ABLE: Optimizing Mixed Arithmetic and Boolean Garbled Circuit

Jianqiao Cambridge Mo
jm8782@nyu.edu
New York University
Tandon School of Engineering
Brooklyn, New York, USA

Brandon Reagen
bjr5@nyu.edu
New York University
Tandon School of Engineering
Brooklyn, New York, USA

ABSTRACT

Privacy and security have become critical priorities in many scenarios. Privacy-preserving computation (PPC) is a powerful solution that allows functions to be computed directly on encrypted data. Garbled circuit (GC) is a key PPC technology that enables secure, confidential computing. GC comes in two forms: Boolean GC supports all operations by expressing functions as logic circuits; arithmetic GC is a newer technique to efficiently compute a set of arithmetic operations like addition and multiplication. Mixed GC combines both Boolean and arithmetic GC, in an attempt to optimize performance by computing each function in its most natural form. However, this introduces additional costly conversions between the two forms. It remains unclear if and when the efficiency gains of arithmetic GC outweigh these conversion costs.

In this paper, we present Arithmetic Boolean Logic Exchange for Garbled Circuit, the first real implementation of mixed GC. ABLE profiles the performance of Boolean and arithmetic GC operations along with their conversions. We assess not only communication but also computation latency, a crucial factor in evaluating the end-to-end runtime of GC. Based on these insights, we propose a method to determine whether it is more efficient to use general Boolean GC or mixed GC for a given application. Rather than implementing both approaches to identify the better solution for each unique case, our method enables users to select the most suitable GC realization early in the design process. This method evaluates whether the benefits of transitioning operations from Boolean to arithmetic GC offset the associated conversion costs. We apply this approach to a neural network task as a case study.

Implementing ABLE reveals opportunities for further GC optimization. We propose a heuristic to reduce the number of primes in arithmetic GC, cutting communication by 14.1% and compute latency by 15.7% in a 16-bit system. Additionally, we optimize mixed GC conversions with row reduction technique, achieving a 48.6% reduction in garbled table size for bit-decomposition and a 50% reduction for bit-composition operation. These improvements reduce communication overhead in stream GC and free up storage in the GC with preprocessing approach. We open source our code for community use.

KEYWORDS

privacy-preserving computation, secure multi-party computation, cryptography

1 INTRODUCTION

With the rapid growth of cloud computing and data-driven services, the handling of sensitive data has become a critical concern. As more businesses and individuals rely on cloud-based platforms for processing and storing their private information, the need for robust data protection techniques has intensified. This demand has spurred the development of privacy-preserving computation (PPC), a set of technologies that ensure both confidentiality and control over sensitive data during computation. PPC, including secure multi-party computation (MPC), allows parties to perform computations on encrypted data without exposing their private inputs. This is particularly valuable in scenarios where data confidentiality is paramount, such as in financial transactions, healthcare analytics, and secure machine learning. PPC technologies not only protect data from third-party interception during transmission, but also keep it encrypted throughout the entire computation process, safeguarding privacy and security even from the service provider.

One of the most widely adopted PPC techniques is Yao's garbled circuit (GC) protocol [75, 76]. This protocol enables two parties, the garbler and the evaluator, to jointly evaluate a function while keeping their inputs private. Since its introduction, GC has become a fundamental tool in cryptography and has been applied to a variety of secure computation applications, including secure machine learning [52, 56, 71], private inference [28, 31, 39, 49], and blockchain technologies [32]. For instance, in a cloud-based machine machine learning service, GC can be used to ensure that a user's private data remains confidential during image inference. This allows the service provider to generate results without accessing the raw data. Similarly, in the context of Ethereum, GC can securely evaluate operations such as those containing logical conditions like IF, OR, or EQUALS, without revealing sensitive transaction data to the participants, thereby maintaining privacy while ensuring the integrity of operations like balance changes. The key advantage of GC compared to other PPC techniques like homomorphic encryption and additive secret sharing is that GC is general, enabling computation on any arbitrary function, and GC has fixed-round communication.

GC was initially implemented using Boolean logic. Much like traditional logic synthesis, a function is first translated into a combinational logic circuit before being securely computed. In this approach, both the binary bits of the function's inputs and the logic gates are processed in an encrypted form. At the protocol level, one party (the garbler) prepares the encrypted inputs and the garbled circuit, then sends them to the other party (the evaluator). The evaluator performs evaluation on the garbled circuit using the encrypted inputs to produce the results. Extensive research has focused on improving the efficiency of Boolean GC, particularly in reducing communication complexity and encryption overhead. This is crucial because the total cost of GC can become prohibitively high for circuits with large input bit lengths and complex operations. Numerous prior works [12, 41, 43, 58, 64, 67, 78] have been developed to optimize GC construction. For example, the Row Reduction [58] technique reduces the communication required for transmitting the

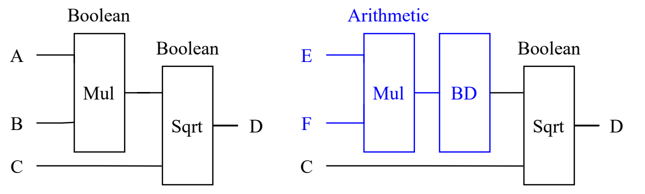

Figure 1: Different GC realizations for a function: traditional Boolean GC on the left, mixed GC on the right. Blue represents operands / operations in arithmetic. Since square root is not supported by arithmetic GC, a BD gate is required to convert it to Boolean.

garbled circuit, while FreeXOR [43] and HalfGate [78] techniques eliminate the communication cost for XOR gates and halve the cost for AND gates, respectively. Furthermore, low-cost and hardware accelerations [26, 34, 35, 54, 74] have made GC over 500× faster compared to software implementations on CPU.

Although improvements of Boolean GC exist, the total cost remains relatively high due to its tight binding with Boolean circuits. For instance, garbling a Boolean 32-bit multiplication circuit can require up to 2.8k gates. These gates cannot be reused due to security constraints, causing the total complexity of the circuit to grow linearly with an increasing number of operations. On the other hand, Applebaum et al. [4] introduced arithmetic GC, and Ball et al. [9] further provided an efficient approach to arithmetic GC. Unlike Boolean GC, where inputs are processed bit by bit and operations are performed at the gate level, arithmetic GC processes inputs as numbers over a ring $\mathcal{R}$, allowing for arithmetic operations on inputs $x \in \mathcal{R}$. The advantages of arithmetic GC are substantial. For example, multiplication, which requires many individual gates in Boolean GC, can be handled as a single, straightforward multiplication procedure in arithmetic GC. The construction developed in [7, 9] improves arithmetic GC with free addition / subtraction, free multiplication by a public constant, and low-cost general ciphertext multiplication and exponentiation by a public constant. Similar to FreeXOR, *free* here means that the operation requires no communication or encryption during evaluation.

Despite these advantages, many PPC frameworks [5, 22, 40, 48, 63, 72, 77] still rely on Boolean GC because it is more general with broader applicability, whereas arithmetic GC is restricted to specific arithmetic operations. Accurately computing arbitrary functions with only arithmetic GC – including addition, multiplication, and exponentiation – remains challenging, particularly for non-linear operations like comparison, conditional logic, and ReLU activation, which generally require "bit-wise" processes. Therefore, a standard approach is to combine both Boolean and arithmetic GC, leveraging the generality of Boolean GC while exploiting the efficiencies of arithmetic GC for specific tasks. This combination necessitates constructing conversion mechanisms to switch between arithmetic and Boolean GC.

Arithmetic GC encodes values over a large ring $\mathcal{R}$ using the Residue Number System (RNS) [62]. Naively garbling a conversion circuit for RNS arithmetic and Boolean GC – specifically, garbling a Chinese Remainder Theorem (CRT) function to recover values from RNS encoding – can lead to exponentially large and inefficient circuits. Recently, Li et al. [47] proposed an efficient method for managing these conversions. This approach includes efficient bit-composition (BC) and bit-decomposition (BD) gates that facilitate seamless conversion between Boolean and arithmetic within a circuit. It enables the construction of mixed GC, which integrates both Boolean and arithmetic operations and values within a single circuit.

For practical mixed GC, it is crucial that the benefits of using arithmetic GC for certain operations outweigh the costs of conversion. Figure 1 demonstrates a general function that includes both multiplication and square root operations. While arithmetic GC can efficiently handle multiplication, the square root operation has to be computed using Boolean GC. The challenge is, it will be unclear to GC users to determine whether the efficiency gains from arithmetic multiplication are sufficient to offset the additional costs of BD gates required for conversion. Although arithmetic multiplication is generally more efficient than its Boolean counterpart, if the cost of BD gates is significantly higher, the overall cost could exceed the original Boolean approach. Our goal is to eliminate the need to garble both Boolean and mixed approaches for comparison at each time, and ensure that users select the most suitable, cost-effective GC realization for their tasks.

Some PPC frameworks [22, 40, 68] combine Boolean and arithmetic computation with different protocols, such as Additive Secret Sharing or Homomorphic Encryption. However, switching between protocols introduces additional costs, including the need for extra authentication during transitions, increased communication rounds, or potential exposure to plaintext. These conversion techniques are generally interactive, whereas mixed GC remains entirely within the (non-interactive) GC paradigm. The intent of this paper is not to argue whether switching protocols is superior, but rather to demonstrate how to determine an efficient GC realization for a given workload, thereby enhancing the performance of applications involving the GC protocol.

1.1 Contributions

In this paper, we implement and optimize the mixed GC and its practical BC/BD methods. First, we improve prime selection for arithmetic GC. The RNS of arithmetic GC consists of a series of primorial modulus mod-p arithmetic circuits. By heuristically removing certain primes, we maintain a sufficiently large range for arithmetic numbers while reducing unnecessary primes, thus decreasing communication overhead and runtime. Next, we refine the order and key generation procedure using the row reduction technique in BD and BC to minimize ciphertext communication. These optimizations result in more efficient implementations of arithmetic GC and GC conversion. By evaluating the conversion cost in different GC conditions, including stream GC and GC with preprocessing, we provide a clear and intuitive method for users to determine the efficient GC realization for their application early in the design phase. Our work can be summarized as follows:

(1) Comprehensive characterization of mixed GC. We present the first end-to-end analysis of mixed GC, evaluating various operations with respect to their garbled table costs,

garbling and evaluation latency. This analysis spans different GC approaches, including stream and preprocessing GC, and different bit-widths.

(2) Decision tools for GC realization. We introduce the BC and BD equivalent ratios to help GC users determine the most performant GC configuration (either pure Boolean or mixed) for their specific applications at an early stage, before committing to the garbling process.
(3) Optimized prime selection for arithmetic GC. We propose a new method for selecting a minimal set of primes in arithmetic GC, reducing the overall prime sum by an average of 5.3%. We list the optimized primes for different bit-widths, allowing users to apply them directly in future work without the need for further re-compute.
(4) Reducing communication overhead in GC conversions. We introduce significant optimizations for the conversion between Boolean and arithmetic GC, achieving a 50% reduction in garbled table size for BC gates and a 48.6% reduction for BD gates, on average.

In Section 3 we detail our optimizations for arithmetic GC and mixed GC conversions. Section 4 and 5 characterize the performance improvements from these optimizations and discuss the trade-offs between different GC realizations and approaches. To provide a practical perspective, we apply our findings to a case study involving a specific machine learning model in Section 6. We conclude our discussion in Section 8. Our code is publicly available at this repository [1].

2 PRELIMINARIES

This section provides a brief primer on GC, aiming to equip the reader with sufficient background to understand our contributions. For a complete review, we refer those interested to related material [15, 73]. We denote sets using $i \in [k]$ to represent $i \in \{0, 1, \ldots, k-1\}$. $\mathbb{Z}$ denotes integers, and $\mathbb{Z}_m$ denotes the ring of integers modulo natural number m, that is, $\{0, 1, \ldots, m-1\}$. Logarithms are base 2.

2.1 Boolean Garbled Circuit

We adhere to the garbling schemes abstraction and security definitions outlined in [9, 47]. Briefly, semi-honest GC is a type of PPC with two main phases: garbling and evaluation. It allows two parties, Alice (garbler) and Bob (evaluator), to jointly compute $y = f(a, b)$ on secret inputs: a from Alice and b from Bob. During the garbling phase, the garbler transforms a circuit f into its garbled version F, and defines encoding and decoding schemes to convert between plaintext and garbled values. The evaluation phase uses the garbled circuit and garbled inputs to generate garbled outputs, which can then be decoded to obtain the plaintext result.

