Rabbit: Efficient Comparison for Secure Multi-Party Computation

Eleftheria Makri[1,5], Dragos Rotaru[2,1], Frederik Vercauteren[1], and Sameer Wagh[3,4]

[1] imec-COSIC, KU Leuven, Belgium
[2] Cape Privacy
[3] University of California, Berkeley, USA
[4] Princeton University, Princeton, USA
[5] ABRR, Saxion University of Applied Sciences, The Netherlands
emakri@esat.kuleuven.be, dragos@capeprivacy.com,
frederik.vercauteren@kuleuven.be, swagh@berkeley.edu

Abstract. Secure comparison has been a fundamental challenge in privacy-preserving computation, since its inception as Yao's millionaires' problem (FOCS 1982). In this work, we present a novel construction for general n-party private comparison, secure against an active adversary, in the dishonest majority setting. For the case of comparisons over fields, our protocol is more efficient than the best prior work (edaBits: Crypto 2020), with $\sim 1.5\times$ better throughput in most adversarial settings, over $2.3\times$ better throughput in particular in the passive, honest majority setting, and lower communication. Our comparisons crucially eliminate the need for bounded inputs as well as the need for statistical security that prior works require. An important consequence of removing this "slack" (a gap between the bit-length of the input and the MPC representation) is that multi-party computation (MPC) protocols can be run in a field of smaller size, reducing the overhead incurred by privacy-preserving computations. We achieve this novel construction using the commutative nature of addition over rings and fields. This makes the protocol both simple to implement and highly efficient and we provide an implementation in MP-SPDZ (CCS 2020).

Keywords: Secure Comparison·Multi-party Computation·Unconditional Security·Dishonest Majority.

1 Introduction

After years of active research, both in theoretical results and system building, multi-party computation (MPC) is becoming practical as a paradigm. Recent research results and practical implementations [13,1], deployment of MPC in real-life applications [3], as well as organizations beyond academia offering commercial MPC solutions [30,27,26], confirm that MPC is reaching maturity. However, MPC, just like any other cryptographic primitive deployed to enhance privacy, comes at a significant efficiency penalty, in terms of computation and communication. While some research focuses on tailoring MPC solutions to a particular problem, to compensate for this

efficiency penalty, other works focus on improving the efficiency of fundamental MPC building blocks, which are applicable to a wide variety of problems.

Secure comparison is an important problem in multi-party computation – it involves the comparison of two or more secret values in a privacy-preserving manner. Comparison is a fundamental building block, necessary for the realization of various larger tasks: from online auctions to big data analytics and machine learning. Given the privacy considerations that today's digital infrastructure entails, protocols for secure comparison are a fundamental MPC tool in privacy-preserving applications.

Since the introduction of the secure comparison problem by Andrew Yao in 1982 [34] as the millionaires' problem, research efforts have pushed the frontiers of performance of this primitive. MPC has traditionally been efficient either on linear operations, when it is based on arithmetic circuits, or on non-linear operations, when it is based on Boolean circuits. Recent applications require a combination of linear and non-linear operations, and they are most of the time addressed with solutions based on arithmetic circuits, because these are significantly more efficient than Boolean circuits for the linear part, which presents itself as the bulk of the computation. Given the non-linear nature of the comparison operation, protocols for secure comparison still remain a bottleneck for privacy-preserving computation. Thus, any improvement in this line of work has a compounding impact on improving the overall efficiency of privacy-preserving computations.

In this work, we present a novel comparison protocol that is secure against an active adversary in the dishonest majority setting and holds for general n-party computation. Our work improves upon the state-of-the-art protocol for comparison in dishonest majority in both the total time and communication by a factor of two for the OT-based preprocessing. In addition, our protocol is easy to implement requiring no heavy cryptography. Notably, our protocol is highly conducive to amortization and preprocessing, which makes it attractive for deployment in real-life applications, as these are important considerations in building practical secure systems.

1.1 Our Contribution

We present Rabbit[1], a novel secure comparison protocol, which leverages the commutative nature of addition over rings and fields. Our protocol exploits recent advances in the generation and deployment of *doubly authenticated shared bits* (daBits [25]), which are bits living both in $\mathbb{F}_p$ and in $\mathbb{F}_{2^k}$, as well as *extended doubly authenticated bits* (edaBits [14]), which correspond to shared integers in the arithmetic domain, whose bit decomposition is shared in the binary domain. The proposed comparison is more efficient than previously proposed secure comparison protocols, while at the same time removing the dependence on bounds and statistical parameters. This allows the MPC engines used for our secure comparison to be smaller than the ones required by previous protocols, which has a positive impact on the concrete efficiency of the MPC protocols. Concretely we make the following contributions:

(i) **Novel comparison protocol:** We propose Rabbit, a novel secure comparison protocol based on the commutative nature of addition over rings and fields.

[1] The name is an extension of the daBit [25], maBit [24] and edaBit [14] line of work.

Rabbit is a general n-party protocol and crucially eliminates the need for any "slack" – a statistically larger dataspace to ensure security of computations, and thus enables computations over smaller datatypes. For instance, to compute over 64-bits, prior works require the use of 128-bit datatypes, while we can support these computations in standard 64-bit datatypes.

(ii) **Security:** Since we eliminate the slack and keep an exact tab of overflows, our protocols are unconditionally secure even against active adversaries in the dishonest majority setting. In the case of comparison over fields, we do have to account for a statistical security parameter, because of the existing implementation of edaBits [14]. In general, when implemented in a larger body of MPC computation, our comparison inherits the security properties of the platform, such as statistical security when using MP-SPDZ [13].

(iii) **Simplicity and Efficiency:** Our protocol is straightforward to implement. As shown in Fig. 1b, it is merely a few lines of code in MP-SPDZ. This also makes our protocol highly amenable to secure implementation. As for efficiency, the benefits of our work over the state-of-the-art are most pronounced in the case of comparison over fields. In this case, we improve end-to-end computations such as secure evaluation of ResNet-50 up to 2x faster, albeit at a higher communication.

1.2 Technical Overview

Our central focus in this work is to propose novel and efficient protocols for secure comparison. Comparison protocols usually rely on statistical security or bit-decomposition combined with prefix computation to achieve the results. We observe that:

(i) When considering arithmetic secret shares, the bit encoding modulus overflow of secrets enables exact integer relations between the secret, the secret shares, and the modulus.

(ii) Using the commutativity of addition over standard structures, such as rings and fields, we can express a sum in two different ways and thus equate the corresponding constraint equations.

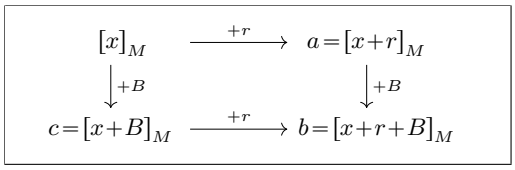
These two observations together enable more efficient protocols for comparisons. More specifically, the core idea behind our comparison protocols lies in our ability to detect when a sum over a particular modulus overflows (i.e., wraps around), and when this happens we can correct it. Observe that given two integers $x,y \in \mathbb{Z}_M$, their sum $x+y \bmod M$ is less than either of the two summands, iff the sum wrapped around the modulus. That is, given a comparison function:

$$\mathsf{LT}(\cdot,\cdot):\mathbb{Z}\times\mathbb{Z}\to\{0,1\} \subseteq \mathbb{Z} : \begin{cases} \mathsf{LT}(x,y)=1 & \text{if } (x<y); \\ \mathsf{LT}(x,y)=0 & \text{otherwise,} \end{cases}$$

we can compute the modular sum $x+y \bmod M$, by performing computations over the integers as:

$$x+y \bmod M = x+y-M\cdot\mathsf{LT}(x+y \bmod M,x) = x+y-M\cdot\mathsf{LT}(x+y \bmod M,y)$$

