Constant-Overhead Secure Computation for Boolean Circuits in the Preprocessing Model

Ivan Damgård, Sarah Zakarias

Dept. of Computer Science, Aarhus Univesity

Abstract. We present a protocol for securely computing a Boolean circuit C in presence of a dishonest and malicious majority. The protocol is unconditionally secure, assuming access to a preprocessing functionality that is not given the inputs to compute on. For a large number of players the work done by each player is the same as the work needed to compute the circuit in the clear, up to a constant factor. Our protocol is the first to obtain these properties for Boolean circuits. On the technical side, we develop new homomorphic authentication schemes based on asymptotically good codes with an additional multiplication property. We also show a new algorithm for verifying the product of Boolean matrices in quadratic time with exponentially small error probability, where previous methods would only give a constant error.

1 Introduction

In multiparty computation, a set of players each holding a private input wish to compute an agreed function such that the intended result is the only new information that is revealed. This must hold also if some subset of the players are corrupted by an adversary. Even in the most difficult case where all but one player can be corrupt (aka. dishonest majority), it is known that any efficiently computable function can be computed securely, under appropriate complexity assumptions [CLOS02], also when asking for universal composable security [Can01]. This leads naturally to the question of what the price of security is, i.e., how much extra resources must we invest to compute the function securely, as opposed to just computing it?

The case of dishonest majority can be handled by different approaches. It is well known that using fully homomorphic encryption, the communication complexity can be made very small, and only needs to depend on the number of inputs and outputs of the function. Although the computational overhead is usually very large, for some cryptosystems even this can be made small, in fact poly-logarithmic in the security parameter, see [GHS12]. However, the exponent hidden in the poly-logarithmic notation is quite large because the approach requires so-called bootstrapping and currently does not yield practical protocols. The overhead can also be made small using "MPC-in-the-head" techniques [IKOS08], at least for a constant number of parties, but still using (somewhat non-standard) computational assumptions. Furthermore in these cases, only computational security can be obtained.

On the other hand, if we allow a preprocessing phase where the inputs to the function need not be known, then unconditional, active security can be obtained, and the complexity can be reduced to the point where we get eminently practical protocols. This was first demonstrated by Bendlin et al. [BDOZ10], who showed that given a preprocessing functionality, an arithmetic circuit over a large field $\mathbb{F}_q$ can be computed with unconditional security and very efficiently: if the number of players is constant and $\log q \in \Omega(\kappa)$, where κ is the security parameter, then the total computational work invested by each player is a constant times the work one needs to compute the same circuit in the clear. More recently, Damgård et al.[DPSZ11] improved on this result by showing the same

for *any* number of players. In both cases, the preprocessing works independently of the inputs, and simply produces "raw material" for the computation phase. The preprocessing can be implemented using a general MPC protocol which can then be run at any time prior to the actual computation. However, [BDOZ10,DPSZ11] show particularly efficient preprocessing protocols based on public-key cryptosystems with special properties.

In this paper we are interested in computing *Boolean* circuits securely in the preprocessing model. Here, the techniques from the online phases of [BDOZ10,DPSZ11] also work, in particular Nielsen et al.[NNOB11] uses the approach from [BDOZ10] to do Boolean circuits. However, some of the efficiency is lost: for every AND-gate in the circuit, each player must do a constant number of multiplications in a large field $\mathbb{F}_{2^\kappa}$, resulting in a computational overhead that is super-linear in κ. Getting the constant overhead also for small fields was left as an open problem in [DPSZ11].

To be more precise about the cost of these protocols, we define three different types of overhead: the *data-overhead* is the total number of bits players must store from the preprocessing divided by $N \cdot |C|$, where $|C|$ is the size of the circuit to compute. The *communication-overhead* and the *computation-overhead* is the communication complexity and the computational complexity respectively (in bit operations) of the protocol divided by $N \cdot |C|$. For some protocols, these overheads turn out to be the same up to constant factors, and in such a case, we just speak of the overhead of the protocol. In a nutshell, the overhead represents the amount of resource each player needs to invest per gate in the circuit.

In this terminology, the protocol from [NNOB11] has overhead $\Omega(\kappa)$. It is nevertheless very practical and has been implemented with promising results. On the other hand, Damgård et al. [DIK10] show that based on the "MPC in the head" technique one can obtain a protocol with overhead essentially $\log(|C|) polylog(\kappa)$. This is based on preprocessing of a large number of oblivious transfers and using them to convert a multiparty protocol for honest majority into a two-party protocol. The constants involved here are very large, however, and the protocol is in fact not practical. Both these protocols are for the two-party case. The one from [DIK10] generalizes to several players but then the overhead would be $\Omega(N \log(|C|) polylog(\kappa))$.

1.1 Our Contribution

In this paper we show a multiparty computation protocol in the preprocessing model, for computing Boolean circuits securely. It is information theoretically secure against an active adversary corrupting up to $N-1$ players. We assume synchronous communication and secure point-to-point channels.

We focus on circuits that are not too "oddly shaped". Concretely, we assume that every layer of the circuit is $\Omega(\kappa)$ gates wide, where κ is the security parameter (except perhaps for a constant number of layers). Second, we want that the number of bits that are output from layer i in the circuit and used in layer j is either 0 or $\Omega(\kappa)$ for all $i < j$ (where again a constant number of exceptions are allowed). We call such circuits *well-formed*. In a nutshell, well-formed circuit are those that allow a modest amount of parallelization, namely a program computing the circuit can always execute $\Omega(\kappa)$ bit operations in parallel and when storing bits for later use or retrieving, it can always access $\Omega(\kappa)$ bits at a time.

Since our protocol has error probability $2^{-\kappa}$ and is unconditionally secure, the value of κ can be quite small, e.g., 80 and would not be affected by future advances in cryptanalysis. From a practical point of view one may therefore think of κ as being very small compared the the circuit size, and

hence the requirement that the circuit be well-formed seems rather modest. We stress that our protocols work for arbitrary circuits, but the claims we can make on the overhead will be weaker.

Throughout, we think of the circuit size as being also much larger than the number of players. Our statements on overheads below therefore ignore additive terms that are $O(\kappa N/|C|)$.

Our results differ somewhat depending on whether the preprocessing is supposed to be useful for computing any circuit, we call this *universal preprocessing* – or the preprocessing knows the circuit (but not the inputs) and only has to generate data for computing this circuit. We call this *dedicated preprocessing.*

Our protocols are based on families of codes with some specific nice properties that we explain in more detail below. The simplest construction follows from Reed-Solomon codes and from this we get:

Theorem 1. *There exists an N-party protocol for computing securely a well-formed Boolean circuit C in the dedicated preprocessing model, statistically secure against $N-1$ active corruptions. For error probability $2^{-\kappa}$, the overhead is $O(polylog(\kappa))$, where κ is the security parameter.*

There also exists an N-party protocol for computing securely a well-formed Boolean circuit C in the universal preprocessing model, statistically secure against $N-1$ active corruptions. For error probability $2^{-\kappa}$, the overhead is $O(\log(|C|) \cdot polylog(\kappa))$.

The second result applies a technique from [DIK10] to restructure C into a new circuit that has a more regular structure, but still computes the same function. The result from [DIK10] leads in general to circuits that have size $O(\log(|C|)|C| + d^2\kappa \log |C|)$, where d is the depth of C. However, in case of well-formed circuits the term depending on d disappears.

In comparison, the protocol one can construct from [DIK10] would have a larger overhead, namely $\Omega(\log(|C|) \cdot polylog(\kappa) \cdot N)$ in *both* the universal and dedicated preprocessing model[1]. In comparison to [NNOB11], we clearly do better asymptotically in the dedicated model. But also for concrete efficiency, our method can offer an improvement, particularly for the case where the circuit does a computation that is "born" parallel and hence lends itself easily to block-wise computation. See more details in section C.

Using algebraic geometry codes and results on strongly multiplicative secret sharing from [CCX11] we obtain a result that is better than Theorem 1 when the number of players is large.

Theorem 2. *There exists an N-party protocol for computing securely a well-formed Boolean circuit C in the dedicated preprocessing model, statistically secure against $N-1$ active corruptions. For an error probability of $2^{-\kappa}$, the data and communication overhead are $O(1)$ while the computation-overhead is $O(1+\frac{\kappa}{N})$, where κ is the security parameter.*

There also exists an N-party protocol for computing securely a well-formed Boolean circuit C in the universal preprocessing model, statistically secure against $N-1$ active corruptions. For an error probability of $2^{-\kappa}$, the data and communication overheads are $O(\log(|C|))$, while the computation overhead is $O(\log(|C|)(1+\frac{\kappa}{N}))$.

If we are willing to assume that the layers in C are κ^2 gates wide, then we can get computational overhead $O(1+\frac{\kappa^{\epsilon}}{N})$, Where ϵ is defined as the smallest value for which multiplication of n by n

[1] The reason why that protocol does not benefit from dedicated preprocessing is that it is based on processing bits in parallel in large blocks, and in between permuting bits inside these blocks. The efficiency, even in the online phase, crucially depends on the fact that only a logarithmic number of different permutations are needed. For this one needs to always transform C to a more regular form, leading to the $\log(|C|)$-factor.

matrices can be done in time $O(n^{2+\epsilon})$. Based on the best known matrix multiplication algorithms, we can have $\epsilon \approx 0.3727$. It may even be that any $\epsilon > 0$ suffices, but this is an open problem.

Note that none of the overheads we obtain increase with N and in fact most of them do not depend on N. In particular our protocols have constant storage overhead and constant computation overhead for a large enough number of players. They are the first protocols in the preprocessing model for Boolean circuits with this property.

Techniques. We use the idea from [DPSZ11] of having the values we compute on be secret-shared among the players, where also a Message Authentication Code (MAC) on this value is secret-shared. Using precomputed values for multiplication, linear operations then suffice for executing the computation.

