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

Abdelrahaman Aly[1,2] ⓘ, Kashif Nawaz[1] ⓘ, Eugenio Salazar[1] ⓘ, and Victor Sucasas[1] ⓘ

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

**Abstract.** Comparisons or Inequality Tests are an essential building block of Rectified Linear Unit functions (**ReLU**'s), ever more present in Machine Learning, specifically in Neural Networks. Motivated by the increasing interest in privacy-preserving Artificial Intelligence, we explore the current state of the art of privacy preserving comparisons over Multi-Party Computation (MPC). We then introduce constant round variations and combinations, which are compatible with customary fixed point arithmetic over MPC. Our main focus is implementation and benchmarking; hence, we showcase our contributions via an open source library, compatible with current MPC software tools. Furthermore, we include a comprehensive comparative analysis on various adversarial settings. Our results improve running times in practical scenarios. Finally, we offer conclusions about the viability of these 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 across 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 to build realistic applications. New results followed, introducing novel approaches to achieve basic functionalities for the vast array of applications that could make use of MPC.

Among them, Privacy Preserving Machine Learning (PPML) has gathered attention, due to the strong privacy requirements from machine learning model owners and clients. This has highlighted the need for faster performing protocols for a variety of different tasks, especially for inequality tests(INQ) and fixed point arithmetic. Note how both of them are indispensable building blocks to

implement activation functions such as ReLU's. We can mention for instance, the contributions from Makri et al [1] and Catrina and De Hoogh [2] on INQ tests and, Catrina et al. [3] on fixed point arithmetic.

Modern Multiparty frameworks, such as the popular SCALE-MAMBA [4] or MP-SPDZ [5], already incorporate some of these results, (specifically [2, 3]). The aim of these tools is to give developers the opportunity to build and deploy applications that incorporate state of the art MPC. In this work, we continue these efforts and provide researchers and implementers with a more complete view of the state of the art of INQ protocols. Among them, the results from Makri et al., commonly referred to as Rabbit, as well as other families of comparisons. We go a step further and, besides the conventional flavors, we offer our own variations and combinations, improving performance. Notice that our main interest is the implementation of fast performing ReLU's (for which we also include a basic formulation in Appendix Section 2).

Furthermore, and thanks to recent contributions such as Zaphod [6], we show how comparison protocols can be implemented in constant rounds[3], using mixed circuits in a multiparty setting.

Finally, we aim to provide implementers with useful tools for protocol selection. For this reason, we have included an exhaustive benchmarking (under various adversarial settings) to 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

Putting aside the large existing body of work related to secure comparisons for MPC for instance, the seminal work from Damgård et al. [7], we focus our attention on both of the most recent contributions that motivated this work, i.e. Makri et al. [1] or Rabbit, and Escudero et al. [8]. Both introduced comparison mechanisms that reduce communication complexity, by mixing Boolean and Arithmetic circuit evaluations via edaBits. In this section we discuss some of the key differences:

*On Fixed Point and the use of Slack:* One of the main contributions from Rabbit [1] is that the protocols do not require slack (unused domain space required to achieve statistical security). This implies, for instance, that private comparisons of 64 bits integers require a 64 bits domain. However, inequality tests (INQ) are common tools for applications that extensively use fixed point arithmetic. The current state of the art on the topic [3] still needs extended domains. More

---

[3] Following MPC analysis common practices, we evaluate protocols using round complexity as metric

precisely, it requires the presence of slack during truncation (the precision adjustment after multiplication). Additionally, albeit slack might not be present in the protocols for comparisons, it is still present in several protocols that are required by Rabbit, e.g. edaBits [8]. These factors still force application designers to use larger prime sizes. Hence, our work pursues to exploit the advantages of Rabbit whilst still bounded to the realities of state of the art and of practical deployments.

*On Round Evaluation:* Our novel combinations assimilate a selection of the most efficient elements from Makri et al. and Escudero et al. to reduce round complexity. We also incorporate Zaphod as means to achieve constant round complexity (via the multiparty garbled circuit evaluation of boolean components). For instance, we mix the protocol structure used by Catrina and De Hoogh [2], with the bitwise LTEQZ (Less Than Equal to Zero) test explored by Rabbit and make use the upgraded version of daBits from Zaphod, to sample randomness.

*Tool Selection:* We aim to produce comparison protocols compatible with commonly used results for fixed point arithmetic [3] and circuit evaluation, i.e. Zaphod. Both of which are only present in `SCALE-MAMBA`. To the best of our knowledge, `MP-SPDZ` currently does not implement conversion between their `BMR` and `LSSS` processors.

### 1.2 Our Contributions

- We present a series of constant round variations and adaptations of the privacy preserving comparison protocols, introduced by Makri et al. [1]. Our designs are focused on the implementation of ReLU's in the context of fixed point arithmetic. We introduce, implement and benchmark three different flavors using, in the two first instances, techniques that were succinctly mentioned by the authors, but not fully explored. Let $q$ be a sufficiently large prime e.g. 128 bits, then the variations are the following:
    - i). **Conventional Rabbit:** Our closest interpretation to traditional Rabbit. Here, bitwise sampling happens either from above i.e. $\lceil \log_2 q \rceil$ or below i.e. $\lfloor \log_2 q \rfloor$, as suggested by the original authors. It is assumed that $q$ is close to a power of 2. This makes Rabbit probabilistic in nature.
    - ii). **Rejection List Rabbit:** For any field $\mathbb{F}_q$, we sample randomness in $\mathbb{Z}_{2^{\lceil \log_2 q \rceil}}$, and reject samples that are above $q$.
    - iii). **Slack Rabbit:** A variation of the protocol where we relax security (becoming statistical instead of information theoretic, from an ideal perspective). We do this so that it can exploit the setup of protocols that require slack to instantiate fixed point representation.
- Much in the style of Escudero et al. [8], we adapt and optimize the commonly used comparison construction of Catrina et al. [2], more precisely:
    - i). We transform it into a constant round construction, thanks to Aly et al. Zaphod [6] and Hazay et al. [HSS17] [9];

   ii). we replace the fundamental bitwise-less-than (`LT`) construction, by the recently introduced bitwise, less than equal (`LTEQ`) protocol from [1] and;

   iii). we incorporate share conversion when needed.

