frac{(c\ell-r)!}{(c\ell-r)!} \\
\le & \left(\frac{c\ell}{c\ell+\ell}\right)^r = \left(\frac{b^3}{b^3+1}\right)^r.
\end{aligned}$$

□

Observe that if $E_{t,b,r}$ is true we have that $r = kb$ for some $k \in \mathbb{N}$; if not there is at least one mixed bucket. Thus, the second event happens if the random permutation chosen in Phase-II gives k buckets filled with incorrect quadruples. The latter happens with probability

$$Pr[E_{r,t,b}] \le \binom{t}{k} \cdot \binom{tb}{kb}^{-1}. \tag{12}$$

On the other hand if D_r is true we may assume that $r \le tb$; otherwise one incorrect quadruple has not been partially opened properly in Phase-I resulting in abort in the final execution of Π_{MACCheck}. We make a second claim:

Claim. If we see $p(r,t,b)$ as a function in r then $p(b,t,b) \ge p(r,t,b)$ for $r = b, 2b, \ldots, tb$. Thus the probability is maximized in $r = b$.

Intuitively we can see the second claim as follows: for each $r = kb$, with $k \le t$, using equations 11, 12 we have that $p(kb,t,b) \le Pr[D_r] \cdot \binom{t}{k} \cdot \binom{tb}{kb}^{-1}$. The term $\binom{t}{k} \cdot \binom{tb}{kb}^{-1}$ is symmetric with respect to the value $k = t/2$ since

$$\binom{t}{k} \cdot \binom{tb}{kb}^{-1} = \binom{t}{t-k} \cdot \binom{tb}{tb-kb}^{-1},$$

and it strictly decreases for $1 \le k \le t/2$ with the minimum being in $k = t/2$; the term $\left(\frac{b}{b+1}\right)^{k\cdot b}$ is less than 1 and it strictly decreases when k grows. So when we multiply the two terms by the probability of the first event, we have that the final probability $p(kb,t,b)$ for any $k \le t/2$ is bigger than $p(kb,t,b)$ for "symmetric" values $k \ge t/2$. The maximum is therefore for $k = 1$.

With the two claims we bound the probability of passing Phase-I and Phase-II with r incorrect quadruples:

$$p(b,t,b) \le \left(\frac{b^3}{b^3+1}\right)^b \cdot t \cdot \binom{t\cdot b}{b}^{-1} \le \left(\frac{b^3}{b^3+1}\right)^b \cdot t^{(1-b)}$$

$$= 2^{(\log_2(t))(1-b)+b(\log_2(b^3/(b^3+1))}$$
$$\leq 2^{(\log_2(t))(1-b)}$$

where the last inequality follows because $\log_2 b^3/(b^3+1)$ is negative. In particular, not aborting in Phase-I or Phase-II with incorrect quadruples happens with probability upper bounded with $2^{-\sigma}$ provided that we set

$$(b-1)\log_2(\ell) \geq \sigma$$

This concludes the proof of the lemma. □

The next result shows that no information is leaked after executing Phase-III of the **Sacrifice** step.

Lemma 14 (Privacy). *Let σ the size of the MAC key vector, ℓ the total number of output quadruples in Π_{AUROT}, and b the size of the buckets used in step* **Sacrifice**. *Then if $(b-1)\cdot(\log_2(\ell)+1) \geq \sigma$ Phase-III outputs leaky quadruples with probability $q \leq 2^{-\sigma+1}$.*

Proof. In step **Authenticate** of protocol Π_{AUROT} dishonest parties can choose to authenticate faulty quadruple of the form $Q = [\![x^{(0)} \oplus \Delta_0]\!], [\![x^{(1)} \oplus \Delta_1]\!], [\![e \oplus \Delta_e]\!], [\![z \oplus \Delta_z]\!])$ where $(x^{(0)}, x^{(1)}, e, z)$ is private and correct and the offset $(\Delta_0, \Delta_1, \Delta_e, \Delta_z) \in \{0,1\}^4$ is adversarial. That they can only influence in this way follows from the fact that the honest parties are authenticating their own shares, so dishonest behaviour can only expect to offset the secret-shared bits. Using that triplet $(x^{(0)}, x^{(1)}, e)$ is randomly distributed, any offset vector will give a correct quadruple with probability one half (after authenticating). In other words, if subprotocol Π_{BCC} is given as input r faulty quadruples, and D_r is the event that all are correct, then we have $Pr[D_r] = 2^{-r}$. Without loss of generality we assume that Phase-III starts with exactly r leaky quadruples.

Let $q_1(r, \ell b^2, b)$ be the probability that the first combination in step 7 of Π_{BCC} over ℓb^2 quadruples contains at least one bucket with only leaky quadruples. Here the bucketing is inherited from the permutation of Phase-II splitting the surviving quadruples in chunks of size b.

First observe that predicate $\mathcal{L}(X^{(0)}, X^{(1)}, E, Z) = E$ never leaks in the combined quadruples. In Theorem 15 we will show that if $\log(\ell b) \geq \sigma/(b-1) - 1$ then for any value of r we have that $2^{-r} \cdot q_2(r, \ell b, b) \leq 2^{-\sigma}$. Combining quadruples in a given bucket $\mathcal{B}_i$ gives a new quadruple with decision bit $e^i = e^{i,1} \oplus \ldots \oplus e^{i,b}$; since these bits are never revealed the resulting decision bit is uniformly distributed in the joint view of corrupted parties, unless all the quadruples of the bucket are leaky, but this happens with probability negligible in σ.

Second, after expanding out step 6 of Π_{BCC} on quadruples with adversarial offsets [19], one sees that the predicates that dishonest parties potentially learn are E or $X^{(0)} \oplus X^{(1)}$ or $E \oplus X^{(0)} \oplus X^{(1)}$, and their flipped counterparts. In particular predicate $X^{(0)}$ is never learnt by the corrupt parties, neither is predicate E after the first combination. Now, the local swap of step 9 is such that maps

$$x^{(0)} \mapsto \hat{x}^{(0)} = x^{(0)}$$
$$(x^{(1)}, e) \mapsto \hat{x}^{(1)} = x^{(0)} \oplus e$$

Therefore the two secret bits after the swap are randomized via the old bits $x^{(0)}$, e. This ensures that any potential leakage in the swapped quadruple must be in the decision bit $\hat{e}$. Applying again the result of Theorem 15, and assuming that $\log \ell \geq \sigma/(b-1) - 1$ after the second combination we remove all the leakage unless one of the buckets contains only leaky quadruples, which happens with probability $2^{-r} \cdot q_2(r, \ell b, b) \leq 2^{-\sigma}$ for any value of r. Summing up, Phase-III outputs leaky quadruples with probability

$$q \leq 2^{-r} q_1(r, \ell b^2, b) + 2^{-r} q_2(r, \ell b, b) \leq 2^{-\sigma+1}.$$

□

We now state the main result of this section.

[19] One expands the product $z \oplus \Delta_z = x_0 \oplus ((e \oplus \Delta_e) \cdot (x_0 \oplus x_1 \oplus \Delta_0 \oplus \Delta_1))$, and see for fixed $\boldsymbol{\Delta}$ the relation on (x_0, x_1, e, z) that gives the equality

Theorem 10. *Let σ a security parameter, and let additional parameters t, b, ℓ in $\mathbb{N}$ with $b \geq 2$, and $\log \ell \geq \frac{\sigma}{b-1}$. Define $\gamma = \ell \cdot (b^3 + 1)$, then in the $(\mathcal{F}_{\mathrm{COMM}}, \mathcal{F}_{\mathrm{ssOT}}(1, n, \gamma, \sigma), \mathcal{F}_{\mathrm{ssCOT}}(1, n, 4\gamma, \sigma))$ model, and assuming the existence of $\mathsf{PRF}_s^{\mathbb{F},t}(\cdot)$, protocol $\Pi_{\mathrm{AUROT}}(\ell, \sigma, b)$ of Figure 29 implements $\mathcal{F}_{\mathrm{AUROT}}(\ell, \sigma)$ with computational security in σ, against static adversaries corrupting up to $n-1$ parties,*

Proof. Let $\mathcal{Z}$ denote the environment, the description of the ideal adversary $\mathcal{S}$ is as follows. It starts invoking an internal copy of the real adversary $\mathcal{A}$ and setting dummy parties π_i for $i \in 1, \ldots, n$. It then runs an internal execution of Π_{AUROT} between $\mathcal{A}$ and the π_i's, where every incoming communication from $\mathcal{Z}$ is forwarded to $\mathcal{A}$ as if it were coming from $\mathcal{A}$'s environment, and any outgoing communication from $\mathcal{A}$ is forwarded to $\mathcal{Z}$. Then $\mathcal{S}$ proceeds as specified now. (Indices corresponding to the subset of corrupted parties are denoted with A.)

1. $\mathcal{S}$ sets shares of the MAC key $\bar{\boldsymbol{\alpha}}^h$ of honest π_h at random, and activates internal copies of $\mathcal{F}_{\mathrm{ssOT}}$ and $\mathcal{F}_{\mathrm{ssCOT}}$. It runs an internal execution of Π_{AUROT} doing the following:
   - In step 2, if $\mathcal{A}$ gives inconsistent inputs $\mathbf{d}_s^c$ across the n calls to the internal $\mathcal{F}_{\mathrm{ssOT}}$, for corrupted π_c, then $\mathcal{S}$ sets flag badGen to true.
   - In step 5, if $\mathcal{A}$ misbehaves in any way during the $4n$ calls to $\mathcal{F}_{\mathrm{ssCOT}}$ such that dummy $\{\pi_1, \ldots, \pi_n\}$ hold an incorrect authenticated bit, then $\mathcal{S}$ sets flag $\mathsf{badAuth}$ to true. Additionally, if $\mathcal{A}$ authenticates something different to what it was created in the **Share** step, then $\mathcal{S}$ sets badGen to true.
   - In step **Output** it runs Π_{MACCHECK} with $\mathcal{A}$ using the dummy input $\bar{\boldsymbol{\alpha}}^h$ as the share of the MAC key corresponding to honest π_h.
2. After completing the internal execution, if badGen or $\mathsf{badAuth}$ are true, and no abort has occurred $\mathcal{S}$ sends $\mathsf{IdealWorld}$ to $\mathcal{Z}$, and halts.
   Otherwise, if an abort occurred $\mathcal{S}$ externally sends `abort` to $\mathcal{F}_{\mathrm{AUROT}}$ and halts. Else, $\mathcal{S}$ puts in a list every bit $\bar{y} \in \{\bar{x}_0, \bar{x}_1, \bar{e}, \bar{z}\}$, corresponding to the corrupt bit shares of the output quadruples (they where specified by $\mathcal{A}$ as inputs in the call to $\mathcal{F}_{\mathrm{ssCOT}}$), and the corrupt shares $\bar{\boldsymbol{\alpha}}^c$ given as inputs where, say dummy π_1 was authenticating his shares.
   Then it externally sends $(\mathsf{init}, A, (\bar{\boldsymbol{\alpha}}^c)_{c \in A})$ and auROT together with the extracted corrupt bit shares y to $\mathcal{F}_{\mathrm{AUROT}}$, outputs what corrupt π_c outputs and halts.

We now argue indistinguishability showing that for each action $\mathcal{S}$ takes in response to his internal adversary, the environment never manages to distinguish.

Case `abort` . In this case there is no outputs to which $\mathcal{Z}$ can distinguish against, so the simulation is perfect because if the real process aborts so does the ideal process.

Case $\mathsf{badAuth} \wedge \neg$`abort`. Here $\mathcal{S}$ acknowledges himself to $\mathcal{Z}$. He does it because $\mathcal{Z}$ is going to distinguish anyways: if $\mathcal{S}$ would let $\mathcal{F}_{\mathrm{AUROT}}$ output, then in the real process we have bad authentications, and in the ideal process correct ones. Now, in the internal execution of Π_{AUROT}, the adversary managed to pass Π_{MACCHECK} with at least one bad authenticated bit, using Lemma 1 this case happens with probability $p = 2^{-\sigma}$, where σ is the size of the MAC key $\boldsymbol{\alpha}$.

Case Honest Execution.($\neg\mathsf{badGen} \wedge \neg\mathsf{badAuth} \wedge \neg$`abort`) By the correctness of $\mathcal{F}_{\mathrm{ssOT}}$ and $\mathcal{F}_{\mathrm{ssCOT}}$ and the arithmetic on $[\cdot]$ representations, the real process never aborts, and the output quadruples distributed as the quadruples of $\mathcal{F}_{\mathrm{AUROT}}$. Now, the only difference, between the ideal and the real execution of Π_{MACCHECK}, is that in the former $\mathcal{S}$ uses dummy share $\bar{\boldsymbol{\alpha}}^h$ for π_h, and in the latter P_h uses his own share $\boldsymbol{\alpha}^h$ (all the other values corresponding to honest parties are identically distributed); in the real process all the values that honest parties and $\mathcal{A}$ exchange during the check are padded with the MACs of the check quadruples used in **Phase-II** of **Sacrifice**, and these MACs are not part of the public transcript, the ideal transcript could have also be seen in the real process, (with the appropriate choice of randomness for $\mathcal{S}$); in other words, the public transcript is not bind to any value of the real honest shares $\boldsymbol{\alpha}^h$, and hence the environment can not distinguish both processes. The simulation is perfect.

Case $\mathsf{badGen} \wedge \neg\mathsf{badAuth} \wedge \neg\mathtt{abort}$**.** The environment distinguishes iff the output quadruples of the real execution are either incorrect or leaky. Combining Lemma 13 and Lemma 14 this happens with probability upper bounded with $2^{-\sigma+2}$.

In total the simulation fails if the check on bad partial openings passes or there is one bucket filled only with incorrect or leaky quadruples, so we have a failure probability upper bounded with $2^{-\sigma+3}$. This concludes the proof of the theorem. $\square$

6.3 Implementing the Offline Phase

With all the machinery already developed, implementing $\mathcal{F}_{\text{Prep}}$ of Figure 10 is a relatively easy task, see Figure 31. Recall that we have to create authenticated bits to a single party, and authenticated multiplicative triples.

Authenticated bits to a single party We are not interested in producing robust authentications, since we are also using Π_{MACCheck} in the online phase. With this in mind, and observing that an authenticated bit $[\![r]\!]_{\boldsymbol{\alpha}}$ held by P_i is easily derived from $[r]^i_{\boldsymbol{\alpha}}$, the parties can obtain authenticated bits simply querying $\mathcal{F}_{\text{ssCOT}}$. These bits will be used in the online phase to share the inputs.

