Implementing AES via an Actively/Covertly Secure Dishonest-Majority MPC Protocol

Ivan Damgård[1], Marcel Keller[2], Enrique Larraia[2], Christian Miles[2], and Nigel P. Smart[2]

[1] Department of Computer Science, University of Aarhus, IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark

[2] Dept. Computer Science, University of Bristol, Woodland Road, Bristol, BS8 1UB, United Kingdom

Abstract. We describe an implementation of the protocol of Damgård, Pastro, Smart and Zakarias (SPDZ/Speedz) for multi-party computation in the presence of a dishonest majority of active adversaries. We present a number of modifications to the protocol; the first reduces the security to covert security, but produces significant performance enhancements; the second enables us to perform bit-wise operations in characteristic two fields. As a bench mark application we present the evaluation of the AES cipher, a now standard bench marking example for multi-party computation. We need examine two different implementation techniques, which are distinct from prior MPC work in this area due to the use of MACs within the SPDZ protocol. We then examine two implementation choices for the finite fields; one based on finite fields of size 2^8 and one based on embedding the AES field into a larger finite field of size 2^{40}.

1 Introduction

The invention of secure multi-party computation is one of the crowning achievements of theoretical cryptography, yet despite being invented around twenty-five years ago it has only recently been implemented and tested in practice. In the last few years a number of MPC "systems" have appeared [4,7,8,9,12,15,22], as well as experimental research results [13,16,21,25,26].

The work (both theoretical and practical) can be essentially divided into two camps. On one side we have techniques based on Yao circuits [28], which are mainly focused on two party computations, and on the other we have techniques based on secret sharing [6,11], which can be applied to more general numbers of players. This is rather a coarse divide as some techniques, such as that from [25], only apply in the two party case but it is based on secret sharing as opposed to Yao circuits. Following this coarse divide we can then divide work into those which consider only honest-but-curious adversaries and those which consider more general active adversaries.

I. Visconti and R. De Prisco (Eds.): SCN 2012, LNCS 7485, pp. 241–263, 2012.
© Springer-Verlag Berlin Heidelberg 2012

As in theory, it turns out that in practice obtaining active security is a much more challenging task; requiring more computational and communication resources. All prior implementation reports to our knowledge for active adversaries have either been in the two party setting, or have restricted themselves to the multi-party setting with honest majority. In the two party setting one can adopt specialist protocols, such as those based on Yao circuits, whilst the restriction to honest majority in the multi-party setting means that cheaper information theoretic constructions can be employed. Recently, Damgård et al [14] following on from work in [5], presented an actively secure protocol (dubbed "SPDZ" and pronounced "Speedz") in the multi-party setting which is secure in the presence of dishonest majority. The paper [14] contains some simple implementation results, and extrapolated estimates, but it does not report on a fully working implementation which computes a specific function.

Whilst active security is the "gold standard" of security, many applications can accept a weaker notion called covert security [1,2]. In this model a dishonest party deviating from the protocol will be detected with high probability; as opposed to the overwhelming probability required by active security. Due to the weaker requirements, covert security can often be achieved for less computational effort.

Our Contribution. As already remarked much progress has been made on implementation of MPC protocols in the last few years, but most of the "fast" implementations have been for simpler security models. For example prior work has focused on protocols for two party computation only, or honest-but-curious adversaries only, or for threshold adversaries only. In this work we extend the prior implementation work to the most complex setting namely covert and active security against a dishonest majority. In addition we examine more than four players; with some experiments being carried out with ten players. Thus our work shows that even such stringent security requirements and parameter settings are beginning to be within reach of practical application of MPC technology.

More concretely, we show how to simplify the SPDZ protocol so that it achieves covert security for a greatly improved computational performance, we present the first implementation results for the SPDZ protocol (in both the active and covert cases), and we describe an evaluation of the AES functionality with this protocol. Our protocol implementation is in the random oracle model, specifically the zero-knowledge proofs required by SPDZ are implemented using the Fiat–Shamir heuristic. We also simplify some other parts of the SPDZ protocol in the random oracle model (details are provided below), and present extensions to enable bit-wise operations in characteristic two fields.

Since the work of [26] it has become common to measure the performance of an MPC protocol with the time it takes to evaluate the AES functionality. This is for a number of reasons: Firstly AES provides a well understood function which is designed to be highly non-linear, secondly AES has a regular and highly mathematical structure which allows one to investigate various different optimization techniques in a single function, and thirdly "oblivious" evaluation

of AES on its own is an interesting application which if one could make it fast enough could have practical application.

The paper is structured as follows. We start by covering details of prior work on using MPC to implement AES. In Section 3 we detail the basics of the SPDZ protocol and the minor changes we made to the presentation in [14]. Then in Section 4 we describe how we implemented the S-Box, this is the only non-linear component in AES and so it is the only part which requires interaction. Finally in Section 5 we present our implementation results.

2 Prior Work on Evaluating AES via MPC Protocols

As noted earlier the first MPC evaluation of the AES functionality was presented in [26]. This paper presented a protocol for the case of two parties, using Yao circuits as the basic building block. On their own Yao circuits only provide security against semi-honest adversaries, and in this case the authors obtained a run-time of 7 seconds to evaluate a single AES block (the model being that party A holds the key, and party B holds a message, with B wishing to obtain the encryption of their message under A's key). To obtain security against active adversaries a variant of the cut-and-choose methodology of Lindell and Pinkas [20] was used, this resulted in the run-time dropping to 19 minutes to evaluate an AES encryption.

In [15] Henecka et al again look at two-party computation based on Yao circuits, but restrict to the case of semi-honest adversaries only. They reduce the run time per block from the previous 7 seconds down to 3.3 seconds. Huang et al [16] improve this even further obtaining a time of 0.2 seconds per block for semi-honest adversaries.

In [25] the authors present a two party protocol, but instead of their protocol being based on Yao circuits they instead base it on OT extension in the Random Oracle Model, and a form of "secret sharing with MACs" (similar to the SPDZ protocol which we examine below). This enables the authors to obtain active security and to improve on the prior performance of other implementations. The run time for a single evaluation of the AES circuit is 64 seconds, however this drops to around 2.5 seconds when amortized over a number of encryption blocks.

The most recent result in the two party setting is [17], which returns to using Yao circuit based protocols. By use of clever engineering of the overall run-time design the authors are able to significantly improve the execution time for a single AES evaluation down to 1s in the case of active adversaries.

Moving to the case of more than two players, all prior implementation results have either been for three or four players; and have been in the semi-honest setting for the case of three players. Like our work, in this setting one utilizes secret sharing but prior work has been based on Shamir secret sharing, or specialised protocols; and in the case of active security has been based on Verifiable Secret Sharing.

The main paper which is related to our work is that of [13], so we now spend some time to explain the differences between our approach and that of [13]. In [13]

the authors examine an AES implementation in the case of standard threshold-secret-sharing based MPC protocols. An implementation for one semi-honest adversary amongst three players and one active adversary amongst four players is described using the VIFF framework [12]. The VIFF framework works much like the SPDZ protocol, in that it utilizes Beaver's [3] method for MPC evaluation. In an Offline Phase "multiplication triples" are produced, and then in an Online Phase the function specific calculation is performed. The two key differences between the protocol in [13] and the use of SPDZ is that the method to produce the triples is different, and the method to ensure non-cheating adversaries during the evaluation of the circuit is also different. These differences are induced since [13] is interested in threshold adversary structures, whereas we are interested in the more challenging case of dishonest majority.

The protocol of [13] is however similar to our work in that it looks at the AES circuit as a circuit over the finite field $\mathbb{F}_{2^8}$, and not as an arbitrary binary circuit. The S-Box in AES is (usually) composed of two operations an inversion in the field $\mathbb{F}_{2^8}$ followed by a linear operation on the bits of the resulting element. In [13] the authors discuss various techniques for computing the inversion, and for the bitwise linear operation they utilize a trick of bit-decomposition of the shared value. This bit-decomposition is itself implemented using the technique of pseudorandom secret sharing (PRSS) of bits.