- A thorough benchmark aimed for implementers. It includes the aforementioned `Rabbit` constructions and results, together with the optimized comparison protocol for ReLU's from [1]. We also consider different compilation styles[4] (VHDL to Bristol Fashion) of the same bitwise `LTEQ` circuit, including:

   i). `AND_XOR`$'s$**:** The circuit *as presented* in [1], 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.

- We give special emphasis to **Usability**. Hence, we provide open source `ReLU` and comparison libraries on publicly available repositories, written for `SCALE-MAMBA`.

### 1.3   Outline

This work is structured as follows: In Section 1, we present the main motivations for this work as well some basic comparison with the state of the art. Section 2, introduces some generalities needed for understanding our contributions. Sections 3 and 4 explore our novel `Rabbit` and non-`Rabbit` protocol combinations. Finally, we include an extensive benchmarking on Section 5 and some further discussion and our conclusions in Section 6. We have also included comprehensive Appendices, explaining in detail the background and inner workings of `Rabbit` and of our privacy preserving `ReLU` protocol.

## 2   Preliminaries

### 2.1   Notation

Let $q$ be a sufficiently large prime such that: *i).* It can instantiate the underlying MPC protocol; *ii).* Supports fixed point arithmetic with a public bitwise precision $p$ and; *iii).* Allows for a sufficiently large statistical security parameter sec.

   We make use of the same integer representation employed by the existing tooling, e.g. `SCALE-MAMBA` and `MP-SPDZ`. That is, we identify the finite field $F_q$ with the centered (integer) interval $[-\frac{q-1}{2}, \frac{q-1}{2}) \cap \mathbb{Z}$ and similarly, the ring $\mathbb{Z}_{2^k}$ with $[-2^{k-1}, 2^{k-1}) \cap \mathbb{Z}$, often assumed to be $k = 64$.

   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. $2^{64} - 1$. Because of the requirements of fixed point arithmetic, a typical mantissa of 40 bits would require

---

[4] Note that, as stated in `Zaphod`, `SCALE-MAMBA` adjusts the number of rounds (based on the circuit depth) when the setup is honest majority, instead of Full Threshold.

larger $q$'s, in this case, of at least 121 bits, as we see later on. Note that the sizes of the 2 domains are vastly different. We choose this homogenic representation, albeit it complicates the design of protocols of this kind, as this is more in line with the available tooling.

Additionally, we differentiate between secret shared fixed point elements ($\langle x \rangle$) and secret integers ($[\![y]\!]$), by the use square brackets. Integers can be secret shared in both $\mathbb{F}_q$ and $\mathbb{Z}_{2^k}$, whereas fixed points only in $\mathbb{F}_q$.

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. $[\![x]\!]_2$. More precisely, we adopt the fixed point representation introduced by Catrina and Saxena [3] used by SCALE-MAMBA and MP-SPDZ. Let $[\![v]\!]$ be a mantissa bounded by some $2^l$, and $p$ the bitwise fixed point precision. We can have a secret shared fixed point number $\langle x \rangle$ such that:

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

For simplicity and w.l.g. we also assume $l = 40$. This way multiplications can be supported whilst using typical prime size selections, e.g. a 128 bits prime. This is because the bitsize of the multiplication (before truncation) could double i.e. 80 bits. Furthermore, extra sec bits (typically sec = 40), need to be added as well. This comes from the probabilistic truncation protocol introduced in [3]. Hence, in regards to the field size, $q$ should be much larger than $2^{2 \cdot l + \texttt{sec}}$ for $\mathbb{F}_q$.

Finally, following convention, constants and vectors are uppercased. For completness, in this work we differentiate our protocols security parameter $\kappa$ from sec. However, in practice they are the same.

## 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 [10] and it is flexible enough to allow extensions. This technique has been adopted by other works in the area e.g. [7, 11, 12]. Furthermore, it has been employed by works of similar nature, with an emphasis on implementation e.g. [13]. In our case, the extensions correspond to (ideally extendable): fixed point arithmetic functionality and, a set of complex building blocks. We also specify the UC secure [14] realizations we based this functionality from. We define our arithmetic black box or $\mathcal{F}_{ABB}$, as specified in Table 1.

*Invoking* Rabbit ($\texttt{rabbit\_sint}([\![y]\!])$): For simplicity, we provide an encapsulation of the invocation of Rabbit. The method is just an abstraction tool, for the invocation of any of the variations of Rabbit presented in this work. The method simply extracts the system parameters and then invokes any desired Rabbit construction. Let $\texttt{sint}$ be a register type for elements in $\mathbb{F}_q$, then the construction can be simply expressed as indicated by Figure 1.

