MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer

Marcel Keller* Emmanuela Orsini†
Peter Scholl‡
Department of Computer Science, University of Bristol
{m.keller,emmanuela.orsini,peter.scholl}@bristol.ac.uk

ABSTRACT

We consider the task of secure multi-party computation of *arithmetic circuits* over a finite field. Unlike Boolean circuits, arithmetic circuits allow natural computations on integers to be expressed easily and efficiently. In the strongest setting of malicious security with a dishonest majority — where any number of parties may deviate arbitrarily from the protocol — most existing protocols require expensive public-key cryptography for each multiplication in the preprocessing stage of the protocol, which leads to a high total cost.

We present a new protocol that overcomes this limitation by using *oblivious transfer* to perform secure multiplications in general finite fields with reduced communication and computation. Our protocol is based on an *arithmetic* view of oblivious transfer, with careful consistency checks and other techniques to obtain malicious security at a cost of less than 6 times that of semi-honest security. We describe a highly optimized implementation together with experimental results for up to five parties. By making extensive use of parallelism and SSE instructions, we improve upon previous runtimes for MPC over arithmetic circuits by more than 200 times.

Keywords

Multi-party computation; oblivious transfer

1. INTRODUCTION

Secure multi-party computation (MPC) allows a set of parties to jointly compute a function on their private inputs, learning only the output of the function. In the last decade, MPC has rapidly moved from purely theoretical study to an object of practical interest, with a growing interest in practical applications, and many implementations now capable of handling complex computations [28, 29].

Most MPC protocols either perform secure computation of Boolean circuits, or arithmetic circuits over a finite ring or field such as $\mathbb{F}_p$, for some prime p. Historically, the Boolean circuit approach has led to fast protocols that mostly need only symmetric cryptography, such as two-party protocols based on Yao's garbled circuits [41], or protocols based on fast oblivious transfer techniques [30, 34]. In contrast, protocols for arithmetic circuits are typically based on more expensive, public-key technology (except for special cases when a majority of the parties are honest).

Despite the need for expensive techniques, secret-sharing-based MPC protocols for arithmetic circuits have the key advantage that secure addition requires no communication and essentially come 'for free', whereas with current Boolean circuit-based 2-PC, the only 'free' operation is XOR.

The following motivating examples further highlight the practical applicability of integer-based secure computation, compared with Boolean circuits:

- Bogdanov et al. [8, 9] describe using MPC to perform secure statistical analysis of income tax records for the Estonian government. The latter work analyzed a large database with over 600000 students and 10 million tax records. The kinds of computations involved were very simple statistics, but made heavy use of the fact that secure additions are non-interactive.

- In [13], an application of MPC to *confidential benchmarking* was presented, allowing banks to jointly evaluate customers' risks whilst retaining privacy for the customers' data. They used secure linear programming, which is a highly complex task in MPC, requiring either secure floating point arithmetic or very large integer arithmetic (to emulate real numbers without overflow), both of which would be impractical using Boolean circuits.

- MPC has been suggested as a tool for helping prevent collisions between satellites, by securely performing collision detection using sensitive location and trajectory data. Kamm et al. [25] showed how to implement the relevant conjunction analysis algorithms in MPC with a protocol based on secret-sharing. This also requires secure floating point operations.

*Supported by EPSRC via grant EP/M016803.
†Supported by ERC Advanced Grant ERC-2010-AdG-267188-CRIPTO.
‡Supported by the Defense Advanced Research Projects Agency (DARPA) and Space and Naval Warfare Systems Center, Pacific (SSC Pacific) under contract No. N66001-15-C-4070.

CCS'16, October 24 - 28, 2016, Vienna, Austria

ISBN 978-1-4503-4139-4/16/10...$15.00
DOI: http://dx.doi.org/10.1145/2976749.2978357

Unfortunately, all of the above case studies are somewhat limited, in either the security properties obtained, or the efficiency. The first and third examples above used the Sharemind system [1], which is restricted to semi-honest security with three parties, where at most one is corrupt. The second example used the SPDZ MPC protocol [17], which has security against any number of maliciously corrupted parties, but is much slower. They report a fairly quick evaluation time of around 20–30 s with a prototype implementation, but this does not include the costly 'preprocessing' stage required in SPDZ, which would likely take several hours.

We conclude that although these applications are practical, the MPC protocols used still fall short: in many real-world applications, semi-honest adversaries and an honest majority are not realistic assumptions, and MPC may not be cost-effective if it requires several hours of heavy computation.

Furthermore, it is the case that *all* known practical protocols for MPC with integer operations either require an honest majority, or expensive public-key techniques for every multiplication in the circuit. For example, the SPDZ protocol [15, 17] mentioned above uses a *somewhat homomorphic encryption* scheme to perform secure multiplications, whilst the BDOZ protocol [6] uses additively homomorphic encryption, and both of these require expensive zero-knowledge proofs or cut-and-choose techniques to achieve security against malicious adversaries.

These protocols mitigate this cost to an extent by restricting the expensive computation to a *preprocessing phase*, which is independent of the inputs and can be done in advance. Although this is highly effective for reducing the *latency* of the secure computation — as the online phase is indeed very efficient — the *total cost* of these protocols can still be thousands of times greater than the online phase, which may render them ineffective for many applications.

Frederiksen et al. [19] recently showed how to efficiently use oblivious transfer to generate multiplication triples — the main task of the SPDZ preprocessing — in binary fields, and estimated much improved performance, compared with previous methods. However, this does not give the benefits of general arithmetic circuits that allow integer operations.

1.1 Our contributions

In this paper, we present MASCOT: a new MPC protocol designed to overcome the above limitations of the preprocessing phase, allowing for efficient, secure computation of general arithmetic circuits using almost exclusively fast, symmetric cryptography.

Protocol	Field	Comms. (kbit)	Throughput, $n = 2$ (/s)
SPDZ (active)	$\mathbb{F}_p$, 128-bit	$215n(n-1)$	23.5
	$\mathbb{F}_{2^{40}}$	$2272n(n-1)$	3.68
SPDZ (covert, pr. 1/10)	$\mathbb{F}_p$, 128-bit	$66n(n-1)$	204
	$\mathbb{F}_{2^{40}}$	$844n(n-1)$	31.9
Ours (active)	$\mathbb{F}_p$, 128-bit	$180n(n-1)$	**4842**
	$\mathbb{F}_{2^{128}}$	$180n(n-1)$	**4827**

Table 1: Comparing the cost of n-party secure multiplication in our OT-based protocol with previous implementations of SPDZ [14, 15].

Arithmetic-circuit MPC from OT.

We present a practical protocol for secure multi-party computation of arithmetic circuits based on oblivious transfer (OT), for the first time with malicious security in the dishonest majority setting. We achieve this by taking an "arithmetic" view of OT (as was done by Gilboa for two-party RSA key generation [20] and Demmler et al. [18] for two-party computation in the semi-honest model), which allows us to generalize the preprocessing protocol by Frederiksen et al. [19] to create multiplication triples in any (sufficiently large) finite field, instead of just binary fields. We achieve security against malicious adversaries using simple consistency checking and privacy amplification techniques, with the result that our maliciously secure protocol is only 6 times less efficient than a semi-honest version of the protocol. Moreover, our protocol can be based entirely on symmetric primitives, after a one-time setup phase, by using efficient OT extensions [23, 26].

Implementation.

A key advantage of our approach to triple generation is that we obtain a streamlined protocol, which is highly amenable to a parallelized and pipelined implementation that interleaves computation and communication. Table 1 highlights this: the time for a single secure multiplication in a prime field is 200 times faster than the previous best actively secure implementation based on somewhat homomorphic encryption [15], in spite of a fairly small improvement in communication cost. Compared with a covertly secure implementation[1] using SHE [15], our actively secure protocol requires slightly more communication, but still runs over 20 times faster. In binary fields, where SHE is much less suited, the improvement is over 1000 times, compared to previous figures [14]. Note that the online phase of our protocol is identical to that of SPDZ, which has been previously reported to achieve very practical performance for a range of applications [28].

Our optimized implementation utilizes over 80% of the network's capacity, whereas the previous schemes based on SHE are so computation-intensive that the network cannot come close to capacity. We also describe new techniques for reducing the cost of OT extension using consumer hardware instructions, namely efficient matrix transposition using SSE instead of Eklundh's algorithm, and hashing using the Matyas–Meyer–Oseas construction from any block cipher, which allows hashing 128-bit messages with AES-NI whilst avoiding a re-key for every hash.

More general assumptions.

We also improve upon the previous most practical protocol by allowing a much wider variety of cryptographic assumptions, since we only require a secure OT protocol, which can be built from DDH, quadratic residuosity or lattices [36]. In contrast, security of the SHE scheme used in SPDZ is based on the ring learning with errors assumption, which is still relatively poorly understood — it is possible that new attacks could surface that render the protocol totally impractical for secure parameters. So as well as increasing efficiency, we obtain much greater confidence in the

[1] For $\mathbb{F}_{2^{40}}$ in SPDZ with covert security, we could not find precise figures so the throughput in Table 1 is estimated based on other results.

security of our protocol, and it seems more likely to withstand the test of time.

1.2 Technical overview

The main goal of our MPC protocol is to create multiplication triples, which are essentially additive secret sharings of tuples $(a, b, a \cdot b, a \cdot \Delta, b \cdot \Delta, a \cdot b \cdot \Delta)$ where a, b are random values and Δ is a secret-shared global random MAC key. Shares of a, b and Δ can be generated by every party choosing a random share. It remains to generate secret sharings of the products.

Our starting point is the passively secure two-party product-sharing protocol of Gilboa [20], which uses k oblivious transfers to multiply two k-bit field elements. By running OT instances between every pair of parties, the multiplication triples can be created.

However, corrupted parties can deviate by providing inconsistent inputs to the different OT instances.[2] These deviations will not only lead to potentially incorrect results when the triples are used in SPDZ but also to selective failures, that is, the checks used in SPDZ might fail (or not) depending on secret information.

To obtain an actively secure protocol, we use two different strategies: one to ensure correctness of the products in the MAC generation, and one to ensure correctness and privacy of the multiplication triples themselves.

For the MAC generation, it turns out the passively secure protocol is almost enough; we just need to check random linear combinations of the MACs immediately after creation, and also when later opening values. Proving the security of this, however, is not straightforward and requires a careful, technical analysis of the possible deviations. To simplify this as much as possible, we model the MAC generation and opening requirements in a separate functionality, $\mathcal{F}_{[\![\cdot]\!]}$, which can be seen as a generalization of verifiable secret-sharing to the case of full-threshold corruption. This greatly reduces the work in proving higher-level protocols secure, as these can then be made independent of the MAC scheme and underlying MAC keys.