Classic Boolean GC [15] represents functions using combinational logic circuits and binary values as inputs. Each operator is referred to as a gate (typically AND and XOR), and the binary operands are called wires (gate inputs and outputs). For each wire, the garbler selects two random wire labels W^0 and W^1, where each W^x is a λ-bit binary string encoding the truth value x. Here, λ denotes the security parameter (in the paper we implement $\lambda = 128$).

The table is shuffled before being sent to the evaluator. For a typical fan-in-2 gate, the garbler generates a garbled table consisting of four rows of ciphertext as the gate's property. Similar to the gate's truth table, suppose the gate has input wires A (held by the garbler) and B (held by the evaluator), an output wire C, and gate functionality $f : \{0, 1\}^2 \rightarrow \{0, 1\}$. Each row of the garbled table is $\text{Tab} = \mathbb{E}_{W_A^a, W_B^b}(W_C^{f(a,b)})$ for $a \in \{0, 1\}$ and $b \in \{0, 1\}$. Here $\mathbb{E}_k(m)$ denotes an encryption scheme, meaning the input wire labels are used as keys to encrypt the output wire label. The garbler sends the garbled table Tab and the label W_A^a representing her input to the evaluator. The evaluator obtains his label W_B^b corresponding to his value b via oblivious transfer [11, 37, 79]. At the evaluator side, since he holds one wire label per wire, he can decrypt only one ciphertext per gate from Tab and learn one label for the output wire. This output label can be used as an input for subsequent gates, or decrypted as a plaintext result.

Many optimizations have been proposed since the classic Boolean GC scheme. **Point-and-permute** [12] allows the evaluator to decrypt only one ciphertext instead of attempting to decrypt all four. In this technique, a random "color bit" p^0 and $p^1 = p^0 \oplus 1$ are appended to the end of each wire label, ensuring that W^0 and W^1 have opposite color bits. The color bit p^b does not reveal any information about the truth value b. The garbler arranges the ciphertexts in the garbled table according to the color bits of the input wire labels. For instance, if W_A^0 has color 1 and W_B^0 has color 0, then the ciphertext encrypting from input labels W_A^0 and W_B^0 is placed in the third row (binary 0b10) of the garbled table. This way, the evaluator only needs to decrypt one ciphertext, indicated by the color bits of the input wire labels.

Row reduction [58] eliminates one ciphertext from the garbled table. Instead of choosing the output wire labels W_C^0 and W_C^1 at random, the garbler selects one label such that the first ciphertext of garbled table is always an all-zero string. For example, if the first ciphertext is $\mathbb{E}_{W_A^0, W_B^1}(W_C^0)$, then setting $W_C^0 = \mathbb{E}^{-1}_{W_A^0, W_B^1}(\mathbf{0}^\lambda)$ makes the first ciphertext to be zero, where $\mathbb{E}^{-1}$ is the decryption of $\mathbb{E}$ and $\mathbf{0}^\lambda$ is a λ-bit zero string. Consequently, only the remaining three rows of the garbled table need to be communicated.

FreeXOR [43] enables XOR gate computation in GC without garbled table. It fixes the relationship between labels W^0 and W^1. The garbler randomly selects a global label $\Delta \in \{0, 1\}^\lambda$, and W^0 for a wire. Then, W^1 is set to $W^1 = W^0 \oplus \Delta$. The global label Δ is consistent across the entire circuit, meaning any wire label encoding $x \in \{0, 1\}$ can be expressed as $W^x = W^0 \oplus x\Delta$. This allows the evaluator to compute the XOR of two wire labels directly (without using garbled table or any encryption), obtaining the correct garbled XOR result $(W_A^a \oplus W_B^b) = (W_A^0 \oplus W_B^0) \oplus (a \oplus b)\Delta$, by setting $W_C^0 = W_A^0 \oplus W_B^0$ as the output label at 0. Naturally, $W_C^1 = W_A^0 \oplus W_B^0 \oplus \Delta = W_C^0 \oplus \Delta$. To be compatible with point-and-permute, the color bit can be appended to the wire label and the global label. By setting the last bit of Δ to 1, every pair of wire labels W^0 and W^1 will have opposite color bits.

HalfGate [78] is a technique that requires only two ciphertexts per garbled AND gate and is compatible with FreeXOR. These make the implementation of AND, XOR, and NOT gates (i.e., XORing 1) efficient, achieving functional completeness in GC. A logic circuit

[1]https://github.com/jianqiaomo/mixed_boolean_arith_garble

with HalfGate and FreeXOR achieves low ciphertexts and encryption needs.

2.2 Arithmetic Garbled Circuit

Arithmetic GC was introduced in [4] as a natural extension of Boolean GC, where the logic circuit is generalized to an arithmetic circuit. Instead of representing binary truth values $\{0, 1\}$, the wire labels in arithmetic GC encode values $x \in \mathbb{Z}_m$. Advancements within this framework were made in [7, 9], extending FreeXOR from modulus 2 to modulus p, thereby enabling free garbling of addition gates. The HalfGate technique was generalized to efficiently support multiplication gates. These specific improvements are implemented in [27]. Although there are other promising developments in arithmetic GC [8, 33], they are either not yet implemented or incompatible with mixed GC, which limits their applicability to our goal of developing a practical mixed GC.

In arithmetic GC, a wire label encoding $x \in \mathbb{Z}_p$ is denoted by $W^x \in \mathbb{Z}_p^l$, where p is a prime number, and W^x is a vector with elements in $\mathbb{Z}_p$, and length l. For different modulus p, a unique global label $\Delta_p \in \mathbb{Z}_p^l$ is sampled at random. Similar to FreeXOR, the wire label is defined as $W^x = W^0 + x\Delta_p$, where the addition and multiplication is performed element-wise in modulo p. The point-and-permute technique is also generalized in this context: instead of appending a binary color bit, a color "number" (an element of $\mathbb{Z}_p$) is appended to each wire label, with $1 \in \mathbb{Z}_p$ appended to Δ_p. Let α_x denote the color number of W^x; we have $\alpha_x = \alpha_0 + x$, meaning the possible color for a wire are cyclically shifted from a random value α_0, the color of W^0. Importantly, observing the color of any wire label W^x reveals nothing about its encoded value x. The label length l is determined by the security parameter λ, specifically $l = \lceil \lambda / \log p \rceil$, ensuring that each wire label is at least λ bits long. Given that AES-128 is commonly used to implement GC, the color number is typically integrated into the label itself (not externally appended), making the total length l. If p is not prime and can be factored into smaller primes, the label length should be $\lceil \lambda / \log p_0 \rceil$, where p_0 is the smallest prime factor. In practice, to be compatible with the residue number system, which will be discussed later, we consider only prime moduli in this section.

Arithmetic GC supports a variety of fundamental operations. Free addition allows for addition modulo p without requiring a garbled table, as $W_A^x + W_B^y = (W_A^0 + W_B^0) + (x + y)\Delta_p \pmod{p}$. Free public multiplication enables multiplication by a public constant c modulo p without a garbled table. General multiplication over secret values is supported through the generalized HalfGate, with the cost of garbled table $q + p - 1$ ciphertexts for multiplying two wire labels with $x \in \mathbb{Z}_p$ and $y \in \mathbb{Z}_q$. Additionally, the projection gate allows for conversion from a wire modulo p to a wire modulo q, applying an arbitrary function $\phi : \mathbb{Z}_p \to \mathbb{Z}_q$, with a garbling cost of $p - 1$ ciphertexts. This projection gate is useful for converting between Boolean (mod-2) labels and arithmetic label in mod-p.

Using a wire label in mod-p to represent $x \in \mathbb{Z}_p$ becomes inefficient and impractical when x is large, as the cost of the garbled table scales linearly with p. To address this, [9] introduced the Residue Number System (RNS) [62], which represents values over a composite primorial modulus $N = 2 \cdot 3 \cdots p_k$ using the first k primes. This approach encodes a large value $x \in \mathbb{Z}_N$ as $(\llbracket x \rrbracket_2, \llbracket x \rrbracket_3, \dots \llbracket x \rrbracket_{p_k})$, where $\llbracket x \rrbracket_{p_i}$ is the remainder of $x \bmod p_i$. Consequently, in arithmetic GC, a value x is represented by a bundle of wire labels corresponding to moduli $\{2, 3, \dots p_k\}$. For example, a bundle of wire labels in moduli $\{2, 3, 5, 7, 11\}$ is used to represent an 8-bit value $x \in \mathbb{Z}_{2^8}$. By the Chinese Remainder Theorem (CRT), each $x \in \mathbb{Z}_N$ has a unique, corresponding RNS encoding. Addition, multiplication and exponentiation are allowed in RNS, which are simply performed component-wise. For example, to add x and y, we can simply add their residues $\llbracket z \rrbracket_{p_i} = \llbracket x \rrbracket_{p_i} + \llbracket y \rrbracket_{p_i} \pmod{p_i}$ for each p_i in RNS encoding.

In summary, arithmetic GC supports operations including addition, multiplication, and exponentiation with a public exponent, which offer significant advantages over Boolean GC. However, arithmetic GC with RNS becomes less general, missing efficient bit-wise operations which are fundamental to modern computer architecture. This limitation poses challenges for securely computing operations like comparison, division, and floating-point. While the primorial mixed-radix (PMR) system in [7] helps build comparison gate and ReLU functions in arithmetic GC, converting from RNS to PMR involves non-prime moduli which have longer label length and are not implemented. In this paper we rather focus on a more general problem of implementing the conversion between RNS arithmetic and Boolean.

2.3 Mixed Boolean and arithmetic GC

Mixed circuits integrate Boolean and arithmetic computations by combining Boolean wire bundles with RNS arithmetic wire bundles. The foundation of these circuits includes standard Boolean gates, arithmetic operations, and specialized gates for conversion between Boolean and RNS arithmetic. Specifically, these conversions are achieved through bit-decomposition (BD) and bit-composition (BC) gates. In fact, the BC gate, which converts Boolean wire labels to RNS arithmetic, can be constructed using existing tools. To convert a binary value of b bits into RNS arithmetic, the following transformation is applied: $\llbracket x \rrbracket_{p_j} = \sum_{i=0}^{b-1} 2^i x_i \pmod{p_j}$, where $p_j \in \{2, 3, \cdots p_k\}$ and x_i represents the i-th bit. This can be achieved by leveraging the projection gate and free addition in each modulus p_j. However, simply decomposing each RNS digit into Boolean using the projection gate does not yield a correct Boolean encoding of the original value.

To efficiently convert RNS arithmetic to Boolean, a mod-p^k arithmetic wire label is introduced in [47], where p is a prime and k is a natural number, along with BD/BC gates for the mod-p^k label. The mod-p^k wire label is similar to general arithmetic wire label in mod-p: it is a vector of $\mathbb{Z}_{p^k}$-elements; it represents $x \in \mathbb{Z}_{p^k}$ in the form of $W^x = W^0 + x\Delta_{p^k}$, with an appended color number for point-and-permute. A key difference is that the length of mod-p^k label $l = \lceil \lambda / \log p \rceil$. Then, since each $\mathbb{Z}_{p^k}$ element is $k\lceil \log p \rceil$-bit, the label is at least $k\lambda$-bit in total, making it k times longer than the general arithmetic label in mod-p.

