New Primitives for Actively-Secure MPC over Rings with Applications to Private Machine Learning

Ivan Damgård*, Daniel Escudero*, Tore Frederiksen†, Marcel Keller‡, Peter Scholl*, Nikolaj Volgushev†
*Aarhus University , {ivan, escudero, peter.scholl}@cs.au.dk
†Alexandra Institute , {tore.frederiksen, nikolaj.volgushev}@alexandra.dk
‡Data61, CSIRO , mks.keller@gmail.com

***Abstract*—At CRYPTO 2018 Cramer *et al.* presented SPD$\mathbb{Z}_{2^k}$, a new secret-sharing based protocol for actively secure multi-party computation against a dishonest majority, that works over rings instead of fields. Their protocol uses slightly more communication than competitive schemes working over fields. However, their approach allows for arithmetic to be carried out using native 32 or 64-bit CPU operations rather than modulo a large prime. The authors thus conjectured that the increased communication would be more than made up for by the increased efficiency of implementations.**

In this work we answer their conjecture in the affirmative. We do so by implementing their scheme, and designing and implementing new efficient protocols for equality test, comparison, and truncation over rings. We further show that these operations find application in the machine learning domain, and indeed significantly outperform their field-based competitors. In particular, we implement and benchmark oblivious algorithms for decision tree and support vector machine (SVM) evaluation.

***Keywords*-MPC; SVM; Decision trees;**

I. Introduction

In the setting of secure multi-party computation, or MPC, a set of parties $P_1, \ldots, P_n$ jointly compute a function $z = f(x_1, \ldots, x_n)$, where P_i holds some input x_i, in a secure manner. On a high level this means that all parts of the computation must remain secret towards all parties, in particular each party's input must remain private, and no party is able to modify the function being computed. In recent years, many new applications have been discovered for MPC, including wide-ranging areas such as distributed key management, secure auctions, private genome analysis and data mining.

The security of an MPC protocol is formulated by the requirement that an execution of the protocol can be simulated and shown to be equivalent to execution by a trusted third party [1]. Security can then be specified according to the powers an adversary is assumed to have. One of the most important security metrics is whether we assume the adversary is *passively* or *actively* corrupted. In the passive corruption case we assume that the adversary follows the prescribed protocol (but tries to break privacy by analyzing the transcript of execution), whereas in the active corruption case the adversary may deviate arbitrarily (to possibly break both privacy and correctness). Of these, active security is the most desired, but also the hardest to achieve. In particular actively secure protocols tend to be orders of magnitude slower than semi-honestly secure protocols, although recent developments have made the gap much smaller [2], [3], [4]. Another important security metric is the number of parties an adversary is allowed to corrupt. Of particular interest is the setting where the adversary is allowed to corrupt more than half of the participating parties, known as *dishonest majority*. This include the interesting case of two-party computation, but is much harder to achieve than the case of an honest majority, for example with three parties and one corruption.

A. Computational Models in MPC

Different MPC protocols may require different representations of the function f, which can greatly affect the overhead of the protocol, compared with computing f in the clear. The most common approach is to consider f as a circuit where input, output, and internal values are from some algebraic structure and gates represent operations over this structure. A typical choice of algebraic structure is the finite field $\mathbb{F}_2$ [5], [6], [7], [8], [2] , which means that f computes over bits, addition is equivalent to XOR, and multiplication is equivalent to AND. Another popular choice is the ring $\mathbb{Z}_q$ [9], [10], [11], [4], [3] where addition and multiplication are carried out over the integers modulo some large q. Some protocols also use a binary extension field $\mathbb{F}_{2^k}$ (for a large k), where addition is equivalent to XOR but multiplication is binary polynomial multiplication, which is particularly well-suited to computing certain cryptographic functions such as AES [12], [13], [14].

Each of these have their strengths and weaknesses, for example $\mathbb{F}_2$ is best for bitwise computations such as comparison of two integers, symmetric encryptions and hash functions, while arithmetic modulo q is suitable for arithmetic operations such as computing statistics or linear programming [15]. However, in an application it will often be useful to convert between different representations, depending on the requirements at various stages of the program. For example, this has been done successfully in the ABY framework [16] and subsequent works [17], which convert between arithmetic and binary sharings for applica-

tions such as private biometric matching and classification using support vector machines, linear/logistic regression and neural networks. The downside of these approaches is that they only offer security against a passive adversary, or, in the case of [17], can only achieve active security in the restricted setting of three parties with an honest majority (that is, no collusions). MPC protocols with active security against a *dishonest majority* tend to be much more complex, and also typically only support arithmetic modulo q, where q is a large prime. This restriction makes it much more difficult to convert between $\mathbb{Z}_q$ sharings and binary sharings, making the protocols less suitable for applications where these conversions are needed.

B. The SPD$\mathbb{Z}_{2^k}$ Protocol

A recent work by Cramer *et al.* [18] took a first step in overcoming the above hurdle, with a protocol named SPD$\mathbb{Z}_{2^k}$ (after the SPDZ family of protocols [10], [19]) allowing actively secure, dishonest majority MPC over $\mathbb{Z}_q$ even when q is not a prime, for example $q = 2^k$. This gives hope that we may be able to exploit arithmetic in $\mathbb{Z}_{2^k}$ to improve the efficiency of applications and obtain similar benefits to that seen recently in the honest majority setting with the aforementioned protocols.

One of the main advantages of working modulo 2^k is that it corresponds naturally to 32/64-bit computations done in standard CPUs, allowing for very simple and efficient implementations without finite field arithmetic. Furthermore, the fact that 32 and 64-bit computation has been the norm for many years means that there are many algorithms optimized for this domain. These cannot trivially be leveraged in MPC applications working over $\mathbb{F}_p$.

Despite these advantages, we note that Cramer *et al.* [18] only described how to do additions and multiplications securely over $\mathbb{Z}_{2^k}$, which on its own is not enough to realize complex applications. This is because a large number of applications require efficient sub-routines for operations such as equality testing, comparison, and truncation, which do not give rise to efficient arithmetic circuits. Subprotocols for these tasks are well-studied when the computation is over $\mathbb{F}_p$ [20], [21], [22], but it is not immediately clear whether these techniques apply directly to the ring setting over $\mathbb{Z}_{2^k}$. In particular, many of the techniques rely on properties of fields, like the simple fact that division by 2 is possible (as long as the characteristic of the field is not 2). However, this does not work modulo 2^k since 2 is not invertible, so some workarounds are needed.

C. Contributions

In this work we present new primitives and applications for actively secure computation with a dishonest majority using arithmetic modulo 2^k. We first describe efficient protocols for conversion between binary and arithmetic sharings in $\mathbb{Z}_{2^k}$, and then leverage these to design efficient protocols for equality testing, comparison and truncation that work over the ring of integers modulo 2^k. Finally, we show how these protocols can be applied to solve problems in machine learning, namely, private classification using decision trees and support vector machines (SVMs). An overview of the different protocols considered in this work is provided in Fig. 1.

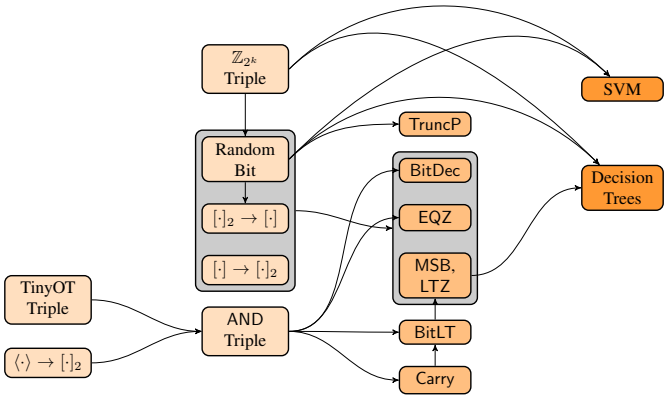


Figure 1: Relations among the protocols considered in this work. An arrow pointing from A to B means that protocol B requires protocol A.

We introduce several optimizations and implement our protocols in the FRESCO framework [23], along with the underlying SPD$\mathbb{Z}_{2^k}$ protocol of Cramer *et al.* [18]. We benchmark and compare our implementation with SPDZ, the state-of-the-art MPC protocol in the dishonest majority setting, also implemented in the FRESCO framework. Our implementation shows a speedup of 4–6x for SPD$\mathbb{Z}_{2^k}$ over SPDZ for computing multiplication, equality and comparison. We also implemented the preprocessing of SPD$\mathbb{Z}_{2^k}$, which is independent of the function to be computed, on top of Bristol-SPDZ [24]. We show this implementation to be highly competitive with the OT-based MASCOT [11] protocol in both WAN and LAN settings. Compared with the more recent Overdrive protocol [25] based on homomorphic encryption, our preprocessing comes close to meeting Overdrive's performance in a LAN setting, but is several times slower in a WAN due to the high communication costs.

To demonstrate our new building blocks, we consider the application of oblivious evaluation of decision trees and SVMs, and show that using our subprotocols for comparison, coupled with the SPD$\mathbb{Z}_{2^k}$ protocol, is around 2–5.3x faster in the online execution phase.

D. Overview of our Techniques

Both SPDZ, SPD$\mathbb{Z}_{2^k}$, and in fact many contemporary MPC protocols are cast in the *online/offline* setting. In this setting a "slow", function independent, *preprocessing* phase is first carried out to construct some raw material. When the parties know the specific function to compute, along with their respective inputs, then this raw material

is used in the *online* phase to complete the actual computation. The raw material consists of random elements, and random triples for multiplications. During the online phase the random elements can be used to obliviously give input and similarly the multiplication triples can be used to realize multiplication gates. We embrace this model in our protocols, which are typically based on random preprocessed triples, bits or random values, and we also show how to generate this preprocessing data over the appropriate ring for our binary and arithmetic protocols where this was not previously studied.

For our arithmetic-to-binary conversions, we start with the observation that an arithmetic SPD$\mathbb{Z}_{2^k}$ sharing of $x \in \mathbb{Z}_{2^k}$, denoted $[x]$, can be *locally converted* into a sharing of $x \bmod 2$, but under a different secret-sharing scheme, namely a SPD$\mathbb{Z}_{2^k}$ instance with $k = 1$. We therefore define this instance with $k = 1$ to be our secret-shared representation of binary values. This also immediately gives us a complete arithmetic-to-binary conversion, assuming we can first bit-decompose x into SPD$\mathbb{Z}_{2^k}$ shares of its bits x_i, which we turn to later. To convert the other way, from binary to arithmetic sharings, we can take a random SPD$\mathbb{Z}_{2^k}$-shared bit $[r]$, convert r to a binary share, then open $x \oplus r$ and use this to adjust $[r]$ into an arithmetic sharing of x, which can be done as a local computation. We can also perform computations on binary-shared values similarly to operations on SPD$\mathbb{Z}_{2^k}$ sharings, using multiplication triples designed for our $k = 1$ instance of SPD$\mathbb{Z}_{2^k}$ to implement AND gates.

To complete the picture, we need to be able to generate the necessary preprocessed random bits over $\mathbb{Z}_{2^k}$ and random multiplication triples over $\mathbb{Z}_2$ (the case of triples over $\mathbb{Z}_{2^k}$ was shown in [18]). Generating random bits modulo 2^k is not as simple as applying standard techniques from the field setting [19] since this relies on taking square roots modulo p, but square roots modulo a power of 2 have a more complex structure, so this cannot be directly applied. However, we show how to exploit the nature of the secret-sharing scheme in SPD$\mathbb{Z}_{2^k}$ such that it is still possible to generate random bits using one multiplication triple, as in SPDZ.

We also show that *binary* SPD$\mathbb{Z}_{2^k}$ triples, with $k = 1$, can be generated *very efficiently* by exploiting TinyOT-style protocols [7], [26] based on XOR-sharings. To do this, we give a conversion protocol which takes a batch of TinyOT-like XOR sharings and converts them to binary SPD$\mathbb{Z}_{2^k}$ sharings with almost no overhead. Since our conversion protocol guarantees that the new sharings will be of the same value, this means creating the new type of triples costs just the same as in TinyOT. This gives us a huge advantage over using native SPD$\mathbb{Z}_{2^k}$ triples, since TinyOT triples can be generated at over *250 000 triples per second*, more than 10x the throughput of our SPD$\mathbb{Z}_{2^k}$ implementation.

For our other key building blocks like secure comparison, equality and bit decomposition, we adapt existing solutions over finite fields [22] to the ring setting. Since many of these protocols have key sub-components consisting only of bit-wise operations, we can apply our conversion protocols to optimize them. We thus obtain very fast online phases for secure comparison and equality, with an online communication complexity of just $O(k)$ bits for k-bit integers. This gives up to a *85-fold reduction* compared with the online complexity of protocols used in SPDZ, which typically require sending $O(k)$ field elements per comparison or equality.

E. Related Work

Many of our subprotocols' optimizations rely on moving between computation over bits and over $\mathbb{Z}_{2^k}$. Several previous works have studied conversions between different types of secret-sharing representations for MPC, most notably the ABY framework [16], which has passively secure two-party protocols for converting between arithmetic, binary and Yao-based secret data types. Chameleon [27] extended this to a setting with an external, non-colluding third party to assist in the computation, and ABY^3 [17] extended this to a more general three-party honest majority setting, also with some support for active security. On the theoretical side, share conversion between different secret-sharing schemes was first studied by Cramer, Damgård and Ishai [28].

In the last few years there has been a lot of research in private machine learning applications using secure computation. For our applications to decision tree and SVM evaluation, the most relevant are the works by de Cock *et al.* [29], Demmler *et al.* [27] and Makri *et al.* [30]. For a more thorough survey including other machine learning applications, we refer the reader to [31], [17].

On the side of MPC primitives like comparison, there has been some other work in the setting of general, dishonest majority MPC over the ring $\mathbb{Z}_{2^k}$ [32]. Although their protocols are quite efficient asymptotically, they unfortunately have quite large hidden constants and local computation, compared to the state-of-the-art protocols working over fields [22], and in turn our protocols as well.

Even though SPD$\mathbb{Z}_{2^k}$ is the only MPC protocol we are aware of that works over the ring $\mathbb{Z}_{2^k}$ *and* is actively secure against a dishonest majority, other authors have worked on MPC protocols over $\mathbb{Z}_{2^k}$, but with less stringent security requirements. Of particular interest is Sharemind [33], as this scheme also allows mixing boolean and arithmetic operations. However, security is only in the passive, 3-party setting for an honest majority. Sharemind has also been extended to the active case [34]. Another relevant work in this area is the compiler by Damgård *et al.* [35], which can transform a passively secure protocol for t corruptions into an actively secure protocols for $\sqrt{t}$ corruptions (meaning an honest majority). Recently Araki *et al.* [36] presented a highly efficient stand-alone protocol for passive security in the honest majority setting.

F. Outline

We organize the paper as follows: In Sec. II we give background on the SPD$\mathbb{Z}_{2^k}$ protocol. Sec. III discusses the connection between this and the TinyOT [7] protocol, which only works over bits. We then show how to convert between representations of SPD$\mathbb{Z}_{2^k}$ and TinyOT elements, develop protocols for preprocessing bits for SPD$\mathbb{Z}_{2^k}$ in Sec. IV, and show how this can be used to efficiently compute equality testing, comparison, and truncation in SPD$\mathbb{Z}_{2^k}$. Sec. V shows machine learning applications that rely heavily on comparison, in particular, oblivious decision tree and SVM evaluation. We discuss our implementation of both SPD$\mathbb{Z}_{2^k}$, subprotocols and applications in Sec. VI, evaluate its performance in Sec. VII, and conclude with Sec. VIII.

II. Preliminaries

A. Notation

Given a natural number M, we denote by $\mathbb{Z}_M$ the set of integers x such that $0 \leq x \leq M-1$. We abbreviate the congruence $x \equiv y \mod 2^k$ as $x \equiv_k y$. We let $x \bmod M$ denote the remainder of x when divided by M, and we take this representative as an element of the set $\mathbb{Z}_M$. When we write $c = a \overset{?}{<} b$, we mean that c is 1 if $a < b$, and 0 otherwise.

