When It's All Just Too Much: Outsourcing MPC-Preprocessing

Peter Scholl, Nigel P. Smart, and Tim Wood

Dept. Computer Science, University of Bristol, United Kingdom.

Abstract. Most modern actively secure multiparty computation protocols make use of a function and input independent pre-processing phase. This pre-processing phase is tasked with producing some form of correlated randomness and distributing it to the parties. Whilst the "online" phase of such protocols is exceedingly fast, the bottleneck comes in the pre-processing phase. In this paper we examine situations where the computing parties in the online phase may want to outsource the pre-processing phase to another set of parties, or to a sub-committee. We examine how this can be done, and also describe situations where this may be a benefit.

1 Introduction

Secure multi-party computation (MPC) is the idea of allowing multiple parties to compute on their combined inputs in a "secure" manner. We use the word secure to mean that the interaction provides no party with any information on the secret inputs of the other parties, bar what can be learned from the output (a property called "privacy" or "secrecy"). In this paper we will focus on protocols which can tolerate a majority of the parties being corrupted. In such a situation we know there is no hope that the honest parties can always obtain the correct output. In such a situation we require that either the honest parties obtain the correct result, or they abort (with overwhelming probability).

For a long time, MPC remained a theoretical exercise and implementations were impractical. However, much work has recently been undertaken on developing practical MPC protocols in the so-called "pre-processing model". In this model, the protocol is split up into an offline phase and an online phase. In the offline (i.e. pre-processing) phase the parties execute a protocol which emulates a "trusted dealer" who distributes "raw material" (pre-processed data) to parties; this data is then used up in the online phase as the circuit is evaluated. The advantage of doing this is that the pre-processing involves expensive public key operations which can be isolated to the pre-processing phase. In addition, the pre-processed data can be made independent of both the inputs and the circuit, so it can be computed at any point prior to the evaluation of the circuit. The online phase is then executed with (essentially) information theoretic primitives, and is thus very fast, whilst all the computationally expensive machinary can be relegated to the offline phase.

This protocol idea goes back to Beaver [Bea96]. It was first used in a practical (and implemented) MPC system in the VIFF protocol [DGKN09], which was a protocol system built for the case of honest majority MPC. Modern dishonest majority MPC protocols make use of information theoretic MACs to achieve active security, an idea which goes back to [RBO89] When combined with the pre-processed triple idea of Beaver these two ideas have, in the last five years, resulted in a step change in what can be implemented efficiently by MPC protocols.

The first protocol in this area was BDOZ [BDOZ11], which demonstrated that if the number of parties was constant and the parties had access to a functionality which would provide the pre-processed data then the overhead of computing the (arithmetic) circuit (over a large finite field) securely is only a constant factor times the work required to compute it in the clear. The SPDZ [DPSZ12] protocol showed that the mere constant factor overhead encountered in the BDOZ protocol holds for any number of parties. Further improvements were presented in [DKL+13] to the SPDZ protocol. In the BDOZ and SPDZ protocols the pre-processing is produced using forms of homomorphic encryption, and so the protocols are more suited to MPC over a large finite field. In TinyOT [NNOB12], similar results in the two-party case for Boolean circuits were given, where the pre-processing was implemented using oblivious transfer (OT) extension. In [LOS14,BLN+15], the TinyOT protocol was extended to the multi-party case, and the online phase was made consistent (in terms of computational pattern) with that of the SPDZ protocol from [DKL+13]. Further unification of these protocol families occured with the replacement of the homomorphic encryption based pre-processing phase of SPDZ with an OT based pre-processing [KOS16], forming what is known as the MASCOT protocol. To simplify exposition, since all of these protocols are essentially the same at a high level, we shall refer to them as one in this paper, namely as the "SPDZ family".

As already remarked, the SPDZ family of protocols has an efficient online phase; indeed, the online phase has a number of interesting properties:

- **Computational Efficiency:** Since the online phase is made up of information theoretic primitives, the basic arithmetic operations are incredibly simple, requiring only a constant multiplicative factor increase in the number of operations when compared to evaluating the function in the clear. Before every output operation, the execution of a PRF is also required for MAC checking, but for a large computation this can be ignored.
- **Communication Efficiency:** The basic protocol requires interaction for each multiplication operation[1]. This interaction need only be conducted over authenticated channels, rather than private channels, and the communication required grows linearly in the number of players.
- **Deterministic:** Given the correlated randomness from the offline phase, the function to be computed, and the parties' inputs, the online phase is essentially deterministic. The only random data needed is a small amount

[1] For simplicity of expression we assume the MPC functionality is evaluating an arithmetic circuit over a finite field. This is purely for exposition: in practice the usual MPC tricks to remove the need for circuit based computation will be used.

per party to ensure that dishonest parties are detected in the MAC checking protocol. Indeed this random data can be created in the offline phase and then stored for later use.

The simplicity and efficiency of the online phase, however, comes with a penalty in the offline phase. Using either method to generate the pre-processed data, viz. homomorphic encryption or OT, the offline phase requires expensive public key machinery, and in practice is a couple of orders of magnitude slower than the online phase. In some instances, while the online phase is computationally cheap enough to be executed by a relatively low powered computing device, the same device would not be sufficiently powerful to perform the associated offline phase efficiently. This can cause a problem when we have parties with very different computing power. Similarly the offline phase requires the transmission of a larger amount of data per multiplication gate than the online phase. Again, this can be a problem in practice if certain parties are on a slow part of the network.

The offline phase also requires each party to input a large amount of randomness, and it is well known that one of the major challenges of running any cryptography in the real world is the generation of randomness. Small hardware devices may not have the capability of producing random values easily, as they usually have very limited access to good sources of entropy. For example, devices such as mobile phones and tablets still have problems with good entropy sources. Moreover, it does not suffice simply to be able to generate random numbers: in many cryptographic applications (including MPC) it is necessary that it be high quality randomness. This has led to high-end applications requiring expensive dedicated hardware to produce entropy; but such dedicated hardware may not be on all computers in a computation. Thus, even in the case of high-end servers executing the MPC protocol, it may easily not be the case that all have access to a sufficient entropy source.

For these reasons, we propose a method of outsourcing the offline pre-processing for the SPDZ family of protocols to a different set of parties. We will let Q denote the set of n_Q parties who are to run the online phase; the set Q will outsource the computation of the pre-processing to a set of parties R of size n_R. This set R may be a strict subset of Q, or they could be a completely different set all together. The idea is that Q is unable to execute the pre-processing as an n_Q-party protocol, due to some limitation of resources (computation, bandwidth, or randomness, for example), whereas R is "more able" to execute the pre-processing as an n_R-party protocol. Our protocol to perform this outsourcing will also aim to minimise the communication needed to transfer the pre-processing data from the set R to the set Q.

Of course for this to make sense it is important that the set Q trust the set R to perform this task, and that the protocol respect this trust relationship. In particular we assume an adversary which can corrupt a majority of parties in Q and a majority of parties in R. We suppose the adversary can neither corrupt all parties in R nor all parties in Q: indeed, in this case we clearly would not even be *expected* to guarantee any security. In particular, this means that each

honest party in Q believes that there is at least one honest party in R, but they may not know which one is honest.

To understand this trust relationship in more detail: our protocol divides the set Q into (not necessarily disjoint) subsets $\{Q_i\}_{i \in R}$ and associates each subset with a party in R; namely, party $i \in R$ is assigned the set Q_i. Our protocol will be secure if there is at least one pair (i, Q_i) where $i \in R$ is an honest party in R, and Q_i contains at least one honest party from Q. This raises (at least) three potential ways for the subsets to arise:

- If $R \subseteq Q$ then we make sure Q_i contains honest player i from R. This can be done, for example, by each party i in R including itself in the set Q_i. Thus, here we guarantee that Q_i contains an honest party if i is honest, so a pair (i, Q_i) as described above exists.
- It may be the case that every party in Q trusts at least one party in R already. In this case, our cover, $\{Q_i\}_{i \in R}$, can be produced by letting parties in Q elect which parties in R they want to be associated with.
- If no prior trust relation is known then this cover must be defined either deterministically or probabilistically. If deterministically, to satisfy the requirement above we must choose $Q_i = Q$ for all i. This guarantees a pair (i, Q_i) as described above, but results in an inefficient network topology (since each party in R needs a secure channel to each party in Q). Alternatively, we make a probabilistic assignment and derive bounds on n_Q and n_R which ensure that the assignment preserves security with overwhelming probability: see Section 4 for details.

As an example of the practicality of using the protocol and probabilistic algorithm we describe: if there are 5 parties in R of which at most 3 are corrupt, and 1000 parties in Q of which at most 781 are corrupt, each party in R need only send to 200 parties in Q (i.e. exactly as many are needed to provide a cover) for the cover to be secure (i.e. so the adversary cannot with with probability greater than $1 - 2^{-80}$).

Besides the ability of the protocol we describe to enable localising the generation of pre-processed data, another potential application of the protocol is to increase the number of parties involved in a given instance of the SPDZ protocol dynamically (i.e. during the online phase). For example, suppose a set of parties already running an instance of the SPDZ protocol want to (efficiently and securely) allow another set of parties to join them during a reactive computation. It may make more sense to transform the already pre-processed data (or even just a few pre-processed values) via our protocol to a form that is amenable for use by a larger number of parties, and then distribute it to the parties who want to join in on the computation, instead of requiring that the parties halt the computation and then engage in a new round of pre-processing. This would only make sense if the parties joining the computation trusted at least one of the pre-existing parties, which is likely to be the case in any reasonable application of this use-case.

At its heart our technique can be described as follows, where we let $\mathcal{F}_{\mathrm{Prep}}^{P,A}$ denote the SPDZ offline functionality for a set of parties P of size n with set of

corrupt parties A. We suppose we now have a set of parties $\mathbb{P} = R \cup Q$ and a set of parties which are considered corrupt $\mathbb{A} \subset \mathbb{P}$.

We then define a cover $\{Q_i\}_{i \in R}$ of Q such that there is at least one pair (i, Q_i) where $i \in R$ is an honest party in R, and Q_i contains at least one honest party from Q. The cover is such that each party in Q_i is connected by a *secure* channel to the associated party i in R. Just as (i, Q_i) associated a subset $Q_i \subset Q$ to a party $i \in R$, we also let (j, R_j) denote the subset $R_j \subset R$ associated to a party $j \in Q$.

We then extend $\mathbb{A}$ to a set $\overline{\mathbb{A}} \subset \mathbb{P}$ as follows $\overline{\mathbb{A}} = \mathbb{A} \cup \{j : R_j \subset \mathbb{A}\}$. The set $\overline{\mathbb{A}}$ is referred to as the set of *effectively corrupt honest parties* with respect to the online phase of the protocol. The protocol we present allows $\mathcal{F}_{\text{Prep}}^{Q,Q \cap \overline{\mathbb{A}}}$ to be implemented in the $\mathcal{F}_{\text{Prep}}^{R,R \cap \mathbb{A}}$-hybrid model.