This extended mod-p^k wire label also supports free addition and free multiplication with a public constant. In [47], it provides a BC gate that merges b Boolean labels to a mod-p^k label, and a BD gate that converts a mod-p^k label to k Boolean labels. The primary advantage of the mod-p^k wire label lies in its capacity to represent

Algorithm 1 Improve primes selection for arithmetic GC

```
Input: Bit-width requirement $b$, initial $k$ primes set $\{2, 3 \dots p_k\}$
Output: Optimized primes set $\{p_1 \dots p_m\}$
 1: function PrimeSelect
 2:     $d \leftarrow \frac{2 \cdot 3 \dots p_k}{2^b}$
 3:     if $d < 2$ then
 4:         return $\{2, 3 \dots p_k\}$
 5:     else
 6:         $n \leftarrow \max\left\{j \mid \prod_{i=1}^{j} p_i \le d\right\}$
 7:         $excl \leftarrow \{2, 3, \dots p_n\}$
 8:         $p_d \leftarrow \max\{p_i \mid p_i \in \{2, 3 \dots p_k\}$ and $p_i \le d\}$
 9:         for $l \leftarrow 1$ to $n - 1$ do
10:             for $\{p_i\}_l \leftarrow$ GetCombination($l$, $\{2, 3 \dots p_d\}$) do
11:                 if $\sum \{p_i\}_l > \sum excl$ and $\prod \{p_i\}_l \le d$ then
12:                     $excl \leftarrow \{p_i\}_l$
13:         $rest \leftarrow \{2, 3 \dots p_k\} \setminus excl$
14:         $s \leftarrow \sum rest$
15:         if $k - n > 2$ then for $m \leftarrow k - n - 1$ to 1 do
16:             if $(\frac{s}{m})^m < 2^b$ then
17:                 return $rest$
18:             else
19:                 $P \leftarrow \{2, 3, \dots, p_r \mid p_r \le s\}$      ▷ Primes that $\le s$
20:                 for $\{p_i\}_m \leftarrow$ GetCombination($m$, $P$) do
21:                     if $\sum \{p_i\}_m < s$ and $\prod \{p_i\}_m \ge 2^b$ then
22:                         $rest \leftarrow \{p_i\}_m$
23:                         $s \leftarrow \sum rest$
24:         return $rest$
```

large $\mathbb{Z}_{p^k}$ value, facilitating the reversion of an RNS encoding to its original value. For an RNS system $N = p_1 p_2 \cdots p_k$, where x is encoded as $(\llbracket x \rrbracket_{p_1}, \cdots \llbracket x \rrbracket_{p_k})$, there exist constants $c_1, \cdots c_k \in \mathbb{Z}_N$ that allow the recovery of x via $x = \sum_{i=1}^{k} c_i \cdot \llbracket x \rrbracket_{p_i} \pmod N$. These constants can be precomputed using the CRT when the set $\{p_1, p_2 \cdots p_k\}$ is determined. Thus, by merging RNS arithmetic labels into a mod-p^k label using the BC gate, performing a modulo N operation for the actual value, and then applying a BD gate, we can successfully convert RNS arithmetic label bundle into Boolean labels. The modulo N operation for mod-p^k label is also realized by BD gates, BC gates, free addition and multiplication in [47]. Figure 2 illustrates the key (non-free) steps in this conversion process.

3 OPTIMIZING MIXED GARBLED CIRCUIT

Classic Boolean GC is commonly deployed [14, 22, 40, 48, 72], while arithmetic GC is realized in fancy-garbling [27]. We improve the bit-composition (BC) and decomposition (BD) gates based on existing work, employing a grab-and-go design to facilitate the integration of mixed GC. Before incorporating the BC and BD gadgets, we enhance the current implementation of arithmetic GC. Following these enhancements, we design the concrete BC and BD gates, select parameters based on real-world requirements, and reorganize the algorithm to minimize actual communication using row reduction.

3.1 Prime Reduction in Arithmetic GC

Our starting point is to optimize the prime selection in arithmetic GC. To garble arithmetic gates over large moduli, as done in [27], they use the Chinese Remainder Theorem (CRT), also known as Residue Number System (RNS) to break down computations over a large ring into smaller primorial modulus rings. Details are provided in Section 2. Given a bit-width b, the standard approach is to select the first k primes and construct over ring $\mathbb{Z}_N$ where $N = 2 \cdot 3 \dots p_k$ so that $\mathbb{Z}_N$ can fit b bits (i.e., $N \ge 2^b$). While it is intuitive to select the first k primes because they are small yet their product exceeds 2^b, the communication and computation costs in RNS-based arithmetic GC depends on the sum of the primes. Therefore, our optimization is to find a set of primes whose product is sufficiently large to fit the given bit-width b, while **minimizing their sum**. Selecting small individual primes does not necessarily yield the smallest final sum. Although there is research on prime selection in RNS, it does not account for the cost in GC, instead they prefer fewer or closer primes [62].

Finding a mathematical method to identify the smallest prime sum with the product large enough to meet requirements is challenging. A naive approach involves going through all possible prime combinations and filtering those that meet the requirements. However, this approach is expensive due to the vast number of potential combinations and the broad range of numbers to consider without additional constraints. For example, if we need three primes to represent a range of 2^8, there would be over 24,000 potential combination possibilities to randomly choose three primes. To address this, we introduce a new heuristic method to select the fewer RNS primes to optimize arithmetic GC latency. Instead of randomly choosing primes, we adopt an incremental approach that eliminates unnecessary primes, significantly reducing the number of possible traversals.

Our approach consists of two steps. We start by removing unnecessary primes from the initial set of the first k continues primes generated by the original method. Then, we demonstrate that no other prime set can perform better than the resulting primes. To remove unnecessary primes, we calculate how far the product of the k primes exceed the target 2^b by dividing them. The division $d = \frac{2 \cdot 3 \dots p_k}{2^b}$ indicates the largest prime that can be removed. If d is less than 2, no primes can be removed. With d calculated, we find the longest removable prime set by maximizing the length n such that the product of the first n primes $2 \cdot 3 \dots p_n \le d$. In other words, we can remove at least one and up to n primes. We then go through the combination of primes not greater than d, with a fixed combination length $l < n$. Note that for 2^{64}, the original prime set contains just 16 primes, and the primes that no greater than d are even fewer, making it feasible to evaluate all combinations quickly. This allows us to identify all removable prime combinations. By removing the combinations with the largest sum, we get the remaining primes that meet the 2^b requirement while having a smaller sum. The sum of remaining primes is minimal as replacing any of them with a smaller one would fail to meet the 2^b requirement, and replacing any with a larger one would increase the sum.

For example, to represent 2^{16}, the initial set is {2, 3, 5, 7, 11, 13, 17}, with a product of 510,510 which is 7.79× than 2^{16}. The longest combination we can remove is {2, 3}, leaving {5, 7, 11, 13,

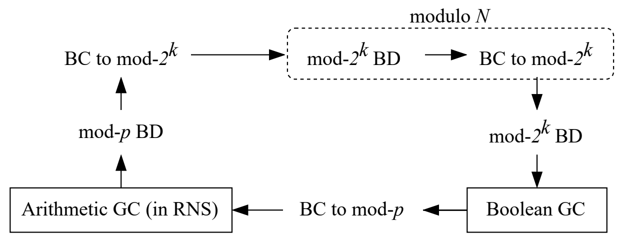

Figure 2: The overall conversions flow. Some miscellaneous operations such as free addition are omitted. The path on the upper shows the BD path to convert RNS arithmetic to Boolean, while the lower part shows the BC from Boolean to RNS arithmetic.

17}. Alternatively, we can remove {7}, then we have {2, 3, 5, 11, 13, 17}. We prefer the latter set due to its smaller sum.

After finding the smallest sum s and the resulting prime set, we take an additional step to ensure that no prime combinations with lengths lower than $k-n$ can meet the 2^b requirement while having a smaller sum. Suppose there exists a better combination with length $m < k - n$ that has lower sum $\sum_{i=1}^{m} q_i < s$. These m primes are not necessary the first m primes. According to the AM–GM inequality we have $\prod_{i=1}^{m} q_i \leq (\frac{\sum_{i=1}^{m} q_i}{m})^m < (\frac{s}{m})^m$. Note that for $m < k - n$, the expression $(\frac{s}{m})^m$ is monotonically increasing[2] with m. Thus if the upper bound of product $\prod_{i=1}^{m} q_i$, which is $(\frac{s}{m})^m$ at $m = k-n-1$, is smaller than 2^b, we can conclude that such combination with shorter length and smaller sum assumption cannot fit 2^b. Detailed steps are shown in Algorithm 1.

This targeted approach allows us to efficiently find an optimal set of primes. The key idea is to narrow the range of potential combinations, and skip generating those unnecessary combinations to speed up the finding. The selection of primes can be applied to all future arithmetic GC implementations. We show the primes in Appendix Table 7. The original method using the first k primes has 6.9× exceeding the necessary range on average, while the optimized primes are only 1.2× exceeding. For the primes that are optimized, their sum is 5.3% less than the original primes on average.

3.2 Optimize Mixed GC Conversion

Practical real-world mixed GC involves conversions between RNS-based arithmetic GC and Boolean GC. When converting labels from Boolean to RNS arithmetic, the Boolean labels are composed into each prime modulus of the RNS (see Section 2.3). The conversion from RNS-arithmetic to Boolean is more complex, requiring a longer mod-p^k arithmetic label as intermediate variables. To streamline the process, we choose $p = 2$, eliminating the need for an additional conversion from mod-p to mod-2. Thus, the mod-p^k arithmetic label is reduced to a mod-2^k arithmetic label. The core idea behind the conversion is to merge all RNS remainders using a linear function, similar to the CRT inversion. The result, being large, requires a mod-2^k arithmetic label to fit. We then compute modulo N to obtain

[2]For $k - n > 2$, the $s/e = \frac{\sum_{rest} p_i}{e} > \frac{(k-n)e}{e} > k - n$ because s comes from at least $(k - n)$ primes. That is, $m < k - n < s/e$. The expression $(\frac{s}{m})^m$ is monotonically increasing when $m < s/e$.

Table 1: The number of garbled table rows (each representing λ bits) required for 4 to 64-bit RNS arithmetic and Boolean BD/BC conversions. The improvement column shows the reduction in communication overhead achieved through optimization.

$\mathbb{Z}$	BD	Opt. BD	Imp.	BC	Opt. BC	Imp.
2^4	539	275	48.98%	16	8	50.00%
2^8	1827	937	48.71%	48	24	50.00%
2^{16}	4998	2582	48.34%	160	80	50.00%
2^{32}	18138	9322	48.61%	576	288	50.00%
2^{64}	62837	32377	48.47%	1920	960	50.00%

the correct value, and finally, apply bit-decomposition to produce the Boolean labels. The overall flow diagram is shown in Figure 2. It includes four significant processes: mod-p BD/BC and mod-2^k BD/BC.

In contrast to [47], we can leverage the intrinsic projection gate to construct mod-p BD/BC operations. The projection gate, as described in [9, 27], enables swapping between Boolean labels and mod-p arithmetic labels. For mod-2^k BC/BD conversions, we further optimize these operations using Row Reduction [58]. Originally, the garbled table for mod-2^k BC/BD consists of ciphertexts encrypted by applying a secure hash function H to the input labels. Instead of randomly choosing output labels (the Boolean labels of BD, or the mod-2^k label of BC), the garbler can select them such that the first ciphertext of the garbled table is always an all-zero string. This reduces communication because the all-zero row in the garbled table does not need to be sent. Row reduction is particularly effective in optimizing conversions between Boolean and RNS arithmetic: converting RNS arithmetic to Boolean invokes multiple base mod-2^k BC/BD operations, and the optimization to these base operations culminate in a significant overall improvement.

We have adapted the mod-2^k BC/BD methods from [47] to incorporate row reduction. Detailed steps of the adjusted garbling procedure are shown in Figure 3. In mod-2^k BC, the original method generates the resulting mod-2^k label by sampling random B_i. Row reduction allows us to set B_i such that $\mathsf{C}_{i,0}$ is zero, enabling the garbler to omit sending that row of Tab to the evaluator. Since a mod-2^k label is k times longer than the hash, we use an XOR similar to the output feedback (OFB) mode block cipher, with the K_i as the key to encrypt the longer mod-2^k label. To generate B_i, we encrypt a long zero string, pack the result as a mod-2^k label, and add the global label to ensure $\mathsf{C}_{i,0}$ returns zero correctly. The mod-2^k BD is similar but includes a bit-composition sub-process to generate each $(\mathsf{A}^{(i)}, \alpha^{(i)})$, which is then used to set up the K_i to reduce row $\mathsf{C}_{i,0}$.

Using projection gate (which is row-reduced) in mod-p BD, and mod-p BC from b Boolean labels, the size of garbled table needed for communication is reduced from $\lambda p \log(p)$ to $\lambda(p-1)\log(p)$ bits, and from $2b$ to b bits, respectively. After optimizing the mod-2^k BD and mod-2^k BC from b Boolean labels, the size of garbled table needed for communication is reduced from $\lambda(k^2 + 3k - 2)$ to $\frac{1}{2}\lambda(k^2 + 3k - 2)$ bits, and from $2\lambda bk$ to λbk bits, respectively. Here λ denotes the security parameter. By combining these base operations, we significantly reduce the communication overhead

Garbling mod-2^k bit-composition takes:
Input: k Boolean labels K_i whose color bit is α_i ($i \in [k]$), a mod-2^k global label A, and the Boolean global label Δ'.
Output: a garbled table Tab, and a mod-2^k labels B.