This is due to the observation that $\mathsf{LT}(x+y \bmod M,x)$ (or $\mathsf{LT}(x+y \bmod M,y)$) is true, iff the sum wrapped around. Given that the $\mathsf{LT}(\cdot,\cdot)$ function detects (i.e., outputs



(a) Intuition behind Rabbit comparison protocol

```
# Secure comparison in MP-SPDZ
k = program.bit_length
r, r_bits = sint.get_edabit(k, True)
a = (x+r).reveal()
b = a + M - R

w1 = LTBits(a, r_bits, k)
w2 = LTBits(b, r_bits, k)
w3 = (b < M - R)

movs(s, sint.conv(w1-w2+w3))
return
```

(b) Rabbit code snippet

Fig. 1: Our protocol relies on the commutative properties of addition over rings/fields as shown in Fig. 1a. This diagram indicates the two different ways we can obtain the value b. The $[\cdot]_M$ notation indicates that the corresponding values or sums are taken modulo M. The horizontal arrows indicate addition of a uniformly random value $r \in \{0,...,M-1\}$, used to mask the secret input of the comparison x (so that we can later open it without information leakage, to perform a comparison). The vertical arrows indicate addition of a known constant $B \in \{0,...,M-1\}$ related to the public quantity to be compared against. These two ways of computing the sum b, are necessary for the comparison protocol between a secret value x and a public constant $M-B$. The code on the right (Fig. 1b) shows the simplicity of implementing our protocol, implemented in this case in the MP-SPDZ codebase [13].

true) when a wrap around happens, we can indeed realize the modular sum, while performing computations over the integers, by conditionally subtracting the quantity of the wrap around (i.e., M), when $\mathsf{LT}(\cdot,\cdot)$ returns true.

Notation. We use $[x]_N$ to denote the sharing of a secret x in the ring $\mathbb{Z}_N$. We primarily consider two values of the modulus: $N=M$ and $N=2$, where M is a fixed constant, set to either a prime p, or a power of two 2^k. The types of sharings are:

(i) $[x]_M$, where the secret is $x \in \mathbb{Z}_M$ or $[b]_2$, where the secret is a bit $b \in \mathbb{F}_2$;
(ii) $[x]_M$ and $[x_0]_2,...[x_{m-1}]_2$ such that $x = \sum_{i=0}^{m-1} x_j \cdot 2^j \pmod M$ and $M < 2^m$ (this is also known as an edaBit [14])

Similarly, given a (public) constant value $R \in \mathbb{Z}_M$, we denote by $R_0,...,R_{m-1}$ the bit decomposition of R, and by R_i its individual bits (at the corresponding position i).

2 Comparison Protocols

In this section we present our comparison protocols and their workings on a step-by-step basis. Then, for each presented protocol, we also show correctness. We do not provide any formal proofs of security of our protocols, as these follow trivially from the arithmetic black box functionality paradigm [11]. We present the protocols in the following order:

(i) First, we present the protocol Π_{LTBits} (Fig. 3), which realizes a comparison between a secret bit-decomposed value, and a public value, and outputs a secret

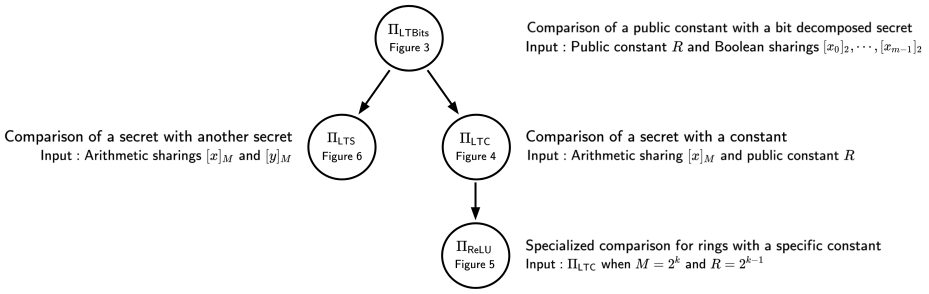


Fig. 2: Proposed comparison protocols, their inputs, and their interdependencies.

bit indicating the result of the comparison. This is a building block that uses prefix computation for comparison.

(ii) Second, we introduce the protocol Π_{LTC} (Fig. 4), which invokes Π_{LTBits} and performs a comparison between a secret value (without bit-decomposition), and a public value, where the output is a secret bit indicating the result of the comparison.

(iii) Third, we present a specialized comparison protocol, Π_{ReLU} (Fig. 5), that can be applied when the modulus is a power of 2 and the public constant against which we compare is half the modulus. Note that this is an important case, as it corresponds to computation of the ReLU function, which is widely used in machine learning.

(iv) Finally, in Π_{LTS} (Fig. 6), we show how to generalize Π_{LTC} to compare two secret shared values, where once again the output is a secret bit.

Note that given our novel approach of comparison, there is a difference between secret-public constant comparison (Π_{LTC}) and secret-secret comparison (Π_{LTS}), which often comes for free when using standard techniques that require a slack. For more details on this, we refer the reader to Section 4. Finally, for all proposed protocols, the output can either be an element of $\mathbb{Z}_M$ or $\mathbb{F}_2$ (depending on the needs of the follow-up computations) indicating the result of the comparison. An overview of all our comparison protocols, their inputs, and their interdependencies is given in Fig. 2.

2.1 Comparison with Bitwise Shared Input – LTBits Protocol

The protocol Π_{LTBits}, listed in Fig. 3, follows a standard bit decomposition idea to privately compute a secret bit, indicating the result of a comparison. It is essentially an adaptation of the BIT-LT protocol by Damgård *et. al.* [9], which instead of two secret bit-decomposed inputs (that BIT-LT receives), it receives one bitwise secret shared input and a public arithmetic value to compare upon, while its output is a secret Boolean value indicating the result of the comparison. Notably, each component of our bit-decomposed secret lives in $\mathbb{F}_2$, unlike Damgård *et. al.*'s [9] construction, where each secret bit lives in $\mathbb{F}_p$. The protocol LTBits computes the following:

1. The XOR of each bit of the secret input $[x_i]_2$ value with the corresponding bit of the public value R_i. This results in a bit-string $[y_0]_2$,...$[y_{m-1}]_2$ with ones on all positions where the bits of the values to be compared differ.

2. A prefix OR (circuit computes for each position i of a bit vector, the OR between all previous bits in the vector up to position i. - more details in Catrina and de Hoogh [6]) of the previously computed bits $[y_i]_2$, which results in a vector $[z_i]_2$ of 0's followed by 1's with the transition from 0 to 1 occurring at the first bit where the secret and the public value differ.
3. In this step, the previous vector is converted into a vector $[w_i]_2$, $i=0,...,m-1$ of all 0's and a single 1 at the index of the first differing bit.
4. In the last step, we take the inner product between the vector $\mathbf{w}$ (which is a vector of 0's in all positions, except for the position of the first differing bit of the values to be compared) and the bits of the public value R. This inner product results in 0, if at the position of the differing bit R was 0, which further implies that x is larger than R, and it results in 1 otherwise. We have computed the value $[(x<R)]_2$, but we are actually after $[(R<x)]_2$, thus $1-[(x<R)]_2$ concludes the protocol.