The main idea of the protocol is conceptually quite simple, and is essentially a standard 're-sharing' technique similar to [BOGW88]. The main novelty is in showing that this can be efficiently applied to the SPDZ protocol, without the need for any expensive zero-knowledge proofs. In doing this, the difficulty comes in proving that the protocol is actually secure in the UC framework, and also in creating and analysing an (efficient) algorithm for assigning a cover to the network so that the adversary can only win with negligible probability in the security parameter in the case where we randomly assign the covers.

Related work. There is a long line of works on scalable secure computation with a large number of parties [DI06, HN06, DKMS14, BCP15] (to name a few), which use similar techniques to ours. These works often divide the parties into random *committees* (or *quorums*) to distribute the workload of the computation.

Most of these papers target asymptotic efficiency, and strong models such as adaptive security, asynchronicity and RAM computation. This gives interesting theoretical results, but the practicality of these techniques has not been demonstrated. In contrast, our work focuses on applying simple techniques to to modern, practical MPC protocols. Furthermore, we give a concrete analysis and examples of parameters that can be used for different numbers of parties in real-world settings, at a given security level.

2 Preliminaries

In this section, we describe the notation used in subsequent sections, formally define *secure cover*, and give an overview of the SPDZ protocol, and the offline phase in particular.

General Concepts and Notation: Parties in the network are labelled i for $i \in [n] = \{1, ..., n\}$, where n is the total number of parties. We denote by $\mathbb{P}$ this complete set of parties and we split the network into two parts, which we call R and Q (so each is a subset of n). We let n_r (resp. n_Q) denote the number of parties in R (resp. Q). We let $\mathbb{A} \subset \mathbb{P}$ denote the indexing set of corrupt parties

in $\mathbb{P}$, and $\overline{\mathbb{A}}$ denote the superset of $\mathbb{A}$ which possibly contains additional honest parties in Q, called *effectively corrupt honest parties* from the introduction. We assume there is an authenticated communication channel between every pair of parties in Q and every pair of parties in R. We define a *secure cover* $\{Q_i\}_{i \in R}$ of Q by R in the following way:

Definition 1. *Let $[n]$ be the indexing set of a set of parties in a given network and suppose we are also given subsets $R, Q \subset [n]$ of sizes n_r and n_q respectively. Each party in the network is either corrupt or honest. We call a set $\{Q_i\}_{i \in R}$ of (not necessarily disjoint) subsets of Q a* secure cover *if the following hold:*

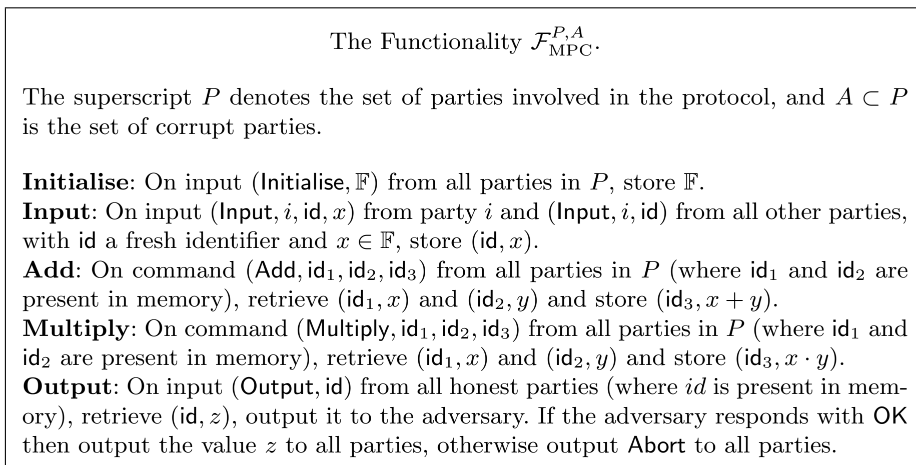
- *All parties in Q_i are connected to player $i \in R$ via a* private *channel.*
- *The subsets cover Q, i.e. $Q = \bigcup_{i \in R} Q_i$.*
- *There is* at least one *pair (i, Q_i) where $i \in R$ is an honest party in R, and Q_i contains* at least one *honest party from Q.*

We will also let R_j denote the set of parties in R which are connected to party $j \in Q$. We will use λ to denote the security parameter, and we will say an event occurs with overwhelming probability in the security parameter λ if it occurs with probability at least $1 - 2^{\lambda}$. We denote by $\mathbb{F}_q$ the finite field of order q, a (large) prime power. A function $\nu \in \mathbb{F}_q[x]$ is called negligible if for every polynomial $p \in \mathbb{F}_q[x]$, there exists a $C \in \mathbb{F}_q$ such that $\nu(x) \leq 1/p(x)$ for all $x > C$. We write $\alpha \leftarrow \mathbb{F}_q$ to mean that α is sampled uniformly at random from the field $\mathbb{F}_q$.

In Appendix B, we discuss the different network topologies of secure channels between our parties in R and parties in Q. In particular, we explore the different ways in which to define the cover $\{Q_i\}_{i \in R}$, taking into account, for example, the fact that the Q_i's are not necessarily all the same size. Section 4 then builds on these considerations by providing concrete methods of creating the cover and analysing the resulting protocols. This involves, for example, examining how the likelihood of the cover being secure changes (if we define it probabilistically) as we change the value of ℓ if we require that all parties in R send to the same number ℓ of parties in Q.

In Appendix C, we give a brief overview of the Universally Composability (UC) framework, which is the model in which we give the proof of our main theorem. The power of UC is well demonstrated in the pre-processing model, since it allows us to split up the functionality into separate independent parts and prove security of separate protocols for each part such that when we execute the protocols together we are guaranteed they be secure, even though we only prove them secure individually. In this model, we define some functionality $\mathcal{F}_{\text{Prep}}$ for the pre-processing and a separate functionality $\mathcal{F}_{\text{Online}}$ for the online phase. A protocol is designed for each, Π_{Prep} and Π_{Online}, the protocol Π_{Prep} is shown to implement $\mathcal{F}_{\text{Prep}}$ securely, and finally Π_{Online} is shown to implement $\mathcal{F}_{\text{Online}}$ securely in the $\mathcal{F}_{\text{Prep}}$-hybrid model (meaning the simulator in this proof assumes the existence of $\mathcal{F}_{\text{Prep}}$). This is particularly useful in our situation where we only want to change how pre-processing is done since we only need to revamp the pre-processing, and can leave the online phase unchanged, avoiding the need to reprove security.

SPDZ Overview: In general, computation will be done over a finite field $\mathbb{F} = \mathbb{F}_q$ where q is a (large) prime power. The protocol called $\mathsf{MACCheck}$ in the SPDZ paper [DPSZ12] requires that the field be large enough to make MAC forgery unfeasible by pure guessing. In particular this means that $1/q$ is negligible in λ. For smaller finite fields, and in particular the important case of binary circuits, adaptions to the $\mathsf{MACCheck}$ protocol can be made; see [LOS14], for example. For this paper we will assume the simpler case of large q purely for exposition. The SPDZ MPC protocol allows parties to compute an arithmetic circuit on their combined secret input. More specifically, for an arbitrary set of parties P and a subset set of corrupt parties $A \subset P$, the SPDZ protocol implements the functionality $\mathcal{F}_{\text{MPC}}^{P,A}$ described in Figure 1 provided $P \setminus A \neq \emptyset$.



The Functionality $\mathcal{F}_{\text{MPC}}^{P,A}$.

The superscript P denotes the set of parties involved in the protocol, and $A \subset P$ is the set of corrupt parties.

Initialise: On input $(\mathsf{Initialise}, \mathbb{F})$ from all parties in P, store $\mathbb{F}$.
Input: On input $(\mathsf{Input}, i, \mathsf{id}, x)$ from party i and $(\mathsf{Input}, i, \mathsf{id})$ from all other parties, with id a fresh identifier and $x \in \mathbb{F}$, store (id, x).
Add: On command $(\mathsf{Add}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$ from all parties in P (where id_1 and id_2 are present in memory), retrieve (id_1, x) and (id_2, y) and store $(\mathsf{id}_3, x + y)$.
Multiply: On command $(\mathsf{Multiply}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$ from all parties in P (where id_1 and id_2 are present in memory), retrieve (id_1, x) and (id_2, y) and store $(\mathsf{id}_3, x \cdot y)$.
Output: On input $(\mathsf{Output}, \mathsf{id})$ from all honest parties (where id is present in memory), retrieve (id, z), output it to the adversary. If the adversary responds with OK then output the value z to all parties, otherwise output Abort to all parties.

Figure 1. The Functionality $\mathcal{F}_{\text{MPC}}^{P,A}$.

The main motivation for this paper is that the "standard" protocols which implement $\mathcal{F}_{\text{MPC}}^{P,A}$ in the pre-processing model, for some set of parties P and corrupt parties A, require a lot of work by the parties in P during pre-processing. Our goal is to implement $\mathcal{F}_{\text{MPC}}^{P,A}$ using a (possibly larger) set of parties in which some specified set of parties execute the expensive pre-processing part of the protocol and only the parties in P (who are the only ones interested in the computation itself) execute the cheap online part of the protocol. In our terminology, the parties in Q outsource the pre-processing to a set of parties R (which possibly includes some parties in R) and then compute using this data.

We will elaborate a little here; in what follows we use the notation and functionalities of the latest version of the SPDZ protocol, based on OT, called MASCOT [KOS16]. We will describe the SPDZ protocol for an arbitrary set of parties P (later we will specialise this in our protocol to the sets Q and R in specific instances). In the initialisation stage, the parties sample (and keep private) random shares α_i, one for each party, whose sum is taken to be a global (secret) MAC key α, i.e. $\alpha = \sum_{i \in P} \alpha_i$.

A value $x \in \mathbb{F}_q$ is secret shared among the parties in P by sampling $(x_i)_{i \in P} \leftarrow \mathbb{F}_q^{|P|}$ subject to $x = \sum_{i \in P} x_i$, with party i holding the value x_i. In addition, we sample $(\gamma(x)_i)_{i \in P} \leftarrow \mathbb{F}_q^{|P|}$ subject to $\sum_{i \in P} \gamma(x)_i = \alpha \cdot x$ and party i holding the share $\gamma(x)_i$. Thus $\gamma(x)_i$ is a sharing of the MAC $\gamma(x) := \alpha \cdot x$ of x. We write the following to denote that x is a secret value, where party $i \in P$ holds x_i and $\gamma(x)_i$.

$$\langle x \rangle := ((x_i)_{i \in P}, (\gamma(x)_i)_{i \in P}))$$

Since this sharing scheme is linear, linear operations on secret values comes "for free", in the sense that adding secret values or multiplying them by a public constant requires no communication. Crucially, since the MAC is linear, the same operations applied to the corresponding MAC shares will result in MACs on the result of the said linear computation.