However, directly usage of the MACs from [DPSZ11] or any other previous construction would not be efficient enough here, since we would have to use values from a large field ($\mathbb{F}_{2^\kappa}$) to authenticate single bits. A naive approach where one groups κ bits together and authenticate them using a single MAC over $\mathbb{F}_{2^\kappa}$ fails: we will need to do bit-wise addition and multiplication on such κ-bit vectors, and since this does not commute with multiplication in $\mathbb{F}_{2^\kappa}$, we lose the homomorphic property of the MACs that is crucial for the protocol to work.

The key to our results consists of two technical contributions: first, we develop a new authentication scheme based on families of linear codes where each code as well as its so-called Schur-transform have minimum distance and dimension a constant times their length. This scheme will have the homomorphic property we need, and will be able to authenticate κ-bit vectors using MACs and keys of size $O(\kappa)$. We note that the idea of using small MACs on entries in an error correcting code appeared in a different context in [IKOS08]. However, that application did not use any homomorphic properties of the MACs, or properties of the Schur transform.

The second technique is an efficient method for verifying membership in a linear binary code for a batch of purported codewords. We show how to do this with constant overhead per data bit. The underlying algorithm is of independent interest, as it is actually a general method for verifying multiplication of $\Theta(n)$ by $\Theta(n)$ binary matrices in time $O(n^2)$ with exponentially small error probability. The best previous methods give only constant error probability in the same time.

2 Linear Codes

In the following, we will consider a $[n, k, d]$ linear code C over a field $\mathbb{F} = \mathbb{F}_{2^u}$, i.e., C has length n, dimension k and minimum distance d. We will assume throughout that n, k, d are all $\Theta(\kappa)$, where κ is the security parameter. We will use boldface such as $\boldsymbol{x}$ to denote vectors, and when $\boldsymbol{x}, \boldsymbol{y}$ are vectors of the same length, we let $\boldsymbol{x} * \boldsymbol{y}$ denote the coordinate-wise product of $\boldsymbol{x}$ and $\boldsymbol{y}$.

For a vector $\boldsymbol{x} \in \{0, 1\}^k$, we let $C(\mathbf{x})$ be the encoding of $\boldsymbol{x}$ as a codeword in C. Without loss of generality, we assume throughout that the encoding is *systematic*, so that $\boldsymbol{x}$ itself appears as the first k entries in $C(\mathbf{x})$.

For a linear code C with parameters as above, the *Schur-transform* of C, written C^*, is a linear $[n, k^*, d^*]$-code, defined as the span of the set of vectors $\{\boldsymbol{x} * \boldsymbol{y} | \ \boldsymbol{x}, \boldsymbol{y} \in C\}$. It is easy to see that $k^* \geq k$ and $d^* \leq d$. However, we will assume that d^* is still large, namely $d^* \in \Theta(\kappa)$. This is by no means always the case, but can be obtained if C is properly constructed.

Let $\boldsymbol{x}, \boldsymbol{y}$ be k-bit strings. We define $C^*(\mathbf{x})$ to the set of codewords in C^* where $\boldsymbol{x}$ appears in the first k coordinates. This is indeed a set and not a single codeword: since k^* can be larger than

k, $\boldsymbol{x}$ does not necessarily uniquely determine a codeword in C^*. Note that since $C(\mathbf{x}) * C(\mathbf{y}) \in C^*$, and since furthermore $\boldsymbol{x} * \boldsymbol{y}$ appears in the first k coordinates of this codeword, we have

$$C(\mathbf{x}) * C(\mathbf{y}) \in C^*(\mathbf{x} * \mathbf{y})$$

This also shows that $C^*(\mathbf{x})$ is always non-empty, by taking $\boldsymbol{y} = (1, 1, ..., 1)$.

Reed-Solomon Codes As a first example, we give a simple construction showing that Reed-Solomon type codes have the right properties, if we assume that u is $\Theta(\log \kappa)$. Then we set $n = 3k$, and we may assume that $\mathbb{F}$ has at least n distinct elements $a_1, .., a_n$. Now define C to be the code consisting of vectors of form $(f(a_1), ..., f(a_n))$ where f is a polynomial over F of degree at most $k - 1$. Then C^* will be a code of the same form, but defined using polynomials of degree at most $2(k - 1)$.

It follows immediately from Lagrange interpolation that C and C^* have minimum distances as large as required. Note that we can put C in systematic form also using interpolation: given k field elements to encode, we interpolate a polynomial f such that $f(a_1), ..., f(a_k)$ equal these elements and then evaluate f in the remaining points to complete the codeword. Note also that in this case encoding and verifying membership in the codes is very efficient because it can be done by multiplication by Van der Monde matrices or their inverses. Using well-known algorithms based on the fast Fourier transform, this can be done in time $n \cdot \mathit{polylog}(n)$. In Section 4.1 we cover the complexity of our protocol when using Reed-Solomon codes.

Algebraic Geometry Codes Using the work of Cascudo et al. [CCX11], one can do even better: based on Algebraic Geometry, they construct families of codes with properties as we require over *constant size fields*. In Section 4.2 we cover the complexity of our protocol when using Algebraic Geometry codes.

3 Authentication Schemes based on Linear Codes

We will assume in this section that a code C as described above is given, of length n, dimension k and minimum distance d.

We present a new authentication scheme that we will need in the following. In its most basic version there is a receiver who knows a key $\boldsymbol{\alpha} \in \mathbb{F}^n$ chosen at random. The value to authenticate is a k-vector $\boldsymbol{x}$ and the message authentication code (MAC) is $\boldsymbol{m} = C(\mathbf{x}) * \boldsymbol{\alpha}$. Note that we use coordinate-wise multiplication by $\boldsymbol{\alpha}$, and therefore the scheme is actually doing a standard MAC over the field $\mathbb{F}$ for every coordinate of $C(\mathbf{x})$. Since $\mathbb{F}$ has constant size, this would normally not be good enough since one can forge such a MAC with constant probability $1/|\mathbb{F}|$, namely by guessing the corresponding entry in $\boldsymbol{\alpha}$. But in our case, the coding saves the day: to forge a MAC, one would need to change to a different codeword and therefore forge not 1, but d MACs.

Of course, if both $\boldsymbol{x}$ and $\boldsymbol{m}$ were known, information on $\boldsymbol{\alpha}$ would leak and other MACs could perhaps be forged. But in the protocol to follow, the MAC will be unknown to the adversary (because it will be secret shared). It therefore turns out that the following basic result on security for these MACs is what we need:

Lemma 1. *Using the above notation, suppose the adversary is given* $\boldsymbol{x}$ *and then outputs* $\boldsymbol{x'}$ *and a "MAC-error"* $\boldsymbol{\Delta}$*. We say the adversary wins if* $C(\mathbf{x'}) * \boldsymbol{\alpha} = \boldsymbol{m} + \boldsymbol{\Delta}$ *and* $\boldsymbol{x} \neq \boldsymbol{x'}$*. The probability that the adversary wins is at most* 2^{-d}.

Proof. Assuming the adversary wins, we know $C(\mathbf{x}')*\boldsymbol{\alpha} = \boldsymbol{m}+\boldsymbol{\Delta}$ holds. Plugging in $\boldsymbol{m} = C(\mathbf{x})*\boldsymbol{\alpha}$, we obtain $\boldsymbol{\alpha}*(C(\mathbf{x})-C(\mathbf{x}')) = \boldsymbol{\Delta}$. Since the adversary wins, $C(\mathbf{x})-C(\mathbf{x}')$ is a non-zero codeword, so it is non-zero in at least d coordinates. The equation therefore determines $\boldsymbol{\alpha}$ in at least d positions so we see that the adversary must guess at least d coordinates of the key to win.

Next, we consider a different way to use these MACs that turns out to be more efficient when many messages are authenticated. In this variant the scheme will use a single global key $\boldsymbol{\alpha}$ that will be secret shared so it is unknown to the adversary. The MAC on a value $\boldsymbol{x}$ is defined as before as $\boldsymbol{m} = \boldsymbol{\alpha} * C(\mathbf{x})$. In the protocol we make sure to reveal nothing about $\boldsymbol{\alpha}$ nor about any of the MACs until the end of the protocol where it will be too late for corrupted players to forge any values.

The way we prove security is thus to design the following security game modeling the way this scheme is used in the protocol.

1. The challenger generates the secret key $\boldsymbol{\alpha}$ and MACs $\boldsymbol{m}_i \leftarrow \boldsymbol{\alpha} * \boldsymbol{w}_i$ and sends the messages $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_T$ to the adversary. Note that here messages are codewords.
2. The adversary sends back messages $\boldsymbol{w}'_1, \ldots, \boldsymbol{w}'_T$.
3. The challenger generates random values $\boldsymbol{e}_1, \ldots, \boldsymbol{e}_T \leftarrow \mathbb{F}^n$ and sends them to the adversary.
4. The adversary provides an error Δ.
5. Set $\boldsymbol{w} \leftarrow \sum_{i=1}^{T} \boldsymbol{e}_i * \boldsymbol{w}'_i, \boldsymbol{m} \leftarrow \sum_{i=1}^{T} \boldsymbol{e}_i * \boldsymbol{m}_i$. Now, the challenger checks that all $\boldsymbol{w}'_i$'s are valid codewords and that $\boldsymbol{\alpha} * \boldsymbol{w} = \boldsymbol{m} + \Delta$.

The adversary wins the game if there is an i for which $\boldsymbol{w}'_i \neq \boldsymbol{w}_i$ and the final checks go through.

Generating the $\boldsymbol{e}$*'s.* We will show below that the adversary can only win this game with negligible probability if the $\boldsymbol{e}_j$'s are uniformly random. However implementing such a (trusted) random choice in our protocol turns out to be expensive, so we consider instead a way to choose them pseudorandomly using a smaller number of random bits. What we will require is a way to generate (pseudo) random strings $\mathbf{v} = (v_1, ..., v_T) \in \mathbb{F}^T$ that are *linearly* ϵ*-biased*. By this we mean that for any fixed non-zero vector $\mathbf{u}$, we have $Pr[\mathbf{u} \cdot \mathbf{v} \neq 0]x \geq \epsilon$ for some constant ϵ.