Authenticated multiplicative triples As we have seen in Section 4.2, to do online multiplications we only need precomputed authenticated triples $[\![a]\!]$, $[\![b]\!]$, $[\![c]\!]$, such that: (1) they are correct, i.e. $c = a \cdot b$, and (2) they are private, i.e. bits (a, b) are randomly distributed in $\mathbb{F}_2$. Recall from Section 6.2 that an authenticated ROT quadruple $[\![x^{(0)}]\!]_{\boldsymbol{\alpha}}, [\![x^{(1)}]\!]_{\boldsymbol{\alpha}}, [\![e]\!]_{\boldsymbol{\alpha}}, [\![z]\!]_{\boldsymbol{\alpha}}$ is such that $(x^{(0)}, x^{(1)}, e)$ is randomly distributed in $\mathbb{F}_2$, and $z = x^{(e)}$. Given access to $\mathcal{F}_{\text{auROT}}$, the parties can (locally) obtain multiplicative triples simply observing that $z = x^{(0)} \oplus \big(e \cdot (x^{(0)} \oplus x^{(1)})\big)$.

Protocol $\Pi_{\text{Prep}}(\ell, \kappa)$

This protocol is parametrized with the number of (authenticated) bits and multiplicative triples ℓ, and the length of the MAC key κ.

Initialize The parties call init command of $\mathcal{F}_{\text{auROT}}(\ell, \kappa)$, obtaining their shares $\boldsymbol{\alpha}^i$ of the MAC key $\boldsymbol{\alpha} = \boldsymbol{\alpha}^1 \oplus \cdots \oplus \boldsymbol{\alpha}^n$. (The vector $\boldsymbol{\alpha}$ is randomly distributed in $\mathbb{F}_2^{\kappa}$.)

Share-to-Party On input (Share, i) from all the parties, they do the following:

1. P_i samples random bit vector $\mathbf{r}$ in $\mathbb{F}_2^{\ell}$.
2. All parties call $\mathcal{F}_{\text{ssCOT}}(1, n, \ell, \kappa)$, where P_i inputs $(\mathbf{r}, \boldsymbol{\alpha}^i)$, and $P_j \neq P_i$ inputs $\boldsymbol{\alpha}^j$. As a result they obtain ℓ representations $[r_s]^i_{\boldsymbol{\alpha}} = (\langle \mathbf{u}_s \rangle^i, \langle \mathbf{v}_s \rangle, \langle \boldsymbol{\alpha} \rangle)$, such that $\mathbf{u}_s = \mathbf{v}_s \oplus r_s \cdot \boldsymbol{\alpha}$. (Here r_s is the sth bit of $\mathbf{r}$)
3. P_i sets his bit shares as $r^i_s = r_s$, and his MAC shares as $\mathbf{m}^i_s = \mathbf{u}_s \oplus \mathbf{v}^i_s$. Party $P_j \neq P_i$ set his bit shares $r^j_s = 0$, and his MAC shares as $\mathbf{m}^j_s = \mathbf{v}^j_s$, thus $(\langle r_s \rangle, \langle \mathbf{m}_s \rangle, \langle \boldsymbol{\alpha} \rangle) = [\![r_s]\!]_{\boldsymbol{\alpha}}$.

MTriples On input $(\mathsf{MTriple})$, the parties call auROT command of $\mathcal{F}_{\text{auROT}}(\ell, \kappa)$ obtaining ℓ ROT quadruples $[\![x^{(0)}_s]\!]_{\boldsymbol{\alpha}}, [\![x^{(1)}_s]\!]_{\boldsymbol{\alpha}}, [\![e_s]\!]_{\boldsymbol{\alpha}}, [\![z_s]\!]_{\boldsymbol{\alpha}}$. The quadruples are such that $z_s = x^{(e_s)}_s$. Then, for $s \leq \ell$ they set (a_s, b_s, c_s) as follows

1. $[\![a_s]\!]_{\boldsymbol{\alpha}} = [\![e_s]\!]_{\boldsymbol{\alpha}}$
2. $[\![b_s]\!]_{\boldsymbol{\alpha}} = [\![x^{(0)}_s]\!]_{\boldsymbol{\alpha}} \oplus [\![x^{(1)}_s]\!]_{\boldsymbol{\alpha}}$
3. $[\![c_s]\!]_{\boldsymbol{\alpha}} = [\![x^{(0)}_s]\!]_{\boldsymbol{\alpha}} \oplus [\![z_s]\!]_{\boldsymbol{\alpha}}$

The parties output authenticated sharing $([\![a_s]\!]_{\boldsymbol{\alpha}}, [\![b_s]\!]_{\boldsymbol{\alpha}}, [\![c_s]\!]_{\boldsymbol{\alpha}})$ for $s \leq \ell$.

Figure 31 Bit Authentication and Multiplicative Triples in the $(\mathcal{F}_{\text{ssCOT}}, \mathcal{F}_{\text{auROT}})$ Model

Theorem 11. *In the $(\mathcal{F}_{ssCOT}(\ell, \sigma), \mathcal{F}_{auROT}(\ell, \sigma)$ hybrid model, protocol Π_{Prep} of Figure 31 implements $\mathcal{F}_{\text{Prep}}$ of Figure 10 with perfect security against static adversaries corrupting up to $n - 1$ parties*

Proof.

The proof is straightforward because there is no communication between the parties. Correctness of the multiplicative triples hold, since $a_s \cdot b_s = e \cdot (x_s^{(0)} \oplus x_s^{(1)})) = z_s \oplus x_s^{(0)} = c_s$. □

6.4 Complexity Analysis

We examine the cost of outputting one ROT quadruple for a total of ℓ outputs in Π_{AUROT}. The cost (per party and output) is in terms of the number of calls to $\mathcal{F}_{\text{ROT}}$ and $\mathcal{F}_{\Delta\text{-ROT}}$, say each party makes $c(\ell)$ calls. This also gives the computational complexity of doing a multiplication in Π_{ONLINE} since from each quadruple we derive one multiplicative triple.

We let the number of calls to $\mathcal{F}_{\text{ROT}}$ and $\mathcal{F}_{\Delta\text{-ROT}}$ needed to generate one authenticated ROT quadruple in Π_{AUROT} be denoted by c_g. The resulting quadruple might be faulty, i.e. either non-private or incorrect; so we pass it through Π_{BCC} that "sacrifice" $s(\ell)$ quadruples, to check that it is really a ROT quadruple. The sacrificed quadruples per output quadruple depends on the total number of outputs ℓ. The number of queries per output quadruple and party is then $c(\ell) = \frac{c_g}{n} \cdot s(\ell)$.

Queries per generation. Creating one quadruple requires $n(n-1)$ queries to $\mathcal{F}_{\text{ROT}}$ and $4n(n-1)$ queries to $\mathcal{F}_{\Delta\text{-ROT}}$. To see the former, in Π_{AUROT} the parties execute step **Share OT** n times to create $(\langle x^{(0)}\rangle, \langle x^{(1)}\rangle, \langle e\rangle, \langle z\rangle)$; per execution one call to $\mathcal{F}_{\text{ssOT}}$ is done, which in turn queries $\mathcal{F}_{\text{ssOT}}$ $n-1$ times; thus $n(n-1)$ calls to $\mathcal{F}_{\text{ROT}}$ in step **Share OT**. For the latter, the above four bits of the quadruple are MAC'ed in step **Authenticate OT**; each authentication calls $\mathcal{F}_{\text{ssCOT}}$ once; and the latter calls $\mathcal{F}_{\Delta\text{-ROT}}$ $n-1$ times. Assuming the cost of $\mathcal{F}_{\text{ROT}}$ and $\mathcal{F}_{\Delta\text{-ROT}}$ are similar, the total number of queries per generated quadruple is $c_g = 5n(n-1)$.

Checked quadruples. In step **Sacrifice OT** the parties call Π_{BCC} with a bucket of size $b = b(\ell)$. The protocol receive as input γ quadruples generated in the previous steps, and outputs $\ell < \gamma$ checked ones. We now analyze how many quadruples are wasted in the process.

- In Phase-I we make sure not all the generated quadruples are faulty opening at random ℓ of them. Thus we keep $\gamma_1 = \gamma - \ell$ unopened quadruples.
- In Phase-II from the remaining γ_1 quadruples, we waste $(1 - \frac{1}{b})\gamma_1$ to check that the others are correct; so we obtain $\gamma_2 = \frac{\gamma_1}{b}$ correct quadruples.
- In Phase-III from the remaining γ_2 we first combine once to remove the leakage in the decision bits of the quadruples, obtaining $\gamma_3 = \frac{\gamma_2}{b}$ combined ones; a second combination is done to remove the leakage in the two secret bits of the just combined ones; so we obtain $\gamma_4 = \frac{\gamma_3}{b}$ twice-combined ones.
- The last γ_4 resulting quadruples are the ℓ outputs of Π_{AUROT}. From the above we have $\ell = \frac{\gamma - \ell}{b^3}$; and therefore the (amortized) number of sacrificed quadruples is $s(\ell) = \frac{1}{\ell}(\gamma - \ell) = (b(\ell))^3 + 1$.

Summing up, to produce ℓ quadruples in Π_{AUROT}, the number of queries to $\mathcal{F}_{\text{ROT}}$ and $\mathcal{F}_{\Delta\text{-ROT}}$ per party and output is given by

$$c(\ell) = 5 \cdot (n-1) \cdot ((b(\ell))^3 + 1).$$

Figure 32 illustrates the overhead for concrete choice of parameters: the size of the buckets and the number of outputs. It is for a failure event happening in Theorem 10 with probability of 2^{-37}, i.e. MACs of length $\sigma = 40$; recall that for this to hold, outputting ℓ quadruples need a bucket of size $b(\ell) \geq \frac{\sigma}{\log \ell} + 1$; hence the size of the bucket increases if less outputs are produced.

Output quadruples ℓ	Bucket size $b(\ell)$	Queries $c(\ell)$
2^{20}	3	$140(n-1)$
2^{14}	4	$325(n-1)$
2^{10}	5	$630(n-1)$

Figure 32 Π_{AUROT} Calls to $\mathcal{F}_{\text{ROT}}$, $\mathcal{F}_{\Delta\text{-ROT}}$ per Party and Output

7 Preprocessing: The Two Party Case

As discussed in Section 2 we can easily pass from $\wr x \int$ sharings to the $[\![x]\!]$ sharings required in the preprocessing functionality $\mathcal{F}_{\text{PREP}}$ from Section 4. The only part of the $\mathcal{F}_{\text{PREP}}$ functionality now required is that of producing multiplication triples in the case of two parties. Thus if we can present a protocol which produces multiple triples in the $\wr x \int$ sharing, we can trivially produce an implementation of the remaining part of $\mathcal{F}_{\text{PREP}}$. Thus it is to this last part of the jigsaw that we devote this section. Note, that we could in the two party case use our methods from Section 6, but the following method of producing multiplication triples is faster both asymptotically and in practice.

We present the protocol in a top-down fashion as follows.

- In Section 7.1 we first show how to create multiplication triples using a *dealer* functionality that provides random bit authentications (aBits), authenticated local ANDs (aANDs) and authenticated OTs (aOTs). The argument for security is that a malicious adversary can never deviate from the protocol without being detected, as this would correspond to forging a MAC. As forging a MAC involves guessing a random string $\Delta \in \{0,1\}^\psi$ this can only happen with negligible probability.
- Then in Section 7.2 we start implementing the dealer functionality by giving a way to produce random aBits. This turns out to be equivalent $\mathcal{F}_{\Delta\text{-ROT}}$ functionality we implemented very efficiently in Section 5.
- In Section 7.3 we extend the dealer functionality with a way of giving random aANDs. We do this in a two step fashion similar inspired by the efficient cut and choose technique used in [NO09]: we first produce a large amount aANDs in a slightly naive way such that up to σ them may be insecure. Then we combine these into a slightly smaller amount of fully maliciously secure aANDs.
- In Section 7.4 we finish the implementation of the the dealer functionality by extending it with a way to give random aOTs. As for aANDs we use an approach similar to the cut an chose technique of [NO09].
- In Section 7.5 we sketch a complexity analysis of the protocol counting the symmetric primitives used in the protocol.

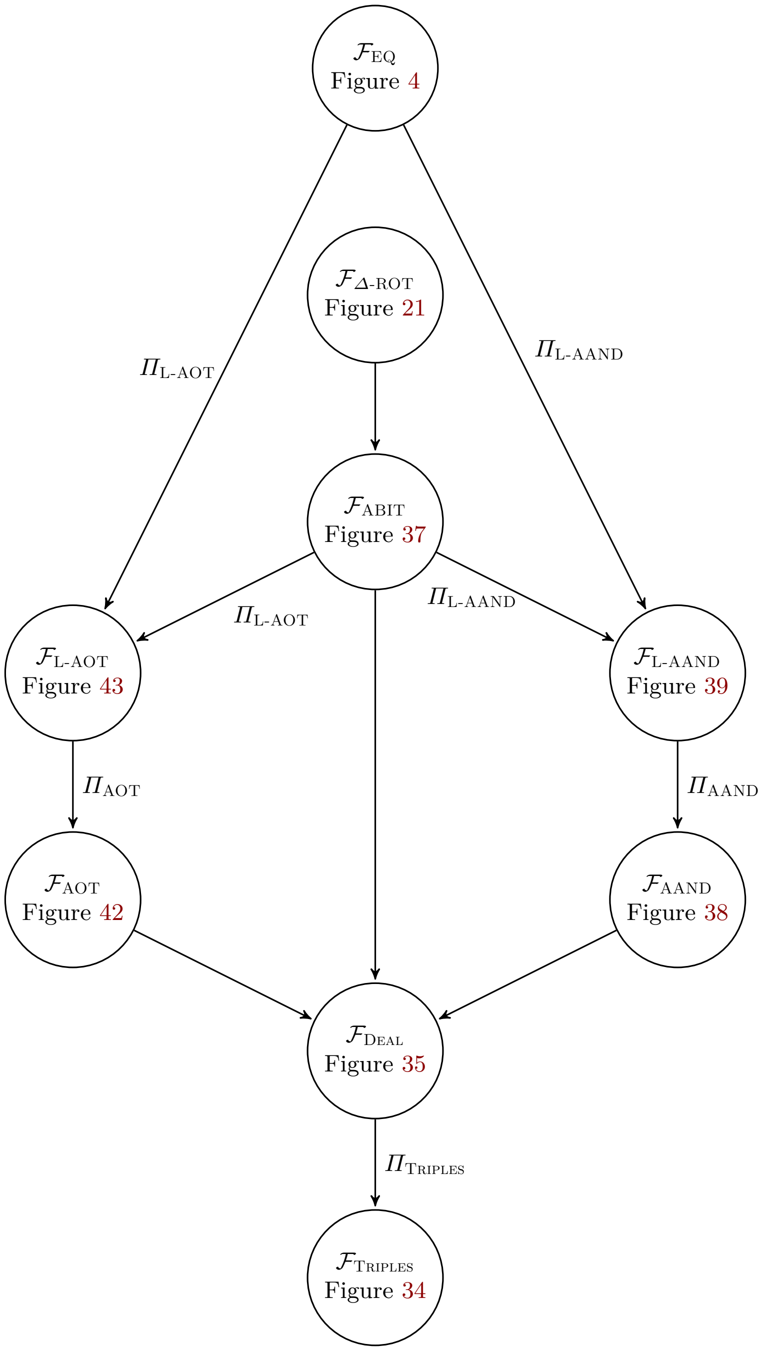
The relationships between the various functionalities of this section is outlined in Figure 33.