Unfortunately, multiplication of secret values requires a little more work, and is the reason we must generate data offline. At its heart SPDZ uses Beaver's method [Bea96] to multiply secret-shared values, which we outline here. In the offline phase, we generate a large number of multiplication triples, which are triples $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$ such that $c = a \cdot b$. Note that while other forms of pre-processing can help in various computations, such as shared squares and shared bits, in this paper we focus on the basic form of pre-processing and leave the interested reader to consult [DKL+13] and [KSS13]. To multiply secret-shared elements $\langle x \rangle$ and $\langle y \rangle$ in the online phase, we take a triple $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$ and partially open $\langle x \rangle - \langle a \rangle$ and $\langle x \rangle - \langle b \rangle$ to obtain $\varepsilon := \langle x \rangle - \langle a \rangle$ and $\delta := \langle x \rangle - \langle b \rangle$. By "partially open $\langle x \rangle - \langle a \rangle$", we mean that each party i sends the value $x_i - a_i$ to every other party, but does not send the corresponding MAC share. Then

$$\langle z \rangle = \langle c \rangle + \varepsilon \cdot \langle b \rangle + \delta \cdot \langle a \rangle + \varepsilon \cdot \delta$$

is a correct secret sharing of $z = x \cdot y$. A similar use of pre-processed data is used for the parties to enter their inputs into the computation.

SPDZ Preprocessing: To formalise things a little more, we now discuss the functionality $\mathcal{F}_{\text{Prep}}^{P,A}$, given in Figure 2, which implements the necessary pre-processing. The superscript P denotes the indexing set of parties involved in the computation, and set $A \subset P$ is a set of parties in P under the control of the adversary. As explained in the introduction, this includes all corrupt parties and also *effectively corrupt parties*, which are those nominally honest parties which receive reshares from only corrupt parties. If the parties generate the pre-processing themselves and do not make use of our protocol, this set A is exactly the set of corrupt parties; when the pre-processing is outsourced, then we have to worry about (the possibility of) effectively corrupt parties.

The functionality is a little more general than the functionality presented in [KOS16] as we allow the corrupt parties to introduce more errors: the standard SPDZ offline functionality only allows errors to be introduced into the MAC shares and not the data shares, whereas this new functionality allows errors on

The Offline Functionality $\mathcal{F}_{\text{Prep}}^{P,A}$ for SPDZ.

The set $A \subset P$ contains both the corrupt parties in P and also any honest parties we wish to consider as corrupt. This is to account for the fact that honest parties which have shares sent to it by corrupt parties only should be considered adversarially controlled. When A is exactly the set of corrupted parties in P, the functionality is the same as SPDZ (with the adversary's additional power to add errors into the secret values contained in shares as well as the MACs on them).

Initialise: On input $(\mathsf{Initialise}, q)$ from all players and the adversary, the functionality does the following:

1. The functionality samples $\alpha \leftarrow \mathbb{F}_q$ to be the global MAC key.
2. The functionality receives some error Δ_α from the adversary and, for each corrupted player $i \in A$, a share α_i.
3. It then samples at random α_i for each $i \notin A$ subject to $\sum_{i \in P} \alpha_i = \alpha + \Delta_\alpha$.
4. The functionality sends α_i to party i, for all $i \in P$.

Macro: $\mathsf{Angle}(x)$ The following will be run by the functionality at several points to create $\langle \cdot \rangle$ representations.

1. The functionality accepts $(\{x_i, \gamma(x)_i\}_{i \in A}, \Delta_x, \Delta_\gamma)$ from the adversary.
2. The functionality samples at random $\{x_i, \gamma(x)_i\}_{i \notin A}$ subject to $\sum_{i \in P} x_i = x + \Delta_x$ and $\sum_{i \in P} \gamma(x)_i = \alpha \cdot x + \Delta_\gamma$.
3. Finally, the macro returns $((x_i)_{i \in P}, (\gamma(x)_i)_{i \in P})$.

Computation: On input $(\mathsf{DataGen}, \mathit{DataType})$ from all players and the adversary, it executes the data generation procedures specified below.

- On input $\mathit{DataType} = \mathsf{InputPrep}$ and a value $i \in P$,
  1. The functionality samples $r^{(i)} \leftarrow \mathbb{F}_q$ if $i \notin A$, otherwise the functionality accepts $r^{(i)}$ from the adversary.
  2. The parties calls $\mathsf{Angle}(r^{(i)})$.
  3. If $i \notin A$, the functionality sends party i the values $\{r^{(i)}, (r_i^{(i)}, \gamma(r^{(i)})_i)\}$. (I.e. three values.)
  4. For $j \in P \setminus A$ and $j \neq i$ the functionality sends party j the pair $(r_i^{(j)}, \gamma(r^{(j)})_i)$.

  Thus all parties obtain a sharing $\langle r^{(i)} \rangle$ of a value $r^{(i)}$ known only to party i.
- On input $\mathit{DataType} = \mathsf{Triple}$,
  1. The functionality samples $a, b \in \mathbb{F}_p$ and computes $c = a \cdot b$.
  2. The functionality calls $\mathsf{Angle}(a)$, $\mathsf{Angle}(b)$ and $\mathsf{Angle}(c)$.
  3. For $i \notin A$, the functionality sends $((a_i, \gamma(a)_i), (b_i, \gamma(b)_i), (c_i, \gamma(c)_i))$ to $i \in P$.

Figure 2. The Offline Functionality $\mathcal{F}_{\text{Prep}}^{P,A}$ for SPDZ.

both. It is fairly intuitive that we will retain a security using this functionality as opposed to the standard one, as an adversary winning having changed shared values and MACs needs to have forged the same MAC equation as an adversary winning after just altering MAC values. The extra ability of altering share values gives him no advantage, a fact which we will prove shortly.

In [KOS16] the following theorem is (implicitly) proved, where $\mathcal{F}_{\text{OT}}$ and $\mathcal{F}_{\text{Rand}}$ are functionalities implementing OT and shared randomness for the parties.

Theorem 1. *There is a protocol $\Pi_{Prep}^{P,A}$ that securely implements $\mathcal{F}_{Prep}^{P,A}$ against static, active adversaries in the $\mathcal{F}_{OT}, \mathcal{F}_{Rand}$-hybrid model, where P is the complete set of parties and A the set of corrupt parties in P.*

We do not give the definition of the $\Pi_{\text{Prep}}^{P,A}$ protocol here as it is identical to MASCOT when based on OT, or identical to the original SPDZ pre-processing when based on homomorphic encryption (in spite of the slight difference in functionalities). Note that the paper [KOS16] proves the above theorem by giving a number of different protocols which, when combined, securely implement the required functionality $\mathcal{F}_{\text{Prep}}^{P,A}$.

The $\mathsf{MACCheck}$ Protocol from SPDZ/MASCOT.

On input an opened value s, a MAC share $\gamma(s)_i$ and a MAC key share α_i from each party i and a session id sid, each party i does the following:

1. Compute $\sigma_i \leftarrow \gamma(s)_i - s \cdot \alpha_i$ and call $\mathcal{F}_{\text{Commit}}.\mathsf{Commit}(\sigma_i, i, \mathsf{sid})$ to commit to this, and receive the handle τ_i.
2. When commitments are performed by all parties call $\mathcal{F}_{\text{Commit}}.\mathsf{Open}(i, \mathsf{sid}, \tau_i)$ to open the commitments.
3. If $\sum_{i=1}^{n} \sigma_i \neq 0$, output $\perp$ and abort; otherwise, continue.

Figure 3. The $\mathsf{MACCheck}$ Protocol from SPDZ/MASCOT.

SPDZ Online Protocol: The SPDZ online protocol is given in Appendix A as Figure 7 which itself uses the subprocedure $\mathsf{MACCheck}$ presented in Figure 3, which itself makes use of a commitment functionality given in Figure 8 (also in the appendix). It has been shown that UC commitment schemes in the plain model cannot exist, though they do exist in the *common reference string model* (in which one assumes the existence of common string known to both parties) [CF01], or, alternatively, the *random oracle model* (e.g. [HMQ04]).

The MAC check passes if the MAC is correct for the corresponding share. Importantly, the check fails if the MAC is incorrect for the shared value, which occurs if the MAC *or* the value it authenticates (or both) is incorrect. Proofs can be found in [KOS16, App. B] and [DPSZ12, App. D3]. It is precisely because $\mathsf{MACCheck}$ detects errors in either the MAC value or share value or both that we can use an offline phase which introduces errors into the share values themselves, and not restrict ourselves to an offline phase in which only errors on MACs are allowed (as in the original SPDZ papers). Given these procedures we can then prove:

Theorem 2. *The protocol $\Pi_{Online}^{P,A}$ securely implements the functionality $\mathcal{F}_{MPC}^{P,A}$ in the $\mathcal{F}_{Prep}^{P,A}, \mathcal{F}_{Commit}, \mathcal{F}_{Rand}$-hybrid model.*

Proof. The proof is identical to that in [DPSZ12], except that the pre-processing may now introduce errors into the share values as well as the MAC values. To prove the theorem, we must show that no environment can distinguish between an adversary interacting as in the protocol $\Pi_{\text{Prep}}^{P,A}$ and a simulator interacting with the functionality $\mathcal{F}_{\text{Prep}}^{P,A}$. Thus the proof runs exactly as in [DPSZ12, App.

D3], except that when we run the MACCheck protocol, the error can now be on the value in the share or the MAC. However, the security game presented in [DPSZ12] allowed the adversary to introduce errors on the shares, so the original protocol already offers the stronger guarantee that no error can occur on either the MAC or the value of the share it authenticated (or both). Note that if the adversary can alter the share *and* the MAC and have MACCheck pass, this is equivalent to computing, on some corrupted party's share, $a_i' \leftarrow a_i + c$ and $\gamma(a)_i' \leftarrow \gamma(a)_i' + \alpha \cdot c$ for some $c \in \mathbb{F}$, which still implies it knows α, the global MAC key.

3 Feeding One Protocol From Another

In this section we give our main result on feeding an instance of $\mathcal{F}_{\text{Prep}}^{Q,Q\cap\overline{\mathbb{A}}}$ from another instance $\mathcal{F}_{\text{Prep}}^{R,R\cap\mathbb{A}}$. We assume we have a *secure cover* $\{Q_i\}_{i\in R}$ of Q.