For triple generation, we need to ensure correctness and privacy of the triples. Correctness is easily verified with a standard sacrifice technique [16, 17], which checks a pair of triples such that one can then be used securely. To guarantee privacy we use a simple variant of privacy amplification, where first several leaky triples are produced, from which a single, random triple is extracted by taking random combinations.

In more detail, the protocol starts by generating shares of a correlated vector triple $(\mathbf{a}, b, \mathbf{c})$, where $b \in \mathbb{F}$ and $\mathbf{a}, \mathbf{c} \in \mathbb{F}^\tau$ for some constant τ, using Gilboa's multiplication protocol. If at this point the triple is checked with a sacrifice, b is guaranteed to be uniformly random, but the fact that the sacrifice passes may leak a few bits of $\mathbf{a}$, if a corrupt party used inconsistent inputs to some of the OTs. To counteract this, the parties sample a public random vector $\mathbf{r} \in \mathbb{F}^\tau$ and obtain the triple (a, b, c) by defining

$$a = \langle \mathbf{a}, \mathbf{r} \rangle, \quad c = \langle \mathbf{c}, \mathbf{r} \rangle$$

In the security proof, the simulator can precisely define any leakage and bound the min-entropy of $\mathbf{a}$ by analysing the adversary's inputs to the OTs. We then use the leftover hash lemma to show that a is uniformly random when τ is large enough.

At this point, we could repeat the process to obtain another triple, then authenticate both triples and check correctness with a sacrifice. However, we observe that this stage can be optimized by using the original vector triple $(\mathbf{a}, b, \mathbf{c})$ to obtain a second, *correlated* triple, with the same b value, at a lower cost. To do this, we simply sample another random vector $\hat{\mathbf{r}}$ and compute $\hat{a}, \hat{c}$ accordingly. Again, we can show (for suitable τ) that $\hat{a}$ is uniformly random and independent of a. We can then use $(\hat{a}, b, \hat{c})$ to check correctness of (a, b, c), as follows. After adding MACs to both triples, the parties sample a random value $s \in \mathbb{F}$ and open $\rho = s \cdot a - \hat{a}$. Now, we have:

$$s \cdot c - \hat{c} - b \cdot \rho = s \cdot (c - a \cdot b) + (\hat{a} \cdot b - \hat{c})$$

Since the left-hand side is linear in the shared values, the parties can compute this and check that it opens to zero. If one or both triples are incorrect then this is non-zero with probability at most $1/|\mathbb{F}|$, since s is uniformly random and unknown at the time of authentication.

It turns out that for this optimized method, using $\tau = 4$ suffices to give a correct triple and ensure a distinguishing advantage in $O(1/|\mathbb{F}|)$. If we allow this to be $O(1/\sqrt{|\mathbb{F}|})$ then we can have $\tau = 3$. Concretely, this means that we can use $\tau = 3$ for ≥ 128-bit fields with 64-bit statistical security.

Comparison with Previous Techniques.

Previous works have used similar privacy amplification techniques for MPC. In [16], privacy amplification was done on a large batch of triples using packed Shamir secret-sharing, which leads to high computation costs. In contrast, our protocol only requires removing leakage on *one* of the three triple values, which we do very efficiently by combining a constant-sized vector of correlated triples. In situations where leakage is possible on more than one triple component, our techniqe would have to be repeated and [16] may be more efficient, at least in terms of communication. Other works use more complex 'bucketing' techniques [35] to remove leakage in $\mathbb{F}_2$, but when working in large finite fields this is not needed.

We also note that our authentication method is similar to that of the triple generation protocol for binary fields in [19], except there, MACs are only checked after opening values, whereas we also check MACs at time of creation. That work did not describe the online phase of the resulting MPC protocol, and it turns out that for creating inputs in the online phase, this is not enough, and our additional check is crucial for security of the whole protocol.

Roadmap.

We model oblivious transfer and random oblivious transfer with $\mathcal{F}_{\mathsf{OT}}$ and $\mathcal{F}_{\mathsf{ROT}}$, respectively. The multiplication with fixed element provided by OT extension with $\mathcal{F}_{\mathsf{COPEe}}$ described in Section 3. This functionality is then used to implement $\mathcal{F}_{[\![\cdot]\!]}$ in Section 4, which guarantees the correctness of linear operations. Both $\mathcal{F}_{\mathsf{ROT}}$ and $\mathcal{F}_{[\![\cdot]\!]}$ are required to implement the triple generation functionality $\mathcal{F}_{\mathsf{Triple}}$ in Section 5, which is used for the online protocol described in the full version [27]. In Section 6, we evaluate the complexity

[2] We assume that the OT instances themselves are secure against malicious parties.

and the implementation of our protocol. Fig. 1 illustrates the relationship between our functionalities.

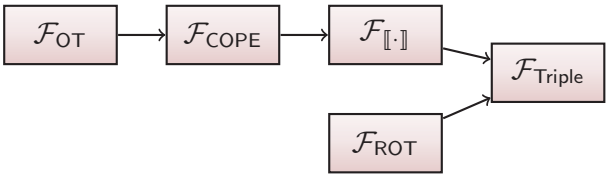

Figure 1: Dependency among functionalities

1.3 Related work

Aside from the works already mentioned, many other secure computation protocols use oblivious transfer. Protocols based on GMW [2, 21] and TinyOT [10, 30, 34] use OT extensions for efficient MPC on binary circuits, and fast garbled circuit protocols use OT extensions in the input stage of the protocol [31]. Pinkas et al. [37, 38] used OT extensions to achieve a very efficient and scalable protocol for the dedicated application of private set intersection.

Ishai et al. [24] present another protocol achieving malicious security based on OT. However, they only give asymptotic complexity measures. Furthermore, the building blocks of their protocol such as codes and fast fourier transforms suggest more expensive computation than our protocol, where the computation mainly consists of a few field operations.

Baum et al. [3] described improvements to the 'sacrifice' step and the zero-knowledge proofs used with somewhat homomorphic encryption in SPDZ. Their sacrifice technique requires generating triples that form codewords, which does not seem straightforward with our protocol. Their zero-knowledge proofs improve upon the method by Damgård et al. [15] by roughly a factor of two, but our protocol still performs much faster.

2. PRELIMINARIES

In this section, we describe the security model, introduce some important notation, define the oblivious transfer primitive, and give a basic overview of the SPDZ protocol.

Security model.

We prove our security statements in the universal composition (UC) framework of Canetti [11], and assume familiarity with this. Our protocols work with n parties from the set $\mathcal{P} = \{P_1, \ldots, P_n\}$, and we consider security against malicious, static adversaries, i.e. corruption may only take place before the protocols start, corrupting up to $n-1$ parties. When we say that a protocol Π securely implements a functionality $\mathcal{F}$ with statistical (resp. computational) security parameter κ (resp. λ), our theorems guarantee that the advantage of any environment in distinguishing the real and ideal executions is in $O(2^{-\kappa})$ (resp. $O(2^{-\lambda})$). Here, κ and λ denote the statistical and computational security parameters, respectively.

Notation.

The protocols we present in this paper work in both $\mathbb{F}_p$, for prime $p = 2^k + \mu$, and $\mathbb{F}_{2^k}$; we introduce some notation to unify the two finite fields. First note that if $k \geq \kappa$, for statistical security parameter κ, and $\mu \in \mathsf{poly}(k)$ then with overwhelming probability a random element of $\mathbb{F}_p$ can be represented with k bits in $\{0,1\}$, and likewise for any element of $\mathbb{F}_{2^k}$. Let $\mathbb{F}$ denote the finite field, which will be either $\mathbb{F}_p$ or $\mathbb{F}_{2^k}$, and write $\mathbb{F}_{2^k} \cong \mathbb{F}_2[X]/f(X)$ for some monic, irreducible polynomial $f(X)$ of degree k. We use lower case letters to denote finite field elements and bold lower case letters for vectors in $\mathbb{F}$, for any finite field $\mathbb{F}$. If $\mathbf{x}, \mathbf{y}$ are vectors over $\mathbb{F}$, then $\mathbf{x} * \mathbf{y}$ denotes the component-wise products of the vectors. We denote by $a \xleftarrow{\$} A$ the uniform sampling of a from a set A, and by $[d]$ the set of integers $\{1, \ldots, d\}$.

Following notation often used in lattice-based cryptography, define the 'gadget' vector $\mathbf{g}$ consisting of the powers of two (in $\mathbb{F}_p$) or powers of X (in $\mathbb{F}_{2^k}$), so that

$$\mathbf{g} = (1, g, g^2, \ldots, g^{k-1}) \in \mathbb{F}^k,$$

where $g = 2$ in $\mathbb{F}_p$ and $g = X$ in $\mathbb{F}_{2^k}$. Let $\mathbf{g}^{-1} : \mathbb{F} \to \{0,1\}^k$ be the 'bit decomposition' function that maps $x \in \mathbb{F}$ to a bit vector $\mathbf{x}_B = \mathbf{g}^{-1}(x) \in \{0,1\}^k$, such that $\mathbf{x}_B$ can be mapped back to $\mathbb{F}$ by taking the inner product $\langle \mathbf{g}, \mathbf{g}^{-1}(x) \rangle = x$. These basic tools allow us to easily switch between field elements and vectors of bits whilst remaining independent of the underlying finite field.

Oblivious Transfer.

Oblivious transfer (OT) is a protocol between a sender and a receiver, where the sender transmits one of several messages to the receiver, whilst remaining oblivious to which message was sent. All known constructions of OT require public-key cryptography, but in 1996, Beaver [5] introduced the concept of OT extensions, where cheap, symmetric primitives (often available in consumer hardware) are used to produce many OTs from only a few. Ishai et al. [23] later optimized this concept to the form that we will use in this paper.

Recently, Keller et al. [26] presented a simple consistency check that allows maliciously secure OT extension at essentially no extra cost: the cost for a single OT on random strings is almost that of computing two hash function evaluations and sending one string.

