Through the Looking-Glass: Benchmarking Secure Multi-Party Computation Comparisons for ReLU's

Abdelrahaman Aly$^{1,2[0000-0003-2038-5668]}$, Kashif Nawaz$^{1[0000-0002-3887-7364]}$, Eugenio Salazar$^{1[0000-0003-0489-7727]}$, and Victor Sucasas$^{1[0000-0002-7981-401X]}$

[1] Cryptography Research Centre, Technology Innovation Institute, Abu Dhabi, UAE.
[2] imec-COSIC, KU Leuven, Leuven, Belgium.
{firstname.lastname}@tii.ae

Abstract. Comparisons are a basic component of Rectified Linear Unit functions (ReLU's), ever more present in Machine Learning and specifically in Neural Networks. Motivated by the increasing interest on privacy-preserving Artificial Intelligence, we explore the state of the art of Multi-Party Computation (MPC) protocols for privacy preserving comparisons. We then introduce constant round variations of these protocols, which are compatible with customary fixed point arithmetic over MPC and, geared towards realistic ReLU implementations. Furthermore, we provide novel constructions, inspired by commonly used comparison protocols equipped with state of the art elements. We translate these results into practice and provide an open source library, compatible with current MPC software tools, showcasing our contribution. We include a comprehensive benchmarking on various adversarial settings. Finally, we offer conclusions about the viability of our protocols, when adopted for privacy-preserving Machine Learning.

Keywords: Secure Multi-Party Computation · ReLU Functions · Applied Cryptography.

1 Introduction

Secure Multi-Party Computation, has continued to evolve, through the years, becoming more practical, and useful. This is specially true in the last few years, when fundamental results have considerably reduced computing times. It was just logical, for researchers to start asking questions regarding how they can build useful applications on top of these protocols. New results followed, introducing novel ways to achieve basic functionality, that was all but indispensable, for the vast array of applications that could make use of MPC. Among them, new ways to interpret rational numbers, via fixed point arithmetic, and, of course, how to perform inequality tests (INQ).

Since then, MPC has been considered as a viable option, for many modern applications [1]. Among them, Privacy Preserving Machine Learning (PPML), has gathered attention in the field, given the need for faster performing protocols, for a variety of different tasks. This is especially true, of course, for comparisons and fixed point arithmetic. We can name for instance, the contributions from Makri et al [2] and Catrina and De Hoogh [3] on INQ and Catrina et al. [4] on and fixed point arithmetic. Building blocks, that are indispensable tools to implement activation functions such as ReLU's, which are increasingly more popular. Additionally, given their simplicity, ReLU's are becoming a viable vehicle for the implementation of privacy preserving machine learning. Any practical deployment of ReLU's, fully depend on the combination of these 2 elements and their performance.

Modern Multiparty frameworks, such as the popular SCALE-MAMBA [5] or the well known MP-SPDZ [6], already incorporate some of these results, (specifically [3, 4]), and have given developers the opportunity to build and deploy solutions that incorporate MPC. In this work, we continue these efforts and provide researchers and implementers, with the means to also assess, incorporate and combine state of the art results. Among them, the results from Makri et al., commonly referred to as Rabbit, as well as other families of comparisons, are included in this paper. We go a step further and, besides the conventional flavors, offer our own variations of the protocols. Our main focus are ReLU's (we include basic formulations), and any other realistic application, that use the common tools and protocols aforementioned.

Furthermore, and thanks to recent contributions, of the likes of Zaphod [7], we also show how these protocols could be implemented in constant rounds. We introduce variations that exploit the characteristics of the environments where they would run, specifically SCALE-MAMBA, and give developers the necessary tools for the selection of the underlying MPC protocols, via exhausting benchmarking. Basically, in this work, we answer the questions, i). What do we need to implement state of the art comparison protocols ii). How can we improve them and, iii). What protocols perform better in the context of implementing ReLU's mixed with fixed point arithmetic.

1.1 Related Work

Besides the existing body of work, related to secure comparisons from MPC, starting with [8], we focus our attention to both of the most recent contributions that motivated this work, i.e. Makri et al. [2] or Rabbit, and Escudero et al. [9] or edaBits. Both introduced comparison mechanisms that reduce the number of rounds, by mixing Boolean and Arithmetic circuit evaluations via edaBits. Maybe, the more notable difference however is on adoption and practicality. We are interested on making our protocols compatible with existing frameworks, specifically SCALE-MAMBA, and commonly used results for, among others, fixed point arithmetic and Zaphod. With this in mind, we also make available our implementation and include extensive benchmarking. Besides this, we would like to explore some of the main differences with the state of the art:

On the Elimination of Slack: One of the main contributions from Rabbit [2] is that it does not require slack, which MPC frameworks typically set at 40 bits. This implies that private comparisons of integers represented with 64 bits can be performed with smaller data-types of 64 bits, instead of the 128 of previous protocols. However, inequality tests (INQ) are common tools on applications that do extensive use of other representations, besides integers, namely fixed point. Current state of the art [4] does still require extended domains beyond 64 bits, to have meaningful precision, and they would nonetheless require the presence of slack during truncation (the precision adjustment after multiplication). Additionally, albeit slack might not be present in the protocols for comparisons, it still is ever present in several protocols that are required by Rabbit and its adaptations, e.g. Zaphod [7] or daBits [10] or even edaBits [9]. These factors, in turn, still force application designers, to use larger prime sizes. Hence, our work pursues to exploit the advantages of Rabbit whilst still bounded to to the realities of practical deployments.

On Randomness Sampling: Another differentiator, with the current state of the art, is how we sample randomness. Both Escudero et al. and Makri et al. rely on edaBits [9] for the sampling. We instead, rely on Zaphod and daBits, as presented in [7]. This gives us the control over each random bit, but more importantly, it allows us to execute our protocols relying on, for instance Garbled Circuits (GC) for Boolean and SPDZ [11] for Arithmetic circuits.

On Boolean Evaluation: Our protocols incorporate a selection of the most efficient elements of both Makri et al. and Escudero et al., for all things related to Boolean Circuits. We adapt them to be able to use them in constant round, with Zaphod. This includes mixing (much in the style of Escudero et al.) the principles used by Catrina and De Hoogh for their LTZ (Less Than Zero) construciton, with the bitwise LTEQZ (Less Than Equal to Zero) test explored by Rabbit.

1.2 Our Contributions

- We present newly constant round variations of the state of the art protocols on privacy preserving comparisons, introduced by Makri et al. [2], together with some constant-round inspired new designs. We focus our attention to the use case at hand, that is, the implementation of ReLU's. We introduce, implement and benchmark 3 flavors, of such protocols, using, among others, the techniques, succinctly mentioned by the authors on [2]. The variations are namely the following:
  - i). **Rejection List Rabbit:** Regardless of the $\mathbb{F}_q$, we sample randomness in $\mathbb{Z}_{2^{\lceil \log_2 q \rceil}}$, and reject samples that are above q. As we see, because of setup restrictions, q needs to be relatively large in realistic setups, e.g. $q > 2^{121}$ (traditionally, of 128 bits size).
  - ii). **Conventional Rabbit:** Bitwise sampling happens either from above, sampling $\lceil \log_2 q \rceil$ or, below $\lfloor \log_2 q \rfloor$ bits. It is assumed q is close to a power of 2. This makes, Rabbit probabilistic in nature.

iii). **Slack Rabbit:** A variation of the protocol, where we relax security (becoming statistical, instead of information theoretic, from an ideal perspective), so that it can exploit the setup, whilst used in conjunction with protocols that require slack for fixed point representation and other protocols.

- Much in the style of Escudero et al. [9], we adapt and optimize the commonly used comparison construction of Catrina et al [3]. More precisely:
  i). We transform it into a constant round construction, thanks to Aly et al. Zaphod [7] and Hazay et al. [HSS17] [12];
  ii). we replace the fundamental bitwise less than (LT) construction, by the recently introduced bitwise, less than equal (LTEQ) protocol from [2].
- A thorough benchmark, aimed towards implementers that, on top of the basic Rabbit LTZ construction, and the variations mentioned above, also includes the optimized protocol for ReLU's from [2]. Our benchmark includes different compilations of the same bitwise LTEQ comparison circuit from Makri et al., including (in Bristol Fashion):
  i). **AND_XOR$'s$:** The circuit *as presented* in [2], mixing `xor` with `and` gates, to benefit from *free-xor*.
  ii). **AND_NOT$'s$:** The same circuit, as it would be typically derived by commonly used synthesizers. The circuit, in this case, is entirely made of `and` mixed with `not` gates.
- More importantly, **usability**. We provide open source implementations on publicly available repositories for our code written for SCALE-MAMBA.

Note that Zaphod and SCALE-MAMBA, adjust the number of rounds (based on the circuit depth) when the setup is honest majority, instead of Full Threshold with SPDZ. We also explore such setups in our Benchmarking.