7.1 Secure Two-Party Computation in the Dealer Model

We want to implement the functionality $\mathcal{F}_{\text{TRIPLES}}$ for Boolean two-party secure computation as described in Figure 34. It is clear that from $\mathcal{F}_{\text{TRIPLES}}$ we can produce random multiplication triples by calling **Rand** and then **AND** on the functionality. We will implement the $\mathcal{F}_{\text{TRIPLES}}$ functionality in the $\mathcal{F}_{\text{DEAL}}$-hybrid model of Figure 35. The $\mathcal{F}_{\text{DEAL}}$ functionality provides the parties with aBits, aANDs and aOTs, and models the preprocessing phase of our protocol. The protocol implementing $\mathcal{F}_{\text{TRIPLES}}$ in the dealer model is described in Figure 36. The dealer offers random authenticated bits (to P_1 or P_2), random authenticated local AND triples and random authenticated OTs. Those are all the ingredients that we need to build the protocol to generate multiplication triples.

Theorem 12. *The protocol Π_{TRIPLES} in Figure 36 securely implements the functionality $\mathcal{F}_{\text{TRIPLES}}$ in the $\mathcal{F}_{\text{DEAL}}$-hybrid model with security parameter ψ.*

Proof. The simulator can be built in a standard way, incorporating the $\mathcal{F}_{\text{DEAL}}$ functionality and learning all the shares, keys and MACs that the adversary was supposed to use in the protocol.



$\mathcal{F}_{\text{EQ}}$ Figure 4
$\mathcal{F}_{\Delta\text{-ROT}}$ Figure 21
$\Pi_{\text{L-AOT}}$
$\Pi_{\text{L-AAND}}$
$\mathcal{F}_{\text{ABIT}}$ Figure 37
$\Pi_{\text{L-AOT}}$
$\Pi_{\text{L-AAND}}$
$\mathcal{F}_{\text{L-AOT}}$ Figure 43
$\mathcal{F}_{\text{L-AAND}}$ Figure 39
Π_{AOT}
Π_{AAND}
$\mathcal{F}_{\text{AOT}}$ Figure 42
$\mathcal{F}_{\text{AAND}}$ Figure 38
$\mathcal{F}_{\text{Deal}}$ Figure 35
Π_{Triples}
$\mathcal{F}_{\text{Triples}}$ Figure 34

Figure 33 Relationship Between the Functionalities

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

Rand On input ($\texttt{rand}, \mathit{vid}$) from P_1 and P_2, with vid a fresh identifier, the functionality picks $r \in_R \{0,1\}$ and stores (vid, r).

AND On command ($\texttt{AND}, \mathit{vid}_1, \mathit{vid}_2, \mathit{vid}_3$) from both parties (if $\mathit{vid}_1, \mathit{vid}_2$ are defined and vid_3 is fresh), the functionality retrieves (vid_1, x), (vid_2, y) and stores $(\mathit{vid}_3, x \wedge y)$.

At each command the functionality leaks to the environment which command is being executed, and delivers messages only when the environment says so.

Figure 34 The Functionality $\mathcal{F}_{\text{Triples}}$ for Two-party Generation of Multiplication Triples.

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

Initialize
On input ($\texttt{init}$) from P_1 and ($\texttt{init}$) from P_2, the functionality samples $\Delta_1, \Delta_2 \in \{0,1\}^\psi$, stores them and outputs Δ_2 to P_1 and Δ_1 to P_2. If P_1 (resp. P_2) is corrupted, she gets to choose Δ_2 (resp. Δ_1).

Authenticated Bit
On input ($\texttt{aBIT}, P_i$) from P_1 and P_2, the functionality samples a uniformly random $(x, M_x, K_x) \in \{0,1\}^{1+2\psi}$ with $M_x = K_x \oplus x\Delta_i$ where $i, j \in \{1,2\}$ and $i \neq j$. The functionality then outputs (x, M_x) to P_i and K_x to P_j. If P_j is corrupted he gets to choose K_x. If P_i is corrupted she gets to choose (x, M_x), and the functionality sets $K_x = M_x \oplus x\Delta_i$. In the following we will refer to process of sampling and outputting this way as simply *sampling* and *outputting* $[x]^i_{\Delta_i,j}$.

Authenticated local AND
On input ($\texttt{aAND}, P_i$) from P_1 and P_2, the functionality samples and outputs random $[x]^i_{\Delta_i,j}$,$[y]^i_{\Delta_i,j}$ and $[z]^i_{\Delta_i,j}$ with $z = xy$, where $i, j \in \{1,2\}$ and $j \neq i$. As in **Authenticated Bit**, corrupted parties can choose their own randomness.

Authenticated OT
On input ($\texttt{aOT}, P_i, P_j$) from P_1 and P_2, the functionality samples random $[x_0]^i_{\Delta_i,j}$,$[x_1]^i_{\Delta_i,j}$ and $[c]^j_{\Delta_j,i}$ and $[z]^j_{\Delta_j,i}$ with $z = x_c = c(x_0 \oplus x_1) \oplus x_0$ and outputs them. As in **Authenticated Bit**, corrupted parties can choose their own randomness.

Global Key Queries
On input (P_i, Δ) from the adversary the functionality outputs $\texttt{correct}$ if $\Delta = \Delta_i$ and $\texttt{incorrect}$ otherwise.

Figure 35 The Functionality $\mathcal{F}_{\text{Deal}}$ for Dealing Preprocessed Values.

In a little more detail, knowing all outputs from $\mathcal{F}_{\text{Deal}}$ to the corrupted parties allows the simulator to extract inputs used by corrupted parties and input these to the functionality $\mathcal{F}_{\text{Triples}}$ on behalf of the corrupted parties.

Honest parties are run on uniformly random inputs, and when an honest party (P_1 say) is supposed to help open $\wr x \int$, the simulator learns from $\mathcal{F}_{\text{Triples}}$ the value x' that $\wr x \int$ should be opened to. The simulator computes the share x_2 that P_2 holds, which is possible from the outputs of $\mathcal{F}_{\text{Deal}}$ to P_2. From the outputs of $\mathcal{F}_{\text{Deal}}$ to P_2 the simulator also learns the key K_{x_1} that P_2 uses to authenticate x_1. Finally the simulator lets $x_1 = x' \oplus x_2$ and and lets $M_{x_1} = K_{x_1} \oplus x_1 K_{x_1}$ and sends (x_1, M_{x_1}) to P_2.

The simulator aborts if the adversary ever successfully sends some inconsistent bit, i.e., a bit different from the bit it should send according to the protocol and its outputs from $\mathcal{F}_{\text{Deal}}$.

It is easy to see that the protocol is passively secure and that if the adversary never sends such an inconsistent bit, then he is perfectly following the protocol up to input substitution. So, to prove security it is enough to prove that, in the real world protocol, the adversary can only get away with using an inconsistent bit with negligible probability. Note that, to get away with using an inconsistent bit includes the adversary providing a correct MAC for the inconsistent bit. By Corollary 3 (on page 54) we have that using an inconsistent bit is equivalent to guessing the global key Δ of the opposing player. Since all inputs to the adversary are independent of Δ the adversary can guess Δ with at most negligible probability. □

We now show that an adversary getting away with using an inconsistent bit in the protocol in Figure 36 is equivalent to guessing the global key Δ of the opposing player.

The Protocol Π_{Triples}

Initialize

When activated the first time, P_1 and P_2 activate $\mathcal{F}_{\text{Deal}}$ and receive Δ_2 and Δ_1 respectively.

Rand

P_1 and P_2 ask $\mathcal{F}_{\text{Deal}}$ for random authenticated bits $[r_1]^1_{\Delta_1,2}, [r_2]^2_{\Delta_2,1}$ and stores $\wr r \int = [r_1|r_2]$ under *vid*.

AND

P_1 and P_2 retrieve $\wr x \int, \wr y \int$ and compute $\wr z \int = \wr xy \int$ as follows:

1. The parties ask $\mathcal{F}_{\text{Deal}}$ for a random AND triplet $[u]^1_{\Delta_1,2}, [v]^1_{\Delta_1,2}, [w]^1_{\Delta_1,2}$ with $w = uv$.
   P_1 opens $[f]^1_{\Delta_1,2} = [u]^1_{\Delta_1,2} \oplus [x_1]^1_{\Delta_1,2}$ and $[g]^1_{\Delta_1,2} = [v]^1_{\Delta_1,2} \oplus [y_1]^1_{\Delta_1,2}$.
   The parties compute $[x_1y_1]^1_{\Delta_1,2} = f[y_1]^1_{\Delta_1,2} \oplus g[x_1]^1_{\Delta_1,2} \oplus [w]^1_{\Delta_1,2} \oplus fg$.
2. Symmetrically the parties compute $[x_2y_2]^2_{\Delta_2,1}$.
3. The parties ask $\mathcal{F}_{\text{Deal}}$ for a random authenticated OT $[u_0]^1_{\Delta_1,2}, [u_1]^1_{\Delta_1,2}, [c]^2_{\Delta_2,1}, [w]^2_{\Delta_2,1}$ with $w = u_c$.
   They also ask for an authenticated bit $[r_1]^1_{\Delta_1,2}$.
   Now P_2 opens $[d]^2_{\Delta_2,1} = [c]^2_{\Delta_2,1} \oplus [y_2]^2_{\Delta_2,1}$.
   P_1 opens $[f]^1_{\Delta_1,2} = [u_0]^1_{\Delta_1,2} \oplus [u_1]^1_{\Delta_1,2} \oplus [x_1]^1_{\Delta_1,2}$ and $[g]^1_{\Delta_1,2} = [r_1]^1_{\Delta_1,2} \oplus [u_0]^1_{\Delta_1,2} \oplus d[x_1]^1_{\Delta_1,2}$.
   Compute $[s_2]^2_{\Delta_2,1} = [w]^2_{\Delta_2,1} \oplus f[c]^2_{\Delta_2,1} \oplus g$. Note that at this point $[s_2]^2_{\Delta_2,1} = [r_1 \oplus x_1y_2]^2_{\Delta_2,1}$.
4. Symmetrically the parties compute $[s_1]^1_{\Delta_1,2} = [r_2 \oplus x_2y_1]^1_{\Delta_1,2}$.

P_1 and P_2 compute $[z_1]^1_{\Delta_1,2} = [r_1]^1_{\Delta_1,2} \oplus [s_1]^1_{\Delta_1,2} \oplus [x_1y_1]^1_{\Delta_1,2}$ and $[z_2]^2_{\Delta_2,1} = [r_2]^2_{\Delta_2,1} \oplus [s_2]^2_{\Delta_2,1} \oplus [x_2y_2]^2_{\Delta_2,1}$ and let $\wr z \int = [z_1|z_2]$.

Figure 36 Protocol for $\mathcal{F}_{\text{Triples}}$ in the $\mathcal{F}_{\text{Deal}}$-hybrid Model

To formalize this claim we consider the following game $\text{Game}_{I,I}$ played by an attacker A:

Global key: A global key $\Delta \leftarrow \{0,1\}^\psi$ is sampled with some distribution and A might get side information on Δ.

MAC query I: If A outputs a query $(\texttt{mac}, b, l)$, where $b \in \{0,1\}$ and l is a label which A did not use before, sample a fresh local key $K \in_{\text{R}} \{0,1\}^\psi$, give $M = K \oplus b\Delta$ to A and store (l, K, b).

Break query I: If A outputs a query $(\texttt{break}, a_1, l_1, \ldots, a_p, l_p, M')$, where p is some positive integer and values $(l_1, K_1, b_1), \ldots, (l_p, K_p, b_p)$ are stored, then let $K = \oplus_{i=1}^p a_iK_i$ and $b = \oplus_{i=1}^p a_ib_i$. If $M' = K \oplus (1 \oplus b)\Delta$, then A wins the game. This query can be used only once.

We want to prove that if any A can win the game with probability q, then there exist an adversary B which does not use more resources than A and which guesses Δ with probability q without doing any MAC queries. Informally this argues that breaking the scheme is linear equivalent to guessing Δ without seeing any MAC values.

For this purpose, consider the following modified game $\text{Game}_{II,II}$ played by an attacker A:

Global key: *No change.*

MAC query II: If A outputs a query $(\texttt{mac}, b, l, M)$, where $b \in \{0,1\}$ and l is a label which A did not use before and $M \in \{0,1\}^\psi$, let $K = M \oplus b\Delta$ and store (l, K, b).

Break query II: If A outputs a query $(\texttt{break}, \Delta')$ where $\Delta' = \Delta$, then A wins the game. This query can be used only once.

We let $\text{Game}_{II,I}$ be the hybrid game with **MAC query II** and **Break query I**.

We say that an adversary A is no stronger than adversary B if A does not perform more queries than B does and the running time of A is asymptotically linear in the running time of B.