B. Background on SPD$\mathbb{Z}_{2^k}$ Shares and Core Protocols

Our protocols build upon the secret-sharing scheme from SPD$\mathbb{Z}_{2^k}$ [18] based on additive secret-sharing with information-theoretic MACs, and its subprotocols used for computing on shares. The main idea behind this secret-sharing scheme is that, to perform a secure computation on additive shares modulo 2^k with active security, the parties will run a computation *over a larger ring* modulo 2^{k+s}, where $\sigma = s - \log(s)$ is a statistical security parameter, but *correctness is only guaranteed modulo* 2^k. The reason for this is that in a ring with many zero-divisors, traditional information-theoretic MACs cannot protect the integrity of an entire ring element $x' \in \mathbb{Z}_{2^{k+s}}$, however, they can offer integrity on the lower-order k bits, namely $x = x' \bmod 2^k$.

Given $x \in \mathbb{Z}_{2^k}$, we denote by $[x]_{2^k}$ the situation in which the parties have additive shares $x^1, \ldots, x^n, m^1, \ldots, m^n \in \mathbb{Z}_{2^{k+s}}$ and $\alpha^1, \ldots, \alpha^n \in \mathbb{Z}_{2^s}$ such that $x \equiv_k \sum_j x^j$ and $\left(\sum_j \alpha^j\right) \cdot \left(\sum_j x^j\right) \equiv_{k+s} m^j$. If there is no chance of ambiguity we use $[x]$ to denote $[x]_{2^k}$ when k is a large integer, e.g. $k = 32$ or 64.

We now summarize the core protocols for manipulating SPD$\mathbb{Z}_{2^k}$ shares, based on [18], which we use.

Input value. $[x] \leftarrow \mathsf{Input}(x, P_i)$, where $x \in \mathbb{Z}_{2^k}$. Secret-shares and authenticates a private input x from party P_i.

Linear operations. $[z] \leftarrow a[x]+[y]+b$. Any linear function or addition by a constant can be performed without interaction, resulting in a sharing of $z = ax + y + b \bmod 2^k$. The shares $z^j, t^j \in \mathbb{Z}_{2^{k+s}}$ of z and its MAC can be computes as follows. Let $x^j, m^j \in \mathbb{Z}_{2^{k+s}}$ be the shares of x and the shares of its MAC for party P_j, and let $y^j, h^j \in \mathbb{Z}_{2^{k+s}}$ be the analogous for y. Party P_1 sets $z^1 = ax^1 + y^1 + b \bmod 2^{k+s}$ and, for $j \geq 2$, party P_j sets $z^j = ax^j + y^j \bmod 2^{k+s}$. Finally, all parties P_j compute $t^j = am^j + h^j + b\alpha^j \bmod 2^{k+s}$.

Secret-shared multiplication. $[z] \leftarrow [x] \cdot [y]$. Given a secret-shared multiplication triple, that is, shares $[a], [b], [c]$ for random $a, b \in \mathbb{Z}_{2^k}$ and $c = a \cdot b \bmod 2^k$, a sharing of the product of any two sharings $[x]$ and $[y]$ can be obtained with 1 round of interaction.

Open. $x' \leftarrow \mathsf{Open}_{k'}([x], P_i)$. Opens the sharing $[x]$ modulo $2^{k'}$ towards party P_i, where $k' \leq k$, so that P_i learns only $x' := x \bmod 2^{k'}$. The MAC on $[x]$ is checked for authenticity, although sometimes when opening many values at once, the checks can be deferred and batched for greater efficiency. If k' is omitted, we assume $k' = k$. Furthermore, if the argument P_i is omitted, we assume the share is opened towards all parties.

Remark II.1. *We highlight the fact that* $\mathsf{Open}_{k'}$ *can even be used when* $k' < k$*, so that the parties can switch to a smaller modulus during the opening phase. This corresponds to only opening the lower* k' *bits of the shared value* x*.*

Security Model: The security properties of the above protocols, and all the protocols in this work, can be formalized using the *arithmetic black box model*, see for instance [32]. In this exposition we omit the formal definitions and proofs in this model. Instead we prove basic correctness and privacy properties of our protocols, but these can be extended to the formal model.

C. Preprocessing Material in SPD$\mathbb{Z}_{2^k}$

The SPD$\mathbb{Z}_{2^k}$ protocol runs in two separate phases, the *preprocessing phase*, which is independent of the parties' inputs and can be done in advance, and the *online phase*. There are several different types of random preprocessing data that are needed for different operations in the online phase. As mentioned above, we need a preprocessed multiplication triple for every secret-shared multiplication. For each input by a party P_i, we also need a preprocessed random shared mask known to P_i. Additionally, in some of our protocols we use random shared bits, which we show how to generate from a multiplication triple. The Open protocol also uses a preprocessed random mask, however, when opening and checking MACs on many values in a batch the same mask can be used for one check of all values, so we do not count this cost in our evaluation.

Multiplication triples are the most performance-intensive type of preprocessing data to generate. Random bits cost around the same as a triple; together these form the bottleneck of the preprocessing phase. The masks used for inputs

and opening are cheaper, requiring around 30x less communication than triples when using the protocols from [18].

III. Converting Between Binary and Arithmetic Sharings

A. Binary sharings

To represent a binary shared value, we simply use a standard mod 2^k sharing with $k = 1$. That is, the bit b and the MAC $\alpha \cdot b$ are both additively shared modulo 2^{s+1}, where the shares of b are only guaranteed to be of the correct value modulo 2. We denote this by $[b]_2$, in contrast with $[b]$ for an arithmetic sharing. Given two binary shared values $[a]_2$ and $[b]_2$, if the parties locally add the shares then they obtain a valid sharing of the XOR of the two bits, $a \oplus b$. Multiplication corresponds to AND, and requires a binary shared triple $[x]_2, [y]_2, [z]_2$ such that $z \equiv x \cdot y \mod 2$. We remark that, just as with SPD$\mathbb{Z}_{2^k}$ triples, it is *not* necessary for the multiplicative relation to hold modulo 2^{s+1}. So, even though the parties hold additive shares of x, y and z modulo 2^{s+1}, we may have $z \neq x \cdot y \mod 2^{s+1}$. In fact, this is exploited by our protocol for efficiently converting XOR-shared binary triples into $[\cdot]_2$-sharings.

B. Efficient binary triple generation

The online phase of the SPD$\mathbb{Z}_{2^k}$ protocol works for any k, but unfortunately its offline phase (more specifically, the sacrifice step in the triple generation protocol) requires k to be at least the security parameter. To combine SPD$\mathbb{Z}_{2^k}$ with binary operations, we need another way of generating multiplication triples. One way could be to generate triples with a large k and then reduce them to get $[\cdot]_2$ shares (as explained in Sec. III-D). However, binary triples $[x]_2, [y]_2, [z]_2$ can be generated *much more efficiently* by exploiting TinyOT-style protocols [7], [37], [26], which generate triples with *XOR-shared* MACs and shares, as we now show.

We will present a general technique for converting between two different types of sharings, which both support linear computations over $\mathbb{F}_2$. Given this conversion protocol, we can convert triples generated using TinyOT—or any other authenticated, $\mathbb{F}_2$-linear secret-sharing scheme—into a triple based on our binary share representation.

Let $\langle x \rangle$ denote that the bit x is shared and authenticated using TinyOT, that is, each party P_i holds a bit x^i and a MAC $M^j_{x_i} \in \{0,1\}^s$ on x^i, as well as a MAC key $K^i_{x_j} \in \{0,1\}^s$ for P_j's share x^j, for all $j \neq i$. Each party also has a global MAC key $\Delta^i \in \{0,1\}^s$. The shares and MACs are set up such that $x = \bigoplus_i x^i$ and $M^j_{x_i} = K^j_{x_i} \oplus x^i \Delta^j$, for all $j \neq i$. TinyOT-shared values can be XORed together locally and multiplied by 0/1 constants in the usual manner. To convert a batch of TinyOT sharings $\langle x_1 \rangle, \ldots, \langle x_m \rangle$ into $[\cdot]_2$ sharings, we use the protocol in Fig. 2. The basic idea is that, for each input x, every party will authenticate their XOR shares x^i using SPD$\mathbb{Z}_{2^k}$ to create a new binary sharing and obtain $[x]_2$. Note that even though the original shares $x^i \in \{0,1\}$ are now summed over the integers modulo 2^{s+1} to form $[x]_2$, they should still give a valid sharing of $x \bmod 2$, since the upper s bits do not matter. To verify that everyone inputs the correct shares x^i, we take a random $\mathbb{F}_2$-linear combination of all m shares, masked by an additional random share, then open this using both the TinyOT and the SPD$\mathbb{Z}_{2^k}$ sharings and check consistency. This check has soundness $1/2$, so we repeat it σ times (using σ additional random masked bits) to achieve a cheating probability of $2^{-\sigma}$.

Protocol $\Pi_{\langle x \rangle \to [x]_2}$

Input: TinyOT sharings $\langle x_1 \rangle, \ldots, \langle x_m \rangle$.
Output: Binary sharings $[x_1]_2, \ldots, [x_m]_2$.

1) Sample s additional random TinyOT-shared bits $\langle r_1 \rangle, \ldots, \langle r_\sigma \rangle$.
2) Each party P_i inputs the shares $x^i_1, \ldots, x^i_m, r^i_1, \ldots, r^i_\sigma$ with $\mathsf{Input}(\cdot, P_i)$, and then the parties sum up the shares to obtain (possibly incorrect) sharings $[x_1]_2, \ldots, [x_m]_2$ and $[r_1]_2, \ldots, [r_\sigma]_2$.
3) Sample $m \cdot s$ random bits $\chi_{i,j} \leftarrow_R \{0,1\}$, for $i = 1, \ldots, m$ and $j = 1, \ldots, \sigma$, using a coin-tossing protocol.
4) For each j, let $y_j = \mathsf{Open}_2([r_j]_2 \oplus \sum_{i=1}^m \chi_{i,j} \cdot [x_i]_2)$ and $y'_j = \Pi_{\mathsf{TinyOT.Open}}(\langle r_j \rangle_2 \oplus \sum_{i=1}^m \chi_{i,j} \cdot \langle x_i \rangle_2)$.
5) Check that $y_j = y'_j$ for all j. If not, abort.
6) Output the sharings $[x_1]_2, \ldots, [x_m]_2$.

Figure 2: TinyOT share to binary SPD$\mathbb{Z}_{2^k}$ share conversion. $\Pi_{\mathsf{TinyOT.Open}}$ denotes the TinyOT share opening protocol.

Lemma III.1. *If the inputs $\langle x_1 \rangle, \ldots, \langle x_m \rangle$ form consistent TinyOT sharings of bits $x_1, \ldots, x_m$ under uniformly random MAC keys, then the output sharings $[x_1]_2, \ldots, [x_m]_2$ form consistent SPD$\mathbb{Z}_{2^k}$ sharings with $k = 1$, except with probability at most $2^{-\sigma}$.*

Proof: Suppose the adversary causes incorrect values $x'_1, \ldots, x'_m, r'_1, \ldots, r'_\sigma$ to be authenticated in the $[\cdot]_2$ sharings, and write $x'_i = x_i + \delta_i \bmod 2$ and $r'_i = r_i + \epsilon_i \bmod 2$. If all consistency checks pass then from the j-th check we have:

$$\epsilon_j + \sum_{i=1}^m \chi_{i,j} \cdot \delta_i = 0.$$

Since each $\chi_{i,j}$ is uniformly random and independent of δ_i, ϵ_j, if any $\delta_i \neq 0$ then this holds with probability at most $1/2$ for a single j. Taking all σ checks into account, it follows that the outputs are correct with probability at least $1 - 2^{-\sigma}$. ∎

C. Arithmetic to Binary

Given a SPD$\mathbb{Z}_{2^k}$ sharing $[x]$, the parties can obtain a correct binary sharing of the least significant bit of x by simply truncating the upper $k-1$ bits of the shares and MAC shares of $[x]$. This protocol is given in Fig. 3, and it is easy to see that this gives a consistent sharing of $x \bmod 2$.

If sharings of different bits of x are required, we must first perform a bit decomposition using the techniques that will

be presented in Sec. IX-D to obtain sharings $[x_1], \ldots, [x_k]$ of the bits of x, and then run Π_{A2B} on these.

Protocol Π_{A2B}

INPUT: Arithmetic sharing $[x]$.
OUTPUT: Binary sharing $[y]_2$, where $y = x \mod 2$.

1) Let x^i, m_x^i be P_i's share and MAC share of $[x]$.
2) P_i defines $y^i = x^i \bmod 2^{s+1}$ and $m_y^i = m_x^i \bmod 2^{s+1}$ to obtain shares of $[y]_2$.

Figure 3: Arithmetic to binary SPD$\mathbb{Z}_{2^k}$ share conversion

D. Binary to Arithmetic

To convert a binary share $[x]_2$ into a SPD$\mathbb{Z}_{2^k}$ sharing, we use the protocol in Fig. 4. This uses a subprotocol Π_{RandBit} for generating a sharing $[r]_{2^k}$ of a random bit r known to none of the parties, which we show how to do in Sec. IV-A. Given this, we can locally compute $[r]_2$ using arithmetic-to-binary conversion, and then open $c = x + r \mod 2$, which perfectly hides b. Finally, using c and $[r]$ we can locally compute an arithmetic sharing of $x = c \oplus r$.

Protocol Π_{B2A}

INPUT: Binary sharing $[x]_2$.
OUTPUT: Arithmetic sharing $[x]$.

1) Let $[r] = \Pi_{\mathsf{RandBit}}()$.
2) Compute $[r]_2 = \Pi_{\mathsf{A2B}}([r])$.
3) Let $c = \mathsf{Open}_2([x]_2 + [r]_2)$.
4) Output $[x] = c + [r] - 2 \cdot c \cdot [r]$.

Figure 4: Binary to arithmetic SPD$\mathbb{Z}_{2^k}$ share conversion

IV. Basic primitives

The SPD$\mathbb{Z}_{2^k}$ protocol allows for computation modulo 2^k, and we already mentioned in the introduction the potential advantages that this scenario could bring. However, in order to being able to use the SPD$\mathbb{Z}_{2^k}$ protocol (or any MPC protocol in general) in practice, a toolkit of efficient subprotocols for basic primitives that are often used in real-life applications is needed. For instance, comparison and equality tests are an essential tool, and they are used regularly, such as in the machine learning domain.

Developing subprotocols for these tasks has been an active and fruitful line of research (e.g. [20], [31], [22], [16], [38]). Unfortunately, most of the existing solutions require the use of field arithmetic. For instance, shifting integers down, which can be seen as division by 2, is often needed when working over a field of odd characteristic. However, this seemingly simple task is already highly non-trivial over $\mathbb{Z}_{2^k}$ since 2 is non-invertible in this domain.

In this section we show how to overcome this and some other issues that appear in the $\mathbb{Z}_{2^k}$ setting and develop primitives like truncation and comparison. Our most notable contribution in this domain is the generation of shared bits $[b]$, $b \in \{0, 1\}$ for use in the arithmetic setting of SPD$\mathbb{Z}_{2^k}$.

A. Shared bits

In this protocol the parties obtain a random bit secret-shared under SPD$\mathbb{Z}_{2^k}$. Notice that the shares themselves are elements in $\mathbb{Z}_{2^{k+s}}$, whose underlying secret may be k-bits long, but the protocol guarantees that the secret is only one bit.

The concept of shared bits has been considered before in the SPDZ setting [19], and, working over a field, these bits can be obtained by making use of the property that every non-zero quadratic residue has exactly one root. Although this is not true over the ring $\mathbb{Z}_{2^k}$, something similar holds, as we now show.

Lemma IV.1. *Let $\ell > 2$. If $x \in \mathbb{Z}$ is such that $x^2 \equiv_\ell 1$, then x is congruent mod 2^ℓ to either $1, -1, -1 + 2^{\ell-1}$ or $1 + 2^{\ell-1}$.*

Proof: It is clear that $x^2 \equiv_\ell 1$ if and only if $(x-1)(x+1) \equiv_\ell 0$ so 2^ℓ divides $(x-1)(x+1)$. The case $x = \pm 1$ is trivial, so we may assume that $x \neq \pm 1$. Let 2^u and 2^v be the largest power of 2 dividing the non-zero integers $x-1$ and $x+1$ respectively, then 2^{u+v} is the largest power of 2 dividing $x^2 - 1$, so $u + v \geq \ell$. On the other hand, since $2^{\min(u,v)}$ divides both $x+1$ and $x-1$, it also divides $(x+1) - (x-1) = 2$ and therefore $\min(u, v) \in \{0, 1\}$. Thus, either $u = 0$, $u = 1$, $v = 0$ or $v = 1$. If $u = 0$ then $v \geq \ell$ and therefore $x + 1 \equiv_\ell 0$. If $u = 1$ then $v \geq \ell - 1$, so $x + 1 \equiv_{\ell-1} 0$. A similar analysis follows if $v = 0$ or $v = 1$, which finishes the proof. ∎

Our protocol is described in Fig. 5. At the end of the execution the parties will get shares $b^1, \ldots, b^n \in \mathbb{Z}_{2^{k+s}}$ and $t^1, \ldots, t^n \in \mathbb{Z}_{2^{k+s}}$ such that $b^1 + \cdots + b^n \equiv_k b$ and $h^1 + \cdots + h^n \equiv_{k+s} \alpha \cdot (\sum_i b^i)$, where b is random in $\{0, 1\}$.