The ideal functionality for a single 1-out-of-2 oblivious transfer on k-bit strings is specified as follows, along with the random OT variant, where the sender's messages are sampled at random:

$$\begin{aligned} \mathcal{F}_{\mathsf{OT}}^{1,k} : ((s_0, s_1), b) &\mapsto (\perp, s_b) \\ \mathcal{F}_{\mathsf{ROT}}^{1,k} : (\perp, b) &\mapsto ((r_0, r_1), r_b), \end{aligned}$$

where $r_0, r_1 \xleftarrow{\$} \{0,1\}^k$, and $b \in \{0,1\}$ is the receiver's input. We use the notation $\mathcal{F}_{\mathsf{OT}}^{l,k}, \mathcal{F}_{\mathsf{ROT}}^{l,k}$ to denote l sets of oblivious transfers on k-bit strings.

2.1 The SPDZ Protocol

The online phase of SPDZ [15, 17] uses additive secret sharing over a finite field, combined with information-theoretic MACs to ensure active security. A secret value $x \in \mathbb{F}$ is represented by

$$[\![x]\!] = (x^{(1)}, \ldots, x^{(n)}, m^{(1)}, \ldots, m^{(n)}, \Delta^{(1)}, \ldots, \Delta^{(n)}),$$

where each party P_i holds the random share $x^{(i)}$, the random

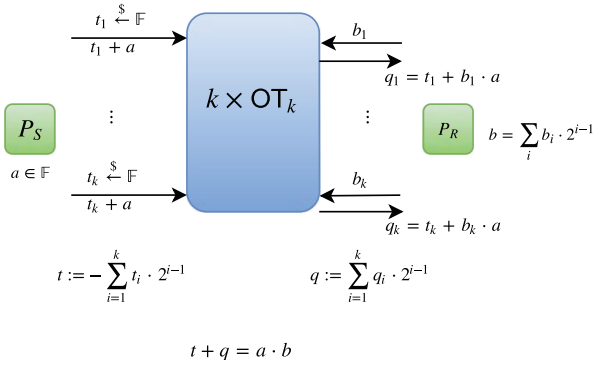

Figure 2: Two-party secret-shared multiplication in $\mathbb{F}_p$ using 1-out-of-2 OT

MAC share $m^{(i)}$ and the fixed MAC key share $\Delta^{(i)}$, such that the MAC relation $m = x \cdot \Delta$ holds, for

$$x = \sum_i x^{(i)}, \quad m = \sum_i m^{(i)}, \quad \Delta = \sum_i \Delta^{(i)}$$

over $\mathbb{F}$.

When opening a shared value $[\![x]\!]$, parties first broadcast their shares $x^{(i)}$ and compute x. To ensure that x is correct, they then check the MAC by committing to and opening $m^{(i)} - x \cdot \Delta^{(i)}$, and checking these shares sum up to zero. To increase efficiency when opening many values, a random linear combination of the MACs can be checked instead.

The main task of the SPDZ preprocessing phase is to produce the following types of random, authenticated shared values:

> **Input** P_i**:** $([\![r]\!], i)$ a random, shared value r, such that only party P_i knows the value r.
>
> **Triple:** $([\![a]\!], [\![b]\!], [\![c]\!])$ for uniformly random a, b, with $c = a \cdot b$.

In the online phase, parties interact and use the **Input** values to create shared representations of their private inputs, and the **Triple** values to perform multiplications on secret-shared values. Note that since the $[\![\cdot]\!]$ representation is linear, additions and linear functions can be computed locally.

3. CORRELATED OBLIVIOUS PRODUCT EVALUATION

In this section we describe an arithmetic generalization of the passively secure OT extension of Ishai et al. [23], which we call *correlated oblivious product evaluation* (COPE). This allows two parties to obtain an additive sharing of the product $x \cdot \Delta$, where one party holds $x \in \mathbb{F}$ and the other party holds $\Delta \in \mathbb{F}$. The correlation, Δ, is fixed at the start of the protocol, and then future iterations create sharings for different values of x.

Oblivious product evaluation.

The key mechanism behind COPE is a general method for (possibly non-correlated) oblivious product evaluation, which is illustrated for $\mathbb{F}_p$ in Fig. 2, and also used in our triple generation protocol later. The two parties run k sets of OTs on k-bit strings, where in each OT the sender, P_S, inputs a random value $t_i \xleftarrow{\$} \mathbb{F}$ and the correlated value $t_i + a$, where $a \in \mathbb{F}$ is the sender's input. The receiver inputs the bit decomposition of their input, $(b_1, \ldots, b_k) \in \{0,1\}^k$, and receives back either t_i or $t_i + a$, depending on the bit b_i. Since the sender's correlation is computed over $\mathbb{F}$, we have the relation

$$q_i = t_i + b_i \cdot a,$$

where q_i is the receiver's output in the i-th OT. Now both parties simply compute the inner product of their values $(q_i)_i, (-t_i)_i$ with the gadget vector $\mathbf{g}$ to obtain values q and t which form an additive sharing of the product of the inputs, so that

$$q + t = a \cdot b \in \mathbb{F}.$$

Correlated OPE.

To obtain COPE, where one party's input is fixed for many protocol runs, we only need to perform the k OTs once, where the receiver, P_B, inputs their bits of $\Delta \in \mathbb{F}$ and the sender, P_A, inputs k pairs of random λ-bit seeds (recall that λ is the computational security parameter and $k = \lfloor \log |\mathbb{F}| \rfloor$). This is the **Initialize** phase of Π_{COPEe} (Protocol 1).

After initialization, on each **Extend** call the parties expand the original seeds using a PRF to create k bits of fresh random OTs, with the same receiver's choice bits $\mathbf{\Delta}_B$. Party P_A now creates a correlation between the two sets of PRF outputs using their input, x (step (b)). The masked correlation is sent to P_B, who uses this to adjust the PRF output accordingly; now both parties have k correlated OTs on field elements. These are then mapped into a single field element by taking the inner product of their outputs with the gadget vector $\mathbf{g}$ to obtain an additive sharing of $x \cdot \Delta$ in steps 4–5.

Malicious behavior.

Now consider what happens in Π_{COPEe} if the parties do not follow the protocol. Party P_B fixes their input Δ at the start of the protocol, and sends no more messages thereafter, so cannot possibly cheat. On the other hand, P_A may use different values of x in each u^i that is sent in step 2 of **Extend**. Suppose a corrupt P_A uses x^i to compute u^i, for $i \in [k]$, then in step 4 we will instead have $\mathbf{q} = \mathbf{t} + \mathbf{x} * \mathbf{\Delta}_B$, where $\mathbf{x} = (x^1, \ldots, x^k)$, which then results in

$$t + q = \langle \mathbf{g} * \mathbf{x}, \mathbf{\Delta}_B \rangle$$

We do not prevent this in our protocol, but instead model this behavior in the functionality $\mathcal{F}_{\mathsf{COPEe}}$ (given in the full version [27]).

The proof of the following theorem, showing that our protocol securely implements $\mathcal{F}_{\mathsf{COPEe}}$ in the $\mathcal{F}_{\mathsf{OT}}$-hybrid model if F is a PRF, is given in the full version.

THEOREM 1. *The protocol Π_{COPEe} securely implements $\mathcal{F}_{\mathsf{COPEe}}$ in the $\mathcal{F}_{\mathsf{OT}}$-hybrid model with computational security parameter λ, if F is a PRF.*

Protocol 1 The protocol Π_{COPEe}: Oblivious correlated product evaluation with errors over the finite field $\mathbb{F}$.

The protocol uses a PRF $F : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \mathbb{F}$ and maintains a counter $j := 0$. After initialization, **Extend** may be called multiple times.

Initialize: On input $\Delta \in \mathbb{F}$ from P_B:

1: P_A samples k pairs of seeds, $\{(\mathbf{k}_0^i, \mathbf{k}_1^i)\}_{i=1}^k$, each in $\{0,1\}^\lambda$.
2: Both parties call $\mathcal{F}_{\mathsf{OT}}^{k,\lambda}$ with inputs $\{\mathbf{k}_0^i, \mathbf{k}_1^i\}_{i\in[k]}$ from P_A and $\boldsymbol{\Delta}_B = (\Delta_0, \ldots, \Delta_{k-1}) \in \{0,1\}^k$ from P_B.
3: P_B receives $\mathbf{k}_{\Delta_i}^i$ for $i \in [k]$.

Extend: On input $x \in \mathbb{F}$ from P_A:

1: For each $i = 1, \ldots, k$:
   (a) Define
   $$t_0^i = F(\mathbf{k}_0^i, j) \in \mathbb{F} \quad \text{and} \quad t_1^i = F(\mathbf{k}_1^i, j) \in \mathbb{F}$$
   so P_A knows (t_0^i, t_1^i) and P_B knows $t_{\Delta_i}^i$.
   (b) P_A sends $u^i = t_0^i - t_1^i + x$ to P_B.
   (c) P_B computes
   $$\begin{aligned} q^i &= \Delta_i \cdot u^i + t_{\Delta_i}^i \\ &= t_0^i + \Delta_i \cdot x \end{aligned}$$
2: Store $j := j + 1$
3: Let $\mathbf{q} = (q^1, \ldots, q^k)$ and $\mathbf{t} = (t_0^1, \ldots, t_0^k)$. Note that
$$\mathbf{q} = \mathbf{t} + x \cdot \boldsymbol{\Delta}_B \in \mathbb{F}^k.$$
4: P_B outputs $q = \langle \mathbf{g}, \mathbf{q} \rangle$.
5: P_A outputs $t = -\langle \mathbf{g}, \mathbf{t} \rangle$.
6: Now it holds that $t + q = x \cdot \Delta \in \mathbb{F}$.

Complexity.

The communication complexity of a single iteration of our COPE protocol, after the k base OTs in initialization, is k field elements, for a total of k^2 bits. The computation cost is $3k$ PRF evaluations and $8k$ finite field operations between the two parties.

4. AUTHENTICATING AND OPENING ADDITIVE SHARES

In this section we show how to create authenticated SPDZ shares using COPE and securely open linear combinations of these shares with a MAC checking procedure. The main challenge is to ensure that an adversary who inputs errors in our COPE protocol cannot later open an authenticated share to the incorrect value. We model these requirements in a single functionality, $\mathcal{F}_{[\![\cdot]\!]}$ (Fig. 3), which is independent of the details of the MAC scheme used and the underlying MAC keys. One can see this functionality as a generalization of verifiable secret sharing with the main difference that it allows full-threshold corruption. We first explain the mechanics of the functionality, and then describe the protocols for implementing it.

Inputs are provided to the functionality with the Input command, which takes as input a list of values $x_1, \ldots, x_l$ from one party and stores these along with the identifiers $\mathsf{id}_1, \ldots, \mathsf{id}_l$. Linear functions can then be computed on values that have been input using the LinComb command.