Lemma 15. *For any adversary $A_{I,I}$ for* $\text{Game}_{I,I}$ *there exists an adversary $A_{II,I}$ for* $\text{Game}_{II,I}$ *which is no stronger than $A_{I,I}$ and which wins the game with the same probability as $A_{I,I}$.*

Proof. Given an adversary $A_{I,I}$ for $\text{Game}_{I,I}$, consider the following adversary $A_{II,I}$ for $\text{Game}_{II,I}$. The adversary $A_{II,I}$ passes all side information on Δ to $A_{I,I}$. If $A_{I,I}$ gives the output $(\texttt{mac}, b, l)$, then $A_{II,I}$ samples $M \in_{\text{R}} \{0,1\}^\psi$, outputs $(\texttt{mac}, b, l, M)$ to $\text{Game}_{II,I}$ and returns M to $A_{I,I}$. If $A_{I,I}$ outputs $(\texttt{break}, a_1, l_1, \ldots, a_p,$

l_p, M'), then $A_{II,I}$ outputs $(\texttt{break}, a_1, l_1, \dots, a_p, l_p, M')$ to $\text{Game}_{II,I}$. It is easy to see that $A_{II,I}$ makes the same number of queries as $A_{I,I}$ and has a running time which is linear in that of $A_{I,I}$, and that $A_{II,I}$ wins with the same probability as $A_{I,I}$. Namely, in $\text{Game}_{I,I}$ the value K is uniform and $M = K \oplus b\Delta$. In $\text{Game}_{II,I}$ the value M is uniform and $K = M \oplus b\Delta$. This gives the exact same distribution on (K, M). □

Lemma 16. *For any adversary $A_{II,I}$ for* $\text{Game}_{II,I}$ *there exists an adversary $A_{II,II}$ for* $\text{Game}_{II,II}$ *which is no stronger than $A_{II,I}$ and which wins the game with the same probability as $A_{II,I}$.*

Proof. Given an adversary $A_{II,I}$ for $\text{Game}_{II,I}$, consider the following adversary $A_{II,II}$ for $\text{Game}_{II,II}$. The adversary $A_{II,II}$ passes any side information on Δ to $A_{II,I}$. If $A_{II,I}$ outputs $(\texttt{mac}, b, l, M)$, then $A_{II,II}$ outputs $(\texttt{mac}, b, l, M)$ to $\text{Game}_{II,II}$ and stores (l, M, b). If $A_{II,I}$ outputs $(\texttt{break}, a_1, l_1, \dots, a_p, l_p, M')$, where values $(l_1, M_1, b_1), \dots, (l_p, M_p, b_p)$ are stored, then let $M = \oplus_{i=1}^{p} a_i M_i$ and $b = \oplus_{i=1}^{p} a_i b_i$ and output $(\texttt{break}, M \oplus M')$. For each (l_i, M_i, b_i) let K_i be the corresponding key stored by $\text{Game}_{II,II}$. We have that $M_i = K_i \oplus b_i \cdot \Delta$, so if we let $K = \oplus_{i=1}^{p} a_i K_i$, then $M = K \oplus b\Delta$. Assume that $A_{II,I}$ would win $\text{Game}_{II,I}$, i.e., $M' = K \oplus (1 \oplus b)\Delta$. This implies that $M \oplus M' = K \oplus b\Delta \oplus K \oplus (1 \oplus b)\Delta = \Delta$, which means that $A_{II,II}$ wins $\text{Game}_{II,II}$. □

Consider then the following game Game_{II} played by an attacker A:

Global key: *No change.*
MAC query: *No MAC queries are allowed.*
Break query II: *No change.*

Lemma 17. *For any adversary $A_{II,II}$ for* $\text{Game}_{II,II}$ *there exists an adversary A_{II} for* Game_{II} *which is no stronger than $A_{II,II}$ and which wins the game with the same probability as $A_{II,II}$.*

Proof. Let $A_{II} = A_{II,II}$. The game Game_{II} simply ignores the MAC queries, and it can easily be seen that they have no effect on the winning probability, so the winning probability stays the same. □

Corollary 3. *For any adversary $A_{I,I}$ for* $\text{Game}_{I,I}$ *there exists an adversary A_{II} for* Game_{II} *which is no stronger than $A_{I,I}$ and which wins the game with the same probability as $A_{I,I}$.*

Why the global key queries? The $\mathcal{F}_{\text{DEAL}}$ functionality (Figure 35) allows the adversary to guess the value of the global key, and it informs it if its guess is correct. This is needed for technical reasons: When $\mathcal{F}_{\text{DEAL}}$ is proved UC secure, the environment has access to either $\mathcal{F}_{\text{DEAL}}$ or the protocol implementing $\mathcal{F}_{\text{DEAL}}$. In both cases the environment learns the global keys Δ_1 and Δ_2. In particular, the environment learns Δ_1 even if P_2 is honest. This requires us to prove the sub-protocol for $\mathcal{F}_{\text{DEAL}}$ secure to an adversary knowing Δ_1 even if P_2 is honest: to be able to do this, the simulator needs to recognize Δ_1 if it sees it—hence the global key queries. Note, however, that in the context where we use $\mathcal{F}_{\text{DEAL}}$ (Figure 36), the environment does *not* learn the global key Δ_1 when P_2 is honest: A corrupted P_1 only sees MACs on one bit using the same local key, so all MACs are uniformly random in the view of a corrupted P_1, and P_2 never makes the local keys public.

Amortized MAC checks In the protocol of Figure 36, there is no need to send MACs and check them every time we do an opening. Note as we are not dealing with the $[\![x]\!]$ sharing at this point in the protocol, we need a special MAC check method for the sharings $[a|b]$. In fact, it is straightforward to verify that once all required triples have been produced, or after a fixed number, the protocol is perfectly secure even if the MACs are not checked. Notice then that a keyholder checks a MAC M_x on a bit x by computing $M'_x = K_x \oplus x\Delta$ and comparing M'_x to the M_x which was sent along with x. These equality checks can be deferred and amortized. Initially the MAC holder, e.g. P_1, sets $N = 0^\psi$ and the key holder, e.g. P_2, sets $N' = 0^\psi$. As long as no **Output** command is executed, when P_1 opens x she updates $N \leftarrow H(N, M_x)$ for the MAC M_x she should have sent along with x, and P_2 updates $N' \leftarrow H(N', M'_x)$. Before executing an **Output**, P_1 sends N to P_2 who aborts if $N \neq N'$. Security of this check is easily proved in the random oracle model where the hash function H is modelled as a random oracle. The optimization brings the communication complexity of the protocol down from $O(\psi|C|)$ to $O(|C| + o\psi)$, where o is the number of rounds in which outputs are opened. For a circuit of depth $O(|C|/\psi)$, the communication is $O(|C|)$.

Implementing $\mathcal{F}_{\textbf{Deal}}$ In the following sections we show how to implement $\mathcal{F}_{\text{DEAL}}$. In Section 7.2 we describe how to implement a limited version of $\mathcal{F}_{\text{DEAL}}$ that only provides the **Initialize** and **Authenticated Bit** parts of the functionality. I.e, a functionality that only deals random authenticated bits, aBits. This limited dealer functionality can then be extended seperately with the remaining parts. In Section 7.4 we show how to extend the limited $\mathcal{F}_{\text{DEAL}}$ functionality with the **Authenticated OT** part by showing how to implement many aOTs using many aBits. In Section 7.3 we then show how to extend with the **Authenticated local AND** part, by showing how to implement many aANDs from many aBits.

We describe the extensions separately, and to simplify the description we will describe each extension as its own functionality. We name the functionalities that provides aBits, aOTs and aANDs $\mathcal{F}_{\text{ABIT}}$, $\mathcal{F}_{\text{AOT}}$ and $\mathcal{F}_{\text{AAND}}$ respectively. These functionalities will all provide large *batches* of their respective primitive, i.e. each of the functionalities take a parameter ℓ and produces ℓ aBits, aOTs or aANDs as one batch. To fully implement $\mathcal{F}_{\text{DEAL}}$ we run each of the underlying functionalities when initialized, and then draw from these batches to implement each part of $\mathcal{F}_{\text{DEAL}}$. We note that this imposes a restriction on how many times we can use each of $\mathcal{F}_{\text{DEAL}}$'s commands. However, this is not a problem as long as we have an upper bound on the number of aBits, aANDs and aOTs we will need. Such an upper bound, however, should be easy to compute if only the circuit to be evaluated is known when setting up $\mathcal{F}_{\text{DEAL}}$.

7.2 Bit Authentication

We describe how to implement the **Initialize** and **Authenticated Bit** commands of the $\mathcal{F}_{\text{DEAL}}$ functionality, i.e. we show how to provide oblivious authentication of random bits (aBits). We call the functionality providing aBits $\mathcal{F}_{\text{ABIT}}$. As the command is completely symmetric for P_1 and P_2 we will only describe how to provide authenticated bits to P_1. As described above an aBit outputs to P_1 a bit $x \in_{\text{R}} \{0,1\}$ and MAC $M_x \in_{\text{R}} \{0,1\}^{\psi}$ and to P_2 a key $K_x \in_{\text{R}} \{0,1\}^{\psi}$, so that $M_x = K_x \oplus x\Delta_1$. Here $\Delta_1 \in_{\text{R}} \{0,1\}^{\psi}$ output to P_2 in the initialization, is the global key used to authenticate P_1's bits in each aBit.

To implement the functionality $\mathcal{F}_{\text{ABIT}}$ providing aBits we simply notice that aBits corresponds exactly to the Δ-ROTs we implemented in Section 5. Namely a Δ-ROT is a random OT that outputs to P_1 a choice bit b and a message N_b and to P_2 a message N_0 so that $N_b = N_0 \oplus b\Delta$ for a global value Δ known by P_2. Thus if we let $x = b$, $M_x = N_b$, $K_x = N_0$ and $\Delta_1 = \Delta$ we have exactly the aBit described above.

The Functionality $\mathcal{F}_{\text{ABIT}}(\ell, \psi)$

Honest Parties

On input `start` from both P_1 and P_2 the functionality does the following.

1. The functionality samples $\Delta_1 \in_{\text{R}} \{0,1\}^{\psi}$ and outputs it to P_2.
2. For all $i \in [\ell]$ the functionality samples $x_i \in_{\text{R}} \{0,1\}$ and $K_{x_i} \in_{\text{R}} \{0,1\}^{\psi}$ and lets $M_{x_i} = K_{x_i} \oplus x_i\Delta_1$.
3. The functionality outputs $([x_i]_1 = (x_i, M_{x_i}, K_{x_i}))_{i\in[\ell]}$, i.e. for each $i \in [\ell]$ it outputs (M_{x_i}, x_i) to P_1 and K_{x_i} to P_2.

Corrupt Parties

1. If P_1 is corrupted, the functionality waits to give output till it receives the message $(\hat{M}_{\hat{x}_i}, \hat{x}_i)_{i\in[\ell]}$ from P_1, where $\hat{M}_{\hat{x}_i} \in \{0,1\}^{\psi}$ and $\hat{x}_i \in \{0,1\}$. The functionality then sets $x_i = \hat{x}_i$, $M_{x_i} = \hat{M}_{\hat{x}_i}$ and $K_{x_i} = M_{x_i} \oplus x_i\Delta_1$ and outputs as described above (with Δ_1 sampled as above).
2. If P_2 is corrupted, the functionality waits to give output till it receives the message $(\hat{\Delta}_1, (\hat{K}_{x_i})_{i\in[\ell]})$ from P_1, where $\hat{\Delta}_1, \hat{K}_{x_i} \in \{0,1\}^{\psi}$. The functionality then sets $\Delta_1 = \hat{\Delta}_1$, $K_{x_i} = \hat{K}_{x_i}$ and $M_{x_i} = K_{x_i} \oplus x_i\Delta_1$ and outputs as described above (with x_i sampled as above).

Global Key Queries

If the adversary inputs $(\texttt{guess}, \hat{\Delta})$ to the functionality, the functionality outputs back `correct` if $\hat{\Delta} = \Delta_1$ and `incorrect` otherwise.

Figure 37 The $\mathcal{F}_{\text{ABIT}}(\ell, \psi)$ Functionality

To be compatible with notation used in this section in Figure 37 we rephrase $\mathcal{F}_{\Delta\text{-ROT}}$ from Section 5, as the functionality $\mathcal{F}_{\text{ABIT}}$ providing ℓ aBits with keys of length ψ. There is only one significant difference between the $\mathcal{F}_{\Delta\text{-ROT}}$ and $\mathcal{F}_{\text{ABIT}}$ functionalities, namely the global key queries, which were not present for the $\mathcal{F}_{\Delta\text{-ROT}}$ functionality. However, since we can clearly use the functionality *without* the global key queries to implement the functionality *with*, we can also use the protocol for $\mathcal{F}_{\Delta\text{-ROT}}$ from Section 5 to implement $\mathcal{F}_{\text{ABIT}}$. We get the following theorem simply restating Corollary 2 of Section 5.

Theorem 13. *Let ψ denote the security parameter and let $\ell = \text{poly}(\psi)$. The functionality $\mathcal{F}_{\text{ABIT}}(\ell, \psi)$ can be reduced to $(\mathcal{F}_{\text{OT}}(\frac{\mathbf{110}}{\mathbf{6}}\psi, \psi), \mathcal{F}_{\text{EQ}}(\psi))$. The communication is $O(\psi\ell + \psi^2)$ and the work is $O(\psi^2\ell)$.*

Now given the $\mathcal{F}_{\text{ABIT}}$ functionality, we can implement the limited $\mathcal{F}_{\text{DEAL}}$ functionality that only provides the **Initialize** and **Authenticated Bit** parts using just two $\mathcal{F}_{\text{ABIT}}$'s. One to provide authenticated bits to each party.

7.3 Authenticated local AND

In this section we show how to extend the limited $\mathcal{F}_{\text{DEAL}}$ functionality to add the **Autenticated local AND** part. Namely, we show how to provide authenticated local AND triples (aAND) of the form $[x]^i_{\Delta_i,j}$,$[y]^i_{\Delta_i,j}$ and $[z]^i_{\Delta_i,j}$ with $z = xy$. As the functionality for P_2 is symmetric, we only present how to provide aANDs to P_1.

The Functionality $\mathcal{F}_{\text{AAND}}(\ell)$

Honest Parties
On input `start` from both parties the functionality samples and outputs uniformly random authenticated bits $([x_i]^1_{\Delta_1,2}, [y_i]^1_{\Delta_1,2}, [z_i]^1_{\Delta_1,2})_{i\in[\ell]}$ under the condition that $z_i = x_i y_i$ for all $i \in [\ell]$.

Corrupt Parties
If P_i is corrupt he gets to choose all of his random values (including keys and macs).

Global Key Queries
On input (P_i, Δ) from the adversary the functionality outputs `correct` if $\Delta = \Delta_i$ and `incorrect` otherwise.

Figure 38 The Functionality $\mathcal{F}_{\text{AAND}}(\ell)$ for ℓ Authenticated Local ANDs.

More formally we will show how to implement the functionality $\mathcal{F}_{\text{AAND}}$ in Figure 38. Note that, strictly speaking, the $\mathcal{F}_{\text{AAND}}$ functionality should also sample Δ_1 to make sense. However, we leave this out as we view $\mathcal{F}_{\text{AAND}}$ as part of the $\mathcal{F}_{\text{DEAL}}$ functionality, where Δ_1 is sampled in the **Initialize** part. This is to emphasize that in $\mathcal{F}_{\text{DEAL}}$ both the aBit's and aAND's will be using the same values of Δ_1 and Δ_2. We achieve this by using the same underlying $\mathcal{F}_{\text{ABIT}}$ functionality to implement all parts of $\mathcal{F}_{\text{DEAL}}$.

The Functionality $\mathcal{F}_{\text{L-AAND}}(\ell)$

Honest Parties
On input `start` from both parties the functionality samples and outputs uniformly random authenticated bits $([x_i]^1_{\Delta_1,2}, [y_i]^1_{\Delta_1,2}, [z_i]^1_{\Delta_1,2})_{i\in[\ell]}$ under the condition that $z_i = x_i y_i$ for all $i \in [\ell]$.

Corrupt Parties

1. If P_i is corrupt he gets to choose all of his random values (including keys and macs).
2. Additionally, on input (i, g_i) from corrupt P_2, if $x_i \neq g_i$ or if P_1 has already received output, the functionality outputs `incorrect` and terminates before giving outputs. Otherwise the functionality outputs `correct`.

Global Key Queries
On input (P_i, Δ) from the adversary the functionality outputs `correct` if $\Delta = \Delta_i$ and `incorrect` otherwise.

Figure 39 The Functionality Leaky $\mathcal{F}_{\text{L-AAND}}(\ell)$ for ℓ Leaky Authenticated Local ANDs.