Less Than Bits $\Pi_{\mathsf{LTBits}}(R,[x_0]_2,...[x_{m-1}]_2)$

Inputs: Secret value x shared bitwise, such that parties hold $[x_0]_2,...[x_{m-1}]_2$, where $x=\sum_{i=0}^{m-1} x_j \cdot 2^j$, and public value R.
Outputs: Compute the Boolean value $[c]_2=[(R\leqslant x)]_2$.
Protocol: Complete steps 1-3 for all $i\in\{0,1,...,m-1\}$

1. Parties compute $[y_i]_2=[x_i]_2\oplus R_i$.
2. Parties compute $[z_i]_2=\vee_{j=i}^{m-1}[y_j]_2$ using PrefixOR circuit.
3. Parties compute $[w_i]_2=[z_i]_2-[z_{i+1}]_2$, where $z_m=0$.
4. Output $[c]_2=1-[(x<R)]_2$, where $[(x<R)]_2=\sum_{i=0}^{m-1} R_i\cdot[w_i]_2$.

Fig. 3: Protocol for comparison between an input shared bitwise and a public value.

Correctness of Π_{LTBits}: To see the correctness of Π_{LTBits}, note the following series of observations:

1. To compare two numbers, we start from the most significant bit (MSB) and look for the first bit where the two numbers differ. This is precisely what is computed in Step 1 of Π_{LTBits}. Thus, $y_{m-1},...,y_0$ contains a series of 0's, followed by a 1, which in turn is followed by bits that are irrelevant to the comparison.
2. As a consequence, $z_{m-1},...,z_0$ contains a series of 0's followed by 1's starting at the first location where x_i and R_i differ. Let $k\in\{0,...,m-1\}$ be the largest index where $x_i\neq R_i$. Thus, $w_i=1$ iff $i=k$ and $w_i=0$ otherwise.
3. Finally, multiplying w_i by R_i ensures the following:

$$\mathsf{output}=\begin{cases}1 & \text{if } R_k=1,\ x_k=0 \text{ (implying } R>x)\\ 0 & \text{otherwise (implying } R\leqslant x)\end{cases}$$

■

2.2 Comparison with a Constant – LTC Protocol

The protocol Π_{LTC}, listed in Fig. 4, is a comparison protocol between a shared secret value, and a public constant. Unlike Π_{LTBits}, it does not require the secret input

value to be bitwise secret shared, but it invokes the protocol Π_{LTBits} twice. These two invocations can be parallelized, decreasing the total number of rounds of the comparison protocol. Π_{LTC} requires an edaBit as an input. An edaBit is a shared value in the arithmetic domain, accompanied by its own bit decomposition in the binary domain [14]. The core idea behind this comparison protocol is that addition in a ring or field is commutatitve as explained in Fig. 1a.

Less Than Constant $\Pi_{\mathsf{LTC}}([x]_M, R)$

Inputs: Value x secret shared, such that parties hold $[x]_M$, a shared $\mathsf{edaBit}([r]_M, [r_0]_2, ..., [r_{m-1}]_2)$ and public value R.
Outputs: Compute the Boolean value $[(x < R)]_2$.
Protocol:

1. Parties compute the value $[a]_M = [x + r]_M$ (and $[b]_M = [x + r + M - R]_M$).
2. Parties open the value a ($b \equiv a + M - R$ can be opened locally).
3. Parties compute the following quantities:
   - $[w_1]_2 = \Pi_{\mathsf{LTBits}}(a, [r_0]_2, ..., [r_{m-1}]_2)$.
   - $[w_2]_2 = \Pi_{\mathsf{LTBits}}(b, [r_0]_2, ..., [r_{m-1}]_2)$.
   - $w_3 = (b < M - R)$.
4. Output $[w]_2 = 1 - ([w_1]_2 - [w_2]_2 + w_3)$ or use one classical daBit to output $[w]_M$.

Fig. 4: Protocol for comparison between *an input shared in* $\mathbb{Z}_M$ *and a public value* R for any modulus M (in particular, M can be 2^k or a prime p).

The Π_{LTC} protocol proceeds as follows:

1. Using the arithmetic value $[r]_M$ of the random edaBit from the input, the parties mask the input value x, computing $[a]$.
2. $[a]$ is opened, without revealing any information about x.
3. The parties then do the following:
   (a) Invoke Π_{LTBits} to compare the masked value $[a]$ against the random edaBit (in bitwise sharing), resulting in $[w_1]_2$.
   (b) Invoke Π_{LTBits} to compare $b = [a + M - R]_M$ against the random edaBit (in bitwise sharing), resulting in $[w_2]_2$.
   (c) Compare in the clear b against the public value $B = M - R$, resulting in w_3.
4. Finally, they conclude the comparison test by computing $[w]_2 = 1 - ([w_1]_2 - [w_2]_2 + w_3)$. This equation follows from the way we exploited the commutative property of addition, and its correctness is explained in the next paragraph. The output at this step is the binary value indicating the result of the comparison, shared in $\mathbb{F}_2$. Depending on the follow-up computations in the larger MPC protocol that uses the comparison, if the next input needs to be arithmetic, a classical daBit [25] can be used to transform the representation of this bit in $\mathbb{Z}_M$.

Correctness of Π_{LTC}**:** Let us denote by $[x]$ the value of $x \in \mathbb{Z}_M$, i.e., the modular reduction in $\{0, 1, ..., M-1\}$. We are interested in securely computing the Boolean

value ($x<R$), for R a public constant. Furthermore, let $\mathsf{LT}(x,y)$ be defined as follows:

$$\mathsf{LT}(x,y)=\begin{cases}1 & \text{if } x<y\\ 0 & \text{otherwise}\end{cases} \tag{1}$$

Recall from Section 1.2 that the $\mathsf{LT}(x,y)$ function enables writing exact integer relations for the sum of two numbers as follows:

$$\begin{aligned}[x+y]&=[x]+[y]-M\cdot\mathsf{LT}([x+y],[x])\\ &=[x]+[y]-M\cdot\mathsf{LT}([x+y],[y])\end{aligned} \tag{2}$$

To be consistent with the notation followed in Fig. 1a, we define $B=M-R$, and $c=[x+B]$. We then use the commutative nature of addition to represent the sum $b=[x+r+B]$ in two different ways, as shown in Fig. 1a. Using Eq. 2 for the two additions in the top path and noting that $a,b,B\in\mathbb{Z}_M$:

$$\begin{aligned}b=[a+B]&=a+B-M\cdot\mathsf{LT}(b,B)\\ &=x+r-M\cdot\mathsf{LT}(a,r)+B-M\cdot\mathsf{LT}(b,B)\end{aligned} \tag{3}$$

Similarly, using Eq. 2 for the two additions on the bottom path , we get:

$$\begin{aligned}b=[c+r]&=c+r-M\cdot\mathsf{LT}(b,r)\\ &=x+B-M\cdot\mathsf{LT}(c,B)+r-M\cdot\mathsf{LT}(b,r)\end{aligned} \tag{4}$$

Equating the RHS of Eq. 3, and Eq. 4, we get:

$$\mathsf{LT}(a,r)+\mathsf{LT}(b,B)=\mathsf{LT}(c,B)+\mathsf{LT}(b,r) \tag{5}$$

Recall that the result we are after is $\mathsf{LT}(x,R)$, which is equivalent to $(1-\mathsf{LT}(c,B))$, since $B=M-R$, and $c=[x+B]$. Thus, from Eq. 5 we have $\mathsf{LT}(c,B)=1-(\mathsf{LT}(a,r)+\mathsf{LT}(b,B)-\mathsf{LT}(b,r))$, which is exactly what we compute in Step 4 of Π_{LTC}. Finally, to complete the proof, we reiterate that $\mathsf{LT}(c,B)=0$ iff $(x<R)$ and that $\mathsf{LT}(\cdot,\cdot)$ correctly computes the function defined by Eq. 1. ■