Our Implementation A priority of this work is implementability/usability. In that sense, we make use, of current available tooling that is supported by the community. Given that, our protocols make use of Zaphod, [HSS17], and SPDZ; we focus on SCALE-MAMBA, specifically version 1.3 or above [5], which is readily available. In contrast, MP-SPDZ, currently, does not support field conversion between their BMR processor and LSSS.

Note that our protocol descriptions are based on our implementations, specifically over SCALE-MAMBA. Hence we, at times, and, w.l.g., adapt the protocols and pseudocode accordingly. By doing so, we believe we provide better tools to implementers, albeit the minor optimizations that are being omitted.

1.3 Outline

This work is structured as follows, Section 1, we present the main contributions of this work as well some basic comparison with the state of the art. Section 2, introduces some general aspects needed for understanding our contributions, from a theoretical and practical perspective. We explore the relevant minutiae of Rabbit, as introduced by [2], on Section 3. This Section also incorporates variations of some Rabbit constructions, including some of our contributions. We then continue to explore our contributions on Section 4, where we present a novel constant round LTEQZ protocol, mixing Rabbit and Catrina and Hoogh [3]. Finally, we include our extensive benchmarking on Section 5 and some further discussion and our conclusions in Section 6.

2 Preliminaries

In this section, we discuss some of the main aspects that are necessary to understand the contributions ahead. We give main emphasis on including implementation aspects to the mathematical definitions, to facilitate the work of implementers.

2.1 Notation

Let q be a sufficiently large prime such that i). it can instantiate an MPC protocol, ii). supports fixed point arithmetic, with a public bitwise precision p and, iii). sufficient statistical security $\texttt{sec}$. Additionally, constants and vectors are uppercased R.
We make use of the same integer representation commonly used by the existing tooling, e.g. SCALE-MAMBA. That is, we identify the finite field F_q with the centered (integer) interval $[-\frac{q}{2}, \frac{q}{2}) \cap \mathbb{Z}$ and, similarly, the ring $\mathbb{Z}_{2^k}$ with $[-2^{k-1}, 2^{k-1}) \cap \mathbb{Z}$. Note that, in this work we assume $k = 64$.

We encode any negative number x on $\mathbb{F}_q$ as: $q - x$, e.g. -1 becomes $q - 1$. In $\mathbb{Z}_{2^{64}}$, we use a similar formulation $2^{64} - x$, e.g. $2^{64} - 1$. Notice, however that, because of the requirements of fixed point arithmetic, a typical mantissa size e.g. 40 bits, requires larger q's, in this case, of at least 121 bits. Hence, sizes of the 2 domains are vastly different. We choose this homogeneic representation, albeit it complicates the design of protocols of this kind, as this is more in line, to the available tooling, e.g. [5, 6].
More precisely, we adopt the fixed point representation introduced by Catrina and Saxena [4], used by, among others, SCALE-MAMBA, and MP-SPDZ. Additionally, we differentiate between a secret shared fixed point element ($[\![x]\!]$) and, secret integers ($\langle y \rangle$), by the use square brackets. Integers can be shared in both $\mathbb{F}_q$ and $\mathbb{Z}_{2^k}$, whereas fixed point only in base $\mathbb{F}_q$.
Inputs on $\mathbb{F}_q$, can be expressed, either implicitly ($[\![x]\!]$) or explicitly ($[\![x]\!]_q$). Whereas, inputs base 2^k, are explicitly indicated by using the suffix 2^k i.e., $[\![x]\!]_{2^k}$. Values on $\mathbb{F}_2$ follow the same principle, and use the explicit suffix 2 i.e., $[\![x]\!]_2$.
Let $[\![v]\!]$ be a mantissa bounded by some 2^l, and p the bitwise fixed point precision. We can have a $\langle x \rangle$, a secret shared fixed point number, such that:

$$\langle x \rangle = [\![v]\!] \cdot 2^p \bmod q. \tag{1}$$

For simplicity and, w.l.g., we assume $l = 40$ bits. This way multiplications can be supported whilst using the typical prime size selections, e.g. a 128 bits prime. This is because the bitsize of the multiplication (before truncation), doubles, e.g. 80 bits. Additional $\mathtt{sec}$ bits (typically $\mathtt{sec} = 40$), need to be added as well. This comes from the probabilistic truncation protocol introduced in [4]. Hence, in regards to the field size, q should be much larger than $2^{2 \cdot l + \mathtt{sec}}$ for $\mathbb{F}_q$.

2.2 Arithmetic Black Box

To simplify the theoretical analysis of our constructions, we abstract the underlying MPC protocols, via an ideal *arithmetic black box* ($\mathcal{F}_{ABB}$). The original concept was introduced by [13], and it is flexible enough to allow extensions. A practice, commonly used by relatively complex applications in the area e.g. [8,14,15]. Furthermore, it has been used by works of similar nature, with an emphasis of implementation, e.g. [16]. In our case, these extensions correspond to/ideally extend: fixed point arithmetic functionality and, a set of complex building blocks. We also specify the UC secure [17] realizations, we based this functionality from. We define our arithmetic black box, or $\mathcal{F}_{ABB}$, as follows:

Functionality	**Description**	**Rounds**	**Prot.**
$x \leftarrow \llbracket x \rrbracket$	Opening secret field element	1	-
$\llbracket x \rrbracket \leftarrow x$	Storing public input in a secret field element	1	-
$\llbracket z \rrbracket \leftarrow \llbracket x \rrbracket + \llbracket y \rrbracket$	Addition: of secret inputs	0	-
$\llbracket z \rrbracket \leftarrow \llbracket x \rrbracket + y$	Addition: (mixed) secret and public inputs	0	-
$\llbracket z \rrbracket \leftarrow \llbracket x \rrbracket \cdot \llbracket y \rrbracket$	Multiplication: of secret inputs	1	-
$\llbracket z \rrbracket \leftarrow \llbracket x \rrbracket \cdot y$	Multiplication: (mixed) secret and public inputs	0	-
$\langle z \rangle \leftarrow \langle x \rangle + \langle y \rangle$	Addition: secret fixed point	0	-
$\langle z \rangle \leftarrow \langle x \rangle + y$	Addition: (mixed) secret and public fixed point	0	-
$\langle z \rangle \leftarrow \llbracket x \rrbracket + \langle y \rangle$	Addition: secret fixed point with secret input	0	-
$\langle z \rangle \leftarrow \langle x \rangle \cdot \langle y \rangle$	Multiplication: secret fixed point	2	[3, 4]
$\langle z \rangle \leftarrow \langle x \rangle \cdot y$	Multiplication: (mixed) secret and public fixed point	1	[3, 4]
$\langle z \rangle \leftarrow \llbracket x \rrbracket \cdot \langle y \rangle$	Multiplication: secret fixed point with secret input	2	[3, 4]
	—Complex Building Blocks—		
$\llbracket \mathtt{r} \rrbracket_q, \llbracket \mathtt{R} \rrbracket_q, \llbracket \mathtt{r} \rrbracket_{2^k}, \llbracket \mathtt{R} \rrbracket_2 \leftarrow$ $\mathtt{get\_r\_from\_dabits\_list(size)}$	Returns daBits list of size size, and their respective combination.	0	trivial
$\llbracket \mathtt{R} \rrbracket_q, \llbracket \mathtt{R} \rrbracket_2 \leftarrow \mathtt{get\_dabits\_list(size)}$	Returns vector of daBits of size size.	0	trivial
$\llbracket \mathtt{r} \rrbracket_q, \llbracket \mathtt{r} \rrbracket_{2^k} \leftarrow$ $\mathtt{combine\_dabits}(\llbracket \mathtt{R} \rrbracket_\mathtt{q}, \llbracket \mathtt{R} \rrbracket_2)$	Returns vector combinations of $\llbracket \mathtt{R} \rrbracket_q$, $\llbracket \mathtt{R} \rrbracket_2$, daBits.	0	trivial
$\llbracket r_\kappa \rrbracket \leftarrow \mathtt{PRandInt(size)}$	Returns some randomness in $[0, 2^{\mathtt{size}})$.	0	[3, 4]
$\llbracket x \rrbracket_q \leftarrow \mathtt{conv\_sint}(\llbracket \mathtt{y} \rrbracket_2)$	Conversion from $\mathbb{Z}_2$ to $\mathbb{F}_q$.	1	[5, 7, 10]
$\llbracket c \rrbracket_q \leftarrow \mathtt{rabbit\_sint}(\llbracket \mathtt{y} \rrbracket)$	Rabbit encapsulation routine.	4	This Work
$\llbracket c \rrbracket_q \leftarrow \mathtt{bitwise\_lteq}(\llbracket y \rrbracket_q)$	bitwise LTEQZ. Constant round when implemented using GC, via Zaphod.	2	[2]

Table 1: Secure Arithmetic operations provided by the $\mathcal{F}_{ABB}$.