- For gate of id, $i \in [k]$, $\beta \in \{0, 1\}$

$$\mathsf{C}_{i,\beta+\alpha_i \pmod 2} \leftarrow \mathsf{H}(\mathsf{K}_i(\beta), (\mathsf{id}, i)) \bar{\oplus} (2^i \beta \mathrm{A} + \mathrm{B}_i \pmod{2^k})$$

  Since mod-2^k labels are $k\times$ longer than the hash of K_i, a block cipher mode of XOR operation $\bar{\oplus}$ is used. The (mod 2^k) is applied element-wise in the label. The output Tab will consist of ciphertext $(\mathsf{C}_{i,\beta})_{i\in[k],\beta\in\{0,1\}}$.
- For $i \in [k]$, we can choose

$$\mathrm{B}_i \leftarrow \left(\mathsf{H}(\mathsf{K}_i(\alpha_i), (\mathsf{id}, i)) \bar{\oplus} \mathbf{0}^l\right) + 2^i \alpha_i \mathrm{A} \pmod{2^k}$$

  where l is mod-2^k label bit-length, $\mathbf{0}^l$ is a l-bit zero string. Final output $\mathrm{B} \leftarrow \sum_i \mathrm{B}_i \pmod{2^k}$.
  During the communication of Tab, for $i \in [k]$, garbler can omit sending $\mathsf{C}_{i,0}$ and evaluator sets $\mathsf{C}_{i,0} \leftarrow \mathbf{0}^l$.

Garbling mod-2^k bit-decomposition takes:
Input: a mod-2^k label A whose color number is α, a mod-2^k global label Δ, and a Boolean global label Δ'.
Output: a garbled table Tab, and k Boolean labels K_i, $i \in [k]$.

- For gate of id, $i \in [k]$, $\beta \in \{0, 1\}$,

$$\mathsf{C}_{i,\beta+\alpha^{(i)} \pmod 2} \leftarrow \mathsf{H}(\beta\Delta + \mathrm{A}^{(i)} \pmod 2, (\mathsf{id}, i)) \oplus (\beta\Delta' \oplus \mathsf{K}_i)$$

  where $(\mathrm{A}^{(0)}, \alpha^{(0)}) = (\mathrm{A}, \alpha)$. For $0 \leq i < k-1$, a mini bit-composition is invoked as a sub-process that takes K_i and generates $(\mathrm{A}^{(i+1)}, \alpha^{(i+1)})$. The output Tab will consist of ciphertext $(\mathsf{C}_{i,\beta})_{i\in[k],\beta\in\{0,1\}}$.
- For $i \in [k]$, we can choose

$$\mathsf{K}_i \leftarrow \mathsf{H}(\alpha^{(i)}\Delta + \mathrm{A}^{(i)} \pmod 2, (\mathsf{id}, i)) \oplus ((\alpha^{(i)} \bmod 2)\ \Delta')$$

  During the communication of Tab, for $i \in [k]$, garbler can omit sending $\mathsf{C}_{i,0}$ and evaluator sets $\mathsf{C}_{i,0} \leftarrow \mathbf{0}^l$, where l is Boolean label K_i bit-length. The (mod 2) is applied element-wise in the label.

Figure 3: Improve mod-2^k BC and BD algorithm with row reduction scheme. Symbols follow as in [47].

for the garbled table during the conversion between RNS arithmetic and Boolean. Table 1 shows that, on average, the garbler sends 48.6% fewer for the garbled table of RNS arithmetic BD. The key step of BC into RNS arithmetic is the mod-p BC, where we obtain 50% communication overhead improvement.

4 METHODOLOGY

In this section, we describe our experimental setup. The runtime and communication costs for each benchmark were measured on an Intel Core i7-10700K processor running at 3.80GHz [36]. GC implementation leverages the AES-NI instruction set [1] to ensure high performance and provide a competitive baseline.

We focused on two types of workloads: one supported by both arithmetic and Boolean GC and the other is Boolean-only. Typically, each workload involves two secret inputs – one from the garbler and another from the evaluator. We conducted a series of experiments across various input bit-widths. Boolean-specific benchmarks include bitwise *XOR*, bitwise *AND*, division (*Div*), modulo (*Mod*), and greater-than comparison (*Geq*). Common benchmarks that can be executed in both Boolean and RNS arithmetic GC include addition (*Add*), subtraction (*Sub*), multiplication on secret values (*Mul*), multiplication with a public constant (*Pub Mul*, Boolean realized by bit-shifting and addition), multiplexer (*Mux*, computing $c = s \cdot a + (1-s) \cdot b$ where s is a {0 or 1} secret selector), and public exponentiation (*Pub Exp*, computing a^c where c is a public constant). These workloads were compiled and repeatedly executed to profile their performance.

5 EVALUATION

In this section, we assess the optimization of arithmetic GC through the use of fewer prime, and compare the performance of the optimized arithmetic GC with Boolean GC. The evaluation focuses on communication costs – including garbled inputs/outputs, oblivious transfer, and garbled tables – as well as the runtime of garbling and evaluating the circuit. We profile both communication and runtime latency, as we believe their combined impact is crucial for determining end-to-end performance.

5.1 Prime Reduction Optimization

We begin by showing the effectiveness of optimizing prime selection in RNS arithmetic GC. In this experiment, we profile the workloads supported by arithmetic GC and compare the communication overhead and computation latency between the original RNS primes and the optimized primes. Figure 4 highlights the communication differences, showing reduced overhead with optimized primes in arithmetic GC, and Figure 5 illustrates the improved runtime achieved through better prime selection. Given the varying complexity of each function, their communication overhead and

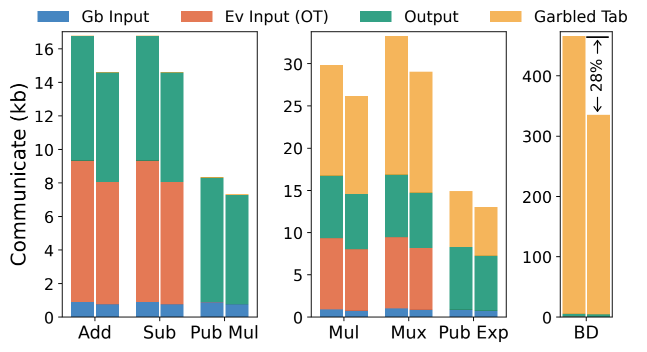

Figure 4: Comparison of communication overhead for 16-bit arithmetic GC. For each operation, left bar represents the uses of original RNS primes; right bar reflects the optimized primes. The total communication is broken down into parts.

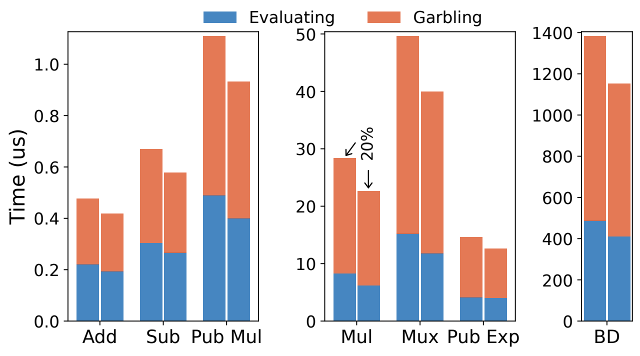

Figure 5: Comparison of run time overhead for 16-bit arithmetic GC. For each operation, left bar represents the uses of original RNS primes; right bar reflects the optimized primes. The total time is broken down into garbling and evaluation.

computation latency differ significantly. To provide clearer insights, we categorize the functions into three groups, each with distinct y-axis scales, allowing us to present the differences more effectively. For instance, the BD function, which converts RNS arithmetic to Boolean, is the most complex, resulting in substantially higher communication and runtime costs; thus, it is displayed in a separate subplot.

Our analysis yields two key observations. First, the results show that optimizing prime selection leads to a reduction in both communication overhead and runtime. In the RNS system, each base operation is performed component-wise for each prime. With fewer primes, the inputs, outputs, and garbled tables experience reduced communication overhead. Simultaneously, computation latency improves as fewer primes need to be processed. Second, the comparison reveals that communication and latency are not always correlated. A function with relatively low communication overhead may not necessarily have the shortest runtime. We find the reason that although row reduction decreases the size of the garbled table, it does not reduce the number of hash or other operation calls. For example, despite the small communication overhead of Pub Mul, it has a longer runtime than addition and subtraction due to the multiple multiplication and modulo operations involved, which slow down performance compared to simpler operations like addition and subtraction.

We also observe that the “free” operations in arithmetic GC (e.g., Add, Sub, and Pub Mul) do not generate garbled table communication, but there is still communication involved for inputs, outputs, and oblivious transfer. Regarding run time, the garbling phase generally takes longer than the evaluation phase, as the garbler must prepare the entire garbled table, while the evaluator only needs to select and evaluate one row using the point-and-permute. We repeated the experiment with different bit-width settings for arithmetic GC and selected the 16-bit setting as a representative example, as shown in Figures 4 and 5. Other bit-widths produced similar results, with differences scaled according to the improvement of the sum of primes. Notably, the BD operation saw the most significant improvement, with a 28% reduction in communication overhead, while multiplication achieved a 20% decrease in latency. Across all the benchmarks, GC with optimized primes demonstrated a 14.1% reduction in communication overhead and a 15.7% decrease in total run time. This aligns with our expectations, as Table 7 shows that the optimized primes have a 12.1% lower sum than the original RNS primes for 16-bit, which translates into the observed performance gains.

5.2 Evaluating GC Realization Strategies

With the availability of conversion gates in GC, we can optimize a particular function between Boolean or a mixed GC realization. If a function consists entirely of arithmetic operations such as addition and multiplication, it is clearly more efficient to use arithmetic GC due to its lower cost. Conversely, for functions that are composed entirely of Boolean operations or other operations unsupported by arithmetic GC, Boolean GC is the clear choice. However, many functions contain a mix of operations – some of which are better handled by arithmetic GC, while others are only supported by Boolean GC. For such mixed circuits, the key is to ensure that the benefits of using arithmetic GC outweigh the conversion costs. Otherwise, it is more efficient to remain in Boolean GC without incurring those costs.

In this section, we identify the balance point where the advantages of arithmetic GC offset the conversion expenses by comparing traditional Boolean realizations with mixed GC realizations. There are three possible approaches for implementing a function. **Pure Boolean** GC: general approach to support any function. **Boolean + BC + arithmetic**: This approach combines Boolean and arithmetic GC with bit-composition gates, assuming the function can be divided into Boolean and arithmetic components connected by a BC gate. That is, the BC gate and arithmetic operations replace part of the pure Boolean circuit. **Arithmetic + BD + Boolean**: This method uses arithmetic and Boolean GC with bit-decomposition gates. By comparing the second and third approaches to the first, we can identify scenarios where the benefits of arithmetic GC surpass the conversion costs, leading to a lower overall cost than the traditional pure Boolean GC.

This analysis is performed for two GC approaches: stream GC and GC with preprocessing.

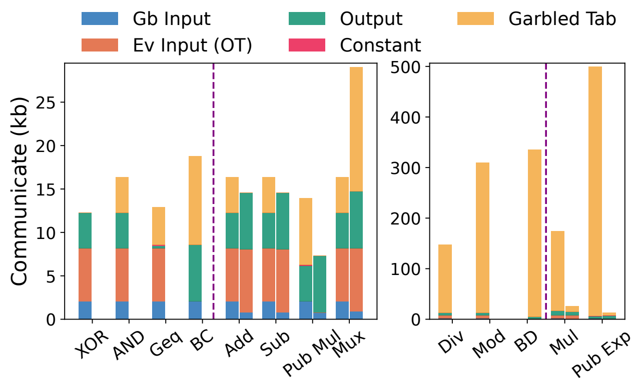

Figure 6: Communication comparison for 16-bit operations in Boolean vs arithmetic GC. Each sub-figure is dash-divided: the right side shows common operations with two bars per operation (left bar for Boolean, right bar for arithmetic); the left side displays operations unique to either Boolean or arithmetic GC with a single bar.

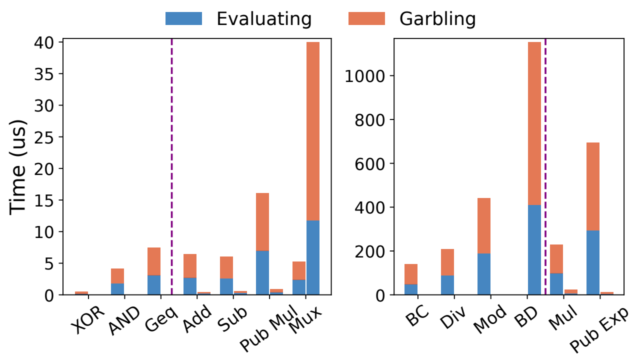

Figure 7: Run time comparison for 16-bit operations in Boolean vs arithmetic GC. Each sub-figure is dash-divided: the right side shows common operations with two bars per operation (left bar for Boolean, right bar for arithmetic); the left side displays operations unique to either Boolean or arithmetic GC with a single bar.