Protocol Π_{RandBit}

OUTPUT: Random shared bit $[b]$ over $\mathbb{Z}_{2^k}$ with MAC shares over $\mathbb{Z}_{2^{k+s}}$.

In the following the parties use an instance of SPD$\mathbb{Z}_{2^k}$ over $\mathbb{Z}_{2^{k+2}}$ with MAC shares over $\mathbb{Z}_{2^{k+s+1}}$.

1) P_i samples $u^i \leftarrow_R \mathbb{Z}_{2^{k+2}}$.
2) Each party inputs their share u^i to to obtain a sharing of a random element $[u]$, where $u = u^1 + \cdots + u^n \bmod 2^{k+2}$.
3) Compute $[a] \leftarrow 2[u] + 1$.
4) Compute $[a^2] \leftarrow [a] \cdot [a]$.
5) Let $e = \mathsf{Open}([a^2])$. If a is not odd then the parties abort.
6) Let c be the smallest root modulo 2^{k+2} of e and let c^{-1} be its inverse modulo 2^{k+2}. Compute $[d] \leftarrow c^{-1}[a] + 1$.
7) Let $d^j, t^j \in \mathbb{Z}_{k+s+1}$ be P_j's shares of d and of its MAC, respectively. Each P_j sets $b^j = \frac{d^j}{2}$ and $h^j = \frac{t^j}{2}$.
8) Output $[b]$, defined by the shares b^j and the MAC shares h^j above.

Figure 5: Protocol for obtaining authenticated shared bits

Proposition IV.1. *If the protocol Π_{RandBit} does not abort, then its output is a random shared bit $[b]$.*

Proof: We begin by arguing correctness. In step 5 the value $e = a^2 \bmod 2^{k+2}$ is opened. Then, notice that $(c^{-1}a)^2 \equiv_{k+2} c^{-2}a^2 \equiv_{k+2} e^{-1}e \equiv_{k+2} 1$, so $c^{-1}a \equiv_{k+1} \pm 1$ due to Lemma IV.1. Moreover, since c is taken to be the smallest square root of e and a is one of such roots, which was chosen at random, we conclude that $c^{-1}a$ is congruent modulo 2^{k+1} to either -1 or $+1$ with equal probability. Clearly, this implies that $c^{-1}a+1$ is congruent modulo 2^{k+1} to either 0 or 2 with equal probability.

Now, the key point to observe is that, due to the fact that $[a]$ was computed as $2[u]+1$, and due to the way the addition of shared values in SPD$\mathbb{Z}_{2^k}$ works (see Section II-B), we have that both the shares of $[a]$ and the shares of its MAC corresponding to party P_j are even for $j > 1$, and odd for $j = 1$. Then, since c^{-1} is odd it follows by a similar argument that $d^j, t^j \in \mathbb{Z}_{2^{k+s+1}}$, the shares of the value $[c^{-1}a + 1]$ and its MAC, are all even, so the division by 2 used in the protocol is well defined. Notice that $\sum_j d^j \equiv_{k+2} c^{-1}a+1$ and $\sum_j t^j \equiv_{k+s+1} \alpha \cdot \left(\sum_j d^j\right)$, so $\sum_j \frac{d^j}{2} \equiv_{k+1} \frac{c^{-1}a+1}{2}$ and $\sum_j \frac{t^j}{2} \equiv_{k+s} \alpha \cdot \left(\sum_j \frac{d^j}{2}\right)$. In particular, $\sum_j b^j$ is congruent modulo 2^k to 0 or 1 with equal probability, so b^j, h^j indeed define authenticated shares of a bit over $\mathbb{Z}_{2^k}$, with shares over $\mathbb{Z}_{2^{k+s}}$, as desired.

As for security, notice that the only possibility of an attack is that the adversary causes a selective abort, therefore biasing the resulting bit. Fortunately this is not possible since, right before step 5, the value a looks uniformly random to the adversary. ∎

B. Extraction of most significant bit

Here the parties have a shared value $[a]$, with $a = \sum_{i=0}^{k-1} a_i 2^i \in \mathbb{Z}_{2^k}$, and they wish to compute shares of the most significant bit (MSB) of a, $[a_{k-1}]$. This is achieved by masking $[a]$ with a random value $[r]$ where the bits are shared individually (using shared random bits) and then opening $c = a+r \bmod 2^k$. Since $a = (c-r) \bmod 2^k$, shares of $a \bmod 2^{k-1}$ can be obtained together from $c \bmod 2^{k-1}$ and the shares of $r \bmod 2^{k-1}$ (obtained from the shares of the bits of r), using a bitwise comparison procedure Π_{BitLT} to account for the sign of the difference $(c \bmod 2^{k-1}) - (r \bmod 2^{k-1})$. Finally, $2^{k-1}a_{k-1}$ is computed as $a - (a \bmod 2^{k-1})$, and the factor 2^{k-1} is removed by masking the k-th bit of $[2^{k-1}a_{k-1}]$ with a shared random bit, opening this result, truncating the lower $k-1$ bits (which are all zero) and removing the mask. For details, see the protocol in Fig. 6.

Note that the main online cost is two openings of k-bit ring elements, and the Π_{BitLT} subprotocol (Sec. IX-E) on length $k-1$ inputs, which has $2k-4$ bit multiplications in $\log(k-1)$ rounds. This gives a total communication complexity of $6k-8$ bits per party in $\log(k-1)+2$ rounds.

The idea of extracting most significant bits by first subtracting the lower bits and then truncating is already present in [22], as well as the idea of extracting lower bits (obtaining $a \bmod 2^{k-1}$ in our case) using bit-decomposed masks. The truncation step is trivial when working in a field, but does not extend to $\mathbb{Z}_{2^k}$, which is why at the end of the protocol we mask by the random bit $[b]$ and shift down the result $[d]$.

Protocol Π_{MSB}

INPUT: Shared value $[a]$.
OUTPUT: Shared value $[a_{k-1}]$, where $a = \sum_{i=0}^{k-1} a_i 2^i \in \mathbb{Z}_{2^k}$.

1) Call $[b], [r_0], \ldots, [r_{k-1}] \leftarrow \Pi_{\mathsf{RandBit}}()$ and compute $[r] = [\sum_{i=0}^{k-1} r_i 2^i]$.
2) Let $c \leftarrow \mathsf{Open}([a] + [r])$.
3) Compute $c' = c \bmod 2^{k-1}$ and $[r'] \leftarrow \sum_{i=0}^{k-2} 2^i [r_i]$.
4) Call $[r_0]_2, \ldots, [r_{k-2}]_2 \leftarrow \Pi_{\mathsf{A2B}}([r_0], \ldots, [r_{k-2}])$.
5) Let $[u]_2 \leftarrow \Pi_{\mathsf{BitLT}}(c', [r_0]_2, \ldots, [r_{k-2}]_2)$.[a]
6) Call $[u] \leftarrow \Pi_{\mathsf{B2A}}([u]_2)$.[b]
7) Compute $[a'] \leftarrow c' - [r'] + 2^{k-1}[u]$ and $[d] \leftarrow [a] - [a']$.
8) Let $e \leftarrow \mathsf{Open}([d] + 2^{k-1}[b])$, and let e_{k-1} be the most significant bit of e.
9) Output $e_{k-1} + [b] - 2e_{k-1}[b]$.

[a] When one of the inputs is public, Π_{BitLT} operates in the same way as if both inputs were shared but using the bits of the public input in the clear.
[b] We can avoid the share-conversion by noticing that $2^{k-1}[u]_2 = [2^{k-1}u]$.

Figure 6: Protocol for extracting most significant bit

Proposition IV.2 (Informal). *Protocol Π_{MSB} correctly computes $[a_{k-1}]$ from $[a]$, where $a = \sum_{i=0}^{k-1} a_i 2^i$. Moreover, it does not reveal any information about a.*

Proof: To argue correctness, we begin by showing that $a' = a \bmod 2^{k-1}$. To see this, notice that $a \equiv_{k-1} c' - r'$, and that $c' - r' \in \{-2^{k-1}, \ldots, 2^{k-1}-1\}$. Therefore, $c' - r'$ is the remainder of a when divided by 2^{k-1} if and only if $c' - r' \geq 0$, and otherwise it is equal to this remainde, minus 2^{k-1}. This can be written in a more compact way as $c' - r' = (a \bmod 2^{k-1}) - u2^{k-1}$, where $u = c' \overset{?}{<} r'$, which implies that $a \bmod 2^{k-1} = c' - r' + u2^{k-1} = a'$. We can see then that $d = a - a' = 2^{k-1}a_{k-1}$, so $e = 2^{k-1}(a_{k-1} \oplus b)$ and therefore $e_{k-1} = a_{k-1} \oplus b$. Thus, a_{k-1} can be computed as $a_{k-1} = e_{k-1} \oplus b = e_{k-1} + b - 2e_{k-1}b$.

Finally, to argue security we show that none of the opened values reveal anything about a. The value c does not leak information since the random mask r is used. On the other hand, we saw above that $d = 2^{k-1}a_{k-1}$, so the mask $2^{k-1}b$ completely hides this value when e is opened. ∎

C. Comparison of Signed Integers

In many applications it makes sense to assume that the underlying data are signed, meaning that it can be negative, positive, or zero. We can represent this using integers modulo 2^k, by taking the class representatives in the interval $[-2^{k-1}, 2^{k-1})$. For computational purposes this is the same as our set $\mathbb{Z}_{2^k}$ of unsigned values, but we can add some

additional interpretation to the numbers in $[-2^{k-1}, 2^{k-1})$ (namely, the sign) that is useful in many applications.

Every integer in $a \in [-2^{k-1}, 2^{k-1})$ can be written as $a = -a_{k-1}2^{k-1} + \sum_{i=0}^{k-2} a_i 2^i$ (this is the so-called two's complement representation), and its corresponding representative in $\mathbb{Z}_{2^k}$ is $a \bmod 2^k = \sum_{i=0}^{k-1} a_i 2^i$. It is easy to see that in this setting, $a \in [-2^{k-1}, 2^{k-1})$ is negative if and only if $a_{k-1} = 1$, so, as in [22], we define the comparison-with-zero operator for a shared value as $\Pi_{\mathsf{LTZ}}([a]) := \Pi_{\mathsf{MSB}}([a])$.

Now, consider $a, b \in [-2^{k-2}, 2^{k-2})$. Clearly, $-2^{k-1} \leq a - b < 2^{k-1}$ so we can determine $u = a \overset{?}{<} b$ (comparison as signed integers) by $u = \Pi_{\mathsf{LTZ}}([a] - [b])$. Therefore, as done in [22], we define comparison of two shared values as $\Pi_{\mathsf{LT}}([a], [b]) = \Pi_{\mathsf{LTZ}}([a] - [b])$.

Finally, notice that we restricted $a, b \in [-2^{k-2}, 2^{k-2})$. This is because if $a, b \in [-2^{k-1}, 2^{k-1})$, then correctness may not hold if $a - b$ overflows. For instance, if $a = -2^{k-1}$ and $b = 1$, then $a < b$ but the most significant bit of $(a - b) \bmod 2^k = \sum_{i=0}^{k-2} 2^i$ is 0, so, in other words, $a - b$ is treated as positive even though it is not. If numbers in $[-2^{k-1}, 2^{k-1})$ must be compared, this can be done at the cost of roughly three calls to Π_{MSB}. Intuitively, this is because subtracting $b - a$ and comparing against zero is only guaranteed to work if a and b have the same sign and, if this is not the case, the sign of a dictates the value of $a \overset{?}{<} b$. Therefore, besides extracting the most significant bit of the difference $b - a$, we also check if a and b have the same sign and choose the right output depending on the case. This is done by extracting the most significant bits of both a and b, which incurs in the two additional calls to Π_{MSB}.

The protocol works, in detail, as follows.

- Let $[a_{k-1}] \leftarrow \Pi_{\mathsf{MSB}}([a])$ and $[b_{k-1}] \leftarrow \Pi_{\mathsf{MSB}}([b])$.
- Compute $[h] \leftarrow [a_{k-1}] + [b_{k-1}] - 2[a_{k-1}][b_{k-1}]$.
- Let $[e] \leftarrow \Pi_{\mathsf{MSB}}([a] - [b])$.
- Output $[d] \leftarrow [h] \cdot [a_{k-1}] + [1 - h] \cdot [e]$.

We argue that this protocol produces the right output. The main observation is that if a and b have the same sign then extracting the most significant bit of $a - b$ will yield the correct bit $a \overset{?}{<} b$. Now, if a and b have different sign then the result is simply the most significant bit of a. Finally, observe that $h = a_{k-1} \oplus b_{k-1}$ and that a and b have the same most significant bit if and only if $h = 0$. This concludes the argument.

D. Equality Test

We introduce a protocol for computing securely $a \overset{?}{=} 0$, where $a \in [-2^{k-1}, 2^{k-1} - 1]$. The protocol can be found in Fig. 7, and it is almost the same as Protocol 3.7 in [22]. However, one relevant change with respect to the original protocol is that, instead of performing step 5 directly on arithmetic shares, we convert these first to binary and then calculate the OR circuit using binary shares, which involve less communication than arithmetic shares. The result is then converted from binary to arithmetic.

Protocol Π_{EQZ}

INPUT: Shared value $[a]$.

OUTPUT: Shared value $[b]$ where $b = (a \overset{?}{=} 0)$, regarding a to be signed (i.e. $a \in [-2^{k-1}, 2^{k-1})$).

1) Call $[r_0], \ldots, [r_{k-1}] \leftarrow \Pi_{\mathsf{RandBit}}()$.
2) Let $[r] \leftarrow -2^{k-1}[r_{k-1}] + \sum_{i=0}^{k-2} 2^i [r_i]$.
3) Let $c \leftarrow \mathsf{Open}([a] + [r])$. Let $(c_0, \ldots, c_{k-1})$ be the binary representation of c.
4) Call $[r_0]_2, \ldots, [r_{k-1}]_2 \leftarrow \Pi_{\mathsf{A2B}}([r_0], \ldots, [r_{k-1}])$.
5) Let $[b]_2 \leftarrow 1 - \bigvee_{i=0}^{k-1} (c_i + [r_i]_2 - 2c_i[r_i]_2)$.
6) Output $[b] \leftarrow \Pi_{\mathsf{B2A}}([b]_2)$.

Figure 7: Protocol for testing equality of a shared value with 0.

To argue correctness, we notice that $a = 0$ if and only if $a + r \equiv_k r$ for some $r \in [-2^{k-1}, 2^{k-1} - 1]$. This is what the protocol does, since it returns 1 if and only if all the bits of c coincide with those of r. For security, we notice that the opened value c is uniformly distributed, so no information is leaked.

We remark that the computation of the OR in the last step can be done in $O(\log k)$ rounds, as shown in [22].

V. Applications

In this section we discuss some applications leveraging our efficient comparison protocol.

A. Decision Trees

We consider the machine-learning application of decision trees which is used for classification. A *decision tree* is a function $\mathcal{T} : \mathbb{R}^n \to \mathbb{Z}_q$, where n is called the dimension of the *feature space* and q is the amount of possible output categories. The input $\boldsymbol{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$ to $\mathcal{T}$ is called the *feature vector*. The function $\mathcal{T}$ is implemented as a binary tree with m internal nodes, where each internal node v_j for $j \in [1, m]$ has associated a Boolean function $f_j : \mathbb{R}^n \to \{0, 1\}$ s.t. $f_j(\boldsymbol{x}) = x_{\iota_j} \overset{?}{<} t_j$ where $\iota_j \in \mathbb{Z}_n$ is an index into the feature vector $\boldsymbol{x}$ and $t_j \in \mathbb{R}$ is a threshold. Thus $f_j(\boldsymbol{x})$ evaluates to 1 if and only if $x_{\iota_j} \leq t_j$, and 0 otherwise. Each leaf node of the tree is associated with an output value $z \in \mathbb{Z}_q$. Now to evaluate $\mathcal{T}(\boldsymbol{x}) = z$, start at the root node and evaluate $f_1(\boldsymbol{x})$. If $f_1(\boldsymbol{x}) = 0$ then proceed to evaluate the left child, if instead $f_1(\boldsymbol{x}) = 1$ then proceed to evaluate the right child. Continue in this manner until reaching a leaf and return the value z of this leaf.