For MPC protocols based on Shamir secret sharing, obtaining a PRSS is relatively straight forward, indeed it is a local operation assuming some set-up. However, for protocols using secret sharing with MACs (as in our approach) it is unknown how to build a PRSS in such a clean way. Thus we produce such shared random bits by executing another stage in the Offline Phase of the SPDZ protocol. We also present a simplification of the technique in [13] to use such bit-decompositions to implement the S-Box. This approach does however assume that the Offline Phase somehow "knows" that the computed function will required shared random bits; which defeats the point of having a function independent Offline stage and also adds to the run time of the Offline stage. Thus we also present a distinct approach which utilizes a surprising algebraic formulation of the S-Box.

The implementation of [13] required less than 2 seconds per AES block (including key expansion) when computing with three players and at most one semi-honest adversary, and less than 7 seconds per AES block when computing with four players and at most one active adversary. These times include the time for the Offline Phase. If one is only interested in the Online Phase times, then the active adversary case can be executed in between three and four seconds per AES block.

More recent work has focused on the case of semi-honest adversaries and three players only. Two recent results [18,19] have used an additive secret sharing scheme and a novel multiplication protocol to perform semi-honest three party MPC in the presence of at most one adversary. In [18] the authors present an AES implementation using a novel implementation of the S-Box component via an MPC table-lookup procedure. They report being able to perform 67 AES block

cipher evaluations per second. In [19] the authors report on an implementation of AES, using the Sharemind framework [7], in which they can accomplish over one thousand AES block cipher evaluations per second.

In summary Table 1 summarizes the different performance figures and security models for prior work on implementing AES using multi-party computation, with also a comparison with our own work. Like all network based protocols a significant time can be spent waiting for data, thus authors have found that executing many calculations in parallel (as in for example AES-CTR mode) can have significant performance enhancements. Thus for papers which report such results we give the improved amortized costs for multiple executions (or just the blocks-per-second count for a single execution if no improvement via amortization occurs). However, single execution costs are still important since this deals with the case of (for example) AES-CBC mode. In our implementation we found little gain in performing multiple AES evaluations in parallel.

Table 1. A comparison of different MPC implementations of AES. We only give the online-times for those protocols which have a pre-processing phase. We also note whether the implementation assumes a pre-expanded key or not.

Paper	Security	Total Number Parties	Max Number Adv.	Time for single AES Block	(Amortized) Blocks per Sec	Expanded Key	Notes
[26]	semi-honest	2	1	7.0s	0.1	N	Yao
[15]	semi-honest	2	1	3.3s	0.3	N	Yao
[16]	semi-honest	2	1	0.2s	5.0	Y	Yao
[13]	semi-honest	3	1	1.2s	0.9	N	Shamir
[18]	semi-honest	3	1	N/A	67	Y	Additive
[19]	semi-honest	3	1	1.0s	1893	Y	Additive
[26]	covert	2	1	95s	≈ 0	N	Yao
This work	covert	2	1	0.17s	10.3	Y	SPDZ
This work	covert	3	2	0.19s	9.6	Y	SPDZ
This work	covert	4	3	0.18s	9.2	Y	SPDZ
This work	covert	5	4	0.19s	7.4	Y	SPDZ
This work	covert	10	9	0.23s	5.2	Y	SPDZ
[26]	active	2	1	19m	≈ 0	N	Yao
[25]	active	2	1	4.0s	32	N	OT
[17]	active	2	1	1.0s	1.0	Y	Yao
[13]	active	4	1	2.1s	0.5	N	Shamir
This work	active	2	1	0.26s	5.0	Y	SPDZ
This work	active	3	2	0.29s	4.7	Y	SPDZ
This work	active	4	3	0.32s	4.6	Y	SPDZ
This work	active	5	4	0.34s	4.4	Y	SPDZ
This work	active	10	9	0.41s	3.6	Y	SPDZ

In interpreting the table one needs to note that Yao based experiments usually implement a different functionality. Namely, the circuit constructor is the player

holding the key. Whether the key is expanded or not refers to whether the garbled circuit has this key hardwired in or not.

3 The SPDZ Protocol

We now give an overview of the SPDZ protocol, for more details see [14]. The reader should however note we make a number of minor alterations to the basic protocol, all of which are describe below. Some of these alterations are due to us working in the random oracle model (which enables us to simplify a number of sub-protocols), whilst some are simply a functional change in terms of how inputs to the parties are created and distributed. In addition we describe how to simplify the SPDZ protocol to the case of covert adversaries.

The SPDZ protocol, being based on the Beaver circuit randomization technique [3], comes in two phases. In the first phase a large number of random triples are produced, such that each party holds a share of the triple, and such that the underlying values in the triple satisfy a multiplicative relation. This phase is referred to as the "Offline Phase" since the triples do not depend on either the function to be evaluated (bar their number should exceed a constant multiple of the number of multiplication gates in the evaluated function), and the triples do not depend on the inputs to the function to be evaluated. In the second phase, called the "Online Phase" the triples are used to evaluate the function on the given input.

The key to understanding the SPDZ protocol is to note that all values are shared with respect to a non-standard secret sharing scheme, which incorporates a MAC value. To describe this secret sharing scheme we fix a finite field $\mathbb{F}_q$. The MAC keys are values $\alpha_j \in \mathbb{F}_q$ for $1 \leq j \leq n_{\mathsf{MAC}}$ such that player i holds the share $\alpha_{j,i} \in \mathbb{F}_q$ where

$$\alpha_j = \alpha_{j,1} + \cdots + \alpha_{j,n}.$$

The shared values are then given by the following sharing of a value $a \in \mathbb{F}_q$,

$$\langle a \rangle := (\delta, (a_1, \ldots, a_n), (\gamma_{j,1}, \ldots, \gamma_{j,n})_{j=1}^{n_{\mathsf{MAC}}}),$$

where a is the shared value, δ is public and we have the equalities

$$a = a_1 + \cdots + a_n,$$
$$\alpha_j \cdot (a + \delta) = \gamma_{j,1} + \cdots + \gamma_{j,n} \text{ for } 1 \leq j \leq n_{\mathsf{MAC}}.$$

Given this data representing a shared value a each player P_i holds the data $(\delta, a_i, \{\gamma_{j,i}\}_{j=1}^{n_{\mathsf{MAC}}})$. To ease notation we write $\gamma_{j,i}(a)$ to denote the share of the jth MAC on item a held by party i. Arithmetic in this representation is componentwise, more precisely we have

$$\langle a \rangle + \langle b \rangle = \langle a + b \rangle, \quad e \cdot \langle a \rangle = \langle e \cdot a \rangle \quad \text{and} \quad e + \langle a \rangle = \langle e + a \rangle,$$

where

$$e + \langle a \rangle = (\delta - e, (a_1 + e, a_2, \ldots, a_n), (\gamma_{j,1}, \ldots, \gamma_{j,n})_{j=1}^{n_{\mathsf{MAC}}}).$$

The simplicity of the above method for adding a constant value to $\langle a \rangle$ is the reason of the public value δ. In [14] the presentation is simplified to having only $n_{\mathsf{MAC}} = 1$, however the case of more general values of n_{MAC} is discussed. In our implementation having $n_{\mathsf{MAC}} > 1$ will be vital to ensure active security when dealing with small finite fields, thus we present the more general case above.

The SPDZ protocol can tolerate active adversaries and dishonest majority (ignoring the case where one of the dishonest players aborts) amongst a total of n parties. Thus we can assume that $n - 1$ of the parties are dishonest and will arbitrarily deviate from the protocol. The SPDZ protocol guarantees that if the protocol terminates then the honest parties know that their resulting output is correct, except with a negligible probability. For active adversaries we set this probability, to mirror the choice in [14], to 2^{-40}. For covert adversaries we adapt the protocol so that the probability that a cheating adversary will be detected is lower bounded by

$$\min\left\{1 - q^{-n_{\mathsf{MAC}}}, 1 - q^{-n_{\mathsf{SAC}}}, \frac{1}{2 \cdot (n-1)}\right\},$$

where n_{MAC} and n_{SAC} are parameters to be discussed later and $\mathbb{F}_q$ is the finite field over which our triples are defined.

3.1 Offline Phase

The Offline Phase makes use of a somewhat homomorphic encryption (SHE) scheme, with a distributed decryption procedure, and zero-knowledge proofs. In our implementation we use the optimized non-interactive zero-knowledge proofs of knowledge (NIZKPoKs) derived from the Fiat–Shamir heuristic which are described in [14]. Thus our Offline Phase is only secure in the Random Oracle model.