2.3 Π_{ReLU} – Special Case of Π_{LTC} for $R=2^{k-1}$, $M=2^k$

Π_{LTC} is a general comparison protocol for comparing against *any* public value. However, a special case of interest is when the modulus is a power of 2 and the public constant to be compared against is half the modulus. When considering privacy-preserving alternatives to machine learning, the use of fixed-point arithmetic converts the widely used $\mathsf{ReLU}(x)=\max(x,0)$ function to the above comparison, when considering such a special modulus (power of 2). In this case, where $R=2^{k-1}$ and $M=2^k$, the protocol can be optimized further to improve performance. We present this optimized protocol in Fig. 5. This comparison setting is useful in a number of privacy-preserving machine learning frameworks [22,32], where fixed point encoding transforms the ReLU function into a comparison with $R=2^{k-1}$ and $M=2^k$. In this case, we can simplify our protocol to open the masked value $a=[x+r]$ (Step 1 of the protocol), subtract the

mask r from it using a binary circuit in the secret shared domain (Steps 2, 3, 4 of the protocol), and extract the MSB of this result (Step 6). This way we are essentially extracting the MSB of x. This replaces the overhead of two invocations of Π_{LTBits} with a single invocation of a binary addition protocol (Π_{BitAdder}). The computation in Step 4 can also be used to perform comparisons when $R=2^\ell$ is another power of two, however that would require additional computation over the bits $s_{k-1},...,s_\ell$.

ReLU $\Pi_{\mathsf{ReLU}}([x]_{2^k},2^{k-1})$

Inputs: Value x secret shared, such that parties hold $[x]_{2^k}$, a shared edaBit $([r]_{2^k},[r_0]_2,...,[r_{k-1}]_2)$ and the public value 2^{k-1}.
Outputs: Compute shares $[y]_{2^k}$ where $y=x$ if $(x\leqslant 2^{k-1})$ and 0 otherwise.
Protocol:

1. Parties compute the value $[a]_{2^k}=[x+r]_{2^k}$ and open a.
2. Parties locally compute $[t_0]_2,...[t_{k-1}]_2=[1-r_0]_2,...,[1-r_{k-1}]_2$
3. Parties set $a_0,...a_{k-1}$ to be the bits of $(a+1)$.
4. $[s_0]_2,...,[s_{k-1}]_2,[s_k]_2=\Pi_{\mathsf{BitAdder}}(a_0,...,a_{k-1},[t_0]_2,...,[t_{k-1}]_2)$[a].
5. Output $[s_{k-1}]_2$ or use one classical daBit to output $[s_{k-1}]_{2^k}$ if only the derivative of ReLU is required in the computation.
6. Use one multiplication triple and output $y=[x]_{2^k}\cdot[s_{k-1}]_{2^k}$.

[a] Π_{BitAdder} is a circuit performing addition over bitwise shared values.

Fig. 5: Protocol for comparison between *an input shared in* $\mathbb{Z}_{2^k}$ and 2^{k-1}.

Correctness of Π_{ReLU}: Observe that in this special case comparison with the constant 2^{k-1} where the modulus is 2^k, the MSB of the secret input defines the result of the comparison. Our protocol essentially performs a bit decomposition of the input $[x]_{2^k}$ by masking it (using the arithmetic version of the edaBit) and then again subtracting this mask in a binary circuit (using the binary version of the edaBit). This results in the bit decomposition of x, and by extracting its MSB we conclude the comparison, and hence the computation of this ReLU function.

Remark – Optimizing Π_{ReLU}: Note that Step 4 in Figure 5 can be optimized as we only require a single bit $[s_{k-1}]_2$. In particular, this requires $\log_2 k$ rounds and $k\log_2 k$ invocations of bit-triples. This can be reduced to $\log_2 k$ rounds and $2k-2$ bit-triples by simply modifying the MSB values and using a prefix computation protocol Π_{PreOpL} (cf [6]). We modify the most significant bit of the input tuple to be (1,0) before passing to the Π_{PreOpL}. Consequently, the second element of the output tuple of the Π_{PreOpL} protocol is the carry bit $[s_{k-2}]_2$ and thus $[s_{k-1}]_2$ can be computed locally as the XOR of the MSB's of the two bits and the bit $[s_{k-2}]_2$.

2.4 Comparison with Secret – LTS Protocol

While the protocol described in Sec. 2.2 provides an efficient way to compare with a public constant, the protocol described in this section, Π_{LTS}, listed in Fig. 6, enables

the comparison of two secret values x and y. In most prior works, due to the use of a slack or bounds on inputs, the corresponding protocols for these two settings are nearly identical. In our case, the elimination of slack requires slightly different protocols. We provide a brief discussion on applications of either of these protocols in Sec. 4.2.

Less Than Secret $\Pi_{\mathsf{LTS}}([x]_M,[y]_M)$

Inputs: Values x and y secret shared, such that parties hold $[x]_M$ and $[y]_M$, two shared $\mathsf{edaBits}([r]_M,[r_0]_2,...,[r_{m-1}]_2)$ and $([r']_M,[r'_0]_2,...,[r'_{m-1}]_2)$.
Outputs: Compute the Boolean value $[(x<y)]_2$.
Protocol:

1. Parties compute the values $[b]_M=[y+r]_M$, $[a]_M=[r'-x]_M$
2. Parties open the values a and b, and compute $T\equiv a+b \pmod M$ locally.
3. Parties compute the following quantities:
   - $[w_1]_2=\Pi_{\mathsf{LTBits}}(b,[r_0]_2,...,[r_{m-1}]_2)$.
   - $[w_2]_2=\Pi_{\mathsf{LTBits}}(a,[r'_0]_2,...,[r'_{m-1}]_2)$.
   - $w_3=(T<b)$.
   - $[s_0]_2,...,[s_{m-1}]_2,[s_m]_2=\Pi_{\mathsf{BitAdder}}([r_0]_2,...,[r_{m-1}]_2,[r'_0]_2,...,[r'_{m-1}]_2)$.
   - $[w_4]_2=[s_m]_2$
   - $[w_5]_2=\Pi_{\mathsf{LTBits}}(T,[s_0]_2,...,[s_{m-1}]_2)$.
4. Output $[w]_2=[w_1]_2+[w_2]_2+w_3-[w_4]_2-[w_5]_2$, or use one classical daBit and output $[w]_M$.

Fig. 6: Protocol for comparison between *two arithmetic inputs shared in* $\mathbb{Z}_M$, for any modulus M (in particular, M can be 2^k or a prime p).

Each step of the protocol Π_{LTS} computes the following:

1. Parties mask the input values $[y]$ and $[x]$ using the arithmetic shares of two random $\mathsf{edaBits}$ $[r]$ and $[r']$, resulting in shared values $[b]$ and $[a]\in\mathbb{Z}_M$.
2. These masked values are opened (without revealing any information about x or y) and the value $T\equiv a+b \pmod M$ is computed locally.
3. The parties then perform the following computations:
   (a) Using Π_{LTBits}, a secret comparison between the open value b and the bitwise sharing of the edaBit r, and store the result $[w_1]_2$.
   (b) A similar comparison between a and the bitwise sharing of r', and store the output in $[w_2]_2$.
   (c) Check in the clear whether $(T<b)$, and store this value in w_3.
   (d) Compute a circuit for bitwise addition of two binary (secret) vectors, where the result is a bitwise secret shared vector of the bits of $(r+r')$.
   (e) Extract the last carry bit from the binary adder (Step 3d) as $[w_4]_2$.
   (f) Finally, using Π_{LTBits}, compare the value T against the bitwise secret sharing of $r+r'$ (computed in Step 3d), and store the output in $[w_5]_2$.
4. In the end, the parties conclude the comparison protocol by computing the output $[w]_2=[w_1]_2+[w_2]_2+w_3-[w_4]_2-[w_5]_2$. This final step, similarly to the LTC protocol follows from the way we exploit the commutative nature of addition,

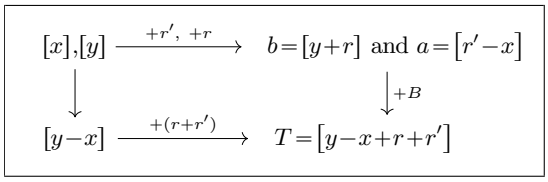


Fig. 7: Intuition behind the comparison protocol for two secret values, once again based on the commutative nature of addition over rings and fields.

and we show correctness subsequently. The final output is the binary sharing of the comparison result, which can be transformed to a shared bit in $\mathbb{Z}_M$ if needed.

