Distributing any Elliptic Curve Based Protocol: With an Application to MixNets

Nigel P. Smart[1,2] and Younes Talibi Alaoui[2]

[1] University of Bristol, Bristol, UK.
[2] KU Leuven, Leuven, Belgium.
nigel.smart@kuleuven.be,younes.talibialaoui@kuleuven.be

Abstract. We show how to perform a full-threshold n-party actively secure MPC protocol over a subgroup of order p of an elliptic curve group $E(K)$. This is done by utilizing a full-threshold n-party actively secure MPC protocol over $\mathbb{F}_p$ in the pre-processing model (such as SPDZ), and then locally mapping the Beaver triples from this protocol into equivalent triples for the elliptic curve. This allows us to transform essentially *any* one-party protocol over an elliptic curve, into an n-party one. As an example we show how to transform the shuffle protocol of Abe into an n-party protocol. This application requires us to also give an MPC protocol to derive the switches in a Waksman network from a generic permutation, which may be of independent interest.

1 Introduction

Over the years there have been a number of protocols developed for elliptic curves, starting with basic protocols such as encryption and signature, through to zero-knowledge proofs, and secure shuffles. In some application instances one wants to perform these protocols where the secret data of a party is not held by a single party but held by a set of parties via a secret sharing scheme. Obvious examples include distributed decryption and distributed signing protocols. Indeed the case of distributed signatures for EC-DSA has recently undergone a renewed interest, see [7,13–15], due to applications to block-chain. In addition, general distributed cryptographic solutions for decryption and signature operations are becoming more in vogue, as evidenced by the recent NIST workshop in this space[3].

There are however a large number of other protocols which applications may require to be distributed in this manner. For example take a simple elliptic curve based Sigma-protocol to prove equality of two discrete logarithms. If the application requires the two discrete logarithms to be secret shared, then the protocol to produce the proof must be executed in a distributed manner. In this work we present a *simple* method to produce n-party actively secure distributed elliptic curve based protocols.

We take the underlying elliptic curve as $E(K)$ where the (cryptographically interesting) subgroup order is a prime p. For such protocols we need to secret share both finite field elements in $\mathbb{F}_p$, and elliptic curve group elements in $E(K)$. In both cases we do this via an additive secret sharing scheme. Linear operations in $\mathbb{F}_p$ and in $E(K)$ are then able to be performed for free, and the problem then comes in performing the non-linear operations. For non-linear operations in $\mathbb{F}_p$ (i.e. multiplication) we utilize Beaver triples, and thus our protocol is in the offline/online paradigm. For non-linear operations in $E(K)$, which are point multiplications of a secret shared point by a secret shared field element, we can utilize *the same* Beaver triples. Thus supporting additively secret shared elements in $E(K)$ can be accomplished using *the same* offline phase as is needed to perform MPC over $\mathbb{F}_p$.

To achieve active security we utilize the methodology of the SPDZ protocol [6] and its improvements, e.g. [5]. In particular for each field element $x \in \mathbb{F}_p$ which is secret shared, we also secret share a MAC-value $\alpha \cdot x$ for some global secret shared MAC value α. This is then translated to the elliptic curve sharing by not only additively sharing an element $P \in E(K)$, but also additively sharing it's MAC value $[\alpha]P$, for *the same* MAC key α as used to authenticate the shares over $\mathbb{F}_p$.

[3] https://www.nist.gov/news-events/events/2019/03/nist-threshold-cryptography-workshop-2019.

The first part of this paper is devoted to giving the details of this MPC protocol over elliptic curves, and the associated security proofs. We then give two applications, the first to show how distributed EC-DSA signing can be performed using this protocol. The online time for this EC-DSA signing operation will be very fast, the only drawback being the offline time inherited from the SPDZ protocol for generating authenticated Beaver triples over $\mathbb{F}_p$. We also show how to perform a simple distributed Sigma protocol for an OR-proof, using the same methodology.

We then turn to a more complex application. In a number of applications one has a set of ElGamal ciphertexts and one wishes to perform a secure shuffle on them. The traditional method for doing this is to pass them through a sequence of so-called Mix-Nets. Each mixer applies their own private shuffle, and provides a zero-knowledge proof, that their mix has been performed correctly. The final recipient of the mix needs to verify *each* individual zero-knowledge proof. Thus if we have n-mixers we have a proof n times larger than that produced by a single mixer. Another way of achieving the same security, but with a smaller zero-knowledge proof, would be for the mixers to produce the mix in a distributed manner and generate a single *joint* proof of correctness of the mix.

We show how the mix protocol of Abe [1] can be performed in such a distributed manner using our underlying MPC protocol. Given a permutation as a Waksman network, with each Waksman switch secret shared, we show how to generate in a distributed manner the proof presented by Abe. This is essentially a more complex version of the Sigma protocol for equality of two discrete logarithms discussed earlier. We note, that more efficient proofs of correct shuffle have been given since Abe's work, see for example [3, 8–10], but we concentrate on this one as it shows how our elliptic curve MPC protocol can be applied to more complex higher level protocols.

A problem with Abe's mixer is how to generate the secret shared Waksman network. Simply generating the switch values at random does not produce a uniformly random permutation (as was noticed in [2]). We can easily produce a secret shared uniformly random permutation, by each party P_i generating a permutation σ_i, sharing it, and then using as the secret-shared final permutation the product permutation $\sigma = \sigma_1 \cdots \sigma_n$. However, the question then remains how to convert the secret-shared permutation σ, given by (say) a permutation matrix, into a set of switches for a Waksman network. There is a classical algorithm to do this, which appears to require solving a set of non-linear equations. However, by closely examining this algorithm we see that one can perform the conversion from a permutation matrix to Waksman switches using a relatively simple algorithm which is suitable for implementation via an MPC system. Our algorithm for obtaining the Waksman switches in secret shared form is actively secure, if the underlying MPC system used is actively secure (which is what we assume throughout this work).

We end this introduction by re-iterating that our MPC system is in the full threshold paradigm, where active security is obtained by authenticating a share using a global shared MAC key. We note that our methodology can also be applied in the case of Q2 access structures (for example honest majority threshold access structures) if we accept MPC-with-abort. In such systems one can obtain similar authentication of the shares by utilizing the error detection properties of the underlying error correcting code associated to the secret sharing scheme, see [16]. It can be easily seen that minor adaptations to our MPC protocol over elliptic curves will also enable one to support such Q2 access structures.

2 Preliminaries

In this section we present some basic notation and the underlying MPC protocols we will make extensive use of.

2.1 Notation

We assume that all the parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$ are probabilistic polynomial time Turing machines. We let $[n]$ denote the interval $[1, \ldots, n]$. We let $a \leftarrow X$ denote randomly assigning a value a from a set X, where we assume a uniform distribution on X. If A is an algorithm, we let $a \leftarrow A$ denote assignment of the output, where the probability distribution is over the random tape of A; we also let $a \leftarrow b$ be a shorthand for $a \leftarrow \{b\}$,

i.e. to denote normal variable assignment. If $\mathcal{D}$ is a probability distribution over a set X then we let $a \leftarrow \mathcal{D}$ denote sampling from X with respect to the distribution $\mathcal{D}$.

We let $\mathcal{G} \subseteq E(K)$ denote a subgroup of large prime order of an elliptic curve E over a finite field K. Let the order of $\mathcal{G}$ be p. Any (non-zero) element of $\mathcal{G}$ can be taken as a generator, however we assume that a specific generator P is given as part of the group description. An element $Q \in \mathcal{G}$ can be multiplied by an element $x \in \mathbb{F}_p$ to produce another element $R \in \mathcal{G}$. We call this operation the point multiplication between a point Q and a multiplier x, and we write it as $R \leftarrow [x] \cdot Q$.

2.2 The SPDZ Protocol

Our protocols will be built on top of a number of existing functionalities/protocols. The main two being an ideal commitment functionality (given in Appendix A) and the SPDZ MPC protocol [6] for performing actively secure MPC over $\mathbb{F}_p$ for full-threshold adversaries. The SPDZ protocol processes data using an authenticated secret sharing scheme defined over a finite field $\mathbb{F}_p$, where p is prime.

We describe the variant of authentication and checking presented in [5]. The secret sharing scheme is defined as follows: Each party $\mathcal{P}_i$ holds a share of a global MAC key $\alpha_i \in \mathbb{F}_p$, where the global MAC key is defined to be $\alpha = \sum_i \alpha_i$. A data element $x \in \mathbb{F}_p$ is held in secret shared form as a tuple $\{x_i, \gamma_i\}_{i \in [n]}$, such that $x = \sum_i x_i$ and $\sum \gamma_i = \alpha \cdot x$. We denote a value x held in such a secret shared form as $\langle x \rangle_F$. If we wish to denote the specific value on which γ_i is a MAC share then we write $\gamma_i[x]$.

Functionality $\mathcal{F}_{\mathsf{Online}}[\mathsf{SPDZ}]$

Initialize: On input $(init, p)$ from all parties, the functionality stores $(domain, p)$.

Input: On input $(input, \mathcal{P}_i, varid, x)$ from $\mathcal{P}_i$ and $(input, \mathcal{P}_i, varid, ?)$ from all other parties, with $varid$ a fresh identifier, the functionality stores $(varid, x)$.

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

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

Triple: On input $(triple, varid_1, varid_2, varid_3)$ from all parties (if none of the $varid_i$ are stored in memory), the functionality generates a uniformly random $a, b \in \mathbb{F}_p$ and computes $c = a \cdot b$ and then stores $(varid_1, a)$, $(varid_2, b)$ and $(varid_3, c)$

Output: On input $(output, varid, i)$ from all honest parties (if $varid$ is present in memory), the functionality retrieves $(varid, y)$ and outputs it to the environment. The functionality waits for an input from the environment. If this input is $\mathsf{Deliver}$ then y is output to all players if $i = 0$, or y is output to player i if $i \neq 0$. If the adversarial input is not equal to $\mathsf{Deliver}$ then $\varnothing$ is output to all players.

Figure 1. The ideal functionality for MPC over $\mathbb{F}_p$

The SPDZ protocol implements the functionality given in Figure 1. The SPDZ protocol works in an offline-online manner in which the offline functionality is defined in Figure 2 (the precise offline protocol will not concern us in this work). The main goal of the SPDZ offline phase is to produce random triples $(\langle a \rangle_F, \langle b \rangle_F, \langle c \rangle_F)$ such that $c = a \cdot b$. It is convenient in some protocols to assume that the Beaver triples produced in the offline phase are also available to the user of the online phase. Thus we have a command in the online functionality that exports this data.

The online protocol $\Pi_{\mathsf{Online}}[\mathsf{SPDZ}]$ is given in Figure 4, and can be shown (see e.g. [5]) that it realises the $\mathcal{F}_{\mathsf{Online}}[\mathsf{SPDZ}]$ functionality in the $(\mathcal{F}_{\mathsf{Offline}}[\mathsf{SPDZ}], \mathcal{F}_{\mathsf{Commit}})$-hybrid model. The online protocol makes use of a crucial sub-protocol, called $\Pi_{\mathsf{MACCheck}}[\mathsf{SPDZ}]$, which we have given in Figure 3.

Since $\langle \cdot \rangle_F$ is a linear secret sharing scheme it is easy to compute linear functions on the share values. In particular given $\langle x \rangle_F$ and $\langle y \rangle_F$ and three field constants $a, b, c \in \mathbb{F}_p$ we can compute the sharing of

The functionality $\mathcal{F}_{\mathsf{Offline}}[\mathsf{SPDZ}]$

Let A be the set of indices of corrupted players. Symbols in bold denote vectors in $(\mathbb{F}_p)^k$. Arithmetic is componentwise.

Initialize: On input (Start, p) from honest players and adversary, it does the following:

1. For each corrupted player $i \in A$, the functionality accepts shares α_i from the adversary, and it samples at random α_i for each $i \notin A$. Then the functionality sets $\alpha \leftarrow \alpha_1 + \cdots + \alpha_n$.
2. The functionality calls the macro $\mathsf{Abort}()$ below.
3. The functionality outputs α_i to player i.

Macro $\mathsf{Abort}()$ This macro performs the following steps

1. The functionality waits for signal from the adversary.
2. If it receives $\mathsf{Proceed}$ the macro returns, otherwise the entire protocol aborts.

Macro $\mathsf{Angle}(\mathbf{v}_1, \ldots, \mathbf{v}_n, \Delta_\gamma, k)$ This macro will be run by the functionality at several points to create representations $\langle \cdot \rangle_F$.

1. The functionality obtains $\{\gamma_i\}_{i \in A}$ from the adversary.
2. The functionality calls the macro $\mathsf{Abort}()$.
3. Let $\mathbf{v} = \mathbf{v}_1 + \cdots + \mathbf{v}_n$, set $\gamma[\mathbf{v}] \leftarrow \alpha \cdot \mathbf{v} + \Delta_\gamma$.
4. Sample at random $\gamma_i[\mathbf{v}] \leftarrow (\mathbb{F}_p)^k$ for $i \notin A$, subject to $\gamma[\mathbf{v}] = \sum_i^n \gamma_i[\mathbf{v}]$.
5. Return $(\gamma_1[\mathbf{v}], \ldots, \gamma_n[\mathbf{v}])$.