The specific SHE scheme used is a variant of the BGV scheme [10] over the mth cyclotomic field. We thus have lattices of dimension $\phi(m)$, over a modulus of size Q. Each ciphertext consists of two (or three) polynomials modulo Q of degree less than $\phi(m)$. The underlying plaintext space can hold an element of $(\mathbb{F}_q)^\ell$.

The Offline Phase produces many triples of such sharings $\langle a \rangle, \langle b \rangle, \langle c \rangle$ such that $c = a \cdot b$, where these values are authenticated via a global set of n_{MAC} shared MAC keys as described above. The NIZKPoKs mentioned above have soundness error $1/2$, and so in [14], we "batch" together sec executions so as to reduce the soundness error to $2^{-\mathsf{sec}}$. This batching, combined with the vectoral plaintext space, means that a single execution of the Offline phase produces $\mathsf{sec} \cdot \ell$ triples.

We can trivially modify the Offline Phase so that it also outputs, for characteristic two fields, a set of shared random bits and their associated MACs. We can produce one such shared bit for roughly one third of the cost of one shared triple. As for the shared triples, each invocation of the method to produce shared random bits will produce $\mathsf{sec} \cdot \ell$ bits in one go.

The main cost of the Offline phase is in the production and verification of the zero-knowledge proofs. For n players, for each proof that a player needs to

produce he will need to verify $n-1$ proofs of the other players. For the case of covert adversaries we simplify the Offline Phase as follows. We do not batch together proofs, i.e. we take $\mathsf{sec} = 1$, which results in soundness error for each proof of $1/2$. In addition each player when it receives $n-1$ proofs from all other players only verifies a random proof. This means that a cheating player will be detected with probability at least $1/(2 \cdot (n-1))$ in the Offline phase, as opposed to $1 - 2^{-40}$ when we use the standard actively secure Offline Phase.

3.2 Online Phase

Given that our Offline Phase is given in the Random Oracle Model we alter the Online Phase from [14] so that it too utilizes Random Oracles. This means we can present a more efficient Online Phase than that used in [14]. Our Online Phase makes use of three hash functions: The first one H_1 is used to ensure that broadcast has happened, for this hash function we require it is one which supports an API of standard hash functions consisting of Init, Update and $\mathsf{Finalise}$ methods. The second hash function H_2 is used to generate random values for checking the linear MAC equations and the triples. The third hash function H_3, which we model as a random oracle, is used to define a commitment scheme as follows: To commit to a value x, which we denote by $\mathsf{Commit}(x)$, one generates a random value $r \in \{0,1\}^{\mathsf{sec}}$, for some security parameter sec, and computes $\mathsf{comm} = H_3(x \| r)$. To open $\mathsf{Open}(\mathsf{comm}, x, r)$ one verifies that $\mathsf{comm} = H_3(x \| r)$ returning x if this is true, and $\perp$ if it is not.

The first change we make is in how we guarantee that consistent broadcast occurs. For the Online phase we assume that the point-to-point links between the parties are authenticated, but we need to guarantee that a dishonest party is not allowed to send different messages to different players when he is required to broadcast a single value to all players. This is done by modifying the notion of a "partial opening" from [14] and the notion of "broadcast". The "broadcasts" are ensured to be correct via the parties maintaining a hash of all values received. This is checked before the output is reconstructed; thus in the final broadcast to recover the output we utilize the re-transmit method from [14] to check consistency of the final broadcast.

In the original protocol "partial opening" just means a broadcast of the share of a value held by a party, but not the broadcast of the share of the MAC on that value. Thus only the value is opened, not the MAC on the value. However, we each ensure player maintains the running totals of the linear equations they will eventually check. In [14] these linear equations were of the form $\sum_k e^k a_k$, for some random agreed value e. This gives an error probability of T/q, where T is the number of partial openings in an execution of the Online Phase. For small values of q this is not effective, thus we replace the values e^k by the output of hash function H_2. In Figure 1 we describe our modified partial opening, and broadcast protocol, which maintains a hash value of all values broadcast; as well as a method for checking consistency.

Init(): We initialize the following data:

1. Party i executes $H_1.\mathsf{Init}()$.
2. Party i sets $\mathsf{cnt}_i = 0$.
3. For $j = 1, \ldots, n_{\mathsf{MAC}}$
   (a) Party i sets $\hat{a}_{j,i} = 0$ and $\gamma_{j,i} = 0$.
4. Party i generates a random value $\mathsf{seed}_i \in \{0,1\}^{\mathsf{sec}}$ and sends it to all other players.

Broadcast(v_i): We broadcast v_i and receive the equivalent broadcasts from other players:

1. Party i sends v_i to each player.
2. On receipt of $\{v_1, \ldots, v_n\} \setminus \{v_i\}$ execute $H_1.\mathsf{Update}(v_1 \| \ldots \| v_n)$.
3. Return $\{v_1 + \cdots + v_n\}$.

PartialOpen($\langle a \rangle$): Party i obtains the partial opening of the shared value and updates their partial sums:

1. Execute $\{a_1, \ldots, a_n\} = \mathsf{Broadcast}(a_i)$.
2. $a = a_1 + \cdots + a_n$.
3. $(e_1 \| \ldots \| e_{n_{\mathsf{MAC}}}) = H_2(0 \| \mathsf{seed}_1 \| \ldots \| \mathsf{seed}_n \| \mathsf{cnt}_i) \in \mathbb{F}_q$.
4. $\mathsf{cnt}_i = \mathsf{cnt}_i + 1$
5. For $j = 1, \ldots, n_{\mathsf{MAC}}$
   (a) $\hat{a}_{j,i} = \hat{a}_{j,i} + e_j \cdot (a + \delta_a)$.
   (b) $\gamma_{j,i} = \gamma_{j,i} + e_j \cdot \gamma_{j,i}(a)$.
6. Return a.

Verify(): We check all broadcasts have been consistent:

1. Party i computes $h_i = H_1.\mathsf{Finalise}()$ and sends h_i to each player.
2. On receipt of h_j from player j, if $h_i \neq h_j$ then abort.

Fig. 1. Methods for Partial Opening and Broadcast for Party i

In the Online Phase the key issue is that the triples produced by the Offline Phase may not satisfy the relation $c = a \cdot b$, nor may the MACs verify. This is because we do not ensure that the dishonest parties were "well behaved" in the Offline Phase. Thus these two properties must be checked. The Online Protocol of [14] does this as follows: To check that $c = a \cdot b$ for the triples, we will use for the MPC evaluation we "sacrifice" a set of n_{SAC} extra triples per evaluated triple. For the sacrificing method in our implementation, we adopted the naïve method of [14]. This results in consuming more triples, but is simpler computationally. To check the MAC values a series of n_{MAC} linear equations are checked at the end of the Online Phase.

Each triple sacrifice and MAC equation check can be made to hold by the adversary with probability $1/q$. Thus to reduce this to something negligible we sacrifice many triples, and utilize many MAC equations. But in the case of covert adversaries we select $n_{\mathsf{MAC}} = n_{\mathsf{SAC}} = 1$, and so the probability of a cheating adversary being detected is bounded from below by $1 - 1/q$.

Both of these checks require that the parties agree on some global random values at different points in the protocol. In [14] these extra shared values are determined in the Offline Phase, via a different form of secret sharing; with the sharings being opened at the critical point in the Online protocol. The benefit

of this approach is that one obtains a protocol which is UC secure without the need for Random Oracles; however the down-side is that the Offline Phase becomes relatively complex. In our work we take the view that since Random Oracles have been used in the Offline Phase one might as well exploit them in the Online Phase. Thus these shared values are obtained via a Random Oracle based commitment scheme as we now describe.

The next alteration we make to the Online Phase of [14] is that we assume that the players shares of the input values are "magically distributed" to them. This can be justified in two ways. Firstly we are only interested in timing the main Offline and Online Protocol and the input distribution phase is just an added complication. Secondly, a key application scenario for MPC is when the players are computing a function on behalf of some client. In such a situation the players do not themselves have any input, it is the client which has input. In such a situation the players would obtain their respective input shares directly from the client; thus eliminating the need entirely for a special protocol to deal with obtaining the input shares.