Correctness of Π_{LTS}: Following the same notation set-up as in Sec. 2.2 for Π_{LTC}, we denote by $[x]$ the value of $x \in \mathbb{Z}_M$, and the function $\mathsf{LT}(x,y)$ as defined in Eq. 1. We are interested in securely computing the Boolean value $(x<y)$, for x and y two secret shared values in $\mathbb{Z}_M$. The intuition for our protocol is presented in Fig. 7 and follows the same idea as in Π_{LTC}, viz., computing a sum in two different ways and using Eq. 2 to find a constraint between the various wrappings around the modulus.

First note that $[x]<[y]$ iff $\mathsf{LT}([y-x],[y])=1$. We then mask the inputs y and $-x$ using the two edaBits: $[b]=[y+r]$, $[a]=[r'-x]$. Finally, we look at computing the value $[T]=[y-x+r+r']$ in two different ways, as the sum of a and b, and as the sum of $y-x$ and $r+r'$. Looking at the addition using the first way, we first open the values a and b, and write the exact integer relation (using Eq. 2):

$$T = b + a - M \cdot \mathsf{LT}(T,b) \tag{6}$$

We can also write similar expressions for b and a,

$$\begin{aligned} b &= [y] + [r] - M \cdot \mathsf{LT}(b,[r]) \\ a &= [-x] + [r'] - M \cdot \mathsf{LT}(a,[r']) \end{aligned} \tag{7}$$

Thus the first expression for the sum T is given by (combining Eqs. 6, 7):

$$T = [y] + [r] - M \cdot \mathsf{LT}(b,[r]) + [-x] + [r'] - M \cdot \mathsf{LT}(a,[r']) - M \cdot \mathsf{LT}(T,b) \tag{8}$$

Grouping the terms differently and computing the sum using the latter expression:

$$T = [y-x] + [r+r'] - M \cdot \mathsf{LT}(T,[r+r']) \tag{9}$$

Once again, $[y-x]$ and $[r+r']$ can be expanded using Eq. 2 as:

$$\begin{aligned} [y-x] &= [y] + [-x] - M \cdot \mathsf{LT}([y-x],[y]) \\ [r+r'] &= [r] + [r'] - M \cdot \mathsf{LT}([r+r'],[r]). \end{aligned} \tag{10}$$

Plugging Eq. 10 into Eq. 9, and equating that with the expression in Eq. 8, we get the following expression for $\mathsf{LT}([y-x],[y])$, the quantity of interest:

$$\mathsf{LT}([y-x],[y]) = \mathsf{LT}(b,[r]) + \mathsf{LT}(a,[r']) + \mathsf{LT}(T,b) - \mathsf{LT}([r+r'],[r]) - \mathsf{LT}(T,[r+r'])$$

This completes the correctness proof. To generate an efficient protocol for this expression, the final observation is that $\mathsf{LT}([r+r'],[r])$ is generated as a by-product of the computation required to generate the bit decomposition of $r+r'$ from the bit decompositions of r,r' (to enable a call to Π_{LTBits}). ■

3 Evaluation

We implement our protocol in the MP-SPDZ Framework [13]. The entire protocol is a handful of lines of python code, as shown in Fig. 1b, and reads directly from the pseudocode; this makes it highly amenable to implementation. We evaluate our protocol over various MPC settings and a brief summary of our experiments is provided below:

(i) **Throughput of Comparisons:** In this experiment, we measure the throughput of comparison operations and compare this with prior art. These results are presented in Sec. 3.1.

(ii) **Private Evaluation of ResNet-50:** We provide benchmarks for evaluating ResNet-50 [17] using dishonest majority privacy-preserving computation. We use the state-of-the-art matrix triple generation algorithm [7] and combine that with our comparison protocol and compare that against the prior art [7,14]. These results are presented in Sec. 3.2.

Set-up Details: We use an MPC set-up similar to prior works [14,25,24]. Each party is run on an Intel(R) Core(TM) i9-9900 CPU @ 3.10GHz with 128GB of RAM over a 10Gb/s network switch with an average round-trip ping time of 1ms. For the WAN setting we use two or three machines depending on the protocol wich are equipped with Intel(R) Xeon(R) CPU E5-2690 v3 @ 2.60GHz and 54GB of RAM while the network capability was slowed down using the Linux **tc** command limiting the bandwidth to 100Mb/s and 100ms round-trip ping time.

3.1 Throughput of Rabbit comparisons

We conduct experiments in all combinations of the possible adversarial models (active, passive), adversarial settings (honest majority, dishonest majority), and domains (OT-based in $\mathbb{Z}_{2^k}$, OT-based in $\mathbb{F}_p$, HE-based in $\mathbb{F}_p$), and in both the LAN and WAN network settings. Table 1 provides a summary of the primitives used as preprocessing (i.e., offline cost) for a Rabbit comparison, vs. an edaBit comparison [14], their online round complexity, security, and the need for slack, in $\mathbb{Z}_{2^k}$ and in $\mathbb{F}_p$. As in Escudero *et. al.* [14], we benchmark the time required for a million comparisons between two (DM) or three (HM) servers described in the setup above. Table 2, 3 show the number of comparisons per second (throughput) and communication per party (kbits) for a single operation in the LAN and WAN settings respectively. Our protocol improves prior art in runtime and communication by upto 2×, and in all cases, achieves these without any slack.

Communication for Π_{LTC} over $\mathbb{F}_p$. Note that our protocol incurs higher communication cost, when performing comparisons over fields. This is due to the use of

Sub-protocols	Rabbit		edaBits Comp. [14]	
	$\mathbb{Z}_{2^k}$	$\mathbb{F}_p$	$\mathbb{Z}_{2^k}$	$\mathbb{F}_p$
edaBits	1:$\{k\}$	1:$\{k\}$	1:$\{l\}$	2:$\{l-m+s,m\}$
daBits	1	1	1	1
ANDs	$3(k-1)$	$k\log_2 k$*	$3(l-1)$	$2(k-1)$
# Rounds	$2+\log_2 k$	$\log_2 k$	$2\log_2 l$	$2\log_2 k$
Security, slack	Perfect, No	Statistical, No	Statistical, Yes	Statistical, Yes

Table 1: Theoretical complexity comparison of exact comparison functionality over $\mathbb{Z}_{2^k}$ and $\mathbb{F}_p$ where k is the bit-size of the datatypes, l is the $\log_2$ bound on the inputs/data, and m refers to the number of bits to be truncated.

a more expensive Prefix OR computation. Prior works encode the data in a larger dataspace and simply extract the MSB for the comparison. In a manner similar to the optimization from Π_{LTC} to Π_{ReLU}, we can extract the MSB to compute a comparison. This operation requires using a prefix computation protocol Π_{PreOpL} (cf [6]), which has a linear overhead of $2(k-1)$ bit-triples in $\log_2 k$ rounds – matching that of edaBits [14]. If a different encoding is used, where positive and negative numbers are determined by comparison with $\lfloor p/2 \rfloor$, the same protocol can be used with statistical correctness, determined by the specific choice of prime (with a small gap between p and 2^k). A suitable choice of prime p would also further lower the prepossessing time, when performed using HE.