In [NN93], Naor and Naor present a construction (attributed there to Bruck) that does exactly this, based on codes with good properties, namely the same as we use here: both the dimension and the minimum distance of the code are a constant time the length of the code. Let G be the generator matrix of the code, where the rows of G form a basis of the code. Say that G has m columns where m is in $\Theta(T)$, and that the minimum distance of the code is ϵm for a constant ϵ.

Now the idea is to simply let $\mathbf{v}$ be a random column of G. To see why this works, fix any $\mathbf{u}$ and consider two random experiments: 1) compute the codeword $\mathbf{u}G$ and output a random entry from the result. 2) choose a random column $\mathbf{v}$ from G and output $\mathbf{u} \cdot \mathbf{v}$.

The first experiment clearly gives a non-zero result with probability ϵ, but on the other hand, it is equivalent to the second one since the entries in $\mathbf{u}G$ are the inner products of $\mathbf{u}$ with each column in G. We therefore get $Pr(\mathbf{u} \cdot \mathbf{v} \neq 0) \geq \epsilon$ as desired.

To connect this to the above security game, let $\boldsymbol{e}_i = (e_i^1, ..., e_i^n)$ and define $\boldsymbol{e}^j := (e_1^j, \ldots, e_T^j)$. We can now choose the $\boldsymbol{e}^j$'s to be linearly ϵ-biased instead of choosing them at random. Note that for this we need a seed consisting of $\log m \in O(\log T)$ random bits for each $\boldsymbol{e}^j$, i.e, a total of $O(n \log T)$ random bits. We then have the following:

Lemma 2. *The adversary wins the above security game with probability at most* $2^{-\Theta(n)}$*. This holds, even if the* $\boldsymbol{e}^j$ *are not random but only linearly* ϵ*-biased.*

Proof. Let us start by assuming that the $\boldsymbol{e}_i$'s are completely random and look at the adversary's probability of winning. If the checks hold then we have the following equality $\boldsymbol{\alpha} * \sum_{i=1}^{T} \boldsymbol{e}_i * \boldsymbol{v}_i = \Delta$ where $\boldsymbol{v}_i := \boldsymbol{w}'_i - \boldsymbol{w}_i$ for $i = 1, \dots, T$ are codewords and there exists at least one j for which $\boldsymbol{v}_j \neq \boldsymbol{0}$. Note that since $\boldsymbol{v}_j$ is a codeword it contains at least d entries that are 1.

Consider the sum $\sum_{i=1}^{T} \boldsymbol{e}_i * \boldsymbol{v}_i$. Let v_i^j be the j'th entry in $\boldsymbol{v}_i$, then we define $\boldsymbol{v}^j := (v_1^j, \dots, v_T^j)$. Finally we define the function $f_{\boldsymbol{v}^j}(\boldsymbol{e}^j) := \sum_{i=1}^{T} e_i^j v_i^j$ which is a linear mapping, that is not the $\boldsymbol{0}$-mapping for at least d number of j's since at least one $\boldsymbol{v}_i \neq \boldsymbol{0}$ and hence has at least d entries which are nonzero. From linear algebra we then have the rank-nullity theorem telling us that $\dim(\ker(f_{\boldsymbol{v}^j})) = T - 1$. Furthermore, since $\boldsymbol{e}^j$ is random and the adversary does not know $\boldsymbol{e}^j$ when choosing the $\boldsymbol{w}'_i$'s, the probability of $\boldsymbol{e}^j \in \ker(f_{\boldsymbol{v}^j})$ is $|\mathbb{F}^{T-1}|/|\mathbb{F}^T| = 1/|\mathbb{F}| \leq 1/2$. In the following we assume for simplicity the worst case $|\mathbb{F}| = 2$. So we expect $d/2$ of the $f_{v^j}(\boldsymbol{e}^j)$'s in question to be 1. We can use Hoeffding's inequality [Hoe63] to bound the probability that we are far from the expectation: define random variables $X_1, ..., X_d$ that take the value of the d non-trivial instances of $f_{v^j}(\boldsymbol{e}^j)$ when $\boldsymbol{e}^j$ is chosen at random. We can view the values as the result of d independent experiments where the expected value $E[X_i]$ is 1/2. Then from Hoeffding's inequality we get that $\Pr[|\frac{\sum X_i}{d} - 1/2| \geq t] \leq e^{-2t^2 d}$, for any $t > 0$. This shows that except with exponentially small probability (as a function of d) we can guarantee to deviate with at most a small constant fraction, that is, we can guarantee at least $d/2 - td = cd$ number of nonzero entries in $\sum_{i=1}^{T} \boldsymbol{e}_i * \boldsymbol{v}_i$ for any $c < 1/2$.

Assume we have this many non-zero entries. Then going back to the equality $\boldsymbol{\alpha} * \sum_{i=1}^{T} \boldsymbol{e}_i * \boldsymbol{v}_i = \Delta$, this implies that satisfying it is equivalent to guessing at least cd entries of $\boldsymbol{\alpha}$. However, since the adversary has no information about $\boldsymbol{\alpha}$, guessing can be done with probability at most 2^{-cd}. It follows that the probability the adversary wins is at most the probability that $\sum_{i=1}^{T} \boldsymbol{e}_i * \boldsymbol{v}_i$ has less than cd non-zero entries, plus 2^{-cd}. This is exponentially small in d and hence also in n since d is assumed to be $\Theta(n)$.

If the $\boldsymbol{e}_i$'s are instead chosen such that the $\boldsymbol{e}^j$'s are pseudorandom but independent and linearly ϵ-biased, then we can show the same result using a similar argument. The only difference will be that we expect to see at least ϵd non-zero entries in $\sum_{i=1}^{T} \boldsymbol{e}_i * \boldsymbol{v}_i$. By independence we can still use Hoeffding to guarantee we are close to this number of non-zero entries, and the adversary will now have to guess ϵd entries of $\boldsymbol{\alpha}$.

Note that the above security game and lemma work exactly the same way if we replace the code C by C^*, since we have assumed that the minimum distance of C^* is also $\Theta(n)$. We may even have codewords from both C and C^* in the game, as long as it is agreed in advance which words are supposed to be in C and C^* respectively. This is because the proof only depends on the fact that the non-zero vector $\boldsymbol{v}_j$ we construct has $\Theta(n)$ non-zero coordinates.

4 Protocol for Secure Computation

We are now ready to present our protocol. In structure it is much like the online protocol from [DPSZ11], but with the big difference that we are working with blocks of bits and doing parallel block-wise operations. Therefore, our protocol has an extra operation: Between two layers in the circuit we need to be able to reorganize the output bits so that they match up with the gates where they should be input. Here we assume a dedicated preprocessing phase where we know the circuit to be computed so we will know exactly how we need to move the bits around. However,

as mentioned earlier, we will also show a solution which is general, i.e. it does not depend on a preprocessing where the circuit is known.

We assume synchronous communication and secure point-to-point channels. We also assume for simplicity that broadcast is available at unit cost. This assumption can be removed without affecting the complexity using a method from [DPSZ11] which also works for our protocol since it has a similar structure. Also for simplicity we assume there is only one input from each player and one public output. It is straightforward to remove these restrictions without affecting the complexity.

Representation of values Values in our computation will be bits which are grouped in vectors of length k, i.e. we have $\boldsymbol{x} \in \mathbb{F}^k$ so that we will be doing parallel operations on blocks of k bits. Each value will be secret shared among the players along with a MAC on the value. More concretely, we have $\boldsymbol{x} = \boldsymbol{x}_1 + \cdots + \boldsymbol{x}_N$, where $\boldsymbol{x}_i \in \mathbb{F}^k$. Player i will hold the encoding of a share $C(\boldsymbol{x}_i)$, where C is a linear code as described in the previous sections. Moreover, the MAC $\boldsymbol{m(x)} = \boldsymbol{\alpha} * (C(\mathbf{x}) + \boldsymbol{d_x})$ will also be additively shared such that player i holds $\boldsymbol{m(x)}_i$. This MAC is based on C and computed using a global key $\boldsymbol{\alpha}$ and a public value $\boldsymbol{d_x}$. Summing up we have the following representation for a shared value $\boldsymbol{x}$

$$\langle \boldsymbol{x} \rangle := \big(\boldsymbol{d_x}, \big(C(\boldsymbol{x}_1), \dots, C(\boldsymbol{x}_N)\big), \big(\boldsymbol{m}(\boldsymbol{x})_1, \dots, \boldsymbol{m}(\boldsymbol{x})_N\big)\big),$$

where $\boldsymbol{d_x}$ is a public value that is necessary for easily adding public values to our representation.

It is straight forward to do linear computations with our representation. We simply do the operations component-wise and hence locally. However, we will also need to work with another kind of representation, denoted $\langle \cdot \rangle^*$, which is exactly like the $\langle \cdot \rangle$-representation except that the MACs will be based on C^* instead of C. This representation comes up when we multiply a $\langle \cdot \rangle$-representation with a public constant We can still do linear computations with $\langle \cdot \rangle^*$ since C^* is linear. On the other hand, multiplying two codewords in C^* is not guaranteed to give a result in C^*, so multiplication with $\langle \cdot \rangle^*$ is not possible. We solve this later in the protocol, where we show how to convert $\langle \cdot \rangle^*$ back into a $\langle \cdot \rangle$-representation. This will be done immediately after a multiplication, enabling us to continue with any kind of computation.

For the linear operations we write

$$\langle \boldsymbol{x} \rangle + \langle \boldsymbol{y} \rangle = \langle \boldsymbol{x} + \boldsymbol{y} \rangle \qquad \boldsymbol{e} * \langle \boldsymbol{x} \rangle = \langle \boldsymbol{e} * \boldsymbol{x} \rangle^*, \text{ and } \qquad \boldsymbol{e} + \langle \boldsymbol{x} \rangle = \langle \boldsymbol{e} + \boldsymbol{x} \rangle,$$

where $\boldsymbol{e} + \langle \boldsymbol{x} \rangle := \big(\boldsymbol{d_x} - \boldsymbol{e}, \big(C(\boldsymbol{x}_1) + \boldsymbol{e}, \dots C(\boldsymbol{x}_N)\big), \big(\boldsymbol{m}(\boldsymbol{x})_1, \dots, \boldsymbol{m}(\boldsymbol{x})_N\big)\big)$. Note that the public values $\boldsymbol{\delta}$ and $\boldsymbol{e}$ are codewords (or should be encoded when included in the representation).