5.2.1 Stream GC. A direct approach to implementing GC is through streaming, where the circuit is processed sequentially as a stream. As each gate is garbled, it is immediately transmitted to the evaluator without waiting for the entire circuit to be completed. The evaluator can begin evaluating the received garbled gate right away, eliminating the need to store the entire garbled circuit. This method allows for the garbling and evaluation phases to overlap, reducing idle time since the evaluator doesn't have to wait for the full circuit to arrive. In this setup, the communication of the garbled table becomes crucial, as it is transmitted in real-time. The overall latency will be largely determined by the garbling phase, given that evaluation can occur simultaneously.

We present the detailed communication costs for each 16-bit benchmark in Figure 6, and the corresponding garbling and evaluation latency in Figure 7. To enhance clarity, the results are divided into two scales. Each subplot categorizes the benchmarks into two types: Boolean-only or arithmetic-only (i.e., BD) operations, and common functions that can be executed in both Boolean and arithmetic GC, where two bars are used to compare the costs. Starting with the communication analysis, it is evident that for most common functions, arithmetic GC incurs lower costs than Boolean GC, particularly in terms of reduced garbled table, because Boolean has to process many gates. The output communication in arithmetic is slightly higher than in Boolean GC: In Boolean, the garbler sends two label hashes per wire, while in arithmetic GC, the garbler sends hashes of all p labels for each wire in mod-p, leading to increased communication. Focusing on garbled table size is crucial. Suppose the numbers of input and output are fixed, the cost of different functions will manifest as in the garbled table, represented by the yellow portion in Figure 6. An exception is the multiplexer gate, where the arithmetic implementation has a higher cost. This is because the Boolean multiplexer can be constructed using a small number of simple AND and XOR operations, while the arithmetic multiplexer requires addition and multiplication, which are more resource-intensive. Therefore, it is generally advantageous to execute these common functions using arithmetic GC, except multiplexer which performs better in Boolean GC. Similar to Figure 4 and 5 in Section 5.1, communication and latency are not always correlated. In the common functions (except Mux), the arithmetic GC reduces the total communication by an average of 34.2% compared to Boolean GC. Additionally, arithmetic GC speeds up total run time by 17.3×. These findings suggest that if a function can be fully implemented in arithmetic GC, it should be.

To decide whether to garble a mixed circuit or use pure Boolean GC throughout, it is crucial to weigh the benefits of arithmetic operations against the additional costs of conversion. We define two key metrics: the **BC equivalent ratio**, $r_{op}^{BC} = \frac{c_{BC}}{\Delta c_{op}}$, and the **BD equivalent ratio**, $r_{op}^{BD} = \frac{c_{BD}}{\Delta c_{op}}$. Here, the numerator (c_{BC}, c_{BD}) represents the cost of a BC or BD gate, while the denominator reflects the cost difference of a common operation between Boolean and arithmetic GC. In the context of stream GC, we primarily consider the garbled table communication and garbling time. These ratios indicate how many instances of a common operation in arithmetic GC are needed to offset the extra cost of a BC or BD gate compared to using Boolean GC alone. For example, the BC ratio for a 16-bit addition operation, based on garbling time, is $r_{Add}^{BC} = 26.1$; it means that for a function with fixed numbers of inputs and outputs, if it includes 26.1 addition operations, the time to garble a pure Boolean circuit is equivalent to garbling a BC gate and performing these additions in arithmetic GC. In other words, if a function has more than 26 add operations, it is advantageous to garble a "Boolean + BC + arithmetic" mixed GC. Conversely, if the number of additions is fewer than 26.1, it is preferable to use pure Boolean GC, as the BC gate would negate the speed gains from arithmetic additions.

We conducted experiments to determine the BC and BD equivalent ratios for each common function across various bit-widths. In the stream GC, we focus on the ratios related to garbled table communication (Figures 8A and 9A), and garbling latency (Figures 8B and 9B). Generally, the BD ratio is larger than the BC ratio because BD operation is more costly than BC, leading to a larger numerator. For the garbled table communication, as the bit-width increases, the BC and BD ratios for Mul and Pub Exp decrease because their

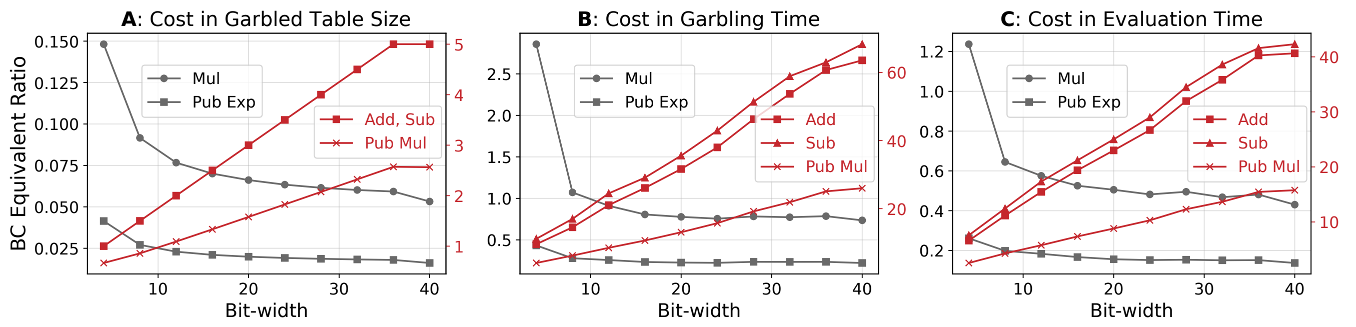

Figure 8: BC equivalent ratio of common operations. Comparison based on garbled table communication, garbling time, and evaluation time. Red lines correspond to the right y-axis. In sub-figure A, the “Add” line overlaps with the “Sub” line.

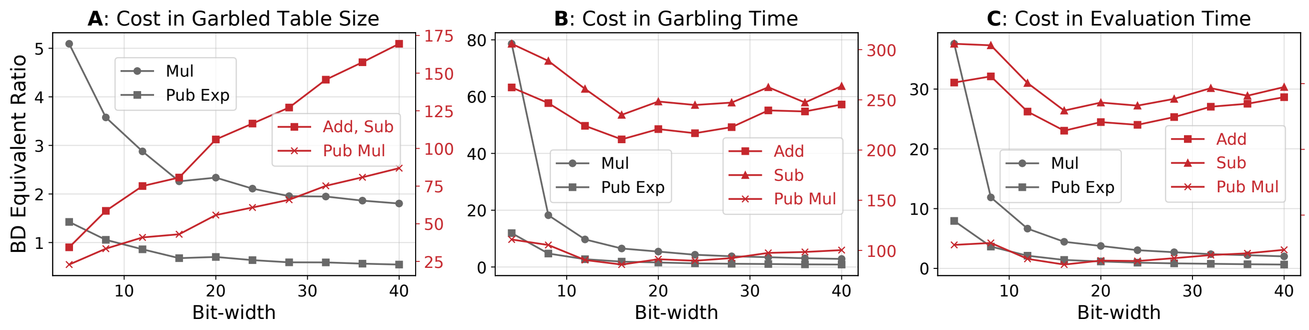

Figure 9: BD equivalent ratio of common operations. Comparison based on garbled table communication, garbling time, and evaluation time. Red lines correspond to the right y-axis. In sub-figure A, the “Add” line overlaps with the “Sub” line.

Boolean costs increase more rapidly than their arithmetic counterparts. Notably, the BC ratio for Mul and Pub Exp drops below 1, indicating that converting from Boolean to arithmetic for these operations will be always cost-effective (assuming the subsequent circuit remains in arithmetic). For Add, Sub and Pub Mul, their ratios in garbled table are related to the number of RNS primes, leading to an overall increase with bit-width.

When we examine garbling time, we find that the ratios are higher than those for communication. This discrepancy arises because optimizations reduce garbled table communication but do not affect computation, resulting in relatively high run times for BC and BD operations. The ratios for Mul and Pub Exp continue to decline. The BC ratio trends for Add, Sub, and Pub Mul indicate that the garbling time for BC gate increases faster than the advantages gained from arithmetic. This means that for higher bit-widths, more Add, Sub, or Pub Mul operations are needed to offset the extra cost of BC gate. On the other hand, their BD ratio trends remain relatively stable, suggesting that the increase in BD garbling time is closely matched by the arithmetic advantage gain of these operations.

The equivalent ratio can guide the decision whether to garble a function as mixed GC. Their inverse ratios reveal that per common operation can offset $1/r_{op}^{BC}$ of BC gates (or $1/r_{op}^{BD}$ of BD gates). For instance, the 16-bit add operation has $1/r_{Add}^{BC} = 0.038$ regarding cost in garbling time, meaning the benefit gained from garbling an arithmetic add gate can offset the cost of 0.038 BC gates. If a function involves a combination of multiple operations, summing the inverse ratios, $\sum_{op}(1/r_{op}^{BC})$, can tell us whether these operations can collectively cover the number of BC gates required. The same logic applies to BD gates. To easily analyze and choose a GC realization for a function, we detail the inverse BC and BD ratios in Appendix Tables 5 and 6.

5.2.2 GC with Preprocessing. To minimize real-time latency in privacy-preserving computation applications, high-performance protocols divide GC into a preprocessing phase and an online phase. This approach leverages the fact that some steps are input-independent and can be prepared offline, allowing the rest computation to proceed immediately once inputs are available [30, 39, 52]. The preprocessing includes preparing evaluator’s inputs via OT and circuit garbling. When the actual inputs are ready, the evaluator begins evaluating the pre-garbled circuit. The key difference between stream and preprocessing GC is that in the latter, garbled tables are precomputed and stored on the evaluator side, rather than being streamed in real-time. Consequently, the effectiveness of deploying GC with preprocessing depends on the evaluator’s storage capacity and computational speed, which correspond to the size of the garbled tables and the evaluation run time. Faster evaluation speed directly translates to shorter real-time latency.

For garbled table storage, please refer to Section 5.2.1 and Figures 8A, 9A. Here, we focus on the BC and BD equivalent ratio when considering cost in evaluation time, as presented in Figure 8C and 9C. Generally, the trends are similar to those observed in garbling time. However, due to the point-and-permute optimization, evaluation time is shorter than garbling time. As a result, the evaluation

Table 2: Technical specifications of an example LeNet layer and the ReLU substitution.

Layer	Conv2D
Filter Size	(3, 3)
Num. of Filters	32
Strides	(1, 1)
Padding	No
Input Shape	(28, 28, 1)
Output Shape	(26, 26, 32)
Multiplication per Conv	9
Addition per Conv	8
Total Multiplications	194688
Total Additions	173056
Num. of ReLU	21632
Per Approx. ReLU:	
• **Pub Exp**	54
• **Pub Mul**	57
• **Add**	57
• **Mul**	1

cost equivalent ratios are relatively lower than those for garbling time. Similarly, for functions that combine multiple operations, the inverse ratio can also be used to assess whether it is advantageous to garble a mixed circuit.

6 A MACHINE LEARNING CASE STUDY

In this section, we apply the comparison of different GC realizations to a machine learning inference task, using it to estimate their practical applicability. This analysis provides an intuitive understanding of when garbling a mixed circuit is likely to be efficient.

We use a convolutional layer of the LeNet model [45] with MNIST dataset as an example. The layer details are shown in Table 2. This layer includes both arithmetic operations (convolution) and Boolean operations (ReLU activation), making it an ideal candidate for testing the performance of mixed GC. The convolution operations consist of multiplications and additions, while the ReLU activation is implemented using a comparison gate and a multiplexer. Previous works [23, 24, 29, 52] have explored replacing the original ReLU with other methods like high-degree polynomial approximations. Lattigo [46, 57] provides such a polynomial approximation for ReLU, involving only Add, Pub Mul, Mul, and Pub Exp operations, as listed in Table 2. We incorporate this approximation into our comparison, resulting in four GC realizations for this convolutional layer:

1. **Pure Boolean (Baseline)**: Implements all convolutions and ReLU activations using traditional Boolean GC.
2. **Full Convert**: Implements convolution in arithmetic GC, then converts to Boolean (BD) for ReLU, and finally converts back (BC) to arithmetic.
3. **Smart Convert**: Implements convolution in arithmetic GC, converts to Boolean (BD) for ReLU, but skips converting back to arithmetic since the subsequent layer may require Boolean operations (e.g., max pooling).
4. **Pure Arithmetic**: Implements both convolution and ReLU using arithmetic GC, with ReLU replaced by a polynomial approximation.