Method of Redistributing Data: Recall the parties in R will be performing the offline phase on behalf of the parties in Q. The parties in Q will share data in the standard manner (see Section 2), and the same will happen for parties in R. To avoid confusion, a data item $x \in \mathbb{F}$ secret shared amongst partes in Q will be denoted by $\langle x\rangle_Q$, whilst the same data item shared amongst parties in R will be denoted by $\langle x\rangle_R$, where implicitly we are assuming the same MAC key α is shared amongst the parties in R and the parties in Q.

When parties in Q want to evaluate a circuit amongst themselves, they follow the online protocol above and whenever they require a pre-processed data-item, they will ask R to provide one[2]. Thus we simply require a methodology to translate $\langle x\rangle_R$ sharings into $\langle x\rangle_Q$ sharings. Recall a shared value in the network R is denoted by

$$\langle x\rangle_R = ((x_i)_{i\in R}, (\gamma(x)_i)_{i\in R})$$

The principal idea of the protocol is, for each $i \in R$, to take the value x_i held by i and sample a set $\{x_i^j\}_{j\in Q_i}$ subject to $x_i = \sum_{j\in Q_i} x_i^j$ so that

$$\sum_{i\in R} x_i = \sum_{i\in R}\sum_{j\in Q_i} x_i^j = \sum_{j\in Q}\sum_{i\in R_j} x_i^j = \sum_{j\in Q} x^j$$

which holds because, by definition,

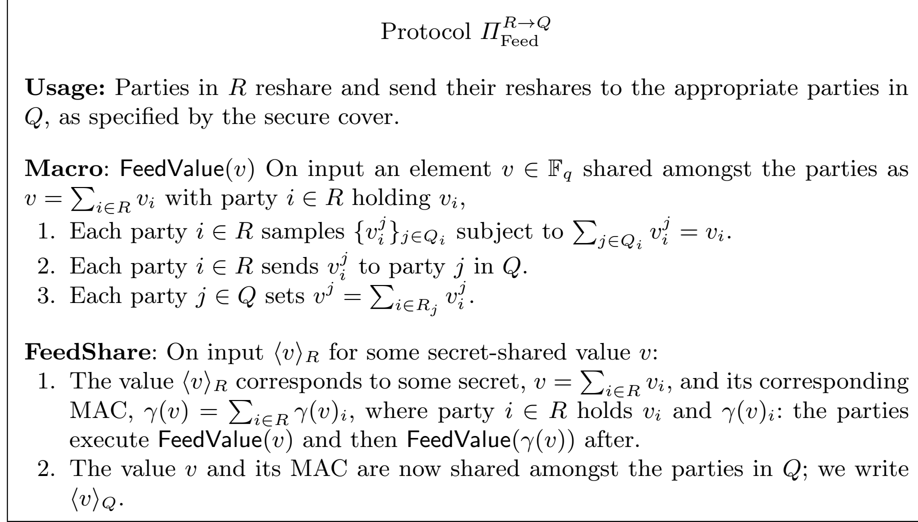
$$\{(i,j) : i \in R,\ j \in Q_i\} = \{(i,j) : j \in Q, i \in R_j\}.$$

If we do the same for the MAC shares, and at initialisation also share the global MAC key α in the same way, we obtain the same secret value x under the same global MAC key but shared instead amongst the parties in Q, which we denote by

$$\langle x\rangle_Q = ((x^j)_{j\in Q}, (\gamma(x)^j)_{j\in Q}).$$

[2] Of course, Q could ask R for these to be obtained all in one go in a form of outsourced pre-processing.

It is hopefully now clear how to define a feeding protocol to send shares from the R parties to the Q parties. We give the protocol in Figure 4.



Protocol $\Pi_{\text{Feed}}^{R \to Q}$

Usage: Parties in R reshare and send their reshares to the appropriate parties in Q, as specified by the secure cover.

Macro: $\mathsf{FeedValue}(v)$ On input an element $v \in \mathbb{F}_q$ shared amongst the parties as $v = \sum_{i \in R} v_i$ with party $i \in R$ holding v_i,

1. Each party $i \in R$ samples $\{v_i^j\}_{j \in Q_i}$ subject to $\sum_{j \in Q_i} v_i^j = v_i$.
2. Each party $i \in R$ sends v_i^j to party j in Q.
3. Each party $j \in Q$ sets $v^j = \sum_{i \in R_j} v_i^j$.

FeedShare: On input $\langle v \rangle_R$ for some secret-shared value v:

1. The value $\langle v \rangle_R$ corresponds to some secret, $v = \sum_{i \in R} v_i$, and its corresponding MAC, $\gamma(v) = \sum_{i \in R} \gamma(v)_i$, where party $i \in R$ holds v_i and $\gamma(v)_i$: the parties execute $\mathsf{FeedValue}(v)$ and then $\mathsf{FeedValue}(\gamma(v))$ after.
2. The value v and its MAC are now shared amongst the parties in Q; we write $\langle v \rangle_Q$.

Figure 4. Protocol $\Pi_{\text{Feed}}^{R \to Q}$

It is important to note that honest parties use incoming shares in an entirely deterministic manner; as such, observe that if some party $j \in Q$ is honest but it receives shares from only corrupt parties in R, the adversary has complete control over what this party's share will look like. For this reason, we consider them as effectively corrupt, contained in the extended adversary set $\overline{\mathbb{A}}$. This is why in the online protocol, run by the parties in Q, we also need to consider the set $\overline{\mathbb{A}}$. Note, a nominally honest party in $\overline{\mathbb{A}}$ which also lies in R will behave honestly when running the offline phase. Thus we need to consider $R \cap \mathbb{A}$ and not $R \cap \overline{\mathbb{A}}$ when running the online phase amongst parties in Q.

Composing the Protocol with the Functionality: The idea of the protocol is to convert the pre-processing generated by the parties in R to pre-processing that can be used by the parties in Q. Our goal, then, is to show that if the set of parties $R \cup Q$ is provided with the functionality $\mathcal{F}_{\text{Prep}}^{R, R \cap \mathbb{A}}$ and the parties engage in the protocol $\Pi_{\text{Feed}}^{R \to Q}$ to send their pre-processing to the parties in Q, then this "looks the same" to the parties in Q as a functionality $\mathcal{F}_{\text{Prep}}^{Q, Q \cap \overline{\mathbb{A}}}$.

The precise way in which the functionality $\mathcal{F}_{\text{Prep}}^{R, R \cap \mathbb{A}}$ and protocol $\Pi_{\text{Feed}}^{R \to Q}$ compose to approximate $\mathcal{F}_{\text{Prep}}^{Q, Q \cap \overline{\mathbb{A}}}$ is given in Figure 5. To help make it clear that the composition should provide the same interface as the functionality $\mathcal{F}_{\text{Prep}}^{Q, Q \cap \overline{\mathbb{A}}}$, Figure 5 gives the composition in a directly comparable structure to $\mathcal{F}_{\text{Prep}}^{R, R \cap \mathbb{A}}$ as presented in Figure 2.

Definition of the Composition $\Pi_{\text{Feed}}^{R \to Q} \circ \mathcal{F}_{\text{Prep}}^{R, R \cap \mathbb{A}}$

Initialise: On input $(\mathsf{Initialise}, q)$ from all parties in Q, parties in R execute $\mathcal{F}_{\text{Prep}}^{R, R \cap \mathbb{A}}$.**Initialise** and then from $\Pi_{\text{Feed}}^{R \to Q}$.$\mathsf{FeedValue}(\alpha)$ to share the MAC amongst the parties in Q.

Macro: $\mathsf{Angle}(x)$ This is never called, since below all calls involving Angle are to $\mathcal{F}_{\text{Prep}}^{R, R \cap \mathbb{A}}$.$\mathsf{Angle}$.

Computation: On input $(\mathsf{DataGen}, DataType)$ from all players in Q,

- On input $DataType = \mathsf{InputPrep}$ and a value $j \in Q$,
  1. We call $\mathcal{F}_{\text{Prep}}^{R}$.**Computation**$(\mathsf{DataGen}, \mathsf{InputPrep})$ a total of $|R_j|$ times, once for each $i \in R_j$ which is given as input: this means each party $i \in R$ has a set of shares $\{\tilde{r}_i^{(k)}\}_{k \in R_j}$, and each party $i \in R_j$ additionally holds $\tilde{r}^{(i)}$.
  2. The parties in R then locally sum their shares to get $\tilde{r}_i := \sum_{k \in R_j} \tilde{r}_i^{(k)}$. Then we let $r^{(j)} := \sum_{i \in R} \tilde{r}_i = \sum_{k \in R_j} \tilde{r}^{(k)}$. Doing the same with the MACs gives us $\langle r^{(j)} \rangle_R$.
  3. Each party $i \in R_j$ sends its value $\tilde{r}^{(i)}$ to party $j \in Q$, who computes the sum, which is $r^{(j)}$ by definition. (We can think of this as running $\Pi_{\text{Feed}}^{R \to Q}$.**FeedValue** where each party $i \in R \setminus R_j$ sends a 0 to all parties in Q_i.)
  4. Finally, we run $\Pi_{\text{Feed}}^{R \to Q}$.**FeedShare** on $\langle r^{(j)} \rangle_R$.

  Thus all parties in Q obtain a sharing $\langle r^{(j)} \rangle_Q$ of a value $r^{(j)}$ known only to party $j \in Q$.
- On input $DataType = \mathsf{Triple}$ by parties in Q,
  1. The functionality $\mathcal{F}_{\text{Prep}}^{R, R \cap \mathbb{A}}$ samples $a, b \in \mathbb{F}_p$ and computes $c = a \cdot b$.
  2. The functionality runs $\mathcal{F}_{\text{Prep}}^{R, R \cap \mathbb{A}}$.$\mathsf{Angle}(a)$, $\mathcal{F}_{\text{Prep}}^{R, R \cap \mathbb{A}}$.$\mathsf{Angle}(b)$ and $\mathcal{F}_{\text{Prep}}^{R, R \cap \mathbb{A}}$.$\mathsf{Angle}(c)$.
  3. For $i \notin A$, the functionality sends $((a_i, \gamma(a)_i), (b_i, \gamma(b)_i), (c_i, \gamma(c)_i))$ to $i \in R$.
  4. The parties in R now run $\Pi_{\text{Feed}}^{R \to Q}$.**FeedShare** on $\langle a \rangle_R$, $\langle b \rangle_R$ and $\langle c \rangle_R$.