For multiplication we need to use the preprocessing which will output random triples $\langle \boldsymbol{a} \rangle, \langle \boldsymbol{b} \rangle, \langle \boldsymbol{c} \rangle$, where $\boldsymbol{c} = \boldsymbol{a} * \boldsymbol{b}$. Given such a triple, we can do multiplication in the following standard way: To compute $\langle \boldsymbol{x} * \boldsymbol{y} \rangle$ we first open $\langle \boldsymbol{x} \rangle - \langle \boldsymbol{a} \rangle$ to get $\boldsymbol{\epsilon}$, and $\langle \boldsymbol{y} \rangle - \langle \boldsymbol{b} \rangle$ to get $\boldsymbol{\delta}$. Then $\boldsymbol{x} * \boldsymbol{y} = (\boldsymbol{a} + \boldsymbol{\epsilon}) * (\boldsymbol{b} + \boldsymbol{\delta}) = \boldsymbol{c} + \boldsymbol{\epsilon} * \boldsymbol{b} + \boldsymbol{\delta} * \boldsymbol{a} + \boldsymbol{\epsilon} * \boldsymbol{\delta}$. Thus, the new representation can be computed as

$$\langle \boldsymbol{x} \rangle * \langle \boldsymbol{y} \rangle = \langle \boldsymbol{c} \rangle + \boldsymbol{\epsilon} * \langle \boldsymbol{b} \rangle + \boldsymbol{\delta} * \langle \boldsymbol{a} \rangle + \boldsymbol{\epsilon} * \boldsymbol{\delta} = \langle \boldsymbol{x} * \boldsymbol{y} \rangle^*.$$

A final operation we need is reorganizing of bits between layers s.t. the output bits become input bits to the intended gates. This may involve permuting bits, duplicating bits and/or leaving out some bits. Clearly this reorganizing can be expressed as a linear function F that takes as input all the output bits of a given layer, which in our representation will be a vector of blocks of bits

$\boldsymbol{B} = (\boldsymbol{b}_1, \ldots, \boldsymbol{b}_l)$. The output of F is a new vector of blocks $F(\boldsymbol{B}) = \boldsymbol{B}' = (\boldsymbol{b}'_1, \ldots, \boldsymbol{b}'_{l'})$. For this purpose we extend our notation of $\langle\cdot\rangle$ to also include a vector of blocks instead of only one single block. We will write this as $\langle \boldsymbol{B} \rangle := \langle \boldsymbol{b}_1 \rangle, \ldots, \langle \boldsymbol{b}_l \rangle$. In general we will have that capital letters in $\langle\cdot\rangle$ denotes a vector of blocks. With this representation we still maintain the linear properties, simply by doing the operations coordinate-wise, i.e. $\langle \boldsymbol{A} \rangle + \langle \boldsymbol{B} \rangle := \langle \boldsymbol{a}_1 \rangle + \langle \boldsymbol{b}_1 \rangle, \ldots, \langle \boldsymbol{a}_l \rangle + \langle \boldsymbol{b}_l \rangle$ and so on. This means that we can compute $F(\langle \boldsymbol{B} \rangle)$ and obtain $\langle F(\boldsymbol{B}) \rangle$ for any linear function F. If we assume that we know in advance the circuit to be computed, then we also know exactly which reorganizing functions we need between the circuit layers. Thus, for each needed reorganizing function F, we will preprocess pairs of representations $\langle \boldsymbol{R} \rangle, \langle F(\boldsymbol{R}) \rangle$, where $\boldsymbol{R}$ is random and of appropriate length. As shown later, these pairs will then be used in the protocol to reorganize the actual bits.

An important note is that during our protocol we are actually not guaranteed that we are working with the correct results, since we do not immediately check the MACs of the opened values. During the first part of the protocol, parties only do what we define as a *partial opening*, meaning that each party P_i sends his share $C(\boldsymbol{a}_\mathbf{i})$ to one chosen party, say P_1. Then, P_1 computes and broadcasts $C(\boldsymbol{a})$ and the other parties verify that $C(\boldsymbol{a})$ is a valid codeword. We assume here for simplicity that we always go via P_1, whereas in practice, one would balance the workload over the players.

The checking is postponed to the end of the protocol in the output phase. To check the MACs the global key $\boldsymbol{\alpha}$ is needed. This key is provided by the preprocessing but in a slightly different representation:

$$[\![\boldsymbol{\alpha}]\!] := \big(\big(C(\boldsymbol{\alpha}_1), \ldots, C(\boldsymbol{\alpha}_N)\big), \big(\boldsymbol{\beta}_i, \boldsymbol{m}(\boldsymbol{\alpha})^i_1, \ldots, \boldsymbol{m}(\boldsymbol{\alpha})^i_N\big)_{i=1,\ldots N}\big),$$

where $\boldsymbol{\alpha} = \sum_{i=1}^N \boldsymbol{\alpha}_i$ and $\sum_{j=1}^N \boldsymbol{m}(\boldsymbol{\alpha})^j_i = C(\boldsymbol{\alpha}) * \boldsymbol{\beta}_i$. Player P_i holds $C(\boldsymbol{\alpha}_i), \boldsymbol{\beta}_i, \boldsymbol{m}(\boldsymbol{\alpha})^i_1, \ldots, \boldsymbol{m}(\boldsymbol{\alpha})^i_N$. The idea is that $\boldsymbol{m}(\boldsymbol{\alpha})_i \leftarrow \sum_{j=1}^N \boldsymbol{m}(\boldsymbol{\alpha})^j_i$ is the MAC authenticating $\boldsymbol{\alpha}$ under P_i's private key $\boldsymbol{\beta}_i$. To open $[\![\boldsymbol{\alpha}]\!]$, each P_j sends to each P_i his share $C(\boldsymbol{\alpha}_j)$ of $\boldsymbol{\alpha}$ and his share $\boldsymbol{m}(\boldsymbol{\alpha})^j_i$ of the MAC on $\boldsymbol{\alpha}$ made with P_i's private key and then P_i checks that $\sum_{j=1}^N \boldsymbol{m}(\boldsymbol{\alpha})^j_i = C(\boldsymbol{\alpha}) * \boldsymbol{\beta}_i$. (To open the value to only one party P_i, the other parties will simply send shares only to P_i, who will do the checking.)

The protocol assumes access to a commitment functionality $\mathcal{F}_{\text{COM}}$ for commitments. A player commits by calling it with a secret value $\boldsymbol{s}$ as input. The functionality stores the value until the committer calls open, in which case the value is revealed to all players. This can be implemented based only on $\mathcal{F}_{\text{PREP}}$: We open a random $[\![\boldsymbol{r}]\!]$ only to the committer, who then broadcasts $\boldsymbol{r} + \boldsymbol{s}$. When opening, we open $[\![\boldsymbol{r}]\!]$ to all players so $\boldsymbol{s}$ can be computed.

The Protocol For our protocol we assume an ideal preprocessing functionality [2] $\mathcal{F}_{\text{PREP}}$ which is shown in Figure 5. Given $\mathcal{F}_{\text{PREP}}$ and the techniques described earlier we can construct a protocol that securely implements the ideal functionality in Figure 3. The protocol is presented in Figure 1. We assume here that the circuit to be computed is structured such that there is only one type of gate per layer. This can be done without loss of generality since any function to be computed can be expressed by NAND-gates only, which then can be expressed by AND and XOR which are the operations we support.

[2] Note that we don't show a specific implementation of $\mathcal{F}_{\text{PREP}}$, since that is not the core of our result. An implementation can always be done by any general MPC protocol. However, in [DPSZ11] an efficient preprocessing protocol is shown which works on vectors of values with coordinate-wise operations just as we need in our case. Furthermore, since our online protocol in structure resembles theirs, we can use basically the same kind of preprocessing. We will elaborate on this in the full version of the paper.

Protocol Π_{MPC}

Initialize: The parties first invoke the preprocessing to get the shared secret key $[\![\boldsymbol{\alpha}]\!]$, a sufficient number of multiplication triples $(\langle \boldsymbol{a}\rangle, \langle \boldsymbol{b}\rangle, \langle \boldsymbol{c}\rangle)$, and pairs of random values of the form $\langle \boldsymbol{r}\rangle, [\![\boldsymbol{r}]\!]$ and $\langle \boldsymbol{s}\rangle, \langle \boldsymbol{s}\rangle^*$, as well as single random values $\langle \boldsymbol{t}\rangle$, $[\![\boldsymbol{t}']\!]$. Finally the preprocessing also provides pairs $\langle \boldsymbol{R}\rangle$, $\langle F(\boldsymbol{R})\rangle$ of representations where the first vector contains blocks with random bits and the second vector contains bits from the first one but reorganized according to some linear function matching the different bit reorganizing needed between layers in the circuit.

Then the steps below are performed in sequence according to the structure of the circuit to compute.

Rand: The parties take an available single $\langle \boldsymbol{t}\rangle$ and store with identifier *vid*. In the following we drop explicit mentioning of variable identifiers for the sake of brevity.

Input: To share P_i's input $\boldsymbol{x}_i$, P_i takes an available pair $\langle \boldsymbol{r}\rangle, [\![\boldsymbol{r}]\!]$ and do the following:

1. $[\![\boldsymbol{r}]\!]$ is opened to P_i.
2. P_i broadcasts $\boldsymbol{\epsilon} \leftarrow C(\mathbf{x_i}) - C(\mathbf{r})$.
3. The parties verify that $\boldsymbol{\epsilon}$ is a codeword and if so, compute $\langle \boldsymbol{x}_i\rangle \leftarrow \langle \boldsymbol{r}\rangle + \boldsymbol{\epsilon}$.