Invoking Rabbit (rabbit_sint($[\![y]\!]$)): For simplicity, we provide an encapsulation of the invocation of rabbit. The method is just an abstract tool, for the invocation of any of the variations of Rabbit, presented in this work. This way, protocols, that depend on them, such as our construction of a ReLU. It simply extracts the system parameters and then invokes any desired Rabbit construction. Let sint be a register type for elements in $\mathbb{F}_q$, then construction can be simply expressed as indicated by Figure 1.

```
rabbit_sint([[x]])
        //instantiate vector params
        //with system parameters for default rabbit mode.
         PARAMS[] := get_system_params(rabbit.mode);
         c := rabbit.mode(PARAMS, [[x]]);
         return: type(c) != sint ? conv_sint(c): c;
```

Fig. 1: Rabbit Encapuslation

Arithmetic Black Box in Practice. The abstraction, albeit useful, from a theoretical perspective, meets some limitations in practice. In reality an implementer have to adjust with, among other things, what is provided by the tooling, in this concrete case SCALE-MAMBA. MPC frameworks of this kind, tend to operate storing secret and public inputs, in the form of registers. Frameworks also include a selection of instructions (**RISK** and **CISC**), to interact with these registers. Additionally, the parametrization and coupling of different contributions across the field, in a single tool, are also challenging. In the most ambitious setting, for our use case, our constructions require constant round evaluation of Garbled Circuits (hence, Boolean Circuits), mixing them with arithmetic operations over Full Threshold, using LSSS protocols such as Low Gear Overdrive [11] (A member of the SPDZ's family of protocols). Zaphod [7], in this case, allows us to realize such setting, implementing in SCALE-MAMBA a variation of [12, 18], for modulo 2 arithmetic, and [11] on modulo q. Note that, SCALE-MAMBA, is the only framework currently available, that includes all the elements that are necessary to use Zaphod, hence the one used by this work.
On finer grain, we can see that, in general terms, our $\mathcal{F}_{ABB}$ defines 4 types of operations:

i). **Fixed point arithmetic:** Operations are implemented using the protocols introduced [3, 4] as presented in [5]. This representation is independent of the underyilng protocols that instantiate $\mathbb{F}_q$ arithmetic.

ii). **Integer arithmetic modulo some prime q:** Instantiated by Zaphod [7] and implemented by [5]. We consider mainly 2 flavors: Full Threshold (Overdrive, Low Gear) [11] and, Honest Majorities, with error correction [19].

iii). **Integer/boolean arithmetic modulo $\mathbf{2^k}$**: We consider the approach taken by Zaphod [7] and, implemented on [5], which is a flavor of `[HSS17]` [12], incorporating elements from [18].

iv). **Complex Building Blocks:** Such as the conversion between elements modulo 2^k and q. Implementations of these mechanisms are as indicated by their respective authors. However, we do provide a small revision in regards to the conversion, further in this section, as it is of interest for complexity analysis.

Conversion $\mathbb{Z}_2 \leftarrow \mathbb{F}_q$ (`conv_sint`($[\![\mathtt{y}]\!]_2$))*:* Our constructions are thought to be used by conventional implementations, mixing with fixed point operations. Just in the same way the results from Rabbit [2], the protocols in this work return results on $\mathbb{F}_2$. However they can be trivially converted to $\mathbb{F}_q$, by using a single daBit. This process is also explained in `SCALE-MAMBA`'s documentation, and is derived from [10]. Let $[\![y]\!]_2$ be a secret element in $\mathbb{F}_2$. We locally calculate the `xor` , between a fresh daBit (modulo 2) and $[\![y]\!]_2$. We then, simply open the result and `xor` the result again, just that this time with the daBits modulo q. The process requires a single round. We refer the reader to [5], for a more detailed description.

2.3 Implementation Challenges

Modulo Size We prioritize prime selection for other environmental needs rather than comparisons. Moreover, in our experimentation we do not exploit the natural advantages of working in 64 bits. We are more interested on scenarios, where implementations of ReLU's, require a field with a bigger modulo. Essentially, where multiplications, of the form discussed above, are present and, more importantly, fixed point number representations, is already present, and imposing its own limitations. In practice, this means that, realistic uses for, ReLU's (with mantissas, that are larger than 40 bits) require longer fields e.g. 128 bits.
One might be tempted to assume, that, regardless the size of q, we could work directly on 64 bits, exclusively when computing INQ. Trivially, this could be enabled via field conversion, which would offset, any benefit acquired by the field reduction. We test an analogous proposition via the implementation of the ReLU optimized Rabbit variation, introduced by [2].

Domain Shifting We give special attention to the number domain. That is, the interaction between the numeric domain used in $\mathbb{F}_q$ and $\mathbb{Z}_{2^k}$. This is further complicated by the fact that some of Rabbit constructions consider that the sign is directly encoded in the Most Significative Bit (MSB). Note that,given our setup, protocols need to mix values that are in different domains.

2.4 ReLU

Rectified Linear Unit functions are simple activation functions, popular in deep learning. They are specially interesting for PPML, given that their linearity, makes their performance more competitive versus sigmoidal functions. A ReLU can be defined as follows:

$$\mathsf{ReLU}(x) = \begin{cases} 0 & \text{for}(x < 0); \\ x & \text{for}(x \geq 0). \end{cases} \tag{2}$$

It is clear that a ReLU can be derived from any LTEQZ protocol. For context, a ReLU activation layer is part of a whole Neural Network architecture, composed of linear transformations, folding layers and more complex activation functions, frequently based on exponential operations (e.g. sigmoid or softmax). Thus, the LTEQZ protocol adopted for ReLU must be compatible with the MPC protocols supporting the operations for the other layers. This implies slack and fixed point arithmetic [4].

3 Ad(a/o)pting Rabbit

In this section, we explore the basic building blocks an ideas behind Rabbit [2], as well as our propose variations and various aspects, necessary for adoption in practice.

3.1 Inequality Tests

We mainly concern with one specific instance of INQ, more specifically, Less than Equal Zero (LTEQZ) tests. By using LTEQZ we can trivially instantiate any other INQ, in various configurations. However, and for the purpose of this work, and the applications tackle, we can define the problem as follows:

$$\mathsf{LTEQ}(x, 0) \rightarrow \mathsf{LTEQZ}(x) : \mathbb{Z} \rightarrow \{0, 1\} \subseteq: \begin{cases} \mathsf{LTEQZ}(x) = 1 & \text{if}(x \leq 0); \\ \mathsf{LTEQZ}(x) = 0 & \text{otherwise.} \end{cases} \tag{3}$$

The problem The state of the art proposes mechanisms for comparison, mainly based on bit decomposition. Such constructions tend to be expensive, in terms of multiplicative depth and work for both, MPC and HE schemes. They also tend to be sublinear with some associated non-negligible constant cost.
To solve this issue, the literature has proposed to relax the security model, as a trade-off for efficiency. The principle is to reduce the impact of bit decomposition in exchange of security under statistical constraints, i.e. statistical security. This statistical security should not be confused with cryptographic security, and it rather relates to the probability distribution over masked inputs.
The current state of the art (the protocol that is discussed in this section) addresses this, reducing the impact of bit decomposition overall and keeping the more robust security model than previous works.

The Rabbit Principle The method itself is based on the commutative properties of addition. It formulates 2 different but equivalent equations. Generally speaking, it derives a simplified algebraic construction, comprised of 3 INQ, one of which, relies solely on public inputs, whereas, the other 2 partially depend on public inputs as well and can be executed in parallel. Furthermore, the secret shared inputs are some precomputed randomness.
To make it competitive against statistical secure constructions, the authors make use of $\mathsf{edaBits}$ [9]. For the purpose of this section, we abstract $\mathsf{edaBits}$, as a construction that allows us to generate some randomness in $\mathbb{Z}_{2^k}$ and its bit expansion, at the same time, over $\mathbb{Z}_2$.
Let $\mathbb{Z}_M$ be some commutative ring bounded by $M \in \mathbb{Z}$, $\llbracket x \rrbracket$ and $\llbracket r \rrbracket$, some secret shared input and randomness in $\mathbb{Z}_M$ and, R be some public element of $\mathbb{Z}_M$. Then, we can establish the following:

$$
\begin{aligned}
B &= M - R, &(4)\\
\llbracket a \rrbracket &= \llbracket x \rrbracket + \llbracket r \rrbracket, &(5)\\
\llbracket b \rrbracket &= \llbracket x \rrbracket + \llbracket r \rrbracket + B, &(6)\\
\llbracket c \rrbracket &= \llbracket x \rrbracket + B. &(7)
\end{aligned}
$$