Input Production: On input $\mathsf{DataType} = (\mathsf{InputPrep}, n_I)$,

1. The functionality choose random values $I = \{\mathbf{r}^{(i)} \in (\mathbb{F}_p)^{n_I} \mid i \notin A\}$.
2. It accepts from the adversary corrupted values $\{\mathbf{r}^{(i)} \in (\mathbb{F}_p)^{n_I} \mid i \in A\}$, corrupted shares $\{\mathbf{r}_k^{(i)} \in (\mathbb{F}_p)^{n_I} \mid k \in A, i \leq n\}$, and offset for data and MACs $\{\Delta_r^{(i)}, \Delta_\gamma^{(i)} \in (\mathbb{F}_p)^{n_I} \mid i \leq n\}$.
3. The functionality calls the macro $\mathsf{Abort}()$.
4. The functionality samples honest shares $\{\mathbf{r}_k^{(i)} \mid k \notin A, i \leq n\}$ subject to $\mathbf{r}^{(i)} + \Delta_r^{(i)} = \sum_{k=1}^n \mathbf{r}_k^{(i)}$.
5. It then runs the macro $\mathsf{Angle}(\mathbf{r}_1^{(i)}, \ldots, \mathbf{r}_n^{(i)}, \Delta_\gamma^{(i)}, n_I)$, for $i \leq n$.
6. Finally it outputs $\{\mathbf{r}^{(i)}, (\mathbf{r}_i^{(j)}, \gamma_i[\mathbf{r}^{(j)}])_{j \leq n}\}$ to player i.

Multiplication Triples: On input $\mathsf{DataType} = (\mathsf{Triples}, n_m)$,

1. Choose $2 \cdot n_m$ honest shares $I = \{(\mathbf{a}_i, \mathbf{b}_i) \in (\mathbb{F}_p)^{2 \cdot n_m} \mid i \notin A\}$.
2. It accepts corrupted shares $\{(\mathbf{a}_i, \mathbf{b}_i, \mathbf{c}_i) \in (\mathbb{F}_p)^{3 \cdot n_m} \mid i \in A\}$ and MAC offsets $\{(\Delta_\gamma^{(a)}, \Delta_\gamma^{(b)}, \Delta_\gamma^{(c)}) \in (\mathbb{F}_p)^{3 \cdot n_m}\}$ from the adversary.
3. The functionality calls the macro $\mathsf{Abort}()$.
4. The functionality sets $\mathbf{c} \leftarrow (\mathbf{a}_1 + \cdots + \mathbf{a}_n) \cdot (\mathbf{b}_1 + \cdots + \mathbf{b}_n)$.
5. It computes a set of honest shares $\{\mathbf{c}_i \mid i \notin A\}$ subject to $\mathbf{c} = \sum_{i=1}^n \mathbf{c}_i$.
6. The functionality then runs the macros $\mathsf{Angle}(\mathbf{a}_1, \ldots, \mathbf{a}_n, \Delta_\gamma^{(a)}, n_m)$, $\mathsf{Angle}(\mathbf{b}_1, \ldots, \mathbf{b}_n, \Delta_\gamma^{(b)}, n_m)$, $\mathsf{Angle}(\mathbf{c}_1, \ldots, \mathbf{c}_n, \Delta_\gamma^{(c)}, n_m)$.
7. Finally it outputs $\{(\mathbf{a}_i, \gamma_i[\mathbf{a}]), (\mathbf{b}_i, \gamma_i[\mathbf{b}]), (\mathbf{c}_i, \gamma_i[\mathbf{c}])\}$ to player i.

Figure 2. SPDZ Offline Functionality

$z = a \cdot x + b \cdot y + c$ locally by each player computing

$$\begin{aligned} z_1 &\leftarrow a \cdot x_i + b \cdot y_i + c && \text{for } i = 1 \\ z_i &\leftarrow a \cdot x_i + b \cdot y_i && \text{for } i \neq 1 \\ \gamma_i[z] &\leftarrow a \cdot \gamma_i[x] + b \cdot \gamma_i[y] + \alpha_i \cdot c && \text{for all } i. \end{aligned}$$

Protocol $\Pi_{\mathsf{MACCheck}}[\mathsf{SPDZ}]$

Usage: Each player has input α_i and $(\gamma_i[a_j])$ for $j = 1, \ldots, t$. All players have a public set of opened values $\{a_1, \ldots, a_t\}$; the protocol either succeeds or outputs failure if an inconsistent MAC value is found.

$\mathsf{MACCheck}(\{a_1, \ldots, a_t\})$**:**

1. Every player $\mathcal{P}_i$ samples a seed s_i and asks $\mathcal{F}_{\mathsf{Commit}}$ to broadcast $\tau_i^s \leftarrow \mathsf{Comm}(s_i)$.
2. Every player $\mathcal{P}_i$ calls $\mathcal{F}_{\mathsf{Commit}}$ with $\mathsf{Open}(\tau_i^s)$ all players obtain s_j for all j.
3. Set $s \leftarrow s_1 \oplus \cdots \oplus s_n$.
4. Players using s sample a random vector $\mathbf{r}$ from $\mathbb{F}_p^t$; note all players obtain the same vector as they have agreed on the seed s.
5. Each player computes the public value $a \leftarrow \sum_{j=1}^{t} r_j \cdot a_j$.
6. P_i computes $v_i \leftarrow \sum_{j=1}^{t} r_j \cdot \gamma_i[a_j]$, and $w_i \leftarrow v_i - \alpha_i \cdot a$.
7. P_i asks $\mathcal{F}_{\mathsf{Commit}}$ to broadcast $\tau_i^w \leftarrow \mathsf{Comm}(w_i)$.
8. Every player calls $\mathcal{F}_{\mathsf{Commit}}$ with $\mathsf{Open}(\tau_i^w)$, all players obtain w_i for all i.
9. If $w_1 + \cdots + w_n \neq 0_{\mathbb{F}_p}$, the players output $\varnothing$ and abort.

Figure 3. Method To Check MACs On Partially Opened Values

Protocol $\Pi_{\mathsf{Online}}[\mathsf{SPDZ}]$

Initialize: The parties call $\mathcal{F}_{\mathsf{Offline}}$ to get the shares α_i of the MAC key, a number of multiplication triples $(\langle a \rangle_F, \langle b \rangle_F, \langle c \rangle_F)$ and mask values $(r_i, \langle r_i \rangle_F)$ as needed for the circuit being evaluated. If $\mathcal{F}_{\mathsf{Offline}}$ aborts then abort, otherwise the operations specified below are performed according to the circuit.

Input: To share his input x_i, player i takes an available mask value $(r_i, \langle r_i \rangle_F)$ and does the following:

1. Broadcast $e \leftarrow x_i - r_i$.
2. The players compute $\langle x_i \rangle_F \leftarrow \langle r_i \rangle_F + e$.

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

Multiply: On input $(\langle x \rangle_F, \langle y \rangle_F)$, the players do the following:

1. Take one multiplication triple $(\langle a \rangle_F, \langle b \rangle_F, \langle c \rangle_F)$ and open $\langle x \rangle_F - \langle a \rangle_F, \langle y \rangle_F - \langle b \rangle_F$ to get s and t respectively.
2. Locally each player computes $\langle z \rangle_F \leftarrow \langle c \rangle_F + s \cdot \langle b \rangle_F + t \cdot \langle a \rangle_F + s \cdot t$

Triple: Here the players simply take one multiplication triple $(\langle a \rangle_F, \langle b \rangle_F, \langle c \rangle_F)$ off the pre-computed list obtained in the offline phase.

Output: This procedure is entered once the players have finished the circuit evaluation, but still the final output $\langle y \rangle_F$ has not been opened.

1. The players call the $\mathsf{MACCheck}[\mathsf{SPDZ}]$ protocol on input all opened values so far.
2. The players open $\langle y \rangle_F$ and call $\mathsf{MACCheck}[\mathsf{SPDZ}]$ on input y to verify its MAC.

Figure 4. Operations for Secure Function Evaluation

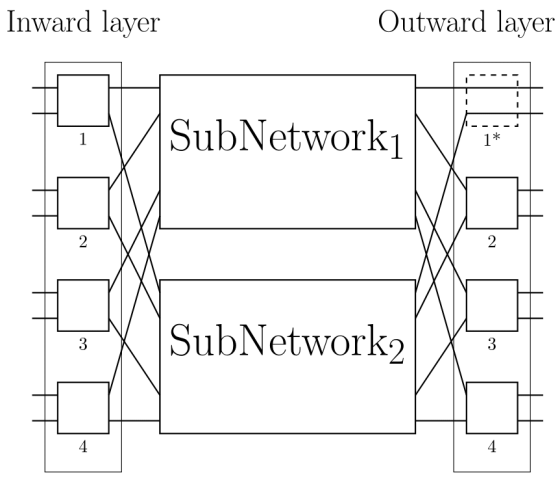

Fig. 5. Waksman network of size 8

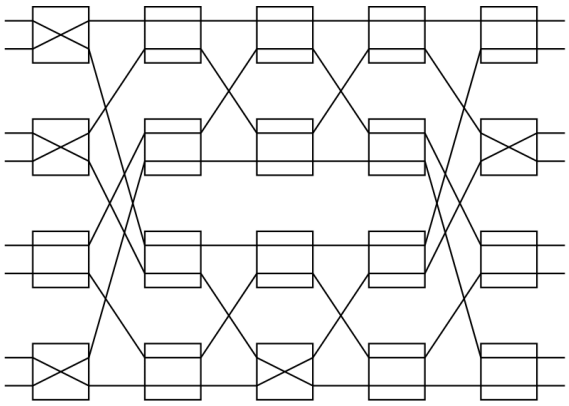

Fig. 6. A realization of the permutation $\tilde{\pi}$

2.3 Waksman Networks

A Waksman network [17] is a circuit with m input and output wires, to make our discussion cleaner we will assume m is a power of two. Within the circuit, inputs are shuffled with respect to a permutation. The building blocks of a Waksman network are switches; where a switch is a circuit with two input and output wires, with a hardwired bit called the control bit. If the control bit equals one, the switch swaps its inputs, otherwise, the switch simply forwards its inputs to the output wires.

The construction of a Waksman network follows a recursive structure (Figure 5), that is to say a Waksman network contains:

- One inward layer.
- One outward layer.
- Two parallel subnetworks of size $\frac{m}{2}$, linked in a butterfly manner to the inward and outward layers.

The inward layer contains $\frac{m}{2}$ switches, whereas the outward layer contains $\frac{m}{2} - 1$ switches. That is, the missing switch (switch 1^*) is fixed by setting its control bit to be zero. The inner networks are constructed in a similar manner, and there are $2 \cdot \log_2(m) - 1$ layers in total. Thus the total number of switches within the whole networks is $m \cdot \log_2(m) - m + 1$. Given any permutation, there is a classical algorithm to determine a set of control bits realizing it. That is to say, this algorithm takes as input the permutation, and outputs a control bit for every switch, such that the resulting Waksman network realizes this permutation. In our work (for ease of implementing the algorithm to create a Waksmn network in a data-oblivious manner) we use a more relaxed definition of a Waksman network in which the first gate of the outer layer is not fixed to be an empty switch. This increases the total number of switches in a network by $m/2 - 1$ to the value $m \cdot \log_2(m) - m/2$.

As an example consider the permutation matrix

$$\tilde{M} = \begin{bmatrix} 0&1&0&0&0&0&0&0 \\ 1&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&1 \\ 0&0&0&1&0&0&0&0 \\ 0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&1&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&0&0&0&0&0&1&0 \end{bmatrix}$$

realizing the permutation $\tilde{\pi} \in S_8$. This can be represented by the Waksman network in Figure 6. One immediately sees an advantage of using a Waksman network to represent a permutation on a vector v. Instead of evaluating π on v by a matrix vector product, we apply a series of transpositions on v using the permutation networks. This is more efficient in terms of operations performed. That is, the matrix based approach requires m^2 multiplications to apply a permutation on a vector, whereas a network based approach using (our modified) Waksman network, would require at most $m \cdot \log_2(m) - m/2$ swaps. In terms of an MPC based implementation of the network for secret shared control bits, one swap consists of two multiplications.

3 Multiparty Computation Over Elliptic Curve Groups

Our goal in this section is to define a protocol to perform efficient actively secure (with abort) MPC in the context of elliptic curve calculations. This will enable us to efficiently transfer ECC based cryptographic protocols into the distributed domain. Such protocols require us to perform arithmetic not only in the elliptic curve group, but also over the finite field given by the order of the large prime subgroup (the so-called exponent group, even though it is a field). Our basic strategy is to use the SPDZ protocol to conduct the MPC in the exponent group, and to use a similar protocol with *the same MAC key* to define the MPC protocol to work in the elliptic curve group itself.