| Functionality | Description | Rounds | Prot. |
|---|---|---|---|
| $x \leftarrow [\![x]\!]$ | Opening secret field element | 1 | - |
| $[\![x]\!] \leftarrow x$ | Storing public input in a secret field element | 1 | - |
| $[\![z]\!] \leftarrow [\![x]\!] + [\![y]\!]$ | Addition: of secret inputs | 0 | - |
| $[\![z]\!] \leftarrow [\![x]\!] + y$ | Addition: (mixed) secret and public inputs | 0 | - |
| $[\![z]\!] \leftarrow [\![x]\!] \cdot [\![y]\!]$ | Multiplication: of secret inputs | 1 | - |
| $[\![z]\!] \leftarrow [\![x]\!] \cdot y$ | 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 | - |
| $\langle z \rangle \leftarrow [\![x]\!] + \langle y \rangle$ | Addition: secret fixed point with secret input | 0 | - |
| $\langle z \rangle \leftarrow \langle x \rangle \cdot \langle y \rangle$ | Multiplication: secret fixed point | 2 | [2, 3] |
| $\langle z \rangle \leftarrow \langle x \rangle \cdot y$ | Multiplication: (mixed) secret and public fixed point | 1 | [2, 3] |
| $\langle z \rangle \leftarrow [\![x]\!] \cdot \langle y \rangle$ | Multiplication: secret fixed point with secret input | 2 | [2, 3] |
| —**Complex Building Blocks**— | | | |
| $[\![\hat{\mathbf{r}}]\!]_q, [\![\hat{\mathbf{R}}]\!]_q, [\![\hat{\mathbf{r}}]\!]_{2^k}, [\![\hat{\mathbf{R}}]\!]_2 \leftarrow$ `get_r_from_dabits_list(size)` | Returns daBits list of size `size`, and their respective combination. | 0 | trivial |
| $[\![\hat{\mathbf{R}}]\!]_q, [\![\hat{\mathbf{R}}]\!]_2 \leftarrow$ `get_dabits_list(size)` | Returns vector of daBits of size `size`. | 0 | trivial |
| $[\![\hat{\mathbf{r}}]\!]_q, [\![\hat{\mathbf{r}}]\!]_{2^k} \leftarrow$ `combine_dabits(`$[\![\hat{\mathbf{R}}]\!]_q, [\![\hat{\mathbf{R}}]\!]_2$`)` | Returns vector combinations of $[\![\hat{\mathbf{R}}]\!]_q$, $[\![\hat{\mathbf{R}}]\!]_2$, daBits. | 0 | trivial |
| $[\![r_\kappa]\!] \leftarrow$ `PRandInt(size)` | Returns some randomness in $[0, 2^{\texttt{size}})$. | 0 | [2, 3] |
| $[\![x]\!]_q \leftarrow$ `conv_sint(`$[\![y]\!]_2$`)` | Conversion from $\mathbb{Z}_2$ to $\mathbb{F}_q$. | 1 | [4, 6, 15] |
| $[\![c]\!]_q \leftarrow$ `rabbit_sint(`$[\![y]\!]$`)` | Rabbit encapsulation routine. | 4 | This Work |
| $[\![c]\!]_q \leftarrow$ `bitwise_lteq(`$[\![y]\!]_q$`)` | bitwise LTEQZ circuit evaluation. | 2 | [1] |

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

**Arithmetic Black Box in Practice.** The abstraction, albeit useful from a theoretical perspective, meets some limitations in practice. In reality, an implementer has to adjust to what is provided by the tooling, e.g. `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 (**RISC** and **CISC**), to interact with registers. In the most ambitious setting, our constructions require constant round multiparty evaluation of Garbled Circuits (hence, Boolean Circuits), mixing them with arithmetic operations over Full Threshold, using High Gear Overdrive [16] (a member of the `SPDZ`'s family of protocols). `Zaphod` [6] achieves this, allowing us to combine a variation of [9, 17], for modulo 2 arithmetic, and [16] on modulo $q$ via daBits. As mentioned, `SCALE-MAMBA`, is the only framework currently available, implementing `Zaphod`, hence the one used by this work.

*On the Conversion* $\mathbb{Z}_2 \leftarrow \mathbb{F}_q$ (`conv_sint(`$[\![y]\!]_2$`)`): Just in the same way as in Rabbit [1], 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 [15]. 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

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

Fig. 1: Rabbit Encapuslation

that this time with the daBits modulo $q$. The process requires a single round. We refer the reader to [4], for a more detailed description.

### 2.3   ReLU

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

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

From this definition, 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 by 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's must be compatible with the MPC protocols supporting the operations for the other layers. This implies slack and fixed point arithmetic [3].

## 3   Ad(a/o)pting Rabbit

We explore our novel protocol combinations and discuss various aspects, necessary for their adoption in practice. For a complete revision on the definitions of Rabbit that are used in this section, we kindly refer the reader to Appendix Section 1.

### 3.1   Inequality Tests

Our main interest is the implementation of Less than Equal Zero (LTEQZ) tests. By using LTEQZ we can trivially instantiate any other form of INQ test. We can define the problem as follows:

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

### 3.2   Beyond **Rabbit**

Let us consider one of the main protocols introduced by [1], an INQ between a secret shared value and some public scalar. In this section, We discuss 3 flavors of such protocol, including one novel slack based version, later in this section. Note that the protocols in this section work directly on the mantissa of any fixed point value $\langle z \rangle$.

**Conventional Rabbit** Following the discussion from Marki et al. [1]: Consider the sampling distribution of the random masks the protocol uses. It is clear, that the vanilla Rabbit construction, as presented by the authors cannot achieve perfect security[5], when implemented over $\mathbb{F}_q$. To address this, authors succinctly mentioned possible alternatives, namely: *i).* selecting a prime close to a power of 2 and, *ii).* the use of a rejection list for sampling. We therefore provide our own constant-round understanding of these 2 scenarios.

***Probabilistic (Statistical Security):*** Protocol 1 proposes a version of Rabbit, in line (as is) with the original contributions outlined by [1]. This is the simplest of the constructions shown by this work. It is our interpretation of the author's original intent. We do include some minor adaptations, namely the use of constant round building blocks and the domain shifting, in order to make it compatible with fixed point arithmetic.
Notice that the selection of the prime field $\mathbb{F}_q$ is of importance. 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  [1], the gap defined as $|q - \lceil \log_2(q) \rceil|$ affects its security. We take this into account in our implementation and benchmarking. Furthermore, depending on 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. Our design addresses this latter concern as well.

*Correctness:* The protocol is correct, as demonstrated by [1]. We revise these principles on Appendix Section 1.

*Complexity:* Complexity is constant, with 2 parallel invocations of `bitwise_lteq` (2 rounds). We also have to take into account the opening in *line 9*. Note that the result given is modulo 2, hence an extra round for conversion is required for a total of 4 rounds. This last step is critical because we have to operate on fixed point arithmetic.

*Discussion:* As noted in previous sections, we deal with the domain shift, by adding `R` in places such as *line 13*. Besides this, and for implementation purposes, we have to trivially adapt the protocol so that it can discard/reduce a bit from

---

[5] Furthermore, current implementations of protocols for share conversion can only achieve statistical security.