For simplicity, and since we want to hide the structure of the tree, we assume that it is complete. We note that this is always possible as dummy nodes can be inserted as needed, which always evaluate to 0.

The depth of $\mathcal{T}$ is the longest path from the root node to any leaf, which we denote by d. Thus the tree will consist of d layers which we index by i, starting with the root layer

being 0. This also means that layer i contains exactly 2^i nodes.

We index nodes starting with 1 for the root node and then indexing by reading each layer top to bottom and left to right; thus if v_j is an internal node then v_{2j} is the left child of v_j and v_{2j+1} is the right child. We say the depth is the amount of nodes in the path from the root to, and including, the leaf; defining the root to be level 0. Thus the tree will have $m = 2^{d-1} - 1$ internal nodes and 2^d leaves. Note that the leaves will have index 2^d to 2^{d+1}.

Concretely we define $\mathcal{T}$ as a tuple of values $(\boldsymbol{t}, \boldsymbol{v}, \boldsymbol{z})$, where $\boldsymbol{t} \in \mathbb{R}^m$, $\boldsymbol{v} \in \mathbb{Z}_n^m$ and $\boldsymbol{z} \in \mathbb{Z}_q^{2^d}$. That is, $\boldsymbol{t} = (t_1, \ldots, t_m)$ and $\boldsymbol{v} = (v_1, \ldots, v_m)$ are lists of cardinality m. We view as ordered such that the j'th entry describe the j'th internal node in the tree. That is, each internal node v_j will compute the value $f_j = x_{v_j} \overset{?}{<} t_j$. $\boldsymbol{z} = (z_1, \ldots, v_{2^d})$ is an ordered list of integers, each representing an output of a leaf, thus each leaf node v_j (i.e. with $j \in]m, m + 2^d]$) will output the value $f_j = \boldsymbol{z}_{j-2^d}$.

Furthermore we consider the two-party setting where one party, called the *client* holds the feature vector $\boldsymbol{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$. The other party, called the *server* holds the decision tree $\mathcal{T}$. The parties then wish to compute $\mathcal{T}(\boldsymbol{x}) = z$ where the client learns z and the server learns nothing. We express this functionality formally in Fig. 13 (Sec. X).

To evaluate a decision tree privately we work over a finite set of integers $\mathbb{Z}_{2^k}$ instead of the real numbers. We convert a model based on real numbers by simply multiplying every decimal number in the model by a set constant and then rounding to nearest integer. This of course causes loss in accuracy, however, this rarely causes a problem and for real data the constant does not necessarily have to be large to avoid losing classification accuracy [30]. We furthermore note that this conversion still allows us to work with negative integers by considering the positive integers up to 2^k as a value in two's complement, as in Section IV-C, thus representing the positive integers up to $2^{k-1} - 1$ and following these, the negative integers from -2^{k-1} to -1. Because our computations will take place over a ring this representation will ensure arithmetic operations act as expected (assuming no over- and underflow).

1) An actively secure protocol: Our protocol takes departure in the work by De Cock *et al.* [29] which presents a protocol for evaluating decision trees based on secret sharing. We picked this protocol since it works in the arithmetic black box setting, whereas other approaches such as the one by Wu *et al.* [39] or Joye and Sahali [40] require homomorphic encryption. Still, the scheme by De Cock *et al.* is only secure in the semi-honest setting. We show how to make it actively secure by adding a cheap extra step.

The overall idea of their scheme is to first pick each relevant value from the input feature vector $\boldsymbol{x}$ for each node j, i.e. x_{v_j}. This is done by having the party holding the tree, P_1, input an n-bit vector for each of the m nodes. This bitvector will contain a single 1-bit in the position of the feature to use. That is, we associate a bit $c_{j,i} \in \{0, 1\}$ with each feature for each node (i.e. for all $i \in [1, n], j \in [1, m]$) s.t. $\sum_{i \in [1,n]} c_{j,i} = 1$ and $c_{j,v_j} = 1$. With these indicator bits we can arithmetically compute the attribute to use in the j'th node as $\sum_{i \in [1,n]} c_{j,i} \cdot x_i$.

If the tree holder is actively corrupted then it will be able to input value $c_{j,i}$ s.t. $\sum_{i \in [1,n]} c_{j,i} \neq 1$. This is a problem since this would allow a linear combination of $(x_1, \ldots, x_n)$ to be used for the comparison in each node of the tree. This would make it hard to write a simulation proof since the simulator would not know $\boldsymbol{x}$. To fix this issue we propose a solution that consists of enforcing that $c_{j,i}$ is a bit, then open $\sum_{i \in [1,n]} c_{j,i}$ for $j \in [1, m]$ and check if this is always 1. It is easy to see that this check is sufficient and clearly does not leak any information (as it is public knowledge that the opened value is supposed to be 1). Furthermore, it is also easy to enforce that $c_{j,i}$ is a bit, even if the whole ring $\mathbb{Z}_{2^k}$ is allowed as input: simply compute and open the value $(1 - c_{j,i}) \cdot c_{j,i}$ and check if it is 0. Again it can be argued that this is sufficient as $c_{j,i}$ equal to 0 or 1 are the only values for which $(1 - c_{j,i}) \cdot c_{j,i} = 0$ when working over $\mathbb{Z}_{2^k}$. Alternatively one could use Π_{RandBit} in Fig. 5 to get a random bit $[b]$, then open this towards P_1. P_1 could then publish a public value telling whether to let $c_{j,i} = [b]$ or $c_{j,i} = 1 - [b]$.

Adding this check allows us to compute the correct attributes for each node with active security. Via the attributes the output of the comparison in each node can be computed by the comparison subprotocol; the output is a bit indicating whether to go left (0) or right (1) down the tree. To evaluate the tree obliviously it is not possible to simply follow the correct path from the root to a leaf, as this would leak too much. Thus, we must visit every node in the evaluation. This is done by computing a bit for each leaf, which is the product of the output of the comparison for all the nodes on the path to the root.[1] There will be only one leaf for which this bit is 1. This is the leaf whose value is the final output of the decision tree evaluation. Since the evaluator is oblivious to which leaf this is, we multiply the bit of each leaf with the leaf's value and sum this for all leaves. Because the bit for every leaf, other than the correct one is 0, the output of this computation gives the correct result. This means that the comparisons can be done once; for each internal node of the tree we can then compute if it is part of the root-to-leaf path that is the result of the decision tree evaluation. Still, this requires $O(d)$ rounds of communication as all nodes on a given layer are dependent on a partial result of the nodes higher up the tree.

[1]The output of the comparison is negated for each node if it is a left child.

We can compute the partial values of all nodes in the tree using a "reduction" approach by exploiting the fact that multiplication is associative, i.e. that $x_1 \cdot x_2 \cdot x_3 \cdot x_4$ can be computed as $(x_1 \cdot x_2) \cdot (x_3 \cdot x_4)$, rather than $((x_1 \cdot x_2) \cdot x_3) \cdot x_4$. Thus we can compute the product of d values with $d-1$ multiplications and $\log(d-1)$ sequential rounds. For each node in every second layer from the root to the leaves, we compute the product of the output of its comparison with the output of the comparison of its parent (negated if it is a left node). Next we use these results to compute a product for every four layers, by multiplying the result of every node with the result of its grandparent (negated if its parent is a left child). We continue until we have computed a product between every layer in the tree.

Computing these products dominates protocol round cost, since both selecting the feature for all nodes, along with computing the comparison can be done in constant rounds (assuming we use the constant round comparison protocol). We express our actively secure protocol in Fig. 14 (Sec. X).

We note that De Cock *et al.* have implemented their protocol using boolean values, whereas we use arithmetic values. Using boolean values and replacing multiplication and addition with component-wise AND and XOR respectively would unfortunately not directly work on our fix to get active security. This is because XOR'ing two 1's would give 0, so an actively corrupted model holder would be able to have the classification happen using XOR combinations of the different values of the inputting party's feature vector. Even more importantly, as the feature values are not binary but rather elements from $\mathbb{Z}_{2^k}$, using a binary protocol would require k multiplications (AND gates) to compute $c_{j,i} \cdot x_i$ for $i \in [m]$ and $j \in [n]$, needed for each node in the tree. Even for relatively small values of k, like 32, this would probably not be faster using a binary protocol. In particular, using the optimized TinyOT protocol [37] this would be slower as the construction of a TinyOT triple is only about 12x faster than a SPD$\mathbb{Z}_{2^k}$ triple.

B. Support Vector Machines (SVMs)

We consider the machine-learning application of Support Vector Machines (SVMs), which is a type of supervised learning model used for classification. In its simple form it is used as a binary classifier, but it can easily be extended to classify data into any finite set of categories. More specifically an *SVM* is a function $\mathcal{S} : \mathbb{R}^n \to \mathbb{Z}_q$, where n is the dimension of the feature space and q the amount of categories (each represented by a non-negative integer). Similarly to the decision trees, the input $\boldsymbol{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$ to the function $\mathcal{S}$ is called the *feature vector*. The SVM $\mathcal{S}$ is implemented as a matrix $\boldsymbol{F} \in \mathbb{R}^{q \times n}$ where the rows are known as the *support vectors* and a vector $\boldsymbol{b} = (b_1, \ldots, v_q) \in \mathbb{R}^n$ which is called the *bias*. Conceptually, each support vector, along with a scalar from the bias vector, can classify an input $\boldsymbol{x}$ into a specific category (or not). Specifically denoting the rows of $\boldsymbol{F}$ as $F_1, \ldots, F_q$, the value $F_i \cdot \boldsymbol{x} + b_i$ is computed to give a score of how likely $\boldsymbol{x}$ is to be in category i. Thus, to find the most likely category of $\boldsymbol{x}$ we compute $\text{category}(\boldsymbol{x}) = \text{argmax}_{i \in [1,q]} F_i \cdot \boldsymbol{x} + b_i$ where the result is an integer representing the corresponding category.

Like the case for decision trees we consider the two-party setting where one party, called the *client* holds the feature vector $\boldsymbol{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$. The other party, called the *server* holds the SVM $\mathcal{S}$. The parties then wish to compute $\mathcal{S}(\boldsymbol{x}) = z$ where the client learns z and the server learns nothing. We express this functionality formally in Fig. 15 (Sec. X). Similarly to the decision trees, we work over a finite set of integers $\mathbb{Z}_{2^k}$, assuming two's complement representation to allow for integers in the range $[2^{k-1}, 2^k)$

1) An actively secure protocol: Our protocol follows the equation for SVM classification, $\text{category}(\boldsymbol{x}) = \text{argMax}_{i \in [1,q]} F_i \cdot \boldsymbol{x} + b_i$, very straight forward: In parallel compute the multiplication part of the inner products between $\boldsymbol{x}$ and F_i for all $i \in [1, q]$, as these are all independent. Next we note that addition does not require communication and thus we sequentially have the parties sum up the component-wise product computed, in order to compute the whole inner product. Next, for each inner product the parties add b_i. These steps only require constant rounds of communication and $q \cdot n$ multiplications. Finally computing the largest element of the q element list is done in $O(\log(q))$ rounds as follows: In a recursive manner divide the list of elements in halves until two or three elements remain. Compare these obliviously, and based on this comparison construct a binary list where the index of the maximum of these two or three elements is 1 and the is rest 0. This requires one or two comparisons and at most four multiplications. The merging of the partial results then require $O(q)$ comparisons and multiplications. Thus we end with a total of $O(q \cdot \log(q))$ comparisons and multiplications for the arg-max computation. We express this actively secure protocol in detail in Fig. 16 and 17 of Sec. X.

VI. Implementation

To reach a compromise between usability and efficiency we chose to implement the online and offline phases of SPD$\mathbb{Z}_{2^k}$ and our protocols in different frameworks.

We implement the online phase in FRESCO [23], an active open-source Java framework for MPC with a strong track record [15], [41]. We chose FRESCO as it offers an accessible API-based approach for writing MPC applications. This eased the implementation of the decision tree and SVM evaluation. Since FRESCO is written in Java, it also eases integration with broad, cross-platform pieces of software. Though Java is less efficient than C/C++ we consider the lower implementation and maintenance time required for Java to make the trade-off worthwhile.

As the benchmarks in the next section show, the offline phase requires orders of magnitude more time to execute than the online phase. As such, time spent ensuring an efficient offline phase gives a noticeable payoff in the view of total execution time. We therefore implement the offline phase in C/C++. The offline protocol we implemented is the same as described in the original SPD$\mathbb{Z}_{2^k}$ paper [18]. That is, authentication of elements and construction of triples is based on a vector Oblivious Linear function Evaluation (vOLE) construction through correlated OT, using the recent OT extension protocol by Scholl [42]. We integrated our implementation into the Bristol-SPDZ framework [24]. Bristol-SPDZ is a highly efficient framework for preprocessing. The framework already supports OT based preprocessing through MASCOT [11], and so integrating SPD$\mathbb{Z}_{2^k}$ preprocessing required little work.

FRESCO supports a bring-your-own-backend approach, and implements the most efficient SPDZ online phase, SPDZ-2 [19]. Besides containing an implementation of MASCOT, Bristol-SPDZ also implements the most efficient SPDZ preprocessing protocol, Overdrive [25], making the combination of FRESCO and Bristol-SPDZ a sensible choice for doing a fair comparison of both SPD$\mathbb{Z}_{2^k}$ and SPDZ, from preprocessing to online execution.

A. Optimizations

Here we detail several optimizations we used when implementing the online and preprocessing phases. Our preprocessing phase optimizations allow us to reduce the computation time so that in most cases, the networking is the bottleneck of the protocol. We describe our core online phase optimization (Sec. VI-A), and the three main preprocessing-phase optimizations (Sec. VI-A, VI-A, X-A).

Fast Integer Arithmetic for the Online Phase: FRESCO uses the BigInteger class to implement finite field arithmetic. The largest primitive type supported by Java is long, at 64 bits. Thus, in order to fully leverage the option of working with 2^k bit integers in SPD$\mathbb{Z}_{2^k}$ we implemented our own data-type for 128 bit integers, working on top of longs. This implementation outperforms the BigInteger class, even without taking into account that we don't need to reduce values modulo a large prime. Fig. 8 shows how much this implementation reduces the time spent on multiplications in the online phase. The figure compares the execution time for doing 1,000,000 online multiplications in FRESCO using the SPDZ protocol (which uses BigInteger and requires a modular reduction of a large prime), SPD$\mathbb{Z}_{2^k}$ based on BigInteger (that is, no modular reduction) and our optimized approach for SPD$\mathbb{Z}_{2^k}$ where multiplication is done directly with long types. Using our custom class based directly on longs is up to 4.7x times faster than Java's BigInteger class. Comparing this with the amount of computation required by SPDZ, the our SPD$\mathbb{Z}_{2^k}$ implementation becomes up to a factor 24.5 faster. Even using BigInteger for SPD$\mathbb{Z}_{2^k}$, still

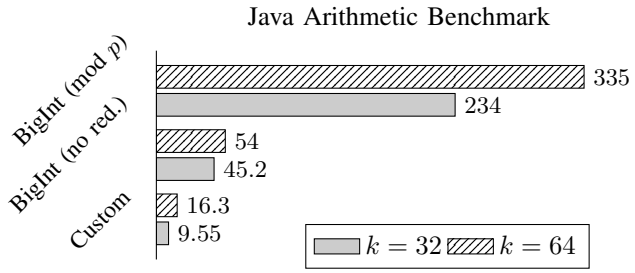


Figure 8: Time in milliseconds for 1,000,000 multiplications in Java, using different implementations. Numbers are the average of 100 experiments. "BigInt (mod p)" represents multiplications based on Java's BigInteger class *with* moduluar reducations, and "BigInt (no red.)" without. "Custom" represents the time used by our custom implementation.

results in a factor 5.2 improvement. This affirms statements made by Cramer *et al.* [18] that not needing to do modulo reduction of a large prime will have a noticeable impact on practical efficiency.