If we also observe carefully the constructions above, we can appreciate that B is simply the complement of R. Before, we discuss the algebraic elements of Rabbit, let us, now consider the following statement:

$$
[x+y] = \begin{cases} \llbracket x \rrbracket + \llbracket y \rrbracket - M \cdot \mathsf{LT}(\llbracket x+y \rrbracket, \llbracket x \rrbracket); \\ \llbracket x \rrbracket + \llbracket y \rrbracket - M \cdot \mathsf{LT}(\llbracket x+y \rrbracket, \llbracket y \rrbracket). \end{cases} \tag{8}
$$

Which is always true for any element of $\mathbb{Z}_M$. Given the equations above, we can establish the following relations, let us start with $[a+B]$:

$$
\begin{aligned}
\llbracket b \rrbracket &= \llbracket a + B \rrbracket \\
&= \llbracket a \rrbracket + B - M \cdot (\llbracket a + B \rrbracket < B) \\
&= \llbracket x \rrbracket + \llbracket r \rrbracket - M(\llbracket x + r \rrbracket < r) + B - M \cdot (\llbracket a + B \rrbracket < B).
\end{aligned}
$$

When we expand $\llbracket c + r \rrbracket$ in the same fashion we can derive the following:

$$
\begin{aligned}
\llbracket b \rrbracket &= \llbracket c + r \rrbracket \\
&= \llbracket c \rrbracket + r - M \cdot (\llbracket c + r \rrbracket < \llbracket r \rrbracket) \\
&= \llbracket x \rrbracket + \llbracket b \rrbracket - M \cdot (\llbracket x + B \rrbracket < \llbracket r \rrbracket) + \llbracket r \rrbracket - M \cdot (\llbracket c + r \rrbracket < \llbracket r \rrbracket).
\end{aligned}
$$

If we equate both expansions of $\llbracket b \rrbracket$, using $\llbracket a + B \rrbracket$ and $\llbracket c \rrbracket + \llbracket r \rrbracket$, we can obtain the following (after simplifications):

$$
\llbracket a < r \rrbracket + \llbracket b < B \rrbracket = \llbracket c < B \rrbracket + \llbracket b < r \rrbracket.
$$

Let us now replace a, b and c, and express the equation above, in terms of $[\![x]\!]$ and $[\![r]\!]$:

$$[\![x + r]\!] < [\![r]\!] + [\![c + r]\!] < B = [\![x + B]\!] < B + [\![x + r + B]\!] < [\![r]\!],$$
$$[\![x + B]\!] < B = [\![x + r]\!] < [\![r]\!] + [\![x + B + r]\!] < B - [\![x + r + B]\!] < [\![r]\!].$$

The INQ that we have conveniently now placed on the left of the equation, expresses the relation between x and the complement of R. In this case the INQ would be true, only if x is greater than R, minding we are still working on $\mathbb{Z}_M$, as any excess over R would force an overflow, and a subsequent wraparound. Consider that R is some public element of $\mathbb{Z}_M$, including 0. From this point, we can trivially build the LTEQZ(x) functionality, as described at the beginning of this section.

Given that we can freely disclose masked secret $[\![x]\!] + [\![r]\!]$ without compromising security, the equation above would finally look like:

$$[\![x + B]\!] < B = (x + r) < [\![r]\!] + (x + B + r) < B - (x + r + B) < [\![r]\!].$$

As previously stated both inequalities can be calculated in parallel, and the tests themselves, implemented bitwise, by using an offline phase, or edaBits directly. This method has sublinear performance, without any associated hidden cost.

3.2 Rabbit Variations

We focus our attention on 2 of the main protocols introduced by [2]. First, the basic test against a constant value (what we call, conventional Rabbit) and, its ReLU optimized version (optimized Rabbit). We propose 2 flavors for the former, and provide some discussion and benchmarking for the latter. Note that,all variations proposed in this and further sections, return LTEQZ. INQ tests can then be trivially derived.

Conventional Rabbit: Following the discussion from [2]: Consider the distribution of the mask. It is clear, that the vanilla Rabbit construction, as presented by the authors could not achieve perfect security,[3] when implemented over $\mathbb{F}_q$. To address this, authors discussed possible alternatives such that i). selecting a prime close to a power of 2 and, ii). the use of a rejection list for sampling. We, therefore, explore these 2 scenarios, and present constant round protocol flavors of both variations.Note that our designs are also application oriented.

Probabilistic (Statistical Security): We propose a version of Rabbit, in line (as is) with the original contributions outlined by [2]. The main difference being, the use of constant round building blocks.

[3] Furthermore, in practice, realizations of the underlying protocols for share conversion, can only achieve statistical security.

Notice that the selection of the prime field $\mathbb{F}_q$ is of importance in realistic settings. For security reasons, it has to be close to a power of 2. This is motivated by the fact that, as indicated by the original work [2], the gap defined as $|q - \lceil \log_2(q) \rceil|$ affects its security. Furthermore, depending of the implementation, it could also affect correctness given that oversampling of the bitwise mask could cause incongruousness, i.e. sampled randomness is bigger than q causing discrepancies between its value in $\mathbb{F}_q$ and its bit expansion. We note, our design addresses this latter concern as well.

Protocol 1: Simple Rabbit (rabbit_fp)

Input: The prime: (q), the prime flag: (above), and secret input: ($\llbracket x \rrbracket_q$).
Output: secret shared LTEQZ (x) in $\mathbb{F}_2$
Pre:

```
1  $k = \texttt{int}(\lfloor \log_2(q) \rfloor) + 1$; // int converts to integer
2  $\mathtt{R} = \lceil (q-1)/2 \rceil$
3  if above == 1 then
4      $\llbracket \mathtt{r} \rrbracket_q, \llbracket \mathtt{R} \rrbracket_q, \llbracket \mathtt{r} \rrbracket_{2^k}, \llbracket \mathtt{R} \rrbracket_2 \leftarrow$ get_r_from_dabits_list(k);
5  else
6      $\llbracket \mathtt{R} \rrbracket_q, \llbracket \mathtt{R} \rrbracket_2 \leftarrow \{0\}^k$;
7      $\llbracket \mathtt{R} \rrbracket_q, \llbracket \mathtt{R} \rrbracket_2 \leftarrow$ get_dabits_list(k − 1);
8      $\llbracket \mathtt{r} \rrbracket_q, \llbracket \mathtt{r} \rrbracket_{2^k} \leftarrow$ combine_dabits($\llbracket \mathtt{R} \rrbracket_\mathtt{q}, \llbracket \mathtt{R} \rrbracket_2$);
   Pos:
9  $\llbracket a \rrbracket \leftarrow (\llbracket \mathtt{r} \rrbracket_q + \llbracket x \rrbracket_q)$;
10 $a \leftarrow \llbracket a \rrbracket$;
11 $b = (a + q - \mathtt{R})$;
12 $\llbracket w_1 \rrbracket_2 \leftarrow$ bitwise_lteq $(a, \llbracket \mathtt{R} \rrbracket_2, k)$;
13 $\llbracket w_2 \rrbracket_2 \leftarrow$ bitwise_lteq $(b, \llbracket \mathtt{R} \rrbracket_2, k)$;
14 $\llbracket w_3 \rrbracket_2 = b + \mathtt{R} < \texttt{cint}(q - \mathtt{R}) + \mathtt{R}$; //cint converts to register
15 return $\llbracket w_1 \rrbracket_2 - \llbracket w_2 \rrbracket_2 + \llbracket w_3 \rrbracket$;
16 in return conv_sint converts to $\mathbb{F}_q$
```

PROTOCOL 1: Rabbit for $\mathbb{F}_q$ with statistical security

Correctness: The protocol is correct, as demonstrated by [2], using the principles explained earlier in this section. We also control for oversampling.

Complexity: When implemented, using Zaphod, complexity is constant, with 2 parallelizabe invocations of the bitwise_lteq, which translate in 2 rounds. It includes 1 opening. Note that the result given is modulo 2, hence an extra round for conversion is be required, with 4 in total.

Discussion: As noted in previous sections, we deal with the domain shift, by adding R, in places such as *line 14*. Besides this, and for implementations purposes, we have to also to trivially adapt the protocol, so that it can discard/reduce a bit from the randomness sampling. Note that, we also include some specificities from `SCALE-MAMBA`, e.g. (`cint` invocation to transform input to a `SCALE-MAMBA` register). *Security* follows directly from [2]. i.e, Simulation remains the same, as we add no opening nor leakage, and the $\mathcal{F}_{ABB}$ is modeled as an ideal functionality, thus trivially secure under composition (As in the original proofs, we remind the reader that the hybrid model allows us to replace UC secure realizations for ideal ones [17]). This is also true for all constructions present in this work, hence we do not come back to it later. Note that, because of the nature of the sampling (which is not in $\mathbb{F}_q$), security is rather statistical in nature.