The functionality we aim to produce a secure realisation of is $\mathcal{F}_{\mathsf{Online}}[\mathsf{ECC}]$, given in Figure 7. One immediately notes that this is essentially an extension of the $\mathcal{F}_{\mathsf{Online}}[\mathsf{SPDZ}]$ functionality in that we have added an additional set of variables corresponding to elliptic curve points. Given this similarity it should not be surprising that we utilize the same secret sharing as in SPDZ to share values in the exponent group.

In our realisation of this functionality, an elliptic curve data element $Q \in \mathcal{G}$ is held in secret shared form as a tuple $\{Q_i, \Gamma_i\}_{i \in [n]}$, such that $Q = \sum_i Q_i$ and $\sum \Gamma_i = [\alpha] \cdot Q$ where α is the same MAC key as used in the secret sharing $\langle \cdot \rangle_F$. We denote a value Q held in such a secret shared form as $\langle Q \rangle_E$. Again if we want to denote the specific value on which Γ_i is a MAC share we will write $\Gamma_i[Q]$.

Again, as $\langle \cdot \rangle_E$ is a linear secret sharing scheme it is easy to compute linear functions on the share values. In particular given $\langle Q \rangle_E$ and $\langle R \rangle_E$, two field constants $a, b \in \mathbb{F}_p$ and one public curve point S we can compute the sharing of $T = [a] \cdot Q + [b] \cdot R + S$ locally by each player computing

$$\begin{aligned} T_0 &\leftarrow [a] \cdot Q_i + [b] \cdot R_i + S && \text{for } i = 0 \\ T_i &\leftarrow [a] \cdot Q_i + [b] \cdot R_i && \text{for } i \neq 0 \\ \Gamma_i[T] &\leftarrow [a] \cdot \Gamma_i[Q] + [b] \cdot \Gamma_i[R] + [\alpha_i] \cdot S && \text{for all } i. \end{aligned}$$

One can also compute $U = [a] \cdot Q$ when a is shared and Q is public, with only local computation. This can be done by having each player locally compute

$$\begin{aligned} U_i &\leftarrow [a_i] \cdot Q && \text{for all } i. \\ \Gamma_i[U] &\leftarrow [\gamma_i[a]] \cdot Q && \text{for all } i. \end{aligned}$$

Which we write as

$$\langle U \rangle_E \leftarrow [\langle a \rangle_F] \cdot Q. \tag{1}$$

The only complex part is then to perform non-linear operations, namely to compute the point multiplication of a secret shared element in $\mathbb{F}_q$ by a secret shared elliptic curve point. An operation which we write as

$$\langle U \rangle_E \leftarrow [\langle a \rangle_F] \cdot \langle Q \rangle_E. \tag{2}$$

3.1 MACCheck Protocol

Before presenting our protocol for performing the arithmetic operation in equation (2), we first modify the SPDZ MACCheck protocol so that it also checks the MAC values on the authenticated sharings of elliptic

Functionality $\mathcal{F}_{\mathsf{Online}}[\mathsf{ECC}]$

Initialize: On input $(init, \mathcal{G})$ from all parties, the functionality stores $(domain, \mathcal{G})$. Two lists of identifiers are established, one called field identifiers and one called curve identifiers.

Input-F: On input $(inputF, \mathcal{P}_i, varid, x)$ with $x \in \mathbb{F}_p$ from $\mathcal{P}_i$ and $(input, \mathcal{P}_i, varid, ?_{\mathbb{F}_p})$ from all other parties, with $varid$ a fresh identifier, the functionality stores $(varid, x)$ in the list of field identifiers.

Input-G: On input $(inputG, \mathcal{P}_i, varid, Q)$ with $Q \in \mathcal{G}$ from $\mathcal{P}_i$ and $(input, \mathcal{P}_i, varid, ?_{\mathcal{G}})$ from all other parties, with $varid$ a fresh identifier, the functionality stores $(varid, P)$ in the list of curve identifiers.

Add-F: On command $(addF, varid_1, varid_2, varid_3)$ from all parties where $varid_1, varid_2$ are in the list of field identifiers and $varid_3$ is not, the functionality retrieves $(varid_1, x)$, $(varid_2, y)$ from the list of field identifiers and stores $(varid_3, x + y)$ in the list of field identifiers.

Add-G: On command $(addG, varid_1, varid_2, varid_3)$ from all parties where $varid_1, varid_2$ are in the list of curve identifiers and $varid_3$ is not, the functionality retrieves $(varid_1, Q)$, $(varid_2, R)$ from the list of curve identifiers and stores $(varid_3, Q + R)$ in the list of curve identifiers.

Multiply-F: On command $(multiplyF, varid_1, varid_2, varid_3)$ from all parties where $varid_1, varid_2$ are in the list of field identifiers and $varid_3$ is not, the functionality retrieves $(varid_1, x)$, $(varid_2, y)$ from the list of field identifiers and stores $(varid_3, x \cdot y)$ in the list of field identifiers.

Triple: On input $(triple, varid_1, varid_2, varid_3)$ from all parties (if none of the $varid_i$ are stored in memory), the functionality generates a uniformly random $a, b \in \mathbb{F}_p$ and computes $c = a \cdot b$ and then stores $(varid_1, a)$, $(varid_2, b)$ and $(varid_3, c)$ in the list of field identifiers.

Multiply-G-P: On command $(multiplyGP, varid_1, Q, varid_2)$ from all parties where $varid_1$ is in the list of field identifiers, $Q \in \mathcal{G}$, and $varid_2$ is a fresh identifier from the list of the curve identifiers, the functionality retrieves $(varid_1, x)$, from the list of field identifiers and stores $(varid_2, [x] \cdot Q)$.

Multiply-G-S: On command $(multiplyGS, varid_1, varid_2, varid_3)$ from all parties where $varid_1$ is in the list of field identifiers and $varid_2$ is in the list of curve identifiers and $varid_3$ is not, the functionality retrieves $(varid_1, x)$, $(varid_2, Q)$ from the respective lists and stores $(varid_3, [x] \cdot Q)$.

Output-F: On input $(outputF, varid, i)$ from all honest parties (if $varid$ is present in the list of field identifiers), the functionality retrieves $(varid, y)$ from the set of field identifiers and outputs it to the environment. The functionality waits for an input from the environment. If this input is $\mathsf{Deliver}$ then y is output to all players if $i = 0$, or y is output to player i if $i \neq 0$. If the adversarial input is not equal to $\mathsf{Deliver}$ then $\varnothing$ is output to all players.

Output-G: On input $(outputG, varid, i)$ from all honest parties (if $varid$ is present in the list of curve identifiers), the functionality retrieves $(varid, R)$ from the set of curve identifiers and outputs it to the environment. The functionality waits for an input from the environment. If this input is $\mathsf{Deliver}$ then R is output to all players if $i = 0$, or R is output to player i if $i \neq 0$. If the adversarial input is not equal to $\mathsf{Deliver}$ then $\varnothing$ is output to all players.

Figure 7. The ideal functionality for MPC over $\mathcal{G} \subseteq E(K)$ with $\#\mathcal{G} = p$

curve points. This is done in Figure 8. The intuition behind correctness and soundness of this protocol, comes from the fact that we took the MAC check protocol from SPDZ operating over elements in $\mathbb{F}_p$, and replicated it for elements of $\mathcal{G}$. Then, we combined these two checks in order to execute the protocol only once. So we perform the MAC check in a single operation on both opened elements in $\mathbb{F}_p$ and opened elements in $\mathcal{G}$. Thus at step 7 we map the share $v_i - \alpha_i \cdot a$ of player i to the point $[v_i - \alpha_i \cdot a] \cdot P$ over $\mathcal{G}$. This makes the protocol inherit the correctness and soundness properties from the one in SPDZ.

Protocol $\Pi_{\mathsf{MACCheck}}[\mathsf{ECC}]$

Usage: Each player has input α_i, $(\gamma_i[a_j])$ for $j = 1, \ldots, t$ and $(\Gamma_i[A_k])$ for $k = 1, \ldots, u$. All players have two public sets of opened values : $\{a_1, \ldots, a_t\}$ over $\mathbb{F}_p$, and $\{A_1, \ldots, A_u\}$ over $\mathcal{G}$; the protocol either succeeds or outputs failure if an inconsistent MAC value is found.

$\mathsf{MACCheck}(\{a_1, \ldots, a_t\}, \{A_1, \ldots, A_u\})$**:**

1. Every player $\mathcal{P}_i$ samples a seed s_i and asks $\mathcal{F}_{\mathsf{Commit}}$ to broadcast $\tau_i^s \leftarrow \mathsf{Comm}(s_i)$.
2. Every player $\mathcal{P}_i$ calls $\mathcal{F}_{\mathsf{Commit}}$ with $\mathsf{Open}(\tau_i^s)$ and all players obtain s_j for all j.
3. Set $s \leftarrow s_1 \oplus \cdots \oplus s_n$.
4. Players using s sample a random vector $\mathbf{r}$ from $\mathbb{F}_p^{t+u}$; note all players obtain the same vector as they have agreed on the seed s.
5. Each player computes the public values $a \leftarrow \sum_{j=1}^{t} r_j \cdot a_j$ and $A \leftarrow \sum_{k=1}^{u} [r_{k+t}] \cdot A_k$
6. Player i computes $v_i \leftarrow \sum_{j=1}^{t} r_j \cdot \gamma_i[a_j]$, and $\Gamma_i \leftarrow \sum_{k=1}^{u} [r_{k+t}] \cdot \Gamma_i[A_k]$
7. Player i computes $W_i \leftarrow [v_i - \alpha_i \cdot a] \cdot P + \Gamma_i - [\alpha_i] \cdot A$, where P is the generator of $\mathcal{G}$
8. Player i asks $\mathcal{F}_{\mathsf{Commit}}$ to broadcast $\tau_i^W \leftarrow \mathsf{Comm}(W_i)$.
9. Every player calls $\mathcal{F}_{\mathsf{Commit}}$ with $\mathsf{Open}(\tau_i^W)$, and all players obtain W_i for all i.
10. If $W_1 + \cdots + W_n \neq 0_{\mathcal{G}}$, the players output $\varnothing$ and abort.

Figure 8. Method To Check MACs On Partially Opened Values

Theorem 1. *The protocol $\Pi_{\mathsf{MACCheck}}[\mathsf{ECC}]$ is correct and sound. That is, it accepts if all values $\{a_1, \ldots, a_t\}$ and $\{A_1, \ldots, A_u\}$ along with their corresponding MACs were correctly computed, and it rejects except with negligible probability if at least one value or MAC was not correctly computed.*

The proof of Theorem 1 is given in Appendix B.

3.2 MPC Online Protocol

We introduce now in Figure 9 our protocol for the online phase. Similar to $\mathcal{F}_{\mathsf{Online}}[\mathsf{ECC}]$, which was constructed by extending $\mathcal{F}_{\mathsf{Online}}[\mathsf{SPDZ}]$, we do the same here while realizing $\mathcal{F}_{\mathsf{Online}}[\mathsf{ECC}]$. That is, the subfunctionalities **Initialize**, **Input-F**, **Add-F**, and **Multiply-F**, will be realized the same way as in the protocol $\Pi_{\mathsf{Online}}[\mathsf{SPDZ}]$. For the remaining functionalities, we will realize them using essentially the same techniques as in the SPDZ protocol.

- **Input-G** is realized using the same trick as used to impement **Input-F**. That is, assuming player i holds $(R_i, \langle R_i \rangle_E)$, this player can share a point $Q_i \in \mathcal{G}$ by broadcasting $E = Q_i - R_i$, then players compute $\langle Q_i \rangle_E = \langle R_i \rangle_E + E$ to obtain a share of Q_i. However, as we are using only the preprocessing of SPDZ, we need to somehow provide $(R_i, \langle R_i \rangle_E)$ to player i, using only the generated data from $\mathcal{F}_{\mathsf{Offline}}[\mathsf{SPDZ}]$. This can be done using the generator P, That is, from a SPDZ input mask $(r_i, \langle r_i \rangle_F)$, one can obtain a mask $(R_i, \langle R_i \rangle_E)$ by setting $\langle R_i \rangle_E \leftarrow [\langle r_i \rangle_F] \cdot P$, which requires only local computation.
- **Add-G** is realized similarly to **Add-F**. That is, players will locally compute $\langle Q + R \rangle_E \leftarrow \langle Q \rangle_E + \langle R \rangle_E$.
- **Multiply-G-P** is realized by having players locally compute $\langle R \rangle_E \leftarrow \langle x \rangle_F \cdot Q$.

Protocol $\Pi_{\mathsf{Online}}[\mathsf{ECC}]$

Initialize: The parties call $\mathcal{F}_{\mathsf{Offline}}$ to get the shares α_i of the MAC key, a number of multiplication triples $(\langle a\rangle_F, \langle b\rangle_F, \langle c\rangle_F)$, and mask values $(r_i, \langle r_i\rangle_F)$ as needed for the circuit being evaluated. If $\mathcal{F}_{\mathsf{Offline}}$ aborts then abort, otherwise the operations specified below are performed according to the circuit.