Add: To add two representations $\langle \boldsymbol{x}\rangle, \langle \boldsymbol{y}\rangle$,the parties locally compute $\langle \boldsymbol{x}+\boldsymbol{y}\rangle \leftarrow \langle \boldsymbol{x}\rangle + \langle \boldsymbol{y}\rangle$.

Multiply: To multiply $\langle \boldsymbol{x}\rangle, \langle \boldsymbol{y}\rangle$ the parties take a triple $(\langle \boldsymbol{a}\rangle, \langle \boldsymbol{b}\rangle, \langle \boldsymbol{c}\rangle)$ and a pair of random values $\langle \boldsymbol{s}\rangle, \langle \boldsymbol{s}\rangle^*$ from the set of the available ones and do the following:

1. Partially open $\langle \boldsymbol{x}\rangle - \langle \boldsymbol{a}\rangle$ to get $\boldsymbol{\epsilon}$ and $\langle \boldsymbol{y}\rangle - \langle \boldsymbol{b}\rangle$ to get $\boldsymbol{\delta}$
2. Compute $\langle \boldsymbol{x} * \boldsymbol{y}\rangle^* \leftarrow \langle \boldsymbol{c}\rangle + \boldsymbol{\epsilon} * \langle \boldsymbol{b}\rangle + \boldsymbol{\delta} * \langle \boldsymbol{a}\rangle + \boldsymbol{\epsilon} * \boldsymbol{\delta}$.
3. Partially open $\langle \boldsymbol{x} * \boldsymbol{y}\rangle^* - \langle \boldsymbol{s}\rangle^*$. As a result, P_1 learns and broadcasts a codeword $\boldsymbol{\sigma}^* \in C^*$. From this he can extract $\boldsymbol{x} * \boldsymbol{y} - \boldsymbol{s}$. He encodes this value into a codeword $\boldsymbol{\sigma} \in C$ and broadcasts also $\boldsymbol{\sigma}$.
4. All players check that $\boldsymbol{\sigma}^*, \boldsymbol{\sigma}$ are codewords and that the same value occurs in the first k coordinates. Then compute $\langle \boldsymbol{x} * \boldsymbol{y}\rangle \leftarrow \boldsymbol{\sigma} + \langle \boldsymbol{s}\rangle$.

Reorganize Let $\boldsymbol{B} = (\boldsymbol{b}_1, \ldots, \boldsymbol{b}_l)$ be the vector of blocks containing the output bits of a given layer in the circuit. To reorganize these bits as inputs for the next layer the parties first identify which function F matches this particular reorganizing and take preprocessed representations $\langle \boldsymbol{R}\rangle$, $\langle F(\boldsymbol{R})\rangle$ for a random vector of blocks $\boldsymbol{R}$ of same length as $\boldsymbol{B}$. The parties then do the following steps.

1. Partially open $\langle \boldsymbol{B}\rangle - \langle \boldsymbol{R}\rangle$. This involves broadcasting a set of codewords, $C(\boldsymbol{B}-\boldsymbol{R})$.
2. P_1 extracts $\boldsymbol{B}-\boldsymbol{R}$, computes $F(\boldsymbol{B}-\boldsymbol{R})$ and encodes this into a set of codewords, $C(F(\boldsymbol{B}-\boldsymbol{R}))$ that he broadcasts.
3. All players verify that $C(\boldsymbol{B}-\boldsymbol{R})$, $C(F(\boldsymbol{B}-\boldsymbol{R}))$ are sets of valid codewords and that the encoded bits are related via F. Then compute $\langle F(\boldsymbol{R})\rangle \leftarrow C(F(\boldsymbol{B}-\boldsymbol{R})) + \langle F(\boldsymbol{R})\rangle$.

Output: We enter this stage when the players have $\langle \boldsymbol{y}\rangle$ for the output value $\boldsymbol{y}$, but this value has been not been opened (the output value is only correct if players have behaved honestly). We then do the following:

1. Let $C(\boldsymbol{a}_1), \ldots, C(\boldsymbol{a}_{T'})$ be all the values encoded in C that have been publicly opened so far, where $\langle \boldsymbol{a}_j\rangle = (\boldsymbol{\delta}_j, (C(\boldsymbol{a})_{j,1}, \ldots, C(\boldsymbol{a})_{j,n}), (\boldsymbol{m}(\boldsymbol{a}_j)_1, \ldots, \boldsymbol{m}(\boldsymbol{a}_j)_n))$. Similarly, let $C^*(\boldsymbol{a})_{T'+1}, \ldots, C^*(\boldsymbol{a})_T$ be the opened values encoded in C^*. The parties open $\lceil c \cdot n \log(T)/k \rceil$ single random values $[\![\boldsymbol{t}]\!]$ (where c is a constant) and use the bits as seed to generate $\boldsymbol{e}_1, \ldots, \boldsymbol{e}_T$ as described in Section 3. All players now compute $\boldsymbol{a} \leftarrow \sum_{j=1}^{T} \boldsymbol{e}_j * C(\boldsymbol{a})_j + \sum_{j=T'+1}^{T} \boldsymbol{e}_j * C^*(\boldsymbol{a})_j$.
2. Each P_i calls $\mathcal{F}_{\text{Com}}$ to commit to $\boldsymbol{m}_i \leftarrow \sum_{j=1}^{T} \boldsymbol{e}_j * \boldsymbol{m}(\boldsymbol{a}_j)_i$. For the output value $\langle \boldsymbol{y}\rangle$, P_i also commits to his share $\boldsymbol{y}_i$, and his share $\boldsymbol{m}(\boldsymbol{y})_i$ in the corresponding MAC.
3. $[\![\boldsymbol{\alpha}]\!]$ is opened.
4. Each P_i asks $\mathcal{F}_{\text{Com}}$ to open $\boldsymbol{m}_i$, and all players check that $\boldsymbol{\alpha} * (\boldsymbol{a} + \sum_{j=1}^{T} \boldsymbol{e}_j * \boldsymbol{\delta}_j) = \sum_{i=1}^{N} \boldsymbol{m}_i$. If a check fails, the protocol aborts. Otherwise the players conclude that the output value is correctly computed.
5. To get the output value $\boldsymbol{y}$, the commitments to $\boldsymbol{y}_i, \boldsymbol{m}(\boldsymbol{y})_i$ are opened. Now, $\boldsymbol{y}$ is defined as $\boldsymbol{y} := \sum_{i=1}^{N} \boldsymbol{y}_i$ and each player checks that $\boldsymbol{\alpha} * (\boldsymbol{y} + \boldsymbol{\delta}) = \sum_{i=1}^{N} \boldsymbol{m}(\boldsymbol{y})_i$, if so, $\boldsymbol{y}$ is the output.

Fig. 1. The online protocol.

We can now state the theorem on security of the online protocol.

Theorem 3. *In the $\mathcal{F}_{\text{PREP}}, \mathcal{F}_{\text{COM}}$-hybrid model, the protocol Π_{MPC} implements $\mathcal{F}_{\text{MPC}}$ with statistical security against any static active adversary corrupting up to $N-1$ parties.*

Proof (Theorem 3).

We construct a simulator $\mathcal{S}_{\mathrm{MPC}}$ such that a poly-time environment $\mathcal{Z}$ cannot distinguish between the real protocol system $\mathcal{F}_{\mathrm{PREP}}, \mathcal{F}_{\mathrm{COM}}$ composed with Π_{MPC} and $\mathcal{F}_{\mathrm{MPC}}$ composed with $\mathcal{S}_{\mathrm{MPC}}$. We assume here static, active corruption. The simulator will internally run a copy of $\mathcal{F}_{\mathrm{PREP}}$ composed with Π_{MPC} where it corrupts the parties specified by $\mathcal{Z}$. The simulator relays messages between parties/$\mathcal{F}_{\mathrm{PREP}}$ and $\mathcal{Z}$, such that $\mathcal{Z}$ will see the same interface as when interacting with a real protocol. The specification of the simulator $\mathcal{S}_{\mathrm{MPC}}$ is presented in Figure 2.

To see that the simulated and real processes cannot be distinguished, we will show that the view of the environment in the ideal process is statistically indistinguishable from the view in the real process. This view consists of the corrupt players' view of the protocol execution as well as the inputs and outputs of honest players.

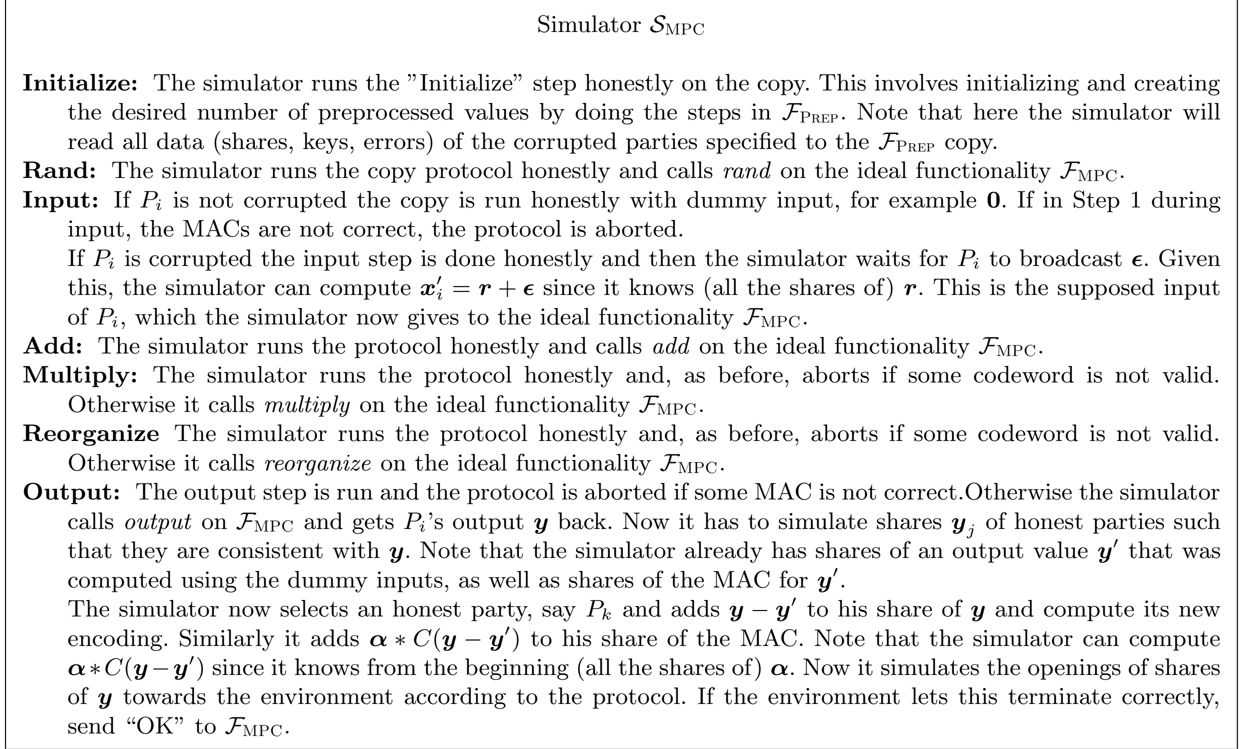
We first argue that the view up to the point where the output value is opened (step 5 of the 'output' stage of the protocol) has exactly the same distribution in the real and in the simulated case: First, the value broadcast by honest players in the input stage are always uniformly random. Second, when a value is partially opened in a secure multiplication or a reorganize step, fresh shares of a random value are subtracted, so the honest players will always send a set of uniformly random and independent values. Third, the honest players hold shares in MACs on the opened values, these are random sharings of a correct MAC. Therefore, also the MAC and shares revealed in step 4 of 'output' have the same distribution in the simulated as in the the real process. Finally note that if the simulated protocol aborts, the simulator makes the ideal functionality fail, so the environment will see that honest players generate no output, just as when the real process aborts.