Rejection List: We extend the previous construction by eliminating the probabilistic sampling, and incorporating a rejection list. We show how to perform this inclusion in a way that, from an ideal perspective, could give this construction perfect security. The protocol is described as follows:

Protocol 2: Rejection List Rabbit (`req_list`)

```
Input: The prime: ($q$), word size: ($k$) i.e. 64,
and secret input ($[\![x]\!]_q$).
Output: secret shared LTEQZ (x) in $\mathbb{F}_2$
Pre:
$k = \texttt{int}(\lfloor \log_2(q) \rfloor) + 1$; // int converts to integer
$\texttt{R} = \lceil (q-1)/2 \rceil$;
repeat
    $[\![\texttt{R}]\!]_q, [\![\texttt{R}]\!]_2 \leftarrow \texttt{fill\_dabits\_array}(\texttt{k})$;
    $[\![\texttt{above}]\!]_2 \leftarrow \texttt{bitwise\_lteq}\ (q, [\![\texttt{R}]\!]_2, k)$;
    $\texttt{above} \leftarrow [\![\texttt{above}]\!]_2$;
until ($\texttt{above} == 1$);
$[\![\texttt{r}]\!]_q, [\![\texttt{r}]\!]_{2^k} \leftarrow \texttt{combine\_dabits}([\![\texttt{R}]\!]_\texttt{q}, [\![\texttt{R}]\!]_2)$;
Pos:
$[\![a]\!] \leftarrow ([\![\texttt{r}]\!]_q + [\![x]\!]_q)$;
$a \leftarrow [\![a]\!]$;
$b = (a + q - \texttt{R})$;
$[\![w_1]\!]_2 \leftarrow \texttt{bitwise\_lteq}\ (a, [\![\texttt{R}]\!]_2, k)$;
$[\![w_2]\!]_2 \leftarrow \texttt{bitwise\_lteq}\ (b, [\![\texttt{R}]\!]_2, k)$;
$[\![w_3]\!]_2 = b + \texttt{R} < \texttt{cint}(q - \texttt{R}) + \texttt{R}$; //cint converts to register
return $[\![w_1]\!]_2 - [\![w_2]\!]_2 + [\![w_3]\!]$;
//in return conv_sint converts to $\mathbb{F}_q$
```

Protocol 2: *Perfect Secure* Rabbit for $\mathbb{F}_q$.

Correctness: The correctness, once again follows from [2]. In this case, no further adaptations to the sampling are needed, as it always returns some randomness bounded by q. After sampling, Rabbit then proceeds as expected.

Complexity: We still have the previous 3 basic invocations (one opening and 2 rounds from the parallel calls to `bitwise_lteq`. However, because of the rejection list (that involves additional calls to `bitwise_lteq`), performance now depends on the prime, where probability (ϕ) of every subsequent 2 rounds can be defined as:

$$\phi = \frac{2^{\lceil \log_2(q) \rceil} - q}{2^{\lceil \log_2(q) \rceil}}. \tag{9}$$

Just as in previous cases, the protocol delivers an output on modulo 2. Hence, 1 additional round for conversion to modulo q should be added.

Discussion: As before, we incorporate elements that are relatively important the protocol implementation, in `SCALE-MAMBA`. Note that in this case, the prime selection is also important i.e. the prime is not far below the immediate next power of 2, as the sampling, will have a lower probability of requiring extra rounds per execution. *Security*, also follows directly from [2]. Moreover, the incorporation of the rejection list, is the only method available with which any Rabbit variation can (ideally) achieve prefect security in $\mathbb{F}_q$.

Optimized* Rabbit *for* ReLU *'s: Makri et al. [2], also explore, an optimized comparison protocol, specifically for ReLU's. Assuming words w.l.g., of 64 bits, they imagine scenarios where the ring size is compatible with the aforementioned word size. Whilst using a specific number representation, the protocol itself is quite different to the constructions we present in this paper. In fairly general terms, and cater to our needs, it can be summarized as follows:

1. Conversion of some secret $[\![x]\!]_q$ to modulo 2^k (if needed).
2. Extract the MSB from $[\![x]\!]_{2^k}$.
3. Interpret and return the MSB as the result of the comparison.

To make it compatible with $\mathbb{F}_q$, and [4] (fixed point arithmetic), we rely on the rather expensive primitive $\mathbb{F}_q$ to $\mathbb{Z}2^{64}$ conversion from Zaphod, also described in `SCALE-MAMBA`. We have implemented this variation as well, for benchmarking purposes. We call our construction `rabbit_conv`, and it uses the already built-in primitives included in `SCALE-MAMBA`, for conversion. No further adaptations are required, given that the framework also uses words of 64 bits.

Slack Rabbit One of the contributions of this work is the inclusion of a variation of Rabbit, that reduces the sizes of the bitwise inequality tests `bitwise_lteq`. It achieves this, by relaxing the security of the protocol to statistical. Basically, the protocol reduces the number of daBits the protocol requires (whilst still working on larger q's. Further adaptations were required to incorporate, among other things, shifts in the numeric domain, that originated by the now shorter circuits, as we later explain. The protocol is described as follows:

Protocol 3: Slack Rabbit (`rabbit_slack`)

```
Input: The prime: ($q$), input size: ($k$), sec param: ($\kappa$),
and secret input: ($[\![x]\!]_q$).
Output: secret shared LTEQZ (x) in $\mathbb{F}_2$
Pre:
$\mathtt{M} = 2^k$;
$\mathtt{R} = \lceil \frac{\mathtt{M}_2}{2} \rceil - 1$;
$[\![r_\kappa]\!] \leftarrow \mathtt{PRandInt}(\kappa)$;
$[\![\mathtt{r}]\!]_q, [\![\mathtt{R}]\!]_q, [\![\mathtt{r}]\!]_{2^k}, [\![\mathtt{R}]\!]_2 \leftarrow \mathtt{get\_r\_from\_dabits\_list}(\mathtt{k})$;
Pos:
$[\![z]\!] \leftarrow [\![x]\!]_q + \mathtt{R}$;
$[\![a]\!] \leftarrow ([\![\mathtt{r}]\!]_q + [\![z]\!]) + [\![r_\kappa]\!] \cdot 2^{\mathtt{k}}$;
$a \leftarrow [\![a]\!]$;
$b \leftarrow a + \mathtt{M} - \mathtt{R}$;
$\mathtt{a}' = a \ \mathtt{mod}\ 2^{\mathtt{k}}$;
$\mathtt{b}' = b \ \mathtt{mod}\ 2^{\mathtt{k}}$;
//comparisons
$[\![w_1]\!]_2 \leftarrow \mathtt{bitwise\_lteq}\ (a', [\![\mathtt{R}]\!]_2, k)$;
$[\![w_2]\!]_2 \leftarrow \mathtt{bitwise\_lteq}\ (b', [\![\mathtt{R}]\!]_2, k)$;
$[\![w_3]\!] = b - \mathtt{cint}(\mathtt{M} + \mathtt{R}) < 0$; //cint converts to register
return $1 - ([\![w_1]\!]_2 - [\![w_2]\!]_2 + [\![w_3]\!])$;
```

PROTOCOL 3: Rabbit on statistical security.

Correctness: The basic inner workings of the protocol use the same Rabbit principle, explored in previous sections, hence it follows from [2]. Changes were applied to the generation of the random mask. To guarantee it correctly masks inputs, we follow the common construction process utilized by [3, 20]. This in turn, reduces the security of the protocol from abstract *perfect security*, to a statistically secure on the size of security parameter κ.

Complexity: Complexity remains the same, i.e. constant round $\mathcal{O}(1)$. The improvement comes from a reduction on the number of daBits invocations. In this case we require k, instead of $\lceil \log_2(q) \rceil$. Furthermore, the size of the circuit used by the `bitwise_lteq`'s invocations is reduced to k, e.g. 64 bits. Given that the bottleneck is no longer communication, upticks on, CPU time, RAM, and preprocessing, could be of importance.

Discussion: This version incorporates some elements of Lipmaa and Toft [20] and Catrina and Hoogh [3]. A preprocessing phase reconstructs some bounded randomness using a mix of k daBits (that are later used as the bit expansion of the masked randomness in $\mathbb{Z}_{2^k}$, i.e. 64 bits) and `PRandInt`, of the size of our security parameter κ. After some domain switching (see Section 2.1), which is further complicated by the types limitations in `SCALE-MAMBA`, we can then proceed to

the familiar flow of Rabbit, but over k, instead of q. *Security* Follows directly, from the discussion on [3, 20], as there is no further openings, nor leakage.
Note that the protocol below could be further customized to include smaller sizes smaller domains than 64 bits. Randomness generation parameters, could be then tuned, accordingly so that the bitsize of comparison can be reduced. This could be achieved by invoking $\texttt{PRandInt}(\mathtt{k} + \kappa - \mathtt{m})$, instead of κ.
Note that the bitwise comparisons are done in absolute terms. As mentioned, frameworks like `SCALE-MAMBA` require some level of encoding to support negative values whilst using native types. Hence, Some domain shifting is required. This is indeed the case in our implementation.