The Open command lets the adversary output inconsistent or incorrect values. However, if this happened to honest parties, the Check command will reveal this.

Protocol 2 MAC checking subprotocol

On input an opened value y, a MAC share $m^{(i)}$ and a MAC key share $\Delta^{(i)}$ from party P_i, each P_i does the following:

1: Compute $\sigma^{(i)} \leftarrow m^{(i)} - y \cdot \Delta^{(i)}$ and call $\mathcal{F}_{\mathsf{Comm}}$ to commit to this and receive the handle τ_i.
2: Call $\mathcal{F}_{\mathsf{Comm}}$ with (Open, τ_i) to open the commitments.
3: If $\sigma^{(1)} + \cdots + \sigma^{(n)} \neq 0$, output $\perp$ and abort, otherwise continue.

4.1 Authentication using COPE

We first consider a natural approach for one party to create an authenticated sharing of their private inputs using the correlated oblivious product evaluation protocol, and describe why this is not sufficient for active security on its own. We then show that an actively secure protocol can be obtained by authenticating one extra random value and checking a random linear combination of all MACs during the input phase. For ease of exposition, we restrict ourselves to the two-party setting, and briefly explain at the end how $\Pi_{[\![\cdot]\!]}$ (Protocol 3) extends this to n parties.

Suppose party P_1 is honest and wishes to authenticate an input $x \in \mathbb{F}$. P_1 runs an instance of $\mathcal{F}_{\mathsf{COPEe}}$ with P_2 and inputs x into the **Extend** command, whilst P_2 inputs a MAC key share $\Delta^{(2)}$. P_1 then receives t and P_j receives q such that $q + t = x \cdot \Delta^{(2)}$. P_1 then defines the MAC share $m^{(1)} = x \cdot \Delta^{(1)} + t$, and P_2 defines the MAC share $m^{(2)} = q$. Clearly, we have $m^{(1)} + m^{(2)} = x \cdot \Delta$, as required.

To convert x into shares, P_1 simply generates random additive shares $x^{(1)}, x^{(2)}$ and sends $x^{(2)}$ to P_2. Note that since the shares and MACs are linear, computing linear combinations on authenticated values is straightforward. Parties can also add a constant by adjusting their MAC shares accordingly, and choosing one party (say, P_1) to adjust their share.

Now consider a corrupt P_1^*, who can input a vector $\mathbf{x}$ into $\mathcal{F}_{\mathsf{COPEe}}$. If P_1^* chooses $\mathbf{x} = (1, \ldots, 1, 0, \ldots, 0) \in \mathbb{F}^k$, where this is 1 in the first $k/2$ positions and 0 elsewhere, then the relation between the MAC shares becomes:

$$m^{(1)} + m^{(2)} = \langle \mathbf{g} * (1, \ldots, 1, 0, \ldots, 0), \boldsymbol{\Delta}_B \rangle$$

where we have defined $m^{(1)} = t + \langle \mathbf{g} * \mathbf{x}, \boldsymbol{\Delta}^{(1)} \rangle$ for convenience.

If P_1^*'s input is later opened, then to pass the MAC check, P_1^* essentially needs to come up with a value x and a valid MAC share m^* such that $m^* + m^{(2)} = x \cdot \Delta$. One possibility is to guess the first $k/2$ bits of Δ, denoted Δ', and compute

$$m^* = m^{(1)} - \langle \mathbf{g}, \boldsymbol{\Delta}'_B \rangle$$

which gives a valid MAC relation for $x = 0$. However, P_1^* could similarly try to guess the latter $k/2$ bits of Δ, which corresponds to opening to $x = 1$. Note that each of these openings only succeeds with probability $2^{-k/2}$, but for effi-

Functionality $\mathcal{F}_{[\![\cdot]\!]}$

The functionality maintains a dictionary, Val, to keep track of the authenticated values. Entries of Val lie in the (fixed) finite field $\mathbb{F}$ and cannot be changed, for simplicity.

Input: On receiving $(\mathsf{Input}, \mathsf{id}_1, \ldots, \mathsf{id}_l, x_1, \ldots, x_l, P_j)$ from party P_j and $(\mathsf{Input}, \mathsf{id}_1, \ldots, \mathsf{id}_l, P_j)$ from all other parties, where $x_i \in \mathbb{F}$, set $\mathsf{Val}[\mathsf{id}_i] \leftarrow x_i$ for $i = 1, \ldots, l$.

Linear comb.: On receiving $(\mathsf{LinComb}, \overline{\mathsf{id}}, \mathsf{id}_1, \ldots, \mathsf{id}_t, c_1, \ldots, c_t, c)$ from all parties, where $(\mathsf{id}_1, \ldots, \mathsf{id}_t) \subseteq \mathsf{Val.keys}()$ and the combination coefficients $c_1, \ldots, c_t, c \in \mathbb{F}$, set $\mathsf{Val}[\overline{\mathsf{id}}] \leftarrow \sum_{i=1}^{t} \mathsf{Val}[\mathsf{id}_i] \cdot c_i + c$.

Open: On receiving $(\mathsf{Open}, \mathsf{id})$ from all parties, where $\mathsf{id} \in \mathsf{Val.keys}()$, send $\mathsf{Val}[\mathsf{id}]$, wait for x from the adversary, and output x to all parties.

Check: On receiving $(\mathsf{Check}, \mathsf{id}_1, \ldots, \mathsf{id}_t, x_1, \ldots, x_t)$ from every party P_i, wait for an input from the adversary. If it inputs OK, and $\mathsf{Val}[\mathsf{id}_j] = x_j$ for all j, return OK to all parties, otherwise return $\perp$ and terminate.

Abort: On receiving Abort from the adversary, send $\perp$ to all parties and terminate.

Figure 3: Functionality for authenticating, computing linear combinations of, and opening additively shared values

ciency we would like to achieve a failure probability much closer to 2^{-k}.

The main problem here is that P_1^* can choose, at the time of opening, what to open to, and is not committed to one particular value. This means the simulator cannot compute a valid input during the **Input** stage, and we cannot securely realize the functionality.

To get around this problem, we require two changes to the **Input** stage. Firstly, P_1 samples a random dummy input $x_0 \xleftarrow{\$} \mathbb{F}$, and authenticates this as well as the m actual inputs. Secondly, after computing the MACs using $\mathcal{F}_{\mathsf{COPEe}}$, P_1 opens a random linear combination of the inputs $x_0, \ldots, x_l$, and the MAC on this is checked by all parties. This ensures that P_1 is committed to their inputs during the input stage and cannot later open to a different value, whilst x_0 masks the actual inputs in this opening.

We now examine in more detail why this suffices. Suppose a corrupt P_1^* is meant to input m values to be shared, in the actual protocol $\Pi_{[\![\cdot]\!]}$. A dummy value $x_0 \in \mathbb{F}$ is sampled, and P_1^*, P_2 can obtain MAC shares such that:

$$m_h^{(1)} + m_h^{(2)} = \langle \mathbf{g} * \mathbf{x}_h, \mathbf{\Delta}_B \rangle, \quad \text{for } h = 0, \ldots, l$$

where $\mathbf{x}_h$ are P_1^*'s inputs to $\mathcal{F}_{\mathsf{COPEe}}$. In the MAC check of the **Input** stage, the parties sample a random $\mathbf{r} \in \mathbb{F}^{l+1}$, and P_1^* then opens the value y, which P_1^* can force to be any value. Next, P_2 computes during steps 8–9 the values:

$$m^{(2)} = \sum_{h=0}^{l} r_h \cdot m_h^{(2)}$$
$$\sigma^{(2)} = m^{(2)} - y \cdot \Delta.$$

P_1^* must then come up with a value $\sigma^{(1)}$ such that $\sigma^{(1)} + \sigma^{(2)} = 0$, which implies:

$$\sigma^{(1)} = -\sigma^{(2)} = y \cdot \Delta - \sum_{h=0}^{l} r_h \cdot (\langle \mathbf{g} * \mathbf{x}_h, \mathbf{\Delta}_B \rangle - m_h^{(1)})$$

$$\Leftrightarrow \sigma^{(1)} - \sum_h r_h \cdot m_h^{(1)} = y \cdot \Delta - \sum_{h=0}^{l} r_h \cdot \langle \mathbf{g} * \mathbf{x}_h, \mathbf{\Delta}_B \rangle. \quad (1)$$

Since $r_h, m_h^{(1)}$ are known to P_1^*, this is equivalent to guessing the right-hand side of (1), after choosing $\mathbf{x}_h$ (independently of r_h) and y.

Clearly, one way of achieving this is letting $\mathbf{x}_h = (x_h, \ldots, x_h)$ for some $x_h \in \mathbb{F}$, which implies that $\langle \mathbf{g} * \mathbf{x}_h, \mathbf{\Delta}_B \rangle = x_h \cdot \Delta$, and letting $y = \sum_{h=0}^{l} r_h \cdot x_h$. This corresponds to the honest behavior. Otherwise, we prove in the full version that for P_1^*, passing the check implies being able to compute a correct MAC share for x_h. Once a correct MAC share for a specific value is known, passing a later MAC check for another value implies knowledge of the MAC key.

As an example, consider the case of $\mathbf{x}_h = (0, x_h, \ldots, x_h)$ for some $x_h \neq 0$, $h \in [l]$. This implies that

$$\sum_{i=0}^{l} r_h \cdot \langle \mathbf{g} * \mathbf{x}_h, \mathbf{\Delta}_B \rangle = \sum_{h=0}^{l} r_h \cdot (x_h \cdot \Delta - x_h \cdot \Delta_1)$$
$$= \sum_{h=0}^{l} r_h \cdot x_h \cdot (\Delta - \Delta_1),$$

where Δ_1 denotes the first bit of $\mathbf{\Delta}_B$. Define $\Delta' = \Delta - \Delta_1$. Then, (1) can be written as

$$\sigma^{(1)} - \sum_{h=0}^{l} r_h \cdot m_h^{(1)} = (y - \sum_{h=0}^{l} r_h \cdot x_h) \cdot \Delta' - \sum_{h=0}^{l} r_h \cdot x_h \cdot \Delta_1.$$

If $y \neq \sum_{h=0}^{l} r_h \cdot x_h$, P_1^* has only negligible chance of passing the check. Otherwise, P_1^* can succeed with probability $1/2$ by "guessing" Δ_1. If successful, P_1^* can compute $m_h^{(1)} + x_h \cdot \Delta_1$, which is a correct MAC share for x_h because

$$m_h^{(1)} + x_h \cdot \Delta_1 + m_h^{(2)} = \langle \mathbf{g} * \mathbf{x}_h, \mathbf{\Delta}_B \rangle + x_h \cdot \Delta_1$$
$$= \langle \mathbf{g} \cdot x_h, \mathbf{\Delta}_B \rangle$$
$$= x_h \cdot \Delta.$$

This means that P_1^* is effectively committed to x_h. Finally, the simulation involves solving

$$0 = \left\langle \mathbf{g} \cdot y - \mathbf{g} * \sum_{h=0}^{l} r_h \cdot \mathbf{x}_h, \tilde{\mathbf{\Delta}}_B \right\rangle$$

Protocol 3 $\Pi_{[\![\cdot]\!]}$, creating $[\![\cdot]\!]$ elements

This protocol additively shares and authenticates inputs in $\mathbb{F}$, and allows linear operations and openings to be carried out on these shares. Note that the **Initialize** procedure only needs to be called once, to set up the MAC key.