Fast Hashing with AES-NI: At several key places in the preprocessing phase, we perform many calls to a hash function on short inputs. Instead of using a standard hash function such as SHA-256, we use the Matyas-Meyer-Oseas construction [43], which builds a hash function out of a block cipher and is secure in the ideal cipher model. This greatly improves performance, since we can take advantage of Intel's AES-NI instructions on modern CPUs.

When the input and output of the hash are a single 128-bit block, the hash function can be done using fixed-key AES (for a random, pre-agreed key) with the simple construction :

$$H(x) = \mathsf{AES}_k(x) \oplus x$$

Note that this optimization was previously used for MPC in [11].

Fast Hashing for Large Domains: The MMO construction is less efficient when applied to a large domain, since processing multiple input blocks requires a re-keying operation for AES, which is a lot more expensive than fixed-key AES encryption with AES-NI. The construction of correlated OTs using Scholl's protocol [42] (needed for the SPD$\mathbb{Z}_{2^k}$ triple generation [18]) requires computing several hashes on very long inputs. We propose a new approach to implementing these by combining a 2-universal family of hash functions, $\mathcal{H}$, and a strong cryptographic hash such as SHA-256, with the function

$$H(x) = \mathsf{SHA256}(h(X)), \quad \text{where } h \leftarrow \mathcal{H} \tag{1}$$

The advantage over using only, say, SHA-256 or MMO, is that we can use a linear universal hash function over $\mathbb{F}_{2^{128}}$ such as GMAC, and this finite field arithmetic can be implemented very efficiently using carryless multiplication from the AES-NI instruction set. Note that when implementing this in the protocol, we require that the function h is sampled

at random by the receiver, just before the consistency check is carried out by the (possibly corrupt) sender.

We now argue that this approach still suffices for the security of the correlated OT protocol over $\mathbb{Z}_{2^k}$ from [42, Section 5]. In the consistency check, a hash function is used to allow a possibly corrupt sender in the protocol to 'prove' knowledge of certain values known to the receiver, to show that the sender's previous protocol messages were computed correctly. This proof consists of sending various hashes on very long inputs for the receiver to check.

Recall that when h is sampled at random from a 2-universal family of hash functions, collision resistance holds with overwhelming probability, as long as the *inputs* to h are independent of the random choice of h. In general, this may not hold in a protocol where the inputs can be adversarial, since given h it is easy to find two inputs that generate a collision. However, in our case there is no problem, since it turns out that the only inputs for which collision resistance is required to hold are *already fixed before the consistency check*. This is because the check is always carried out by the (honest) receiver on inputs known after the previous round of messages, as can be seen from the proof of [42, Lemma 8] or [44, Lemma 3.1]. In conclusion, as long as h is sampled after this then we are fine.

Note that if collision-resistance was the only property we needed, then we could even omit the SHA256 call in (1) . However, for the case of a corrupt *receiver* the protocols of [42], [44] also need a pseudorandomness property, so we apply a strong hash function on the output to act as a randomness extractor.

We microbenchmark these optimizations in Appendix X-A.

VII. Performance Evaluation

Further, we evaluate the concrete performance of our implementation of the online phase (Sec. VII-A) and the offline phase (Sec. VII-B). In Sec. XI of the appendix, we also evaluate memory usage, and show that it is not a bottleneck.

For the online phase, we run micro-benchmarks for our basic primitives as well as end-to-end evaluations of our two high-level applications on realistic datasets. We then compare our online implementation of SPD$\mathbb{Z}_{2^k}$ in FRESCO with the baseline SPDZ implementation in FRESCO. The SPDZ implementation in FRESCO is based on SPDZ-2 [19], which is the most recent and efficient online protocol for SPDZ. In our evaluation of the offline phase, we evaluate the SPD$\mathbb{Z}_{2^k}$ triple generation protocol across varying security parameters and network configurations. We then compare our offline implementation in the Bristol-SPDZ C++ framework with the two most recent and efficient protocols for SPDZ triple generation; MASCOT [11] and Overdrive [25]. Both of these are also implemented in the Bristol-SPDZ framework, which ensures a more fair comparison. Our implementation forms part of MP-SPDZ [45], a successor to Bristol-SPDZ.

Furthermore, we are unaware of any other *practically competitive* protocols considering a dishonest majority of malicious parties in the arithmetic setting and thus believe that comparing to SPDZ is sufficient.

We chose to benchmark our protocols in the two-party setting, although all our constructions (except the protocols for the specific setting of oblivious decision tree and SVM evaluation) generalize to an arbitrary amount of parties. We did this for simplicity and since both SPDZ and SPD$\mathbb{Z}_{2^k}$ generalize to more parties with similar overheads.

Setup. We run all experiments in the two-party setting. Each party executes on an m5d.xlarge AWS EC2 instance running Ubuntu 16.04, with 4 vCPUs and 16GB memory. The instances are hosted within the same region and connected over an up to 10 Gbps link. To investigate how different network settings affect the performance of our protocols, we use tc to simulate bandwidth restrictions and latency. For all experiments, we performed a minimum of 20 total runs and report the average result. We discard the first run in order to ensure the JVM has warmed up.

A. Online Phase

For our online phase experiments, we consider two bit length settings. For the low bit length setting, we use $k = s = 32$ (total bit length of 64) which supports 32-bit comparisons and equality operations and affords 26 bit statistical security. We compare this setting to running SPDZ over a 64 bit field; the larger field is necessary to ensure at least 26 bits of statistical security in the comparison protocol used by SPDZ. Similarly, we compare the larger bit setting with $k = 64$, $s = 64$, total bit length 128, and 57 bit statistical security to SPDZ over a 128 bit field with 57 bit statistical security.[2]

Table I shows throughput times (operations per second) for three non-linear operations: multiplication, equality, and comparison on a 1 Gbps network. We believe a 1 Gbps LAN to be a suitable setting for the family of SPD$\mathbb{Z}_{2^k}$ and SPDZ protocols; the high latency of lower bandwidth WAN networks would significantly limit performance due to the protocols' non-constant round complexity. Constant round protocols are more appropriate for such settings. Conversely, we do not report numbers for a faster network since at 1 Gbps our implementation is not network-bound.

We obtain the throughput numbers from batched runs, i.e., parallel[3] operations with batched communication. We use batches of 100,000 parallel operations for multiplications and 5,000 for equality and comparison.

[2] 26, respectively 57 bits of security, are chosen for a fair comparison with SPD$\mathbb{Z}_{2^k}$, as SPD$\mathbb{Z}_{2^k}$ has a logarithmic deterioration of the statistical security, because of batched MAC checks.

[3] Parallel here does not imply running on multiple threads; it merely means that the operations are independent and communication can thus be batched.

Table I: Throughput in elements per second for the online phase of micro operations over 1 Gbps network. The factor columns express the runtime improvement factor of SPD$\mathbb{Z}_{2^k}$ over SPDZ in FRESCO.

	$k = 32$			$k = 64$		
	SPD$\mathbb{Z}_{2^k}$ ($\sigma = 26$)	SPDZ ($\sigma = 26$)	Factor	SPD$\mathbb{Z}_{2^k}$ ($\sigma = 57$)	SPDZ ($\sigma = 57$)	Factor
Multiplication	687041	141346	4.9x	522258	114071	4.6x
Equality	15334	3213	4.8x	6902	1282	5.4x
Comparison	9153	1769	5.2x	4514	756	6.0x

For multiplications we see between a 4.6 and 4.9-fold improvement for the different bit-length settings. This performance gain stems from a speed up in local computation as well as reduced communication. Local computation improves since we do not need to perform modular reductions and use a custom class for ring elements of specific bit-length (64 and 128 bit) which significantly outperforms BigInteger arithmetic as discussed in Sec. VI-A. The total amount of data sent is also reduced; for all protocols that require communicating an element to the other parties, we only need to send the k least significant bits, as opposed to an entire element for SPDZ. This alone cuts communication in half.

Comparison and equality (for $k = 64$) show an even higher increase in performance, with the biggest improvement for comparison, six-fold for $k = 64$ and five-fold for $k = 32$.

Switching to boolean mode for the comparison protocol replaces a majority of the underlying multiplications with bit-multiplications, which require sending only 2 bits per party, in contrast to two whole field elements. This drastically reduces communication as shown in Table V. The improvement in throughput is not directly proportional to the reduction in communication since our implemention is not network-bound at 1 Gbps. We nonetheless observe an improvement since reducing data sent also reduces the amount of local serialization and data copying FRESCO does as part of networking.

Equality also benefits from switching to boolean mode, though the performance improvement is less pronounced; we operate in arithmetic mode by default and must convert the boolean output of the Π_{EQZ} protocol to an arithmetic sharing. This introduces an additional protocol round. We avoid this conversion for comparisons (see Step 5 of Fig. 6).

We note that for $k = 32$, multiplication yields a slightly higher relative improvement than equality. This is due to the fact that the benefit of reduced communication for equality is not high enough to outweigh the internal framework-related overhead of executing a more complex protocol.

The lower communication of multiplication and comparison directly affects the communication and computation required for the more advanced applications of decision trees and SVMs, as can be seen in Tab. II, III and V.

B. *Offline Phase*

Fig. 9 compares our implementation of triple generation to the two state-of-the-art preprocessing protocols of the SPDZ family; MASCOT [11], and Overdrive [25]. All three implementations are part of the MP-SPDZ framework [45]. We first note that SPD$\mathbb{Z}_{2^k}$ saturates the network for all number threads we tested in the WAN setting, and for 2 and 4 threads on a 1 Gbps LAN. However, SPD$\mathbb{Z}_{2^k}$ becomes computationally bounded in the case for one thread on the 1 Gbps LAN and for all number of threads we tested in the 10 Gbps LAN setting. This is visible from the graphs by noting the convergence of throughput of SPD$\mathbb{Z}_{2^k}$ in the WAN setting and at 2 threads in the 1 Gbps LAN.

For similar bit-lengths, the efficiency of SPD$\mathbb{Z}_{2^k}$ and MASCOT is almost the same. This is expected as our implementation is closely related to MASCOT. For smaller bit-lengths, *i.e.*, $k = 32$, our implementation is significantly more efficient since it requires far less communication. We note that the MASCOT implementation is hard-coded for fields of 128 bits and thus we cannot compare how it fares with a smaller field. Overdrive performs significantly better than SPD$\mathbb{Z}_{2^k}$ in the WAN setting, but the difference shrinks in a LAN. This is not surprising as Overdrive uses significantly less communication than MASCOT, and thus fares much better in a slower network than MASCOT, and consequently SPD$\mathbb{Z}_{2^k}$. SPD$\mathbb{Z}_{2^k}$ can nonetheless compete with Overdrive, given a fast enough network; (Fig. 9c) shows that the low bit setting for SPD$\mathbb{Z}_{2^k}$ matches Overdrive performance in a 10 Gbps LAN.

We ran SPD$\mathbb{Z}_{2^k}$ and MASCOT in batches of 1024 triples, and Overdrive in low-gear mode [25], the most efficient mode in the two-party setting. Increasing the thread count further did not significantly improve the throughput of any of the protocols we benchmarked.

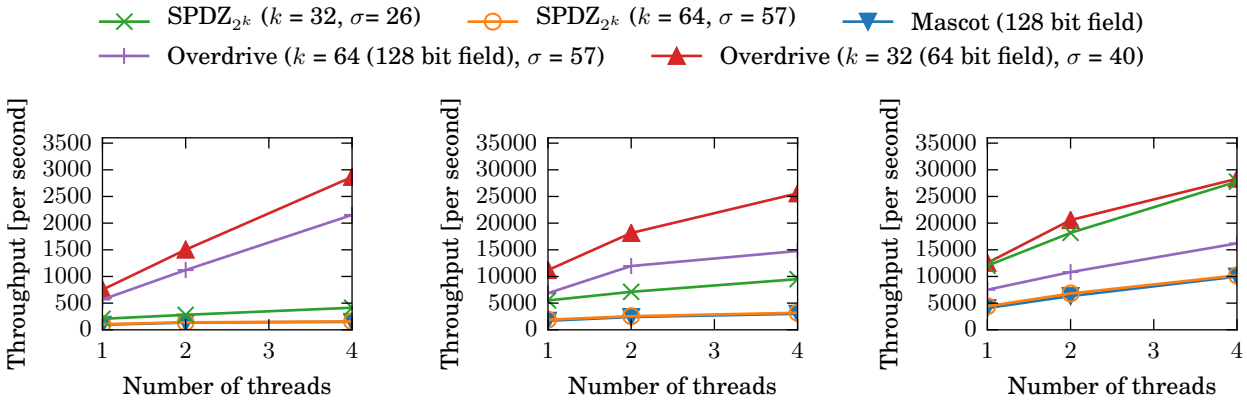
The amount of preprocessed material needed for the operations/applications considered in this work can be found in Table IV. The table includes count of the arithmetic and bit triples needed for both SPD$\mathbb{Z}_{2^k}$ and SPDZ, along with the amount of random bits needed (which require an arithmetic multiplication triple for both SPD$\mathbb{Z}_{2^k}$ and SPDZ). We note the timing column is only an estimate, based on the time required for triple generation and bit triple generation. Thus the true time will be slightly larger for both SPD$\mathbb{Z}_{2^k}$ and SPDZ, because of the usage of authentication and input masks. However, these are in the order of a magnitude faster

Table II: Online phase benchmarking of evaluation of decision trees over 1 Gbps network. The factor columns express the runtime improvement factor of SPD$\mathbb{Z}_{2^k}$ over SPDZ in FRESCO. Times are in milliseconds per sample.

			$k = 32, \sigma = 26$			$k = 64, \sigma = 57$		
Dataset	Depth, Num. Features	Batch Size	SPD$\mathbb{Z}_{2^k}$	SPDZ	Factor	SPD$\mathbb{Z}_{2^k}$	SPDZ	Factor
Hill Valley	3, 100	1	21 ms	24 ms	1.2x	26 ms	34 ms	1.3x
Spambase	6, 57	1	48 ms	104 ms	2.2x	56 ms	128 ms	2.3x
Diabetes	9, 8	1	80 ms	215 ms	2.7x	122 ms	443 ms	3.6x
Hill Valley	3, 100	5	6 ms	10 ms	1.7x	7 ms	15 ms	2.1x
Spambase	6, 57	5	14 ms	40 ms	2.9x	17 ms	68 ms	4.0x
Diabetes	9, 8	5	41 ms	185 ms	4.5x	78 ms	376 ms	4.8x

Table III: Online phase benchmarking of SVM evaluation over 1 Gbps network. The factor columns express the runtime improvement factor of SPD$\mathbb{Z}_{2^k}$ over SPDZ in FRESCO. Times are in milliseconds per sample.

			$k = 32, \sigma = 26$			$k = 64, \sigma = 57$		
Dataset	Num. Classes, Features	Batch Size	SPD$\mathbb{Z}_{2^k}$	SPDZ	Factor	SPD$\mathbb{Z}_{2^k}$	SPDZ	Factor
CIFAR	10, 2048	1	82 ms	214 ms	2.6x	99 ms	255 ms	2.6x
MIT	67, 2048	1	379 ms	1318 ms	3.5x	499 ms	1582 ms	3.2x
ALOI	463, 128	1	242 ms	857 ms	3.5x	362 ms	1312 ms	3.6x
CIFAR	10, 2048	5	39 ms	168 ms	4.3x	57 ms	209 ms	3.7x
MIT	67, 2048	5	225 ms	1101 ms	4.9x	294 ms	1428 ms	4.9x
ALOI	463, 128	5	162 ms	741 ms	4.6x	244 ms	1220 ms	5.0x



(a) WAN (50 Mbps, 100 ms latency)

(b) LAN (1 Gbps, 0.1 ms latency)

(c) LAN (10 Gbps, 0.1 ms latency)

Figure 9: Triple generation throughput across different protocols and network settings.

to construct compared to triples. Furthermore, the amount needed is fewer than the number of triples required and so the true impact of constructing these will be minuscule. Most importantly though, the amount required by both SPD$\mathbb{Z}_{2^k}$ and SPDZ is almost the same and so the effect on the relative difference between the two will be insignificant.

C. Communication

Table V shows theoretical communication complexity for the online phase as well as preprocessing.