Selection Criteria for ReLU Implementation The original Rabbit protocol, requires comparisons to be done against edaBits directly, for which the protocol already holds a bit decomposition. This, does not hold when the extraction of the MSB is considered, as removing the mask in modulo 2^k itself requires a subtraction modulo mod p.

There are 2 circumstances, however when the optimization becomes more efficient:

1. Inputs and Masks are bounded (elements of $\mathbb{Z}_{\langle k \rangle}$ and $\mathbb{Z}_{k+\kappa}$, respectively). implying statistical security on the size of κ. In this case no mod p arithmetic is required.
2. All values are elements of some space $\mathbb{Z}_{\langle k \rangle}$, as it is indeed the case on $\texttt{SPDZ}_{2^k}$. Noting, the implementation of security related primitives becomes cumbersome, and implementations scarce, as well the limitations that might appear during preprocessing.

If the scenario contemplates any of the items exposed above, the recommendation is to implement the ReLU, based on the extraction of the MSB.

4 Constant Round Catrina and De Hoogh

In this section, we explore a one round variation of the commonly used LTZ formulation from Catrina and De Hoogh [3], adapted with some of the enhances introduced by Rabbit. To achieve this, we once again make use Zaphod, and present 2 protocols i).to solve the modulo 2^m of any secret shared input and, ii). A LTZ adaptation that summarizes some of the contributions from [3].

4.1 Modulo 2^m (`mod2m`)

We start, by discussing our `mod2m` construction, in Protocol 4. It reflects the same process flow of its namesake protocol from Catrina and De Hoogh [3]. However, we introduced some notable changes:

- **Randomness**: We sample the random mask using daBits, in the first k bits, instead of using the 1 round construction from [8]. Note how Escudero et al. in [9], when adapting Catrina and De Hoogh results, replace them, using edaBits instead. One key difference, however, is that we keep the original call to PRandInt(k′), to construct the mask material that is not subject to the comparison.

- **Bitwise INQ**: We replace the original bitwise LTZ provided by Catrina and De Hoogh, by the low complexity LTEQZ test provided by Rabbit. This change forces us, however to further adapt the protocol, so that it can still mimic the original result. To achieve this, we manipulate the daBits randomness material in $\mathbb{F}_q$.
- **Conversion to $\mathbb{F}_q$**: The result of the bitwise comparison, is obtained via Zaphod, using, for instance, Garbled Circuits. Hence, we invoke the 1 round / 1 daBits, conversion protocol (conv_sint($[\![x]\!]$)), to transform the result back to $\mathbb{F}_q$, and be able to adhere to the process flow, from the original Catrina and De Hoogh results.

The protocol is denoted as follows:

Protocol 4: constant round mod 2^m (mod2m)

```
Input: the bitlength of inputs:(k), a power of Two: ($m$), sec
    param: ($\kappa$), and secret input: ($[\![x]\!]_p$).
Output: secret shared LTZ (x) in $\mathbb{F}_q$
Pre:
1  // size of $\kappa$ + non_dabits bits,
2  // if any e.g. 40 + 1
3  $\mathtt{k}' = \mathtt{k} + \kappa - \mathtt{m}$;
4  $[\![r_\kappa]\!] \leftarrow$ PRandInt(k′);
5  $[\![\mathtt{r}]\!]_q, [\![\mathtt{R}]\!]_q, [\![\mathtt{r}]\!]_{2^k}, [\![\mathtt{R}]\!]_2 \leftarrow$ get_r_from_dabits_list(m);
6  $[\![\mathtt{r}]\!]_q = [\![\mathtt{r}]\!]_q + 1$;
Pos:
7  $[\![a]\!] \leftarrow 2^{k-1} + [\![\mathtt{r}]\!]_q + [\![x]\!]_q + [\![r_\kappa]\!] \cdot 2^{\mathtt{m}}$;
8  $a \leftarrow [\![a]\!]$;
9  $a' = a \bmod 2^m$;
10 $[\![w]\!]_2 \leftarrow$ bitwise_lteq $(a', [\![\mathtt{R}]\!]_2, k)$;
11 $[\![w]\!]_q \leftarrow$ conv_sint($[\![\mathtt{w}]\!]_2$) ;
12 return $a' - [\![\mathtt{r}]\!]_q + (2^m) \cdot [\![w]\!]_q$;
```

PROTOCOL 4: Constant round Catrina and De Hoogh mod 2^m .

Correctness: The protocol replicates the process from Catrina and De Hoogh [3]. The main source of concern is the change of the bitwise INQ test from LTZ to LTEQZ. This is addressed in *line 6.*. There, we add a 1 to the value is secretly contained in $[\![\mathtt{r}]\!]_q$. This small change then propagates to a. as a result the composition of a' is altered as it might contain $[\![\mathtt{r}]\!]_q+1$ instead of $[\![\mathtt{r}]\!]_q$. What this means in practice, is that the comparison below, when using the unaltered modulo 2 version of the mask, behaves just as the original LTZ would do. This achieves to subside any issue originated by the switch, given that, all related behaviour is the same. Besides the LTEQZ, other changes introduced to the protocol are orthogonal to its correctness.

Complexity: Protocol has a constant round count, similar to other protocols studied in this paper (4 rounds). Caused by an opening, 1 invocation of a GC circuit, and a 1 extra round for the conversion from*line 11.* Results are delivered in $\mathbb{F}_q$, hence no further changes are required.

Discussion: Protocol keeps the basic and simple flow, that made it so popular to use with fixed point values. Given that, in principle, the protocol flow did not change, this means: *Security* remains unscaved from [3]. Meaning the statistical security of the protocol is the same. In practice, the typical 40 bits of security present in frameworks such as SCALE-MAMBA, can remain unchanged. Simulation also remains the same, given that there are no additional openings or leakage, and the generation of the random mask and conversion can be modeled as ideal functionalities. Finally, we could envision a circuit that merge *line 10*, and *line 11*, and thus further reducing the round count by 1.

4.2 Catrina and De Hoogh Constant Round LTZ (cons_ltz)

We can now make use of our mod2m construction, to build the INQ protocol introduced by Catrina and De Hoogh. For simplicity, we sintetize thir results on a single protocol, replacing the modulo 2^k invocation, by our own mod2m. Traditionally, Catrina and de Hoogh construction returns LTZ. We notice, however, that it can be trivially adapted to return a LTEQZ (reverting the input size and negating the output). The Protocol can be constructed as follows:

Protocol 5: constant round LTZ (cons_ltz)

```
Input: the bitlength of inputs:(k) sec param: ($\kappa$), and secret
       input: ($[\![x]\!]_q$).
Output: secret shared LTZ (x) in $\mathbb{F}_q$
$[\![x']\!] \leftarrow$ mod2m ($k, k-1, \kappa, [\![x]\!]$);
$[\![z]\!] = \frac{[\![x]\!]-[\![x']\!]}{2^m}$ mod $q$;
return $-[\![z]\!]$;
```

Protocol 5: Constant round Catrina and Hoogh.

Correctness: It follows from [3]. Although, we have replaced some elements in the mod2m construction, as analysed. Assuming behaviour mimics the original, this protocol is basically the integration of the truncation and the LTZ primitives introduced by Catrina and De Hoogh.

Complexity: It is bound to `mod2m`. As it does not include operations that require round based computations. So in this case it maintains in 3 rounds. Its complexity matches all Rabbit constructions, after conversion to $\mathbb{F}_q$, however, the number of invocations of `bitwise_lteq`, is reduce to 1, as previously stated.

Discussion: The protocol meant to be intuitive and simple. For simplicity, we omitted the construction of a non probabilistic truncation protocol, however it can be trivially derived from Protocol 5. *Security*, follows directly from `mod2m`, as there are no openings nor leakage originated by the protocol. As with previous works, simulation in this case is also trivial (hybrid model [17], given that, besides than the no leakage, we would only compose it with ideal functionalities.

4.3 Constant Round ReLU's

Our proposed ReLU construction, follows the same line of thought from previous sections. Indeed, Catrina and Saxena fixed point representation is heavily interlinked with the protocol. In fact, Protocol 6, optimizes the fixed point multiplication from the ReLU, by skipping the probabilistic truncation (`PRTrunc`).

In line with the definitions, introduced in Section 2.4. Our constant round ReLU can be trivially implemented as follows:

Protocol 6: constant round ReLU (relu(x))

```
  Input: $\langle x \rangle$.
  Output: secret shared relu($\langle$x$\rangle$) in $\mathbb{F}_q$
1 $[\![v]\!]_q \leftarrow [\![x.v]\!]_q$;
2 $[\![c]\!]_q \leftarrow 1 - \texttt{rabbit\_sint}(\texttt{v})_q$;
3 $[\![x.v]\!]_q \leftarrow [\![c]\!]_q \cdot [\![v]\!]_q$; // zero otherwise
4 return $\langle x \rangle$;
```

Protocol 6: Constant round protocol for ReLU's.