Figure 5. Definition of the Composition $\Pi_{\text{Feed}}^{R \to Q} \circ \mathcal{F}_{\text{Prep}}^{R, R \cap \mathbb{A}}$

Main theorem: Before we give the statement of the theorem, we briefly give some intuition as to why our construction gives us the desired security. We defined a cover to be secure if at least one honest party in R sends to at least one honest party in Q; the negation of this statement is that all honest parties in R send *only* to corrupt parties in Q. Observe that if we do *not* have a secure cover, then for every secret value v, the adversary either has share v_i (when $i \in R$ is corrupt), or all "reshares", $\{v_i^j\}_{j \in Q_i}$ (when $i \in R$ is honest but all $j \in Q_i$ are corrupt); using these shares and reshares, the adversary can construct v and hence he breaks secrecy: thus a secure cover is necessary. Conversely, a secure cover also suffices, since if the cover is secure, there is at least one party in $Q \setminus A$ (i.e. one effectively honest party in Q). This is sufficient for running the SPDZ preprocessing securely. Our main theorem is then given by the following (and we defer the proof to Appendix D).

Theorem 3. *The construction $\Pi_{Feed}^{R\to Q} \circ \mathcal{F}_{Prep}^{R,R\cap\mathbb{A}}$, where $\Pi_{Feed}^{R\to Q}$ converts an R-sharing to a Q-sharing as in Figure 4, securely implements the functionality $\mathcal{F}_{Prep}^{Q,Q\cap\overline{\mathbb{A}}}$ in the presence of static, active adversaries assuming a secure cover of Q is given and we augment the set A of corrupt parties in Q to include those honest parties in Q which are only sent to by corrupt parties in R.*

4 Making a Secure Cover

In our scenario for outsourcing we assumed three potential use cases. The first situation was where the subset R is contained in the set Q, in which case guaranteeing a secure cover is trivial; in the second case, each party in Q knows a subset of parties in R in which it thinks there is (at least) one honest party; and in the third case, no such knowledge is known. In this last scenario we have two choices: either to set each covering subset Q_i equal to the whole set Q, or to assign the players randomly to subsets of Q whose union is the whole. For communication efficiency, here we analyse this last possibility by giving an algorithm to produce the assignment, and working out the associated probability of obtaining a secure cover. We no longer at this point assume at most $n_q - 1$ dishonest parties in Q and $n_r - 1$ dishonest parties in R, since in such a situation essentially the best we can do is to set $Q_i = Q$ for all $i \in R$. Thus we assume that a certain proportion of the sets are corrupt. In particular, we assume t_q parties in Q are corrupt, and t_r parties in R. We set $\epsilon_r = t_r/n_r$ and $\epsilon_q = t_q/n_q$ to be the associated ratios.

Recall, when creating the secure cover, is necessary to ensure that at least one honest party in Q receives a share from at least one honest party in R with overwhelming probability in the security parameter λ. If this is not true, while the MAC checks ensure the adversary cannot tamper with the share (recall the MACs ensure correctness), he is able to reconstruct the share (i.e. the adversary will break secrecy).

To help with the analysis, and for efficiency and load-balancing reasons, we will assume that each party in R sends to the same number of parties $\ell \geq \lceil n_q/n_r \rceil$ in Q. Note, any assignment of sets to parties in R which covers Q where $\ell = t_q + 1$ is automatically secure, since every party in R necessarily sends to at least one honest party in Q. We will see how small ℓ can be to provide statistical security for a given security parameter.

To assign a cover randomly in such a situation we use the Algorithm in Figure 6. The high-level idea of the algorithm is the following:

1. For each party in Q, we assign a random party in R, until each party in R has $\lceil n_q/n_r \rceil$ parties in Q assigned to it (or, equivalently, until the sets of parties in Q assigned to parties in R forms a disjoint cover).
2. For each party in R, we assign random parties in Q until each party in R has ℓ total parties which it sends to.

Note that in practice, the parties may want to run this algorithm using a trusted source of randomness (such as a blockchain or lottery), or execute a coin-tossing protocol to generate the necessary randomness.

Algorithm for randomly assigning elements of a cover of Q to parties in R.

For ease of notation, we label parties in R as i_k for $k \in [n_r]$ and parties in Q as j_k for $k \in [n_q]$; then the output array M is a binary matrix with a 1 in the $(k,l)^{\text{th}}$ position if and only if i_k in R sends to j_l in Q.
Inputs: n_r, n_q, $n = n_r + n_q$, ℓ, and sets $R, Q \subset [n]$ whose disjoint union is $[n]$.
Outputs: Matrix $M \in \mathbb{F}_2^{n_r \times n_q}$.
Method: (Note that ℓ is a constant, whereas l is an index.)

1. Set $M[1..n_r,\ 1..n_q] \leftarrow \{\{0,0,\ldots,0\},\ldots,\{0,0,\ldots,0\}\}$
2. Set NoOfOnes$[1..n_r] \leftarrow \{0,\ldots,0\}$
3. For $l \in [n_q]$,
   - Do
     - $k \leftarrow \mathcal{F}_{\text{Rand}}([n_r])$
   - Until NoOfOnes$[k] < \lceil n_q/n_r \rceil$
   - $M[k,l] \leftarrow 1$, NoOfOnes$[k] \leftarrow$ NoOfOnes$[k] + 1$
4. For $k \in [n_r]$,
   (a) While NoOfOnes$[k] < \ell$,
      - Do
        - $l \leftarrow \mathcal{F}_{\text{Rand}}([n_q])$
      - Until $M[k,l] <> 1$,
      - $M[k,l] \leftarrow 1$, NoOfOnes$[k] \leftarrow$ NoOfOnes$[k] + 1$
5. Output matrix M.

Figure 6. Algorithm for randomly assigning elements of a cover of Q to parties in R.

The algorithm allows different parties in Q to receive from different numbers of parties in R, whilst parties in R always send to the same number of parties in Q. Over $\mathbb{Z}$, each row of the matrix we generate, M, sums to ℓ, whilst the array NoOfOnes records how many parties in Q the i_k^{th} party in R sends to. Step 3 assigns all parties in Q to a party in R: this is the part of the algorithm which ensures we have a cover. In fact, this is done in such a way that each party in R sends to the *same* number of parties in Q, namely $\lceil n_q/n_r \rceil$. The reason for doing this is that it lends itself better to analysis of relevant probabilities below. Step 4 assign parties in Q to parties in R at random until each party in R is assigned ℓ parties in Q.

In the worst case, there is only one honest party in each of R and Q. Since we ensure that each party in R is assigned the same number of parties, the probability we obtain a secure cover is given by:

$$1 - \Pr[\text{Every good party in R is assigned only dishonest parties in Step 3}]$$
$$\cdot \Pr[\text{Every good party in R is assigned only dishonest parties in Step 4}]$$

Then the probability that we obtain a secure cover is given by:

$$1 - \left(\frac{\binom{t_q}{\lceil n_q/n_r \rceil} \cdot \binom{t_q - \lceil n_q/n_r \rceil}{\lceil n_q/n_r \rceil} \cdot \ldots \cdot \binom{t_q - (n_r - t_r - 1)\lceil n_q/n_r \rceil}{\lceil n_q/n_r \rceil}}{\binom{n_q}{\lceil n_q/n_r \rceil} \cdot \binom{n_q - \lceil n_q/n_r \rceil}{\lceil n_q/n_r \rceil} \cdot \ldots \cdot \binom{n_q - (n_r - t_r - 1)\lceil n_q/n_r \rceil}{\lceil n_q/n_r \rceil}} \right) \cdot \left(\frac{\binom{t_q - \lceil n_q/n_r \rceil}{\ell - \lceil n_q/n_r \rceil}}{\binom{n_q - \lceil n_q/n_r \rceil}{\ell - \lceil n_q/n_r \rceil}} \right)^{n_r - t_r - 1}$$

After some simplification we find that this is equal to

$$1 - \frac{t_q! \cdot (n_q - (n_r - t_r - 1)\lceil n_q/n_r \rceil)!}{n_q! \cdot (t_q - (n_r - t_r - 1)\lceil n_q/n_r \rceil)!} \cdot \left(\frac{\binom{t_q - \lceil n_q/n_r \rceil}{\ell - \lceil n_q/n_r \rceil}}{\binom{n_q - \lceil n_q/n_r \rceil}{\ell - \lceil n_q/n_r \rceil}} \right)^{n_r - t_r - 1} \tag{1}$$

To see what happens in the extreme case where all but one party is corrupt in each of R and Q, we set have $t_q = n_q - 1$ and $t_r = n_r - 1$. Then the probability that we obtain a secure cover is given by

$$\frac{1}{n_r} + \left(1 - \frac{1}{n_r}\right) \cdot \frac{\binom{(n_q-1) - \lceil n_q/n_r \rceil}{\ell - \lceil n_q/n_r \rceil - 1}}{\binom{n_q - \lceil n_q/n_r \rceil}{\ell - \lceil n_q/n_r \rceil}} = \frac{1}{n_r} + \left(1 - \frac{1}{n_r}\right) \cdot \frac{\ell - \lceil n_q/n_r \rceil}{n_q - \lceil n_q/n_r \rceil} \approx \frac{\ell}{n_q}.$$

When ℓ is equal to n_q, i.e. each party in R sends to every party in Q, we obtain a secure cover. For any other choice of ℓ with this high proportion of corruptions, we do not obtain a sufficiently high probability of obtaining a secure cover.

When ℓ is at least t_q, the generated cover is secure by default (since each party necessarily sends to an honest party). If there are 5 parties in R of which at most 2 are corrupt, and 50 parties in Q of which at most 25 are corrupt, each party in R must be assigned $\ell = 23$ parties in Q to ensure at least one honest party in R sends to one honest party in Q with probability at least $1 - 2^{-80}$. So we have saved on communication overall, since to guarantee a secure cover, we need $\ell = 25$.

If we now consider the case where R is still 5 with at most 2 corruptions but now we increase n_q to 1000 parties, of which at most 500 are corrupt, each party in R now need only send to 200 parties in Q (i.e. the same number as is required for a cover) to get a secure cover with overwhelming probability. In fact, we can even allow 3 parties in R and 750 parties in Q to be corrupt and keep $\ell = 200$ and still get 80 bits of security.

For the data in Table 1, we fix the number of parties in R at 5, fix the number of allowable corruptions to be at most 3, and compute the lower bound on the size of Q (i.e. on n_q) to guarantee that the adversary cannot win even where ℓ is fixed as the smallest number of connections necessary to make $\{Q_i\}_{i \in R}$ to cover Q, and vary the number of corruptions we allow in Q. In other words, ℓ need be no larger to provide 80-bit security than it need be for enabling the partition to be an exact cover (i.e. each party in Q sent to by at most one party in R).

n_r	t_r	t_q/n_q	Min. n_q for $\lambda = 80$ and $\ell = \lceil n_q/n_r \rceil$
5	3	1/2	336
5	3	1/3	201
5	3	1/4	148
5	3	1/5	125

Table 1. We fix $n_r = 5$, $t_r = 3$ and vary the fraction of corruptions in Q; the last column in the table is the least n_q such that the cover is secure even if each party in R only sends to $\ell = \lceil n_q/n_r \rceil$ parties.