To construct $\mathcal{F}_{\text{AAND}}$ we first construct a leaky version we call $\mathcal{F}_{\text{L-AAND}}$, described in Figure 39. The $\mathcal{F}_{\text{L-AAND}}$ functionality is leaky in the sense that P_2 may learn some of the values x_i: a corrupted P_2 is allowed to try to guess the x_i bits, but if the guess is wrong the functionality aborts revealing that P_2 cheated. This means that if the functionality does not abort, with very high probability P_2 only tried to guess a few bits.

The intuition behind the protocol for $\mathcal{F}_{\text{L-AAND}}$, described in Figure 40, is to let P_1 compute the AND locally and then authenticate the result. P_1 and P_2 then perform some computation on the keys and MACs, in a way so that P_1 will be able to guess P_2's result only if she behaved honestly during the protocol: P_1 behaved honestly (sent $d = z \oplus r$) iff she knows $W_0 = (K_x || K_z)$ or $W_1 = (K_x \oplus \Delta_1 || K_y \oplus K_z)$. In fact, she knows W_x. As an example, if $x = 0$ and P_1 is honest, then $z = 0$, so she knows $M_x = K_x$ and $M_z = K_z$. Had she cheated, she would know $M_z = K_z \oplus \Delta_1$ instead of K_z. P_2 checks that P_1 knows W_0 or W_1 by sending her $H(W_0) \oplus H(W_1)$ and ask her to return $H(W_0)$. This, however, allows P_2 to send $H(W_0) \oplus H(W_1) \oplus E$ for an error term $E \neq 0^\psi$. The returned value would be $H(W_0) \oplus xE$. To prevent this attack, we use the $\mathcal{F}_{\text{EQ}}$ functionality to compare the values instead. If P_2 uses $E \neq 0^\psi$, he must now guess x to pass the protocol. However, P_2 still may use this technique to guess a few x bits.

The Protocol $\Pi_{\text{L-AAND}}$

The protocol runs ℓ times in parallel. Here described for a single leaky authenticated local AND:
1. P_1 and P_2 ask the dealer for $[x]^1_{\Delta_1,2}, [y]^1_{\Delta_1,2}, [r]^1_{\Delta_1,2}$.
2. P_1 computes $z = xy$ and sends $d = z \oplus r$ to P_2.
3. The parties compute $[z]^1_{\Delta_1,2} = [r]^1_{\Delta_1,2} \oplus d$.
4. P_2 sends $U = H(K_x || K_z) \oplus H(K_x \oplus \Delta_1 || K_y \oplus K_z)$ to P_1.
5. If $x = 0$, then P_1 lets $V = H(M_x || M_z)$. If $x = 1$, then P_1 lets $V = U \oplus H(M_x || M_y \oplus M_z)$.
6. P_1 and P_2 call the $\mathcal{F}_{\text{EQ}}$ functionality, with inputs V and $H(K_x || K_z)$ respectively. If $\mathcal{F}_{\text{EQ}}$ outputs `unequal` they abort the protocol, otherwise they proceed. All the ℓ calls to $\mathcal{F}_{\text{EQ}}$ are handled using a single call to $\mathcal{F}_{\text{EQ}}(\ell\psi)$.
7. If the strings are equal, the parties output $[x]^1_{\Delta_1,2}, [y]^1_{\Delta_1,2}, [z]^1_{\Delta_1,2}$.

Figure 40 Protocol for Leaky Authenticated Local AND

The protocol in Figure 40 uses a limited dealer that only gives random authenticated bits, to get 3ℓ authenticated bits to P_1 (3 per provided aAND). As shown above this dealer can be implemented using $\mathcal{F}_{\text{ABIT}}(3\ell, \psi)$, and thus we prove the follwing.

Theorem 14. *The protocol in Figure 40 securely implements $\mathcal{F}_{\text{L-AAND}}(\ell)$ in the $(\mathcal{F}_{\text{ABIT}}(3\ell, \psi), \mathcal{F}_{\text{EQ}}(\ell\psi))$-hybrid model.*

The simulator answers global key queries to the dealer by doing the identical global key queries on the $\mathcal{F}_{\text{L-AAND}}(\ell)$ functionality and returning the reply. This gives a perfect simulation of these queries, and we ignore them below.

Notice that for honest P_1 and P_2 correctness of the protocol follows immediately from correctness of the $\mathcal{F}_{\text{ABIT}}$ functionality. Thus we prove Theorem 14 by proving security separately against corrupted P_1 and P_2 respectively. We do this in Lemma 18 and 19.

Lemma 18. *The protocol in Figure 40 securely implements the $\mathcal{F}_{\text{L-AAND}}$ functionality against corrupted P_1.*

Proof. We first focus on the simulation of the protocol before outputs are given to the environment. Notice that before outputs are given to the environment, the global key Δ_1 is uniformly random to the environment, as long as P_2 is honest.

We consider the case of a corrupt sender P_1 running the above protocol against a simulator $\mathcal{S}$ for honest P_2.

1. $\mathcal{S}$ receives P_1's input $(M_x, x), (M_y, y), (M_r, r)$ to the dealer, and the bit $d \in_{\text{R}} \{0,1\}$.

2. $\mathcal{S}$ samples a random $U \in_R \{0,1\}^{2\psi}$ and sends it to P_1.
3. $\mathcal{S}$ reads P_1's input to the $\mathcal{F}_{\text{EQ}}$ functionality, $\overline{V}$. If

$$\overline{V} \neq (1-x)H(M_x, M_z) \oplus x(U \oplus H(M_x, M_y \oplus M_z))$$

   or

$$d \oplus r \neq xy \ ,$$

   $\mathcal{S}$ aborts the protocol. Otherwise, it inputs $(x, y, z, M_x, M_y, M_z = M_r)$ to the $\mathcal{F}_{\text{L-AAND}}$ functionality.

The first difference between the real protocol and the simulation is that

$$U = H(K_x, K_z) \oplus H(K_x \oplus \Delta_1, K_y \oplus K_z)$$

in the real protocol and U is uniformly random in the simulation. Since H modeled as a random oracle, this is perfectly indistinguishable to the adversary until it queries on both (K_x, K_z) and $(K_x \oplus \Delta_1, K_y \oplus K_z)$. Since Δ_1 is uniformly random to the environment and the adversary during the protocol, this will happen with negligible probability during the protocol. We later return to how we simulate after outputs are given to the environment.

The other difference between the protocol and the simulation is that the simulation always aborts if $z \neq xy$. Assume now that P_1 manages, in the real protocol, to make the protocol continue with $z = xy \oplus 1$. If $x = 0$, this means that P_1 queried H on

$$(K_x, K_z) = (M_x, M_z \oplus \Delta_1).$$

Since $\mathcal{S}$ knows the outputs of P_1, which include M_z, and sees the input $M_z \oplus \Delta_1$ to H, if P_1 queries the oracle on $(K_x, K_z) = (M_x, M_z \oplus \Delta_1)$, $\mathcal{S}$ can compute Δ_1. If $x = 1$ then P_1 must have queried the oracle on

$$(K_x \oplus \Delta_1, K_y \oplus K_z) = (M_x, M_y \oplus M_z \oplus \Delta_1) \ ,$$

which again would allow $\mathcal{S}$ to compute Δ_1. Therefore, in both cases we can use such an P_1 to compute the global key Δ_1 and, given that all of P_1's inputs are independent of Δ_1 during the protocol, this happens only with negligible probability.

Consider now the case after the environment is given outputs. These outputs include Δ_1. It might seem that there is nothing more to simulate after outputs are given, but recall that H is a random oracle simulated by $\mathcal{S}$ and that the environment might keep querying H. Our concern is that U is uniformly random in the simulation and

$$U = H(K_x, K_z) \oplus H(K_x \oplus \Delta_1, K_y \oplus K_z)$$

in the real protocol. We handle this as follows. Each time the environment queries H on an input of the form $(Q_1, Q_2) \in \{0,1\}^{2\psi}$, go over all previous queries (Q_3, Q_4) of this form and let $\Delta = Q_1 \oplus Q_3$. Then do a global key query to the dealer to determine if $\Delta = \Delta_1$. If $\mathcal{S}$ learns Δ_1 this way, she proceeds as follows. Note that since P_1 is corrupted, $\mathcal{S}$ knows all outputs to P_1, i.e., $\mathcal{S}$ knows all MACs M and all bits b. If $b = 0$, then $\mathcal{S}$ also knows the key, as $K = M$ when $b = 0$. If $b = 1$, $\mathcal{S}$ computes the key as $K = M \oplus \Delta_1$. So, when $\mathcal{S}$ learns Δ_1, she at the same time learns all keys. Then for each U she simply programs the RO such that

$$U = H(K_x, K_z) \oplus H(K_x \oplus \Delta_1, K_y \oplus K_z) \ .$$

This is possible as $\mathcal{S}$ learns Δ_1 no later than when the environment queries on two pairs of inputs of the form $(Q_1, Q_2) = (K_x, K_z)$ and $(Q_3, Q_4) = (K_x \oplus \Delta_1, K_y \oplus K_z)$. So, when $\mathcal{S}$ learns Δ_1, either $H(K_x, K_z)$ or $H(K_x \oplus \Delta_1, K_y \oplus K_z)$ is still undefined. If it is $H(K_x, K_z)$, say, which is undefined, $\mathcal{S}$ simply programs the RO so that

$$H(K_x, K_z) = U \oplus H(K_x \oplus \Delta_1, K_y \oplus K_z) \ .$$

□

Lemma 19. *The protocol described in Figure 40 securely implements the $\mathcal{F}_{\text{L-AAND}}$ functionality against corrupted P_2.*

Proof. We consider the case of a corrupt P_2 running the above protocol against a simulator $\mathcal{S}$. The simulation runs as follows.

1. The simulation starts by $\mathcal{S}$ getting P_2's input to the dealer K_x, K_y, K_r and Δ_1.
2. The simulator samples a random $d \in_R \{0,1\}$, sends it to P_2 and computes $K_z = K_r \oplus d\Delta_1$.
3. $\mathcal{S}$ receives $\overline{U}$ from P_2, and reads $\overline{V}$, P_2's input to the equality functionality.
4. If
$$\overline{U} = H(K_x, K_z) \oplus H(K_x \oplus \Delta_1, K_y \oplus K_z)$$
and
$$\overline{V} = H(K_x, K_z) \ ,$$
$\mathcal{S}$ inputs (K_x, K_y, K_z) to the $\mathcal{F}_{\text{L-AAND}}$ functionality and completes the protocol (in this case P_2 is behaving honestly).
5. Otherwise, if
$$\overline{U} \neq H(K_x, K_z) \oplus H(K_x \oplus \Delta_1, K_y \oplus K_z)$$
and
$$\overline{V} = H(K_x, K_z)$$
or
$$\overline{V} = \overline{U} \oplus H(K_x \oplus \Delta_1, K_z \oplus K_z) \ ,$$
$\mathcal{S}$ inputs $g = 0$ or $g = 1$ resp. to the $\mathcal{F}_{\text{L-AAND}}$ functionality as a guess for the bit x (i.e., say this is the i'th aAND, $\mathcal{S}$ inputs (g, i)). If the functionality outputs `incorrect`, $\mathcal{S}$ outputs `incorrect` and aborts, and otherwise outputs `correct` and completes the protocol.

The simulation is perfect: the view of P_2 consists only of the bit d, and whether or not P_1 aborts. Here d is uniformly distributed both in the real world and in the simulation, and the protocol aborts based on the same event in the real and in the simulated world. □

Combining Lemma 18 and 19, we get the proof of Theorem 14.

We now handle a few guessed x bits by random bucketing and a straight-forward combiner. In doing this efficiently, it is central that the protocol was constructed such that only x could leak. Had P_2 been able to get information on both x and y we would have had to do the amplification twice.

We start by producing $b\ell$ leaky aANDs. Then we randomly distribute the $b\ell$ leaky aANDs into ℓ buckets of size b. Finally we combine the leaky aANDs in each bucket into one aAND which is non-leaky if at least one leaky aAND in the bucket was not leaky. The protocol is described in Figure 41. We prove the following theorem.

Theorem 15. *Let $\mathcal{F}_{\text{AAND}}(\ell)$ denote the functionality providing ℓ aANDs as in $\mathcal{F}_{\text{DEAL}}$. For $b \geq \frac{\sigma}{1+\log_2(\ell)} + 1$, the protocol in Figure 41 securely implements $\mathcal{F}_{\text{AAND}}(\ell)$ in the $\mathcal{F}_{\text{L-AAND}}(b\ell)$-hybrid model with statistical security parameter σ.*

Proof. The simulator answers global key queries to leaky $\mathcal{F}_{\text{L-AAND}}(b\ell)$ by doing the identical global key queries on the ideal functionality $\mathcal{F}_{\text{AAND}}(\ell)$ and returning the reply. This gives a perfect simulation of these queries, and we ignore them below.

It is easy to check that the protocol is correct and secure if both parties are honest or if P_1 is corrupted.

What remains is to show that, even if P_2 is corrupted and tries to guess some x's from the leaky $\mathcal{F}_{\text{L-AAND}}$ functionality, the overall protocol is secure.

We argue this in two steps. We first argue that the probability that P_2 learns the x-bit for all triples in the same bucket is negligible. We then argue that when all buckets contain at least one triple for which x is unknown to P_2, then the protocol can be simulated given leaky $\mathcal{F}_{\text{L-AAND}}(b\ell)$.

The Protocol Π_{AAND}

The protocol is parametrized by positive integers b and ℓ.

1. P_1 and P_2 call $\mathcal{F}_{\text{L-AAND}}(\ell')$ with $\ell' = b\ell$. If the call to $\mathcal{F}_{\text{L-AAND}}$ aborts, this protocol aborts. Otherwise, let $\{[x_i]^1_{\Delta_1,2}, [y_i]^1_{\Delta_1,2}, [z_i]^1_{\Delta_1,2}\}_{i=1}^{\ell'}$ be the outputs.
2. P_1 picks a b-wise independent permutation π on $\{1, \dots, \ell'\}$ and sends it to P_2. For $j = 0, \dots, \ell - 1$, the b triples
$$\{[x_{\pi(i)}]^1_{\Delta_1,2}, [y_{\pi(i)}]^1_{\Delta_1,2}, [z_{\pi(i)}]^1_{\Delta_1,2}\}_{i=jb+1}^{jb+b}$$
are defined to be in the j'th bucket.
3. P_1 and P_2 combine b triples in each bucket. Here we describe how to combine two triples: Let be them $[x^1]^1_{\Delta_1,2}, [y^1]^1_{\Delta_1,2}, [z^1]^1_{\Delta_1,2}$ and $[x^2]^1_{\Delta_1,2}, [y^2]^1_{\Delta_1,2}, [z^2]^1_{\Delta_1,2}$. Let the result be $[x]^1_{\Delta_1,2}, [y]^1_{\Delta_1,2}, [z]^1_{\Delta_1,2}$.
   (a) P_1 opens $d = y^1 \oplus y^2$.
   (b) P_1 and P_2 compute
$$[x]^1_{\Delta_1,2} = [x^1]^1_{\Delta_1,2} \oplus [x^2]^1_{\Delta_1,2} \quad , \quad [y]^1_{\Delta_1,2} = [y^1]^1_{\Delta_1,2} \ ,$$
   and
$$[z]^1_{\Delta_1,2} = [z^1]^1_{\Delta_1,2} \oplus [z^2]^1_{\Delta_1,2} \oplus d[x^2]^1_{\Delta_1,2} \ .$$
   To combine all b triples in a bucket, just iterate by taking the result and combine it with the next element in the bucket.