Table 3: The number of BD gate required by mixed GC for the example layer, and the BD gate that can be offset by switching convolution from Boolean to arithmetic GC.

Num. of BD Required				21632
Num. of BD Covered in	**4-bit**	**8-bit**	**16-bit**	**32-bit**
Cost in Garbled Table	43264	57393	88254	101143
Cost in Garbling Time	3136	11382	30709	57204
Cost in Evaluation Time	6043	17235	44806	82668

For the pure arithmetic approach, note that the ReLU approximation is limited only when the input x is within the range $-1 \leq x \leq 1$. Therefore, additional data processing is required before applying the approximation in the convolutional layer. It is not our target to discuss how to approximate ReLU for real-world private ML, but we include this approach in our performance comparison to provide a comprehensive overview of the trade-offs between pure Boolean GC, mixed GC, and pure arithmetic GC. ReLU approximation could also require additional adjustments (such as retraining or fine-tuning) for ML deployment due to the errors between the approximation and the actual result.

Using the inverse equivalent ratio discussed in Section 5.2, we can quickly estimate whether it is advantageous to convert to a mixed GC or to remain within pure Boolean GC. The inverse equivalent ratios are calculated based on the costs in terms of garbled table size, garbling time, and evaluation time. As shown in Table 3, the number of BD (or BC) gates required corresponds to the required ReLU operations. Instead of fully pure Boolean, by running the convolution (i.e., multiplication and addition) in arithmetic GC and converting to Boolean for ReLU with a BD gate, we aim to offset the BD gate cost with the benefits gained from arithmetic GC. If such benefit covers the BD gates required, the mixed GC is likely to outperform the pure Boolean GC. Table 3 shows in most scenarios, the benefits of arithmetic operations cover enough BD gates, especially for larger input bit-widths. However, there are exceptions with inputs of 4-bit and 8-bit, where the costs in garbling time and evaluation time fall short of covering the required BD gates. This indicates that for 4-bit and 8-bit convolution layers, the mixed GC has worse garbling and evaluation times than the pure Boolean GC.

Table 4 presents the overall comparison of the four GC realizations, considering garbled table size, garbling time, and evaluation time. The results align with our expectations from Table 3. Although the full convert involves BC gates, the BC cost is much smaller than BD. If the BD gates are adequately covered, we will find that both the full convert and smart convert approaches will outperform the pure Boolean realization. From 4-bit to 32-bit, the higher coverage of BD gates allows for more slack in mixed circuits, resulting in lower costs in the full and smart convert approaches. Conversely, in the 4-bit and 8-bit scenarios, the full and smart convert approaches show worse performance than the pure Boolean baseline, consistent with the inadequate BD gate coverage.

In most cases, the pure arithmetic approach with approximation performs better than the smart convert, and the smart convert outperforms the full convert. For larger bit-widths, mixed GC tends to outperform pure Boolean, as Boolean multiplications and additions become increasingly costly. This case study challenges the

Table 4: Layer cost comparison at different input bit-widths of four GC realizations. The comparing difference to the baseline is shown as percentage. The more reduction rate indicates the better performance.

Bit-width	4-bit	8-bit	16-bit	32-bit
Garbled Table (kbit):				
Pure Boolean Baseline	1.89E+06	7.78E+06	3.15E+07	1.27E+08
Full Convert	1.15E+06 (-39%)	3.55E+06 (-54%)	9.70E+06 (-69%)	3.27E+07 (-74%)
Smart Convert	1.13E+06 (-40%)	3.49E+06 (-55%)	9.48E+06 (-70%)	3.19E+07 (-75%)
Pure Arithmetic	1.43E+06 (-24%)	3.48E+06 (-55%)	9.22E+06 (-71%)	2.44E+07 (-81%)
Garbling Time (us):				
Pure Boolean Baseline	1.70E+06	6.68E+06	2.61E+07	1.03E+08
Full Convert	6.04E+06 (+254%)	1.18E+07 (+76%)	2.13E+07 (-18%)	5.18E+07 (-50%)
Smart Convert	5.86E+06 (+244%)	1.12E+07 (+68%)	1.94E+07 (-26%)	4.37E+07 (-58%)
Pure Arithmetic	4.30E+06 (+152%)	7.60E+06 (+14%)	1.46E+07 (-44%)	3.17E+07 (-69%)
Evaluation Time (us):				
Pure Boolean Baseline	1.28E+06	5.15E+06	1.97E+07	7.78E+07
Full Convert	3.30E+06 (+157%)	6.62E+06 (+28%)	1.12E+07 (-43%)	2.63E+07 (-66%)
Smart Convert	3.21E+06 (+150%)	6.31E+06 (+22%)	1.02E+07 (-48%)	2.24E+07 (-71%)
Pure Arithmetic	1.89E+06 (+47%)	2.98E+06 (-42%)	6.79E+06 (-65%)	1.29E+07 (-83%)

intuition that pure arithmetic or mixed GC is always superior – small bit-width Boolean operations can be more efficient than arithmetic, regarding in terms of the run time. The findings also suggest that garbling an entire convolutional layer using GC can be slow and resource-intensive, highlighting the potential for hybrid protocol solutions to enhance performance. For workloads that need to be implemented using GC, particularly if pure arithmetic GC is not feasible, we suggest estimating performance between traditional pure Boolean GC and mixed GC using the equivalent ratio (Appendix Tables 5 and 6) before implementation. This approach enables selecting the most suitable GC realization with the lowest performance overhead. At the same time, the run time of mixed GC does not gain as much advantage as the communication reduction, which suggests that GC requires the development of specialized accelerators to eliminate the run time latency.

7 RELATED WORK

Many PPC frameworks incorporate a combination of secure MPC protocols to achieve robust data privacy, especially in the context of private machine learning applications. These hybrid frameworks often integrate multiple cryptographic techniques, including GC, oblivious transfer (OT), homomorphic encryption (HE), and secret sharing (SS), to balance the trade-offs between security and performance [16, 17, 39, 44, 52, 55, 68]. Comprehensive surveys have explored the effectiveness of these protocols across various scenarios, providing valuable insights for practitioners [2, 19, 53, 60]. Additionally, approaches like trusted execution environments (TEE) [21], have been proposed to enhance security in an alternative threat model settings.

While our case study focuses on private ML, the application of GC extends far beyond this domain. As a PPC protocol, GC is employed in a wide range of scenarios, including oblivious RAM [50], privacy-preserving benchmarking [13], graph analysis [38, 59], decision trees [69], medical and health applications [10, 61], web security [20], auction and election [6, 42] and private edge computing [51, 65]. One of the key strengths of GC lies in its adaptability to various private settings, including multi-party (not just two-party) scenario [3, 55, 66], post-quantum security [18], and scenarios involving malicious adversaries by integrated with additional processes (such as zero-knowledge proofs) [25, 70]. The diverse and complex application scenarios for GC highlight the ongoing need to optimize GC implementations and determine efficient GC realizations for different workloads.

8 CONCLUSION

In this paper, we present ABLE, the first real implementation to evaluate the benefits and overheads of mixed GC. We introduce a heuristic method to improve the arithmetic GC by using fewer primes, which significantly reduces communication and computation overhead. Our approach decreases communication by 14.1% and latency by 15.7% in a 16-bit arithmetic GC system. We also perform row reduction optimization to the bit-composition (BC) and bit-decomposition (BD) processes, achieving a 48.6% reduction in garbled table communication (or storage) for BD operations, and a 50% reduction for BC operations.

To determine whether an operation should be transferred to arithmetic GC, we conduct a comprehensive evaluation of various operations, considering their garbled table costs, garbling and evaluation latency, and different approaches – stream and preprocessing GC. We introduce the BC and BD equivalent ratios as tools to quickly estimate whether incorporating BC and BD gates to build the mixed circuit is worthwhile. This method changes the traditional GC realization process by enabling potential GC users to determine the suitable circuit configuration early in the development stage, before actual deployment. Finally, we demonstrate our approach with a machine learning case study, analyzing performance across different realizations. The results reveal that there is no universally the best scheme; instead, careful cost estimation at the outset allows users to select their own GC scheme for a specific application. By optimizing GC schemes, ABLE advances the secure multi-party computation and improves the efficiency in the domain of privacy-preserving computations.

REFERENCES

[1] Kahraman Akdemir, Martin Dixon, Wajdi Feghali, Patrick Fay, Vinodh Gopal, Jim Guilford, Erdinc Ozturk, Gil Wolrich, and Ronen Zohar. 2010. Breakthrough AES performance with intel AES new instructions. *White paper, June* 12 (2010), 217.

[2] Ghada Almashaqbeh and Ravital Solomon. 2022. Sok: Privacy-preserving computing in the blockchain era. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 124–139.

[3] Abdelrahaman Aly, Karl Cong, Daniele Cozzo, Marcel Keller, Emmanuela Orsini, Dragos Rotaru, Oliver Scherer, Peter Scholl, Nigel P Smart, Titouan Tanguy, et al. 2021. Scale–mamba v1. 14: Documentation. *Documentation. pdf* (2021).

[4] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. 2014. How to garble arithmetic circuits. *SIAM J. Comput.* 43, 2 (2014), 905–929.

[5] Toshinori Araki, Assi Barak, Jun Furukawa, Marcel Keller, Yehuda Lindell, Kazuma Ohara, and Hikaru Tsuchida. 2018. Generalizing the SPDZ compiler for other protocols. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 880–895.

[6] Michael Backes, Pascal Berrang, Lucjan Hanzlik, and Ivan Pryvalov. 2022. A framework for constructing single secret leader election from MPC. In *European Symposium on Research in Computer Security*. Springer, 672–691.

[7] Marshall Ball, Brent Carmer, Tal Malkin, Mike Rosulek, and Nichole Schimanski. 2019. Garbled neural networks are practical. *Cryptology ePrint Archive* (2019).

[8] Marshall Ball, Hanjun Li, Huijia Lin, and Tianren Liu. 2023. New ways to garble arithmetic circuits. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 3–34.

[9] Marshall Ball, Tal Malkin, and Mike Rosulek. 2016. Garbling gadgets for boolean and arithmetic circuits. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 565–577.

[10] Mauro Barni, Pierluigi Failla, Vladimir Kolesnikov, Riccardo Lazzeretti, Ahmad-Reza Sadeghi, and Thomas Schneider. 2009. Secure evaluation of private linear branching programs with medical applications. In *Computer Security–ESORICS 2009: 14th European Symposium on Research in Computer Security, Saint-Malo, France, September 21-23, 2009. Proceedings 14*. Springer, 424–439.

[11] Donald Beaver. 1996. Correlated pseudorandomness and the complexity of private computations. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. 479–488.

[12] Donald Beaver, Silvio Micali, and Phillip Rogaway. 1990. The round complexity of secure protocols. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*. 503–513.

[13] Kilian Becher, Martin Beck, and Thorsten Strufe. 2019. An enhanced approach to cloud-based privacy-preserving benchmarking. In *2019 International Conference on Networked Systems (NetSys)*. IEEE, 1–8.

[14] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. 2013. Efficient garbling from a fixed-key blockcipher. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 478–492.

[15] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. 2012. Foundations of garbled circuits. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 784–796.

[16] Fabian Boemer, Rosario Cammarota, Daniel Demmler, Thomas Schneider, and Hossein Yalame. 2020. MP2ML: A mixed-protocol machine learning framework for private inference. In *Proceedings of the 15th international conference on availability, reliability and security*. 1–10.

[17] Lennart Braun, Daniel Demmler, Thomas Schneider, and Oleksandr Tkachenko. 2022. Motion–a framework for mixed-protocol multi-party computation. *ACM Transactions on Privacy and Security* 25, 2 (2022), 1–35.

[18] Niklas Büscher, Daniel Demmler, Nikolaos P Karvelas, Stefan Katzenbeisser, Juliane Krämer, Deevashwer Rathee, Thomas Schneider, and Patrick Struck. 2020. Secure two-party computation in a quantum world. In *Applied Cryptography and Network Security: 18th International Conference, ACNS 2020, Rome, Italy, October 19–22, 2020, Proceedings, Part I 18*. Springer, 461–480.