Acknowledgements

This work has been supported in part by ERC Advanced Grant ERC-2015-AdG-IMPaCT, by EPSRC via grant EP/N021940/1 and by Defense Advanced Research Projects Agency (DARPA) and Space and Naval Warfare Systems Center, Pacific (SSC Pacific) under contract No. N66001-15-C-4070.

References

BCP15. Elette Boyle, Kai-Min Chung, and Rafael Pass. Large-scale secure computation: Multi-party computation for (parallel) RAM programs. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *Advances in Cryptology – CRYPTO 2015, Part II*, volume 9216 of *Lecture Notes in Computer Science*, pages 742–762, Santa Barbara, CA, USA, August 16–20, 2015. Springer, Heidelberg, Germany.

BDOZ11. Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In Kenneth G. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 169–188, Tallinn, Estonia, May 15–19, 2011. Springer, Heidelberg, Germany.

Bea96. Donald Beaver. Correlated pseudorandomness and the complexity of private computations. In *28th Annual ACM Symposium on Theory of Computing*, pages 479–488, Philadephia, PA, USA, May 22–24, 1996. ACM Press.

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

BOGW88. Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th Annual ACM Symposium on Theory of Computing*, pages 1–10, Chicago, IL, USA, May 2–4, 1988. ACM Press.

Can00. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000. `http://eprint.iacr.org/2000/067`.

CF01. Ran Canetti and Marc Fischlin. Universally composable commitments. Cryptology ePrint Archive, Report 2001/055, 2001. `http://eprint.iacr.org/2001/055`.

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

DI06. Ivan Damgård and Yuval Ishai. Scalable secure multiparty computation. In Cynthia Dwork, editor, *Advances in Cryptology – CRYPTO 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 501–520, Santa Barbara, CA, USA, August 20–24, 2006. Springer, Heidelberg, Germany.

DKL$^+$13. Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS 2013: 18th European Symposium on Research in Computer Security*, volume 8134 of *Lecture Notes in Computer Science*, pages 1–18, Egham, UK, September 9–13, 2013. Springer, Heidelberg, Germany.

DKMS14. Varsha Dani, Valerie King, Mahnush Movahedi, and Jared Saia. Quorums quicken queries: Efficient asynchronous secure multiparty computation. In *ICDCN*, volume 8314 of *Lecture Notes in Computer Science*, pages 242–256. Springer, 2014.

DPSZ12. Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany.

HMQ04. Dennis Hofheinz and Jörn Müller-Quade. Universally composable commitments using random oracles. In Moni Naor, editor, *TCC 2004: 1st Theory of Cryptography Conference*, volume 2951 of *Lecture Notes in Computer Science*, pages 58–76, Cambridge, MA, USA, February 19–21, 2004. Springer, Heidelberg, Germany.

HN06. Martin Hirt and Jesper Buus Nielsen. Robust multiparty computation with linear communication complexity. In Cynthia Dwork, editor, *Advances in Cryptology – CRYPTO 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 463–482, Santa Barbara, CA, USA, August 20–24, 2006. Springer, Heidelberg, Germany.

KOS16. Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. Cryptology ePrint Archive, Report 2016/505, 2016. `http://eprint.iacr.org/2016/505`.

KSS13. Marcel Keller, Peter Scholl, and Nigel P. Smart. An architecture for practical actively secure MPC with dishonest majority. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 13: 20th Conference on Computer and Communications Security*, pages 549–560, Berlin, Germany, November 4–8, 2013. ACM Press.

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, Part II*, volume 8617 of *Lecture Notes in Computer Science*, pages 495–512, Santa Barbara, CA, USA, August 17–21, 2014. Springer, Heidelberg, Germany.

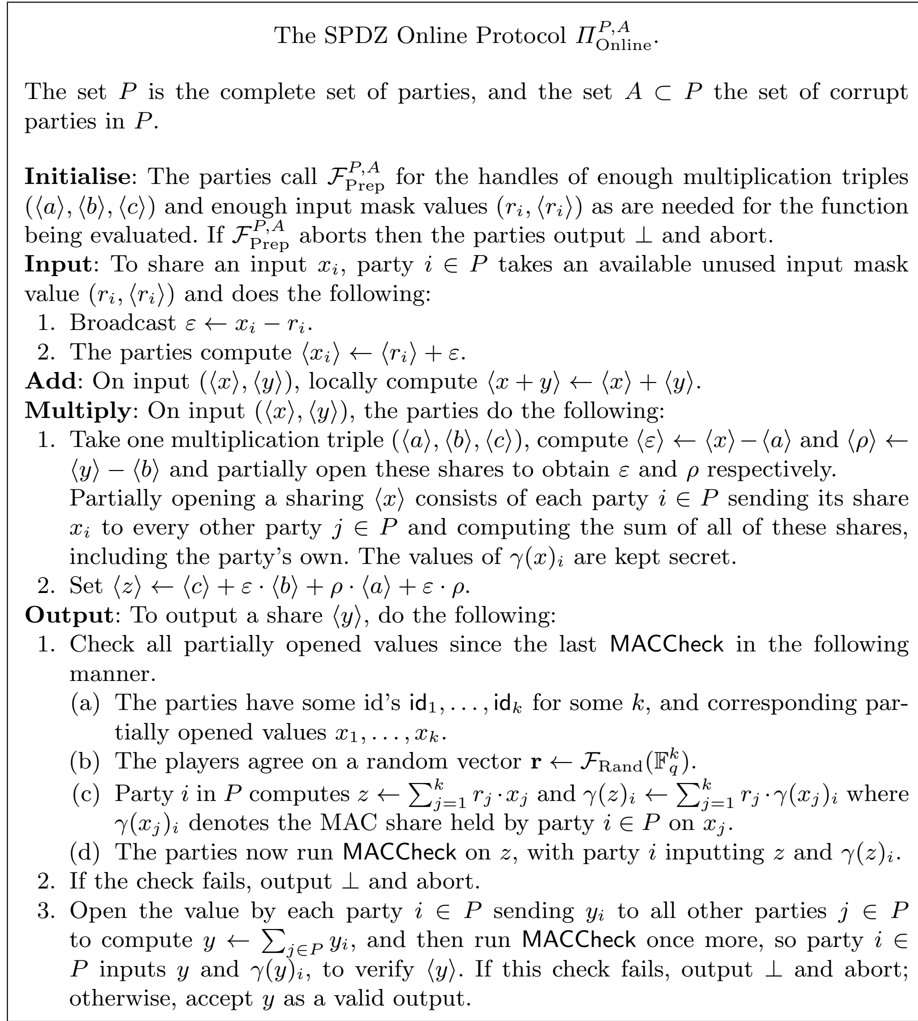
NNOB12. Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 681–700, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany.

RBO89. Tal Rabin and Michael Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority (extended abstract). In *21st Annual ACM*

Symposium on Theory of Computing, pages 73–85, Seattle, WA, USA, May 15–17, 1989. ACM Press.

A SPDZ Online Protocol

In Figure 7 we give the SPDZ online protocol. In Figure 8 we give the commitment functionality.



The SPDZ Online Protocol $\Pi_{\mathrm{Online}}^{P,A}$.

The set P is the complete set of parties, and the set $A \subset P$ the set of corrupt parties in P.

Initialise: The parties call $\mathcal{F}_{\mathrm{Prep}}^{P,A}$ for the handles of enough multiplication triples $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$ and enough input mask values $(r_i, \langle r_i\rangle)$ as are needed for the function being evaluated. If $\mathcal{F}_{\mathrm{Prep}}^{P,A}$ aborts then the parties output $\perp$ and abort.

Input: To share an input x_i, party $i \in P$ takes an available unused input mask value $(r_i, \langle r_i\rangle)$ and does the following:

1. Broadcast $\varepsilon \leftarrow x_i - r_i$.
2. The parties compute $\langle x_i\rangle \leftarrow \langle r_i\rangle + \varepsilon$.

Add: On input $(\langle x\rangle, \langle y\rangle)$, locally compute $\langle x+y\rangle \leftarrow \langle x\rangle + \langle y\rangle$.

Multiply: On input $(\langle x\rangle, \langle y\rangle)$, the parties do the following:

1. Take one multiplication triple $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$, compute $\langle \varepsilon\rangle \leftarrow \langle x\rangle - \langle a\rangle$ and $\langle \rho\rangle \leftarrow \langle y\rangle - \langle b\rangle$ and partially open these shares to obtain ε and ρ respectively. Partially opening a sharing $\langle x\rangle$ consists of each party $i \in P$ sending its share x_i to every other party $j \in P$ and computing the sum of all of these shares, including the party's own. The values of $\gamma(x)_i$ are kept secret.
2. Set $\langle z\rangle \leftarrow \langle c\rangle + \varepsilon \cdot \langle b\rangle + \rho \cdot \langle a\rangle + \varepsilon \cdot \rho$.

Output: To output a share $\langle y\rangle$, do the following:

1. Check all partially opened values since the last MACCheck in the following manner.
   (a) The parties have some id's $\mathsf{id}_1, \ldots, \mathsf{id}_k$ for some k, and corresponding partially opened values $x_1, \ldots, x_k$.
   (b) The players agree on a random vector $\mathbf{r} \leftarrow \mathcal{F}_{\mathrm{Rand}}(\mathbb{F}_q^k)$.
   (c) Party i in P computes $z \leftarrow \sum_{j=1}^{k} r_j \cdot x_j$ and $\gamma(z)_i \leftarrow \sum_{j=1}^{k} r_j \cdot \gamma(x_j)_i$ where $\gamma(x_j)_i$ denotes the MAC share held by party $i \in P$ on x_j.
   (d) The parties now run MACCheck on z, with party i inputting z and $\gamma(z)_i$.
2. If the check fails, output $\perp$ and abort.
3. Open the value by each party $i \in P$ sending y_i to all other parties $j \in P$ to compute $y \leftarrow \sum_{j \in P} y_i$, and then run MACCheck once more, so party $i \in P$ inputs y and $\gamma(y)_i$, to verify $\langle y\rangle$. If this check fails, output $\perp$ and abort; otherwise, accept y as a valid output.

Figure 7. The SPDZ Online Protocol $\Pi_{\mathrm{Online}}^{P,A}$.

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

Commit: On input $\mathsf{Commit}(v, i, \mathsf{sid})$ by party i, where v is the value to committed, sample a handle τ_v and send $(i, \mathsf{sid}, \tau_v)$ to all parties.
Open: On input $\mathsf{Open}(i, \mathsf{sid}, \tau_v)$ by party i, output $(v, i, \mathsf{sid}, \tau_v)$ to all parties. If some party P_i is corrupt and the adversary inputs $(\mathsf{Abort}, i, \mathsf{sid}, \tau_v)$, the functionality outputs $(\perp, i, \mathsf{sid}, \tau_v)$ to all parties.