Our final alteration is that we utilize a new online operation, in addition to local addition and multiplication, called BitDecomposition. We first note that we can given a sharing $\langle a \rangle$ of a finite field element $a \in \mathbb{F}_{2^k} = \mathbb{F}_2[X]/F(X)$, and a set of k randomly shared bits $\langle r_i \rangle$ for $i = 0, \ldots, k-1$. Suppose we write a as $\sum_{i=0}^{k-1} a_i \cdot X^i$, our goal is to produce $\langle a_i \rangle$. Firstly via a local operation we compute a sharing of $r = \sum r_i \cdot X^i$ by computing $\langle r \rangle = \sum \langle r_i \rangle \cdot X^i$. Then we produce a masked value of a, via $\langle c \rangle = \langle a \rangle + \langle r \rangle$. The value of $\langle c \rangle$ is then opened to reveal c and we compute the decomposition $c = \sum c_i \cdot X^i$. Then we can locally compute $\langle a_i \rangle = c_i + \langle r_i \rangle$. Note, if a is known to be in a subfield of $\mathbb{F}_{2^k}$, as it will be in one of our implementations for $k = 40$, we can utilize the embedding of the subfield into the larger field to reduce the number of shared random bits needed for this decomposition down to the degree of the subfield. We refer to Appendix A for more details.

Given these alterations to the Online Phase of [14] we present the modified protocol in Figure 2 of the Appendix.

4 S-Box Implementation

We present two distinct methodologies to implement the S-Box. The first requires the Offline Phase to only produce multiplication triples, and utilizes the algebraic properties of the S-Box. The second requires the Offline Phase to also produce sharings (and associated MACs) of random bits.

4.1 S-Box via Algebraic Operations

A key design criteria of any block cipher is that it should be highly non-linear. In addition it should be hard to write down a series of simple algebraic equations to describe the cipher. Since such equations could give rise to an attack via algebraic cryptanalysis. Indeed one reason for choosing AES as an example benchmark for

MPC protocols, is that being a block cipher it should be highly non-linear and hence a challenge for MPC protocols. However, as was soon realised after the standardization of AES the S-Box (the only non-linear component in the entire cipher) can be represented in a relatively clean algebraic manner.

Our algebraic method to implement the S-Box operation is based on the analysis of AES of Murphy and Robshaw [23]. In this work the authors demonstrate that actually AES can be described by (relatively simple) algebraic formulae over $\mathbb{F}_{2^8}$, in other words the transform between byte-wise and bit-wise operations in the standard representation of the AES S-Box is a bit of a MacGuffin.

Recall the AES S-Box consists of an inversion in $\mathbb{F}_{2^8}$ (which is indeed a highly non-linear function) followed by a linear operation over the bits of the result. This is usually explained that the mixture of the two operations in two distinct finite fields "breaks any algebraic structure". This was shown to be false in [23]. Indeed one can express the S-Box calculation via the following simple polynomial

$$\begin{aligned}\text{S-Box}(z) = \text{0x63} + \text{0x8F} \cdot z^{127} + \text{0xB5} \cdot z^{191} + \text{0x01} \cdot z^{223} + \text{0xF4} \cdot z^{239} \\ + \text{0x25} \cdot z^{247} + \text{0xF9} \cdot z^{251} + \text{0x09} \cdot z^{253} + \text{0x05} \cdot z^{254}.\end{aligned}$$

where (as is usual) operations are in the finite field defined by $\mathbb{F}_{2^8} = \mathbb{F}_2[x]/(x^8 + x^4 + x^3 + x + 1)$ and the notation 0x12 represents the element defined by the polynomial $x^4 + x$. That the operation can be defined by a polynomial of degree bounded by 255 is not surprising, since by interpolation any functions from $\mathbb{F}_{2^8}$ to $\mathbb{F}_{2^8}$ can be represented in such a way. What is surprising is that the polynomial is relatively sparse, however this can be easily shown from first principles.

Lemma 1. *The AES S-Box can be represented by a polynomial which has a non-zero coefficient for the term i if and only if $i \in \{0, 127, 191, 223, 239, 247, 251, 253, 254\}$.*

Proof. Recall the AES S-Box consists first of inversion $z \to z^{-1} = y$ followed by an $\mathbb{F}_2$ linear operation $\mathbf{w} = A \cdot \mathbf{y}^{\mathsf{T}} + \mathbf{b}$ on the bits of the result, where $\mathbf{y}$ are the bits in y. The bit matrix A and the bit vector $\mathbf{b}$ are fixed. The final result is obtained by forming the dot-product of the $(\mathbb{F}_2)^8$ vector $\mathbf{w}$ with the fixed vector $\mathbf{x} = (1, x, x^2, x^3, x^4, x^5, x^6, x^7) \in (\mathbb{F}_{2^8})^8$.

First note that inversion in $\mathbb{F}_{2^8}$ can be accomplished by computing $z^{-1} = z^{254}$, since $z^{255} = 1$ for all $z \neq 0$. The AES standard "defines" $0^{-1} = 0$, and so the formula of z^{254} can be applied even when $z = 0$ as well.

We then note that extracting the bits $\mathbf{y} = (y_0, \ldots, y_7) \in (\mathbb{F}_2)^8$ of an element $y = y_0 + y_1 \cdot x + \cdots + y_7 \cdot x^7$ can be obtained via a linear operation on the action of Frobenius on y. This follows since Frobenius acts as a linear map, and hence by applying Frobenius eight times we find eight linear equations linking the set $\{y_0, \ldots, y_7\}$ with the Frobenius actions on y. This in turn allows us to solve for the bits $\mathbf{y} = (y_0, \ldots, y_7)$. Thus there is matrix $B \in (\mathbb{F}_{2^8})^{8\times 8}$ such that

$$\mathbf{y} = B \cdot (y, y^2, y^4, y^8, y^{16}, y^{32}, y^{64}, y^{128})^{\mathsf{T}}.$$

Hence, the output of the S-Box can be written as

$$\begin{aligned}\text{S-Box}(z) &= \mathbf{x} \cdot (A \cdot \mathbf{y} + \mathbf{b}),\\ &= \mathbf{x} \cdot (A \cdot B) \cdot (y, y^2, y^4, y^8, y^{16}, y^{32}, y^{64}, y^{128})^{\mathsf{T}} + \mathbf{x} \cdot \mathbf{b},\\ &= \mathbf{s} \cdot (1, y, y^2, y^4, y^8, y^{16}, y^{32}, y^{64}, y^{128})^{\mathsf{T}}\end{aligned}$$

where $\mathbf{s}$ is a fixed nine dimensional vector over $\mathbb{F}_{2^8}$. On replacing y with z^{254} in the above equation, using $z^{255} = 1$ for all $z \neq 0$, we obtain our result. With the result also following for $z = 0$ by inspection.

Finally to implement the S-Box we therefore need an efficient method to obtain from an shared input value z, the shared values of the elements $\{z^{127}, z^{191}, z^{223}, z^{239}, z^{247}, z^{251}, z^{253}, z^{254}\}$. This is equivalent to finding a short addition chain for the set $\{127, 191, 223, 239, 247, 251, 253, 254\}$. We found the shortest such addition chain consists of eighteen additions and is the chain

$$\{1, 2, 3, 6, 12, 15, 24, 48, 63, 64, 96, 127, 191, 223, 239, 247, 251, 253, 254\}.$$

Thus to evaluate a single S-Box requires eighteen MPC multiplication operations, as well as some local computation. Hence, to evaluate the entire AES cipher we require $18 \cdot 16 \cdot 10 = 2880$ MPC multiplications.

Looking ahead each multiplication operation will require interaction, and to reduce execution times we need to ensure that each player is kept "busy", i.e. is not left waiting for data to arrive. To do this we will interleave various different multiplications together; essentially exploiting the instruction level parallelism (ILP) within the basic AES algorithm. Clearly one can execute each of the 16 S-Box operations in a single round in parallel, thus obtaining an immediate 16-fold factor of ILP. However, further ILP can be exploited in the addition chain above as can be seen from its graphical realisation in Figure B. in the Appendix. We see that the addition chain can be executed in twelve parallel multiplication steps; thus the total number of rounds of multiplication need for the entire AES cipher will be $12 \cdot 10 = 120$.

4.2 S-Box via BitDecomposition

As explained in [13] the S-Box can be implemented if one has access to shared random bits, via the BitDecomposition operation. In our second implementation choice we extend this technique, and reduce even further the amount of interaction needed to compute the S-Box.

We use this BitDecomposition trick in two ways. The first way is to decompose an element in $\mathbb{F}_{2^8}$ into it's bit components, so as to apply the linear map of the S-Box. This part is exactly as described in [13]; except when we open the value of $\langle c \rangle$ we perform a partial opening, leaving the checking of the MACs until the end.