Input-F: To share his input x_i, player i takes an available mask value $(r_i, \langle r_i\rangle_F)$ and does the following:

1. Broadcast $e \leftarrow x_i - r_i$.
2. The players compute $\langle x_i\rangle_F \leftarrow \langle r_i\rangle_F + e$.

Input-G: To share his input Q_i, Player i takes an available mask value $(r_i, \langle r_i\rangle_F)$ and does the following:

1. Broadcast $E \leftarrow Q_i - [r_i] \cdot P$.
2. The players compute $\langle Q_i\rangle_E \leftarrow [\langle r_i\rangle_F] \cdot P + E$ using Multiply-G-P.

Add-F: On input $(\langle x\rangle_F, \langle y\rangle_F)$, the players locally compute $\langle x + y\rangle_F \leftarrow \langle x\rangle_F + \langle y\rangle_F$.

Add-G: On input $(\langle Q\rangle_E, \langle R\rangle_E)$, the players locally compute $\langle Q + R\rangle_E \leftarrow \langle Q\rangle_E + \langle R\rangle_E$.

Multiply-F: On input $(\langle x\rangle_F, \langle y\rangle_F)$, the players do the following:

1. Take one multiplication triple $(\langle a\rangle_F, \langle b\rangle_F, \langle c\rangle_F)$ and open $\langle x\rangle_F - \langle a\rangle_F, \langle y\rangle_F - \langle b\rangle_F$ to get s and t respectively.
2. Locally each player sets $\langle x \cdot y\rangle_F \leftarrow \langle c\rangle_F + s \cdot \langle b\rangle_F + t \cdot \langle a\rangle_F + s \cdot t$

Triple: Here the players simply take one multiplication triple $(\langle a\rangle_F, \langle b\rangle_F, \langle c\rangle_F)$ off the pre-computed list obtained in the offline phase.

Multiply-G-P: On input $(\langle x\rangle_F, Q)$, where Q is a public point in $\mathcal{G}$, the players locally compute $\langle [x] \cdot Q\rangle_E \leftarrow [\langle x\rangle_F] \cdot Q$ using the operation defined in (1) .

Multiply-G-S: On input $(\langle x\rangle_F, \langle Q\rangle_E)$, the players do the following:

1. Take one multiplication triple $(\langle a\rangle_F, \langle b\rangle_F, \langle c\rangle_F)$, locally compute $\langle U\rangle_E \leftarrow [\langle b\rangle_F] \cdot P$ and $\langle V\rangle_E \leftarrow [\langle c\rangle_F] \cdot P$ using Multiply-G-P.
2. Open $\langle x\rangle_F - \langle a\rangle_F, \langle Q\rangle_E - \langle U\rangle_E$ to get s and T respectively.
3. Locally each player sets $\langle [x] \cdot Q\rangle_E \leftarrow \langle V\rangle_E + [s] \cdot \langle U\rangle_E + [\langle a\rangle_F] \cdot T + [s] \cdot T$

Output-F: This procedure is entered once the players have finished the circuit evaluation, but still the final output $\langle y\rangle_F$ has not been opened.

1. The players call the $\mathsf{MACCheck}[\mathsf{ECC}]$ protocol on input all opened values so far.
2. The players open $\langle y\rangle_F$ and call $\mathsf{MACCheck}[\mathsf{ECC}]$ on input y to verify its MAC.

Output-G: This procedure is entered once the players have finished the circuit evaluation, but still the final output $\langle R\rangle_E$ has not been opened.

1. The players call the $\mathsf{MACCheck}[\mathsf{ECC}]$ protocol on input all opened values so far.
2. The players open $\langle R\rangle_E$ and call $\mathsf{MACCheck}[\mathsf{ECC}]$ on input R to verify its MAC.

Figure 9. Operations for Secure Function Evaluation

– **Multiply-G-S** is realized using the Beaver trick. That is, assuming players hold a triple $(\langle a\rangle_F, \langle U\rangle_E, \langle V\rangle_E)$ such that $V = [a]\cdot U$, to compute $[\langle x\rangle_F]\cdot\langle Q\rangle_E$, players open $\langle x-a\rangle_F$ and $\langle Q-U\rangle_E$ to obtain s and T. Then the product can be obtained by setting $\langle [x]\cdot Q\rangle_E \leftarrow \langle V\rangle_E + [s]\cdot\langle U\rangle_E + [\langle a\rangle_F]\cdot T + [s]\cdot T$. However, the SPDZ preprocessing doesn't provide this type of triples, nonetheless, we still can obtain them locally by taking a SPDZ-triple $(\langle a\rangle_F, \langle b\rangle_F, \langle c\rangle_F)$ and having players locally compute $\langle U\rangle_E = \langle b\rangle_F \cdot P$ and $\langle V\rangle_E = \langle c\rangle_F \cdot P$. This results in a triple $(\langle a\rangle_F, \langle U\rangle_E, \langle V\rangle_E)$, which is a valid triple since $V = [c]\cdot P = [a\cdot b]\cdot P = [a]\cdot U$.
– **Output-F** and **Output-G** are realized the same way as in SPDZ, where we call here the MAC check protocol $\Pi_{\mathsf{MACCheck}}[\mathsf{ECC}]$ that we defined in the previous section, operating over all opened values in $\mathbb{F}_p$ and $\mathcal{G}$ so far, then we open the final output y (resp. R) and call $\Pi_{\mathsf{MACCheck}}[\mathsf{SPDZ}]$ (resp. $\Pi_{\mathsf{MACCheck}}[\mathsf{ECC}]$)

Theorem 2. *The protocol $\Pi_{\mathsf{Online}}[\mathsf{ECC}]$ securely implements $\mathcal{F}_{\mathsf{Online}}[\mathsf{ECC}]$ in the $\mathcal{F}_{\mathsf{Offline}}[\mathsf{SPDZ}]$ hybrid model.*

Proof. We construct in Figure 10 a simulator $\mathcal{S}_{\mathsf{Online}}[\mathsf{ECC}]$, such that no polynomial time environment, who corrupts a set of parties $\mathcal{A}$ among $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ and chooses inputs for all parties, can distinguish with significant probability, a view obtained by running $\Pi_{\mathsf{Online}}[\mathsf{ECC}]$ between $\mathcal{A}$ and the honest parties, and a simulated execution of the protocol between $\mathcal{S}_{\mathsf{Online}}[\mathsf{ECC}]$ and $\mathcal{F}_{\mathsf{Online}}[\mathsf{ECC}]$. The environment's view consists of the intermediate messages sent and received by $\mathcal{A}$, inputs he chose for all players, along with outputs of all players.

The simulator $\mathcal{S}_{\mathsf{Online}}[\mathsf{ECC}]$

Initialize: The simulation of the initialization procedure is performed running a local copy of $\mathcal{F}_{\mathsf{Offline}}[\mathsf{SPDZ}]$. Notice that all data given to the adversary is known by the simulator.

Input-F: Honest inputs are performed according to the protocol, with a dummy input, for example $0_{\mathbb{F}_p}$. For inputs given by a corrupt player $\mathcal{P}_i$, the simulator waits for P_i to broadcast the (possibly incorrect) value e', computes $x_i' \leftarrow r_i + e'$ and uses x_i' as input to $\mathcal{F}_{\mathsf{Online}}[\mathsf{ECC}]$.

Input-G: Honest inputs are performed according to the protocol, with a dummy input, for example $0_{\mathcal{G}}$. For inputs given by a corrupt player $\mathcal{P}_i$, the simulator waits for P_i to broadcast the (possibly incorrect) value E', computes $Q_i' \leftarrow [r_i]\cdot P + E'$ and uses Q_i' as input to $\mathcal{F}_{\mathsf{Online}}[\mathsf{ECC}]$.

Add-F, Add-G, Multiply-F, Multiply-G-P, Multiply-G-S: These procedures are performed according to the protocol. The simulator also calls the respective sub-functionalities in $\mathcal{F}_{\mathsf{Online}}[\mathsf{ECC}]$.

Output-F, Output-G: $\mathcal{F}_{\mathsf{Online}}[\mathsf{ECC}]$ outputs y (resp. R) to the simulator. The simulator now has to provide the honest players' shares of such value, it already computed an output value y' (resp. R'), using the dummy inputs for the honest players. The simulator will then modify the share of the player P_h (P_h is one of the honest players) by adding $y - y'$ (resp. $R - R'$) and modify the MAC adding $\alpha(y - y')$ (resp. $[\alpha]\cdot(R - R')$), which is possible for the simulator, since it knows α. After that, the simulator is ready to open y (resp. R) according to the protocol. If $\mathsf{MACCheck}[\mathsf{ECC}]$ doesn't abort, the simulator sends $\mathsf{Deliver}$ to $\mathcal{F}_{\mathsf{Online}}[\mathsf{ECC}]$.

Figure 10. Simulator for the Online phase

Prior to the output stage, computation of the protocol is either local, or involves openings that reveal values uniformly at random. Thus, the environment's view, up to this point, will not leak whether inputs used by honest players' are dummy inputs or the ones the environment provided. Note that at the meantime, the simulator is querying the respective sub-functionalities from $\mathcal{F}_{\mathsf{Online}}[\mathsf{ECC}]$.

At the output stage, the simulator queries the output sub-functionality on the functionality $\mathcal{F}_{\mathsf{Online}}[\mathsf{ECC}]$ so as to obtain y (or R). This output is automatically consistent with the values communicated by $\mathcal{A}$. The simulator will then modify the share of the honest player P_h to make the sum of honest players' shares consistent with the output generated by $\mathcal{F}_{\mathsf{Online}}[\mathsf{ECC}]$. The simulator sends then honest players' shares of the output to $\mathcal{A}$.

Note that, while performing an operation that involves only local computation (namely Add-F, Add-G or Multiply-G-P), the share of every player of the resulting value is a linear combination of his shares of other

values. Therefore, for a value to be opened, that can be written using only the 3 operations above on values that are already opened, the honest players' shares are already known to $\mathcal{A}$. This means that the simulator needs to send $\mathcal{A}$ those shares. We guaranteed this by modifying the shares of the same player P_h each time we are in the output stage.

Finally, $\mathsf{MACCheck}[\mathsf{ECC}]$ is executed on all opened values. If it succeeds, the simulator inputs $\mathsf{Deliver}$ into $\mathcal{F}_{\mathsf{Online}}[\mathsf{ECC}]$ and honest players get the output. Otherwise, the simulator aborts. Therefore, the only case where the environment can distinguish between a real execution and a simulated one, is when $\mathcal{A}$ succeeds in introducing an error in opened values, without being caught by $\mathsf{MACCheck}[\mathsf{ECC}]$. This can happen only with negligible probability.

4 Simple Example Applications

In this section we present two toy applications of our methodology to perform MPC over elliptic curves, distributed EC-DSA signing and a distributed zero-knowledge proof.

EC-DSA Signing

Given a secret key $x \in \mathbb{F}_p$ for a public key $Q = [x] \cdot P$, for P an element of order q in $E(K)$, a hash function H with codomain $\mathbb{F}_p$, and a message m, the EC-DSA operation is given by:

1. $k \leftarrow \mathbb{F}_p$.
2. $R \leftarrow [k] \cdot P$.
3. $r \leftarrow \mathsf{x} - \mathsf{coord}(R) \pmod p$.
4. $e \leftarrow H(m)$.
5. $s \leftarrow (e + x \cdot r)/k \pmod p$.
6. Output (r, s).

Figure 11. EC-DSA Signing Operation

4.1 Distributed EC-DSA

The EC-DSA signing operation is given in Figure 11. To produce a distributed version we assume that the secret key x is already secret shared $\langle x \rangle_F$ using our secret sharing scheme. For simplicity we ignore the unlikely event that $r = 0$ in our description. The associated distributed version is given in Figure 12. The protocol requires three multiplication triples from the offline phase (one to produce the initial sharings of $(\langle k \rangle_F, \langle b \rangle_F, \langle k \cdot b \rangle_F)$ and two to enable the secure computation of $\langle u \rangle_F$ and $\langle v \rangle_F$. Note, that we have correctness since the s produced by the distributed EC-DSA protocol is equal to

$$s = \frac{v}{c} = \frac{u \cdot b}{k \cdot b} = \frac{e + x \cdot r}{k}.$$

The trivial simulation of the distributed protocol appears to leak a minor amount of information. In particular the execution of the distributed protocol reveals R, whereas the ideal functionality for distributed signing will only reveal $r = \mathsf{x} - \mathsf{coord}(R) \pmod p$. However, the verification operation recovers R in any case, thus this is not an actual leak of information.