Figure 8. Commitment Functionality $\mathcal{F}_{\text{Commit}}$.

B Communication between subnetworks

Here we discuss the topology of the network of secure channels between the subnetworks R and Q. Recall that Q has been partitioned into sets $\{Q_i\}_{i \in R}$ and party $i \in R$ assigned the set Q_i, and each Q_i is assumed to be of size ℓ. The topology depends primarily on the choice for the size ℓ of each set Q_i. We assume Q_i is the same size for all i, and note that obviously ℓ is lower-bounded by $\lceil n_q/n_r \rceil$ (so that $\ell \cdot n_r \geq n_q$), since $\{Q_i\}_{i \in R}$ together need to cover Q.

B.1 Complete

A naïve approach to connecting the two graphs with bilateral secure channels would be to form the complete bipartite graph between them (so $\ell = n_q$). This topology requires $n_r \cdot n_q$ secure connections and is shown in Figure 9.

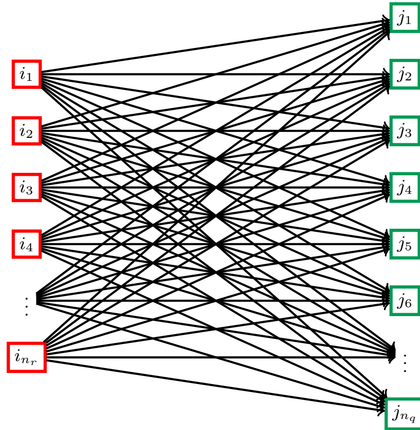


Fig. 9. Complete bipartite graph

If there is at least one honest party in each of R and Q then an adversary controlling any number of other parties still can never recover the MAC key. Unfortunately, there is a big communication overhead. Additionally, each party in R must compute n_q reshares for their share. If we assume the adversary is able to corrupt at most some t of the n total parties, we can clearly improve efficiency by instead requiring each party in R to send to $t+1$ parties in Q, since

then it is guaranteed each party, and in particular at least one honest party, sends to an honest party in Q.

B.2 Load-balanced

To aim for a load-balanced solution, we could instead ask each party $i \in R$ to reshare its share into $\ell = k \cdot \lceil n_q/n_r \rceil$ shares for some integer $k \geq 1$, and sending these to some set Q_i of ℓ parties in Q. If we have a secure cover, then the intuition is that there exist shares held by only honest players which are independent of all shares held by the adversary and are necessary for reconstructing the secret. This is discussed in more detail in the proof of our main theorem (see Appendix D. Figure 10 shows an example of our load-balanced topology for when $n_q \approx 2n_r$ and $k = 1$.

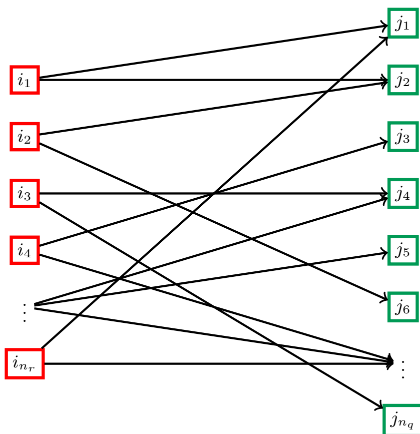


Fig. 10. Load-balanced topology

Note that it is not necessarily the case that each party in Q receive the same number of shares, even though we require each party in R to reshare to the same number of parties in Q.

C UC Model Overview

In this section, we give a brief overview of the UC framework, the model in which we proof our main theorem. Readers familiar with the model can skip this section.

Our proof is in the Universal Composability (UC) framework introduced by Canetti [Can00]. The model was introduced to enable protocols to be "composed", meaning that multiple different protocols (or multiple instances of the same protocol) can be run simultaneously such that the overall system is still "secure". There are some protocols which can offer no security if we permit multiple simulataneous executions. This framework works well in the pre-processing

model in MPC, as we split the circuit evaluation into two phases, the offline phase and the online phase; the overall protocol is the composition of these phases.

In this model, we compare executions in an ideal world with executions in the real world. In the real world, there is a set of honest parties who communicate in a protocol with a real-world adversary $\mathcal{A}$. In the ideal world, there is a set of honest parties and an ideal-world adversary called the simulator $\mathcal{S}$, but now these entities interact not with each other, but some trusted third party called a functionality $\mathcal{F}$, which takes inputs from all parties, computes on these data, and provides the output to all parties including the simulator.

The protocol between all parties in the real world somehow needs to resemble the interaction the parties have with the functionality in the ideal world. More formally, the real-world view of an environment, which provides inputs to all parties, sees all internal actions of the real-world adversary, and sees the outputs of all parties, needs to be indistinguishable from the ideal-world view.

To achieve this in "practice", the adversary engages in the protocol with the *simulator*, which extracts the adversary's inputs and forwards them on to the functionality, which then interacts with honest parties. If the protocol is designed correctly, the indistinguishability of the two views of the environment guarantees that the real-world adversary has no more power than the ideal-world adversary. In the design of the functionality, we limit the power of the adversary according to our security model: for example, if we want to allow the adversary to cause the interaction to abort without output, we design the functionality to allow it to accept an abort flag from the adversary and to halt when it receives it. The reason we make the functionality "weaker" than it could be (i.e. why we allow the adversary to have any control in the interaction at all) is because it may proven that no protocol exists which can stop an adversary doing some particular malicious behaviour. Since the simulator does not have any control over the honest parties and yet is supposed to simulate an execution of the protocol with the real-world adversary, the simulator has to do what it can to provide a view (i.e. messages from the alleged honest parties to the corrupt parties) close to what honest parties would actually send. In practice, this means it internally does what each honest party would do in the protocol and passes this on to the adversary.

The environment models any behaviour of the system in which the protocol is run. Proving the indistinguishability of the views shows there is no efficient attack strategy for breaking the protocol by running other protocols alongside it. If the environment cannot distinguish between the real- and ideal-world executions, the protocol is therefore secure even when run alongside any number of other protocols (as long as those are also proven secure in this model), or even arbitrarily many instances of the same protocol.

D Proof of Theorem 3

The proof is presented via a simulator (see Figure 11 and Figure 12) whose task is to overlay the functionality $\mathcal{F}_{\text{Prep}}^{Q,Q\cap\overline{\mathbb{A}}}$ so that no environment can distinguish whether the adversary is interacting as in $\Pi_{\text{Feed}}^{R\to Q} \circ \mathcal{F}_{\text{Prep}}^{R,R\cap\mathbb{A}}$, or with the simulator and $\mathcal{F}_{\text{Prep}}^{Q,Q\cap\overline{\mathbb{A}}}$. Figure 13 shows an outline of what each entity does during the simulation.

To do this we must show that the view of the environment in each case is the same. The view of the environment consists of the joint distribution of: the inputs and outputs of all parties (honest and corrupt), the adversary's internal state, and all messages the adversary sent and received.

Since it is used at several points throughout the functionality, we start by showing that the view of the environment when the simulator interacts with the functionality $\mathcal{F}_{\text{Prep}}^{Q,Q\cap\overline{\mathbb{A}}}$ during the SFeedValue operation is indistinguishable from its view when instead the real-world adversary is interacting with the composition $\Pi_{\text{Feed}}^{R\to Q} \circ \mathcal{F}_{\text{Prep}}^{R,R\cap\mathbb{A}}$.

By this point in the simulation, the adversary has a set of values $\{v_i\}_{i\in R\cap\mathbb{A}}$ and an error Δ_v^R; the simulator has the complete set $\{v_i\}_{i\in R}$ as well as the error Δ_v^R and the secret value v, since it was emulating the functionality $\mathcal{F}_{\text{Prep}}^{R,R\cap\mathbb{A}}$ when these shares were generated.

For this part of the simulation, there are no inputs for the environment to supply: the environment has already supplied inputs to the adversary and the honest parties for generating the shares and errors above.

The simulator waits for the adversary to send reshares of his shares of some shared value $\langle v\rangle_R$. The simulator then does some sampling in different ways, and computes $\{v^j\}_{j\in Q\cap\overline{\mathbb{A}}}$ and some error Δ_v^Q, which it passes on to the functionality $\mathcal{F}_{\text{Prep}}^{Q,Q\cap\overline{\mathbb{A}}}$. Meanwhile, it also returns to the adversary reshares of honest parties' inputs to corrupt parties. See Figure 12 for more detail.

The functionality sends uniformly randomly sampled elements $\{v^j\}_{j\in Q\setminus\overline{\mathbb{A}}}$ to honest parties in Q except those honest parties which receive from only corrupt parties, which we call *effectively corrupt*, which instead receive v^j computed as the sum of shares from the adversary, not uniformly randomly sampled elements.

The honest parties (including the effectively corrupt honest parties) output their shares to the environment. The environment cannot distinguish between the distributions because in the real world the honest parties' outputs are also uniformly random since they received at least one share from one honest party which was added into their reshare, and the effectively corrupt parties' outputs are exactly the sum of corrupt shares, just as the simulator computed for them to be in the ideal-world simulation.

Finally, observe that the messages the adversary sends and receives are identically distributed in the real and ideal worlds since the simulator computes reshares for honest parties exactly as honest parties compute them themselves in the real world.

Now we turn to the other procedures. In **Initialise**, the simulator acts exactly as the protocol would until the MAC is reshared via FeedValue, which we have just dealt with. However, note that when the simulator sends the call **Initialise** to the functionality $\mathcal{F}_{\text{Prep}}^{Q,Q\cap\overline{\mathbb{A}}}$, the functionality samples a global MAC key which is different from the global MAC key the simulator sampled, α, with high probability. We argue that this does not change the distribution.

The reason the distribution is the same in both cases is that the honest parties' shares in Q are uniformly random regardless of the inputs of the adversary, and the only way for the environment to determine a global MAC key is to reconstruct a value and its corresponding MAC from the shares of all corrupt and honest parties. The functionality samples all inputs for the generation of pre-processing, so the only way for the adversary to obtain the reconstruction of a share is for it to consider a value the functionality tells it: this is the case in the **Computation** phase with $DataType = \mathsf{InputPrep}$ input: the adversary is given $r^{(i)}$ for each corrupt $i \in R$ to use as input. If the adversary reshares honestly and passes this on to the simulator/honest parties, then at the end of the process the environment can reconstruct the value and the MAC on it. In the real world, the MAC is some α sampled by the functionality $\mathcal{F}_{\text{Prep}}^{R,R\cap\mathbb{A}}$; in the ideal world, the simulator runs this functionality internally and also samples some α. However, the functionality $\mathcal{F}_{\text{Prep}}^{Q,Q\cap\overline{\mathbb{A}}}$ samples some β for *its* global MAC key, which differs from α with high probability. While the environment can compute the value of the global MAC key, since in any case it is sampled uniformly, the environment cannot tell whether it was generated by the simulator emulating $\mathcal{F}_{\text{Prep}}^{R,R\cap\mathbb{A}}$ or by the functionality $\mathcal{F}_{\text{Prep}}^{Q,Q\cap\overline{\mathbb{A}}}$.