---

**Protocol 1: Simple Rabbit (rabbit_fp)**

**Input:** The prime: ($q$), the prime flag: (above), and
secret input: ($[\![x]\!]_q$).
**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  $\texttt{R} = \lceil (q-1)/2 \rceil$
3  **if** above $== 1$ **then**
4   |  $[\![\hat{r}]\!]_q, [\![\hat{R}]\!]_q, [\![\hat{r}]\!]_{2^k}, [\![\hat{R}]\!]_2 \leftarrow \texttt{get\_r\_from\_dabits\_list}(\texttt{k});$
5  **else**
6   |  $[\![\hat{R}]\!]_q, [\![\hat{R}]\!]_2 \leftarrow \texttt{get\_dabits\_list}(\texttt{k}-1);$
7   |  $[\![\hat{r}]\!]_q, [\![\hat{r}]\!]_{2^k} \leftarrow \texttt{combine\_dabits}([\![\hat{R}]\!]_q, [\![\hat{R}]\!]_2);$
   **Pos:**
8  $[\![a]\!] \leftarrow ([\![\hat{r}]\!]_q + [\![x]\!]_q);$
9  $a \leftarrow [\![a]\!];$
10 $b = (a + q - \texttt{R});$
11 $[\![w_1]\!]_2 \leftarrow \texttt{bitwise\_lteq}\ (a, [\![\hat{R}]\!]_2, k);$
12 $[\![w_2]\!]_2 \leftarrow \texttt{bitwise\_lteq}\ (b, [\![\hat{R}]\!]_2, k);$
13 $w_3 = b + \texttt{R} < \texttt{cint}(q - \texttt{R}) + \texttt{R};$ //cint converts to register
14 **return** $[\![w_1]\!]_2 - [\![w_2]\!]_2 + w_3;$
15 in return $\texttt{conv\_sint}$ converts to $\mathbb{F}_q$

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

---

the randomness sampling. We also include some specificities from `SCALE-MAMBA`, e.g. (cint invocation to transform input to a `SCALE-MAMBA` register). *Security* follows directly from [1], i.e. simulation remains the same given that we do not perform any additional opening and the $\mathcal{F}_{ABB}$ functionality is ideally modeled. 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 [14]). This is also true for all constructions present in this work, hence we do not come back to it later. For specificities regarding the statistical properties of the security proof, we refer the reader to the original work in [1].

**Rejection List:** We extend the previous construction by modifying the sampling, and incorporating a rejection list. We show how to perform this inclusion in a way that, from an ideal perspective, gives this construction perfect security. We present our interpretation of Rabbit rejection list in Protocol 2.

*Correctness:* The correctness, once again follows from [1]. 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 4 original invocations (one opening and 2 rounds from the parallel calls to `bitwise_lteq` and 1 for the conversion). However, because of the rejection list (that involves additional calls to `bitwise_lteq`), performance depends on the prime, where probability ($\phi$) of every subsequent 2 rounds can be defined as:

$$\phi = \frac{2^{\lceil \log_2(q) \rceil} - q}{2^{\lceil \log_2(q) \rceil}}. \tag{4}$$

---

**Protocol 2: Rejection List Rabbit (`req_list`)**

**Input:** The prime: ($q$)
and secret input ($[\![x]\!]_q$).
**Output:** secret shared LTEQZ (x) in $\mathbb{F}_2$
**Pre:**
1  $k = \text{int}(\lfloor \log_2(q) \rfloor) + 1$; // int converts to integer
2  $\text{R} = \lceil (q-1)/2 \rceil$;
3  **repeat**
4  $\quad [\![\hat{\text{R}}]\!]_q, [\![\hat{\text{R}}]\!]_2 \leftarrow \text{fill\_dabits\_array}(k)$;
5  $\quad [\![\text{above}]\!]_2 \leftarrow \text{bitwise\_lteq } (q, [\![\hat{\text{R}}]\!]_2, k)$;
6  $\quad \text{above} \leftarrow [\![\text{above}]\!]_2$;
7  **until** (above == 1);
8  $[\![\hat{\text{r}}]\!]_q, [\![\hat{\text{r}}]\!]_{2^k} \leftarrow \text{combine\_dabits}([\![\hat{\text{R}}]\!]_q, [\![\hat{\text{R}}]\!]_2)$;
   **Pos:**
9  $[\![a]\!] \leftarrow ([\![\hat{\text{r}}]\!]_q + [\![x]\!]_q)$;
10  $a \leftarrow [\![a]\!]$;
11  $b = (a + q - \text{R})$;
12  $[\![w_1]\!]_2 \leftarrow \text{bitwise\_lteq } (a, [\![\hat{\text{R}}]\!]_2, k)$;
13  $[\![w_2]\!]_2 \leftarrow \text{bitwise\_lteq } (b, [\![\hat{\text{R}}]\!]_2, k)$;
14  $w_3 = b + \text{R} < \text{cint}(q - \text{R}) + \text{R}$; //cint converts to register
15  **return** $[\![w_1]\!]_2 - [\![w_2]\!]_2 + [\![w_3]\!]$;
16  //in return `conv_sint` converts to $\mathbb{F}_q$

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

---

*Discussion:* As before, the prime selection is also important although for different reasons i.e. the prime should not be far below the immediate next power of 2, as the sampling, will have a lower probability of requiring extra rounds per execution. *Security*, as indicated before, trivially follows from [1]. Moreover, as it is shown in the original work, the incorporation of the rejection list, is the only method available by which any Rabbit variation can (ideally) achieve prefect security over $\mathbb{F}_q$.