Now, if the real or simulated protocol proceeds to the last step, the only new data that the environment sees is an output value $\boldsymbol{y}$, plus some shares of honest players. These are random shares that are consistent with $\boldsymbol{y}$ and its MAC in both the simulated and real case. In other words, the environments' view of the last step has the same distribution in real and simulated case as long as $\boldsymbol{y}$ is the same.

In the simulation, $\boldsymbol{y}$ is of course the correct evaluation on the inputs matching the shares that were read from the corrupted parties in the beginning. To finish the proof, it is therefore sufficient to show that the same happens in the real process with overwhelming probability. In other words, we show that the event that the real protocol terminates but the output is not correct occurs with negligible probability. Incorrect outputs result if corrupted parties during the protocol successfully cheat with their shares. We have two kinds of checks on shares corresponding to the two kinds of representations $[\![\cdot]\!]$ and $\langle\cdot\rangle$. The checks related to the openings of $[\![\cdot]\!]$-values are done during 'Input and in steps 1 and 3 of 'Output'. We get from Lemma 1 that the probability of cheating in each of these openings is at most 2^{-d}.

For the check in step 5 (which is for all the opened $\langle\cdot\rangle$ and $\langle\cdot\rangle^*$ values) we turn to the security game of Lemma 2 in Section 3. It is not difficult to see this game indeed models 'Output'(up to step 5): The second step in the game where the adversary sends the $\boldsymbol{w}_i'$'s models the fact that corrupted players can choose to lie about their shares of values opened during the protocol execution. $\boldsymbol{\Delta}$ models the fact that the adversary may modify the shares of MACs held by corrupt players. Finally, since $\boldsymbol{\alpha}$ and $\boldsymbol{m}_i$ are secret shared in the protocol, the adversary has no information on $\boldsymbol{\alpha}$ and $\boldsymbol{m}$ ahead of time in the protocol, just as in the security game. Therefore, we get from Lemma 2 that the probability of a party being able to cheat in step 5 is at most $2^{-\Theta(n)}$. Finally, for the check in step 6, only one MAC is checked for each output, so here the probability of cheating is 2^{-d}, again by Lemma 1.

Since the protocol aborts as soon as a check fails, the probability that it terminates with an incorrect output is the maximum probability with which any single check can be cheated. Since n and d are assumed to be $\Theta(\kappa)$, all these probabilities are $2^{-\Theta(\kappa)}$, and hence the maximum is also exponentially small. □



Simulator $\mathcal{S}_{\mathrm{MPC}}$

Initialize: The simulator runs the "Initialize" step honestly on the copy. This involves initializing and creating the desired number of preprocessed values by doing the steps in $\mathcal{F}_{\mathrm{PREP}}$. Note that here the simulator will read all data (shares, keys, errors) of the corrupted parties specified to the $\mathcal{F}_{\mathrm{PREP}}$ copy.

Rand: The simulator runs the copy protocol honestly and calls *rand* on the ideal functionality $\mathcal{F}_{\mathrm{MPC}}$.

Input: If P_i is not corrupted the copy is run honestly with dummy input, for example $\mathbf{0}$. If in Step 1 during input, the MACs are not correct, the protocol is aborted.
If P_i is corrupted the input step is done honestly and then the simulator waits for P_i to broadcast $\boldsymbol{\epsilon}$. Given this, the simulator can compute $\boldsymbol{x}'_i = \boldsymbol{r} + \boldsymbol{\epsilon}$ since it knows (all the shares of) $\boldsymbol{r}$. This is the supposed input of P_i, which the simulator now gives to the ideal functionality $\mathcal{F}_{\mathrm{MPC}}$.

Add: The simulator runs the protocol honestly and calls *add* on the ideal functionality $\mathcal{F}_{\mathrm{MPC}}$.

Multiply: The simulator runs the protocol honestly and, as before, aborts if some codeword is not valid. Otherwise it calls *multiply* on the ideal functionality $\mathcal{F}_{\mathrm{MPC}}$.

Reorganize The simulator runs the protocol honestly and, as before, aborts if some codeword is not valid. Otherwise it calls *reorganize* on the ideal functionality $\mathcal{F}_{\mathrm{MPC}}$.

Output: The output step is run and the protocol is aborted if some MAC is not correct.Otherwise the simulator calls *output* on $\mathcal{F}_{\mathrm{MPC}}$ and gets P_i's output $\boldsymbol{y}$ back. Now it has to simulate shares $\boldsymbol{y}_j$ of honest parties such that they are consistent with $\boldsymbol{y}$. Note that the simulator already has shares of an output value $\boldsymbol{y}'$ that was computed using the dummy inputs, as well as shares of the MAC for $\boldsymbol{y}'$.
The simulator now selects an honest party, say P_k and adds $\boldsymbol{y} - \boldsymbol{y}'$ to his share of $\boldsymbol{y}$ and compute its new encoding. Similarly it adds $\boldsymbol{\alpha} * C(\boldsymbol{y} - \boldsymbol{y}')$ to his share of the MAC. Note that the simulator can compute $\boldsymbol{\alpha} * C(\boldsymbol{y} - \boldsymbol{y}')$ since it knows from the beginning (all the shares of) $\boldsymbol{\alpha}$. Now it simulates the openings of shares of $\boldsymbol{y}$ towards the environment according to the protocol. If the environment lets this terminate correctly, send "OK" to $\mathcal{F}_{\mathrm{MPC}}$.

Fig. 2. The simulator for $\mathcal{F}_{\mathrm{MPC}}$.

Having shown the construction of the protocol and proved security, the only thing left is to argue about the complexities depending on the concrete linear codes and preprocessing model. In the following two sections we argue the complexities using Reed-Solomon and Algebraic Geometry codes respectively with both dedicated and universal preprocessing, and thereby completing the proofs of Theorem 1 and Theorem 2.

4.1 Using the Protocol with Reed-Solomon Codes

Dedicated Preprocessing First, we note that by our assumptions on the code C, a codeword contains $k \in \Theta(n)$ data bits. Moreover, in the $\langle\cdot\rangle$-representation, each player stores only two n-bit vectors as his share of each encoded block, namely a codeword of the additive share of the value itself and a share of the MAC. It follows that each player stores $O(n)$ field elements per $\langle\cdot\rangle$-representation.

We then define a block operation to be an addition, a multiplication, or an opening of $\langle\cdot\rangle$-representations. Since the circuit is well formed and each $\langle\cdot\rangle$-representation "contains" $\Theta(n)$ data bits, it is not hard to see that the number of block operations we need to compute a circuit of size S is $O(S/n)$.

Now, the storage and communication overheads follow because we use at most a constant number of $\langle\cdot\rangle$- representations from the preprocessing for each block operation, and the communication needed is a most Nn field elements for each such operation. So $O(S/n \cdot Nn) = O(SN)$ field elements need to be stored from the preprocessing and this is also the communication complexity. The field has to have at least n elements for the Reed-Solomon construction to work, hence each field element has size $O(\log n) = O(\log \kappa)$ bits. Putting all this together, we see that the storage and communication overheads are both $O(SN \log \kappa/(SN)) = O(\log \kappa)$. It should be noted that we also use a number of the more expensive $[\![\cdot]\!]$-representations, these cost $O(N^2 n)$ field elements in storage and communication when they are opened. However, we only need $\lceil c \cdot n \log(T)/k \rceil$ of these, which is $O(\log(T))$. Since T is linear in the circuit size S, the storage and communication overhead for this part will be $O(N^2 n \log(\kappa) \cdot \log(S)/(SN)) = O(N\kappa \log(\kappa) \log(S)/S)$. As explained in the introduction, we assume S is much larger than $N\kappa$, so this term can be ignored when we compute the overhead.

As for computation, the most expensive operation done on a block is the re-encoding and membership verification we need for every layer and every multiplication. This costs $O(n \cdot polylog(n))$ bit operations per block, because we are working with Van der Monde matrices, as explained in Section 2. This means the total computational complexity is $O(S/n \cdot N \cdot n \cdot polylog(n))$, so the overhead is $O(polylog(n)) = O(polylog(\kappa))$.