Complexity: The protocol has, once more, constant round complexity ($\mathcal{O}(1)$). It consists of 1 round (from the multiplication on *line 3)*, plus the selected comparison mechanism e.g.(`cons_ltz`, adds 3 rounds).

Discussion: Given that our ReLU's do not requires to invoke `PRTrunc`, they gain on performance. Hence, one might argue, that the `sec` parameter from truncation (assumed 40 bits), no longer impede us to work on smaller fields, of say 64 bits (as suggested by marki et al.). However, ReLU's, are typically surrounded by more complex fixed point operations, that do require conventional fixed point multiplications e.g. [21]. Thus, to be able to use smaller fields, e.g. 64 bits, we would require field conversion. Marbled Circuits [10] initially showed how to achieve this, but as we have also been able to see, it has a non-negligible associated cost. This is nonetheless, a more convoluted and expensive approach than using the conversion based comparison (`rabbit_conv`), introduced by [2], that we also analyze and include in our benchmark. In that sense, it would be more desirable, in terms of performance, to remain working on the larger field.

5 Benchmarking

In this section, we explore the various properties of our prototype, our testing environment, results and relevant issues, practitioners may encounter, in various settings. We benchmark comparisons, given that ReLU's, can be reduced to solving INQ efficiently. Note, that the focus of our work is adoption and practicality, hence, the importance to provide implementers with a library that is easy to incorporate in their own development, as open source.

5.1 Prototype

As mentioned in previous sections, our implementation was built on top of `SCALE-MAMBA`. This open source framework is aligned to our specific needs, it is well maintained and has an active community of developers. Furthermore, it used for experimentation for various applications, including topics related to machine learning, e.g. [16, 22]. More importantly, `SCALE-MAMBA` incorporates Zaphod [7], and a variation of `HSS17` [12]. On `LSSS`, the framework offers several protocol flavors, that can be combined with Boolean circuit evaluation via Zaphod. We are particularly interested in two: i) Low Gear Overdrive, a member of the `SPDZ`'s family [11], and refereed to as Full Threshold configuration (FT) in this section; and ii) the `SCALE-MAMBA` Shamir adaptation of Smart and Wood [19], simply referred to as Shamir in this section.
Among the different development avenues available, our prototype was built on `MAMBA`. We made this design decision, as it was more meaningful contribution for the community. Indeed, there are no high level functionality available in `MAMBA`, for manipulation of the daBits instruction. Hence, our prototype required also to build all the daBits apparatus, described by our $\mathcal{F}_{ABB}$. Our prototype comprises the following:

- Extensions to `library.py`, for random sampling using the daBits instruction.
- Extensions to `comparison.py`, where we incorporate `bitwise_lteq` constructions, using $\mathbb{Z}_2$ instructions, i.e. reactive.
- Two precompiled circuit versions `bitwise_lteq`, in Bristol Fashion, manually edited, to conform to circuit design and generated via the limited tooling already available in `SCALE-MAMBA`. Both target `rabbit_slack` and `cons_ltz`, given that they are invoked, using the `GC` instruction:
  i). `AND_XOR`: Circuit from [2], as is. Convenient for `[HSS17]` [12][4].
  ii). `AND_NOT`: A simple depth reduction. Convenient for `LSSS` [19][4].

[4] See `SCALE-MAMBA` documentation.

- Library `rabbit_lib.py`, the implementation of all INQ tests described in this paper, including all our Rabbit variations and our Catrina and De Hoogh adaptation (i.e. `cons_ltz`).
- Library `relu_lib.py`, includes variations to the basic ReLU, thought specifically for Machine Learning, including 2 and 3 dimensional ReLU invocations.
- Extensive test programs for all the functionality presented/required by this work.

For completeness, we have also incorporated in our benchmarking, the already original LTZ implementation from Catrina and De Hoogh [3], which is already in `SCALE-MAMBA`. This protocol works exclusively over LSSS in $\mathbb{F}_q$, and is fully compatible with our fixed point representation. In our tests, we refer to this protocol as `lsss_ltz`.

Finally, and as already mentioned, the prototype is fully available as open source [5].

Environment: Our test bed comprises up to 5 locally managed, physical servers connected with Gigabit LAN connections. Each one has a static fixed memory allocation of 512 GB of RAM memory. All servers are equipped with 2 `Intel(R) Xeon(R) Silver 4208 @ 2.10GH` CPUs. All machines are running Ubuntu 18. The approximate default ping time of 0.15 ms. This allows us to control network conditions,in our benchmarks, by adding and or substracting latency via `/sbin/tc`.

5.2 Results

The following results mimic the standard configuration described in this work. We use a prime q of 128 bits, with $\texttt{sec} = 40$, a matching κ and a $k = 64$ for all our protocols. The size of the mantissa for fixed point is 40. Times in this section, correspond exclusively to the online phase.

Table 2 shows the performance evaluation in one machine, with no real communications. Time complexity is entirely dominated by the computation costs. As expected, Shamir outperforms FT in all the protocols we consider. This is due to the extra costs related to operating with information theoretic macs, circuit garbling and evaluation. On the other hand, the LSSS version of Catrina and De Hoogh (`lsss_ltz`) remained mostly unaffected by the configuration. This is because it does not requires Garbled Circuits. It is worth noting that our Rabbit rejection list `req_list`, Rabbit with conversion `rabbit_conv` and simple Rabbit `rabbit_fp` protocols, significantly underperform. This phenomena is due to the large size of their corresponding circuits, for the evaluation of $\mathbb{F}_q$ elements.

Regarding the circuit choice, `AND_XOR`, underperfoms `AND_NOT`, on Shamir. This is explained by the higher circuit depth, which translates into more rounds (see [5]). On the other hand, for FT there is no meaningful difference between

[5] `https://github.com/Crypto-TII/beyond_rabbit`

Table 2: Performance evaluation in one machine

Protocol	Circuit	No Comms	
		Shamir	FT
rabbit_slack	AND_NOT	2.3ms	8.5ms
cons_ltz	AND_NOT	1.3ms	4.2ms
rabbit_slack	AND_XOR	3.1ms	8.1ms
cons_ltz	AND_XOR	1.5ms	4.0ms
rabbit_conv	-	8.4ms	12.7ms
req_list	-	44.0ms	119.7ms
rabbit_fp	-	30.7ms	64.0ms
lsss_ltz	-	1.7ms	1.8ms

both circuits when applied in both rabbit_slack and cons_ltz, given their smaller size.

From the results it is also clear that our variation of Catrina and De Hoogh, cons_ltz, outperforms the original lsss_ltz on Shamir. However, this is the opposite on FT, mainly due to the time spent garbling in cons_ltz. This could be corrected by garbling the circuit offline. Unfortunately this functionality is not compatible with the reactiveness from Zaphod, and hence, it has been postponed for future work.

Table 3 shows the results when the protocols run on several machines with real communications. Specifically, the setup considers 3 machines for Shamir and 2 machines for FT with different latencies 0.15ms, 10ms and 20ms (one-way latency). We restrict the evaluation to rabbit_slack, cons_ltz and lsss_ltz since the other Rabbit versions are clearly less efficient.

Results show that rabbit_slack and cons_ltz are more affected by delay on Shamir than on FT. Indeed, for 10ms delay, FT configuration becomes faster than Shamir. This is explained by the fact that Shamir incurs in more communication rounds for circuit evaluation. The main result however is that the classical Catrina and De Hoogh lsss_ltz significantly outperforms the other protocols. In the case of FT configuration, this may be counter-intuitive, since lsss_ltz has 7 rounds and cons_ltz only 4, hence increasing the latency should render cons_ltz more efficient. However, this is not the case because the TCP throughput is affected by latency and the quantity of transmitted data is higher in cons_ltz than in lsss_ltz.

The intuition, when implementing MPC protocols, dictates that higher latency favours protocols with less rounds. This is not necessarily the case when the communication channel implements TCP (or TLS if it is a secure channel) because TCP throughput is heavily affected by latency and packet loss rate. This is due to TCP's congestion control mechanisms and slow start procedure. As a result, sending less data in more rounds may yield better results than sending more data in less rounds. This however could be solved by applying secure communications protocols based on UDP, such as QUIC, since UDP throughput is unaffected by latency.