**Slack Rabbit** We also propose novel a variation of Rabbit that reduces the sizes of the bitwise inequality tests `bitwise_lteq`. This significantly reduces the

number of daBits the protocol requires whilst still working on larger $q$'s. We achieve this by borrowing and adapting the bitwise sampling of randomness from Catrina and De Hoogh. 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: $(\llbracket x \rrbracket_q)$.
> **Output:** secret shared LTEQZ (x) in $\mathbb{F}_2$
> **Pre:**
> 1 $M = 2^k$;
> 2 $R = \lceil \frac{M_2}{2} \rceil - 1$;
> 3 $\llbracket r_\kappa \rrbracket \leftarrow$ PRandInt$(\kappa)$;
> 4 $\llbracket \hat{r} \rrbracket_q, \llbracket \hat{R} \rrbracket_q, \llbracket \hat{r} \rrbracket_{2^k}, \llbracket \hat{R} \rrbracket_2 \leftarrow$ get_r_from_dabits_list$(k)$;
>     **Pos:**
> 5 $\llbracket z \rrbracket \leftarrow \llbracket x \rrbracket_q + R$;
> 6 $\llbracket a \rrbracket \leftarrow (\llbracket \hat{r} \rrbracket_q + \llbracket z \rrbracket) + \llbracket r_\kappa \rrbracket \cdot 2^k$;
> 7 $a \leftarrow \llbracket a \rrbracket$;
> 8 $b \leftarrow a + M - R$;
> 9 $a' = a \bmod 2^k$;
> 10 $b' = b \bmod 2^k$;
> 11 //comparisons
> 12 $\llbracket w_1 \rrbracket_2 \leftarrow$ bitwise_lteq $(a', \llbracket \hat{R} \rrbracket_2, k)$;
> 13 $\llbracket w_2 \rrbracket_2 \leftarrow$ bitwise_lteq $(b', \llbracket \hat{R} \rrbracket_2, k)$;
> 14 $w_3 = b - \text{cint}(M + R) < 0$; //cint converts to register
> 15 **return** $1 - (\llbracket w_1 \rrbracket_2 - \llbracket w_2 \rrbracket_2 + \llbracket w_3 \rrbracket)$;

PROTOCOL 3: Rabbit on statistical security.

---

*Correctness:* The protocol uses the same principles from Rabbit, from previous protocols (see Appendix Section 1). Note that changes were applied to the generation of the random mask. As indicated in [18] and [2], by using black box random bit generation (via daBits), combined with PRandInt, we can build a statistically hiding mask. At the same time, we also hold the bit decomposition of the relevant sections of the randomness.

*Complexity:* Complexity remains the same, i.e. constant round $\mathcal{O}(1)$ and 4 rounds in total. 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.

*Discussion:* The protocol begins by reconstructing some bounded randomness. It combines $k$ daBits (that are later used as the bit expansion of $r \leftarrow \mathbb{Z}_{2^k}$, i.e. 64 bits) and PRandInt (on the size of our security parameter $\kappa$). After some domain switching (see Section 2.1), which is further complicated by the types limitations in SCALE-MAMBA (negative number encoding), the protocol proceeds to the familiar flow of Rabbit. A relevant observation, is that the bitwise comparisons can be done over a smaller domain e.g. $2^k$, rather than in $\log_2(q)$. A fact that we exploit on *lines 12) and 13)*. *Security* analysis is also trivial, as it follows directly from the discussion on [2,18]. This is also the case for the statistical properties of the mask.

### 3.3   Optimized Rabbit for ReLU *'s*

For completeness we also explore the optimized comparison protocol designed specifically for ReLU's from Makri et al. [1]. Authors imagine scenarios where w.l.o.g. the 64 bits domain coincides with the ring size. The protocol itself is quite different to the constructions we present in this paper. In fairly general terms 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.

The protocol proposes challenges when used in the context of the applications explored by this work. More precisely, to make it compatible with $\mathbb{F}_q$, and with the results from Catrina and Saxena [3] (fixed point arithmetic), we have to rely on the rather expensive $\mathbb{F}_q$ to $\mathbb{Z}_{2^{64}}$ conversion primitive from Zaphod[6]. We have implemented this variation as well for benchmarking purposes. We call this construction `rabbit_conv`.

## 4   Constant Round Catrina and De Hoogh

In this section, we explore our constant round adaptation of the commonly used LTZ formulation from Catrina and De Hoogh [2], combined with some of the enhancements introduced by Rabbit. More formally, we present 2 protocols to: Solve the modulo $2^m$ of any secret shared input and then, an oblivious LTZ construction based on the former.

---

[6] The details of the conversion and its development are also explored in detail in SCALE-MAMBA.

### 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 [2]. However, we introduced some notable changes:

- **Randomness**: We sample the random mask using daBits in the first $k$ bits, instead of the construction from [7]. A notable difference with other adaptation efforts i.e. [8] is that we can make use of PRandInt($k'$) to generate the slack.
- **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 to further adapt the protocol so that it can still mimic the original result.
- **Conversion to** $\mathbb{F}_q$: The result of the bitwise comparison is obtained via distributed Garbled Circuits [HSS17], with an output on $\mathbb{Z}_2$. Hence, we invoke conv_sint($[\![x]\!]$) to transform the output back to $\mathbb{F}_q$.

---

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

> **Input:**  the bit length 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  $\mathtt{k}' = k + \kappa - \mathtt{m}$;
> 4  $[\![r_\kappa]\!] \leftarrow \mathtt{PRandInt}(\mathtt{k}')$;
> 5  $[\![\hat{\mathtt{r}}]\!]_q, [\![\hat{\mathtt{R}}]\!]_q, [\![\hat{\mathtt{r}}]\!]_{2^k}, [\![\hat{\mathtt{R}}]\!]_2 \leftarrow \mathtt{get\_r\_from\_dabits\_list}(\mathtt{m})$;
> 6  $[\![\hat{\mathtt{r}}]\!]_q = [\![\hat{\mathtt{r}}]\!]_q + 1$;
> **Pos:**
> 7  $[\![a]\!] \leftarrow 2^{k-1} + [\![\hat{\mathtt{r}}]\!]_q + [\![x]\!]_q + [\![r_\kappa]\!] \cdot 2^{\mathtt{m}}$;
> 8  $a \leftarrow [\![a]\!]$;
> 9  $a' = a \bmod 2^m$;
> 10  $[\![w]\!]_2 \leftarrow \mathtt{bitwise\_lteq}\,(a', [\![\hat{\mathtt{R}}]\!]_2, k)$;
> 11  $[\![w]\!]_q \leftarrow \mathtt{conv\_sint}([\![w]\!]_2)$ ;
> 12  **return** $a' - [\![\hat{\mathtt{r}}]\!]_q + (2^m) \cdot [\![w]\!]_q$;