For the online costs, we see a large reduction in communication due to our use of binary multiplications for comparison and equality. The improvements range from 42x to 85x. For pre-processing, we include the theoretical communication complexity for the SPD$\mathbb{Z}_{2^k}$ offline phase based on vOLE (and hence correlated OT) and Overdrive for SPDZ. Though we reduce communication by using bit-triples for bit-wise multiplication, the communication complexity of our preprocessing is still muchs larger than Overdrive. This is because the communication complexity of an Overdrive triple is less than that of a bit-triple using the most efficient TinyOT preprocessing [37]. As such, adapting the Overdrive preprocessing to the ring setting is promising future work. Notably, if we run preprocessing on a network with enough bandwidth, we outperform both MASCOT and Overdrive for all applications and tested values of k (cf. Table IV); our SPD$\mathbb{Z}_{2^k}$ specific protocol optimizations allow us to use fewer random bits, more efficient bit-triples based on TinyOT, and thus fewer expensive multiplication triples. Thus, while raw triple preprocessing for SPD$\mathbb{Z}_{2^k}$ is slower

Table IV: Costs of the preprocessing for different operations/applications. Timings are estimates based on triples/random bits needed and are based on a 4 threads execution on a LAN supporting up to 10 Gbps. For SPDZ, Overdrive [25] is used. For bit triple generation the optimized TinyOT protocol by Wang *et al.* [37] is used.

	SPD$\mathbb{Z}_{2^k}$, $k = 32$, $\sigma = 26$				SPDZ, $k = 32$, $\sigma = 26$ (64 bit field)			
	# triples	# bit-triples	# random bits	time (ms)	# triples	# bit-triples	# random bits	time (ms)
Comparison	0	60	33	1.43	60	0	58	4.04
Equality	0	31	33	1.34	31	0	58	3.04
DTree (diabetes)	5460	15300	8415	571	20760	0	14790	1216
SVM (aloi)	63332	27720	15246	3055	91052	0	26796	4030

	SPD$\mathbb{Z}_{2^k}$, $k = 64$, $\sigma = 57$				SPDZ, $k = 64$, $\sigma = 57$ (128 bit field)			
	# triples	# bit-triples	# random bits	time (ms)	# triples	# bit-triples	# random bits	time (ms)
Comparison	0	124	65	7.22	124	0	121	14.9
Equality	0	63	65	7.04	63	0	121	11.2
DTree (diabetes)	5460	31620	16575	2417	37080	0	30855	4124
SVM (aloi)	63332	57288	30030	10006	120620	0	55902	10714

Table V: Total theoretical communication complexity counted in (kilo-, mega-, giga-) bytes for the two-party case. Values for SPDZ are based on Overdrive in Low Gear [25]. For bit triples we use the optimized TinyOT protocol of Wang *et al.*[37]. The communication of comparison and equality do not include authenticating input overhead since we assume amortized execution and exclude setup and initialization communication.

	$k = 32$, $\sigma = 26$				$k = 64$, $\sigma = 57$			
	SPD$\mathbb{Z}_{2^k}$		SPDZ (64 bit field)		SPD$\mathbb{Z}_{2^k}$		SPDZ (128 bit field)	
	Preprocessing	Online	Preprocessing	Online	Preprocessing	Online	Preprocessing	Online
Comparison	627 KB	46 B	148 KB	1.89 KB	3.58 MB	94 B	508 KB	7.78 KB
Equality	486 KB	24 B	107 KB	1.01 KB	3.08 MB	48 B	366 KB	3.97 KB
DTree (Diabetes)	209 MB	131 KB	40.8 MB	705 KB	1.10 GB	262 KB	110 MB	2.37 MB
SVM (ALOI)	908 MB	1.44 KB	139 MB	3.24 MB	4.06 GB	2.88 MB	341 MB	8.29 MB

than Overdrive, we still outperform it for more advanced operations and real-world applications.

D. Applications

In tables II and III we show online benchmarking results of Protocols Π_{DecTree} and Π_{SVM} from Sec. V. The tables show the online execution time of these protocols when obliviously classifying data, both using SPDZ and SPD$\mathbb{Z}_{2^k}$. For both decision tree and SVM evaluation, we measure evaluation time for a single data point, and the amortized time of evaluating multiple points in batches of 5 (since a service will likely classify more than a single data point).

1) Decision Trees: Table II shows online times for oblivious evaluation of some binary data models by De Cock *et al.* [29], based on datasets from the UCI repository[4]. The models are used to identify hills vs. valleys on 2-D graphs (Hill Valley), diabetes in women of Pima Indian decent (Diabetes) and spam vs. non-spam e-mail based on textual content (Spambase). We chose these models as they contain a large variation in the amount of features.

We see a noticeable, relative improvement of SPD$\mathbb{Z}_{2^k}$ over SPDZ in all the models we benchmarked, which further increases with the depth of the tree. As expected, batched evaluation yields better throughput; the batched runs also result in a bigger performance improvement for SPD$\mathbb{Z}_{2^k}$ over SPDZ. This shows that comparisons, which are needed for each node of the tree, become the bottleneck. This holds for both SPDZ and SPD$\mathbb{Z}_{2^k}$. Still, the impact is much greater for SPDZ as a depth increase from 3 to 9 results in a relative slowdown of up to 25x, whereas for SPD$\mathbb{Z}_{2^k}$ the slowdown is at most 18x. We thus see how important an efficient realization of an operation like comparison is for the real-world setting of decision trees. Finally, comparing $k = 32$ with $k = 64$ we see that the smaller ring gives up to a 1.9x improvement for SPD$\mathbb{Z}_{2^k}$ and 2.0x for SPDZ, showing the importance of flexibility in domain size.

2) SVMs: Table III show oblivious evaluation of image classification models constructed by Makri *et al.* [30], and a model with few features but many classes[5]. The models by Makri *et al.* are built on the datasets CIFAR-10 [46] and MIT-67 [47] where Inception-v3 is used for feature extraction [48]. We chose these models to get a difference in number of classes and features. We see a large relative improvement of SPD$\mathbb{Z}_{2^k}$ over SPDZ. This holds even for a the smallest amount of classes, and thus smallest amount of comparisons as well. This indicates that the comparison is the main bottleneck in the SVM execution in both systems, as this factor is close to the direct improvement

[4]UC Irvine Machine Learning repository https://archive.ics.uci.edu/ml/datasets.html.

[5]The model aloi at https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multiclass.html#aloi.

of comparison in SPD$\mathbb{Z}_{2^k}$ relative to SPDZ, as shown in Tables I. It is interesting that this holds even for few classes and many features, as shown by the Cifar row in the batched setting.

VIII. Conclusions

In this work we showed how to compute basic functionality like comparison, equality, bit decomposition and truncation when working in the ring $\mathbb{Z}_{2^k}$, thus overcoming issues such as zero-divisors and lack of invertibility that arise in this setting.

We confirmed experimentally the conjecture from [18] that secure computation over the ring $\mathbb{Z}_{2^k}$ provides many advantages in the online phase, with only slight increase in offline cost. In particular we saw up to a 5-fold improvement in computation for various tasks, and up to a 85-fold reduction in online communication costs for secure comparison, as compared to the field setting.

In the future, we plan to explore other applications of SPD$\mathbb{Z}_{2^k}$, *e.g.*, neural network evaluation, where share conversions are known to help [17]. It also important to close the performance gap between SPD$\mathbb{Z}_{2^k}$ pre-processing and Overdrive; SHE-based techniques present a promising venue.

IX. Appendix

A. Carry

This subprotocol computes the carry bit of an addition between $a \in \mathbb{Z}_{2^\ell}$ and $b \in \mathbb{Z}_{2^\ell}$, when the initial carry-in bit is set to $u \in \{0,1\}$. That is, it computes the function $\mathsf{Carry}_\ell(a,b,u) := \left(a + b + u \overset{?}{\geq} 2^\ell\right)$. We will use a variant where a and u are public, and the parties have access to the bits of b in secret-shared form, $[b_0]_2, \ldots, [b_{\ell-1}]_2$. The protocol works by simply running a binary circuit on SPD$\mathbb{Z}_{2^k}$ using AND triples. A circuit with $2\ell-2$ AND gates and depth $\log(\ell)$ can be constructed using standard methods, see for instance CarryOutL as described in [22]. We denote this protocol by $[v]_2 = \Pi_{\mathsf{Carry}}(a_0, \ldots, a_{\ell-1}, [b_0]_2, \ldots, [b_{\ell-1}]_2, u)$.

B. Binary Addition

Sometimes we want to compute the actual result of a binary addition between $a \in \mathbb{Z}_{2^k}$ and $b \in \mathbb{Z}_{2^k}$, instead of just the carry bit. A naive addition circuit has a linear depth and number of AND gates, but to reduce the depth we can instead use a recursive method which gives a logarithmic depth circuit with $k \log k$ AND gates. When a is public and the bits of b are secret-shared, the resulting protocol needs $\log k$ rounds and $k \log k$ secure ANDs. This is the same as the CarryAddL protocol from [22].

C. Probabilistic Truncation

In this section we describe a protocol for computing $[b]$ from $[a]$, where b is an approximation of $\left\lfloor \frac{a}{2^d} \right\rfloor$. With probability at least $1 - 2^{\ell-k}$ the error in the approximation is at most 2^{-d}, where $\ell \ll k$ is the bit-length of the number being truncated.

This protocol is taken from [17], which suits our setting since it does not require division by powers of 2. The protocol works by opening a masked version of a, $c = (a - r) \bmod 2^k$. This masked value can be truncated in the clear to get $\left\lfloor \frac{c}{2^d} \right\rfloor$, and then the truncation of r (which is shared since the parties have shares of the bits of r) can be added to get shares of $\left\lfloor \frac{a}{2^d} \right\rfloor$. However, there is naturally an additive rounding error.

The protocol is stated in detail in Fig. 10. The proof of correctness is similar to [17], and given in the full version.

Protocol Π^d_{TruncP}

Input: Shared value $[a]$, with $a \in \mathbb{Z}_{2^k}$.
Output: Shared value $[b]$, where $b \equiv_k \left\lfloor \frac{a}{2^d} \right\rfloor$.

1) Call $[r_0], \ldots, [r_{k-1}] \leftarrow \Pi_{\mathsf{RandBit}}()$. Let $[r] = \sum_{i=0}^{k-1} [r_i] 2^i$.
2) Compute $c \leftarrow \mathsf{Open}([r] - [a])$. Let $c' = c \bmod 2^d$ and $[r'] = \sum_{i=d}^{k-1} [r_i] 2^{i-d}$.
3) Output $[b] \leftarrow -\left\lfloor \frac{c}{2^d} \right\rfloor + [r']$.

Figure 10: Protocol for truncating a value by d bits probabilistically

Proposition IX.1. *On input $[a]$ with $a \leq 2^\ell$, protocol Π^d_{TruncP} securely computes $[b]$ where, with probability at least $1 - 2^{\ell-k}$, $b = \left\lfloor \frac{c}{2^d} \right\rfloor - v$ for some $v \in \{0,1\}$.*

Proof: Security follows from the fact that a is masked by r. Now, for correctness, notice that since $c = (r - a) \bmod 2^k$, it holds that $c = (r \bmod 2^k) - a + 2^k u$ with $u = \mathsf{Carry}_k(c, a)$, where $\mathsf{Carry}_k(x, y)$ is the k-th carry bit when adding x and y. In particular, $c \bmod 2^d = (r \bmod 2^d) - (a \bmod 2^d) + 2^d v$ where $v = \mathsf{Carry}_d(c, a)$. Now, by recalling that for all $x \in \mathbb{Z}^+$ it holds that $\left\lfloor \frac{x}{q} \right\rfloor = \frac{x - (x \bmod q)}{q}$, we can see that the output b satisfies

$$\begin{aligned} b &= -\frac{c - (c \bmod 2^d)}{2^d} + r' \\ &= -\frac{r - (r \bmod 2^d)}{2^d} + \frac{(a \bmod 2^k) - (a \bmod 2^d)}{2^d} \\ &\quad - 2^{k-d}u + v + r' \\ &= -r' + \left\lfloor (a \bmod 2^k)/2^d \right\rfloor - 2^{k-d}u + v + r' \\ &= \left\lfloor (a \bmod 2^k)/2^d \right\rfloor - 2^{k-d}u + v, \end{aligned}$$

so the error in the approximation is $-2^{k-d}u + v$. Moreover, it is not hard to see that u can also be calculated as $(r \bmod 2^k) \overset{?}{<} a$, so the probability that $u = 1$ is the probability that the random value $r \bmod 2^k$ is strictly smaller than a, and since $a \leq 2^\ell$, this probability is upper bounded by $2^{\ell-k}$. ■

1) Deterministic Truncation: The previous protocol allows us to calculate shares of $\lfloor a/2^d \rfloor$, but it is probabilistic in two aspects. First, there is the bad event in which $\mathsf{Carry}_k(c, r) = 1$, in which makes the error in the truncation to be around 2^{k-d}; fortunately this event happens with at most probability $2^{\ell-k}$. However, there is also the event in which (using the notation from the previous section) $v = \mathsf{Carry}_d(c, a) = 1$, in which case we get an error of 1 in the result. Based on these observations we can obtain different truncation protocols that provide different guarantees about the result, with different costs.

$\Pi_{\mathsf{TruncP1}}(d)$*:* The truncation is exact with probability $1-2^{\ell-k}$. For this the parties use bit-decomposition (Section IX-D) and the carry protocol (Section IX-A) to compute $[v] \leftarrow \mathsf{Carry}_d(c, [a])$ and define the output of the truncation to be $[b] - [v]$.

$\Pi_{\mathsf{TruncP2}}(d)$*:* With probability 1, the truncation has an error of at most 1. To achieve this the parties use bit-decomposition and the carry protocol to compute $[u] \leftarrow \mathsf{Carry}_k(c, [a])$ and define the output of the truncation to be $[b] + 2^{k-d}[u]$.

$\Pi_{\mathsf{TruncD}}(d)$*:* The truncation is exact with probability 1. This is essentially a combination of the two cases above. This is obtained by letting the parties using bit-decomposition and the carry protocol twice to get $[u] \leftarrow \mathsf{Carry}_k(c, [a])$ and $[v] \leftarrow \mathsf{Carry}_d(c, [a])$, and let the output be $[b] + 2^{k-d}[u] - [v]$.

D. Bit-Decomposition

This protocol allows the parties to obtain $([a_0], \ldots, [a_{m-1}])$ from $[a]$, where $a = \sum_{i=0}^{k-1} a_i 2^i$ and $m \leq k$. This protocol is taken from [22], and it is described in Fig. 11.

Protocol Π_{BitDec}

INPUT: Shared value $[a]$.
OUTPUT: Shared values $[a_0], \ldots, [a_{k-1}]$, where $a = \sum_{i=0}^{k-1} a_i 2^i \in \mathbb{Z}_{2^k}$.

1) Call $[r_0], \ldots, [r_{k-1}] \leftarrow \Pi_{\mathsf{RandBit}}()$ and compute $[r] = [\sum_{i=0}^{k-1} r_i 2^i]$.
2) Let $c \leftarrow \mathsf{Open}([a] - [r])$.
3) Call $[r_0]_2, \ldots, [r_{k-1}]_2 \leftarrow \Pi_{\mathsf{A2B}}([r_0], \ldots, [r_{k-1}])$.
4) Use the binary addition protocol from Section IX-B to output $(c_0, \ldots, c_{k-1}) + ([r_0]_2, \ldots, [r_{k-1}]_2)$, getting as output $[a_0]_2, \ldots, [a_{k-1}]_2$.
5) Output $[a_0], \ldots, [a_{k-1}] = \Pi_{\mathsf{B2A}}([a_0]_2, \ldots, [a_{k-1}]_2)$.

Figure 11: Protocol for bit-decomposing a shared value

Correctness is clear since $r+c = r+(a-r) \bmod 2^k \equiv_k a$.

E. Bit-wise Comparison

To compare bitwise-shared values $a = \sum_{i=0}^{k-1} a_i 2^i$ and $b = \sum_{i=0}^{k-1} b_i 2^i$ the parties execute the protocol in Fig. 12.

Security is obvious. To argue the correctness of the protocol, consider $b' = \sum_{i=0}^{k-1} (1 - b_i) \cdot 2^i$ and notice that

$$a < b \Leftrightarrow a - b < 0 \Leftrightarrow a + (2^k - b) < 2^k \Leftrightarrow a + (b' + 1) < 2^k.$$

Protocol Π_{BitLT}

INPUT: Shared bits $[a_0]_2, \ldots, [a_{k-1}]_2, [b_0]_2, \ldots, [b_{k-1}]_2$.
OUTPUT: Shared value $[u]_2$, where $u = a \overset{?}{<} b$ with $a = \sum_{i=0}^{k-1} a_i 2^i$ and $b = \sum_{i=0}^{k-1} b_i 2^i$.