Initialize: Each party P_i samples a MAC key share $\Delta^{(i)} \in \mathbb{F}$. Each pair of parties (P_i, P_j) (for $i \neq j$) calls $\mathcal{F}_{\mathsf{COPEe}}.\mathsf{Initialize}(\mathbb{F})$ where P_j inputs $\Delta^{(j)}$.

Input: On input $(\mathsf{Input}, \mathsf{id}_1, \ldots, \mathsf{id}_l, x_1, \ldots, x_l, P_j)$ from P_j and $(\mathsf{Input}, \mathsf{id}_1, \ldots, \mathsf{id}_l, P_j)$ from all other parties:

1: P_j samples $x_0 \xleftarrow{\$} \mathbb{F}$.
2: For $h = 0, \ldots, l$, P_j generates a random additive sharing $\sum_i x_h^{(i)} = x_h$ and sends $x_h^{(i)}$ to P_i.
3: For every $i \neq j$, P_i and P_j call $\mathcal{F}_{\mathsf{COPEe}}.\mathsf{Extend}$, where P_j inputs $(x_0, \ldots, x_l) \in \mathbb{F}^{l+1}$.
4: P_i receives $q_h^{(i,j)}$ and P_j receives $t_h^{(j,i)}$ such that

$$q_h^{(i,j)} + t_h^{(j,i)} = x_h \cdot \Delta^{(i)}, \text{ for } h = 0, \ldots, l.$$

5: Each P_i, $i \neq j$, defines the MAC shares $m_h^{(i)} = q_h^{(i,j)}$, and P_j computes the MAC shares

$$m_h^{(j)} = x_h \cdot \Delta^{(j)} + \sum_{j \neq i} t_h^{(j,i)}$$

to obtain $[\![x_h]\!]$, for $h = 0, \ldots, l$.
6: The parties sample $\mathbf{r} \leftarrow \mathcal{F}_{\mathsf{Rand}}(\mathbb{F}^{l+1})$.
7: P_j computes and broadcasts $y = \sum_{h=0}^{l} r_h \cdot x_h$.
8: Each party P_i computes $m^{(i)} = \sum_{h=0}^{l} r_h \cdot m_h^{(i)}$.
9: The parties execute Π_{MACCheck} with y and $\{m^{(i)}\}_{i \in [n]}$.
10: All parties store their shares and MAC shares under the handles $\mathsf{id}_1, \ldots, \mathsf{id}_l$.

Linear comb.:

On input $(\mathsf{LinComb}, \overline{\mathsf{id}}, \mathsf{id}_1, \ldots, \mathsf{id}_t, c_1, \ldots, c_t, c)$, the parties retrieve their shares and MAC shares $\{x_j^{(i)}, m(x_j)^{(i)}\}_{j \in [t], i \in [n]}$ corresponding to $\mathsf{id}_1, \ldots, \mathsf{id}_t$, and each P_i computes:

$$y^{(i)} = \sum_{j=1}^{t} c_j \cdot x_j^{(i)} + \begin{cases} c & i = 1 \\ 0 & i \neq 1 \end{cases}$$

$$m(y)^{(i)} = \sum_{j=1}^{t} c_j \cdot m(x_j)^{(i)} + c \cdot \Delta^{(i)},$$

They then store the new share and MAC of $[\![y]\!]$ under the handle $\overline{\mathsf{id}}$.

Open: On input $(\mathsf{Open}, \mathsf{id})$:

1: Each P_i retrieves and broadcasts their share $x^{(i)}$.
2: Parties reconstruct $x = \sum_{i=1}^{n} x^{(i)}$ and output it.

Check: On input $(\mathsf{Check}, \mathsf{id}_1, \ldots, \mathsf{id}_t, x_1, \ldots, x_t)$, the parties do the following:

1: Sample a public, random vector $\mathbf{r} \leftarrow \mathcal{F}_{\mathsf{Rand}}(\mathbb{F}^t)$.
2: Compute $y \leftarrow \sum_{j=1}^{t} r_j \cdot x_j$ and $m(y)^{(i)} \leftarrow \sum_{j=1}^{t} r_j \cdot m_{\mathsf{id}_j}^{(i)}$, where $m_{\mathsf{id}_j}^{(i)}$ denotes P_i's MAC share stored under id_j for all $i \in [n]$ and $j \in [t]$.
3: Execute Π_{MACCheck} with y and $m(y)^{(i)}$.

$$= \sum_{h=0}^{l} r_h \cdot x_h \cdot \tilde{\Delta}_1$$

for $\tilde{\Delta}$. Clearly, the first bit of any solution $\tilde{\Delta}$ must be zero. It is easy to see that

$$\tilde{\Delta}^{-1} \cdot \langle \mathbf{g} * \mathbf{x}_h, \tilde{\Delta} \rangle = x_h$$

for any such $\tilde{\Delta}$. This is how the simulator in our proof computes the value P_1^* is committed to after passing the check.

We need that, once P_1^* has passed the check in the input phase, they are committed to a particular value. However, the adversary has an edge because only a random combination of inputs can be checked (otherwise all the inputs would be revealed). This can be seen as follows: Denote by $x_{h,g}$ the g-th entry of the vector $\mathbf{x}_h$ input when authenticating the h-th value, and denote by $\{r_h\}_{h \in [l]}$ the random coefficients generated using $\mathcal{F}_{\mathsf{Rand}}$. For $g \neq g' \in [k]$, if $x_{h,g} \neq x_{h,g'}$, there is a $1/|\mathbb{F}|$ chance that $\sum r_h x_{h,g} = \sum r_h x_{h,g'}$. Because the check only relates to the randomly weighted sum, the adversary could therefore act as if $x_{h,g} = x_{h,g'}$ and decide later between $\{x_{h,g}\}_{h \in [l]}$ and $\{x_{h,g'}\}_{h \in [l]}$. The fact that there are $\log|\mathbb{F}|(\log|\mathbb{F}|-1)/2$ such pairs $g \neq g'$ explains the $2\log\log|\mathbb{F}|$ subtrahend in the theorem below. It is easy to see that a repeated check would suffice for security parameter $\log|\mathbb{F}|$.

Extension to more than two parties.

Extending the authentication protocol to n parties is relatively straightforward. When party P_j is inputing a value x, P_j runs $\mathcal{F}_{\mathsf{COPEe}}$ (on input x) with every other party $P_i \neq P_j$, who each inputs the MAC key share $\Delta^{(i)}$. Summing up these outputs allows P_j to obtain an authenticated share under the global MAC key, $\Delta = \sum_i \Delta^{(i)}$. Note that this introduces further potential avenues for cheating, as P_j may provide inconsistent x's to $\mathcal{F}_{\mathsf{COPEe}}$ with different parties, and the other parties may not use the correct $\Delta^{(i)}$. However, it is easy to see that except with probability $1/|\mathbb{F}|$, these deviations will cause the MAC check to fail in the **Input** stage, so are not a problem.

The security of our authentication and MAC checking protocols is given formally in the following theorem, which we prove in the full version [27].

THEOREM 2. *The protocol $\Pi_{[\![\cdot]\!]}$ securely implements $\mathcal{F}_{[\![\cdot]\!]}$ in the $(\mathcal{F}_{\mathsf{COPEe}}, \mathcal{F}_{\mathsf{Comm}}, \mathcal{F}_{\mathsf{Rand}})$-hybrid model, with statistical security parameter $\log|\mathbb{F}| - 2\log\log|\mathbb{F}|$.*

5. MULTIPLICATION TRIPLES USING OBLIVIOUS TRANSFER

In the previous section we showed how parties can compute linear functions on their private inputs using the authentication and MAC checking protocols. We now extend this to arbitrary functions, by showing how to create multiplication triples using $\mathcal{F}_{[\![\cdot]\!]}$ and OT.

Recall that a multiplication triple is a tuple of shared values $([\![a]\!], [\![b]\!], [\![c]\!])$ where $a, b \xleftarrow{\$} \mathbb{F}$ and $c = a \cdot b$. Given $\mathcal{F}_{[\![\cdot]\!]}$ and a protocol for preprocessing triples, the online phase of the resulting MPC protocol is straightforward, using Beaver's method for multiplying two secret-shared values [4]. For completeness, this is detailed in the full version [27].

Throughout this section, we write $[\![x]\!]$ to mean that each party holds a random, additive share of x, and the value of x is stored in the ideal functionality $\mathcal{F}_{[\![\cdot]\!]}$.

The protocol Π_{Triple} (Protocol 4) begins with the **Multiply** step, which uses $\mathcal{F}_{\mathsf{OT}}$ to compute a secret sharing of the

Protocol 4 Triple generation protocol, Π_{Triple}

The integer parameter $\tau \geq 3$ specifies the number of triples to be generated per output triple.

Multiply:

1: Each party samples $\mathbf{a}^{(i)} \xleftarrow{\$} \mathbb{F}^\tau, b^{(i)} \xleftarrow{\$} \mathbb{F}$.

2: Every ordered pair of parties (P_i, P_j) does the following:

(a) Both parties call $\mathcal{F}_{\mathsf{ROT}}^{\tau k,k}$ where P_i inputs $(a_1^{(i)}, \ldots, a_{\tau k}^{(i)}) = \mathbf{g}^{-1}(\mathbf{a}^{(i)}) \in \mathbb{F}_2^{\tau k}$.