[19] José Cabrero-Holgueras and Sergio Pastrana. 2021. Sok: Privacy-preserving computation techniques for deep learning. *Proceedings on Privacy Enhancing Technologies* (2021).

[20] Sofía Celi, Alex Davidson, Hamed Haddadi, Gonçalo Pestana, and Joe Rowell. 2023. Distefano: Decentralized infrastructure for sharing trusted encrypted facts and nothing more. *Cryptology ePrint Archive* (2023).

[21] Joseph I Choi, Dave Tian, Grant Hernandez, Christopher Patton, Benjamin Mood, Thomas Shrimpton, Kevin RB Butler, and Patrick Traynor. 2019. A hybrid approach to secure function evaluation using SGX. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. 100–113.

[22] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY-A framework for efficient mixed-protocol secure two-party computation.. In *NDSS*.

[23] Abdulrahman Diaa, Lucas Fenaux, Thomas Humphries, Marian Dietz, Faezeh Ebrahimianghazani, Bailey Kacsmar, Xinda Li, Nils Lukas, Rasoul Akhavan Mahdavi, Simon Oya, Ehsan Amjadian, and Florian Kerschbaum. 2024. Fast and Private Inference of Deep Neural Networks by Co-designing Activation Functions. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 2191–2208. https://www.usenix.org/conference/usenixsecurity24/presentation/diaa

[24] Ye Dong, Wen-jie Lu, Yancheng Zheng, Haoqi Wu, Derun Zhao, Jin Tan, Zhicong Huang, Cheng Hong, Tao Wei, and Wenguang Chen. 2023. Puma: Secure inference of llama-7b in five minutes. *arXiv preprint arXiv:2307.12533* (2023).

[25] Li Duan, Yufan Jiang, Yong Li, Jörn Müller-Quade, and Andy Rupp. 2022. Security Against Honorific Adversaries: Efficient MPC with Server-aided Public Verifiability. *Cryptology ePrint Archive* (2022).

[26] Xin Fang, Stratis Ioannidis, and Miriam Leeser. 2017. Secure function evaluation using an fpga overlay architecture. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 257–266.

[27] Galois, Inc. 2019. swanky: A suite of rust libraries for secure computation. https://github.com/GaloisInc/swanky.

[28] Karthik Garimella, Zahra Ghodsi, Nandan Kumar Jha, Siddharth Garg, and Brandon Reagen. 2023. Characterizing and optimizing end-to-end systems for private inference. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 89–104.

[29] Karthik Garimella, Nandan Kumar Jha, and Brandon Reagen. 2021. Sisyphus: A cautionary tale of using low-degree polynomial activations in privacy-preserving deep learning. *arXiv preprint arXiv:2107.12342* (2021).

[30] Zahra Ghodsi, Nandan Kumar Jha, Brandon Reagen, and Siddharth Garg. 2021. Circa: Stochastic relus for private deep learning. *Advances in Neural Information Processing Systems* 34 (2021), 2241–2252.

[31] Zahra Ghodsi, Akshaj Kumar Veldanda, Brandon Reagen, and Siddharth Garg. 2020. Cryptonas: Private inference on a relu budget. *Advances in Neural Information Processing Systems* 33 (2020), 16961–16971.

[32] Nir Haloani, Avishay Yanai, Meital Levy, and Yair Lavi. 2024. COTI V2: Confidential Computing Ethereum Layer 2. (2024).

[33] David Heath. 2024. Efficient arithmetic in garbled circuits. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 3–31.

[34] Siam U Hussain and Farinaz Koushanfar. 2019. FASE: FPGA acceleration of secure function evaluation. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 280–288.

[35] Nathaniel Husted, Steven Myers, Abhi Shelat, and Paul Grubbs. 2013. GPU and CPU parallelization of honest-but-curious secure two-party computation. In *Proceedings of the 29th Annual Computer Security Applications Conference*. 169–178.

[36] Intel. 2023. Intel® Core™ I7-10700K processor. https://ark.intel.com/content/www/us/en/ark/products/199335/intel-core-i710700k-processor-16m-cache-up-to-5-10-ghz.html

[37] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. 2003. Extending oblivious transfers efficiently. In *Annual International Cryptology Conference*. Springer, 145–161.

[38] Samuel Judson, Ning Luo, Timos Antonopoulos, and Ruzica Piskac. 2020. Privacy Preserving CTL Model Checking through Oblivious Graph Algorithms. In *Proceedings of the 19th Workshop on Privacy in the Electronic Society*. 101–115.

[39] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. 2018. {GAZELLE}: A low latency framework for secure neural network inference. In *27th USENIX security symposium (USENIX security 18)*. 1651–1669.

[40] Marcel Keller. 2020. MP-SPDZ: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*. 1575–1590.

[41] Vladimir Kolesnikov, Payman Mohassel, and Mike Rosulek. 2014. FleXOR: Flexible garbling for XOR gates that beats free-XOR. In *Advances in Cryptology–CRYPTO 2014: 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II 34*. Springer, 440–457.

[42] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. 2009. Improved garbled circuit building blocks and applications to auctions and computing minima. In *Cryptology and Network Security: 8th International Conference, CANS 2009, Kanazawa, Japan, December 12-14, 2009. Proceedings 8*. Springer, 1–20.

[43] Vladimir Kolesnikov and Thomas Schneider. 2008. Improved garbled circuit: Free XOR gates and applications. In *Automata, Languages and Programming: 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II 35*. Springer, 486–498.

[44] Nishant Kumar, Mayank Rathee, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. 2020. Cryptflow: Secure tensorflow inference. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 336–353.

[45] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[46] Eunsang Lee, Joon-Woo Lee, Jong-Seon No, and Young-Sik Kim. 2021. Minimax approximation of sign function by composite polynomial for homomorphic comparison. *IEEE Transactions on Dependable and Secure Computing* 19, 6 (2021), 3711–3727.

[47] Hanjun Li and Tianren Liu. 2024. How to Garble Mixed Circuits that Combine Boolean and Arithmetic Computations. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 331–360.

[48] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. 2015. Oblivm: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 359–376.

[49] Jian Liu, Mika Juuti, Yao Lu, and Nadarajah Asokan. 2017. Oblivious neural network predictions via minionn transformations. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 619–631.

[50] Steve Lu and Rafail Ostrovsky. 2013. How to garble RAM programs?. In *Advances in Cryptology–EUROCRYPT 2013: 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings 32*. Springer, 719–734.

[51] Shagufta Mehnaz and Elisa Bertino. 2020. Privacy-preserving real-time anomaly detection using edge computing. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 469–480.

[52] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. 2020. Delphi: A cryptographic inference system for neural networks. In *Proceedings of the 2020 Workshop on Privacy-Preserving Machine Learning in Practice*. 27–30.

[53] Fan Mo, Zahra Tarkhani, and Hamed Haddadi. 2024. Machine learning with confidential computing: A systematization of knowledge. *ACM computing surveys* 56, 11 (2024), 1–40.

[54] Jianqiao Mo, Jayanth Gopinath, and Brandon Reagen. 2023. Haac: A hardware-software co-design to accelerate garbled circuits. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–13.

[55] Payman Mohassel and Peter Rindal. 2018. ABY3: A mixed protocol framework for machine learning. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 35–52.

[56] Payman Mohassel and Yupeng Zhang. 2017. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE symposium on security and privacy (SP)*. IEEE, 19–38.

[57] Christian Vincent Mouchet, Jean-Philippe Bossuat, Juan Ramón Troncoso-Pastoriza, and Jean-Pierre Hubaux. 2020. Lattigo: A multiparty homomorphic encryption library in go. In *Proceedings of the 8th Workshop on Encrypted Computing and Applied Homomorphic Cryptography*. 64–70.

[58] Moni Naor, Benny Pinkas, and Reuban Sumner. 1999. Privacy preserving auctions and mechanism design. In *Proceedings of the 1st ACM Conference on Electronic Commerce*. 129–139.

[59] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. 2015. Graphsc: Parallel secure computation made easy. In *2015 IEEE symposium on security and privacy*. IEEE, 377–394.

[60] Lucien KL Ng and Sherman SM Chow. 2023. SoK: cryptographic neural-network computation. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 497–514.

[61] Adrien Oliva, Anubhav Kaphle, Roc Reguant, Letitia MF Sng, Natalie A Twine, Yuwan Malakar, Anuradha Wickramarachchi, Marcel Keller, Thilina Ranbaduge, Eva KF Chan, et al. 2024. Future-proofing genomic data and consent management: a comprehensive review of technology innovations. *GigaScience* 13 (2024), giae021.

[62] Amos R Omondi and A Benjamin Premkumar. 2007. *Residue number systems: theory and implementation*. Vol. 2. World Scientific.

[63] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. 2021. {ABY2. 0}: Improved {Mixed-Protocol} secure {Two-Party} computation. In *30th USENIX Security Symposium (USENIX Security 21)*. 2165–2182.

[64] Benny Pinkas, Thomas Schneider, Nigel P Smart, and Stephen C Williams. 2009. Secure two-party computation is practical. In *Advances in Cryptology–ASIACRYPT 2009: 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings 15*. Springer, 250–267.

[65] Erik Pohle, Aysajan Abidin, and Bart Preneel. 2024. Fast Evaluation of S-boxes with Garbled Circuits. *IEEE Transactions on Information Forensics and Security* (2024).

[66] Sai Rahul Rachuri, Ajith Suresh, and Harsh Chaudhari. 2020. Trident: Efficient 4PC Framework for Privacy Preserving Machine Learning. In *Network and Distributed System Security Symposium*. Internet Society, 1–18.

[67] Mike Rosulek and Lawrence Roy. 2021. Three halves make a whole? Beating the half-gates lower bound for garbled circuits. In *Annual International Cryptology Conference*. Springer, 94–124.

[68] Dragos Rotaru and Tim Wood. 2019. Marbled circuits: Mixing arithmetic and boolean circuits with active security. In *International Conference on Cryptology in India*. Springer, 227–249.

[69] Anselme Tueno, Florian Kerschbaum, and Stefan Katzenbeisser. 2019. Private evaluation of decision trees using sublinear cost. *Proceedings on Privacy Enhancing Technologies* (2019).

[70] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. 2021. Falcon: Honest-Majority Maliciously Secure Framework for Private Deep Learning. *Proceedings on Privacy Enhancing Technologies* 1 (2021), 188–208.

[71] Fuyi Wang, Leo Yu Zhang, Lei Pan, Shengshan Hu, and Robin Doss. 2022. Towards Privacy-Preserving Neural Architecture Search. In *2022 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 1–6.

[72] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. 2016. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit.

[73] Sophia Yakoubov. 2017. A gentle introduction to yao's garbled circuits. *preprint on webpage at https://web. mit. edu/sonka89/www/papers/2017ygc. pdf* (2017).

[74] Qing Yang, Ge Peng, Paolo Gasti, Kiran S Balagani, Yantao Li, and Gang Zhou. 2018. MEG: memory and energy efficient garbled circuit evaluation on smartphones. *IEEE Transactions on Information Forensics and Security* 14, 4 (2018), 913–922.

[75] Andrew C Yao. 1982. Protocols for secure computations. In *23rd annual symposium on foundations of computer science (sfcs 1982)*. IEEE, 160–164.

[76] Andrew Chi-Chih Yao. 1986. How to generate and exchange secrets. In *27th annual symposium on foundations of computer science (Sfcs 1986)*. IEEE, 162–167.

[77] Samee Zahur and David Evans. 2015. Obliv-C: A language for extensible data-oblivious computation. *Cryptology ePrint Archive* (2015).

[78] Samee Zahur, Mike Rosulek, and David Evans. 2015. Two halves make a whole: Reducing data transfer in garbled circuits using half gates. In *Advances in Cryptology-EUROCRYPT 2015: 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II 34*. Springer, 220–250.

[79] Shengnan Zhao, Xiangfu Song, Han Jiang, Ming Ma, Zhihua Zheng, and Qiuliang Xu. 2020. An efficient outsourced oblivious transfer extension protocol and its applications. *Security and Communication Networks* 2020, 1 (2020), 8847487.

A APPENDIX

The remaining results are listed in the appendix for reference in future mixed GC implementations.

Table 5: Inverse BC equivalent ratio: showing how many BC gates can be offset by switching one operation from Boolean to arithmetic GC, considering GC cost of different types.