1) Compute $[b'_i]_2 = 1 - [b_i]_2$ for $i = 0, \ldots, k-1$.
2) Return $1 - \Pi_{\mathsf{Carry}}([a_0]_2, \ldots, [a_{k-1}]_2, [b'_0]_2, \ldots, [b'_{k-1}]_2, 1)$.

Figure 12: Protocol for comparing bitwise-shared values

Therefore, $a \overset{?}{<} b$ is equal to 1 if and only if adding a with b', when the carry bit is set, does not result in a carry. This is precisely what is done in the protocol.

X. Protocols and Functionalities for Decision Tree and SVM Evaluation

Fig. 13 presents the ideal functionality for decision tree evaluation, which is realised by the protocol in Fig. 14. Fig. 15 contains the ideal functionality for SVM evaluation, whilst the protocol for implementing this is in Fig. 17, and the arg-max subroutine in Fig. 16. The arg-max protocol is based on a work by Toft [49].

Functionality $\mathcal{F}_{\mathsf{DecTree}}$

Initialization: The functionality is initialized by (Init, d, n, k) from P_1 and P_2.
Compute: On input $(\mathsf{Input}, \mathcal{T})$ from P_1 and $(\mathsf{Input}, \boldsymbol{x})$ from P_2 where $\mathcal{T} = (\boldsymbol{t}, \boldsymbol{v}, \boldsymbol{z})$ with $\boldsymbol{t} \in \mathbb{Z}_{2^k}^m, \boldsymbol{v} \in \mathbb{Z}_n^m, \boldsymbol{z} \in \mathbb{Z}_{2^k}^{2^d}$ and $\boldsymbol{x} \in \mathbb{Z}_{2^k}^n$, return $\mathcal{T}(\boldsymbol{x}) = z$ to P_2.

Figure 13: Functionality for evaluating decision trees

A. Offline Micro-benchmarks

From the micro-benchmark in Fig. 18 we see that these two optimizations alone can cut preprocessing time in half. These benchmarks are done using network supporting communication up to 10 Gbps. The high bandwidth ensures that the execution will be computationally bounded, as shown in Sec. VII-B, and so that the relative improvements don't get distorted by not having enough network bandwidth.

Reducing the Number of Consistency Checks in Correlated OT: We further optimize the correlated OT protocol by reducing the number of consistency checks. The protocol of [42] performs one check between every pair of OT inputs, for a total of k^2 checks. We apply an optimization from [44] (which [42] is based on, but does not use) which reduces the number of checks to $\mu \cdot k$ for a small constant μ. This greatly reduces the computational costs of the protocol, for a small sacrifice in security: based on the analysis in [44], using $\mu = 3$ should result in losing no more than 9 bits of statistical security. In Fig. 19 we show a micro-benchmark of the effect of using $\mu = 3$ instead of $\mu = 10$, which would be the default without the optimization. The optimization reduces the triple generation time by up to 40%.

Protocol Π_{DecTree}

INPUT: $\mathcal{T} = (\boldsymbol{t}, \boldsymbol{v}, \boldsymbol{z})$ with $\boldsymbol{t} = (t_1, \ldots, t_m) \in \mathbb{Z}_{2^k}^m, \boldsymbol{v} = (v_1, \ldots, v_m) \in \mathbb{Z}_n^m, \boldsymbol{z} = (z_1, \ldots, z_{2^d}) \in \mathbb{Z}_{2^k}^{2^d}$ from P_1 and $\boldsymbol{x} = (x_1, \ldots, x_n) \in \mathbb{Z}_{2^k}^n$ from P_2.

1) Let $[t_j] \leftarrow \mathsf{Input}(t_j, P_1)$ for each $j \in [m]$. Similarly let $[z_j] \leftarrow \mathsf{Input}(z_j, P_1)$ for each $j \in [1, 2^d]$ and $[c_{j,i}] \leftarrow \mathsf{Input}(c_{j,i}, P_1)$ for each $i \in [1, n]$ and $j \in [1, m]$ where $c_{j,v_j} = 1$ and $c_{j,i} = 0$ for $i \neq v_j$.
2) The parties compute $[c_j] = \sum_{i=1}^n [c_{j,i}]$ for $j \in [1, m]$.
3) The parties then let $c_j \leftarrow \mathsf{Open}([c_j])$ and verify that $c_j = 1$ for $j \in [1, m]$. If this is not the case for any $j \in [1, m]$ then output Abort.
4) The parties the compute $[\bar{c}_{j,i}] = [c_{j,i}] \cdot (1 - [c_{j,i}])$ and let $\bar{c}_{j,i} \leftarrow \mathsf{Open}([\bar{c}_{j,i}])$ for all $i \in [1, n]$ and $j \in [1, m]$.
5) The parties verify that all $\bar{c}_{j,i} = 0$ for all $i \in [1, n]$ and $j \in [1, m]$, if not they output Abort.
6) Let $[x_i] \leftarrow \mathsf{Input}(x_i, P_2)$ for all $i \in [1, n]$.

OUTPUT: Nothing to P_1 and $\mathcal{T}(\boldsymbol{x}) = z$ to P_2.

1) Compute $[f_j] = \Pi_{\mathsf{LT}}\left(\left(\sum_{i=1}^n [c_{j,i}] \cdot [x_i]\right), [t_j]\right)$ for $j \in [1, m]$.
2) For $l \in [1, d]$, let j_l denote the l'th bit of the index j (letting j_1 be the least significant bit). Then for $j \in [1, 2^d]$ compute

$$[f_j] = \prod_{l \in [d]} \begin{cases} (1 - [f_{\lfloor (j+m)/2^l \rfloor}]), & \text{if } j_l = 0 \\ [f_{\lfloor (j+m)/2^l \rfloor}], & \text{if } j_l = 1 \end{cases}$$

3) Compute $[z] = \sum_{j=1}^{2^d} [z_j] \cdot [f_{j+m}]$ and then let $z \leftarrow \mathsf{Open}([z], P_2)$.

Figure 14: Protocol for evaluating decision trees

Functionality $\mathcal{F}_{\mathsf{SVM}}$

Initialization: The functionality is initialized by (Init, q, n, k) from P_1 and P_2.

Compute: On input $(\mathsf{Input}, \mathcal{S})$ from P_1 and $(\mathsf{Input}, \boldsymbol{x})$ from P_2 where $\mathcal{S} = (\boldsymbol{F}, \boldsymbol{b})$ with $\boldsymbol{F} \in \mathbb{Z}_{2^k}^{q \times n}$, $\boldsymbol{b} \in \mathbb{Z}_{2^k}^q$ and $\boldsymbol{x} \in \mathbb{Z}_{2^k}^n$, return $\mathcal{S}(\boldsymbol{x}) = z$ to P_2.

Figure 15: Functionality for evaluating SVMs

XI. Memory Usage

Supplementary to our peformance benchmarks for the online and preprocessing phases, we also measured peak memory usage.

For the online phase, none of our benchmarks exceeded 6 GB in memory usage; this is well below the available RAM in our experimental setup (16 GB) and our max. heap size JVM setting (15 GB). As such, memory does not present a bottle-neck. Peak memory usage was consistently lower for SPD$\mathbb{Z}_{2^k}$ than SPDZ (up to a 1.4 factor improvement) across all benchmarks.

For the offline phase, SPD$\mathbb{Z}_{2^k}$ and MASCOT memory usage was below half a gigabyte. While Overdrive's memory usage was much higher ($\tilde{4}$GB for $k = 64$) it still fell well below the available RAM (16GB) of our machines.

Protocol Π_{ArgMax} [49, 13.1.1]

Computes $\mathsf{ArgMax}([c_1], \ldots, [c_q]) \rightarrow (([f_1], \ldots, [f_q]), [g])$ where $f_1, \ldots, f_q \in \{0, 1\}$ and $g = \max(c_1, \ldots, c_q)$.

1) If $q = 2$:
   a) $[d_1] = \Pi_{\mathsf{LT}}([c_2], [c_1])$ and $[d_2] = 1 - [d_1]$ and $[g] = [d_1] \cdot ([c_1] - [c_2]) + [c_2]$.
   b) Return $(([d_1], [d_2]), [g])$.
2) Else if $q = 3$:
   a) $[d_1'] = \Pi_{\mathsf{LT}}([c_2], [c_1])$ and $[g'] = [d_1'] \cdot ([c_1] - [c_2]) + [c_2]$.
   b) $[d_2'] = \Pi_{\mathsf{LT}}([c_3], [g'])$ and $[g] = [d_2'] \cdot ([g'] - [c_3]) + [c_3]$.
   c) $[d_1] = [d_1'] \cdot [d_2']$ and $[d_2] = [d_2'] - [d_1'] \cdot [d_2']$ and $[d_3] = 1 - [d_1] - [d_2]$.
   d) Return $(([d_1], [d_2], [d_3]), [g])$.
3) Else, let

$$\left(([d_1], \ldots, [d_{\lfloor q/2 \rfloor}]), [g_1]\right) \leftarrow \mathsf{ArgMax}([c_1], \ldots, [c_{\lfloor q/2 \rfloor}]),$$

$$\left(([d_{\lfloor q} 2 \rfloor + 1], \ldots, [d_q]), [g_2]\right) \leftarrow \mathsf{ArgMax}([c_{\lfloor q/2 \rfloor + 1}], \ldots, [c_q]):$$

   a) $[d] = \Pi_{\mathsf{LT}}([g_2], [g_1])$ and $[g] = [d] \cdot ([g_1] - [g_2]) + [g_2]$.
   b) $[f_j] = [d] \cdot [d_j]$ for $j \in [1, \lfloor q/2 \rfloor]$ and $[f_j] = (1 - [d]) \cdot [d_j]$ for $j \in [\lfloor q/2 \rfloor + 1, q]$.
   c) Return $(([f_1], \ldots, [f_q]), [g])$.

Figure 16: Protocol for finding largest element

Protocol Π_{SVM}

INPUT: $\mathcal{S} = (\boldsymbol{F}, \boldsymbol{b})$ with $\boldsymbol{F} = (F_1, \ldots, F_q) \in \mathbb{Z}_{2^k}^{q \times n}, \boldsymbol{b} = (b_1, \ldots, b_q) \in \mathbb{Z}_{2^k}^q$ from P_1 and $\boldsymbol{x} = (x_1, \ldots, x_n) \in \mathbb{Z}_{2^k}^n$ from P_2.

1) P_1 inputs the values $\boldsymbol{F}, \boldsymbol{b}$ using $\mathsf{Input}(\cdot, P_1)$ to get sharings $[\boldsymbol{F}] = ([F_1], \ldots, [F_q])$ and $[\boldsymbol{b}] = ([b_1], \ldots, [b_q])$ such that $([f_{j,1}], \ldots, [f_{j,n}]) = [F_j]$ for all $j \in [1, q]$.
2) P_2 inputs the values $(x_1, \ldots, x_n) = \boldsymbol{x}$ using $\mathsf{Input}(\cdot, P_2)$ to get sharings $[x_1], \ldots, [x_n]$.

OUTPUT: P_1 learns nothing and P_2 learns $z \in \mathbb{Z}_{2^k}$.

1 For all $i \in [1, n]$ and all $j \in [1, q]$ the parties compute $[c_j] = [b_i] + \sum_{i \in [1,n]} [f_{j,i}] \cdot [x_i]$ for all $j \in [1, q]$.
2 Use Π_{ArgMax} in 16 to compute $(([f_1], \ldots, [f_q]), [g]) \leftarrow \mathsf{ArgMax}([c_1], \ldots, [c_q])$.
3 Compute $f_j \leftarrow \mathsf{Open}([f_j])$ for all $j \in [1, q]$ and return the value $j \cdot f_j \neq 0$.

Figure 17: Protocol for evaluating SVMs

XII. Proofs of Some Propositions

Proof of Lemma III.1: Suppose the adversary causes incorrect values $x_1', \ldots, x_m', r_1', \ldots, r_\sigma'$ to be authenticated in the $[\cdot]_2$ sharings, and write $x_i' = x_i + \delta_i \bmod 2$ and $r_i' = r_i + \epsilon_i \bmod 2$. If all consistency checks pass then from the j-th check we have $\epsilon_j + \sum_{i=1}^m \chi_{i,j} \cdot \delta_i = 0$. Since each $\chi_{i,j}$ is uniformly random and independent of δ_i, ϵ_j, if any $\delta_i \neq 0$ then this holds with probability at most $1/2$ for a single j. Taking all σ checks into account, it follows that the outputs are correct with probability at least $1 - 2^{-\sigma}$. ∎

Proof of Lemma IV.1: It is clear that $x^2 \equiv_\ell 1$ if and only if $(x-1)(x+1) \equiv_\ell 0$ so 2^ℓ divides $(x-1)(x+1)$. The case $x = \pm 1$ is trivial, so we may assume that $x \neq \pm 1$. Let 2^u and 2^v be the largest power of 2 dividing the non-zero

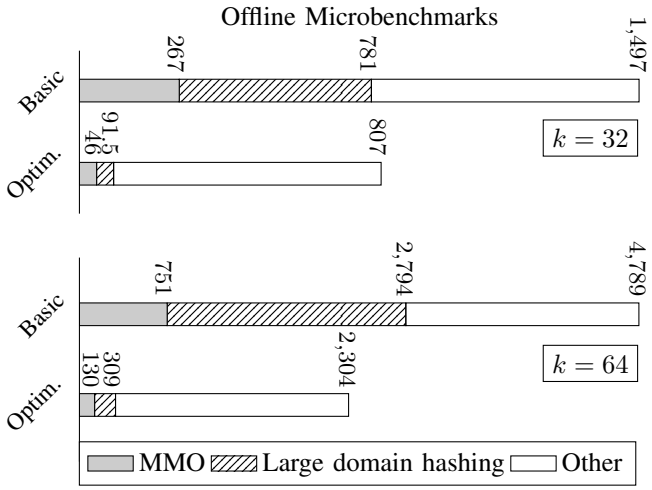


Figure 18: Time in milliseconds for offline preprocessing of 10,000 triples with and without MMO and large domain hashing optimizations with a reduced number of consistency checks in the correlated OTs ($\mu = 3$), running on a single thread on a LAN supporting up to 10 Gbps.

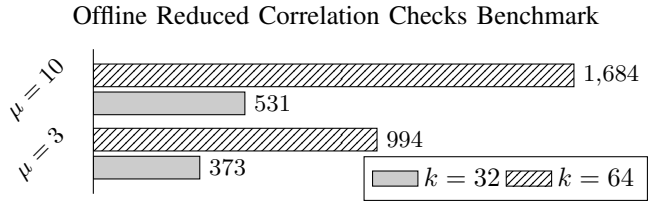


Figure 19: Time in milliseconds for preprocessing 10,000 multiplications running on 4 threads on a 10 Gbps network with difference amounts of correlation checks with our MMO and large domain hashing optimizations applied.

integers $x-1$ and $x+1$ respectively, then 2^{u+v} is the largest power of 2 dividing x^2-1, so $u+v \geq \ell$. On the other hand, since $2^{\min(u,v)}$ divides both $x+1$ and $x-1$, it also divides $(x+1)-(x-1) = 2$ and therefore $\min(u,v) \in \{0,1\}$. Thus, either $u=0$, $u=1$, $v=0$ or $v=1$. If $u=0$ then $v \geq \ell$ and therefore $x+1 \equiv_\ell 0$. If $u=1$ then $v \geq \ell-1$, so $x+1 \equiv_{\ell-1} 0$. A similar analysis follows if $v=0$ or $v=1$, which finishes the proof. ∎

Proof of Proposition IV.1: We begin by arguing correctness. In step 5 the value $e = a^2 \bmod 2^{k+2}$ is opened. Then, notice that $(c^{-1}a)^2 \equiv_{k+2} c^{-2}a^2 \equiv_{k+2} e^{-1}e \equiv_{k+2} 1$, so $c^{-1}a \equiv_{k+1} \pm 1$ due to Lemma IV.1. Moreover, since c is taken to be the smallest square root of e and a is one of such roots, which was chosen at random, we conclude that $c^{-1}a$ is congruent modulo 2^{k+1} to either -1 or $+1$ with equal probability. Clearly, this implies that $c^{-1}a+1$ is congruent modulo 2^{k+1} to either 0 or 2 with equal probability.