4.2 Distributed OR-Proof

The above EC-DSA application did not use the full power of our MPC over elliptic curves, in particular we did not make use of any non-linear operations on the elliptic curve. Here we present a more complex

Distributed EC-DSA Signing

1. Call **Triple** on $\mathcal{F}_{\mathsf{Online}}[\mathsf{ECC}]$ so as to obtain $(\langle k\rangle_F, \langle b\rangle_F, \langle c\rangle_F)$ where $c = k \cdot b$.
2. Compute $\langle R\rangle_E \leftarrow [\langle k\rangle_F] \cdot P$ by calling **Multiply-G-P** on $\mathcal{F}_{\mathsf{Online}}[\mathsf{ECC}]$.
3. Open $\langle R\rangle_E$ so all parties obtain R by calling **Output-G** on $\mathcal{F}_{\mathsf{Online}}[\mathsf{ECC}]$.
4. $r \leftarrow \mathsf{x} - \mathsf{coord}(R) \pmod p$.
5. $e \leftarrow H(m)$.
6. $\langle u\rangle_F \leftarrow e + \langle x\rangle_F \cdot \langle r\rangle_F$ using **Multiply-F**.
7. $\langle v\rangle_F \leftarrow \langle u\rangle_F \cdot \langle b\rangle_F$ using **Multiply-F**.
8. Open $\langle c\rangle_F$ using **Output-F**.
9. $\langle s\rangle_F \leftarrow \langle v\rangle_F / c$.
10. Open $\langle s\rangle_F$ using **Output-F**.
11. Output (r, s).

Figure 12. Distributed EC-DSA Signing Operation

example, which will be useful later when we consider the MixNet proof of Abe, and which does present an application of these additional non-linear operations.

Suppose we want to give a non-interactive zero-knowledge proof the statement

$$\mathcal{L} = \left\{ \; x_b : T_0 = [x_0] \cdot P \text{ or } T_1 = [x_1] \cdot P \; \right\}$$

where $x_b \in \mathbb{F}_p$, for $b \in \{0, 1\}$, is the secret value. Non-interactive zero-knowledge proofs of such statements are trivial to obtain, in the random oracle model, using the OR-proof technique for Sigma protocols [4]. To fix notation for what follows it can make more sense to consider the statement as being given by

$$\left\{ \; b, x_b : T_b = [x_b] \cdot P \; \right\}$$

where $b \in \{0, 1\}$, $x_b \in \mathbb{F}_p$ are the secret values. In Figure 13 we give the standard non-interactive proof for such a statement, again we assume a hash function with codomain $\mathbb{F}_p$.

If we assume the secret inputs to the zero-knowledge proof are now distributed via our secret sharing scheme $\langle b\rangle_F, \langle x_b\rangle_F$, then we need to execute the above protocol using our elliptic curve based MPC protocol. We make use of the standard trick of multiplexing between two values depending on a hidden bit b, via

$$y_b \leftarrow b \cdot y_1 + (1 - b) \cdot y_0.$$

Our distributed protocol then can be described as in Figure 14. Note, that operations of the form $\langle x\rangle_F \leftarrow \mathbb{F}_p$ can be performed by utilizing the first two elements in a Beaver triple produced in the offline phase. Notice how in lines 4, 5 and 6 we use non-linear secret-shared operations on the curve.

5 Application to MixNets

The rest of the paper is devoted to applying the above techniques to providing a more efficient (in terms of bandwidth and verification time) for a standard ElGamal based shuffle due to Abe [1]. As remarked in the introduction more efficient single party shuffles are now known, here we are focused on providing a general n-party shuffle.

Secure shuffling consists of randomly shuffle a vector of m elements $\mathbf{v}$ using a uniformly random permutation $\pi \in S_m$ unknown to the adversary. Secure shuffling is used in several contexts such as Oblivious-RAM, secure voting, etc. Within any context, we can distinguish three sets of parties, the parties A that provide input elements $\mathbf{v}$, the parties B that shuffle $\mathbf{v}$ to get $\mathbf{v}'$, and the parties C that will use $\mathbf{v}'$. These sets are not necessarily disjoint sets. That is, it depends on the context whether a party is part of more than one set.

Prior MPC using in shuffles has primarily considered two cases: In the first case, used in [12], the data donors A provide the sensitive data $\mathbf{v}$ to the computing parties (where here $B = C$) via secret sharing. The

Non-Interactive Zero-Knowledge Proof of the Statement $\mathcal{L}$.

Proof The proof proceeds as follows:

1. If $b = 0$ then
   (a) $k_0, e_1, s_1 \leftarrow \mathbb{F}_p$
   (b) $R_0 \leftarrow [k_0] \cdot P$.
   (c) $R_1 \leftarrow [s_1] \cdot P - [e_1] \cdot T_1$.
   (d) $e \leftarrow H(R_0, R_1, T_0, T_1, P)$.
   (e) $e_0 \leftarrow e - e_1$.
   (f) $s_0 \leftarrow k_0 + e_0 \cdot x_0$.
2. Else
   (a) $k_1, e_0, s_0 \leftarrow \mathbb{F}_p$
   (b) $R_0 \leftarrow [s_0] \cdot P - [e_0] \cdot T_0$.
   (c) $R_1 \leftarrow [k_1] \cdot P$.
   (d) $e \leftarrow H(R_0, R_1, T_0, T_1, P)$.
   (e) $e_1 \leftarrow e - e_0$.
   (f) $s_1 \leftarrow k_1 + e_1 \cdot x_1$.
3. Output (e_0, e_1, s_0, s_1).

Verify Verification of the above proof is done as follows:

1. $R_0 \leftarrow [s_0] \cdot P - [e_0] \cdot T_0$.
2. $R_1 \leftarrow [s_1] \cdot P - [e_1] \cdot T_1$.
3. $e \leftarrow H(R_0, R_1, T_0, T_1, P)$.
4. Reject if $e \neq e_0 + e_1$.

Figure 13. Non-Interactive ZKPoK for the statement $\mathcal{L}$

Distributed Non-Interactive Zero-Knowledge Proof of the Statement $\mathcal{L}$.

1. $\langle k_u \rangle_F, \langle e_v \rangle_F, \langle s_v \rangle_F \leftarrow \mathbb{F}_p$
2. $\langle R_u \rangle_E \leftarrow [\langle k_u \rangle_F] \cdot P$.
3. $\langle T_v \rangle_E \leftarrow [\langle b \rangle_F] \cdot T_0 + [1 - \langle b \rangle_F] \cdot T_1$.
4. $\langle R_v \rangle_E \leftarrow [\langle s_v \rangle_F] \cdot P - [\langle e_v \rangle_F] \cdot \langle T_v \rangle_E$.
5. $\langle R_0 \rangle_E \leftarrow [\langle b \rangle_F] \cdot \langle R_v \rangle_E + [1 - \langle b \rangle_F] \cdot \langle R_u \rangle_E$
6. $\langle R_1 \rangle_E \leftarrow [\langle b \rangle_F] \cdot \langle R_u \rangle_E + [1 - \langle b \rangle_F] \cdot \langle R_v \rangle_E$
7. Open $\langle R_0 \rangle_E$ and $\langle R_1 \rangle_E$.
8. $e \leftarrow H(R_0, R_1, T_0, T_1, P)$.
9. $\langle e_u \rangle_F \leftarrow e - \langle e_v \rangle_F$.
10. $\langle s_u \rangle_F \leftarrow \langle k_u \rangle_F + \langle e_v \rangle_F \cdot \langle x_b \rangle_F$.
11. $\langle e_0 \rangle_F \leftarrow [\langle b \rangle_F] \cdot \langle e_v \rangle_F + [1 - \langle b \rangle_F] \cdot \langle e_u \rangle_F$
12. $\langle e_1 \rangle_F \leftarrow [\langle b \rangle_F] \cdot \langle e_u \rangle_F + [1 - \langle b \rangle_F] \cdot \langle e_v \rangle_F$
13. $\langle s_0 \rangle_F \leftarrow [\langle b \rangle_F] \cdot \langle s_v \rangle_F + [1 - \langle b \rangle_F] \cdot \langle s_u \rangle_F$
14. $\langle s_1 \rangle_F \leftarrow [\langle b \rangle_F] \cdot \langle s_u \rangle_F + [1 - \langle b \rangle_F] \cdot \langle s_v \rangle_F$
15. Open $\langle e_0 \rangle_F$, $\langle e_1 \rangle_F$, $\langle s_0 \rangle_F$ and $\langle s_1 \rangle_F$.
16. Output (e_0, e_1, s_0, s_1).

Figure 14. Distributed Non-Interactive ZKPoK for the statement $\mathcal{L}$

parties in B then shuffle the secret shared data with respect to a uniformly random permutation π to get $\mathbf{v}'$, and then perform computation on $\mathbf{v}'$ on behalf of a client. The permutation $\pi \in S_m$ is generated by each party $i \in B$ locally generating their own permutation π_i and then secret sharing this, with the final permutation being computed via the product $\pi = \prod \pi_i$. If the permutations are represented as permutation matrices this can be achieved by simply multiplying the secret shared permutation matrices. Active security being obtained by performing the obvious checks on the final matrix representing π, i.e. that all entries are in $\{0, 1\}$ and that the row and column sums are all equal to one.

In [11] the case of $A = B = C$ is considered for an application of Oblivious-RAM within an MPC calculation. Parties are already assumed to have secret shares of the values to be shuffled. In order to hide the data access pattern on $\mathbf{v}$, that is which component of $\mathbf{v}$ is queried at any specific point, $\mathbf{v}$ is shuffled with a uniformly random permutation. To generate π, every party i generates their own permutation π_i and (locally) transforms it into control bits for a Waksman network. Then every party secret shares its control bits among the other parties, and all permutations are evaluated in sequence. The switch with respect to a control bit being evaluated using the traditional multiplex $(x, y) \longrightarrow ((1 - b) \cdot x + b \cdot y, b \cdot x + (1 - b) \cdot y)$. To ensure active security, we check whether control bits b are in $\{0, 1\}$ by opening $b \cdot (b - 1)$.

In traditional MixNets one has that the sets A, B and C are disjoint. A MixNet works by A entering a set of input ciphertexts, consider for example ElGamal style ciphertexts over our elliptic curve group $\mathcal{G}$, i.e. the vector $\mathbf{v} = (v_i)$ consists of values of the form

$$v_i = (M_i + [k_i] \cdot Q, [k_i] \cdot P)$$

for some ElGamal public key $Q = [x] \cdot P$. We then want to shuffle these ciphertexts and output a new set of ciphertexts $\mathbf{v}'$ which are the result of the shuffle. Here we utilize the malleability of ElGamal ciphertexts to transform an encryption of a message M_i into another ciphertext encrypting the same message. Traditionally each Mixer in the MixNet performs a shuffle and provides a zero-knowledge proof that the resulting output ciphertexts are in fact the permuted (and randomized) input ciphertexts. Then the data is passed onto another Mixer which does the same operation. Thus B consists of a number of parties all of whom operate in sequence. The receiving parties C need to verify *all* of the zero-knowledge proofs from each Mixer.

In this work we examine whether one can treat B as a single multi-party mixer, and thus end up with a single zero-knowledge proof. To do this we examine the MixNet protocol of Abe [1], and cast it not as a single player protocol but as a multi-party protocol. Our protocol consists of two stages. In the first stage we generate a secret-shared permutation π, then in the second stage we utilize the permutation π to shuffle the ciphertexts and produce the zero-knowledge proofs.

Stage 1: Producing the shared permutation: In this stage each of our n parties generates a random permutation π_i, represented as a permutation matrix. They enter it into the MPC system, and the parties then multiply the permutation matrices together to form a permutation π. We then need to derive switches for a Waksman network producing the same permutation as π. We leave this step to the next section. Note that we cannot generate shared random bits and use these as the control bits for a Waksman network, as this does not result in a uniformly random permutation, as was observed in [2].