Bit-width	Cost	Add	Sub	Pub Mul	Mul	Pub Exp
4	**Gb Table**	1.000[a]	1.000	1.500	6.750	24.125
	Gb Time	0.105	0.090	0.248	0.350	2.307
	Ev Time	0.151	0.132	0.393	0.808	3.826
8	**Gb Table**	0.667	0.667	1.167	10.917	36.958
	Gb Time	0.069	0.059	0.161	0.933	3.587
	Ev Time	0.090	0.080	0.234	1.550	5.046
16	**Gb Table**	0.400	0.400	0.750	14.275	47.538
	Gb Time	0.038	0.034	0.094	1.240	4.276
	Ev Time	0.052	0.047	0.136	1.903	5.991
32	**Gb Table**	0.222	0.222	0.431	16.618	54.726
	Gb Time	0.019	0.017	0.046	1.293	4.259
	Ev Time	0.028	0.026	0.073	2.137	6.633

[a] For example, regarding Garbled Table cost, the benefit of switching a Boolean Add to arithmetic can offset 1.000 BC gate.

Table 6: Inverse BD equivalent ratio: showing how many BD gates can be offset by switching one operation from Boolean to arithmetic GC, considering GC cost of different types.

Bit-width	Cost	Add	Sub	Pub Mul	Mul	Pub Exp
4	**Gb Table**	0.029[a]	0.029	0.044	0.196	0.702
	Gb Time	0.004	0.003	0.009	0.013	0.084
	Ev Time	0.005	0.004	0.013	0.027	0.126
8	**Gb Table**	0.017	0.017	0.030	0.280	0.947
	Gb Time	0.004	0.003	0.009	0.055	0.211
	Ev Time	0.005	0.004	0.013	0.084	0.274
16	**Gb Table**	0.012	0.012	0.023	0.442	1.473
	Gb Time	0.005	0.004	0.012	0.154	0.529
	Ev Time	0.006	0.006	0.016	0.225	0.708
32	**Gb Table**	0.007	0.007	0.013	0.513	1.691
	Gb Time	0.004	0.004	0.010	0.290	0.955
	Ev Time	0.005	0.005	0.014	0.420	1.303

[a] For example, regarding Garbled Table cost, the benefit of switching a Boolean Add to arithmetic can offset 0.029 BD gate.

Table 7: Arithmetic GC prime optimization for bit-width (or ring $\mathbb{Z}$) ranging from 3-bit to 127-bit. The original method selects the first k primes in the RNS system of arithmetic GC, exceeding the necessary range as shown in the Excess column. The Sum column displays the total sum of these primes. The Rm column lists the primes that can be removed through optimization, reducing the total prime sum, as shown in improvement (Imp).

$\mathbb{Z}$	k	Excess	Sum	Rm.	Excess	Sum	Imp.
2^{3}	3	3.75×	10	(3)	1.25×	7	30.0%
2^{4}	3	1.88×	10	(-)	1.88×	10	0.0%
2^{5}	4	6.56×	17	(2, 3)	1.09×	12	29.4%
2^{6}	4	3.28×	17	(3)	1.09×	14	17.6%
2^{7}	4	1.64×	17	(-)	1.64×	17	0.0%
2^{8}	5	9.02×	28	(7)	1.29×	21	25.0%
2^{9}	5	4.51×	28	(3)	1.5×	25	10.7%
2^{10}	5	2.26×	28	(2)	1.13×	26	7.1%
2^{11}	5	1.13×	28	(-)	1.13×	28	0.0%
2^{12}	6	7.33×	41	(7)	1.05×	34	17.1%
2^{13}	6	3.67×	41	(3)	1.22×	38	7.3%
2^{66}	17	26.06×	440	(23)	1.13×	417	5.2%
2^{67}	17	13.03×	440	(13)	1.×	427	3.0%
2^{68}	17	6.51×	440	(2, 3)	1.09×	435	1.1%
2^{69}	17	3.26×	440	(3)	1.09×	437	0.7%
2^{70}	17	1.63×	440	(-)	1.63×	440	0.0%
2^{71}	18	49.67×	501	(47)	1.06×	454	9.4%
2^{72}	18	24.84×	501	(23)	1.08×	478	4.6%
2^{73}	18	12.42×	501	(11)	1.13×	490	2.2%
2^{74}	18	6.21×	501	(2, 3)	1.03×	496	1.0%
2^{75}	18	3.1×	501	(3)	1.03×	498	0.6%
2^{76}	18	1.55×	501	(-)	1.55×	501	0.0%

Continued on next page

Table 7 – Continued from previous page

$\mathbb{Z}$	k	**Excess**	**Sum**	**Rm.**	**Excess**	**Sum**	**Imp.**
2^{14}	6	1.83×	41	(-)	1.83×	41	0.0%
2^{15}	7	15.58×	58	(13)	1.2×	45	22.4%
2^{16}	7	7.79×	58	(7)	1.11×	51	12.1%
2^{17}	7	3.89×	58	(3)	1.3×	55	5.2%
2^{18}	7	1.95×	58	(-)	1.95×	58	0.0%
2^{19}	8	18.5×	77	(17)	1.09×	60	22.1%
2^{20}	8	9.25×	77	(7)	1.32×	70	9.1%
2^{21}	8	4.63×	77	(3)	1.54×	74	3.9%
2^{22}	8	2.31×	77	(2)	1.16×	75	2.6%
2^{23}	8	1.16×	77	(-)	1.16×	77	0.0%
2^{24}	9	13.3×	100	(13)	1.02×	87	13.0%
2^{25}	9	6.65×	100	(2, 3)	1.11×	95	5.0%
2^{26}	9	3.32×	100	(3)	1.11×	97	3.0%
2^{27}	9	1.66×	100	(-)	1.66×	100	0.0%
2^{28}	10	24.1×	129	(23)	1.05×	106	17.8%
2^{29}	10	12.05×	129	(11)	1.1×	118	8.5%
2^{30}	10	6.03×	129	(2, 3)	1.×	124	3.9%
2^{31}	10	3.01×	129	(3)	1.×	126	2.3%
2^{32}	10	1.51×	129	(-)	1.51×	129	0.0%
2^{33}	11	23.35×	160	(23)	1.02×	137	14.4%
2^{34}	11	11.67×	160	(11)	1.06×	149	6.9%
2^{35}	11	5.84×	160	(5)	1.17×	155	3.1%
2^{36}	11	2.92×	160	(2)	1.46×	158	1.3%
2^{37}	11	1.46×	160	(-)	1.46×	160	0.0%
2^{38}	12	27.×	197	(23)	1.17×	174	11.7%
2^{39}	12	13.5×	197	(13)	1.04×	184	6.6%
2^{40}	12	6.75×	197	(2, 3)	1.12×	192	2.5%
2^{41}	12	3.37×	197	(3)	1.12×	194	1.5%
2^{42}	12	1.69×	197	(-)	1.69×	197	0.0%
2^{43}	13	34.59×	238	(31)	1.12×	207	13.0%
2^{44}	13	17.29×	238	(17)	1.02×	221	7.1%
2^{45}	13	8.65×	238	(7)	1.24×	231	2.9%
2^{46}	13	4.32×	238	(3)	1.44×	235	1.3%
2^{47}	13	2.16×	238	(2)	1.08×	236	0.8%
2^{48}	13	1.08×	238	(-)	1.08×	238	0.0%
2^{49}	14	23.24×	281	(23)	1.01×	258	8.2%
2^{50}	14	11.62×	281	(11)	1.06×	270	3.9%
2^{51}	14	5.81×	281	(5)	1.16×	276	1.8%
2^{52}	14	2.9×	281	(2)	1.45×	279	0.7%
2^{53}	14	1.45×	281	(-)	1.45×	281	0.0%
2^{54}	15	34.13×	328	(31)	1.1×	297	9.5%
2^{55}	15	17.07×	328	(17)	1.×	311	5.2%
2^{56}	15	8.53×	328	(7)	1.22×	321	2.1%
2^{57}	15	4.27×	328	(3)	1.42×	325	0.9%
2^{58}	15	2.13×	328	(2)	1.07×	326	0.6%
2^{59}	15	1.07×	328	(-)	1.07×	328	0.0%
2^{60}	16	28.27×	381	(23)	1.23×	358	6.0%
2^{61}	16	14.13×	381	(13)	1.09×	368	3.4%
2^{62}	16	7.07×	381	(7)	1.01×	374	1.8%
2^{63}	16	3.53×	381	(3)	1.18×	378	0.8%
2^{64}	16	1.77×	381	(-)	1.77×	381	0.0%
2^{65}	17	52.12×	440	(47)	1.11×	393	10.7%
2^{77}	19	52.×	568	(47)	1.11×	521	8.3%
2^{78}	19	26.×	568	(23)	1.13×	545	4.0%
2^{79}	19	13.×	568	(13)	1.×	555	2.3%
2^{80}	19	6.5×	568	(2, 3)	1.08×	563	0.9%
2^{81}	19	3.25×	568	(3)	1.08×	565	0.5%
2^{82}	19	1.63×	568	(-)	1.63×	568	0.0%
2^{83}	20	57.69×	639	(53)	1.09×	586	8.3%
2^{84}	20	28.84×	639	(23)	1.25×	616	3.6%
2^{85}	20	14.42×	639	(13)	1.11×	626	2.0%
2^{86}	20	7.21×	639	(7)	1.03×	632	1.1%
2^{87}	20	3.61×	639	(3)	1.2×	636	0.5%
2^{88}	20	1.8×	639	(-)	1.8×	639	0.0%
2^{89}	21	65.8×	712	(61)	1.08×	651	8.6%
2^{90}	21	32.9×	712	(31)	1.06×	681	4.4%
2^{91}	21	16.45×	712	(13)	1.27×	699	1.8%
2^{92}	21	8.23×	712	(7)	1.18×	705	1.0%
2^{93}	21	4.11×	712	(3)	1.37×	709	0.4%
2^{94}	21	2.06×	712	(2)	1.03×	710	0.3%
2^{95}	21	1.03×	712	(-)	1.03×	712	0.0%
2^{96}	22	40.61×	791	(37)	1.1×	754	4.7%
2^{97}	22	20.31×	791	(19)	1.07×	772	2.4%
2^{98}	22	10.15×	791	(7)	1.45×	784	0.9%
2^{99}	22	5.08×	791	(5)	1.02×	786	0.6%
2^{100}	22	2.54×	791	(2)	1.27×	789	0.3%
2^{101}	22	1.27×	791	(-)	1.27×	791	0.0%
2^{102}	23	52.67×	874	(47)	1.12×	827	5.4%
2^{103}	23	26.33×	874	(23)	1.14×	851	2.6%
2^{104}	23	13.17×	874	(13)	1.01×	861	1.5%
2^{105}	23	6.58×	874	(2, 3)	1.1×	869	0.6%
2^{106}	23	3.29×	874	(3)	1.1×	871	0.3%
2^{107}	23	1.65×	874	(-)	1.65×	874	0.0%
2^{108}	24	73.24×	963	(73)	1.×	890	7.6%
2^{109}	24	36.62×	963	(31)	1.18×	932	3.2%
2^{110}	24	18.31×	963	(17)	1.08×	946	1.8%
2^{111}	24	9.16×	963	(7)	1.31×	956	0.7%
2^{112}	24	4.58×	963	(3)	1.53×	960	0.3%
2^{113}	24	2.29×	963	(2)	1.14×	961	0.2%
2^{114}	24	1.14×	963	(-)	1.14×	963	0.0%
2^{115}	25	55.5×	1060	(53)	1.05×	1007	5.0%
2^{116}	25	27.75×	1060	(23)	1.21×	1037	2.2%
2^{117}	25	13.88×	1060	(13)	1.07×	1047	1.2%
2^{118}	25	6.94×	1060	(2, 3)	1.16×	1055	0.5%
2^{119}	25	3.47×	1060	(3)	1.16×	1057	0.3%
2^{120}	25	1.73×	1060	(-)	1.73×	1060	0.0%
2^{121}	26	87.59×	1161	(83)	1.06×	1078	7.1%
2^{122}	26	43.8×	1161	(43)	1.02×	1118	3.7%
2^{123}	26	21.9×	1161	(19)	1.15×	1142	1.6%
2^{124}	26	10.95×	1161	(7)	1.56×	1154	0.6%
2^{125}	26	5.47×	1161	(5)	1.09×	1156	0.4%
2^{126}	26	2.74×	1161	(2)	1.37×	1159	0.2%
2^{127}	26	1.37×	1161	(-)	1.37×	1161	0.0%