Now, the key point to observe is that, due to the fact that $[a]$ was computed as $2[u]+1$, and due to the way the addition of shared values in SPD$\mathbb{Z}_{2^k}$ works (see Section II-B), we have that both the shares of $[a]$ and the shares of its MAC corresponding to party P_j are even for $j > 1$, and odd for $j = 1$. Then, since c^{-1} is odd it follows by a similar argument that $d^j, t^j \in \mathbb{Z}_{2^{k+s+1}}$, the shares of the value $[c^{-1}a+1]$ and its MAC, are all even, so the division by 2 used in the protocol is well defined. Notice that $\sum_j d^j \equiv_{k+2} c^{-1}a+1$ and $\sum_j t^j \equiv_{k+s+1} \alpha \cdot \left(\sum_j d^j\right)$, so $\sum_j \frac{d^j}{2} \equiv_{k+1} \frac{c^{-1}a+1}{2}$ and $\sum_j \frac{t^j}{2} \equiv_{k+s} \alpha \cdot \left(\sum_j \frac{d^j}{2}\right)$. In particular, $\sum_j b^j$ is congruent modulo 2^k to 0 or 1 with equal probability, so b^j, h^j indeed define authenticated shares of a bit over $\mathbb{Z}_{2^k}$, with shares over $\mathbb{Z}_{2^{k+s}}$, as desired.

As for security, notice that the only possibility of an attack is that the adversary causes a selective abort, therefore biasing the resulting bit. Fortunately this is not possible since, right before step 5, the value a looks uniformly random to the adversary. ∎

Proof of Proposition IV.2: To argue correctness, we begin by showing that $a' = a \bmod 2^{k-1}$. To see this, notice that $a \equiv_{k-1} c'-r'$, and that $c'-r' \in \{-2^{k-1}, \ldots, 2^{k-1}-1\}$. Therefore, $c'-r'$ is the remainder of a when divided by 2^{k-1} if and only if $c'-r' \geq 0$, and otherwise it is equal to this remainder, minus 2^{k-1}. This can be written in a more compact way as $c'-r' = (a \bmod 2^{k-1}) - u2^{k-1}$, where $u = c' \overset{?}{<} r'$, which implies that $a \bmod 2^{k-1} = c'-r'+u2^{k-1} = a'$. We can see then that $d = a-a' = 2^{k-1}a_{k-1}$, so $e = 2^{k-1}(a_{k-1} \oplus b)$ and therefore $e_{k-1} = a_{k-1} \oplus b$. Thus, a_{k-1} can be computed as $a_{k-1} = e_{k-1} \oplus b = e_{k-1}+b-2e_{k-1}b$.

Finally, to argue security we show that none of the opened values reveal anything about a. The value c does not leak information since the random mask r is used. On the other hand, we saw above that $d = 2^{k-1}a_{k-1}$, so the mask $2^{k-1}b$ completely hides this value when e is opened. ∎

Acknowledgments

This work has been supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme under grant agreement No 669255 (MPCPRO); the European Union's Horizon 2020 research and innovation programme under grant agreement No 731583 (SODA); the European Union's Horizon 2020 research and innovation programme under grant agreement No 74079 (ALGSTRONGCRYPTO); and the Danish Independent Research Council under Grant-ID DFF–6108-00169 (FoCC).

We would like to thank the authors of [29] and [30] for providing us with their machine learning models.

References

[1] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *42nd FOCS*. Las Vegas, NV, USA: IEEE Computer Society Press, Oct. 14–17, 2001, pp. 136–145.

[2] J. Katz, S. Ranellucci, M. Rosulek, and X. Wang, "Optimizing authenticated garbling for faster secure two-party computation," in *CRYPTO 2018, Part III*, ser. LNCS, H. Shacham and A. Boldyreva, Eds., vol. 10993. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 19–23, 2018, pp. 365–391.

[3] K. Chida, D. Genkin, K. Hamada, D. Ikarashi, R. Kikuchi, Y. Lindell, and A. Nof, "Fast large-scale honest-majority MPC for malicious adversaries," in *CRYPTO 2018, Part III*, ser. LNCS, H. Shacham and A. Boldyreva, Eds., vol. 10993. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 19–23, 2018, pp. 34–64.

[4] N. Döttling, S. Ghosh, J. B. Nielsen, T. Nilges, and R. Trifiletti, "TinyOLE: Efficient actively secure two-party computation from oblivious linear function evaluation," in *ACM CCS 2017*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. Dallas, TX, USA: ACM Press, Oct. 31 – Nov. 2, 2017, pp. 2263–2276.

[5] O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game or A completeness theorem for protocols with honest majority," in *19th ACM STOC*, A. Aho, Ed. New York City, NY, USA: ACM Press, May 25–27, 1987, pp. 218–229.

[6] D. Beaver, S. Micali, and P. Rogaway, "The round complexity of secure protocols (extended abstract)," in *22nd ACM STOC*. Baltimore, MD, USA: ACM Press, May 14–16, 1990, pp. 503–513.

[7] J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra, "A new approach to practical active-secure two-party computation," in *CRYPTO 2012*, ser. LNCS, R. Safavi-Naini and R. Canetti, Eds., vol. 7417. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 19–23, 2012, pp. 681–700.

[8] I. Damgård, J. B. Nielsen, M. Nielsen, and S. Ranellucci, "The TinyTable protocol for 2-party secure computation, or: Gate-scrambling revisited," in *CRYPTO 2017, Part I*, ser. LNCS, J. Katz and H. Shacham, Eds., vol. 10401. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 20–24, 2017, pp. 167–187.

[9] M. Ben-Or, S. Goldwasser, and A. Wigderson, "Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract)," in *20th ACM STOC*. Chicago, IL, USA: ACM Press, May 2–4, 1988, pp. 1–10.

[10] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias, "Multiparty computation from somewhat homomorphic encryption," in *CRYPTO 2012*, ser. LNCS, R. Safavi-Naini and R. Canetti, Eds., vol. 7417. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 19–23, 2012, pp. 643–662.

[11] M. Keller, E. Orsini, and P. Scholl, "MASCOT: Faster malicious arithmetic secure computation with oblivious transfer," in *ACM CCS 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. Vienna, Austria: ACM Press, Oct. 24–28, 2016, pp. 830–842.

[12] I. Damgård and M. Keller, "Secure multiparty AES," in *FC 2010*, ser. LNCS, R. Sion, Ed., vol. 6052. Tenerife, Canary Islands, Spain: Springer, Heidelberg, Germany, Jan. 25–28, 2010, pp. 367–374.

[13] I. Damgård and R. W. Zakarias, "Fast oblivious AES A dedicated application of the MiniMac protocol," in *AFRICACRYPT 16*, ser. LNCS, D. Pointcheval, A. Nitaj, and T. Rachidi, Eds., vol. 9646. Fes, Morocco: Springer, Heidelberg, Germany, Apr. 13–15, 2016, pp. 245–264.

[14] M. Keller, E. Orsini, D. Rotaru, P. Scholl, E. Soria-Vazquez, and S. Vivek, "Faster secure multi-party computation of AES and DES using lookup tables," in *ACNS 17*, ser. LNCS, D. Gollmann, A. Miyaji, and H. Kikuchi, Eds., vol. 10355. Kanazawa, Japan: Springer, Heidelberg, Germany, Jul. 10–12, 2017, pp. 229–249.

[15] I. Damgård, K. Damgård, K. Nielsen, P. S. Nordholt, and T. Toft, "Confidential benchmarking based on multiparty computation," in *FC 2016*, ser. LNCS, J. Grossklags and B. Preneel, Eds., vol. 9603. Christ Church, Barbados: Springer, Heidelberg, Germany, Feb. 22–26, 2016, pp. 169–187.

[16] D. Demmler, T. Schneider, and M. Zohner, "ABY - A framework for efficient mixed-protocol secure two-party computation," in *NDSS 2015*. San Diego, CA, USA: The Internet Society, Feb. 8–11, 2015.

[17] P. Mohassel and P. Rindal, "ABY^3: A mixed protocol framework for machine learning," in *ACM CCS 2018*, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds. Toronto, ON, Canada: ACM Press, Oct. 15–19, 2018, pp. 35–52.

[18] R. Cramer, I. Damgård, D. Escudero, P. Scholl, and C. Xing, "SPD $\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority," in *CRYPTO 2018, Part II*, ser. LNCS, H. Shacham and A. Boldyreva, Eds., vol. 10992. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 19–23, 2018, pp. 769–798.

[19] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart, "Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits," in *ESORICS 2013*, ser. LNCS, J. Crampton, S. Jajodia, and K. Mayes, Eds., vol. 8134. Egham, UK: Springer, Heidelberg, Germany, Sep. 9–13, 2013, pp. 1–18.

[20] I. Damgård, M. Fitzi, E. Kiltz, J. B. Nielsen, and T. Toft, "Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation," in *TCC 2006*, ser. LNCS, S. Halevi and T. Rabin, Eds., vol. 3876. New York, NY, USA: Springer, Heidelberg, Germany, Mar. 4–7, 2006, pp. 285–304.

[21] T. Nishide and K. Ohta, "Multiparty computation for interval, equality, and comparison without bit-decomposition protocol," in *PKC 2007*, ser. LNCS, T. Okamoto and X. Wang, Eds., vol. 4450. Beijing, China: Springer, Heidelberg, Germany, Apr. 16–20, 2007, pp. 343–360.

[22] O. Catrina and S. de Hoogh, "Improved primitives for secure multiparty integer computation," in *SCN 10*, ser. LNCS, J. A. Garay and R. D. Prisco, Eds., vol. 6280. Amalfi, Italy: Springer, Heidelberg, Germany, Sep. 13–15, 2010, pp. 182–199.

[23] Alexandra Institute, "FRESCO - a FRamework for Efficient Secure COmputation," https://github.com/aicis/fresco.

[24] University of Bristol, "SPDZ-2," https://github.com/bristolcrypto/SPDZ-2.

[25] M. Keller, V. Pastro, and D. Rotaru, "Overdrive: Making SPDZ great again," in *EUROCRYPT 2018, Part III*, ser. LNCS, J. B. Nielsen and V. Rijmen, Eds., vol. 10822. Tel Aviv, Israel: Springer, Heidelberg, Germany, Apr. 29 – May 3, 2018, pp. 158–189.

[26] X. Wang, S. Ranellucci, and J. Katz, "Global-scale secure multiparty computation," in *ACM CCS 2017*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. Dallas, TX, USA: ACM Press, Oct. 31 – Nov. 2, 2017, pp. 39–56.

[27] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar, "Chameleon: A hybrid secure computation framework for machine learning applications," in *ASIACCS 18*, J. Kim, G.-J. Ahn, S. Kim, Y. Kim, J. López, and T. Kim, Eds. Incheon, Republic of Korea: ACM Press, Apr. 2–6, 2018, pp. 707–721.

[28] R. Cramer, I. Damgård, and Y. Ishai, "Share conversion, pseudorandom secret-sharing and applications to secure computation," in *TCC 2005*, ser. LNCS, J. Kilian, Ed., vol. 3378. Cambridge, MA, USA: Springer, Heidelberg, Germany, Feb. 10–12, 2005, pp. 342–362.

[29] M. D. Cock, R. Dowsley, C. Horst, R. Katti, A. Nascimento, W. Poon, and S. Truex, "Efficient and private scoring of decision trees, support vector machines and logistic regression models based on pre-computation," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1, 2018.

[30] E. Makri, D. Rotaru, N. P. Smart, and F. Vercauteren, "EPIC: Efficient private image classification (or: Learning from the masters)," in *CT-RSA 2019*, ser. LNCS, M. Matsui, Ed., vol. 11405. San Francisco, CA, USA: Springer, Heidelberg, Germany, Mar. 4–8, 2019, pp. 473–492.

[31] P. Mohassel and Y. Zhang, "SecureML: A system for scalable privacy-preserving machine learning," in *2017 IEEE Symposium on Security and Privacy*. San Jose, CA, USA: IEEE Computer Society Press, May 22–26, 2017, pp. 19–38.

[32] H. Lipmaa and T. Toft, "Secure equality and greater-than tests with sublinear online complexity," in *ICALP 2013, Part II*, ser. LNCS, F. V. Fomin, R. Freivalds, M. Z. Kwiatkowska, and D. Peleg, Eds., vol. 7966. Riga, Latvia: Springer, Heidelberg, Germany, Jul. 8–12, 2013, pp. 645–656.

[33] D. Bogdanov, S. Laur, and J. Willemson, "Sharemind: A framework for fast privacy-preserving computations," in *ESORICS 2008*, ser. LNCS, S. Jajodia and J. López, Eds., vol. 5283. Málaga, Spain: Springer, Heidelberg, Germany, Oct. 6–8, 2008, pp. 192–206.

[34] M. Pettai and P. Laud, "Automatic proofs of privacy of secure multi-party computation protocols against active adversaries," in *IEEE Computer Security Foundations Symposium*, C. Fournet, M. W. Hicks, and L. Viganò, Eds. IEEE Computer Society, 2015, pp. 75–89.

[35] I. Damgård, C. Orlandi, and M. Simkin, "Yet another compiler for active security or: Efficient MPC over arbitrary rings," in *CRYPTO 2018, Part II*, ser. LNCS, H. Shacham and A. Boldyreva, Eds., vol. 10992. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 19–23, 2018, pp. 799–829.

[36] T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara, "High-throughput semi-honest secure three-party computation with an honest majority," in *ACM CCS 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. Vienna, Austria: ACM Press, Oct. 24–28, 2016, pp. 805–817.

[37] X. Wang, S. Ranellucci, and J. Katz, "Authenticated garbling and efficient maliciously secure two-party computation," in *ACM CCS 2017*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. Dallas, TX, USA: ACM Press, Oct. 31 – Nov. 2, 2017, pp. 21–37.

[38] G. Couteau, "New protocols for secure equality test and comparison," in *ACNS 18*, ser. LNCS, B. Preneel and F. Vercauteren, Eds., vol. 10892. Leuven, Belgium: Springer, Heidelberg, Germany, Jul. 2–4, 2018, pp. 303–320.

[39] D. J. Wu, T. Feng, M. Naehrig, and K. E. Lauter, "Privately evaluating decision trees and random forests," *PoPETs*, vol. 2016, no. 4, pp. 335–355, 2016.

[40] M. Joye and F. Salehi, "Private yet efficient decision tree evaluation," in *Data and Applications Security and Privacy XXXII - IFIP WG*, ser. Lecture Notes in Computer Science, F. Kerschbaum and S. Paraboschi, Eds., vol. 10980. Springer, 2018, pp. 243–259.

[41] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, B. Grégoire, V. Laporte, and V. Pereira, "A fast and verified software stack for secure function evaluation," in *ACM CCS 2017*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. Dallas, TX, USA: ACM Press, Oct. 31 – Nov. 2, 2017, pp. 1989–2006.

[42] P. Scholl, "Extending oblivious transfer with low communication via key-homomorphic PRFs," in *PKC 2018, Part I*, ser. LNCS, M. Abdalla and R. Dahab, Eds., vol. 10769. Rio de Janeiro, Brazil: Springer, Heidelberg, Germany, Mar. 25–29, 2018, pp. 554–583.

[43] S. M. Matyas, C. H. Meyer, and J. Oseas, "Generating strong one-way functions with cryptographic algorithm," *IBM Technical Disclosure Bulletin*, vol. 27, no. 10A, pp. 5658–5659, 1985.

[44] G. Asharov, Y. Lindell, T. Schneider, and M. Zohner, "More efficient oblivious transfer extensions with security for malicious adversaries," in *EUROCRYPT 2015, Part I*, ser. LNCS, E. Oswald and M. Fischlin, Eds., vol. 9056. Sofia, Bulgaria: Springer, Heidelberg, Germany, Apr. 26–30, 2015, pp. 673–701.

[45] N1 Analytics, "MP-SPDZ - Versatile framework for multi-party computation," https://github.com/n1analytics/MP-SPDZ.

[46] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," 2009, https://www.cs.toronto.edu/~kriz/cifar.html.

[47] A. Quattoni and A. Torralba, "Recognizing indoor scenes," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. IEEE Computer Society, 2009, pp. 413–420.

[48] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *IEEE Conference on Computer Vision and Pattern Recognition*. IEEE Computer Society, 2016, pp. 2818–2826.

[49] T. Toft, "Primitives and applications for multi-party computation," Ph.D. dissertation, Aarhus University, 2007.