Stage 2: Producing the ciphertext permutation and proof: To perform the second step we can concentrate on what happens at a single switching gate in the Waksman network. Let the control bit for this gate be secret shared as $\langle b \rangle_F$, where $b \in \{0, 1\}$, and we assume the input ciphertexts are given by

$$\begin{aligned} v_0 &= (A_0, B_0) = (M_0 + [k_0] \cdot Q, [k_0] \cdot P), \\ v_1 &= (A_1, B_1) = (M_1 + [k_1] \cdot Q, [k_1] \cdot P). \end{aligned}$$

for some unknown messages M_0, M_1 and ephermal keys k_0 and k_1. In Abe's MixNet the output of the switching gate will be the values

$$v'_b = (\overline{A}_b, \overline{B}_b) = (A_0 + [r_0] \cdot Q, B_0 + [r_0] \cdot P),$$

$$v'_{1-b} = (\overline{A}_{1-b}, \overline{B}_{1-b}) = (A_1 + [r_1] \cdot Q, B_1 + [r_1] \cdot P),$$

plus a zero-knowledge proof of the statement that

$$\begin{aligned} &\Big(\log_Q(\overline{A}_0 - A_0) = \log_P(\overline{B}_0 - B_0) = r_0 \\ &\qquad \text{AND} \quad \log_Q(\overline{A}_1 - A_1) = \log_P(\overline{B}_1 - B_1) = r_1 \Big) \\ \text{OR} \quad &\Big(\log_Q(\overline{A}_0 - A_1) = \log_P(\overline{B}_0 - B_1) = r_1 \\ &\qquad \text{AND} \quad \log_Q(\overline{A}_1 - A_0) = \log_P(\overline{B}_1 - B_0) = r_0 \Big) \end{aligned}$$

given the secret input r_0 and r_1. This is (again) a relatively standard Sigma protocol proof, and we have already seen how to produce an (albeit simpler) zero-knowledge proof for an OR statement of equality of discrete logarithms in Section 4. For the values of r_0 and r_1 we take a Beaver triple $(\langle r_0 \rangle_F, \langle r_1 \rangle_F, \langle r_2 \rangle_F)$ from the pre-processing. These values of $\langle r_0 \rangle_F$ and $\langle r_1 \rangle_F$ are also used to generate the zero-knowledge proof. Thus we only need to produce the values of $\overline{A}_b$, $\overline{B}_b$ etc, which can be derived from the assignments

$$\begin{aligned} \langle C_0 \rangle_E &\leftarrow A_0 + [\langle r_0 \rangle_F] \cdot Q, \\ \langle D_0 \rangle_E &\leftarrow B_0 + [\langle r_0 \rangle_F] \cdot P, \\ \langle C_1 \rangle_E &\leftarrow A_1 + [\langle r_1 \rangle_F] \cdot Q, \\ \langle D_1 \rangle_E &\leftarrow B_1 + [\langle r_1 \rangle_F] \cdot P, \\ \langle \overline{A}_0 \rangle_E &\leftarrow [1 - \langle b \rangle_F] \cdot \langle C_0 \rangle_E + [\langle b \rangle_F] \cdot \langle C_1 \rangle_E, \\ \langle \overline{B}_0 \rangle_E &\leftarrow [1 - \langle b \rangle_F] \cdot \langle D_0 \rangle_E + [\langle b \rangle_F] \cdot \langle D_1 \rangle_E, \\ \langle \overline{A}_1 \rangle_E &\leftarrow [1 - \langle b \rangle_F] \cdot \langle C_1 \rangle_E + [\langle b \rangle_F] \cdot \langle C_0 \rangle_E, \\ \langle \overline{B}_1 \rangle_E &\leftarrow [1 - \langle b \rangle_F] \cdot \langle D_1 \rangle_E + [\langle b \rangle_F] \cdot \langle D_0 \rangle_E. \end{aligned}$$

We can then open $(\langle \overline{A}_0 \rangle_E, \langle \overline{B}_0 \rangle_E, \langle \overline{A}_1 \rangle_E, \langle \overline{B}_1 \rangle_E)$, and produce the zero-knowledge proof as well.

6 Generating the Waksman Control Bits

We are now left with the final problem of giving an algorithm which on input of a secret-shared permutation matrix, outputs the secret-shared control bits of the associated Waksman network. Recall we simplify the algorithm, and the network, by not having the fixed switch in the first gate of each output layer (thus increasing the number of gates by $m/2 - 1$ from a traditional Waksman network). There is a classical algorithm for this [17], but it is not obvious how to translate this to work in a data-oblivious manner. Thus in this section we show how to perform this transformation obliviously. We let M denote the permutation matrix which we start with, whose i-th row and j-th column element we refer to as $M_{i,j}$. We assume that (the shared value of) M is guaranteed on input to be a permutation matrix; which can be checked by opening the column and row sums and checking them to be equal to m, as well as opening $M_{i,j} \cdot (M_{i,j} - 1)$ and verifying it is equal to zero for all i and j.

In what follows, to explain the algorithm used to do this conversion from the matrix to control bits, and how we realized it with MPC, we will use a running example which is the permutation $\tilde{\pi}$ of matrix given earlier:

$$\tilde{M} = \begin{bmatrix} 0&1&0&0&0&0&0&0 \\ 1&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&1 \\ 0&0&0&1&0&0&0&0 \\ 0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&1&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&0&0&0&0&0&1&0 \end{bmatrix}$$

The high level idea behind the algorithm uses the fact that a Waksman network has a recursive structure, hence finding the control bits for a given permutation π can be done recursively. From π we determine two permutations π^1, π^2 for the two subnetworks, as well as control bits for the inward and outward layers, such that the composition of these realizes π. Then we apply the same process on π^1, π^2 and so on, till a control bit is determined for every switch. The process proceeds in the three steps:

- **Step One:** The first step consists of taking the $m \times m$ permutation matrix M of π, and merging coordinates corresponding to the same input or output switch, e.g, the first and second rows correspond to the first inward switch thus they will be merged. The first and second columns correspond to the first outward switch, they will be merged as well. Thus, this will result in an $m/2 \times m/2$ matrix M' such that $M'_{i,j} = M_{2\cdot i-1,2\cdot j-1} + M_{2\cdot i,2\cdot j-1} + M_{2\cdot i-1,2\cdot j} + M_{2\cdot i,2\cdot j}$. For instance, for the permutation $\tilde{\pi}$, $\tilde{M'}$ will be

$$\tilde{M'} = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 2 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

- **Step Two:** In the second step, we construct two permutation matrices M^1 and M^2 such that $M^1_{i,j} + M^2_{i,j} = M'_{i,j}$. Those matrices will be the ones corresponding respectively to π^1 and π^2, the permutations of the two sub networks. For our example, $\tilde{M^1}$ and $\tilde{M^2}$ can be (one has a choice obviously)

$$\tilde{M^1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \tilde{M^2} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

- **Step Three:** The last step is then setting the control bits for the inward and outward layers. The algorithm does this by identifying the coordinates of the entries in M, that correspond to the entries in M^1 which are equal to one. So for our example $\tilde{M}^1_{2,2} = 1$ as this entry corresponds to the one in position $(4, 4)$ of $\tilde{M}$, therefore the coordinates $(4, 4)$ are identified. Note that for some entries, two coordinates can be identified instead of one, which is the case for $\tilde{M}^1_{1,1}$ and $\tilde{M}^1_{3,3}$ in our example. When this happens, one of the coordinates is identified (which one does not matter for the algorithm) As such, for example, the coordinates $(2, 1), (4, 4), (5, 5)$ and $(8, 7)$ could be identified.
  Then within each of these coordinates, if there is an even component, the associated switch to this component is identified and its control bit is set to be one. So for our example, the coordinates $(2, 1)$ contain an even component (in the first position) and thus the associated switch to this even component is the switch one, i.e. $2/2$, in the inward layer, i.e. control one bit is set to one.
  Similarly, the coordinates $(4, 4)$, and $(8, 7)$ contain even components, and so the associated switches to these even components are $2 = 4/2$ and $4 = 8/2$ in the inward layer and $2 = 4/2$ in the outward layer, and their associated control bits are also set to one.
  The control bit of all the remaining switches in the inward and outward layers are set to zero.

This process is repeated recursively on the sub networks, untill we reach the subnetworks of size 2 which are simply switches. For such switches, if the corresponding matrix is the identity matrix, we set the switch to be zero, otherwise, we set it to be one. Figure 6 illustrates the resulting realization of the permutation $\tilde{\pi}$.

Making the algorithm suitable for MPC implementation: Recall, our aim is to transform a secret shared permutation matrix M of π, into secret shared control bits realizing it. We can achieve this if we somehow manage to transform the above algorithm into arithmetic operations.

The first step of the algorithm is easy to transform, as constructing M' is by definition done by summing up entries from M of known coordinates, and is thus a local operation when M is presented in secret shared form.

The last step is also relatively easy to transform; it consists of comparing entries from M^1 with entries from M, then checking whether coordinates contain even components. As such, the control bit b_k^{in} for the switch k in the inward layer and b_k^{out} in the outer layer can be computed as

$$b_k^{in} = \sum_{j=1}^{j=m/2} M_{k,j}^1 \cdot (M_{2\cdot k,2\cdot j-1} + M_{2\cdot k,2\cdot j}),$$
$$b_k^{out} = \sum_{i=1}^{i=m/2} M_{i,k}^1 \cdot (M_{2\cdot i-1,2\cdot k} \cdot (1 - M_{2\cdot i,2\cdot k-1}) + M_{2\cdot i,2\cdot k}).$$

The control bit corresponding to a subnetwork of size two can be computed as

$$b^{mid} = M_{1,2},$$

as we have $M = I_2$ if no switch occurs and M is the off-diagonal 2×2 matrix if a switch occurs.

The second step is the most intricate one to transform, given that we need to split a secret shared matrix M' into two secret-shared permutation matrices M^1 and M^2. Our idea for this step is to express constraints on M^1 and M^2 into equations, where variables are entries of M^1 and M^2. Therefore, solving these equations identifies M^1 and M^2. This solution is then accomplished by making use of the fact that the entries are integers in $\{0, 1\}$, which constrains their possible value considerably.

The first constraint on M^1 and M^2 is that they sum up to M', i.e. for $i, j \in \{1, \ldots, m/2\}$ we have

$$M_{i,j}^1 + M_{i,j}^2 = M'_{i,j}$$

The second constraint on M^1 and M^2 is that they are permutation matrices, which translates into the set of linears equations for $j \in \{1, \ldots, m/2\}$ and $k \in \{1, 2\}$

$$\sum_{i=1}^{m/2} M_{i,j}^k = 1 \quad \text{and} \quad \sum_{i=1}^{m/2} M_{j,i}^k = 1,$$

as well as the quadratic equations for $i, j \in \{1, \ldots, m/2\}$ and $k \in \{1, 2\}$

$$M_{i,j}^k \cdot (M_{i,j}^k - 1) = 0.$$

To find M^1 and M^2, the strategy will be to solve the linear equations, while allowing entries $M_{i,j}^k$ to have values only in $\{0, 1\}$, which thus caters for the quadratic equations.

We do this by first initializing the matrices M^1, M^2, O^1 and O^2 by having their entries equal to zero. The matrices M^1 and M^2 will contain at the end the permutation matrices of π^1 and π^2, as soon as we fix an entry in $M_{i,j}^k$ we set $O_{i,j}^k$ equal to one. This is represented by the algorithm $\mathsf{Init}(M^1, M^2, O^1, O^2)$ in Figure 15

The next step of the process consists of setting entries $M_{i,j}^k$ that have only one solution, see the function $\mathsf{StartFix}(M', M^1, M^2, O^1, O^2)$ in Figure 15. When we have $M'_{i,j} = 0$ (resp. 2) then we know that the values $M_{i,j}^k$ must be equal to zero (resp. one), since these are the only ways integers in $\{0, 1\}$ can add up to zero (resp. two).

For our example, after the execution of $\mathsf{StartFix}$ our values of $\tilde{O}_1$ and $\tilde{O}_2$ become

$$\tilde{O}^1 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \qquad \tilde{O}^2 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

If M^k are not fully determined at this stage, we need to deal with entries corresponding to one's in M'. Having a one in $M'_{i,j}$ means that one of the entries $M_{i,j}^1, M_{i,j}^2$ is equal to one, and the other equals zero. We

will take (as a free choice) coordinates (i,j) of the first entry in M' that is equal to one, and we will set $M^1_{i,j}$ to be one. See procedure MakeChoice in Figure 15. In our example, after making such a choice, $\tilde{O}_1$ and $\tilde{O}_2$ become

$$\tilde{O}^1 = \begin{bmatrix} 1&1&1&1 \\ 1&1&1&0 \\ 1&1&1&1 \\ 1&0&1&0 \end{bmatrix} \qquad \tilde{O}^2 = \begin{bmatrix} 1&1&1&1 \\ 1&1&1&0 \\ 1&1&1&1 \\ 1&0&1&0 \end{bmatrix}$$

Init(M^1, M^2, O^1, O^2):
1. For $k \in \{1,2\}$ and $i,j \in \{1,\ldots,\frac{m}{2}\}$:
   (a) $M^k_{i,j} \leftarrow 0$, $O^k_{i,j} \leftarrow 0$.

StartFix(M', M^1, M^2, O^1, O^2):
1. For $i,j \in \{1,\ldots,\frac{m}{2}\}$:
   (a) If $(M'_{i,j} = 0)$ then set $O^1_{i,j} \leftarrow 1$, $O^2_{i,j} \leftarrow 1$.
   (b) If $(M'_{i,j} = 2)$ then set $M^1_{i,j} \leftarrow 1$, $O^1_{i,j} \leftarrow 1$, $M^2_{i,j} \leftarrow 1$, $O^2_{i,j} \leftarrow 1$.

MakeChoice(M', M^1, M^2, O^1, O^2):
1. For $i,j \in \{1,\ldots,\frac{m}{2}\}$
   (a) If $(M'_{i,j} = 1$ and $O^1_{i,j} = 0)$ then set $M^1_{i,j} \leftarrow 1$, $M^2_{i,j} \leftarrow 0$, $O^1_{i,j} \leftarrow 1$, $O^2_{i,j} \leftarrow 1$ and return.