Figure 41 From Leaky Authenticated Local ANDs to Authenticated Local ANDs

Call each of the triples a *ball* and call a ball *leaky* if P_2 learned the x bit of the ball in the call to the leaky $\mathcal{F}_{\text{L-AAND}}(\ell')$ functionality. Let γ denote the number of leaky balls.

For b of the leaky balls to end up in the same bucket, there must be a subset S of balls with $|S| = b$ consisting of only leaky balls and a bucket i such that all the balls in S end up in i.

We first fix S and i and compute the probability that all balls in S end up in i. The probability that the first ball ends up in i is $\frac{b}{b\ell}$. The probability that the second balls ends up in i given that the first ball is in i is $\frac{b-1}{b\ell-1}$, and so on. We get a probability of
$$\frac{b}{b\ell} \cdot \frac{b-1}{b\ell-1} \cdots \frac{1}{b\ell-b+1} = \binom{b\ell}{b}^{-1}$$
that S ends up in i.

There are $\binom{\gamma}{b}$ subsets S of size b consisting of only leaky balls and there are ℓ buckets, so by a union bound the probability that any bucket is filled by leaky balls is upper bounded by
$$\binom{\gamma}{b}\ell\binom{b\ell}{b}^{-1} \ .$$
This is assuming that there are exactly γ leaky balls. Note then that the probability of the protocol not aborting when there are γ leaky balls is $2^{-\gamma}$. Namely, for each bit x that P_2 tries to guess, he is caught with probability $\frac{1}{2}$. So, the probability that P_2 undetected can introduce γ leaky balls and have them end up in the same bucket is upper bounded by
$$\alpha(\gamma, \ell, b) = 2^{-\gamma}\binom{\gamma}{b}\ell\binom{b\ell}{b}^{-1} \ .$$
It is easy to see that
$$\frac{\alpha(\gamma+1, \ell, b)}{\alpha(\gamma, \ell, b)} = \frac{\gamma+1}{2(\gamma+1-b)}.$$
So, $\alpha(\gamma+1, \ell, b)/\alpha(\gamma, \ell, b) > 1$ iff $\gamma < 2b-1$, hence $\alpha(\gamma, \ell, b)$ is maximized in γ at $\gamma = 2b - 1$. If we let $\alpha'(b, \ell) = \alpha(2b-1, \ell, b)$ it follows that the success probability of the adversary is at most
$$\alpha'(b, \ell) = 2^{-2b+1}\ell\frac{(2b-1)!(b\ell-b)!}{(b-1)!(b\ell)!} \ .$$

Writing out the product $\frac{(2b-1)!(b\ell-b)!}{(b-1)!(b\ell)!}$ it is fairly easy to see that for $2 \le b < \ell$ we have that

$$\frac{(2b-1)!(b\ell-b)!}{(b-1)!(b\ell)!} < \frac{(2b)^b}{(b\ell)^b},$$

so

$$\alpha'(b,\ell) \le 2^{-2b+1}\ell\frac{(2b)^b}{(b\ell)^b} = (2\ell)^{1-b} = 2^{(\log_2(\ell)+1)(1-b)}.$$

Thus for $b \ge \frac{\sigma}{1+\log_2(\ell)} + 1$ we have $\alpha'(b,\ell) \le 2^{-\sigma}$ is negligible in σ.

We now prove that assuming each bucket has one non-leaky triple the protocol is secure even for a corrupted P_2.

We look only at the case of two triples, $[x^1]^1_{\Delta_1,2}$,$[y^1]^1_{\Delta_1,2}$,$[z^1]^1_{\Delta_1,2}$ and $[x^2]^1_{\Delta_1,2}$,$[y^2]^1_{\Delta_1,2}$,$[z^2]^1_{\Delta_1,2}$ being combined into $[x]^1_{\Delta_1,2}$, $[y]^1_{\Delta_1,2}$, $[z]^1_{\Delta_1,2}$. It is easy to see why this is sufficient: Consider the iterative way we combine the b triples of a bucket. At each step we combine two triples where one may be the result of previous combinations. Thus if a combination of two triples, involving a non-leaky triple, results in a non-leaky triple, the subsequent combinations involving that result will all result in a non-leaky triple.

In the real world a corrupted P_2 will input keys K_{x^1}, K_{y^1}, K_{z^1} and K_{x^2}, K_{y^2}, K_{z^2} and Δ_1, and possibly some guesses at the x-bits to the leaky $\mathcal{F}_{\text{L-AAND}}$ functionality. Then P_2 will see $d = y^1 \oplus y^2$ and $M_d = (K_{y^1} \oplus K_{y^2}) \oplus d\Delta_1$ and P_1 will output $x = x^1 \oplus x^2$, $y = y^1$, $z = z^1 \oplus z^2 \oplus dx^2$ and $M_x = (K_{x^1} \oplus K_{x^2}) \oplus x\Delta_1$, $M_y = K_{y^1} \oplus y\Delta_1$, $M_z = (K_{z^1} \oplus K_{z^2} \oplus dK_{x^2}) \oplus z\Delta_1$ to the environment.

Consider then a simulator Sim running against P_2 and using an $\mathcal{F}_{\text{AAND}}$ functionality. In the first step Sim gets all P_2's keys like in the real world. If P_2 submits a guess (i, g_i) Sim simply outputs `incorrect` and terminates with probability $\frac{1}{2}$. To simulate opening d, Sim samples $d \in_{\text{R}} \{0,1\}$, sets $M_d = K_{y^1} \oplus K_{y^2} \oplus d\Delta_1$ and sends d and M_d to P_2. Sim then forms the keys $K_x = K_{x^1} \oplus K_{x^2}$, $K_y = K_{y^1}$ and $K_z = K_{z^1} \oplus K_{z^2} \oplus dK_{x^2}$ and inputs them to the $\mathcal{F}_{\text{AAND}}$ functionality on behalf of P_2. Finally the $\mathcal{F}_{\text{AAND}}$ functionality will output random x, y and $z = xy$ and $M_x = K_x \oplus x\Delta_1$, $M_y = K_y \oplus y\Delta_1$, $M_z = K_z \oplus z\Delta_1$.

We have already argued that the probability of P_2 guessing one of the x-bits is exactly $\frac{1}{2}$, so Sim terminates the protocol with the exact same probability as the leaky $\mathcal{F}_{\text{L-AAND}}$ functionality in the real world. Notice then that, given the assumption that P_2 at most guesses one of the x-bits, all bits d, x and y are uniformly random to the environment both in the real world and in the simulation. Thus because Sim can form the keys K_x, K_y and K_z to the $\mathcal{F}_{\text{AAND}}$ functionality exactly as they would be in the real world the simulation will be perfect. □

7.4 Authenticated Oblivious Transfer

In this section we show how to add the **Authenticated OT** part to the limited $\mathcal{F}_{\text{DEAL}}$ functionality. I.e., we show how to provide authenticated OT given the part of the $\mathcal{F}_{\text{DEAL}}$ functionality we implemented using the $\mathcal{F}_{\text{ABIT}}$ functionality in Section 7.2.

The Functionality $\mathcal{F}_{\text{AOT}}(\ell)$

Honest Parties

On input `start` from both parties the functionality samples and outputs uniformly random authenticated bits $([x_0^i]^1_{\Delta_1,2}, [x_1^i]^1_{\Delta_1,2}, [c^i]^2_{\Delta_2,1}, [z^i]^2_{\Delta_2,1})_{i\in[\ell]}$ under the condition that $z^i = c^i(x_0^i \oplus x_1^i) \oplus x_0^i \in \{0,1\}$ for all $i \in [\ell]$.

Corrupt Parties

If P_i is corrupted he gets to choose all his random values (including keys and macs).

Global Key Queries

On input (P_i, Δ) from the adversary the functionality outputs `correct` if $\Delta = \Delta_i$ and `incorrect` otherwise.

Figure 42 The Authenticated OT Functionality

More formally, we show how to implement the functionality $\mathcal{F}_{\text{AOT}}$ described in figure Figure 42, which, as described above, can be used in concert with the $\mathcal{F}_{\text{ABIT}}$ and $\mathcal{F}_{\text{AAND}}$ functionales to implement the $\mathcal{F}_{\text{DEAL}}$ functionality. As with the other functionalities of this section $\mathcal{F}_{\text{AOT}}$ is symmetric so we will show only the construction with P_1 as sender and P_2 as receiver.

The Functionality $\mathcal{F}_{\text{L-AOT}}(\ell)$

Honest Parties

On input `start` from both parties the functionality samples and outputs uniformly random authenticated bits $([x_0^i]^1_{\Delta_1,2}, [x_1^i]^1_{\Delta_1,2}, [c^i]^2_{\Delta_2,1}, [z^i]^2_{\Delta_2,1})_{i\in[\ell]}$ under the condition that $z^i = c^i(x_0^i \oplus x_1^i) \oplus x_0^i \in \{0,1\}$ for all $i \in [\ell]$.

Corrupt Parties

1. If P_i is corrupted he gets to choose all his random values.
2. Additionally, on input (i, g^i) from corrupt P_1, if $c^i \neq g^i$ or if P_2 has already received output, the functionality outputs `incorrect` and terminates before giving outputs. Otherwise the functionality outputs `correct`.

Global Key Queries

On input (P_i, Δ) from the adversary the functionality outputs `correct` if $\Delta = \Delta_i$ and `incorrect` otherwise.

Figure 43 The Leaky Authenticated OT functionality $\mathcal{F}_{\text{L-AOT}}(\ell)$

As with $\mathcal{F}_{\text{AAND}}$ we implement $\mathcal{F}_{\text{AOT}}$ by first implementing a leaky version described in Figure 43. The leaky $\mathcal{F}_{\text{L-AOT}}$ functionality is leaky in the sense that choice bits may leak when P_1 is corrupted: a corrupted P_1 is allowed to make guesses on choice bits, but if the guess is wrong the functionality aborts revealing that P_1 is cheating. This means that if the functionality does not abort, with very high probability P_1 only tried to guess a few choice bits.

The protocol to construct $\mathcal{F}_{\text{L-AOT}}$ (described in Figure 44) proceeds as follows: First P_1 and P_2 get $[x_0]^1_{\Delta_1,2}, [x_1]^1_{\Delta_1,2}$ (P_1's messages), $[c]^2_{\Delta_2,1}$ (P_2's choice bit) and $[r]^2_{\Delta_2,1}$. Then P_1 transfers the message $z = x_c$ to P_2 in the following way: P_2 knows the MAC for his choice bit M_c, while P_1 knows the two keys K_c and Δ_2. This allows P_1 to compute the two possible MACs $(K_c, K_c \oplus \Delta_2)$ respectively for the case of $c = 0$ and $c = 1$. Hashing these values leaves P_1 with two uncorrelated strings $H(K_c)$ and $H(K_c \oplus \Delta_2)$, one of which P_2 can compute as $H(M_c)$. These values can be used as a one-time pad for P_1's bits x_0, x_1 (and some other values as described later), and P_2 can retrieve x_c and send the difference $d = x_c \oplus r$ to P_1, and therefore compute the output $[z]^2_{\Delta_2,1} = [r]^2_{\Delta_2,1} \oplus d$.

The Protocol $\Pi_{\text{L-AOT}}$

The protocol runs ℓ times in parallel, here described for a single leaky authenticated $\mathcal{F}_{\text{OT}}$.

1. P_1 and P_2 get $[x_0]^1_{\Delta_1,2}, [x_1]^1_{\Delta_1,2}, [c]^2_{\Delta_2,1}, [r]^2_{\Delta_2,1}$ from the dealer.
2. Let $[x_0]^1_{\Delta_1,2} = (x_0, M_{x_0}, K_{x_0})$, $[x_1]^1_{\Delta_1,2} = (x_1, M_{x_1}, K_{x_1})$, $[c]^2_{\Delta_2,1} = (c, M_c, K_c)$, $[r]^2_{\Delta_2,1} = (r, M_r, K_r)$.
3. P_1 chooses random strings $T_0, T_1 \in \{0,1\}^{\psi}$.
4. P_1 sends (X_0, X_1) to P_2 where $X_0 = H(K_c) \oplus (x_0||M_{x_0}||T_{x_0})$ and $X_1 = H(K_c \oplus \Delta_2) \oplus (x_1||M_{x_1}||T_{x_1})$.
5. P_2 computes $(x_c||M_{x_c}||T_{x_c}) = X_c \oplus H(M_c)$. P_2 aborts if $M_{x_c} \neq K_{x_c} \oplus x_c\Delta_1$. Otherwise, let $z = x_c$.
6. P_2 sends $d = z \oplus r$ to P_1 and the parties compute $[z]^2_{\Delta_2,1} = [r]^2_{\Delta_2,1} \oplus d$. Let $[z]^2_{\Delta_2,1} = (z, M_z, K_z)$.
7. P_1 sends (I_0, I_1) to P_2 where $I_0 = H(K_z) \oplus T_1$ and $I_1 = H(K_z \oplus \Delta_2) \oplus T_0$.
8. P_2 computes $T_{1\oplus z} = I_z \oplus H(M_z)$. *Notice that now P_2 knows both (T_0, T_1).*
9. P_1 and P_2 both input (T_0, T_1) to $\mathcal{F}_{\text{EQ}}$, aborts the protocol if $\mathcal{F}_{\text{EQ}}$ outputs `unequal`. All ℓ parallel comparisons can be done using a single call to $\mathcal{F}_{\text{EQ}}(2\ell\psi)$.
10. If the values are the equal, P_1 and P_2 output $[x_0]^1_{\Delta_1,2}, [x_1]^1_{\Delta_1,2}, [c]^2_{\Delta_2,1}, [z]^2_{\Delta_2,1}$.

Figure 44 The protocol for Authenticated OT with Leaky Choice Bit

In order to check if P_1 is transmitting the correct bits x_0, x_1, she will transfer the respective MACs together with the bits: as P_2 is supposed to learn x_c, revealing the MAC on this bit does not introduce any insecurity. However, P_1 can now mount a selective failure attack: P_1 can check if P_2's choice bit c is equal to, e.g., 0 by sending x_0 with the correct MAC and x_1 together with an incorrect MAC. Now if $c = 0$ P_2 only sees the valid MAC and continues the protocol, while if $c = 1$ P_2 aborts because of the incorrect MAC. A similar attack can be mounted to check if $c = 1$.