(b) P_j receives $q_{0,h}^{(j,i)}, q_{1,h}^{(j,i)} \in \mathbb{F}$ and P_i receives $s_h^{(i,j)} = q_{a_h^{(i)},h}^{(j)}$, for $h = 1, \ldots, \tau k$.

(c) P_j sends $d_h^{(j,i)} = q_{0,h}^{(j,i)} - q_{1,h}^{(j,i)} + b^{(j)}, h \in [\tau k]$.

(d) P_i sets $t_h^{(i,j)} = s_h^{(i,j)} + a^{(i)} \cdot d_h^{(j,i)} = q_{0,h}^{(j,i)} + a_h^{(i)} \cdot b^{(j)}$, for $h = 1, \ldots, \tau k$. Set $q_h^{(j,i)} = q_{0,h}^{(j,i)}$.

(e) Split $(t_1^{(i,j)}, \ldots, t_{\tau k}^{(i,j)})$ and $(q_1^{(j,i)}, \ldots, q_{\tau k}^{(j,i)})$ into τ vectors of k components each, $(\mathbf{t}_1, \ldots, \mathbf{t}_\tau)$ and $(\mathbf{q}_1, \ldots, \mathbf{q}_\tau)$.

(f) P_i sets $\mathbf{c}_{i,j}^{(i)} = (\langle \mathbf{g}, \mathbf{t}_1 \rangle, \ldots, \langle \mathbf{g}, \mathbf{t}_\tau \rangle) \in \mathbb{F}^\tau$.

(g) P_j sets $\mathbf{c}_{i,j}^{(j)} = -(\langle \mathbf{g}, \mathbf{q}_1 \rangle, \ldots, \langle \mathbf{g}, \mathbf{q}_\tau \rangle) \in \mathbb{F}^\tau$.

(h) Now we have

$$\mathbf{c}_{i,j}^{(i)} + \mathbf{c}_{i,j}^{(j)} = \mathbf{a}^{(i)} \cdot b^{(j)} \in \mathbb{F}^\tau$$

3: Each party P_i computes:

$$\mathbf{c}^{(i)} = \mathbf{a}^{(i)} \cdot b^{(i)} + \sum_{j \neq i} (\mathbf{c}_{i,j}^{(i)} + \mathbf{c}_{j,i}^{(i)})$$

Combine:

1: Sample $\mathbf{r}, \hat{\mathbf{r}} \leftarrow \mathcal{F}_{\mathsf{Rand}}(\mathbb{F}^\tau)$.

2: Each party P_i sets

$$a^{(i)} = \langle \mathbf{a}^{(i)}, \mathbf{r} \rangle, \qquad c^{(i)} = \langle \mathbf{c}^{(i)}, \mathbf{r} \rangle \qquad \text{and}$$
$$\hat{a}^{(i)} = \langle \mathbf{a}^{(i)}, \hat{\mathbf{r}} \rangle, \qquad \hat{c}^{(i)} = \langle \mathbf{c}^{(i)}, \hat{\mathbf{r}} \rangle$$

Authenticate: Each party P_i runs $\mathcal{F}_{[\![\cdot]\!]}.\mathsf{Input}$ on their shares to obtain authenticated shares $[\![a]\!], [\![b]\!], [\![c]\!], [\![\hat{a}]\!], [\![\hat{c}]\!]$.

Sacrifice: Check correctness of the triple $([\![a]\!], [\![b]\!], [\![c]\!])$ by sacrificing $[\![\hat{a}]\!], [\![\hat{c}]\!]$.

1: Sample $s \leftarrow \mathcal{F}_{\mathsf{Rand}}(\mathbb{F})$.

2: Call $\mathcal{F}_{[\![\cdot]\!]}.\mathsf{LinComb}$ to store $s \cdot [\![a]\!] - [\![\hat{a}]\!]$ under $[\![\rho]\!]$.

3: Call $\mathcal{F}_{[\![\cdot]\!]}.\mathsf{Open}$ on input $[\![\rho]\!]$ to obtain ρ.

4: Call $\mathcal{F}_{[\![\cdot]\!]}.\mathsf{LinComb}$ to store $s \cdot [\![c]\!] - [\![\hat{c}]\!] - [\![b]\!] \cdot \rho$ under $[\![\sigma]\!]$.

5: Run $\mathcal{F}_{[\![\cdot]\!]}.\mathsf{Check}([\![\rho]\!], [\![\sigma]\!], \rho, 0)$ and abort if $\mathcal{F}_{[\![\cdot]\!]}$ aborts.

Output: $([\![a]\!], [\![b]\!], [\![c]\!])$ as a valid triple.

product of $b \in \mathbb{F}$ and $\mathbf{a} \in \mathbb{F}^\tau$, where $\tau \geq 3$ is a parameter affecting security. This is done by running τ copies of the basic two-party product sharing protocol between every pair of parties (steps (a)–(g)), followed by each party locally summing up their shares.

During this stage, a corrupt P_j may attempt to guess some bits of $\mathbf{a}$ by using values other than $b^{(j)}$ in step (c). This is why we start with τ components for $\mathbf{a}$ instead of just one, ensuring that $\mathbf{a}$ still has sufficient randomness to produce a triple. Note that there is no need for privacy amplification on b, which is already protected by the protocol because the shares $b^{(j)}$ are only used to compute $d^{(j,i)} = q_{0,h}^{(j,i)} - q_{1,h}^{(j,i)} + b^{(j)}$, which is uniformly random because P_j learns only one of $q_{0,h}^{(j,i)}$ and $q_{1,h}^{(j,i)}$.

After the **Multiply** step, the parties have an additively shared triple $(\mathbf{a}, b, \mathbf{c})$, which may be incorrect if someone was dishonest. In the **Combine** step, they take random linear combinations of the τ components of $(\mathbf{a}, b, \mathbf{c})$ using random $\mathbf{r}$ and $\hat{\mathbf{r}}$ in $\mathbb{F}^\tau$ obtained from $\mathcal{F}_{\mathsf{Rand}}$. By using two sets of random coefficients, this produces two triples with the same b component; later, one of these will be 'sacrificed' to check correctness of the other.

Using random combinations ensures that even if a few bits of the vector $\mathbf{a}$ are leaked to the adversary, the values $a, \hat{a}$ are still statistically close to uniform. The parties then use $\mathcal{F}_{[\![\cdot]\!]}$ to **Authenticate** their shares of $a, \hat{a}, b$, c and $\hat{c}$.

Finally, correctness of the triple $[\![a]\!], [\![b]\!], [\![c]\!]$ is checked in a **Sacrifice** phase, using $[\![\hat{a}]\!]$ and $[\![\hat{c}]\!]$. The idea of this step is similar to the corresponding step in previous works [15, 17], with the key difference that in our case both triples have the same b value. We observe that this still suffices to check correctness of the triples, and means we only need to authenticate 5 values instead of 6.

5.1 Security analysis

We now give some more intuition behind the security of the protocol. Let us first examine the possible adversarial deviations in the **Multiply** step.

Suppose P_j is corrupt. Let $\mathbf{a}^{(j,i)} \in \mathbb{F}^\tau$ and $\mathbf{b}^{(j,i)} \in \mathbb{F}^{\tau k}$ be the *actual* values used by P_j in the two executions of steps 1 and 3 with an honest P_i, instead of $\mathbf{a}^{(j)}$ and $b^{(j)}$. Define the values $\mathbf{a}^{(j)}$ and $b^{(j)}$ to be those values used in the instance with an arbitrary (e.g. lowest index) honest party P_{i_0}.

Then, for each $i \notin A$, let $\boldsymbol{\delta}_a^{(j,i)} = \mathbf{a}^{(j,i)} - \mathbf{a}^{(j)} \in \mathbb{F}^\tau$ and $\boldsymbol{\delta}_b^{(j,i)} = \mathbf{b}^{(j,i)} - (b^{(j)}, \ldots, b^{(j)}) \in \mathbb{F}^{\tau k}$ be the deviation in P_j's input with an honest P_i. Let $\boldsymbol{\delta}_a^{(i)} = \sum_{j \in A} \boldsymbol{\delta}_a^{(j,i)}$ and $\boldsymbol{\delta}_b^{(i)} = \sum_{j \in A} \boldsymbol{\delta}_b^{(j,i)}$, and consider $\boldsymbol{\delta}_b^{(i)}$ as a length τ vector with components in $\mathbb{F}^k$ (similarly to $\mathbf{t}_h, \mathbf{q}_h$ in the protocol).

Now by analyzing the possible adversarial deviations and summing up shares, we can see that the h-th component of $\mathbf{c}$ (for $h \in [\tau]$), at the end of the **Multiply** stage, is

$$\mathbf{c}[h] = \mathbf{a}[h] \cdot b + \underbrace{\sum_{i \notin A} \langle (\mathbf{a}^{(i)}[h])_B, \boldsymbol{\delta}_b^{(i)}[h] \rangle}_{= e_{a_h}} + \underbrace{\sum_{i \notin A} b^{(i)} \cdot \boldsymbol{\delta}_a^{(i)}[h]}_{= e_{b_h}}. \quad (2)$$

Intuitively, it is easy to see that any non-zero $\boldsymbol{\delta}_a^{(i)}$ errors will be blown up by the random honest party's share $b^{(i)}$, so should result in an incorrect triple with high probability. On the other hand, the $\boldsymbol{\delta}_b^{(i)}$ errors can be chosen so that e_{a_h} only depends on single bits of the shares $\mathbf{a}^{(i)}$. This means that a corrupt party can attempt to guess a few bits (or linear combinations of bits) of $\mathbf{a}^{(i)}$. If this guess is incorrect then the resulting triple should be incorrect; however, if all guesses succeed then the triple is correct and the sacrifice step will pass, whilst the adversary learns the bits that were guessed.

This potential leakage (or *selective failure attack*) is mitigated by the **Combine** stage. The intuition here is that, to be able to guess a single bit of the final shares $a^{(i)}, \hat{a}^{(i)}$, the adversary must have guessed *many bits* from the input vector $\mathbf{a}^{(i)}$, which is very unlikely to happen. To prove this

intuition, we analyze the distribution of the honest party's output shares using the Leftover Hash Lemma, and show that if τ is large enough, the combined output is statistically close to uniform to the adversary.

Regarding the **Sacrifice** stage, note that the check first opens $\rho = s \cdot a - \hat{a}$ and then checks that

$$s \cdot c - \hat{c} - b \cdot \rho = 0$$

which is equivalent to $s \cdot (c - a \cdot b) = \hat{c} - \hat{a} \cdot b$. If the triples are incorrect then this will only pass with probability $1/|\mathbb{F}|$, since s is random and unknown when the triples are authenticated.