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

---

*Correctness:* Correctness follows directly from Catrina and De Hoogh [2]. Note the change of the bitwise INQ test, from an LTZ to LTEQZ (on *line 6*). To obtain the equivalent behaviour, we add a 1 to the value that is secretly contained in

$[\![\hat{\mathbf{r}}]\!]_q$. This small change propagates to $a$. As a result, the composition of $a'$ is altered i.e. $[\![\hat{\mathbf{r}}]\!]_q + 1$ instead of $[\![\hat{\mathbf{r}}]\!]_q$. This suffices for $a' \leq [\![\hat{\mathbf{r}}]\!]_q$ to match the expected behaviour. Besides this, all other changes introduced to the protocol are orthogonal to its correctness.

*Complexity:* The protocol has a constant round complexity, similar to other protocols studied in this paper (4 rounds). An opening, 1 invocation of a Garbled circuit (2 rounds), and 1 extra round for the conversion in *line 11.* Hence, results are delivered in $\mathbb{F}_q$.

*Discussion:* Protocol keeps the basic and simple flow, that made it so popular to use with fixed point values. *Security* and the statistical properties of the mask trivially follow from [2]. Simulation is the same, given that there are no additional openings, and the generation of the random mask and conversion are modeled as ideal functionalities. As a technical observation, we could envision a circuit that merges *line 10*, and *line 11*, and thus further reducing the round count by 1. Note that, in practice this means that the typical 40 bits of security present in frameworks such as SCALE-MAMBA can remain unchanged.

### 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 synthesize their results and produce a single protocol, replacing the modulo $2^m$ invocation by our own mod2m. Traditionally, the Catrina and de Hoogh construction returns LTZ. We note that it can be trivially adapted to return a LTEQZ instead (by 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 $\mathbb{F}_q$
1   $[\![x']\!] \leftarrow$ mod2m $(k, k - 1, \kappa, [\![x]\!])$;
2   $[\![z]\!] = \frac{[\![x]\!] - [\![x']\!]}{2^m}$ mod $q$;
3   **return** $-[\![z]\!]$;

PROTOCOL 5:  Constant round Catrina and Hoogh.

---

*Correctness:* It follows directly from [2]. We have replaced some elements in the mod2m construction, namely the bitwise INQ test. However, as previously stated the behaviour does mimic the original. This protocol can be seen as 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 feature any additional operation that requires round based computations i.e. 4 rounds. Its complexity matches all `Rabbit` constructions introduced by this work. However, we reduce the number of invocations of `bitwise_lteq` to 1.

*Discussion:* The protocol is intuitive and simple. It can also be trivially adapted to derive a probabilistic truncation protocol. *Security* follows directly from `mod2m`, as there are no other openings originated by the protocol. As a reminder, simulation in this case is trivial (hybrid model [14]), given that, there is no opening and we only compose it with ideal functionalities.

## 5   Benchmarking

In this section, we provide a performance evaluation geared towards implementers of all our protocols. This section also provides details regarding our testing environment and our prototype.

### 5.1   Prototype

Our implementation was built on top of `SCALE-MAMBA` version 1.13 and it is compatible with version 1.14. This open source framework is commonly used for experimentation by various applications, including on topics related to machine learning [13, 19]. On setups, the framework offers several protocol flavours that can be combined with Boolean circuit evaluation thanks to `Zaphod`. We are particularly interested in two: *i).* High Gear Overdrive, a member of the `SPDZ`'s family [16], what we call from this point forward Full Threshold configuration (FT) and *ii).* A custom made `SCALE-MAMBA` adaptation to evaluate the case when Garbling is done offline, as originally intended in `HSS17` [9], referred to in this section as FT(OG). For completeness, we have included the Shamir based protocol from Smart and Wood [20], simply referred to as Shamir. Note that we can only use Garbled Circuits via `HSS17` in combination with FT setups. It is worth commenting that there are no high level functionalities available in `MAMBA` related to `daBits` (except for the instruction itself). Hence, our prototype required to build all the `daBits` apparatus described by our $\mathcal{F}_{ABB}$. More formally, our prototype comprises of 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 VHDL circuit versions of `bitwise_lteq` (Bristol Fashion). We add few minor manual edits to conform to circuit design. They were generated via the limited tooling already available in `SCALE-MAMBA`. Both target `rabbit_slack` and `cons_ltz` (our fastest protocols), note that they make use the `GC` instruction in 64 bits. More precisely:

    i). `AND_XOR`: Circuit from [1] as is. Convenient for `[HSS17]` [9][7].

    ii). `AND_NOT`: With a simple depth reduction. Convenient for `LSSS` [20][7].

- Library `rabbit_lib.py`. Implements all `INQ` tests described in this paper.
- Library `relu_lib.py`. Implements variations of `ReLU`'s, specifically for Machine Learning, including 2 and 3 dimensional ReLU's.
- Extensive test programs for all the functionality presented/required by this work.

For completeness, we have also incorporated in our benchmarking the original `LTZ` from Catrina and De Hoogh [2], which is already present in `SCALE-MAMBA`. This protocol works exclusively over `LSSS` in $\mathbb{F}_q$. In our tests, we refer to this protocol as `lsss_ltz`. Finally, we note the prototype is fully available as open source [8].

*Performance Evaluation Setup:* Our test bed comprises of 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 is 0.15 ms. This allows us to control network conditions by tuning latency via `/sbin/tc`. In terms of the `SCALE-MAMBA` configuration (default), we use a prime $q$ of 128 bits, with a matching `sec` and $\kappa = 40$, and a $k = 64$, for all our protocols. The size of the mantissa for fixed point is 40. It is also worth commenting that all times (expressed in ms) in our performance evaluation correspond exclusively to the online phase.