Update(M^1, M^2, O^1, O^2):
1. For $i,j \in \{1,\ldots,\frac{m}{2}\}$:
   (a) If $(O^1_{i,j} = 0)$
      i. If $(\prod_{\substack{k=1 \\ k\neq j}}^{\frac{m}{2}} O^1_{i,k} = 1)$ then set $M^1_{i,j} \leftarrow 1 - (\sum_{\substack{k=1 \\ k\neq j}}^{\frac{m}{2}} M^1_{i,k})$, $O^1_{i,j} \leftarrow 1$, $M^2_{i,j} \leftarrow 1 - M^1_{i,j}$ and $O^2_{i,j} \leftarrow 1$.
   (b) If $(O^1_{i,j} = 0)$
      i. If $(\prod_{\substack{k=1 \\ k\neq i}}^{\frac{m}{2}} O^1_{k,j} = 1)$ then set $M^1_{i,j} \leftarrow 1 - (\sum_{\substack{k=1 \\ k\neq i}}^{\frac{m}{2}} M^1_{k,j})$, $O^1_{i,j} \leftarrow 1$, $M^2_{i,j} \leftarrow 1 - (M^1_{i,j})$ and $O^2_{i,j} \leftarrow 1$.

Waksman-Sub(M'):
1. Init(M^1, M^2, O^1, O^2).
2. StartFix(M', M^1, M^2, O^1, O^2).
3. For $c \in \{1,\ldots,\frac{m}{4}\}$
   (a) MakeChoice(M', M^1, M^2, O^1, O^2) .
      i. For $k \in \{1,\ldots,m+1-4\cdot c\}$
         A. Update(M^1, M^2, O^1, O^2).

Figure 15. Waksman Algorithm Step 2 Sub-Procedures

Making a choice will fix one entry in M^k, and fixing an entry in M^k will allow us to fix other entries, see sub-procedure Update in Figure 15. For our example, after executing Update, all entries $\tilde{O}^k_{i,j}$ are equal to one and therefore M^1, M^2 are fully determined. However, executing Update only once does not always guarantee to fix all entries that can be fixed with respect to the choice made, in addition making one choice does not guarantee that all entries will be fixed. That is, some permutations require repeated application of MakeChoice and Update until all the values $\tilde{O}^k_{i,j}$ are equal to one. Then we need to determine the bounds of how many times we should iterate these steps.

At the end of the execution of StartFix, in each row i (resp. column j) of M^k, either $m/2-2$ entries are fixed (which is the case where the row i (resp. column j) in M' contained two one's), or $m/2$ entries are

fixed (which is the case where the row i (resp. the column j) in M' contained an entry that is equal to two). Thus, at most n entries in M^k remain unfixed.

Recall once we make a choice to fix one entry in M^k, this allows us to fix other entries. That is, at least row i and column j in M^k will now contain only one entry that is not fixed, respectively $M^k_{i,f}$ and $M^k_{f',j}$. This is because each row (resp. column) in O^k can contain at most two zero entries before fixing the value (i, j). These two entries will themselves fix one entry in row f' and one entry in column f (if $f = f'$ only one entry will be fixed). Therefore, making a choice and updating with respect to it will fix at least four other entries, with the minimum occuring when $f = f'$. Thus, we will need to execute MakeChoice at most $m/4$ times.

Assuming that the first execution of Update after MakeChoice will fix at least three entries, and each execution of Update after the first execution will at least fix one entry of the entries that can be fixed. Also the choice we made may fix the whole of the matrices M^k. Thus, we will need to execute Update $m+1-4\cdot c$ times where c is the number of choices already made. This gives the final procedure for the second step in sub-algorithm Waksman-Sub in Figure 15.

Transforming the algorithm: Having produced an (almost) data-oblivious methodology to generate a Waksman network we now need to transform it into a fully data-oblivious methodology by replacing all operations with algebraic operations. Again step one and three are trivial, thus we are left with step two. The key step is dealing with the conditional operations, but this can be easily transformed into algebraic operations as follows.

At various points we need to determine whether a value in $\{0, 1, 2\}$ is equal to zero, one or two. This can be done via algebraic operations using three simple quadratic functions, namely

$$\begin{aligned} Q_0(x) &= (x-1)\cdot(x-2)/2, \\ Q_1(x) &= -x\cdot(x-2), \\ Q_2(x) &= x\cdot(x-1)/2. \end{aligned}$$

Given these functions converting Figure 15 into a secret shared format is relatively simple; which we give in Figure 16. That the algorithm implements the conversion from a permutation to a set of Waksman switches in a data-oblivious manner is then immediate, and thus the algorithm when executed via an actively secure MPC system will be an actively secure rendition of the said algorithm also follows from the security of the underlying MPC protocol.

Complexity of the algorithm: Given that the realization of Multiply-F requires interaction between players whereas addition is done locally, the complexity of our method is dominated by the number of multiplications between secret shared values. We present in Table 1 the number of multiplications between secret shared values for the sub-procedures of each step of the algorithm, as well as the number of times each sub-procedure is executed within one recursion. From these formulae it is easy to evaluate the total number of multiplications between shared secret values for any value of m for the whole algorithm. We present such a summary in Table 2 for small values of m. As one can easily see, the number of multiplications grows super-polynomially with m. However the multiplicative depth of the computation only grows logarithmically with m, hence, these multiplications can be efficiently parallelized.

Init($\langle M^1\rangle_F, \langle M^2\rangle_F, \langle O^1\rangle_F, \langle O^2\rangle_F$):

1. For $k \in \{1,2\}$ and $i,j \in \{1,\ldots,\frac{m}{2}\}$:
   (a) $\langle M^k_{i,j}\rangle_F \leftarrow 0$, $\langle O^k_{i,j}\rangle_F \leftarrow 0$.

StartFix($\langle M'\rangle_F, \langle M^1\rangle_F, \langle M^2\rangle_F, \langle O^1\rangle_F, \langle O^2\rangle_F$):

1. For $i,j \in \{1,\ldots,\frac{m}{2}\}$:
   (a) $\langle O^1_{i,j}\rangle_F \leftarrow Q_0(\langle M'_{i,j}\rangle_F) + Q_2(\langle M'_{i,j}\rangle_F)$.
   (b) $\langle O^2_{i,j}\rangle_F \leftarrow Q_0(\langle M'_{i,j}\rangle_F) + Q_2(\langle M'_{i,j}\rangle_F)$.
   (c) $\langle M^1_{i,j}\rangle_F \leftarrow Q_2(\langle M'_{i,j}\rangle_F)$.
   (d) $\langle M^2_{i,j}\rangle_F \leftarrow Q_2(\langle M'_{i,j}\rangle_F)$.

MakeChoice($\langle M'\rangle_F, \langle M^1\rangle_F, \langle M^2\rangle_F, \langle O^1\rangle_F, \langle O^2\rangle_F$):

1. $\langle d\rangle_F \leftarrow 1$.
2. For $i,j \in \{1,\ldots,\frac{m}{2}\}$
   (a) $\langle b\rangle_F \leftarrow \langle d\rangle_F \cdot Q_1(\langle M'_{i,j}\rangle_F) \cdot (1 - \langle O^1_{i,j}\rangle_F)$.
   (b) $\langle M^1_{i,j}\rangle_F \leftarrow (1 - \langle b\rangle_F) \cdot \langle M^1_{i,j}\rangle_F + \langle b\rangle_F$.
   (c) $\langle O^1_{i,j}\rangle_F \leftarrow (1 - \langle b\rangle_F) \cdot \langle O^1_{i,j}\rangle_F + \langle b\rangle_F$.
   (d) $\langle O^2_{i,j}\rangle_F \leftarrow (1 - \langle b\rangle_F) \cdot \langle O^2_{i,j}\rangle_F + \langle b\rangle_F$.
   (e) $\langle d\rangle_F \leftarrow \langle d\rangle_F \cdot (1 - \langle b\rangle_F)$.

Update($\langle M^1\rangle_F, \langle M^2\rangle_F, \langle O^1\rangle_F, \langle O^2\rangle_F$):

1. For $i,j \in \{1,\ldots,\frac{m}{2}\}$:
   (a) $\langle b\rangle_F \leftarrow (1 - \langle O^1_{i,j}\rangle_F) \cdot \prod_{\substack{k=1 \\ k\neq j}}^{\frac{m}{2}} \langle O^1_{i,k}\rangle_F$.
   (b) $\langle M^1_{i,j}\rangle_F \leftarrow (1 - \langle b\rangle_F) \cdot \langle M^1_{i,j}\rangle_F + \langle b\rangle_F \cdot (1 - (\sum_{\substack{k=1 \\ k\neq j}}^{\frac{m}{2}} \langle M^1_{i,k}\rangle_F))$.
   (c) $\langle M^2_{i,j}\rangle_F \leftarrow (1 - \langle b\rangle_F) \cdot \langle M^2_{i,j}\rangle_F + \langle b\rangle_F \cdot (1 - \langle M^1_{i,j}\rangle_F)$.
   (d) $\langle O^1_{i,j}\rangle_F \leftarrow (1 - \langle b\rangle_F) \cdot \langle O^1_{i,j}\rangle_F + \langle b\rangle_F$.
   (e) $\langle O^2_{i,j}\rangle_F \leftarrow (1 - \langle b\rangle_F) \cdot \langle O^2_{i,j}\rangle_F + \langle b\rangle_F$.
   (f) $\langle b\rangle_F \leftarrow (1 - \langle O^1_{i,j}\rangle_F) \cdot \prod_{\substack{k=1 \\ k\neq i}}^{\frac{m}{2}} \langle O^1_{k,j}\rangle_F$.
   (g) $\langle M^1_{i,j}\rangle_F \leftarrow (1 - \langle b\rangle_F) \cdot \langle M^1_{i,j}\rangle_F + \langle b\rangle_F \cdot (1 - (\sum_{\substack{k=1 \\ k\neq i}}^{\frac{m}{2}} \langle M^1_{k,j}\rangle_F))$.
   (h) $\langle M^2_{i,j}\rangle_F \leftarrow (1 - \langle b\rangle_F) \cdot \langle M^2_{i,j}\rangle_F + \langle b\rangle_F \cdot (1 - \langle M^1_{i,j}\rangle_F)$.
   (i) $\langle O^1_{i,j}\rangle_F \leftarrow (1 - \langle b\rangle_F) \cdot \langle O^1_{i,j}\rangle_F + \langle b\rangle_F$.
   (j) $\langle O^2_{i,j}\rangle_F \leftarrow (1 - \langle b\rangle_F) \cdot \langle O^2_{i,j}\rangle_F + \langle b\rangle_F$.

Waksman-Sub($\langle M'\rangle_F$):

1. Init($\langle M^1\rangle_F, \langle M^2\rangle_F, \langle O^1\rangle_F, \langle O^2\rangle_F$).
2. StartFix($\langle M'\rangle_F, \langle M^1\rangle_F, \langle M^2\rangle_F, \langle O^1\rangle_F, \langle O^2\rangle_F$).
3. For $c \in \{1,\ldots,\frac{m}{4}\}$
   (a) MakeChoice($\langle M'\rangle_F, \langle M^1\rangle_F, \langle M^2\rangle_F, \langle O^1\rangle_F, \langle O^2\rangle_F$) .
      i. For $k \in \{1,\ldots,m+1-4\cdot c\}$
         A. Update($\langle M^1\rangle_F, \langle M^2\rangle_F, \langle O^1\rangle_F, \langle O^2\rangle_F$).

Figure 16. Secret Shared Waksman Algorithm Step 2 Sub-Procedures

Table 1. Number of multiplications and invocations for sub-procedures for one recursion.

	Step 1	Step 2			Step 3	
		StartFix	MakeChoice	Update	Comp. inward bits	Comp. outward bits
No. of mult.	0	$3 \cdot m^2/2$	$7 \cdot m^2/4$	$(m+10) \cdot m^2/4$	$m^2/4$	$m^2/2$
No. of exec.	1	1	$m/4$	$m \cdot (m-2)/8$	1	1

Table 2. Number of multiplications between secret shared values for various values of m.

m	No. of multiplications needed
2	0
4	120
8	$2096 + 2 \cdot 120 = 2336$
16	$48960 + 2 \cdot 2336 = 53632$
32	$1306880 + 2 \cdot 53632 = 1414144$
64	$37708800 + 2 \cdot 1414144 = 40537088$
128	$1140494336 + 2 \cdot 40537088 = 1221568512$
256	$35430481920 + 2 \cdot 1221568512 = 37873618944$
512	$1116666920960 + 2 \cdot 37873618944 = 1192414158848$

Acknowledgements

This work has been supported in part by ERC Advanced Grant ERC-2015-AdG-IMPaCT, by the Defense Advanced Research Projects Agency (DARPA) and Space and Naval Warfare Systems Center, Pacific (SSC Pacific) under contracts No. N66001-15-C-4070 and FA8750-19-C-0502, and by the FWO under an Odysseus project GOH9718N. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the ERC, United States Air Force, DARPA or FWO.