The following results (proven in the full version [27]) state the security of our protocol. The first requires the combining parameter set to $\tau = 4$, to obtain a a general result for any k-bit field, whilst the second (which is evident from the proof of the theorem) shows that for k-bit fields and $k/2$-bit statistical security, $\tau = 3$ suffices.

THEOREM 3. *If $\tau = 4$ then the protocol Π_{Triple} (Protocol 4) securely implements $\mathcal{F}_{\mathsf{Triple}}$ in the $(\mathcal{F}_{\mathsf{ROT}}, \mathcal{F}_{[\![\cdot]\!]})$-hybrid model with statistical security parameter k.*

COROLLARY 1. *If $\tau = 3$ then Π_{Triple} securely implements $\mathcal{F}_{\mathsf{Triple}}$ in the $(\mathcal{F}_{\mathsf{ROT}}, \mathcal{F}_{[\![\cdot]\!]})$-hybrid model with statistical security parameter $k/2$.*

6. PERFORMANCE

We first analyse the complexity of our preprocessing protocol, and then describe our implementation and experiments.

6.1 Complexity

We measure the communication complexity of our protocol in terms of the *total* amount of data sent across the network. Note that the number of rounds of communication is constant ($\ll 100$), so is unlikely to heavily impact performance when generating large amounts of preprocessing data. Throughout this section, we exclude the cost of the λ base OTs (between every pair of parties) in the initialization stages, as this is a one-time setup cost that takes less than a second using [12].

Input tuple generation.

The main cost of authenticating one party's field element in a k-bit field with $\Pi_{[\![\cdot]\!]}$ is the $n-1$ calls to Π_{COPEe}, each of which sends k^2 bits, plus sending $n-1$ shares of k bits, for a total of $(n-1)(k^2+k)$ bits. We ignore the cost of authenticating one extra value and performing the MAC check, as this is amortized away when creating a large batch of input tuples.

Triple generation.

To generate a triple, each pair of parties makes τk calls to $\mathcal{F}_{\mathsf{ROT}}$, followed by sending a further τk^2 bits in step (c) and then 5 calls to Π_{COPEe} for authentication (ignoring $\mathcal{F}_{\mathsf{Rand}}$ and sending the input shares as these are negligible). Since each call to $\mathcal{F}_{\mathsf{ROT}}$ requires communicating λ bits, and Π_{COPEe} requires k^2 bits, this gives a total of $n(n-1)(\tau\lambda k + (\tau+5)k^2)$ bits sent across the network.

Table 2 shows these complexities for a few choices of field size, with $\lambda = 128$ and τ chosen to achieve at least 64 bit statistical security. We observe that as k increases, the cost of inputs scales almost exactly quadratically. For triples, $k = 64$ is slightly less efficient as we require $\tau = 4$ (instead of 3), whilst for larger k the cost reduces slightly as k becomes much larger than λ. Note also that the cost of an input is much lower than a triple, as the input protocol does not require any of the expensive sacrificing or combining that we use to obtain active security with triples. This is in contrast to the SPDZ protocol [15, 17], where creating input tuples requires complex zero-knowledge or cut-and-choose techniques.

Comparison with a passive protocol.

A passively secure (or semi-honest) version of our protocol can be constructed by setting $\tau = 1$ and removing the authentication step, saving 5 calls to Π_{COPEe} for every pair of parties. Note that for two parties this is essentially the same as the protocol in ABY [18]. The communication cost of a single triple is then $n(n-1)(\lambda k + k^2)$ bits. For triples where $k \geq 128$, and 64-bit statistical security, the actively secure protocol achieves $\tau = 3$, so is just 5.5 times the cost of the passive variant.

Field bit length	Input cost (kbit)	Triple cost (kbit)
64	$4.16(n-1)$	$53.25n(n-1)$
128	$16.51(n-1)$	$180.22n(n-1)$
256	$65.79(n-1)$	$622.59n(n-1)$
512	$262.66(n-1)$	$2293.76n(n-1)$

Table 2: Communication cost of our protocols for various field sizes, with n parties.

6.2 Implementation

As part of our implementation, we have used the optimizations described below. The first two apply to the OT extension by Keller et al. [26].

Bit matrix transposition.

Asharov et al. [2] mention the bit matrix transposition as the most expensive part of the computation for their OT extension. They propose Eklundh's algorithm to reduce the number of cache misses. Instead of transposing a matrix bit by bit, the matrix is transposed with respect to increasingly small blocks while leaving the blocks internally intact. Keller et al. also use this algorithm.

However, for security parameter λ, the OT extension requires the transposition of a $n \times \lambda$-matrix. We store this matrix as list of $\lambda \times \lambda$-blocks, and thus, we only have to transpose those blocks. For $\lambda = 128$, one such block is 2 KiB, which easily fits into the L1 cache of most modern processors.

Furthermore, we use the PMOVMSKB instruction from SSE2. It outputs a byte consisting of the most significant bits of 16 bytes in a 128-bit register. Together with a left shift (PSLLQ), this allows a 16×8-matrix to be transposed [33] with only 24 instructions (eight of PMOVSKB, PSLLQ, and MOV each).

Pseudorandom generator and hashing.

Keller et al. [26] used AES-128 in counter mode to implement the PRG needed for the OT extension. This allows to

use the AES-NI extension provided by modern processors. We have also implemented the hash function using AES-128 by means of the Matyas–Meyer–Oseas construction [32], which was proven secure by Black et al. [7]. This construction uses the compression function $h_i = E_{g(h_{i-1})}(m_i) \oplus m_i$, where m_i denotes the i-th message block, h_i is the state after the i-th compression, and g denotes a conversion function. In our case, the input is only one block long (as many bits as the computational security parameter of the OT extension), and g is the identity. This gives a hash function $H(m) = E_{IV}(m) \oplus m$ for some initialization vector IV, which allows to precompute the key schedule. This precomputation in turn allows to easily take advantage of the pipelining capabilities of AES-NI in modern Intel processors: While the latency of the AESENC instruction is seven clock cycles, the throughput is one per clock cycle [22]. This means that the processor is capable of computing seven encryptions in parallel.

Inner product computation.

Both Π_{COPEe} and Π_{Triple} involve the computation of $\langle \mathbf{g}, \mathbf{x} \rangle$ for $\mathbf{x} \in \mathbb{F}^{\log|\mathbb{F}|}$. Elements of both $\mathbb{F}_{2^k}$ and $\mathbb{F}_p$ are commonly represented as elements of larger rings ($\mathbb{F}_2[X]$ and $\mathbb{Z}$, respectively), and some operations involves a modular reduction (modulo an irreducible polynomial or p). When computing, we defer this reduction until after computing the sum. Furthermore, we use the `mpn_*` functions of MPIR [40] for the large integer operations for $\mathbb{F}_p$. For $\mathbb{F}_{2^k}$ on the other hand, the computation before the modular reduction is straightforward because addition in $\mathbb{F}_2[X]$ corresponds to XOR.

Multithreading.

In order to make optimal use of resources, we have organized the triple generation as follows: There are several threads independently generating triples, and every such thread controls $n-1$ threads for the OTs with the $n-1$ other players. Operations independent of OT instances, such as amplification and sacrificing, are performed by the triple generation threads. We found that performance is optimal if the number of generator threads is much larger than the number of processor cores. This is an indication that the communication is the main bottleneck.

6.3 Experiments

We have tested our implementation for up to five parties on off-the-shelf machines (eight-core i7 3.1 GHz CPU, 32 GB RAM) in a local network. Fig. 4 shows our results.

We could generate up to 4800 and 1000 $\mathbb{F}_{2^{128}}$ triples per second with two and five parties, respectively, using 100 threads. For $\mathbb{F}_p$ with p a 128-bit prime, the figures are the same. These figures come close to the maximum possible throughput of the correlation steps, which is 5500 and 1400, respectively. The maximum figures are computed from the analysis above, with $\tau = 3$ and $k = \lambda = 128$. Assuming a 1 Gbit/s link per party and unlimited routing capacity gives the desired result.

Using just a single thread, we can produce 2000 triples/s with two parties, which is still over 72 times faster than the single-threaded implementation of SPDZ [15].

By increasing the bandwidth to 2 Gbit/s, we could increase the throughput to 9500 and 1600 triples per seconds for two and five parties, respectively. This confirms the observation that the communication is the main bottleneck.

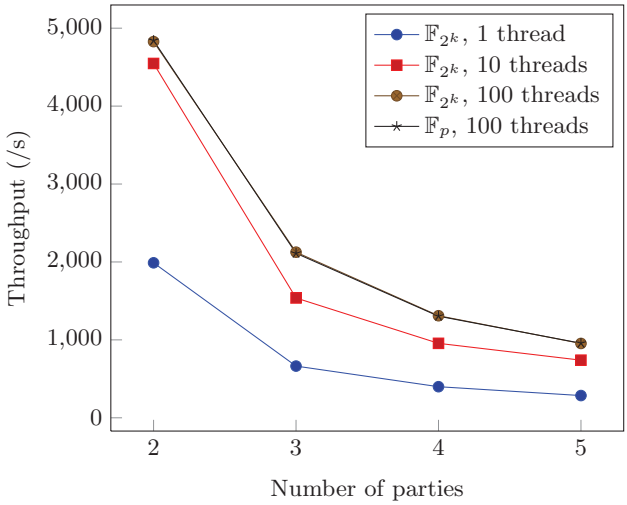

Figure 4: Triple generation throughput for 128-bit fields.

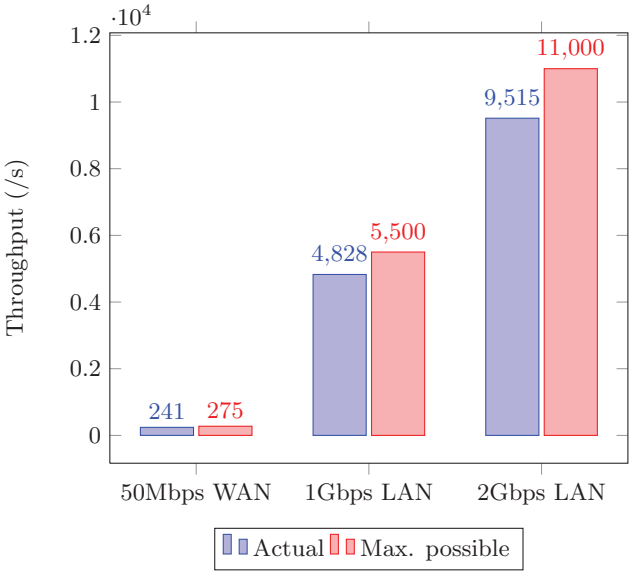

Figure 5: Throughput and maximum possible throughput for different networks with two parties

Fig. 5 shows the throughput for two parties in various network environments. The WAN environment was simulated over a LAN by restricting bandwidth to 50 Mbit/s and a round-trip latency of 100 ms.