In our second application of BitDecomposition we use BitDecomposition to implement the operation $x \longrightarrow x^{254}$. This done as follows: We decompose x into it's constituent bits. Then the operations $x \longrightarrow x^2$, $x \longrightarrow x^4$ are all *linear*

operations, and so can be performed locally. Finally the value of $x^{254} = x^{-1}$ is computed via the combination

$$x^{254} = \left(\left(x^2 \cdot x^4\right) \cdot \left(x^8 \cdot x^{16}\right)\right) \cdot \left(\left(x^{32} \cdot x^{64}\right) \cdot x^{128}\right),$$

which requires a total of six multiplications. We could reduce this down to four multiplications by applying the Frobenius map to other elements [27]; but this will consume even more random bits per S-Box thus we settled for the above implementation which consumes 16 sharings of random bits per S-Box invocation.

5 Experimental Results

We implemented the SPDZ protocol over finite fields of characteristic two and used it to evaluate the AES function, with the S-Box implemented using both the algebraic formulation described earlier and the variant by BitDecomposition. As described earlier we examined the case of dealing with both covert adversaries and fully malicious (a.k.a. active) adversaries (with cheating probability of 2^{-40}). We note that the probability of 2^{-40} could be extended to smaller values, but we used 2^{-40} so as to be comparable with the theoretical run-time estimates given in [14]. For example to reduce the probability down to 2^{-80} would essentially require a doubling of the cost of both the Offline and Online stages.

The first decision one needs to take is as to what finite field one should work with. Since we are evaluating AES it is natural to pick the field

$$K_8 = \mathbb{F}_2[x]/(x^8 + x^4 + x^3 + x + 1).$$

Another choice, particularly suited to our active adversary cheating probability of 2^{-40}, would be to use the field

$$K_{40} = \mathbb{F}_2[y]/(y^{40} + y^{20} + y^{15} + y^{10} + 1).$$

Using this finite field has the advantage that, for active adversaries, we only need to keep one MAC share per data item, and only one triple per multiplication needs to be sacrificed. In addition the field K_8 lies in K_{40} via the embedding $x = y^5 + 1$. We also for means of comparison of the Offline phase implemented the Offline protocol over a finite field $\mathbb{F}_q$ with q a 64-bit prime.

We also experimented with various numbers of players, and different values of n_{MAC} and n_{SAC}. As explained in [14] all such variants lead to different basic parameters (m, Q, ℓ) of the underlying SHE scheme.

We now determine values of (m, Q, ℓ) for our SHE scheme given a specific finite field $\mathbb{F}_q$ (or in the case of q prime a rough size for q), a value for the sec (the number of NIZKPoKs we run in parallel in the Offline stage), and the number of players n. As a "lattice security parameter" we selected $\delta = 1.0052$ which corresponds to roughly 128 bits of symmetric security.

We require finite fields $\mathbb{F}_q$ of size $\mathbb{F}_{2^8}$ and $\mathbb{F}_{2^{40}}$, as well for comparison a finite field where q was a 64-bit prime. We also looked for parameters for $n \in \{2, 3, 4, 5, 10\}$ and $\mathsf{sec} \in \{1, 40\}$. As in [14] we first search for rough estimate of the parameters (m, Q) which fit these needs:

$\mathsf{char}(\mathbb{F}_q)$	n	sec	$\phi(m) \geq$	$\log_2(Q)$
2	$2 \leq n \leq 10$	40	12300	370
2	$2 \leq n \leq 5$	1	8000	200
2	10	1	8000	210
$\approx 2^{64}$	$2 \leq n \leq 10$	40	16700	500
$\approx 2^{64}$	$2 \leq n \leq 5$	1	11000	330
$\approx 2^{64}$	10	1	11300	340

We then selected values for m as follows:

$\mathbb{F}_{2^8}$ and $\mathbb{F}_{2^{40}}$, $\mathsf{sec} = 40$: We select $m = 17425$, which gives us $\phi(m) = 12800$. The polynomial $\Phi_m(X)$ factors modulo two into $\ell = 320$ factors each of degree 40. Thus these parameters can support both our finite fields $\mathbb{F}_{2^8}$ and $\mathbb{F}_{2^{40}}$.

$\mathbb{F}_{2^8}$, $\mathsf{sec} = 1$: We select $m = 13107$, which gives us $\phi(m) = 8192$. The polynomial $\Phi_m(X)$ factors modulo two into $\ell = 512$ factors each of degree 16.

$\mathbb{F}_{2^{40}}$, $\mathsf{sec} = 1$: We select $m = 13175$, which gives us $\phi(m) = 9600$. The polynomial $\Phi_m(X)$ factors modulo two into $\ell = 240$ factors each of degree 40.

$p \approx 2^{64}$, $\mathsf{sec} = 40$: We select, as in [14], $p = 2^{64} + 4867$ and $m = 16729$ so that $\ell = \phi(m) = 16728$.

$p \approx 2^{64}$, $\mathsf{sec} = 1$: We select, as in [14], $p = 2^{64} + 8947$ and $m = 11971$ so that $\ell = \phi(m) = 11970$.

Recall that one invocation of the Offline Phase produces $\mathsf{sec} \cdot \ell$ triples; thus using the choices above we obtain the following summary table, where "# Trip/# Bits" denotes the number of triples/bits produced per invocation of the Offline Phase.

Field	Adversary Type	sec	$n_{\mathsf{MAC}} = n_{\mathsf{SAC}}$	# Trip/ # Bits
K_8	covert	1	1	512
K_8	active	40	5	12800
K_{40}	covert	1	1	240
K_{40}	active	40	1	12800

We ran the Offline phase on machines with Intel i5 CPU's running at 2.8 GHz. with 4 GB of RAM. The ping between machines over the local area network was approximately 0.3 ms. We obtained the executions time given in Table 2 and Table 3, for the two different finite field choices and covert/active security choices, and various numbers of players. We did not run an example with ten players and active adversaries since this took too long. We first ran the Offline Phase in each example to produce a minimum of 5000 triples. Clearly for some parameter sets a single run produced much more than 5000, whilst for others we required multiple runs so as to reach 5000 triples. These results are in Table 2. These runs are compatible with our algebraic S-Box formulation.

This table also presents the average time needed to produce each triple, plus also the amortized time to produce triples per AES invocation (in the case where one wants to evaluate the AES functionality many times). Recall to evaluate the AES functionality with our method requires $10 \cdot 16 \cdot 18 = 2880$ multiplications in total; thus the number of triples needed is $2880 \cdot (n_{\mathsf{SAC}} + 1)$, since each multiplication consumes $n_{\mathsf{SAC}} + 1$ triples. What is clear from the table is that if one is wishing to obtain security against covert adversaries then utilizing the field K_8 is preferable. However, for security against active adversaries the field K_{40} is to be preferred.

Table 2. Offline Run Time Examples For The Algebraic S-Box Method

		Covert Security			Active Security		
Field	Num. Parties	Total Time (h:m:s)	Time per Triple (seconds)	Offline time per AES blk (h:m:s)	Total Time (h:m:s)	Time per Triple (seconds)	Offline time per AES blk (h:m:s)
		No. Triples Produced: 5120			No. Triples Produced: 12800		
K_8	2	0:01:31	0.018	0:01:42	1:25:57	0.403	1:56:02
K_8	3	0:01:32	0.018	0:01:43	1:50:25	0.518	2:29:03
K_8	4	0:01:32	0.018	0:01:43	2:14:16	0.629	3:01:15
K_8	5	0:01:33	0.018	0:01:44	2:37:30	0.738	3:32:37
K_8	10	0:01:48	0.021	0:02:01	4:40:15	1.314	6:18:20
		No. Triples Produced: 5040			No. Triples Produced: 12800		
K_{40}	2	0:05:08	0.061	0:05:52	0:29:34	0.136	0:13:18
K_{40}	3	0:05:13	0.062	0:05:57	0:38:18	0.180	0:17:14
K_{40}	4	0:05:14	0.062	0:05:58	0:46:02	0.216	0:20:42
K_{40}	5	0:05:17	0.063	0:06:02	0:55:51	0.262	0:25:07
K_{40}	10	0:06:02	0.072	0:06:53	1:39:14	0.465	0:44:39

We then run an Offline phase tailored to our BitDecomposition S-Box formulation. Here we need to perform $10 \cdot 16 \cdot 6 = 960$ multiplications, and thus we require $960 \cdot (n_{\mathsf{SAC}} + 1)$ triples to evaluate a single block. But we also require $10 \cdot 16 \cdot 16 = 2560$ shared random bits so as to perform two eight bit, BitDecompositions per S-Box invocation. Thus in Table 3 we present run times for a second invocation of the Offline Phase in which we aimed to produce a minimum of 5000 triples and 6600 shared random bits (which is the correct ratio for covert security). Due to the inbalance between Triple and Bit production the "Offline Time per AES Block" column needs to be taken as rough estimate. Again we see that for covert security K_8 is preferable, and for active security K_{40} is preferable.