3.2 Neural Network Evaluation

In this section, we provide benchmarks for using our approach for comparison on evaluating the ResNet-50 architecture [17]. In our experiments, we consider neural network inference over 64-bit datatypes and compare the offline and online performance of our protocol with the state-of-the-art protocols with active security in the dishonest majority setting. For prior art, we use the recent protocol for matrix triple generation [7] in conjunction with our Π_{LTC} comparison protocol. The results are summarized below.

The work of Chen *et. al.* [7] requires the plaintext modulus to be 128-bits, due to the slack required in the comparison. In this work, we eliminate that slack and hence only require generation of matrix triples using homomorphic encryption (HE) with a plaintext space of 64-bits. While Chen *et. al.* [7] require a 128-bit modulus and $N = 2^{15}$ (degree of the cyclotomic polynomial), we can generate 64-bit triples. This enables us to run the algorithm with lower HE parameters (and consequently better performance). We use $N = 2^{14}$, a plaintext modulus of 64-bits and a ciphertext modulus of 480. With a conservative analysis this leaves enough room for 40-bits of statistical security. We set the block size to 64 instead of 128 and thus pack 4 matrices in a single ciphertext (compared to 2 in Chen *et. al.* [7]). We list the sizes of matrices required for the computations in ResNet-50 and then measure the time required (and communication overhead) for matrix triple generations using these different set-ups. We run the protocols on a similar set-up as Chen *et. al.* [7], using a 5Gb/s LAN bandwidth and about 300 Mbps WAN bandwidth. Hence, just for

		Domain	Rabbit		edaBits Comp. [14]	
			Thru.(ops/s)	Comm.(kb)	Thru.(ops/s)	Comm.(kb)
Dishonest Majority	**Active**	2^k (OT)	2936	1252.4	3038	1252.2
		p (OT)	1537	2847.0	1056	4458.6
		p (HE)	1495	1678	1495	1635.99
	Passive	2^k (OT)	165368	39.5	172211	38.3
		p (OT)	73947	87.8	51478	132.2
		p (HE)	65750	67.63	41175	41.71
Honest Majority	**Active**	2^k	117607	5.62	116616	5.54
		p	88780	9.43	41028	19.62
	Passive	2^k	5706569	0.5	5600265	0.5
		p	1421412	0.96	472316	1.58

Table 2: Throughput and communication for running secure comparisons using Rabbit in contrast to prior art *over LAN* for 16 threads, with 2 million comparisons in total.

		Domain	Rabbit		edaBits Comp. [14]	
			Thru.(ops/s)	Comm.(kb)	Thru.(ops/s)	Comm.(kb)
Dishonest Majority	**Active**	2^k (OT)	33	1237	33	1237
		p (OT)	1.37	29646	0.37	112594
		p (HE)	2	19089	N/A	N/A
	Passive	2^k (OT)	596	39.26	604	38.18
		p (OT)	366	87.59	245	131
		p (HE)	427	67.01	431	41.71
Honest Majority	**Active**	2^k	5444	5.54	5488	5.52
		p	1639	16.96	1463	19.53
	Passive	2^k	15096	0.49	15182	0.49
		p	11492	0.96	7640	1.53

Table 3: Throughput and communication for running secure comparisons using Rabbit in contrast to prior art *over WAN*. All numbers were produced using 2 million comparisons with 8 threads, except in the active security, dishonest majority field cases where we used only 32,000 comparisons due to time constraints. Note that for the active security, dishonest majority field case with HE preprocessing, the 54GB RAM machines ran out of memory due to the large ciphertexts kept in memory by MP-SPDZ - for Rabbit there were no memory issues as the memory footprint is reduced to half due to ciphertexts that only need to accommodate 64-bits plaintexts.

the triple generation, our communication complexity reduces by about 60% and the total time by about 40% of [7] for the same set of triple generations (LAN and WAN settings are fairly similar as the protocols are compute dominated). Furthermore, our computational burden for the matrix triple computations reduces from about 72GB to 9.3GB – a critical improvement for systems based on HE.

We also run the offline and online computations for the comparisons in ResNet-50 and compare the total time. Our protocol takes about 11 hours and 2883.3 GB of communication. When compared to prior art of Chen *et. al.* [7], they evaluate the same network in about 24hrs with 2036 GB (using improved comparisons using edaBits). Thus, our work is 2× faster albeit uses slightly more communication due to the communication gap for Rabbit and edaBit for dishonest majority within a char-

acteristic p field. Thus, our comparison protocol, combined with the improvement in the triple generation phase due to slack elimination, provides a significant throughput improvement over state-of-the-art MPC protocols for neural network evaluation.

4 Discussion

In this section, we provide a deeper discussion on the following aspects of this work. We (1) elaborate on our central contribution of removing the slack and how it enables computation over smaller data types; (2) we discuss applications of these protocols; and (3) provide an analysis of the statistical security provided by our protocol along with the choice of modulus for the case of fields.

4.1 Elimination of "Slack" in Comparisons

One important contribution of this work is the elimination of a "slack" between the inputs (in other words the computable part of the data) and the actual size of the datatypes used in the MPC engines. Note that prior work in the dishonest majority setting requires a slack to accommodate for the statistical parameter. Commonly, this statistical parameter, which is necessary to ensure security, is at least 40-bits. This implies that the actual datatypes used in the MPC are at least 40-bits longer than the values we need to compute upon. As a consequence, prior work requires 128-bit datatypes for the MPC, necessary to support 64-bit computations. On the contrary, our comparison protocol achieves exact comparison without the need for any slack and thus operates on smaller, 64-bit datatypes. As shown in Section 3.2, when the slack removal is combined with recent advances such as the contributions of the work of Chen *et. al.* [7], the smaller MPC datatypes enable faster triple generation, reduce the communication and computational overhead and increase the overall efficiency of the MPC computations, beyond secure comparisons.

4.2 Applications to Machine Learning and Beyond

Privacy-preserving machine learning, which is of increasing interest in the field of MPC, often relies on efficient protocols for computing ReLU, a non-linear function that is given by $\mathsf{ReLU}(x) = \max(x,0)$. Using fixed-point encoding, computation of the ReLU function reduces to a comparison with an encoding of 0 (i.e., a constant). Given that this non-linear function is the bottleneck of many state-of-the-art secure machine learning protocols [21,18], our proposed protocol improves this entire line of work.

The thresholding operation is yet another application where we require a comparison with a public constant. In image processing and computer vision, threasholding is used for segmenting images (e.g., turn a grayscale image into a binary one). In particular, it replaces a pixel with a black (resp. white) pixel, if the image intensity is less (resp. greater) than a fixed constant. In yet another application, Cryptography for #metoo [19], the system heavily relies on the use of public value thresholding. In adversarial machine learning, algorithms for robustness that work over privacy-preserving computation also require thresholdings with small public values. In all these applications, the functionality can be efficiently achieved using our comparison

with constant protocol. Thus, our efficient comparison with constant protocol, Π_{LTC} (Sec. 2.2), is deployable on several application scenarios.

On the other hand, there are applications, where secure comparisons with a constant do not suffice, but a comparison between two values that are both secret is required. In such cases, our comparison with secret protocol, Π_{LTS} (Sec. 2.4) can be deployed. Applications in this line of work go as far in the past as the first instance of the problem: Yao's millionaires' problem [34], and include amongst others also secure auctions [4], and secure linear programming [28].