The procedures **Computation**, **Input Production** and **Multiplication Triples** are indistinguishable from the protocol if and only if the macro Angle is. The simulator behaves in the same way as the functionality $\mathcal{F}_{\text{Prep}}^{R,R\cap\mathbb{A}}$ in the real-world execution (remember we are in the $\mathcal{F}_{\text{Prep}}^{R,R\cap\mathbb{A}}$-hybrid model) for the macro Angle, and so the indistinguishability in the two worlds of FeedValue, gives us what we want.

Proof.

Simulator $\mathcal{S}_{\text{Prep}}^{Q,A}$

The simulator begins by first setting the set of corrupt parties to be the corrupt parties in R with the corrupt parties in Q, augmented to include all parties in Q which receive from only corrupt parties in R.

Initialise:

1. The simulator accepts $(\mathsf{Initialise}, q)$ from all parties in $R \cap A$, and then some error Δ_α^R and shares α_i for all $i \in R \cap A$ from the adversary. The simulator runs an internal copy of **Initialise** from $\mathcal{F}_{\text{Prep}}^{R,A}$, passing on the adversary's input, and thereby determines a global MAC key α and a set of shares α_i for honest parties, $i \in R \setminus A$. More explicitly,
   (a) The simulator samples $\alpha \leftarrow \mathbb{F}_q$ to be the global MAC key.
   (b) The simulator samples at random α_i for each $i \in R \setminus A$ subject to $\sum_{i \in R} \alpha_i = \alpha + \Delta_\alpha^R$, using the error and shares from the adversary.

   The simulator stores the global MAC key α, the shares for all parties, and the error, for later use.
2. The next part to simulate is the sending of the MAC key from R to Q via the protocol: the simulator receives the call $\mathsf{FeedValue}(\alpha)$ from the adversary to copy across the secret-shared MAC key α, at which point they perform the subroutine $\mathsf{SFeedValue}$ in Figure 12.
3. Finally, the simulator passes the shares α^j with $j \in Q \cap A$ along with the error Δ_α^Q computed in $\mathsf{SFeedValue}$ to the functionality $\mathcal{F}_{\text{Prep}}^{Q}$. The simulator receives back the same shares it sent to the functionality (corresponding to corrupt parties' shares), but the "honest" parties in Q who receive shares from only corrupt parties in R receive a share α^j which is not uniformly random, but instead is the sum of reshares from corrupt parties.

Computation: On input $\mathsf{DataGen}$ from the adversary, it executes the data generation procedures as in the functionality, with calls to Angle dealt with as the macro below describes.

Macro: $\mathsf{Angle}(x)$.

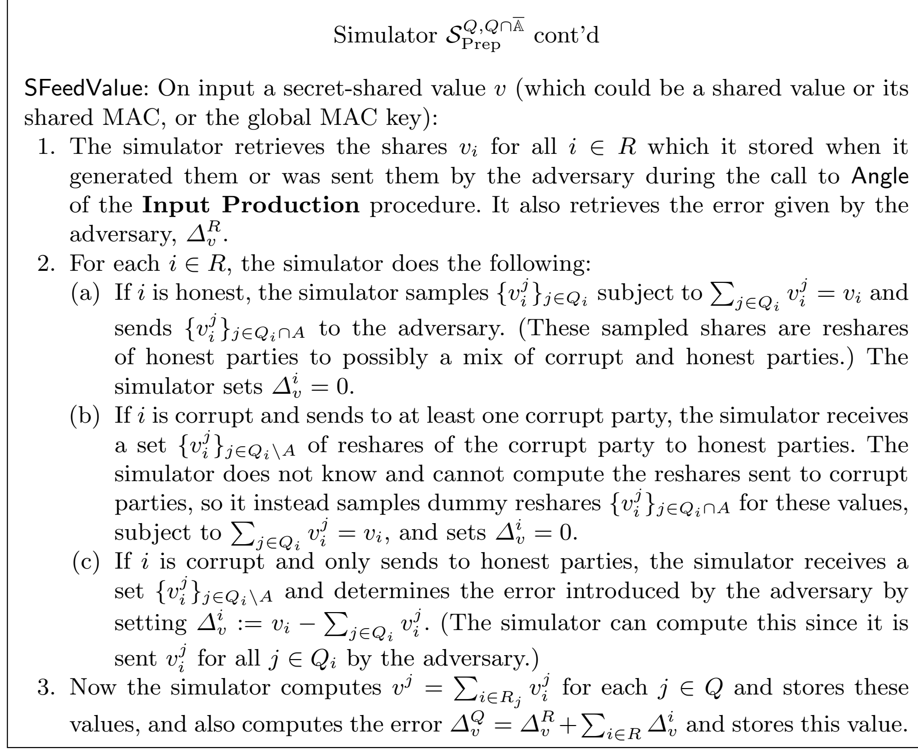
1. Emulating the macro in the functionality $\mathcal{F}_{\text{Prep}}^{R,A}$, the simulator accepts $(\{x_i, \gamma(x)_i\}_{i \in A \cap R}, \Delta_x^R, \Delta_\gamma^R)$ from the adversary.
2. The simulator samples at random $\{x_i, \gamma(x)_i\}_{i \in A \cap R}$ subject to $\sum_{i \in R} x_i = x + \Delta_x^R$ and $\sum_{i \in R} \gamma(x)_i = \alpha \cdot x + \Delta_\gamma^R$, just as the functionality does. The simulator stores the shares and errors for later use.
3. Now the simulator receives the command $\mathbf{FeedShare}(\langle x \rangle_R)$ from the adversary for $\langle x \rangle_R = ((x_i)_{i \in R}, (\gamma(x)_i)_{i \in R})$ in the pre-processed data, so the simulator runs $\mathsf{SFeedValue}$ on x and $\gamma(x)$ in turn.
4. Finally, the simulator takes the input values and MACs corresponding to the corrupt parties' inputs and the errors returned by $\mathsf{SFeedValue}$ and sends these as the adversary's input to the functionality $\mathcal{F}_{\text{Prep}}^{Q,A}$.

DataGen

Input Production: On input $DataType = \mathsf{InputPrep}$, the simulator runs as in the functionality, where the call to Angle is dealt with as above.

Multiplication Triples: On input $DataType = \mathsf{Triples}$, the simulator runs as in the functionality, where the call to Angle is dealt with as above.

Figure 11. Simulator $\mathcal{S}_{\text{Prep}}^{Q,A}$



Simulator $\mathcal{S}_{\text{Prep}}^{Q,Q\cap\overline{\mathbb{A}}}$ cont'd

SFeedValue: On input a secret-shared value v (which could be a shared value or its shared MAC, or the global MAC key):

1. The simulator retrieves the shares v_i for all $i \in R$ which it stored when it generated them or was sent them by the adversary during the call to Angle of the **Input Production** procedure. It also retrieves the error given by the adversary, Δ_v^R.
2. For each $i \in R$, the simulator does the following:
   (a) If i is honest, the simulator samples $\{v_i^j\}_{j\in Q_i}$ subject to $\sum_{j\in Q_i} v_i^j = v_i$ and sends $\{v_i^j\}_{j\in Q_i\cap A}$ to the adversary. (These sampled shares are reshares of honest parties to possibly a mix of corrupt and honest parties.) The simulator sets $\Delta_v^i = 0$.
   (b) If i is corrupt and sends to at least one corrupt party, the simulator receives a set $\{v_i^j\}_{j\in Q_i\setminus A}$ of reshares of the corrupt party to honest parties. The simulator does not know and cannot compute the reshares sent to corrupt parties, so it instead samples dummy reshares $\{v_i^j\}_{j\in Q_i\cap A}$ for these values, subject to $\sum_{j\in Q_i} v_i^j = v_i$, and sets $\Delta_v^i = 0$.
   (c) If i is corrupt and only sends to honest parties, the simulator receives a set $\{v_i^j\}_{j\in Q_i\setminus A}$ and determines the error introduced by the adversary by setting $\Delta_v^i := v_i - \sum_{j\in Q_i} v_i^j$. (The simulator can compute this since it is sent v_i^j for all $j \in Q_i$ by the adversary.)
3. Now the simulator computes $v^j = \sum_{i\in R_j} v_i^j$ for each $j \in Q$ and stores these values, and also computes the error $\Delta_v^Q = \Delta_v^R + \sum_{i\in R} \Delta_v^i$ and stores this value.

Figure 12. Simulator $\mathcal{S}_{\text{Prep}}^{Q,Q\cap\overline{\mathbb{A}}}$ cont'd

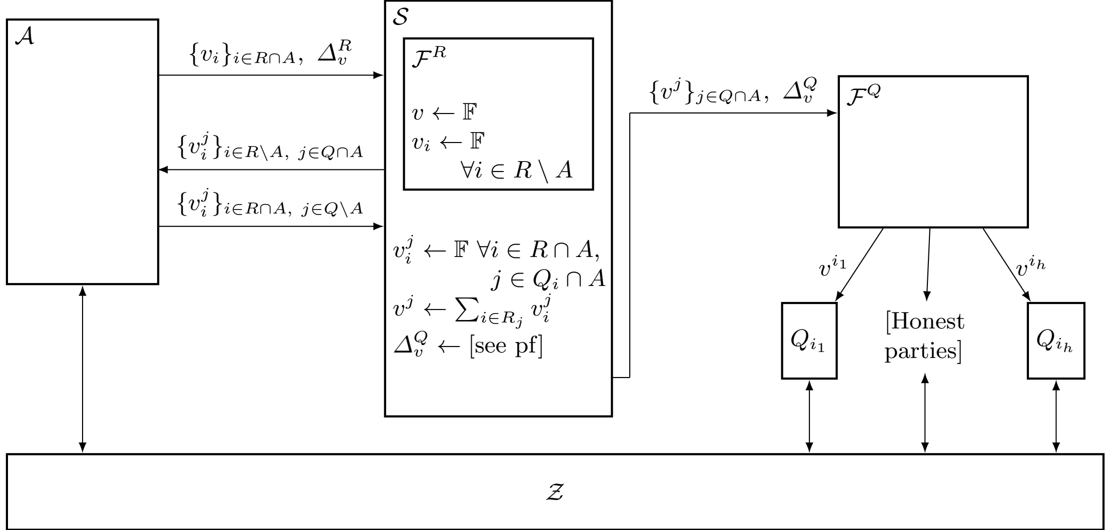


Fig. 13. Rough overview of messages and internal working of simulation