But, these run times do not seem comparable with the 13ms per triple estimated by the authors of [14] for the Offline Phase. However, this discrepancy can easily be explained. The run time estimates in [14] are given for arithmetic circuit evaluation over a finite field of prime characteristic of 64-bits. With the parameter choices in [14] this means one can select parameters for the SHE scheme which enable a 16000-fold SIMD parallelism. For our finite fields of degree two

Table 3. Offline Run Time Examples For The S-Box Via BitDecomposition

Field	Number Players	Covert Security: Total Time (h:m:s)	Covert Security: Offline Time per AES Block (h:m:s)	Active Security: Total Time (h:m:s)	Active Security: Offline Time per AES Block (h:m:s)
		No. Triples/Bits: 5120/6556		No. Triples/Bits: 12800/12800	
K_8	2	0:02:07	0:00:47	1:54:42	0:51:36
K_8	3	0:02:10	0:00:49	2:26:21	1:05:51
K_8	4	0:02:13	0:00:50	2:56:47	1:19:33
K_8	5	0:02:36	0:00:52	3:29:49	1:34:25
K_8	10	0:02:33	0:00:58	6:06:20	2:44:51
		No. Triples/Bits: 5040/6720		No. Triples/Bits: 12800/12800	
K_{40}	2	0:07:12	0:02:43	0:36:14	0:05:26
K_{40}	3	0:07:12	0:02:43	0:47:30	0:07:07
K_{40}	4	0:07:19	0:02:47	0:58:55	0:08:57
K_{40}	5	0:07:24	0:02:49	1:10:33	0:10:34
K_{40}	10	0:08:32	0:03:15	2:10:03	0:19:32

the amount of SIMD parallelism in the Offline Phase is much lower than this. To see the difference that using large prime characteristic fields makes to the Offline Phase we implemented it, using the parameters above to obtain the results in Table 4. As can be seen from the table we produce triples for prime fields of 64-bits in size around twice as fast as the estimates in [14] would predict.

Table 4. Offline Run Time Examples For $\mathbb{F}_p$ With $p \approx 2^{64}$

Number Players	Covert Security: Total Number Triples	Covert Security: Total Time (h:m:s)	Covert Security: Time per Triple (seconds)	Active Security: Total Number Triples	Active Security: Total Time (h:m:s)	Active Security: Time per Triple (seconds)
2	11970	0:00:27	0.002	669120	1:10:48	0.006
3	11970	0:00:27	0.002	669120	1:32:13	0.008
4	11970	0:00:28	0.002	669120	1:55:05	0.010
5	11970	0:00:29	0.002	669120	2:20:42	0.013
10	11970	0:00:31	0.002	669120	4:17:10	0.023

We now turn to the Online Phase; recall that this itself comes in two steps (and two variants). In the first step we evaluate the function itself (consuming the triples produced in the Offline Phase), whereas in the second step we check the MAC values and open the final result. In Table 5 we present the run-times to evaluate the AES functionality for the various parameter sets generated above using our algebraic formulation of the S-Box. These are average run-times from all the players, executed over 20 different runs. The Online Phase was run on the same machines as in the Offline Phase. In Table 6 we present the same times using the S-Box variant utilizing the BitDecomposition method.

Table 5. Online Phase Runtime Examples (all in seconds) – Algebraic S-Box

		Covert Security			Active Security		
Field	Number Players	Function Evaluation	Checking Step	Total Time	Function Evaluation	Checking Step	Total Time
K_8	2	0.284	0.017	0.301	1.319	0.031	1.350
K_8	3	0.307	0.062	0.369	1.381	0.035	1.416
K_8	4	0.316	0.027	0.343	1.422	0.028	1.450
K_8	5	0.344	0.034	0.378	1.461	0.018	1.479
K_8	10	0.444	0.010	0.454	1.659	0.023	1.682
K_{40}	2	0.449	0.012	0.461	0.460	0.021	0.481
K_{40}	3	0.486	0.022	0.498	0.475	0.025	0.500
K_{40}	4	0.490	0.042	0.532	0.486	0.055	0.541
K_{40}	5	0.508	0.037	0.544	0.510	0.026	0.536
K_{40}	10	0.765	0.021	0.786	0.672	0.017	0.689

Table 6. Online Phase Runtime Examples (all in seconds) – S-Box Via BitDecomposition

		Covert Security			Active Security		
Field	Number Players	Function Evaluation	Checking Step	Total Time	Function Evaluation	Checking Step	Total Time
K_8	2	0.156	0.009	0.165	0.569	0.011	0.580
K_8	3	0.178	0.008	0.186	0.616	0.019	0.635
K_8	4	0.169	0.015	0.184	0.620	0.015	0.635
K_8	5	0.173	0.019	0.192	0.727	0.019	0.746
K_8	10	0.211	0.015	0.226	0.722	0.044	0.766
K_{40}	2	0.260	0.006	0.266	0.256	0.004	0.260
K_{40}	3	0.303	0.009	0.312	0.279	0.011	0.290
K_{40}	4	0.303	0.010	0.313	0.287	0.029	0.316
K_{40}	5	0.319	0.022	0.341	0.319	0.016	0.335
K_{40}	10	0.399	0.016	0.415	0.387	0.027	0.414

The networking between players was implemented in a point-to-point fashion with each player acting as both a server and a client. We ensured that data was sent over the sockets as soon as it was ready by disabling Nagle's algorithm [24]. To complete the function evaluation each player first parses a program written in a specialised instruction language. This allows our implementation to take advantage of the instruction level parallelism as described above so as to schedule many multiplication operations to happen in parallel.

Again we see that if security against covert adversaries is the goal then using the field K_8 is to be preferred. However, for security against active adversaries the field K_{40} performs better. We also ran the Online Phase in a run which performed ten AES encryptions in parallel. This resulted in only a small improvement in time per AES block over executing just one AES encryption at a time, thus we do not present these figures. Improving the throughput for parallel execution is the subject of future research.

Overall, the two methods of AES evaluation are roughly comparable. The method via BitDecomposition being faster, and significantly faster when one also takes into account the associated cost of the Offline Phase. However, as remarked previously this method does not result in a generic Offline Phase; since the Offline Phase needs to "know" the expected ratio of Bits to Triples that it needs to produce for the actual function which will be evaluated in the Online Phase.

In summary we have presented the first experimental results for running MPC protocols with large numbers of players (10 as opposed to the four or less of prior work), and for a dishonest majority of active or covert adversaries (as opposed to threshold adversaries). It is expected that our reported execution times will fall as dramatically as those have done for two party MPC protocols in the last couple of years. Thus we can expect actively/covertly secure MPC protocols for dishonest majority to be within reach of some practical applications within a few years.

Acknowledgements. The first author acknowledges the support from the Danish National Research Foundation and The National Science Foundation of China (under the grant 61061130540) for the Sino-Danish Center for the Theory of Interactive Computation, within which [part of] this work was performed; and also from the CFEM research center (supported by the Danish Strategic Research Council) within which part of this work was performed.

The second, third and fifth author were partially supported by EPSRC via grant COED–EP/I03126X. The fifth author was also supported by the European Commission through the ICT Programme under Contract ICT-2007-216676 ECRYPT II and via an ERC Advanced Grant ERC-2010-AdG-267188-CRIPTO, the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under agreement number FA8750-11-2-0079, and by a Royal Society Wolfson Merit Award. The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, AFRL, the U.S. Government, the European Commission or EPSRC.