4.3 Statistical Security

We remark that the protocols Π_{LTBits}, Π_{LTC}, Π_{ReLU} and Π_{LTS} are all inherently information theoretically secure. However, when combined with a larger MPC platform, the overall security is set by the weaker between the MPC platform and the protocol, and hence when using protocols such as SPDZ [12], BDOZa [2], SPD$\mathbb{Z}2^k$ [10], our security reduces to statistical. The current implementation has a small statistical security due to the use of edaBits [14]. The protocol for edaBit generation produces shares:

$$[r]_M \text{ and } \{[r_i]_2\}_{i=0}^{m-1} \text{ such that } r \equiv \sum_{i=0}^{m-1} r_i \cdot 2^i \pmod{M} \tag{11}$$

In particular, for the correctness of Π_{LTC} in Sec. 2.2, we require that $r = \sum r_i \cdot 2^i$, and this condition is different from Eq. 11 in a subtle yet important way. In the case where $M = 2^m$, this does not raise an issue. However, in all other cases, in particular including the field case, we have $2^{m-1} < M < 2^m$, and so we can have $r = (\sum r_i \cdot 2^i) - M$. In this case, the correctness of Π_{LTC} does not hold, as the set of sharings $\{[r_i]_2\}_{i=0}^{m-1}$ does not correspond to the bit decomposition of r. To address this issue, we note that this failure probability depends on the size of the gap between the modulus and the bounding power of 2 in relation to the modulus. The failure probability is given by:

$$\text{Failure probability } = \frac{2^m - M}{2^m} \tag{12}$$

which is simply the probability that r is between M and 2^m. Thus, if $\delta = 2^m - M$, the failure probability can be made small for suitable choice of $\delta/2^m$. Thus, in practice, we choose the largest 64-bit prime $p = 2^{64} - 59$ for our implementation. This gives our protocol a failure probability of less than 2^{-59}. However, from a security point of view, for statistical hiding, we use the fact that $r \xleftarrow{R} \{0,1,\cdots,2^m - 1\}$ when reduced modulo M is still close to uniform in $\mathbb{Z}_M$ (to ensure the masked value is hidden). If the former distribution is D_1 and the latter is D_2, then this statistical distance can be computed exactly as given in Eq. 13. Thus, the statistical closeness can also be made negligible by a suitable choice of $\delta/2^m$. A union bound over the two expressions (Eq. 12 and 13) allows

us to achieve both correctness and privacy with a statistical parameter close to 58-bits.

$$\begin{aligned}\mathsf{Statistical\ closeness} &= \mathsf{Distance}(D_1, D_2) \\ &= \frac{1}{2}\left[\left(\sum_{i=0}^{\delta-1}\frac{2}{2^m} - \frac{1}{M}\right) + \left(\sum_{i=\delta}^{2^m-1}\frac{1}{M} - \frac{1}{2^m}\right)\right] \\ &= \frac{\delta\cdot(M-\delta)}{M\cdot 2^m} \leqslant \frac{\delta}{2^m}\end{aligned} \tag{13}$$

Furthermore, we note that one can use rejection sampling as follows: run Π_{LTBits} over the bit decomposition of r and the modulus M to check if $r \geqslant M$. If this is the case then reject the sample. This way we can eliminate such edaBits and note that the rejections happen with probability similar to the expression in Eq. 12 and is thus ideal once again when the prime p is close to a power of 2.

As an aside, the closer the prime is to the power of two, the lower is the failure probability. However, when combining with other protocols, such as those mentioned in Sec. 3.2, there are other considerations in choosing the prime. For instance, for efficiency reasons BFV [15,5] requires special prime modulus, where $p-1$ has a large factor (around 2^{14} - 2^{16}). One such prime is $p = 2^{64} - 83$, where $33196 \mid p-1$ and $\phi(33196) = 16128$ (with ϕ the Euler's Totient function), which would be secure given the 16k degree and appropriately chosen modulus q.

5 Comparison with Related Work

After the seminal work of Yao [34], which operates in the two-party setting, and is based on garbled circuits, many works studied the problem of secure comparisons, both in the two-party [8,29,35], as well as in the multi-party setting [9,23,6,20]. In this work, we focus on the general n-party setting. Damgård *et. al.* [9] were the first to tackle the challenge of secure, constant-round bit decomposition of secret shared inputs, which is a necessary building block for most comparison protocols. In the same work [9], they extend and apply their bit-decomposition protocol to develop a secure comparison protocol (amongst other applications). Their comparison protocol works in the general n-party setting, with any underlying linear secret sharing scheme (LSSS), and provides unconditional security against active adversaries (assuming that the multiplication protocol of the LSSS is also actively secure), in the honest majority setting.

Improving upon the complexity of Damgård *et. al.*'s [9] bit decomposition, comparison, equality, and interval test protocols, Nishide and Ohta [23] provide new, simplified protocols. In addition, Nishide and Ohta [23] construct new secure comparison, equality, and interval test protocols, which do not rely on bit decomposition. For their deterministic equality test protocol that is independent of bit decomposition, Nishide and Ohta [23] apply a masking technique similar to the one we use in our comparison protocol: they use a random shared value that the parties possess both in its $\mathbb{F}_p$ and in its bit decomposed form to mask and afterwards open the secret shared input of the equality test.

In an attempt to design comparison protocols with concrete efficiency instead of asymptotic, Catrina and de Hoogh [6] propose several versions of secure equality

Protocol	Communication		Computation		Rounds	Security	Adversary	Setting
	Offline	Online	Offline	Online				
[9]	-	$\mathcal{O}(\ell\log\ell)$	-	$\mathcal{O}(\ell\log\ell)$	$\mathcal{O}(1)$	perfect	active	HM
[23]	-	$\mathcal{O}(\ell)$	-	$\mathcal{O}(\ell)$	$\mathcal{O}(1)$	perfect	passive	HM
[6]	-	$\mathcal{O}(\log\ell)$	-	$\mathcal{O}(\log\ell)$	$\mathcal{O}(\log\ell)$	statistical	passive	HM
[20]	$\mathcal{O}(\ell)$	$\mathcal{O}(\log\ell)$	$\mathcal{O}(\log\ell)$	$\mathcal{O}(\log\ell)$	$\mathcal{O}(\log\ell)$	statistical	active	HM
Rabbit	$\mathcal{O}(\ell)$	$\mathcal{O}(\ell\log\ell)$	$\mathcal{O}(\ell)$	$\mathcal{O}(\ell)$	$\mathcal{O}(\log\ell)$	perfect*	active	DM

Table 4: Comparison of the related work in the n-party setting in terms of offline, and online communication and computation complexity; in terms of rounds; in terms of security; and in terms of adversarial model and adversarial settings supported. In the context of adversarial setting HM stands for honest majority, while DM stands for dishonest majority. *perfect security holds only when the underlying secret sharing scheme operates over $\mathbb{Z}_{2^k}$.

and comparison tests. Their protocols run in logarithmic number of rounds, in the bit-length of the values to be compared, but also with logarithmic communication cost (instead of the usually linear communication cost). The efficiency of these protocols comes also at the cost of statistical, instead of unconditional security and have been adopted and implemented in a number of MPC platforms (e.g., [13,1]). Our comparison protocol, in combination with the recent advances in the generation of daBits [25], and edaBits [14] performs concretely better than the one of Catrina and de Hoogh [6], while offering unconditional (instead of statistical) security in $\mathbb{Z}_{2^k}$.

Lipmaa and Toft [20] propose three different comparison protocols. Only one of these comparison protocols works for the general n-party setting with active security, and while it offers sublinear online communication complexity, it is not constant-round and it has linear offline communication cost. Like other protocols in the literature [29,8], the core of [20] lies in the idea of splitting the two strings to be compared into smaller, equal length blocks, and perform the comparison on the first block where they differ. This way the problem of comparison only needs to be addressed on smaller strings (the blocks), and equality testing can be applied to the larger strings (to allow for the necessary reduction of the size of the blocks on which comparison is to be performed). Other recent concretely-efficient comparison protocols such as [16,32,33,31] also eliminate the need for a slack but operate in fixed adversarial models and are tied to a 3-party MPC setting.

