Garbled Neural Networks are Practical

Marshall Ball[*] Brent Carmer[†] Tal Malkin[‡] Mike Rosulek[§]
Nichole Schimanski[¶]

June 24, 2019

Abstract

We show that garbled circuits offer a practical choice for secure evaluation of neural network classifiers, comparable with complex, specialized protocols using less robust assumptions, many rounds of interaction, and/or tailor-made neural networks. In particular, we develop a scheme for garbling "off the shelf" pre-trained neural networks, where the only model preprocessing required is a mild discretization step as opposed to requiring a specialized SFE-friendly model to be independently trained. Moreover, as our solution is a garbling scheme, it inherits a much more diverse range of applications than non-garbling-based solutions, perhaps most notably, efficient compilers for the malicious setting.

At the protocol level, we start with the garbling scheme of Ball, Malkin & Rosulek (ACM CCS 2016) for arithmetic circuits and introduce new optimizations for modern neural network activation functions. We develop fancy-garbling[1], the first implementation of the BMR16 garbling scheme along with our new optimizations, as part of heavily optimized garbled-circuits tool that is driven by a TensorFlow classifier description.

We evaluate our constructions on a wide range of neural networks. We find that our approach is up to 100× more efficient than straight-forward boolean garbling. It is also roughly 40% more efficient than DeepSecure (Rouhani et al., DAC 2018), a recent garbled-circuit-based approach for secure neural network evaluation, which incorporates significant optimization techniques for boolean circuits. Furthermore, our approach provides competitive performance tradeoffs (efficiency and latency vs. communication) also when compared with non-garbled-circuit approaches.

1 Introduction

Consider Alice, who holds a neural network she has trained, and Bob, who holds an input he wants to know the prediction of the neural network on. Both parties prefer to keep their inputs private, revealing only the output of the evaluation. We refer to this problem as *secure neural network inference*, or more generally *secure classification*.

Secure classification is an important ingredient in many emerging applications of secure multiparty computation (MPC). For example, one might wish to securely identify "similar" items in two private data sets of unstructured data (*e.g.*, images), where similarity is determined by a neural network. With machine learning as a service, a cloud holds a store of private data, and secure

[*] Columbia University. marshall@cs.columbia.edu

[†] Galois, Inc. brent@galois.com

[‡] Columbia University. tal@cs.columbia.edu

[§] Oregon State University. rosulekm@eecs.oregonstate.edu

[¶] Galois, Inc. nls@galois.com

[1] https://github.com/GaloisInc/fancy-garbling

classification can be used to identify a subset of records that match a particular condition. For example, one may compute statistics on the metadata (date & geolocation) of all images of a particular subject.

Existing Approaches. The question of secure classification has been studied in several recent works for various types of classifiers, including neural networks, which are the focus of our paper. Existing works, described in more detail below, are each based on one or more of the following cryptographic techniques: homomorphic encryption, secret-sharing-based secure computation, and Yao's garbled circuits. They provide different features and tradeoffs in terms of number of parties, number of rounds, computation, communication, different levels of privacy for the classifier, what types and sizes of neural networks they can practically support, what accuracy, and whether they require special re-training of these neural networks, or can start from any standard trained one.

The common wisdom underlying most of these previous works is that garbled circuits (GCs) are too cumbersome and impractical to be the main tool for secure neural network inference, due to their boolean nature. Indeed, each layer of a neural network has a large linear component (over arithmetic values), where homomorphic encryption or secret-sharing techniques are very fast (addition and multiplication by a constant are extremely efficient and require no communication at all with these methods). In contrast, conversion of these linear operations to a boolean circuit is expensive, and entails creating and communicating a number of ciphertexts proportional to the number of resulting conjunction gates. On the other hand, each layer also has a non-linear component (based on comparison) such as sign or ReLU (Rectified Linear Unit), which would add to the multiplicative degree of the computation, and result in a very high computational overhead for fully homomorphic encryption (FHE), and add rounds of interaction for secret-sharing based secure computation. For these types of non-linear operations, the thinking goes, garbled circuits would be more appropriate and efficient. With this in mind, previous works choose one of the approaches, or combine several, towards a practical system.