References

1. Aumann, Y., Lindell, Y.: Security Against Covert Adversaries: Efficient Protocols for Realistic Adversaries. In: Vadhan, S.P. (ed.) TCC 2007. LNCS, vol. 4392, pp. 137–156. Springer, Heidelberg (2007)
2. Aumann, Y., Lindell, Y.: Security against covert adversaries: Efficient protocols for realistic adversaries. J. Cryptology 23, 281–343 (2010)
3. Beaver, D.: Correlated pseudorandomness and the complexity of private computations. In: Symposium on Theory of Computing, STOC 1996, pp. 479–488. ACM (1996)

4. Ben-David, A., Nisan, N., Pinkas, B.: FairplayMP: a system for secure multi-party computation. In: Computer and Communications Security, CCS 2008, pp. 257–266. ACM (2008)
5. Bendlin, R., Damgård, I., Orlandi, C., Zakarias, S.: Semi-homomorphic Encryption and Multiparty Computation. In: Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 169–188. Springer, Heidelberg (2011)
6. Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for non-cryptographic fault-tolerant distributed computation. In: Symposium on Theory of Computing, STOC 1988, pp. 1–10. ACM (1988)
7. Bogdanov, D., Laur, S., Willemson, J.: Sharemind: A Framework for Fast Privacy-Preserving Computations. In: Jajodia, S., Lopez, J. (eds.) ESORICS 2008. LNCS, vol. 5283, pp. 192–206. Springer, Heidelberg (2008)
8. Bogetoft, P., Christensen, D.L., Damgård, I., Geisler, M., Jakobsen, T., Krøigaard, M., Nielsen, J.D., Nielsen, J.B., Nielsen, K., Pagter, J., Schwartzbach, M., Toft, T.: Secure Multiparty Computation Goes Live. In: Dingledine, R., Golle, P. (eds.) FC 2009. LNCS, vol. 5628, pp. 325–343. Springer, Heidelberg (2009)
9. Bogetoft, P., Damgård, I., Jakobsen, T., Nielsen, K., Pagter, J.I., Toft, T.: A Practical Implementation of Secure Auctions Based on Multiparty Integer Computation. In: Di Crescenzo, G., Rubin, A. (eds.) FC 2006. LNCS, vol. 4107, pp. 142–147. Springer, Heidelberg (2006)
10. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: Fully homomorphic encryption without bootstrapping. In: Innovations in Theoretical Computer Science, ITCS 2012, pp. 309–325. ACM (2012)
11. Chaum, D., Crepeau, C., Damgård, I.: Multiparty unconditionally secure protocols. In: Symposium on Theory of Computing – STOC 1988, pp. 11–19. ACM (1988)
12. Damgård, I., Geisler, M., Krøigaard, M., Nielsen, J.B.: Asynchronous Multiparty Computation: Theory and Implementation. In: Jarecki, S., Tsudik, G. (eds.) PKC 2009. LNCS, vol. 5443, pp. 160–179. Springer, Heidelberg (2009)
13. Damgård, I., Keller, M.: Secure Multiparty AES. In: Sion, R. (ed.) FC 2010. LNCS, vol. 6052, pp. 367–374. Springer, Heidelberg (2010)
14. Damgård, I., Pastro, V., Smart, N.P., Zakarias, S.: Multiparty Computation from Somewhat Homomorphic Encryption. In: Safavi-Naini, R. (ed.) CRYPTO 2012. LNCS, vol. 7417, pp. 643–662. Springer, Heidelberg (2012), `http://eprint.iacr.org/2011/535`
15. Henecka, W., Kögl, S., Sadeghi, A.-R., Schneider, T., Wehrenberg, I.: TASTY: Tool for automating secure two-party computations. In: Computer and Communications Security, CCS 2010, pp. 451–462. ACM (2010)
16. Huang, Y., Evans, D., Katz, J., Malka, L.: Faster secure two-party computation using garbled circuits. In: Proc. USENIX Security Symposium (2011)
17. Kreuter, B., Shelat, A., Shen, C.-H.: Towards billion-gate secure computation with malicious adversaries. IACR e-print 2012/179 (2012), `http://eprint.iacr.org/2012/179`
18. Launchbury, J., Adams-Moran, A., Diatchki, I.: Efficient lookup-table protocol in secure multiparty computation (2012) (manuscript)
19. Laur, S., Talviste, R., Willemson, J.: AES block cipher implementation and secure database join on the SHAREMIND secure multi-party computation framework (2012) (manuscript)
20. Lindell, Y., Pinkas, B.: An Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries. In: Naor, M. (ed.) EUROCRYPT 2007. LNCS, vol. 4515, pp. 52–78. Springer, Heidelberg (2007)

21. Lindell, Y., Pinkas, B., Smart, N.P.: Implementing Two-Party Computation Efficiently with Security Against Malicious Adversaries. In: Ostrovsky, R., De Prisco, R., Visconti, I. (eds.) SCN 2008. LNCS, vol. 5229, pp. 2–20. Springer, Heidelberg (2008)
22. Malkhi, D., Nisan, N., Pinkas, B., Sella, Y.: Fairplay — a secure two-party computation system. In: Proc. USENIX Security Symposium (2004)
23. Murphy, S., Robshaw, M.J.B.: Essential Algebraic Structure within the AES. In: Yung, M. (ed.) CRYPTO 2002. LNCS, vol. 2442, pp. 1–16. Springer, Heidelberg (2002)
24. Nagle, J.: Congestion control in IP/TCP internetworks. IETF RFC 896 (1984)
25. Nielsen, J.B., Nordholt, P.S., Orlandi, C., Sheshank Burra, S.: A new approach to practical active-secure two-party computation. IACR e-print 2011/91 (2011), `http://eprint.iacr.org/2011/91`
26. Pinkas, B., Schneider, T., Smart, N.P., Williams, S.C.: Secure Two-Party Computation Is Practical. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 250–267. Springer, Heidelberg (2009)
27. Rivain, M., Prouff, E.: Provably Secure Higher-Order Masking of AES. In: Mangard, S., Standaert, F.-X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 413–427. Springer, Heidelberg (2010)
28. Yao, A.: Protocols for secure computation. In: Proc. Foundations of Computer Science – FoCS 1982, pp. 160–164. IEEE Press (1982)

A Generalized BitDecomposition

In this section, we describe a generalized variant of BitDecomposition, which includes bit-decomposition in K_8 as a subfield of K_{40}.

Let $f : V \to W$ be a linear map between two vector spaces over $\mathbb{F}_2$. Then, $\langle r \rangle$ and $\langle f(r) \rangle$ for a random element $r \in V$ allows to securely compute $\langle f(x) \rangle$ for any $\langle x \rangle$ by computing and opening $\langle x + r \rangle$, and then computing $\langle f(x) \rangle = f(x + r) + \langle f(r) \rangle$.

For bit-decomposition in K_8, define $f : K_8 \to \mathbb{F}_2^8$ by

$$f\big(\sum_{i=0}^{7} a_i \cdot X^i\big) := (a_0, \ldots, a_7).$$

This function clearly is linear over $\mathbb{F}_2$. In the offline phase, it suffices to generate $\langle (r_0, \ldots, r_7) \rangle = (\langle r_0 \rangle, \ldots, \langle r_7 \rangle)$ for random bits $(r_0, \ldots, r_7)$ because $\langle r \rangle = \sum_{i=0}^{7} \langle r_i \rangle \cdot X^i$ can be computed locally. Note that $r_0, \ldots, r_7$ are understood as elements of K_8, like all variables in the protocol over K_8. Therefore, one has to make sure that they are in fact 0 or 1 and not another element of K_8. This is done by modifying the Offline Phase; in particular each party encrypts a random bit and proves that it is actually a bit. The homomorphic structure of the NIZKPoKs makes this straight-forward. As with the triples components, the secret bit is defined as the sum of all inputs, and the secret sharing with MAC is computed by multiplication via the homomorphic property of the ciphertexts and threshold decryption.

We now move to bit-decomposition for K_8 embedded in K_{40}. Let $\imath$ denote the embedding of K_8 in K_{40}. This embedding is a field homomorphism and thus a

linear map between vector spaces over $\mathbb{F}_2$. The bit-decomposition for $\imath(K_8)$ is defined by $f : \imath(K_8) \to \mathbb{F}_2^8$,

$$f\big(\imath\big(\sum_{i=0}^{7} a_i \cdot X^i\big)\big) := (a_0, \ldots, a_7).$$

Again, f is linear over $\mathbb{F}_2$, and thus, the protocol explained above is applicable. Similarly to the case of K_8, it suffices to generate eight bits $(\langle r_0\rangle, \ldots, \langle r_7\rangle)$ in the offline phase. There is one peculiarity in this case: We defined f over $\imath(K_8) \subset K_{40}$, not K_{40}. That means, we assume that the input of f is an element of $\imath(K_8)$, not an arbitrary element. This is guaranteed in our application, but may not be true in general.