6.3.1 Vickrey Auction

To highlight the practicality of our protocol, we have implemented the Vickrey second-price auction. Figure 6 shows the results for the offline and online phase run between two parties on a local network. This clearly illustrates the 200-fold performance improvement of our protocol, compared with (actively secure) SPDZ. Now the preprocessing phase in within 2–3 orders of magnitude of the online phase.

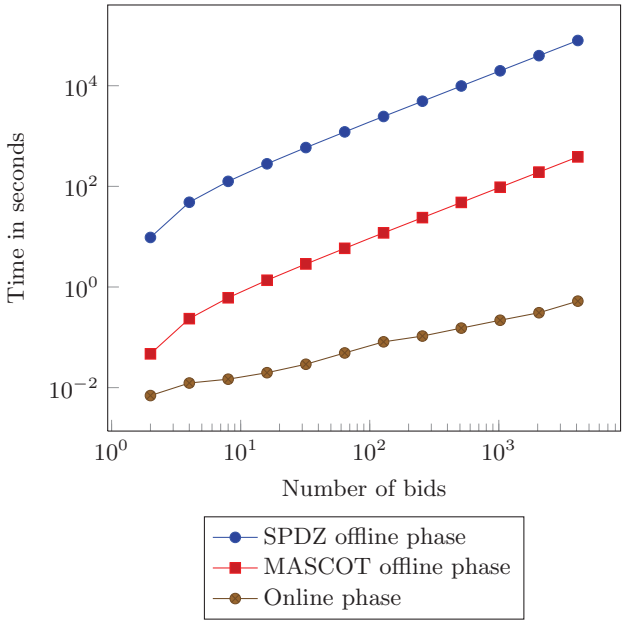

Figure 6: Vickrey auction run by two parties.

Acknowledgements

We thank Claudio Orlandi and the anonymous reviewers for valuable feedback that helped to improve the presentation.

References

[1] The Sharemind project. http://sharemind.cs.ut.ee, 2007.

[2] Asharov, G., Lindell, Y., Schneider, T., and Zohner, M. More efficient oblivious transfer and extensions for faster secure computation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 535–548.

[3] Baum, C., Damgård, I., Toft, T., and Zakarias, R. Better preprocessing for secure multiparty computation. *IACR Cryptology ePrint Archive* (2016).

[4] Beaver, D. Efficient multiparty protocols using circuit randomization. *Advances in Cryptology - CRYPTO 1991* (1992).

[5] Beaver, D. Correlated pseudorandomness and the complexity of private computations. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing* (1996), G. L. Miller, Ed., ACM, pp. 479–488.

[6] Bendlin, R., Damgård, I., Orlandi, C., and Zakarias, S. Semi-homomorphic encryption and multiparty computation. In *Advances in Cryptology - EUROCRYPT 2011* (2011), pp. 169–188.

[7] Black, J., Rogaway, P., Shrimpton, T., and Stam, M. An analysis of the blockcipher-based hash functions from PGV. *J. Cryptology 23*, 4 (2010), 519–545.

[8] Bogdanov, D., Jõemets, M., Siim, S., and Vaht, M. How the Estonian tax and customs board evaluated a tax fraud detection system based on secure multi-party computation. In *Financial Cryptography and Data Security - 19th International Conference, FC 2015, Revised Selected Papers* (2015), pp. 227–234.

[9] Bogdanov, D., Kamm, L., Kubo, B., Rebane, R., Sokk, V., and Talviste, R. Students and taxes: a privacy-preserving social study using secure computation. *IACR Cryptology ePrint Archive* (2015).

[10] Burra, S. S., Larraia, E., Nielsen, J. B., Nordholt, P. S., Orlandi, C., Orsini, E., Scholl, P., and Smart, N. P. High performance multi-party computation for binary circuits based on oblivious transfer. Cryptology ePrint Archive, Report 2015/472, 2015. http://eprint.iacr.org/.

[11] Canetti, R. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science, FOCS* (2001), pp. 136–145.

[12] Chou, T., and Orlandi, C. The simplest protocol for oblivious transfer. In *Progress in Cryptology - LATINCRYPT 2015 - 4th International Conference on Cryptology and Information Security in Latin America* (2015), pp. 40–58.

[13] Damgård, I., Damgård, K., Nielsen, K., Nordholt, P. S., and Toft, T. Confidential benchmarking based on multiparty computation. In *Financial Cryptography* (2016).

[14] Damgård, I., Keller, M., Larraia, E., Miles, C., and Smart, N. P. Implementing AES via an actively/covertly secure dishonest-majority MPC protocol. In *SCN* (2012), I. Visconti and R. D. Prisco, Eds., vol. 7485 of *Lecture Notes in Computer Science*, Springer, pp. 241–263.

[15] Damgård, I., Keller, M., Larraia, E., Pastro, V., Scholl, P., and Smart, N. P. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In *ESORICS* (2013), J. Crampton, S. Jajodia, and K. Mayes, Eds., vol. 8134 of *Lecture Notes in Computer Science*, Springer, pp. 1–18.

[16] Damgård, I., and Orlandi, C. Multiparty computation for dishonest majority: From passive to active security at low cost. In *Advances in Cryptology - CRYPTO* (2010), pp. 558–576.

[17] Damgård, I., Pastro, V., Smart, N. P., and Zakarias, S. Multiparty computation from somewhat homomorphic encryption. In *Advances in Cryptology – CRYPTO 2012* (2012), R. Safavi-Naini and R. Canetti, Eds., vol. 7417 of *Lecture Notes in Computer Science*, Springer, pp. 643–662.

[18] Demmler, D., Schneider, T., and Zohner, M. ABY - A framework for efficient mixed-protocol secure two-party computation. In *22nd Annual Network and Distributed System Security Symposium, NDSS* (2015), The Internet Society.

[19] Frederiksen, T. K., Keller, M., Orsini, E., and Scholl, P. A unified approach to MPC with preprocessing using OT. In *Advances in Cryptology – ASIACRYPT 2015, Part I* (2015), T. Iwata and J. H. Cheon, Eds., vol. 9452 of *Lecture Notes in Computer Science*, Springer, pp. 711–735.

[20] Gilboa, N. Two party RSA key generation. In *Advances in Cryptology - CRYPTO* (1999), pp. 116–129.

[21] Goldreich, O., Micali, S., and Wigderson, A. How to play any mental game or A completeness theorem for protocols with honest majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing* (1987), pp. 218–229.

[22] Intel. Intel intrinsics guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide, 2016. Online; accessed February 2016.

[23] Ishai, Y., Kilian, J., Nissim, K., and Petrank, E. Extending oblivious transfers efficiently. In *Advances in Cryptology - CRYPTO* (2003), pp. 145–161.

[24] Ishai, Y., Prabhakaran, M., and Sahai, A. Secure arithmetic computation with no honest majority. In Reingold [39], pp. 294–314.

[25] Kamm, L., and Willemson, J. Secure floating point arithmetic and private satellite collision analysis. *International Journal of Information Security 14*, 6 (2015), 531–548.

[26] Keller, M., Orsini, E., and Scholl, P. Actively secure OT extension with optimal overhead. In *Advances in Cryptology - CRYPTO 2015, Part I* (2015), pp. 724–741.

[27] Keller, M., Orsini, E., and Scholl, P. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. Cryptology ePrint Archive, Report 2016/505, 2016. http://eprint.iacr.org/2014/505.

[28] Keller, M., Scholl, P., and Smart, N. P. An architecture for practical actively secure MPC with dishonest majority. In *ACM Conference on Computer and Communications Security* (2013), pp. 549–560.

[29] Kreuter, B., Shelat, A., and Shen, C. Billion-gate secure computation with malicious adversaries. In *Proceedings of the 21st USENIX conference on Security symposium* (2012), USENIX Association, pp. 14–14.

[30] Larraia, E., Orsini, E., and Smart, N. P. Dishonest majority multi-party computation for binary circuits. In *Advances in Cryptology - CRYPTO 2014, Part II* (2014), pp. 495–512.

[31] Lindell, Y., and Riva, B. Blazing fast 2PC in the offline/online setting with security for malicious adversaries. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), pp. 579–590.

[32] Matyas, S. M., Meyer, C. H., and Oseas, J. Generating strong one-way functions with cryptographic algorithm. *IBM Technical Disclosure Bulletin 27*, 10A (1985), 5658–5659.

[33] mischasan. What is SSE !@# good for? Transposing a bit matrix. https://mischasan.wordpress.com/2011/07/24/what-is-sse-good-for-transposing-a-bit-matrix, 2011. Online; accessed February 2016.

[34] Nielsen, J. B., Nordholt, P. S., Orlandi, C., and Burra, S. S. A new approach to practical active-secure two-party computation. In *Advances in Cryptology–CRYPTO 2012*. Springer, 2012, pp. 681–700.

[35] Nielsen, J. B., and Orlandi, C. LEGO for two-party secure computation. In Reingold [39], pp. 368–386.

[36] Peikert, C., Vaikuntanathan, V., and Waters, B. A framework for efficient and composable oblivious transfer. In *Advances in Cryptology - CRYPTO* (2008), pp. 554–571.

[37] Pinkas, B., Schneider, T., Segev, G., and Zohner, M. Phasing: Private set intersection using permutation-based hashing. In *24th USENIX Security Symposium, USENIX Security 15* (2015), pp. 515–530.

[38] Pinkas, B., Schneider, T., and Zohner, M. Faster private set intersection based on OT extension. In *Proceedings of the 23rd USENIX Security Symposium* (2014), pp. 797–812.

[39] Reingold, O., Ed. *Theory of Cryptography, 6th Theory of Cryptography Conference, TCC* (2009), vol. 5444 of *Lecture Notes in Computer Science*, Springer.

[40] The MPIR team. Multiple precision integers and rationals. https://www.mpir.org, 2016. Online; accessed February 2016.

[41] Yao, A. C. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science* (1986).