Our Approach. We depart from the above narrative by showing that pure GC techniques are in fact practical for neural networks. Our starting point is the garbling scheme of Ball, Malkin, and Rosulek (BMR) [BMR16], which supports a certain class of *arithmetic* circuits, and turns out to be extremely efficient for linear operations (over bounded integers), but not for non-linear ones such as comparison.[2] We improve the BMR scheme, develop new garbling techniques, and optimize them in ways motivated by neural network (NN) applications, but which are more widely applicable. We also develop the first implementation of BMR (including our new improvements) as part of a neural-network garbling tool that is driven by standard TensorFlow model files.

Because our approach works entirely within the garbled circuits realm, it leads to secure classification of NNs with the same well-known benefits and drawbacks as garbled-circuit based MPC. Specifically:

- Round complexity: GC-based and FHE-based protocols are constant-round, whereas other approaches (like arithmetic MPC and hybrid approaches that switch paradigms) generally require one or more rounds of synchronous interaction for each layer of the neural network. As deep neural networks become more prevalant (*e.g.*, ResNet NN architectures for image classification can be as deep as 150+ layers [HZRS16]), round complexity can become a

[2] BMR also provide a host of optimizations for *boolean* garbling, such as threshold garbling gadgets, which may yield improvements for binary neural networks. Here we focus on the general (arithmetic) setting.

significant bottleneck for secure classification.[3]

- Communication complexity: Garbled circuits require sending a number of ciphertexts that is proportional to the size of the circuit. In contrast, FHE requires very small communication, proportional to the security parameter, and independent of the circuit size (while paying in high computation complexity). However, for many neural networks considered here and in other works, we are able to obtain garbled circuits whose size is quite small.

- Malicious security compatibility: While the focus of our work (and almost all work in this area) is security against *semi-honest* adversaries, it is important to note that not all protocol paradigms have a clear path towards supporting malicious security. Protocols based purely on garbled circuits or purely on arithmetic generic MPC have well-known ways of being promoted to a malicious-secure MPC. Protocols based on FHE or that switch protocol paradigms do not have such well-established transformations to malicious security.

- Support for functionality variants: garbled circuits can be used not only for NN evaluation on private inputs, but also as a black box in other applications like zero-knowledge proofs (*e.g.*, [JKO13]).

Clearly a pure GC approach may not be suitable for all applications of secure classification. However, the approach is practical and it contributes to the design space a combination of benefits that is not enjoyed by any other approach. We offer this work as a challenge to the conventional wisdom that GC is impractical for neural networks.

Discussion: Privacy of the Model. While the motivation to keep the privacy of the NN (or any machine learning model) is clear, we note that it is often not a realistic expectation, since models are inherently learnable, even via black box queries and no other information about the model. Indeed, model extraction (or "model stealing") attacks on popular ML services are well known [TZJ$^+$16, PMG$^+$17].

Putting aside the feasibility of genuine privacy for the classifier, there are many reasonable scenarios where the only privacy concern is hiding the input-to-be-classified. For example, consider a client who is outsourcing a classification task (with respect to a public classifier) to the cloud, because downloading and running the classifier locally is prohibitive. Or consider a public classifier that needs to be applied on data that is secret-shared among several clients. Or, after committing to an image and making a qualitative claim about its contents (*e.g.*, this is an image of the Statue of Liberty), being challenged to prove that a public classifier agrees with that claim.

For these reasons, in our experiments we consider the setting of public neural networks, in addition to the setting where we keep the weights secret.

1.1 Previous Work

We overview the works that are most relevant to our paper: those designing two-party protocols for secure evaluation of neural networks (we design our experiments to compare against these works). We note that there are numerous other works applying secure computation and homomorphic encryption techniques to machine learning tasks in order to securely evaluate other classifiers such as linear classifiers and decision trees (cf. [AEM08, OPJM10, NWI$^+$13, BLN14, BPTG15]). There

[3] We note that our implementation of garbling operates in a streaming fashion, so even very large networks / circuits do not need to be resident in memory all at once.

are also some works that discuss secure *training* of ML model (see SecureML [MZ17a] and references within), while here we focus on secure *evaluation* of an already-trained NN.