It is important to note that the setup considers the smallest party subsets supported by Smart and Wood [20] and Hazay et al. [9] i.e. 3 and 2 parties, respectively. Any setup with additional parties can then be extrapolated.

### 5.2 Results

Table 2 shows the performance evaluation in one machine, with no real communications. Time complexity is entirely dominated by computation costs. As expected, Shamir outperforms FT for all protocols. 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 the same in both configurations. This is because it does not require Garbled Circuits. It is worth noting that Rabbit rejection list `req_list`, Rabbit with conversion `rabbit_conv` and simple Rabbit `rabbit_fp` protocols significantly underperform the rest. 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 [4]). On the other hand, for FT there is no meaningful difference between

---

[7]  See `SCALE-MAMBA` documentation.

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

Table 2: Performance evaluation (ms) in one machine

| Protocol | Circuit | No Comms | | |
|---|---|---|---|---|
| | | Shamir | FT | FT(OG) |
| `rabbit_slack` | `AND_NOT` | 2.3 | 8.5 | 1.4 |
| `cons_ltz` | `AND_NOT` | 1.3 | 4.2 | 1.2 |
| `rabbit_slack` | `AND_XOR` | 3.1 | 8.1 | 1.3 |
| `cons_ltz` | `AND_XOR` | 1.5 | 4.0 | 1.0 |
| `rabbit_conv` | - | 8.4 | 12.7 | - |
| `req_list` | - | 44.0 | 119.7 | - |
| `rabbit_fp` | - | 30.7 | 64.0 | - |
| `lsss_ltz` | - | 1.7 | 1.8 | - |

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

| Protocol | Circuit | D=10ms | | | D=20ms | | |
|---|---|---|---|---|---|---|---|
| | | Shamir | FT | FT(OG) | Shamir | FT | FT(OG) |
| `rabbit_slack` | `AND_NOT` | 623.1 | 297.6 | 124.7 | 1236.2 | 577.2 | 245.1 |
| `cons_ltz` | `AND_NOT` | 317.7 | 148.6 | 62.6 | 627.7 | 289.1 | 122.7 |
| `rabbit_slack` | `AND_XOR` | 1031.5 | 295.7 | 124.7 | 2045.2 | 575.2 | 244.6 |
| `cons_ltz` | `AND_XOR` | 522.0 | 148.6 | 62.5 | 1033.0 | 288.1 | 122.5 |
| `lsss_ltz` | - | 74.9 | 75.2 | - | 144.9 | 144.6 | - |

both circuits when applied in both `rabbit_slack` and `cons_ltz`, given their small size($\approx$ less than 200 gates). 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 for garbling in `cons_ltz`. This is clear when observing the results for the case when the garbling is done offline, i.e. FT(OG). In such case, the proposed `cons_ltz` is noticeably more efficient than `lsss_ltz` (and slightly more efficient than `rabbit_slack`).

Table 3 shows the results when the protocols run on several machines with real communications. Specifically, with two different latencies 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, with only 10ms delay, FT configuration becomes faster than Shamir. This is explained by the fact that Shamir incurs in more communication rounds per bitwise circuit evaluation.

When considering the case for garbling offline our proposed `cons_ltz` outperforms all other protocols, including `lsss_ltz`. This result is consistent on both latencies 10ms and 20ms (Table 3).

In the case of the FT configuration, `lsss_ltz` is faster than its counterparts. Increasing the delay does not favour `cons_ltz` over `lsss_ltz`. This may be counter-intuitive, since `lsss_ltz` has 7 rounds and `cons_ltz` has 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 lesser rounds. This is not necessarily the case when the communication channel implements TCP e.g. TLS, because TCP throughput is heavily affected by latency and packet loss rate. This is due to TCP's congestion control mechanisms and its slow start procedure. To solve this, framework designers could apply secure communications protocols based on UDP, such as QUIC, since UDP throughput is unaffected by latency. Although such profound change falls out of the scope of this work, we believe it is an interesting unexplored research avenue. Hence, we strongly encourage framework developers, namely for SCALE-MAMBA and MP-SPDZ, to incorporate it in future releases.

### 5.3   Practical Scenarios

Realistically, frameworks such as SCALE-MAMBA rely on circuit optimizers to reduce the number of communication rounds during compilation time. Without them, round complexity for the Catrina and de Hoogh (lsss_ltz) protocol increases drastically. Namely, the construction requires as many rounds as multiplications which, for a 128 bit prime implies 121 rounds instead of 7. It has to be stressed that because of their nature, optimizers require to load the dependency graph to memory, becoming impractical for large circuits e.g. neural networks (see [4]). In such scenarios, cons_ltz (with AND_NOT) becomes the far better alternative in all configurations. We tested this case by compiling the comparisons without such optimizations[9]. We show these results in Table 4. While cons_ltz performance remains unaffected by this new configuration, lsss_ltz's performance is significantly impaired.

Table 4: Performance evaluation (ms) with 2/3 machines (FT / Shamir)

| Protocol | Circuit | No Comms | | D=10ms | | D=20ms | |
|---|---|---|---|---|---|---|---|
| | | Shamir | FT | Shamir | FT | Shamir | FT |
| cons_ltz-O1 | AND_NOT | 1.3 | 4.6 | 317.8 | 148.6 | 627.7 | 289.2 |
| cons_ltz-O1 | AND_XOR | 1.6 | 3.9 | 522.1 | 148.6 | 1033.0 | 288.2 |
| lsss_ltz-O1 | - | 4.1 | 4.0 | 1243.8 | 1238.8 | 2453.8 | 2449.2 |

## 6   Conclusion, Discussion and Future Work

We presented a series of variations and novel combinations of Rabbit and other comparison protocols for MPC, with ReLU's in mind. Our results are compatible

---

[9] We achieve this via the -O1 compilation flag, as recommended by the SCALE-MAMBA documentation [4]

with the current state of the art for fixed point arithmetic [3] and constant round evaluation of Boolean circuits i.e. Zaphod [6]. We accompany our findings with thorough benchmarking and provide an open source implementation. Our results show that:

i). The conventional Rabbit variations studied in this work underperform our novel slack based formulation for large primes.
ii). As latency increases, all constructions perform worse than the original Catrina and De Hoogh `lsss_ltz` when Garbling is done Online.
iii). Our `cons_ltz` performs better than classical LSSS Catrina and De Hoogh and any other protocol when Garbling is performed Offline. This is true for all latency configurations.
iv). When compiling without optimizations, `cons_ltz` and `rabbit_slack` are a far better choice (by orders of magnitude) than classic Catrina and De Hoogh `lsss_ltz` under any setup i.e. Shamir, FT and FT(OG). This is specially relevant for Machine Learning, due to the severe compiler limitations of `SCALE-MAMBA` for large circuits.