References

1. Abe, M.: Mix-networks on permutation networks. In: Lam, K.Y., Okamoto, E., Xing, C. (eds.) Advances in Cryptology – ASIACRYPT'99. Lecture Notes in Computer Science, vol. 1716, pp. 258–273. Springer, Heidelberg, Germany, Singapore (Nov 14–18, 1999)
2. Abe, M., Hoshino, F.: Remarks on mix-network based on permutation networks. In: Kim, K. (ed.) PKC 2001: 4th International Workshop on Theory and Practice in Public Key Cryptography. Lecture Notes in Computer Science, vol. 1992, pp. 317–324. Springer, Heidelberg, Germany, Cheju Island, South Korea (Feb 13–15, 2001)
3. Bayer, S., Groth, J.: Efficient zero-knowledge argument for correctness of a shuffle. In: Pointcheval, D., Johansson, T. (eds.) Advances in Cryptology – EUROCRYPT 2012. Lecture Notes in Computer Science, vol. 7237, pp. 263–280. Springer, Heidelberg, Germany, Cambridge, UK (Apr 15–19, 2012)
4. Cramer, R., Damgård, I., Schoenmakers, B.: Proofs of partial knowledge and simplified design of witness hiding protocols. In: Desmedt, Y. (ed.) Advances in Cryptology – CRYPTO'94. Lecture Notes in Computer Science, vol. 839, pp. 174–187. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 21–25, 1994)
5. Damgård, I., Keller, M., Larraia, E., Pastro, V., Scholl, P., Smart, N.P.: Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In: Crampton, J., Jajodia, S., Mayes, K. (eds.) ESORICS 2013: 18th European Symposium on Research in Computer Security. Lecture Notes in Computer Science, vol. 8134, pp. 1–18. Springer, Heidelberg, Germany, Egham, UK (Sep 9–13, 2013)

6. Damgård, I., Pastro, V., Smart, N.P., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Safavi-Naini, R., Canetti, R. (eds.) Advances in Cryptology – CRYPTO 2012. Lecture Notes in Computer Science, vol. 7417, pp. 643–662. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 19–23, 2012)
7. Doerner, J., Kondi, Y., Lee, E., shelat, a.: Secure two-party threshold ECDSA from ECDSA assumptions. In: 2018 IEEE Symposium on Security and Privacy. pp. 980–997. IEEE Computer Society Press, San Francisco, CA, USA (May 21–23, 2018)
8. Fauzi, P., Lipmaa, H., Siim, J., Zajac, M.: An efficient pairing-based shuffle argument. In: Takagi, T., Peyrin, T. (eds.) Advances in Cryptology – ASIACRYPT 2017, Part II. Lecture Notes in Computer Science, vol. 10625, pp. 97–127. Springer, Heidelberg, Germany, Hong Kong, China (Dec 3–7, 2017)
9. Fauzi, P., Lipmaa, H., Zajac, M.: A shuffle argument secure in the generic model. In: Cheon, J.H., Takagi, T. (eds.) Advances in Cryptology – ASIACRYPT 2016, Part II. Lecture Notes in Computer Science, vol. 10032, pp. 841–872. Springer, Heidelberg, Germany, Hanoi, Vietnam (Dec 4–8, 2016)
10. González, A., Ràfols, C.: New techniques for non-interactive shuffle and range arguments. In: Manulis, M., Sadeghi, A.R., Schneider, S. (eds.) ACNS 16: 14th International Conference on Applied Cryptography and Network Security. Lecture Notes in Computer Science, vol. 9696, pp. 427–444. Springer, Heidelberg, Germany, Guildford, UK (Jun 19–22, 2016)
11. Keller, M., Scholl, P.: Efficient, oblivious data structures for MPC. In: Sarkar, P., Iwata, T. (eds.) Advances in Cryptology – ASIACRYPT 2014, Part II. Lecture Notes in Computer Science, vol. 8874, pp. 506–525. Springer, Heidelberg, Germany, Kaoshiung, Taiwan, R.O.C. (Dec 7–11, 2014)
12. Laur, S., Willemson, J., Zhang, B.: Round-efficient oblivious database manipulation. In: Lai, X., Zhou, J., Li, H. (eds.) ISC 2011: 14th International Conference on Information Security. Lecture Notes in Computer Science, vol. 7001, pp. 262–277. Springer, Heidelberg, Germany, Xi'an, China (Oct 25–29, 2011)
13. Lindell, Y.: Fast secure two-party ECDSA signing. In: Katz, J., Shacham, H. (eds.) Advances in Cryptology – CRYPTO 2017, Part II. Lecture Notes in Computer Science, vol. 10402, pp. 613–644. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 20–24, 2017)
14. Lindell, Y., Nof, A.: Fast secure multiparty ECDSA with practical distributed key generation and applications to cryptocurrency custody. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) ACM CCS 2018: 25th Conference on Computer and Communications Security. pp. 1837–1854. ACM Press, Toronto, ON, Canada (Oct 15–19, 2018)
15. Lindell, Y., Nof, A., Ranellucci, S.: Fast secure multiparty ECDSA with practical distributed key generation and applications to cryptocurrency custody. IACR Cryptology ePrint Archive 2018, 987 (2018), https://eprint.iacr.org/2018/987
16. Smart, N.P., Wood, T.: Error detection in monotone span programs with application to communication-efficient multi-party computation. In: Matsui, M. (ed.) Topics in Cryptology – CT-RSA 2019. Lecture Notes in Computer Science, vol. 11405, pp. 210–229. Springer, Heidelberg, Germany, San Francisco, CA, USA (Mar 4–8, 2019)
17. Waksman, A.: A permutation network. J. ACM 15(1), 159–163 (1968)

A Basic Functionalities

The ideal commitment functionality is given in Figure 17 and a realisation in the random oracle model is given in Figure 18.

The Ideal Functionality $\mathcal{F}_{\mathsf{Commit}}$

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

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

Figure 17. The Ideal Functionality for Commitments

The Protocol Π_{Commit}

Commit:

1. In order to commit to v, $\mathcal{P}_i$ sets $o \leftarrow v||r$, where r is chosen uniformly in a determined domain, and queries the Random Oracle $\mathcal{H}$ to get $c \leftarrow \mathcal{H}(o)$.
2. Player $\mathcal{P}_i$ then broadcasts (c, i, τ_v), where τ_v represents a handle for the commitment.

Open:

1. In order to open a commitment (c, i, τ_v), where $c = \mathcal{H}(v||r)$, player $\mathcal{P}_i$ broadcasts $(o = v||r, i, \tau_v)$.
2. All players call $\mathcal{H}$ on o and check whether $\mathcal{H}(o) = c$. Players accept if and only if this check passes.

Figure 18. The Protocol for Commitments.

B Proof of Theorem 1

We design here a similar game, see Figure 19, to the one used in SPDZ, in order to prove that the MAC check is correct, and that the soundness is negligible. The second step in the game models the fact that corrupted players can lie about their shares of values opened throughout the protocol execution, whilst Δ models the fact that the adversary can introduce errors on the MAC values.

The MACCheck Security Game

1. The challenger generates the secret key $\alpha \leftarrow \alpha_1 + \ldots + \alpha_n$ and MACs $\gamma_i[a_j] \leftarrow \alpha \cdot a_j$ and $\Gamma_i[A_k] \leftarrow [\alpha] \cdot A_k$.
2. The challenger sends messages $a_1 \ldots a_t, A_1, \ldots A_u$ to the adversary.
3. The adversary sends back messages $a'_1 \ldots a'_t, A'_1, \ldots A'_u$.
4. The challenger generates random values $r_1, \ldots r_t, r_{t+1}, \ldots r_{t+u}$.
5. The adversary provides an error $\Delta \in E(K)$.
6. The challenger computes

$$a \leftarrow \sum_{j=1}^{t} r_j \cdot a'_j,$$

$$v_i \leftarrow \sum_{j=1}^{t} r_j \cdot \gamma_i[a_j],$$

$$A \leftarrow \sum_{k=1}^{u} r_{k+t} \cdot [A'_k],$$

$$\Gamma_i \leftarrow \sum_{k=1}^{u} [r_{t+k}] \cdot \Gamma_i[A_k],$$

$$W_i \leftarrow [v_i - \alpha_i \cdot a] \cdot P + \Gamma_i - [\alpha_i] \cdot A.$$

7. The challenger checks whether $W_1 + \ldots + W_n = \Delta$
8. The adversary wins if there is an i for which $a'_i \neq a_i$ or $A'_i \neq A_i$, and yet the challengers check passes.

Figure 19. MACCheck Security Game

We now compute the probability of passing the check. The check goes through if the following equalities hold:

$$\Delta = \sum_{i=1}^{n} W_i = \sum_{i=1}^{n} \left([v_i - \alpha_i \cdot a] \cdot P + \Gamma_i - [\alpha_i] \cdot A \right)$$

$$
\begin{aligned}
&= \sum_{i=1}^{n} \Big(\Big[\sum_{j=1}^{t} r_j \cdot \gamma_i[a_j] - \alpha_i \cdot \sum_{j=1}^{t} r_j \cdot a'_j \Big] \cdot P \\
&\qquad + \sum_{k=1}^{u} [r_{t+k}] \cdot \Gamma_i[A_k] - [\alpha_i] \cdot \sum_{k=1}^{u} [r_{t+k}] \cdot A'_k \Big) \\
&= \sum_{i=1}^{n} \Big(\Big[\sum_{j=1}^{t} \big(r_j \cdot \gamma_i[a_j] - \alpha_i \cdot r_j \cdot a'_j \big) \Big] \cdot P \\
&\qquad + \sum_{k=1}^{u} \Big([r_{t+k}] \cdot \Gamma_i[A_k] - [\alpha_i] \cdot [r_{t+k}] \cdot A'_k \Big) \Big) \\
&= \sum_{j=1}^{t} \left([r_j \cdot \sum_{i=1}^{n} \big(\gamma_i[a_j] - \alpha_i \cdot a'_j \big)] \cdot P \right) \\
&\qquad + \sum_{k=1}^{u} \left([r_{t+k}] \cdot \sum_{i=1}^{n} \big(\Gamma_i[A_k] - [\alpha_i] \cdot A'_k \big) \right) \\
&= \sum_{j=1}^{t} \big([r_j \cdot \big(\alpha \cdot a_j - \alpha \cdot a'_j \big)] \cdot P \big) + \sum_{k=1}^{u} \left([r_{t+k}] \cdot ([\alpha] \cdot A_k - [\alpha] \cdot A'_k) \right) \\
&= [\alpha \cdot \sum_{j=1}^{t} r_j \cdot \big(a_j - a'_j \big)] \cdot P + [\alpha] \cdot \sum_{k=1}^{u} [r_{t+k}] \cdot (A_k - A'_k) \\
&= [\alpha] \cdot \sum_{j=1}^{t+u} [r_j \cdot b_j] \cdot B_j
\end{aligned}
$$

Where

$$
b_j = \begin{cases} (a_j - a'_j) & \text{for } j \in \{1, \ldots, t\} \\ 1 & \text{for } j \in \{t+1, \ldots, t+u\} \end{cases}
$$

and

$$
B_j = \begin{cases} P & \text{for } j \in \{1, \ldots, t\} \\ A_{j-t} - A'_{j-t} & \text{for } j \in \{t+1, \ldots, t+u\} \end{cases}
$$

Note that r_j, b_j and B_j are known for the adversary. So the last equality can be written as $\Delta = [\alpha] \cdot N$ where N is known to the adversary. Here we distinguish two cases:

- $N \neq 0_{\mathcal{G}}$: in this case, passing the check is equivalent to guessing α, which is, for the adversary, a random number sampled from $\mathbb{F}_p$. Therefore, passing the check can happen with probability $1/p$.
- $N = 0_{\mathcal{G}}$: in this case, the adversary will pass the check, however, $N = 0_{\mathcal{G}}$ can happen with only a small probability (if the adversary is dishonest), which we now show (following the same argument presented in [5,6]). Since $N = 0_{\mathcal{G}}$ if and only if $\sum_{j=1}^{t+u} [r_j] \cdot ([b_j] \cdot B_j) = 0_{\mathcal{G}}$, we define

  $$
  r = (r_1, \ldots, r_{t+u}) \quad \text{and} \quad M = ([b_1] \cdot B_1, \ldots, [b_{t+u}] \cdot B_{t+u}),
  $$

  along with the linear map

  $$
  f_M(r) \leftarrow [r] \cdot M = \sum_{j=1}^{t+u} [r_j] \cdot M_j.
  $$

  Note, that f_M is not the identically zero mapping because (assuming a dishonest adversary) there is at least one j s.t $a_j \neq a'_j$ or $A_j \neq A'_j$. Following the rank-nullity theorem we can then conclude that $\dim(ker(f_M)) = t + u - 1$. Also since r is randomly chosen and given to the adversary after choosing a'_j and A'_k, the probability of $r \in ker(f_M)$ is $|\mathbb{F}_p^{t+u-1}|/|F_p^{t+u}| = 1/p$.

So to conclude, the probability of wining the game is at most $2/p$.

As for correctness, a honest prover will win the game with probability one (this is the case where $a_j = a'_j$ for $j \in [1, \ldots, t]$, $A_k = A'_k$ for $k \in [1, \ldots, u]$, and $\Delta = 0$.)