Early evaluation of neural networks often used mixed protocols, taking advantage of the cheap linear operations in homomorphic encryption, while using garbled circuits only for nonlinear activation functions [CL05, BOP06, OPB07, BFL+11]. Recent works continue this theme of mixing protocols, handling different kinds of neural networks often in a modified manner that sacrifices some accuracy and functionality for more efficient secure versions. For instance, SecureML [MZ17a] introduces activation functions optimized for MPC using garbled circuits, switching to FHE for the linear operations. Another mixed protocol is MiniONN [LJLA17], which uses GMW [GMW87] (secret-sharing based) secure computation, together with additive homomorphic encryption in a preprocessing stage. Chameleon [RWT+18] uses GMW mixed with garbled circuits for activation functions, with an assumption of a third party dealing correlated randomness in a preprocessing stage. HyCC [BDK+18] provides an automatic compilation of programs to hybrid secure computation protocols, and in particular an automatically found secure protocol for some NNs. Finally, Gazelle [JVC18] demonstrates novel techniques in the FHE part of their protocol. They take advantage of the packed SIMD ciphertexts in certain FHE schemes, achieving a “best-of-both-worlds” efficiency, with the best performance to date. Despite such promising performance, all mixed protocols require a linear number of rounds in depth of the NN.

Sadeghi and Schneider [SS09] presented early work based purely on (Yao) garbled circuits, thus requiring constant communication rounds. A recent protocol that is purely GC based is DeepSecure [RRK18]. Our work follows in this line. DeepSecure has several optimizations which are potentially applicable. First, they preprocess the neural network itself, which requires special retraining. Next, they prune neurons from the network whose weights are below a certain threshold. Finally, they write the neural network in Verilog and use hardware synthesis tools to optimize (following TinyGarble [SHS+15]). The first two optimizations could also apply to our implementation, although we did not implement them in our experiments. Despite this, we outperform DeepSecure by an order of magnitude (see Table 3 for details). The last DeepSecure optimization does not apply to this work, as we use arithmetic rather than boolean garbled circuits. We note that in addition to our comparisons to the results presented DeepSecure, we have also implemented baseline boolean garbling with no optimizations.

Schemes based purely on FHE achieve a single round of communication. These include CryptoNETS [GBDL+16] which uses leveled homomorphic encryption, and replaces the activation function with squaring. More recently, Bourse et. al. [BMMP18] provided an improved FHE-based construction, using the sign activation function. These protocols are limited in the depth of the neural network by the growth of noise in the FHE ciphertexts.

Concurrent to and independent of the present work, XONN also utilizes a purely GC-based approach for constant round evaluation [RSC+19]. However, their optimizations are for the particular case that the inference model is a binary neural network (where multiplicative weights are restricted to $\{\pm 1\}$ and activation is simply the sign function). In contrast, our techniques are for the more general case of arbitrarily discretized neural networks.

We note that the technical contributions in XONN and in our work focus on mostly orthogonal aspects (e.g., machine learning techniques in their work and arithmetic garbling techniques in our work). Thus, it may be possible to combine ideas from both works in a single system to yield significant performance benefits (albeit requiring a specialized NN). This is an interesting avenue for future work.

1.2 Our Contributions

New garbled circuit techniques for neural networks. We extend the Ball, Malkin, & Rosulek (BMR) garbling technique, which we review in Section 3. Very roughly speaking, the BMR scheme supports free addition and multiplication-by-constant (over the integers), for bounded integers $\{-B, \ldots, B\}$.

We present several new improvements to the BMR garbling scheme that were motivated by NN applications, but are generally applicable:

- Improved **garbling of the sign function** (i.e., $\mathsf{sgn}(x) = 0$ if $x < 0$ and $\mathsf{sgn}(x) = 1$ otherwise) for integers in the representation used by BMR. For example, our sign computation is 15% cheaper in ciphertext size than BMR for 24-bit values. As part of the sign function, we introduce an improved technique for garbled addition of numbers represented in mixed-radix number systems.