Universal Preprocessing Here we use the restructuring of the circuit as described [DIK10]. This makes the circuit somewhat larger, namely by a factor of $O(log(S))$ as mentioned in Section 1.1. Now the reorganization of bits between layers can be done simply by permuting inside one block at a time. Moreover, the permutations we need are the same, independently of the circuit we want to compute. Hence a number of random pairs $\langle \boldsymbol{r} \rangle, \langle \pi(\boldsymbol{r}) \rangle$ can be prepared in advance, where π ranges over the permutations needed. This implies the second part of Theorem 1 if we again use Reed-Solomon codes. The only change compared to dedicated preprocessing is that overheads have to be multiplied by the factor by which the circuit gets larger when we apply the restructuring from [DIK10].

4.2 Using the Protocol with Algebraic Geometry Codes

Before going into depth with the argument for complexity using Algebraic Geometry codes, we look at the problem of batch verification of membership in binary codes.

Verifying Membership in (Binary) Codes with Amortized Efficiency. We will need a solution to the following problem: Suppose we are given a set of vectors of length n and it is claimed that they are all in the linear binary code C, of dimension k and length n. Say there are $\Theta(n)$ input vectors. We want to verify that every vector is indeed in C, possibly with an error probability that is negligible in n, and without making any assumptions on C.

Let H be the parity check matrix for C, so H has n columns and $n-k$ rows. We then put the input vectors as columns in a matrix M, where we assume for simplicity that there are $n-k$ input words so M has $n-k$ columns. Now, an obvious method is to just compute $U = HM$ and check this result is the all-0 matrix. Using good matrix multiplication algorithms, this does save time over the naive approach of just multiplying H on every input vector, namely we go from cubic time to $O(n^{2+v})$ where $v > 0$ depends on the matrix algorithm used. However, we want to do better.

Let G be a generator matrix for a linear time encodable code, of dimension $n-k$ and length m. From the results of Spielman [Spi96], it follows that families of such codes exist, that also have constant information and error rate. We can therefore assume that m is in $\Theta(n)$ and the minimum distance of the code generated by G is also in $\Theta(n)$. Using the standard convention that the rows of G form a basis of the code, We assume that G has m columns and $n-k$ rows. Let $G^\dagger$ be the transposed of G.

By linear time encodability, we can multiply a row vector with G in linear time (or multiply $G^\dagger$ by a column), and hence compute $G^\dagger H$ and MG in time $O(n^2)$. This leads to the following algorithm, which is actually a general method for checking whether the product of two matrices is 0:

CheckZeroProduct

1. On input H, M, compute $G^\dagger H$ and MG.
2. Select at random n pairs of indices $(i_\ell, j_\ell) \in \{1, ..., m\}^2$ for $\ell = 1...n$.
3. For $\ell = 1..n$, compute the inner product of row i_ℓ of $G^\dagger H$ and column j_ℓ of MG.
4. If all inner products are 0, output "accept", else "reject".

Theorem 4. *The algorithm* CheckZeroProduct *runs in time $O(n^2)$. If $HM = 0$ it always accepts, and if not, it accepts with probability in $2^{-\Theta(n)}$.*

Proof. Recall that $U = HM$ and note that

$$(G^\dagger H)(MG) = G^\dagger UG = (G^\dagger U)G$$

Now, if $U = 0$, then $GUG^\dagger = 0$ and the algorithm accepts. Otherwise, at least one entry in U is not 0. We can think of the expression $(G^\dagger U)G$ as first encoding each column of U using $G^\dagger$ and then encoding each row of the result using G. Since the code generated by G has minimum weight/distance in $\Theta(n)$, it follows that a constant fraction of the entries in $(G^\dagger U)G$ are non-zero. The algorithm effectively probes n random entries in $(G^\dagger U)G$ and will therefore accept in this case with probability at most $2^{-\Theta(n)}$. We already argued that we can compute $G^\dagger H$ and MG in time $O(n^2)$ and the inner products clearly also take time $O(n^2)$.

Now we return to our protocol and derive the overheads we get if we use algebraic geometry codes and exploit the fast verification of codewords. This will establish the results claimed in Theorem 2.

For the storage and communication overhead, exactly the same arguments as for Reed-Solomon codes apply, with the only difference that the field size is now constant, so this immediately gives us that the storage and communication overheads are constant when we do dedicated preprocessing. For universal preprocessing we have to multiply by the "expansion factor" from [DIK10].

As for the computation overhead, again the re-encoding done for multiplication and reordering of bits is the bottleneck, in fact the overhead from other computation is constant. Note that in the protocol only a single player encodes data, while the other players only verify membership in the codes, and the overhead from verification can be made constant using the above algorithm. We therefore just need to compute the overhead coming from a single player doing $O(S/n)$ encodings, where S is the size of the circuit computed. Doing encoding by simply multiplying by the generator matrix costs $O(n^2)$ operations, so we get an overhead of $O(nS/(NS)) = O(\kappa/N)$.

If the circuit is wide enough that encoding can always be done in batches of size $\Omega(n)$, it can be done by matrix multiplication in time $O(n^{2+\epsilon})$ for a batch, or $O(n^{1+\epsilon})$ per encoded word. This gives computation overhead $O(\kappa^\epsilon/N)$ by the same argument as for Reed-Solomon.

References

[BDOZ10] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorhic enryption and multiparty computation (full version). In *The Eprint Archive, report 2010/514*, 2010.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145, 2001.

[CCX11] Ignacio Cascudo, Ronald Cramer, and Chaoping Xing. The torsion-limit for algebraic function fields and its application to arithmetic secret sharing. In Phillip Rogaway, editor, *CRYPTO*, volume 6841 of *Lecture Notes in Computer Science*, pages 685–705. Springer, 2011.

[CLOS02] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *STOC*, pages 494–503, 2002.

[DIK10] Ivan Damgård, Yuval Ishai, and Mikkel Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. In Henri Gilbert, editor, *EUROCRYPT*, volume 6110 of *Lecture Notes in Computer Science*, pages 445–465. Springer, 2010.

[DPSZ11] I. Damgard, V. Pastro, N.P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. Cryptology ePrint Archive, Report 2011/535, 2011. http://eprint.iacr.org/.

[GHS12] Craig Gentry, Shai Halevi, and Nigel P. Smart. Fully homomorphic encryption with polylog overhead. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT*, volume 7237 of *Lecture Notes in Computer Science*, pages 465–482. Springer, 2012.

[Hoe63] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.

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

[NN93] Joseph Naor and Moni Naor. Small-bias probability spaces: Efficient constructions and applications. *SIAM J. Comput.*, 22(4):838–856, 1993.

[NNOB11] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. Cryptology ePrint Archive, Report 2011/091, 2011. http://eprint.iacr.org/.

[Spi96] Daniel A. Spielman. Linear-time encodable and decodable error-correcting codes. *IEEE Transactions on Information Theory*, 42(6):1723–1731, 1996.

A Acknowledgment

We thank Yuval Ishai for inspiring discussions.

B Functionalities

Functionality $\mathcal{F}_{\text{MPC}}$

Initialize: On input $(init, k)$ from all parties, the functionality activates and stores the block length k.

Rand: On input $(rand, P_i, vid)$ from all parties P_i, with vid a fresh identifier, the functionality picks $\boldsymbol{r} \leftarrow \mathbb{F}_2^k$ and stores $(vid, \boldsymbol{r})$.

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

Add: On command $(add, vid_1, vid_2, vid_3)$ from all parties (if vid_1, vid_2 are present in memory and vid_3 is not), the functionality retrieves $(vid_1, \boldsymbol{x})$, $(vid_2, \boldsymbol{y})$ and stores $(vid_3, \boldsymbol{x} + \boldsymbol{y})$.

Multiply: On input $(multiply, vid_1, vid_2, vid_3)$ from all parties (if vid_1, vid_2 are present in memory and vid_3 is not), the functionality retrieves $(vid_1, \boldsymbol{x})$, $(vid_2, \boldsymbol{y}$ and stores $(vid_3, \boldsymbol{x} * \boldsymbol{y})$.

Reorganize: On input $(reorganize, F, vid_1, \ldots, vid_l, vid'_1, \ldots, vid'_{l'})$ from all parties (if $vid_1, \ldots vid_l$ are present in memory and $vid'_1, \ldots, vid'_{l'}$ are not), the functionality retrieves $(vid_1, \boldsymbol{x}_1), \ldots, (vid_l, \boldsymbol{x}_l)$. Then it applies the function F (if the number of parameters match) to get $F(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_l) = (\boldsymbol{x}'_1, \ldots, \boldsymbol{x}'_{l'})$ and stores $(vid'_1, \boldsymbol{x}'_1), \ldots, (vid''_l, \boldsymbol{x}'_{l'})$.

Output: On input $(output, vid)$ from all honest parties (if vid is present in memory), the functionality retrieves $(vid, \boldsymbol{x})$ and outputs it to the environment. If the environment returns "OK", then output $(vid, \boldsymbol{x})$ to all players, else output $\perp$ to all players.

Fig. 3. The ideal functionality for MPC.

$\mathcal{F}_{\text{Prep}}$ Macros

Usage: We describe two macros, one to produce $[\![\boldsymbol{v}]\!]$ representations and one to produce $\langle \boldsymbol{v} \rangle$ representations. We denote by A the set of players controlled by the adversary.

$\mathsf{Bracket}(\boldsymbol{v}_1, \ldots, \boldsymbol{v}_N, \boldsymbol{\beta}_1, \ldots, \boldsymbol{\beta}_N)$, where $\boldsymbol{v}_i, \boldsymbol{\beta}_i \in \mathbb{F}^n$ and the $\boldsymbol{v}_i$'s are codewords in C.