On the other hand, if P_2 is corrupted, he could be sending the wrong value d. In particular, P_1 needs to check that the authenticated bit $[z]^2_{\Delta_2,1}$ is equal to x_c without learning c. In order to do this, we have P_1 choosing two random strings T_0 and T_1, and append them, respectively, to x_0 and x_1 and the MACs on those bits, so that P_2 learns T_c together with x_c. After P_2 sends d, we can again use the MAC and the keys for z to perform a new transfer: P_1 uses $H(K_z)$ as a one-time pad for T_1 and $H(K_z \oplus \Delta_2)$ as a one-time pad for T_0. Using M_z, the MAC on z, P_2 can retrieve $T_{1 \oplus z}$. This means that an honest P_2, that sets $z = x_c$, will know both T_0 and T_1, while a dishonest P_2 will not be able to know both values except with negligible probability. Using the $\mathcal{F}_{\text{EQ}}$ functionality P_1 can check that P_2 knows both values T_0, T_1. Note that we cannot simply have P_2 send these values, as this would open the possibility for new attacks on P_1's side. We prove the following theorem.

Theorem 16. *The protocol in Figure 44 securely implements the leaky $\mathcal{F}_{\text{L-AOT}}(\ell)$ functionality in the ($\mathcal{F}_{\text{ABIT}}$ $(4\ell, \psi), \mathcal{F}_{\text{EQ}}(2\ell\psi)$)-hybrid model.*

The simulator answers global key queries to the dealer by doing the identical global key queries on the leaky $\mathcal{F}_{\text{L-AOT}}(\ell)$ functionality and returning the reply. This gives a perfect simulation of these queries, and we ignore them below.

Notice that for honest sender and receiver correctness of the protocol follows immediately from correctness of the $\mathcal{F}_{\text{ABIT}}$ functionality. Thus we prove Theorem 16 by proving security separately against corrupted P_1 and P_2 respectively. We do this in Lemma 20 and 21.

Lemma 20. *The protocol in Figure 44 securely implements the leaky $\mathcal{F}_{\text{L-AOT}}(\ell)$ functionality against corrupted P_1.*

Proof. We consider the case of a corrupt sender P_1 running the above protocol against a simulator $\mathcal{S}$. We show how to simulate one instance.

1. First $\mathcal{S}$ receives P_1's input (M_{x_0}, x_0), (M_{x_1}, x_1), K_c, K_r and Δ_2 to the dealer. Then $\mathcal{S}$ samples a bit $y \in_{\text{R}} \{0, 1\}$, sets $K_z = K_r \oplus y\Delta_2$ and inputs (M_{x_0}, x_0), (M_{x_1}, x_1), K_c, K_z and Δ_2 to the leaky $\mathcal{F}_{\text{L-AOT}}$ functionality. The functionality outputs Δ_1, (M_c, c), (M_z, z), K_{x_0} and K_{x_1} to the honest P_2 as described in the protocol.
2. P_1 outputs the message (X_0, X_1). The simulator knows Δ_2 and K_c and can therefore compute
$$X_0 \oplus H(K_c) = (\overline{x}_0 || \overline{M}_{x_0} || T'_{x_0})$$
and
$$X_1 \oplus H(K_c \oplus \Delta_2) = (\overline{x}_1 || \overline{M}_{x_1} || T'_{x_1}) \ .$$
For all $j \in \{0, 1\}$ $\mathcal{S}$ tests if $(\overline{M}_{x_j}, \overline{x}_j) = (M_{x_j}, x_j)$. If, for some j, this is not the case $\mathcal{S}$ inputs a guess to the leaky $\mathcal{F}_{\text{L-AOT}}$ functionality guessing that $c = (1 - j)$. If the functionality outputs `incorrect` $\mathcal{S}$ aborts the protocol. Otherwise $\mathcal{S}$ proceeds by sending y to P_1. Notice that if $\mathcal{S}$ does not abort but does guess the choice bit c it can perfectly simulate the remaining protocol. In the following we therefore assume this is not the case.
3. Similarly $\mathcal{S}$ gets (I_0, I_1) from P_1 and computes
$$I_0 \oplus H(K_z) = T''_1$$
and
$$I_1 \oplus H(K_z \oplus \Delta_2) = T''_0 \ .$$

4. When $\mathcal{S}$ receives P_1's input (T_0, T_1) for the $\mathcal{F}_{\text{EQ}}$ functionality it first tests if $(T'_j, T''_{1\oplus\overline{x}_j}) = (T_{\overline{x}_j}, T_{1\oplus\overline{x}_j})$ for all $j \in \{0,1\}$. If, for some j, this is not the case $\mathcal{S}$ inputs a guess to the leaky $\mathcal{F}_{\text{L-AOT}}$ functionality guessing that $c = (1-j)$. If the functionality outputs `incorrect`, $\mathcal{S}$ aborts the protocol. If not, the simulation is over.

For analysis of the simulation we denote by F the event that for some $j \in \{0,1\}$ P_1 computes values $M^*_{x_j} \in \{0,1\}^\psi$ and $x^*_j \in \{0,1\}$ so that $(M^*_{x_j}, x^*_j) \neq (M_{x_j}, x_j)$ and $M^*_{x_j} = K_{x_j} \oplus x^*_j \Delta_1$. In other words, F is the event that P_1 computes a MAC on a message bit it was not supposed to know. We will now show that, assuming F does not occur, the simulation is perfectly indistinguishable from the real protocol. We then show that F only occurs with negligible probability and therefore that simulation and the real protocol are indistinguishable.

From the definition of the leaky $\mathcal{F}_{\text{L-AOT}}$ functionality we have that $(\overline{M}_{x_j}, \overline{x}_j) = (M_{x_j}, x_j)$ implies $\overline{M}_{x_j} = K_{x_j} \oplus x_j \Delta_1$. Given the assumption that F does not occur clearly we have that $(\overline{M}_{x_j}, \overline{x}_j) \neq (M_{x_j}, x_j)$ also implies $\overline{M}_{x_j} \neq K_{x_j} \oplus \overline{x}_j \Delta_1$. This means that $\mathcal{S}$ aborts in step 2 with exactly the same probability as the honest receiver would in the real protocol. Also, in the real protocol we have $y = z \oplus r$ for $r \in_{\text{R}} \{0,1\}$ thus both in the real protocol and the simulation y is distributed uniformly at random in the view of P_1.

Next in step 4 of the simulation notice that in the real protocol, if $c = j \in \{0,1\}$, an honest P_2 would input T'_j and $T''_{1\oplus\overline{x}_j}$ to $\mathcal{F}_{\text{EQ}}$ (sorted in the correct order). The protocol would then continue if and only if $(T'_j, T''_{1\oplus\overline{x}_j}) = (T_{\overline{x}_j}, T_{1\oplus\overline{x}_j})$ and abort otherwise, i.e., the real protocol would continue if and only if $(T'_j, T''_{1\oplus\overline{x}_j}) = (T_{\overline{x}_j}, T_{1\oplus\overline{x}_j})$ and $c = j$, which is exactly what happens in the simulation. Thus we have that given F does not occur, all input to P_1 during the simulation is distributed exactly as in real protocol. In other words the two are perfectly indistinguishable.

Now assume F does occur, that is for some $j \in \{0,1\}$ P_1 computes values $M^*_{x_j}$ and x^*_j as described above. In that case P_1 could compute the global key of the honest receiver as $M^*_j \oplus M_{x_j} = \Delta_1$. However, since all inputs to P_1 are independent from Δ_1 (during the protocol), P_1 can only guess Δ_1 with negligible probability (during the protocol) and thus F can only occur with negligible probability (during the protocol). After the protocol P_1, or rather the environment, will receive outputs and learn Δ_1, but this does not change the fact that guessing Δ_1 during the protocol can be done only with negligible probability. □

Lemma 21. *The protocol in Figure 44 securely implements leaky $\mathcal{F}_{\text{L-AOT}}(\ell)$ against corrupted P_2.*

Proof. We consider the case of a corrupt receiver P_2 running the above protocol against a simulator $\mathcal{S}$. The simulation runs as follows.

1. The simulation starts by $\mathcal{S}$ getting P_2's input to dealer Δ_1, (M_c, c), (M_r, r), K_{x_0} and K_{x_1}. Then $\mathcal{S}$ simply inputs Δ_1, (M_c, c), $M_z = M_r$, K_{x_0} and K_{x_1} to the leaky $\mathcal{F}_{\text{L-AOT}}$ functionality. The functionality outputs z to $\mathcal{S}$ and Δ_2, (M_{x_0}, x_0), (M_{x_1}, x_1), K_c and K_z to the sender as described above.
2. Like the honest sender $\mathcal{S}$ samples random keys $T_0, T_1 \in_{\text{R}} \{0,1\}^\psi$. Since $\mathcal{S}$ knows $M_c, K_{x_0}, K_{x_1}, \Delta_1, c$ and $z = x_c$ it can compute $X_c = H(M_c) \oplus (z||M_z||T_z)$ exactly as the honest sender would. It then samples $X_{1\oplus c} \in_{\text{R}} \{0,1\}^{2\psi+1}$ and inputs (X_0, X_1) to P_2.
3. The corrupt receiver P_2 replies by sending some $\overline{y} \in \{0,1\}$.
4. $\mathcal{S}$ sets $\overline{z} = r \oplus \overline{y}$, computes $I_{\overline{z}} = H(M_z) \oplus T_{1\oplus\overline{z}}$ and samples $I_{1\oplus\overline{z}} \in_{\text{R}} \{0,1\}^\psi$. It then inputs (I_0, I_1) to P_2.
5. P_2 outputs some $(\overline{T}_0, \overline{T}_1)$ for the $\mathcal{F}_{\text{EQ}}$ functionality and $\mathcal{S}$ continues or aborts as the honest P_1 would in the real protocol, depending on whether or not $(T_0, T_1) = (\overline{T}_0, \overline{T}_1)$.

For the analysis we denote by F the event that P_2 queries the RO on $K_c \oplus (1 \oplus c)\Delta_2$ or $K_z \oplus (1 \oplus z)\Delta_2$. We first show that assuming F does not occur, the simulation is perfect. We then show that F only occurs with negligible probability (during the protocol) and thus the simulation is indistinguishable from the real protocol (during the protocol). We then discuss how to simulate the RO after outputs have been delivered.

First in the view of P_2 step 1 of the simulation is clearly identical to the real protocol. Thus the first deviation from the real protocol appears in step 2 of the simulation where the $X_{1\oplus c}$ is chosen uniformly at random. However, assuming F does not occur, P_2 has no information on $H(K_c \oplus (1 \oplus c)\Delta_2)$ thus in the view of P_2, $X_{1\oplus c}$ in the real protocol is a one-time pad encryption of $(x_{1\oplus c}||M_{x_{1\oplus c}}||T_{x_{1\oplus c}})$. In other words, assuming F does not occur, to P_2, $X_{1\oplus c}$ is uniformly random in both the simulation and the real protocol, and thus all input to P_2 up to step 2 is distributed identically in the two cases.

For steps 3 to 5 notice that in the real protocol an honest sender would set $K_z = K_r \oplus \overline{y}\varDelta_2$ and we would have

$$(K_r \oplus \overline{y}\varDelta_2) \oplus \overline{z}\varDelta_2 = K_r \oplus r\varDelta_2 = M_r \ .$$

Thus we have that the simulation generates $I_{\overline{z}}$ exactly as in the real protocol. An argument similar to the one above for step 2 then gives us that the simulation is perfect given the assumption that F does not occur.

We now show that P_2 can be modified so that if F does occur, then P_2 can find $\varDelta_2$. However, since all input to P_2 are independent of $\varDelta_2$ (during the protocol), P_2 only has negligible probability of guessing $\varDelta_2$ and thus we can conclude that F only occurs with negligible probability.

The modified P_2 keeps a list $Q = (Q_1, \dots, Q_q)$ of all P_2's queries to H. Since P_2 is efficient we have that q is a polynomial in ψ. To find $\varDelta_2$ the modified P_2 then goes over all $Q_k \in_{\mathrm{R}} Q$ and computes $Q_k \oplus M_z = \varDelta'$ and $Q_k \oplus M_c = \varDelta''$. Assuming that F does occur there will be some $Q_{k'} \in Q$ s.t. $\varDelta' = \varDelta_2$ or $\varDelta'' = \varDelta_2$. The simulator can therefore use global key queries to find $\varDelta_2$ if F occurs.

We then have the issue that after outputs are delivered to the environment, the environment learns $\varDelta_2$, and we have to keep simulating H to the environment after outputs are delivered. This is handled exactly as in the proof of Theorem 14 using the programability of the RO. □

To deal with the leakage of the leaky $\mathcal{F}_{\text{L-AOT}}$ functionality, we let P_2 randomly partition and combine the leaky aOTs in buckets, in a way similar to how we dealt with leakage for $\mathcal{F}_{\text{AAND}}$: the leaky AOTs in each bucket will be combined using an OT combiner (as shown in Figure 45), in so that if at least one choice bit in every bucket is unknown to P_1, then the resulting aOT will not be leaky. We prove the following theorem.

The Protocol $\varPi_{\text{AOT}}$

1. P_1 and P_2 generate $\ell' = b\ell$ authenticated OTs using the leaky $\mathcal{F}_{\text{L-AOT}}(\ell')$ functionality. If the functionality does not abort, name the outputs $\{[x_0^i]^1_{\Delta_{1,2}}, [x_1^i]^1_{\Delta_{1,2}}, [c^i]^2_{\Delta_{2,1}}, [z^i]^2_{\Delta_{2,1}}\}_{i=1}^{\ell'}$.
2. P_2 sends a b-wise independent permutation π on $\{1, \dots, \ell'\}$ to P_1. For $j = 0, \dots, \ell-1$, the b quadruples
$$\{[x_0^{\pi(i)}]^1_{\Delta_{1,2}}, [x_1^{\pi(i)}]^1_{\Delta_{1,2}}, [c^{\pi(i)}]^2_{\Delta_{2,1}}, [z^{\pi(i)}]^2_{\Delta_{2,1}}\}_{i=jb+1}^{jb+b}$$
are defined to be in the j'th bucket.
3. P_1 and P_2 combine the b quadruples in each bucket. Here we describe how to combine two quadruples from in a bucket: let them be $[x_0^1]^1_{\Delta_{1,2}}, [x_1^1]^1_{\Delta_{1,2}}, [c^1]^2_{\Delta_{2,1}}, [z^1]^2_{\Delta_{2,1}}$ and $[x_0^2]^1_{\Delta_{1,2}}, [x_1^2]^1_{\Delta_{1,2}}, [c^2]^2_{\Delta_{2,1}}, [z^2]^2_{\Delta_{2,1}}$. Let the result be $[x_0]^1_{\Delta_{1,2}}, [x_1]^1_{\Delta_{1,2}}, [c]^2_{\Delta_{2,1}}, [z]^2_{\Delta_{2,1}}$. To combine more than two, just iterate by taking the result and combine it with the next quadruple in the bucket.
   (a) P_1 opens $d = x_0^1 \oplus x_1^1 \oplus x_0^2 \oplus x_1^2$.
   (b) P_1 and P_2 compute
   $$[c]^2_{\Delta_{2,1}} = [c^1]^2_{\Delta_{2,1}} \oplus [c^2]^2_{\Delta_{2,1}} \ , [z]^2_{\Delta_{2,1}} = [z^1]^2_{\Delta_{2,1}} \oplus [z^2]^2_{\Delta_{2,1}} \oplus d[c^1]^2_{\Delta_{2,1}} \ ,$$
   and
   $$[x_0]^1_{\Delta_{1,2}} = [x_0^1]^1_{\Delta_{1,2}} \oplus [x_0^2]^1_{\Delta_{1,2}} \ , [x_1]^1_{\Delta_{1,2}} = [x_0^1]^1_{\Delta_{1,2}} \oplus [x_1^2]^1_{\Delta_{1,2}} \ .$$

Figure 45 From Leaky Authenticated OTs to Authenticated OTs

Theorem 17. *Let $\mathcal{F}_{AOT}(\ell)$ denote the functionality that provides ℓ non-leaky aOTs as in $\mathcal{F}_{\text{DEAL}}$. If $b \geq \frac{\sigma}{1+\log_2(\ell)} + 1$, then the protocol in Figure 45 securely implements $\mathcal{F}_{AOT}(\ell)$ in the leaky $\mathcal{F}_{\text{L-AOT}}(b\ell)$-hybrid model with statistical security parameter σ.*

Proof. We want to show that the protocol in Figure 45 provides non-leaky aOTs, having access to a functionality that provides leaky aOTs.