In general the function f can easily be extended to $f' : K_{40} \to \mathbb{F}_2^8$ by defining $f'(x) := f(p_{\imath(K_8)}(x))$ for $p_{\imath(K_8)}$ denoting the natural projection to $\imath(K_8)$. However, masking an arbitrary element $x \in K_{40}$ with a random element of $\imath(K_8)$ reveals $x - p_{\imath(K_8)}(x)$. Therefore, one has to mask x additionally with a random $r' \in K_{40}/\imath(K_8)$ before opening it, i.e., compute and open $\langle x + \imath(\sum_{i=0}^{7} r_i \cdot X^i) + r'\rangle$. As above, the homomorphic structure of the NIZKPoKs allow to generate $\langle r'\rangle$ with the same cost as a random element.

The above discussion re $\mathbb{F}_{2^8}$ and $\mathbb{F}_{2^{40}}$ can be extended to an arbitrary field $\mathbb{F}_{2^n}$ and a subfield $\mathbb{F}_{2^m}$ if required.

B Figures

Online Protocol

Initialize: We assume i) the parties have already invoked the Offline Phase to obtain a sufficient number of multiplication triples $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$; ii) each party holds its share of the global MAC keys $\alpha_{j,i}$; iii) that the parties have obtained (by some means) the $\langle\cdot\rangle$ sharing of the input values to the computation.

1. The parties execute $\mathsf{Init}()$ to initialize their local copy of the hash function H_1, and the values seed_i, cnt_i, $\hat{a}_{j,i}$, and $\gamma_{j,i}$.
2. The parties generate global random values $t_j \in \mathbb{F}_q$ for $j = 1, \ldots, n_{\mathsf{SAC}}$ by computing $(t_1 \| \ldots \| t_{n_{\mathsf{SAC}}}) = H_2(1 \| \mathsf{seed}_1 \| \ldots \| \mathsf{seed}_n)$.

The following steps are performed according to the circuit being evaluated.

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

Multiply: To multiply $\langle x\rangle, \langle y\rangle$ the parties do the following:

1. They take $n_{\mathsf{SAC}} + 1$ triples $(\langle a\rangle, \langle b\rangle, \langle c\rangle), (\langle f_i\rangle, \langle g_i\rangle, \langle h_i\rangle)_{i=1}^{n_{\mathsf{SAC}}}$ from the set of the available ones (and update this latter list by deleting these triples).
2. For $j = 1, \ldots, n_{\mathsf{SAC}}$ player P_i computes
   (a) $\rho_j = \mathsf{PartialOpen}(t_j \cdot \langle a\rangle - \langle f_j\rangle)$.
   (b) $\sigma_j = \mathsf{PartialOpen}(\langle b\rangle - \langle g_j\rangle)$.
   (c) $\tau_j = \mathsf{PartialOpen}(t_j \cdot \langle c\rangle - \langle h_j\rangle - \sigma_j \cdot \langle f_j\rangle - \rho_j \cdot \langle g_j\rangle - \sigma_j \cdot \rho_j)$.
   (d) If $\tau_j \neq 0$ then abort.
3. If no player has aborted the triple $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$ is accepted, and the parties execute $\epsilon = \mathsf{PartialOpen}(\langle x\rangle - \langle a\rangle)$ and $\delta = \mathsf{PartialOpen}(\langle y\rangle - \langle b\rangle)$.
4. The parties locally compute the answer $\langle z\rangle = \langle c\rangle + \epsilon \cdot \langle b\rangle + \delta \cdot \langle a\rangle + \epsilon \cdot \delta$

BitDecomposition: This produces the BitDecomposition of a shared value $\langle a\rangle$. We present a simplified protocol for when $q = 2^k$.

1. $c = \mathsf{PartialOpen}\left(\langle a\rangle + \sum_{i=0}^{k-1} \langle r_i\rangle \cdot X^i\right)$.
2. Write $c = \sum_{i=0}^{k-1} c_i \cdot X^i$.
3. Output $\langle a_i\rangle = c_i + \langle r_i\rangle$.

Output: We enter this stage when the players have $\langle y\rangle$ for the output value y, but this value has not yet been opened. This output value is only correct if players have behaved honestly, which we now need to check. Let $a_1, \ldots, a_T$ be all values publicly opened so far, where $\langle a_k\rangle = (\delta_k, (a_{k,1}, \ldots, a_{k,n}), (\gamma_{j,1}(a_k), \ldots, \gamma_{j,n}(a_k))_{j=1}^{n_{\mathsf{MAC}}})$.

1. Player P_i computes $(\mathsf{comm}_i, r_i) = \mathsf{Commit}(y_i \| (\gamma_{j,i}(y))_{j=1}^{n_{\mathsf{MAC}}})$.
2. The players execute $\{\mathsf{comm}_1, \ldots, \mathsf{comm}_n\} = \mathsf{Broadcast}(\mathsf{comm}_i)$.
3. For $j = 1, \ldots, n_{\mathsf{MAC}}$ the players execute
   (a) Player P_i computes $(\mathsf{comm}_{j,i}, r_{j,i}) \leftarrow \mathsf{Commit}(\gamma_{j,i})$.
   (b) Execute $\{\mathsf{comm}_{j,1}, \ldots, \mathsf{comm}_{j,n}\} = \mathsf{Broadcast}(\mathsf{comm}_{j,i})$.
   (c) Execute $\{\alpha_{j,1}, \ldots, \alpha_{j,n}\} = \mathsf{Broadcast}(\alpha_{j,i})$.
   (d) Player P_i computes $\alpha_j = \alpha_{j,1} + \cdots + \alpha_{j,n}$.
   (e) All players open $\mathsf{comm}_{j,i}$ to $\gamma_{j,i}$ (via a call to $\mathsf{Broadcast}$), the commitments are checked and if Open returns $\perp$ for a player then it aborts.
   (f) Each player verifies that $\alpha_j \cdot \hat{a}_{j,i} = \sum_i \gamma_{j,i}$ for his own values of $\hat{a}_{j,i}$.
4. The players execute $\mathsf{Verify}()$ to confirm all broadcasts have been valid.
5. To obtain the output value y, the commitments to $y_i, \gamma_{j,i}(y)$ are opened via each player transmitting to their openings to each player, and each player transmitting what it receives to each other to check consistency.
6. Now, y is defined as $y := \sum_i y_i$ and each player checks that $\alpha_j \cdot (y + \delta_y) = \sum_i \gamma_{j,i}(y)$, for $j = 1, \ldots, n_{\mathsf{MAC}}$.

Fig. 2. The (slightly) modified SPDZ online phase

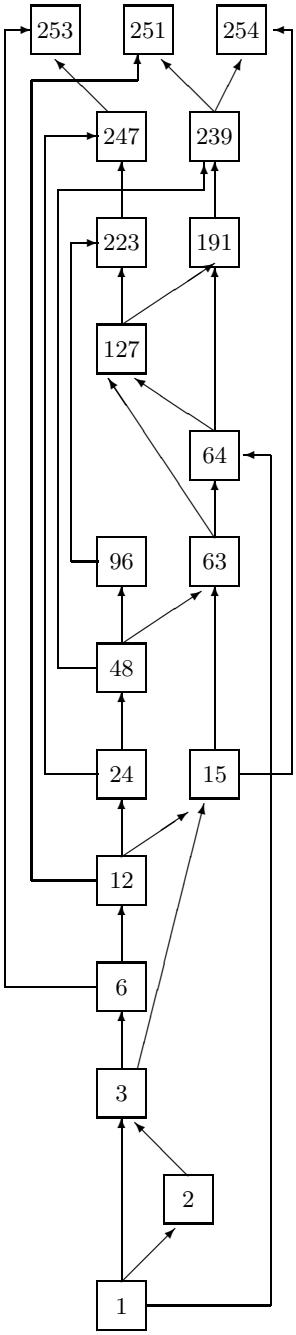

Fig. 3. Data flow graph of our addition chain