Regarding future work, we believe that the use of QUIC can be an interesting avenue of research for framework developers. Notice that this work also raises novel questions regarding the process of compiling Boolean circuits i.e. Circuit optimization via hardware oriented synthesizers.

## Acknowledgements

## References

1. 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
2. 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
3. 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
4. Aly, A., Cong, K., Keller, M., Orsini, E., Rotaru, D., Scherer, O., Scholl, P., Smart, N.P., Tanguy, T., Wood, T.: SCALE and MAMBA v1.14: Documentation (2021) https://homes.esat.kuleuven.be/~nsmart/SCALE/.
5. 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

6. 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
7. 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
8. 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
9. 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
10. 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
11. 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
12. 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
13. 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
14. Canetti, R.: Security and composition of multiparty cryptographic protocols. Journal of Cryptology **13**(1) (January 2000) 143–202
15. 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
16. 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
17. 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
18. 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
19. 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
20. 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
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

# Appendix

## 1 Rabbit Definitions

### 1.1 The problem

The majority of previous works have proposed mechanisms for comparisons based on bit decomposition. Such constructions tend to be expensive in terms of multiplicative depth. In general terms, they are compatible with both MPC and Homomorphic Encryption (HE) schemes. They also tend to have sublinear cost on the inputs size with some associated non-negligible constant. An example of this, is the seminal work from Damgård et al. [7] previously mentioned by this work.

To solve this issue, the literature has proposed to relax the security model, as a trade-off for efficiency e.g. [2, 18]. 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.

### 1.2 The Rabbit Principle

The method itself is based on the commutative properties of addition. It formulates 2 different but equivalent equations. Generally speaking, the authors derive a simplified algebraic construction comprised of 3 INQ's, one of which relies exclusively on public inputs. The other 2 depend on public inputs and secret shared precomputed randomness. Furthermore, they can be all executed in parallel. This construction is equivalent to an LTEQZ on secret inputs.

To generate randomness, Rabbit authors rely on edaBits [8] (which is exclusively present on MP-SPDZ). For the purpose of this section, we abstract edaBits, as a construction that allows us to generate (together with its bit expansion) some bounded randomness in $\mathbb{Z}_{2^k}$.

Let $\mathbb{Z}_M$ be some commutative ring bounded by $M \in \mathbb{Z}$, $[\![x]\!]$ and $[\![r]\!]$ be 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:

$$B = M - R, \tag{5}$$

$$[\![a]\!] = [\![x]\!] + [\![r]\!], \tag{6}$$

$$[\![b]\!] = [\![x]\!] + [\![r]\!] + B, \tag{7}$$

$$[\![c]\!] = [\![x]\!] + B. \tag{8}$$

If we observe carefully the constructions above, we can appreciate that $B$ is simply the complement of $R$. Prior discussing the algebraic elements of Rabbit,

let us now consider the following statement:

$$[x + y] = \begin{cases} [\![x]\!] + [\![y]\!] - M \cdot \mathsf{LT}([\![x]\!] + [\![y]\!], [\![x]\!]); \\ [\![x]\!] + [\![y]\!] - M \cdot \mathsf{LT}([\![x]\!] + [\![y]\!], [\![y]\!]). \end{cases} \quad (9)$$

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 $([\![a]\!] + B)$:

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

When we expand $[\![c + r]\!]$ in the same fashion we can derive the following:

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

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

$$[\![a < r]\!] + [\![b < B]\!] = [\![c < B]\!] + [\![b < r]\!].$$

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 $\mathsf{INQ}$ 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 $\mathsf{INQ}$ would be true, only if $x$ is greater than $R$. Note that we are still working on $\mathbb{Z}_M$, meaning that any excess over $R$ would force an overflow and a subsequent wraparound. Now, let us abuse the notation and consider $R$ to be some public element on $\mathbb{Z}_M$ e.g. 0.

Given that we can freely disclose the 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. The inequality tests themselves are executed bitwise, using for instance, $\mathsf{edaBits}$ or any other mean to obtain the bit decomposition offline.

## 2   Constant Round **ReLU** Protocol

Our proposed $\mathsf{ReLU}$ construction, follows the same line of thought from the contribution sections of this work. Indeed, Catrina and Saxena's fixed point representation is heavily interlinked with the protocol. In fact, Protocol 6, optimizes

the fixed point multiplication needed by the ReLU, extracting the mantissa from $\langle x \rangle$.

In line with the definitions, introduced in Section 2.3. Our constant round ReLU can be trivially implemented as follows:

---

**Protocol 6: constant round ReLU ($\texttt{relu(x)}$)**

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

PROTOCOL 6:  Constant round protocol for ReLU's.

---

*Complexity:* The protocol has constant round complexity ($\mathcal{O}(1)$). It consists of 1 round (from the multiplication on *line 3*), plus what is added by any selected comparison mechanism introduced by this work i.e. 4 rounds.

*Discussion:* Our ReLU itself does not require to invoke $\texttt{PRTrunc}$, ever present in fixed point multiplications. Note that ReLU's are typically surrounded by more complex fixed point operations, that do require conventional fixed point multiplications e.g. [21] (hence the presence of slack and an invocation of $\texttt{PRTrunc}$ per multiplication gate).