In the protocol the receiver randomly partitions ℓb leaky aOTs in ℓ buckets of size b. First we want to argue that the probability that every bucket contains at least one aOT where the choice bit is unknown to the adversary is overwhelming. Repeating the same calculations as in the proof of Theorem 15 it turns out that this happens with probability bigger than $1 - (2\ell)^{(1-b)} \geq 1 - 2^{-\sigma}$.

Once we know that (with overwhelming probability) at least one OT in every bucket is secure for the receiver (i.e., at least one choice bit is uniformly random in the view of the adversary), the security of the protocol follows from the fact that we use a standard OT combiner [HKN+05]. Turning this into a simulation proof can be easily done in a way similar to the proof of Theorem 15. □

This completes the description of our protocol.

7.5 Complexity Analysis

We will now sketch a complexity analysis counting the calls to symmetric primitives used in the TinyOT protocol.

As is shown in Theorem 13, the protocol requires an initial call to an ideal functionality for $(\mathcal{F}_{\mathrm{OT}}(\frac{110}{6}\psi, \psi), \mathcal{F}_{\mathrm{EQ}}(\psi))$. After this, the cost per gate is only a number of invocations to a cryptographic hash function H. In this section we give the exact number of hash functions that we use in the construction of the different primitives. As the final protocol is completely symmetric, we count the total number of calls to H made by both parties. Below b is the "bucket size" used in protocols for aOT and aAND.

Equality $\mathcal{F}_{\mathbf{EQ}}$: The $\mathcal{F}_{\mathrm{EQ}}$ functionality can be securely implemented with 2 calls to a hash function H, as described in Section 3 (as each call to $\mathcal{F}_{\mathrm{EQ}}$ in this section compares strings of length ψ)

Authenticated AND $\mathcal{F}_{\mathbf{AAND}}$: Every aAND costs $3b$ aBits, b calls to $\mathcal{F}_{\mathrm{EQ}}$, and $6b$ calls to H, as described in Figure 39 and Figure 41. Note, that while Figure 39 only has 3 calls to H they are on strings of size 2ψ which means we count them double, as described in Section 3.

Authenticated OT $\mathcal{F}_{\mathbf{AOT}}$: Every aOT costs $4b$ aBits, $2b$ calls to $\mathcal{F}_{\mathrm{EQ}}$, and $9b$ calls to H, as described in Figure 44 and Figure 45. Note, that while Figure 44 only has 6 calls to H, 3 of these expand to strings of size 2ψ which means we count them double.

MTriple Protocol, AND Gate: AND gates cost 2 aOTs, 2 aANDs and 2 aBits.

The cost per aBit, requires 74 calls to H as described in Section 5. By plugging in these values we get that the cost per input gate is 74 calls to H, and the cost per AND gate is $1078b + 148$ calls to H.

As described in Section 7.1 we can greatly reduce communication complexity of our protocol by deferring the MAC checks. However, this trick comes at cost of two calls to H (one for each player) every time we do an opening. This adds $2b$ hashes for each aOT and aAND (as protocols in Figure 44 and Figure 41 involves one opening each) and in total adds $8b+20$ hashes to the cost each AND gate (as the AND gate in Figure 36 uses additional 10 openings).

Acknowledgements

Jesper Buus Nielsen was partially supported by the Danish National Research Foundation and the National Science Foundation of China (under the grant 61061130540) for the Sino-Danish Center for the Theory of Interactive Computation and a Sapere Aude grant from the Danish Council for Independent Research. Claudio Orlandi was supported by the European Research Council as part of the ERC project LAST. Enrique Larraia, Emmanuela Orsini, Peter Scholl, and Nigel P. Smart were supported in part by ERC Advanced Grant ERC-2010-AdG-267188-CRIPTO and by EPSRC via grants EP/I03126X and EP/M012824. Nigel P. Smart was partially supported by Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under agreement number FA8750-11-2-0079[20].

[20] 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.

References

AHI11. Benny Applebaum, Danny Harnik, and Yuval Ishai. Semantic security under related-key attacks and applications. In Bernard Chazelle, editor, *ICS*, pages 45–60. Tsinghua University Press, 2011.

asS11. a. shelat and Chih-Hao Shen. Two-output secure computation with malicious adversaries. In Paterson [Pat11], pages 386–405.

BDNP08. Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: a system for secure multi-party computation. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *CCS*, pages 257–266. ACM, 2008.

BDOZ11. Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In Paterson [Pat11], pages 169–188.

Bea95. Donald Beaver. Precomputing oblivious transfer. In Don Coppersmith, editor, *Advances in Cryptology - CRYPTO '95, 15th Annual International Cryptology Conference, Santa Barbara, California, USA, August 27-31, 1995, Proceedings*, volume 963 of *Lecture Notes in Computer Science*, pages 97–109. Springer, 1995.

Bea96. Donald Beaver. Correlated pseudorandomness and the complexity of private computations. In Gary L. Miller, editor, *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 479–488. ACM, 1996.

BLW08. Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In *European Symposium on Research in Computer Security – ESORICS 20 08*, volume 5283 of *LNCS*, pages 192–206. Springer, 2008.

CDE^{+}18. Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. Spd$F_{2^{k}}$: Efficient MPC mod 2^{k} for dishonest majority. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part II*, volume 10992 of *Lecture Notes in Computer Science*, pages 769–798. Springer, 2018.

CHK^{+}12. Seung Geol Choi, Kyung-Wook Hwang, Jonathan Katz, Tal Malkin, and Dan Rubenstein. Secure multi-party computation of boolean circuits with applications to privacy in on-line marketplaces. In Orr Dunkelman, editor, *CT-RSA*, volume 7178 of *Lecture Notes in Computer Science*, pages 416–432. Springer, 2012.

CKKZ12. Seung Geol Choi, Jonathan Katz, Ranjit Kumaresan, and Hong-Sheng Zhou. On the security of the "free-xor" technique. In Ronald Cramer, editor, *TCC*, volume 7194 of *Lecture Notes in Computer Science*, pages 39–53. Springer, 2012.

DGKN09. Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous multiparty computation: Theory and implementation. In Stanislaw Jarecki and Gene Tsudik, editors, *Public Key Cryptography - PKC 2009, 12th International Conference on Practice and Theory in Public Key Cryptography, Irvine, CA, USA, March 18-20, 2009. Proceedings*, volume 5443 of *Lecture Notes in Computer Science*, pages 160–179. Springer, 2009.

DKL^{+}13. Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure mpc for dishonest majority - or: Breaking the spdz limits. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS*, volume 8134 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2013.

DO10. Ivan Damgård and Claudio Orlandi. Multiparty computation for dishonest majority: From passive to active security at low cost. In Tal Rabin, editor, *CRYPTO*, volume 6223 of *Lecture Notes in Computer Science*, pages 558–576. Springer, 2010.

DPSZ12. Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Safavi-Naini and Canetti [SNC12], pages 643–662.

DZ13. Ivan Damgård and Sarah Zakarias. Constant-overhead secure computation of boolean circuits using preprocessing. In *TCC*, pages 621–641, 2013.

FJN^{+}13. Tore Kasper Frederiksen, Thomas Pelle Jakobsen, Jesper Buus Nielsen, Peter Sebastian Nordholt, and Claudio Orlandi. Minilego: Efficient secure two-party computation from general assumptions. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT*, volume 7881 of *Lecture Notes in Computer Science*, pages 537–556. Springer, 2013.

FKOS15. Tore Kasper Frederiksen, Marcel Keller, Emmanuela Orsini, and Peter Scholl. A unified approach to MPC with preprocessing using OT. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part I*, volume 9452 of *Lecture Notes in Computer Science*, pages 711–735. Springer, 2015.

GMW87. Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *STOC*, pages 218–229. ACM, 1987.

Gol04. Oded Goldreich. *Foundations of Cryptography, Vol. 2*. Cambridge University Press, 2004. `http://www.wisdom.weizmann.ac.il/~oded/foc-vol2.html`.

HEK$^+$11. Yan Huang, David Evans, Jonathan Katz, , and Lior Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium*, 2011.

HIKN08. Danny Harnik, Yuval Ishai, Eyal Kushilevitz, and Jesper Buus Nielsen. OT-combiners via secure computation. In Ran Canetti, editor, *TCC*, volume 4948 of *Lecture Notes in Computer Science*, pages 393–411. Springer, 2008.

HKN$^+$05. Danny Harnik, Joe Kilian, Moni Naor, Omer Reingold, and Alon Rosen. On robust combiners for oblivious transfer and other primitives. In Ronald Cramer, editor, *EUROCRYPT*, volume 3494 of *Lecture Notes in Computer Science*, pages 96–113. Springer, 2005.

HKS$^+$10. Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. Tasty: tool for automating secure two-party computations. In *CCS*, pages 451–462, 2010.

HOSS18a. Carmit Hazay, Emmanuela Orsini, Peter Scholl, and Eduardo Soria-Vazquez. Concretely efficient large-scale MPC with active security (or, tinykeys for tinyot). In Thomas Peyrin and Steven D. Galbraith, editors, *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part III*, volume 11274 of *Lecture Notes in Computer Science*, pages 86–117. Springer, 2018.

HOSS18b. Carmit Hazay, Emmanuela Orsini, Peter Scholl, and Eduardo Soria-Vazquez. Tinykeys: A new approach to efficient multi-party computation. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part III*, volume 10993 of *Lecture Notes in Computer Science*, pages 3–33. Springer, 2018.

HSS17. Carmit Hazay, Peter Scholl, and Eduardo Soria-Vazquez. Low cost constant round MPC combining BMR and oblivious transfer. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 598–628. Springer, 2017.

IKNP03. Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 145–161. Springer, 2003.

IKOS08. Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Cryptography with constant computational overhead. In Cynthia Dwork, editor, *STOC*, pages 433–442. ACM, 2008.

IPS08. Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer - efficiently. In David Wagner, editor, *CRYPTO*, volume 5157 of *Lecture Notes in Computer Science*, pages 572–591. Springer, 2008.

IPS09. Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Secure arithmetic computation with no honest majority. In Reingold [Rei09], pages 294–314.

JMN10. Thomas P. Jakobsen, Marc X. Makkes, and Janus Dam Nielsen. Efficient implementation of the orlandi protocol. In Jianying Zhou and Moti Yung, editors, *ACNS*, volume 6123 of *Lecture Notes in Computer Science*, pages 255–272, 2010.

KOS16. Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 830–842. ACM, 2016.

KS08. Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free xor gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP (2)*, volume 5126 of *Lecture Notes in Computer Science*, pages 486–498. Springer, 2008.

LOP11. Yehuda Lindell, Eli Oxman, and Benny Pinkas. The ips compiler: Optimizations, variants and concrete efficiency. In Phillip Rogaway, editor, *CRYPTO*, volume 6841 of *Lecture Notes in Computer Science*, pages 259–276. Springer, 2011.

LOS14. Enrique Larraia, Emmanuela Orsini, and Nigel P. Smart. Dishonest majority multi-party computation for binary circuits. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II*, volume 8617 of *Lecture Notes in Computer Science*, pages 495–512. Springer, 2014.

LP11. Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. In Yuval Ishai, editor, *Theory of Cryptography - 8th Theory of Cryptography Conference, TCC 2011, Providence, RI, USA, March 28-30, 2011. Proceedings*, volume 6597 of *Lecture Notes in Computer Science*, pages 329–346. Springer, 2011.

LPS08. Yehuda Lindell, Benny Pinkas, and Nigel P. Smart. Implementing two-party computation efficiently with security against malicious adversaries. In Rafail Ostrovsky, Roberto De Prisco, and Ivan Visconti, editors, *SCN*, volume 5229 of *Lecture Notes in Computer Science*, pages 2–20. Springer, 2008.

MK10. Lior Malka and Jonathan Katz. VMCrypt - modular software architecture for scalable secure computation. Cryptology ePrint Archive, Report 2010/584, 2010. `http://eprint.iacr.org/`.

MNPS04. Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - secure two-party computation system. In *USENIX Security Symposium*, pages 287–302. USENIX, 2004.

Nie07. Jesper Buus Nielsen. Extending oblivious transfers efficiently - how to get robustness almost for free. Cryptology ePrint Archive, Report 2007/215, 2007. `http://eprint.iacr.org/`.

NNOB12. Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Safavi-Naini and Canetti [SNC12], pages 681–700. Full version in IACR ePrint 2011/091 `https://eprint.iacr.org/2011/091`.

NO09. Jesper Buus Nielsen and Claudio Orlandi. LEGO for two-party secure computation. In Reingold [Rei09], pages 368–386.

Pat11. Kenneth G. Paterson, editor. *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings*, volume 6632 of *Lecture Notes in Computer Science*. Springer, 2011.

PSSW09. Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. In Mitsuru Matsui, editor, *ASIACRYPT*, volume 5912 of *Lecture Notes in Computer Science*, pages 250–267. Springer, 2009.

Rei09. Omer Reingold, editor. *Theory of Cryptography, 6th Theory of Cryptography Conference, TCC 2009, San Francisco, CA, USA, March 15-17, 2009. Proceedings*, volume 5444 of *Lecture Notes in Computer Science*. Springer, 2009.

SIM. SIMAP Project. SIMAP: Secure information management and processing. `http://alexandra.dk/uk/Projects/Pages/SIMAP.aspx`.

SNC12. Reihaneh Safavi-Naini and Ran Canetti, editors. *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*. Springer, 2012.

WRK17a. Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 21–37. ACM, 2017.

WRK17b. Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 39–56. ACM, 2017.

Yao82. Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164. IEEE, 1982.