Table 3: Performance evaluation with 2/3 machines (FT / Shamir)

Protocol	Circuit	D=0.1ms		D=10ms		D=20ms	
		Shamir	FT	Shamir	FT	Shamir	FT
`rabbit_slack`	`AND_NOT`	8.8ms	13.4ms	623.1ms	297.6ms	1236.2ms	577.2ms
`cons_ltz`	`AND_NOT`	4.4ms	6.8ms	317.7ms	148.6ms	627.7ms	289.1ms
`rabbit_slack`	`AND_XOR`	12.8ms	16.4ms	1031.5ms	295.7ms	2045.2ms	575.2ms
`cons_ltz`	`AND_XOR`	6.2ms	8.4ms	522.0ms	148.6ms	1033.0ms	288.1ms
`lsss_ltz`	-	3.1ms	2.9ms	74.9ms	75.2ms	144.9ms	144.6ms

Additionally, compilation of SCALE-MAMBA programs uses optimizers to reduce the number of rounds. Otherwise the number of communication rounds for Catrina and de Hoogh lsss_ltz construction increases drastically. This means,in practice, the construction, would require as many rounds as multiplications, which for a 128 bit prime implies 121 rounds. The use of these optimizations, however is too inefficient at compilation time, becoming even impractical, for complex protocols such as MPC-based neural networks where the ReLU layers are applied. In such case, cons_ltz becomes reasonable alternative, as this is not the case with the constant time constructions proposed by this work.

Results also show that in general, any Rabbit variation underperforms both Catrina and De Hoogh lsss_ltz and cons_ltz, which, for small application sizes should be the choice when implementing in SCALE-MAMBA. Note that, if future framework iterations, replace the communication channels by QUIC and the garbling is done offline, cons_ltz should outperform lsss_ltz for FT configuration.

6 Conclusion, Discussion and Future Work

This paper presents a series of variations of Rabbit, adapted for machine learning applications over MPC, specifically for ReLU's, with a focus on adoption and geared towards the needs of practitioners. We considered current state of the art protocols for fixed point arithmetic, such as [4], together with the constant round evaluation of Boolean circuits thanks to Zaphod [7]. We then introduce a new protocol variation (cons_ltz), a constant round LTZ version of Catrina and De Hoogh [3], which incorporates elements from Rabbit. We accompany our findings with thorough benchmarking.
Results show that:

i). The Rabbit variations, studied in this work, under the configurations explored by the benchmarking, under-perform the original Catrina and De Hoogh lsss_ltz (which is the default comparison protocol in SCALE-MAMBA).

ii). Our constant round protocol, cons_ltz, performs better than the Rabbit variations presented, but it also under-performs with respect to the classical LSSS Catrina and De Hoogh in almost all configurations.

iii). Online garbling is a sufficiently strong factor, to justify the use of offline garbling in frameworks such as SCALE-MAMBA.

iv). When compiling without round optimization, cons_ltz is still better choice than classic Catrina and De Hoogh lsss_ltz. Thus, the importance of the open source code provided with this paper.

Alternatively, future work could contemplate an implementation of the garbling in the offline phase, and replacing TLS by QUIC, as this should suffice for cons_ltz to over-perform classical Catrina and De Hoogh lsss_ltz.
On a lighter note, this paper also raises novel questions on the use of Boolean circuits, namely regarding circuit optimization via hardware oriented frameworks. It offers interesting and unanswered research questions in an area under active investigation.

Acknowledgements

Authors would like to thank Dragos Rotaru, Titouan Tanguy, Chiara Marcolla, Eduardo Soria-Vazquez and Santos Merino, for their fruitful discussion, that undoubtedly raised the quality of the present work.

References

1. Archer, D.W., Bogdanov, D., Lindell, Y., Kamm, L., Nielsen, K., Pagter, J.I., Smart, N.P., Wright, R.N.: From keys to databases – real-world applications of secure multi-party computation. Cryptology ePrint Archive, Report 2018/450 (2018) https://eprint.iacr.org/2018/450.
2. Makri, E., Rotaru, D., Vercauteren, F., Wagh, S.: Rabbit: Efficient comparison for secure multi-party computation. In Borisov, N., Diaz, C., eds.: Financial Cryptography and Data Security, Berlin, Heidelberg, Springer Berlin Heidelberg (2021) 249–270
3. Catrina, O., de Hoogh, S.: Improved primitives for secure multiparty integer computation. In Garay, J.A., Prisco, R.D., eds.: SCN 10. Volume 6280 of LNCS., Springer, Heidelberg (September 2010) 182–199
4. Catrina, O., Saxena, A.: Secure computation with fixed-point numbers. In Sion, R., ed.: FC 2010. Volume 6052 of LNCS., Springer, Heidelberg (January 2010) 35–50
5. Aly, A., Keller, M., Orsini, E., Rotaru, D., Scholl, P., Smart, N.P., Wood, T.: SCALE and MAMBA documentation (2018) https://homes.esat.kuleuven.be/~nsmart/SCALE/.
6. Keller, M.: MP-SPDZ: A versatile framework for multi-party computation. In Ligatti, J., Ou, X., Katz, J., Vigna, G., eds.: ACM CCS 2020, ACM Press (November 2020) 1575–1590
7. Aly, A., Orsini, E., Rotaru, D., Smart, N.P., Wood, T.: Zaphod: Efficiently combining lsss and garbled circuits in scale. In: Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography. WAHC'19, New York, NY, USA, Association for Computing Machinery (2019) 33–44

8. Damgård, I., Fitzi, M., Kiltz, E., Nielsen, J.B., Toft, T.: Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In: Theory of Cryptography Conference, Springer (2006) 285–304
9. Escudero, D., Ghosh, S., Keller, M., Rachuri, R., Scholl, P.: Improved primitives for MPC over mixed arithmetic-binary circuits. In Micciancio, D., Ristenpart, T., eds.: CRYPTO 2020, Part II. Volume 12171 of LNCS., Springer, Heidelberg (August 2020) 823–852
10. Rotaru, D., Wood, T.: MArBled circuits: Mixing arithmetic and Boolean circuits with active security. In Hao, F., Ruj, S., Sen Gupta, S., eds.: INDOCRYPT 2019. Volume 11898 of LNCS., Springer, Heidelberg (December 2019) 227–249
11. Keller, M., Pastro, V., Rotaru, D.: Overdrive: Making SPDZ great again. In Nielsen, J.B., Rijmen, V., eds.: EUROCRYPT 2018, Part III. Volume 10822 of LNCS., Springer, Heidelberg (April / May 2018) 158–189
12. Hazay, C., Scholl, P., Soria-Vazquez, E.: Low cost constant round MPC combining BMR and oblivious transfer. In Takagi, T., Peyrin, T., eds.: ASIACRYPT 2017, Part I. Volume 10624 of LNCS., Springer, Heidelberg (December 2017) 598–628
13. Damgård, I., Nielsen, J.B.: Universally composable efficient multiparty computation from threshold homomorphic encryption. In Boneh, D., ed.: CRYPTO 2003. Volume 2729 of LNCS., Springer, Heidelberg (August 2003) 247–264
14. Aly, A., Abidin, A., Nikova, S.: Practically efficient secure distributed exponentiation without bit-decomposition. In Meiklejohn, S., Sako, K., eds.: FC 2018. Volume 10957 of LNCS., Springer, Heidelberg (February / March 2018) 291–309
15. Atapoor, S., Smart, N.P., Alaoui, Y.T.: Private liquidity matching using mpc. In Galbraith, S.D., ed.: Topics in Cryptology – CT-RSA 2022, Cham, Springer International Publishing (2022) 96–119
16. Aly, A., Smart, N.P.: Benchmarking privacy preserving scientific operations. In Deng, R.H., Gauthier-Umaña, V., Ochoa, M., Yung, M., eds.: ACNS 19. Volume 11464 of LNCS., Springer, Heidelberg (June 2019) 509–529
17. Canetti, R.: Security and composition of multiparty cryptographic protocols. Journal of Cryptology **13**(1) (January 2000) 143–202
18. Wang, X., Ranellucci, S., Katz, J.: Authenticated garbling and efficient maliciously secure two-party computation. In Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D., eds.: ACM CCS 2017, ACM Press (October / November 2017) 21–37
19. Smart, N.P., Wood, T.: Error detection in monotone span programs with application to communication-efficient multi-party computation. In Matsui, M., ed.: CT-RSA 2019. Volume 11405 of LNCS., Springer, Heidelberg (March 2019) 210–229
20. Lipmaa, H., Toft, T.: Secure equality and greater-than tests with sublinear online complexity. In Fomin, F.V., Freivalds, R., Kwiatkowska, M.Z., Peleg, D., eds.: ICALP 2013, Part II. Volume 7966 of LNCS., Springer, Heidelberg (July 2013) 645–656
21. LeCun, Y., Boser, B., Denker, J.S., Henderson, D., Howard, R.E., Hubbard, W., Jackel, L.D.: Backpropagation Applied to Handwritten Zip Code Recognition. Neural Computation **1**(4) (12 1989) 541–551
22. Makri, E., Rotaru, D., Smart, N.P., Vercauteren, F.: EPIC: Efficient private image classification (or: Learning from the masters). In Matsui, M., ed.: CT-RSA 2019. Volume 11405 of LNCS., Springer, Heidelberg (March 2019) 473–492