1. Let $\boldsymbol{v} = \sum_{i=1}^N \boldsymbol{v}_i$.
2. For $i = 1, \ldots, N$
   (a) The functionality computes the MAC $\boldsymbol{m}(\boldsymbol{v})_i \leftarrow \boldsymbol{v} * \boldsymbol{\beta}_i$.
   (b) For every corrupt player P_j, $j \in A$ the environment specifies a share $\boldsymbol{m}(\boldsymbol{v})_i^j$.
   (c) The functionality sets each share $\boldsymbol{m}(\boldsymbol{v})_i^j$, $j \notin A$, uniformly such that $\sum_{j=1}^N \boldsymbol{m}(\boldsymbol{v})_i^j = \boldsymbol{m}(\boldsymbol{v})_i$.
3. It sends $(\boldsymbol{v}_i, (\boldsymbol{\beta}_i, \boldsymbol{m}(\boldsymbol{v})_1^i, \ldots, \boldsymbol{m}(\boldsymbol{v})_N^i))$ to each honest player P_i. (Dishonest players already have the data).

$\mathsf{Angle}(\boldsymbol{v}_1, \ldots, \boldsymbol{v}_N, \boldsymbol{\alpha})$, where $\boldsymbol{v}_i, \boldsymbol{\alpha} \in \mathbb{F}^n$ and the $\boldsymbol{v}_i$'s are codewords, all either in C or C^*.

1. Let $\boldsymbol{v} = \sum_{i=1}^N \boldsymbol{v}_i$.
2. The functionality computes the MAC $\boldsymbol{m}(\boldsymbol{v}) \leftarrow \boldsymbol{v} * \boldsymbol{\alpha}$.
3. For every corrupt player P_i, $i \in A$ the environment specifies a share $\boldsymbol{m}(\boldsymbol{v})_i$.
4. The functionality sets each share $\boldsymbol{m}(\boldsymbol{v})_i$ $i \notin A$ uniformly such that $\sum_{i=1}^N \boldsymbol{m}(\boldsymbol{v})_i = \boldsymbol{m}(\boldsymbol{v})$.
5. It sends $(0, \boldsymbol{v}_i, \boldsymbol{m}(\boldsymbol{v})_i)$ to each honest player P_i. (Dishonest players already have the data).

Fig. 4. Macros for use in $\mathcal{F}_{\text{Prep}}$.

Functionality $\mathcal{F}_{\text{PREP}}$

Usage: The functionality uses two macros described in Figure 4, to produce $[\![\boldsymbol{v}]\!]$ and $\langle \boldsymbol{v} \rangle$ representations. We denote by A the set of players controlled by the adversary.

Initialize: On input $(init, n, k, d, u, G)$ from all players, the functionality stores the integers n, k, d, u and the generator matrix G for a linear $[n, k, d]$-code C over the field $\mathbb{F} = \mathbb{F}_{2^u}$. It then waits for the environment to call either "stop" or "OK". In the first case the functionality sends "fail" to all honest players and stops. In the second case it does the following:

1. For each corrupt player P_i, $i \in A$, the environment specifies a codeword $C(\boldsymbol{\alpha}_i)$ of a share $\boldsymbol{\alpha}_i$ Note that $\boldsymbol{\alpha}_i$ is easy to extract from $C(\boldsymbol{\alpha}_i)$ since it is the first k entries of $C(\boldsymbol{\alpha}_i)$.
2. The functionality sets each share $\boldsymbol{\alpha}_i$, $i \notin A$ uniformly.
3. For each corrupt player P_i, $i \in A$, the environment specifies a key $\boldsymbol{\beta}_i$.
4. The functionality sets each key $\boldsymbol{\beta}_i$ $i \notin A$ uniformly.
5. It runs the macro $\mathsf{Bracket}(C(\boldsymbol{\alpha}_1), \ldots C(\boldsymbol{\alpha}_n), \boldsymbol{\beta}_1, \ldots, \boldsymbol{\beta}_n)$.

Pairs$(\langle \boldsymbol{r} \rangle, [\![\boldsymbol{r}]\!])$: On input $(pair)$ from all players, the functionality waits for the environment to call either "stop" or "OK". In the first case the functionality sends "fail" to all honest players and stops. In the second case it does the following:

1. For each corrupt player P_i, $i \in A$, the environment specifies a codeword $C(\boldsymbol{r}_i)$ of the share $\boldsymbol{r}_i$.
2. The functionality sets each share $\boldsymbol{r}_i$, $i \notin A$ uniformly.
3. It runs the macros $\mathsf{Bracket}(C(\boldsymbol{r}_1), \ldots, C(\boldsymbol{r}_n), \boldsymbol{\beta}_1, \ldots, \boldsymbol{\beta}_n)$ and $\mathsf{Angle}(C(\boldsymbol{r}_1), \ldots, C(\boldsymbol{r}_n), \boldsymbol{\alpha})$.

Pairs$(\langle \boldsymbol{r} \rangle, \langle \boldsymbol{r} \rangle^*)$: On input $(pair, *)$ from all players, the functionality waits for the environment to call either "stop" or "OK". In the first case the functionality sends "fail" to all honest players and stops. In the second case it does the following:

1. For each corrupt player P_i, $i \in A$, the environment specifies codewords $C(\boldsymbol{r}_i), C^*(\boldsymbol{r}_i)$ of the share $\boldsymbol{r}_i$.
2. The functionality sets each share $\boldsymbol{r}_i$, $i \notin A$ uniformly.
3. It runs the macros $\mathsf{Angle}(C(\boldsymbol{r}_1), \ldots, C(\boldsymbol{r}_n), \boldsymbol{\alpha})$ and $\mathsf{Angle}(\boldsymbol{C}^*(r_1), \ldots, C(\boldsymbol{r}_n), \boldsymbol{\alpha})$.

Pairs$(\langle \boldsymbol{R} \rangle, \langle F(\boldsymbol{R}) \rangle))$: On input $(pair, F(\boldsymbol{X})$ from all players, the functionality waits for the environment to call either "stop" or "OK". In the first case the functionality sends "fail" to all honest players and stops. In the second case it does the following:

1. For each corrupt player P_i, $i \in A$, the environment specifies a vector of codewords $(C(\boldsymbol{r}_{i1}), \ldots, C(\boldsymbol{r}_{il}))$ representing the vector $\boldsymbol{R}_i = (\boldsymbol{r}_{i1}, \ldots, \boldsymbol{r}_{il})$ of length l according to F.
2. The functionality sets each $\boldsymbol{R}_i$, $i \notin A$ uniformly. (Now $\boldsymbol{R} := \sum_i \boldsymbol{R}_i = (\boldsymbol{r}_1, \ldots, \boldsymbol{r}_l)$, where $\boldsymbol{r}_j := \sum_i \boldsymbol{r}_{ij}$.)
3. To get the shared representation of each entry in $\boldsymbol{R}$ and $F(\boldsymbol{R})$, it runs for $j = 1, \ldots, l$ the macros $\mathsf{Angle}(C(\boldsymbol{r}_{1j}), \ldots, C(\boldsymbol{r}_{nj}), \boldsymbol{\alpha})$ and $\mathsf{Angle}(F(\boldsymbol{r}_{1j)}, \ldots, F(\boldsymbol{r}_{nj}), \boldsymbol{\alpha})$, where we abuse the notation slightly to let $F(\boldsymbol{r}_{ij})$ denote the jth entry of the vector $F(\boldsymbol{R}_i)$.

Triples: On input $(triple)$ from all players, the functionality waits for the environment to call either "stop" or "OK". In the first case the functionality sends "fail" to all honest players and stops. In the second case it does the following

1. For each corrupt player P_i, $i \in A$, the environment specifies codewords $C(\boldsymbol{a}_i), C(\boldsymbol{b}_i)$ of shares $\boldsymbol{a}_i, \boldsymbol{b}_i$.
2. The functionality sets each share $\boldsymbol{a}_i, \boldsymbol{b}_i$, $i \notin A$ uniformly. Let $\boldsymbol{a} := \sum_{i=1}^n \boldsymbol{a}_i$, $\boldsymbol{b} := \sum_{i=1}^n \boldsymbol{b}_i$.
3. It sets $\boldsymbol{c} \leftarrow \boldsymbol{a} * \boldsymbol{b}$.
4. For each corrupt player P_i, $i \in A$, the environment specifies a codeword $C(\boldsymbol{c}_i)$ of the share $\boldsymbol{c}_i$.
5. The functionality sets each share $\boldsymbol{c}_i$, $i \notin A$ uniformly with the constraint $\sum_{i=1}^n \boldsymbol{c}_i = \boldsymbol{c}$.
6. It runs $\mathsf{Angle}(C(\boldsymbol{a}_1), \ldots, C(\boldsymbol{a}_n), \boldsymbol{\alpha})$, $\mathsf{Angle}(C(\boldsymbol{b}_1), \ldots, C(\boldsymbol{b}_n), \boldsymbol{\alpha})$, and $\mathsf{Angle}(C(\boldsymbol{c}_1), \ldots, C(\boldsymbol{c}_n), \boldsymbol{\alpha})$.

Fig. 5. The ideal functionality for preprocessing.

C A Practical Scenario with Concrete Parameters

To demonstrate how our protocol can be used in practice, we analyze a concrete example. Assume we use the Reed-Solomon-based construction of the codes we need, using the field with 256 elements. This means that field elements conveniently fit in a byte and we can have codewords of length n up to $n = 256$. Every MAC on a field element can be cheated with probability $1/256 = 2^{-8}$. Say we want 128 bit security. This means that the Schur transform of the code must have minimum distance at least 16, since $(2^{-8})^{16} = 2^{-128}$. When encoding k field elements, we construct a polynomial of degree $k-1$, and this degree doubles when we do the Schur transform, so we need that $256-2(k-1) \geq 16$, or that $k \leq 121$.

We see therefore that we expand a block of 121 bits to 256 bytes, i.e., by a factor of about 17. This can be compared to the approach of [NNOB11] where the factor is 128, that is, we save a factor of 8 on the data and communication overhead. Moreover, in [NNOB11], the error of 2^{-128} can be obtained by computing 16 MACs over $\mathbb{F}_{256}$ for every bit, whereas we need 256 MACs for 121 bits, so for computation we also save a factor of about 8. All this assumes, of course that the computation is such that most of the work can actually be done by block-wise operations so that we do not spend too much time on rearranging bits. However, many computations are naturally structured in this way, i.e., that the same pattern of gates occurs many times in parallel, one can think here of arithmetic on large numbers, for instance.