In Table 4 we detail the asymptotic costs and security features of the related work in secure comparisons for the general n-party setting. It is important to remark that most prior secure comparison protocols require the values to be compared to be smaller than the space where the comparison takes place. Although this may result in efficient protocols for the particular comparison operations, it also requires a larger MPC engine to perform all (other) computations. Essentially, this means that all adjacent computations should be performed in a larger space, and all values to be communicated throughout the protocol need to be larger by a factor proportional to the necessary slack for the secure comparison. Our protocol crucially overcomes this limitation.

6 Conclusion

In this work, we propose novel comparison protocols for general n-party computation. Our protocols enjoy perfect security, when we operate over $\mathbb{Z}_{2^k}$, and crucially eliminate the need for "slack" – a larger dataspace to compute secure comparisons, enabling computations over smaller datatypes. In terms of concrete efficiency, our protocols improve prior art by twice for most adversary structures, while keeping a smaller communication complexity. Given that comparisons are a fundamental secure computation primitive, many MPC applications can benefit from our protocols.

Acknowledgements

This work was supported in part by the Research Council KU Leuven grant C14/18/067, and by CyberSecurity Research Flanders with reference number VR20192203, and by ERC Advanced Grant ERC-2015-AdG-IMPaCT.

References

1. Abdelrahaman Aly, Marcel Keller, Emmanuela Orsini, Dragos Rotaru, Peter Scholl, Nigel P. Smart, and Tim Wood. SCALE-MAMBA v1.2: Documentation, 2018.
2. Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic Encryption and Multiparty Computation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 169–188. Springer, 2011.
3. Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael Schwartzbach, and Tomas Toft. Secure Multiparty Computation Goes Live. In *International Conference on Financial Cryptography and Data Security*, pages 325–343. Springer, 2009.
4. Peter Bogetoft, Ivan Damgård, Thomas Jakobsen, Kurt Nielsen, Jakob Pagter, and Tomas Toft. A Practical Implementation of Secure Auctions based on Multiparty Integer Computation. In *International Conference on Financial Cryptography and Data Security*, pages 142–147. Springer, 2006.
5. Zvika Brakerski. Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. In *Advances in Cryptology—CRYPTO*, pages 868–886. Springer, 2012.
6. Octavian Catrina and Sebastiaan de Hoogh. Improved Primitives for Secure Multiparty Integer Computation. In *International Conference on Security and Cryptography for Networks*, pages 182–199. Springer, 2010.
7. Hao Chen, Miran Kim, Ilya Razenshteyn, Dragoş Rotaru, Yongsoo Song, and Sameer Wagh. Maliciously Secure Matrix Multiplication with Applications to Private Deep Learning. In *Advances in Cryptology—ASIACRYPT*, 2020.
8. Geoffroy Couteau. New Protocols for Secure Equality Test and Comparison. In *Applied Cryptography and Network Security (ACNS)*, 2018.
9. Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally Secure Constant-Rounds Multi-Party Computation for Equality, Comparison, Bits and Exponentiation. In *Theory of Cryptography Conference (TCC)*, pages 285–304. Springer, 2006.

10. Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P Smart. Practical Covertly Secure MPC for Dishonest Majority–or: Breaking the SPDZ Limits. In *European Symposium on Research in Computer Security*, pages 1–18. Springer, 2013.
11. Ivan Damgård and Jesper Buus Nielsen. Universally Composable Efficient Multiparty Computation from Threshold Homomorphic Encryption. In *Annual International Cryptology Conference*, pages 247–264. Springer, 2003.
12. Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty Computation from Somewhat Homomorphic Encryption. In *Annual Cryptology Conference*, pages 643–662. Springer, 2012.
13. Data61. MP-SPDZ: Versatile Framework for Multi-party Computation, 2019. `https://github.com/data61/MP-SPDZ`.
14. Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved Primitives for MPC over Mixed Arithmetic-Binary Circuits. Cryptology ePrint Archive, Report 2020/338, 2020. `https://eprint.iacr.org/2020/338`.
15. Junfeng Fan and Frederik Vercauteren. Somewhat Practical Fully Homomorphic Encryption. Cryptology ePrint Archive, Report 2012/144, 2012. `https://eprint.iacr.org/2012/144`.
16. Wataru Fujii, Keiichi Iwamura, and Masaki Inamura. Secure Comparison and Interval Test Protocols Based on Three-Party MPC. In *6th International Conference on Information Systems Security and Privacy, ICISSP 2020*, pages 698–704. SciTePress, 2020.
17. Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
18. Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1651–1669, 2018.
19. Benjamin Kuykendall, Hugo Krawczyk, and Tal Rabin. Cryptography for# metoo. In *Privacy Enhancing Technologies Symposium (PETS)*, 2019.
20. Helger Lipmaa and Tomas Toft. Secure Equality and Greater-Than Tests with Sublinear Online Complexity. In *International Colloquium on Automata, Languages, and Programming*, pages 645–656. Springer, 2013.
21. Jian Liu, Mika Juuti, Yao Lu, and Nadarajah Asokan. Oblivious Neural Network Predictions via MiniONN Transformations. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 619–631, 2017.
22. Payman Mohassel and Yupeng Zhang. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *IEEE Symposium on Security and Privacy (S&P)*, 2017.
23. Takashi Nishide and Kazuo Ohta. Multiparty Computation for Interval, Equality, and Comparison without Bit-Decomposition Protocol. In *International Workshop on Public Key Cryptography*, pages 343–360. Springer, 2007.
24. Dragos Rotaru, Nigel P Smart, Titouan Tanguy, Frederik Vercauteren, and Tim Wood. Actively Secure Setup for SPDZ. Cryptology ePrint Archive, Report 2019/1300, 2019. `https://eprint.iacr.org/2019/1300`.
25. Dragos Rotaru and Tim Wood. Marbled Circuits: Mixing Arithmetic and Boolean Circuits with Active Security. In *International Conference on Cryptology in India*, pages 227–249. Springer, 2019.
26. Sepior. `https://sepior.com/`, 2020.
27. Sharemind. `https://sharemind.cyber.ee/`, 2020.
28. Tomas Toft. Solving Linear Programs Using Multiparty Computation. In *International Conference on Financial Cryptography and Data Security*, pages 90–107. Springer, 2009.

29. Tomas Toft. Sub-Linear, Secure Comparison with Two Non-Colluding Parties. In *International Workshop on Public Key Cryptography*, pages 174–191. Springer, 2011.
30. Unbound. `https://www.unboundtech.com/`, 2020.
31. Sameer Wagh. *New Directions in Efficient Privacy Preserving Machine Learning*. PhD thesis, Princeton University, 2020.
32. Sameer Wagh, Divya Gupta, and Nishanth Chandran. SecureNN: 3-Party Secure Computation for Neural Network Training. In *Privacy Enhancing Technologies Symposium (PETS)*, 2019.
33. Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. FALCON: Honest-Majority Maliciously Secure Framework for Private Deep Learning. In *Privacy Enhancing Technologies Symposium (PETS)*, 2021.
34. Andrew C Yao. Protocols for Secure Computations. In *23rd annual symposium on foundations of computer science (sfcs 1982)*, pages 160–164. IEEE, 1982.
35. Ching-Hua Yu and Bo-Yin Yang. Probabilistically Correct Secure Arithmetic Computation for Modular Conversion, Zero Test, Comparison, MOD and Exponentiation. In *International Conference on Security and Cryptography for Networks*, pages 426–444. Springer, 2012.