- We show how to garble an **approximate sign** function $\widetilde{\mathsf{sgn}}$ which agrees with sgn only on, say, 99.9% of inputs, but costs significantly less than the exact sgn function. The correctness parameter is tunable and provides a tradeoff with communication cost. For example, allowing 0.01% error for sgn of 16-bit numbers leads to a 65% cost reduction.

- We give an improved technique for **garbled multiplication of an integer by a bit** in the BMR representation, which is roughly 50% cheaper than in BMR. In turn, this can then be used for more common activation functions. In particular, the very common **ReLU activation function** can be written as $\mathsf{relu}(x) = \mathsf{sgn}(x) \cdot x$, which is the product of a bit and an integer. Further, convolutional NNs often include a **max-pooling** layer, and a max can also be written as a combination of free additions/subtractions and our improved components: $\mathsf{max}(x, y) = y + \mathsf{sgn}(x - y) \cdot (x - y)$.

Experiments show that our approximate ReLU and max-pooling lead to minimal effect on the overall NN accuracy while also reducing the cost significantly. For example, in our experiments (cf. Table 5), using a 99% correct ReLU only decreases the classifier accuracy by 2.7% but reduces communication cost by 59%, relative to exact ReLU. We found that approximate sign had less of an impact on overall classifier accuracy than approximate ReLU, but we mostly used ReLU so as to match the experiments of related work as closely as possible.

Because our approach supports standard NN components like ReLU, sign, and max pooling, we are able to support "off-the-shelf" use of classifiers after just a simple discretization step, without having to resort to ad-hoc non-standard NN techniques. Indeed, our NN implementation is directly configured by a TensorFlow model.

Library for garbling neural networks. We developed a library for garbling neural networks, containing the first implementation of the BMR garbling scheme that we are aware of, as well as our new improvements.

Our implementation supports two different privacy modes. In both we assume that the topology of the neural network is known to all parties. Assuming all activations are ReLU or sign, the only potentially private aspect of the NN model is the weights and biases of its neurons.

- **Private Weights.** Here the weights are also private, known only to the garbler. The garbler can "bake them into" the garbled circuit in a way that still hides them from the evaluator.

- **Public Weights.** Here we assume the weights are public and known to all parties. This results in linear operations, something BMR is very good at, and is where we see our best performance.

In addition to the new garbling techniques discussed above, our library implements many engineering-level optimizations for BMR-style garbling. It supports *streaming*, where all circuits (*i.e.*, the NN model and garbled circuit) are processed as a stream. That way, the large garbled circuits do not need to be resident in memory at one time, and the resulting MPC protocol does not require the receiver to wait for receipt of the entire garbled circuit. Finally, our implementation is driven by standard TensorFlow model formats. Details about the implementation are given in Section 7.

Performance. We evaluate our system on a wide variety of neural network classifiers that have been used as benchmarks in other works. As expected, due to our pure garbled-circuit approach, our communication costs are generally higher than other approaches, while our running time is often significantly faster (especially for deeper neural networks). We provide a full comparison in Section 8.

2 Preliminaries

2.1 Mixed-radix Number Systems

We use $\mathsf{MRS}[d_1, \ldots, d_n]$ to refer to the **mixed-radix system**, in which numbers are represented as tuples from $\mathbb{Z}_{d_1} \times \cdots \times \mathbb{Z}_{d_n}$. These representations are associated with the integers $\{0, \ldots, (\prod_i d_i) - 1\}$ in lexicographic order.

Prominent examples include:

- $\mathsf{MRS}[2, \ldots, 2]$, with k terms, are the k-bit binary numbers.
- $\mathsf{MRS}[10, \ldots, 10]$, with k terms, are the k-digit decimal numbers.
- $\mathsf{MRS}[2, 3, 5, 7, 11, \ldots]$, where the terms are the first k primes, corresponds to a **primorial mixed-radix (PMR)** system that is used in BMR garbling [BMR16].

In this work we will consider mixed-radix systems with arbitrary digit bases.

Addition in a mixed-radix system is done using the grade-school algorithm: the rightmost digits are added, with the overflow carrying into the penultimate digit, etc. We consider addition with no final carry-out, corresponding to addition modulo $\prod_i d_i$.

3 BMR Garbling

The BMR garbling scheme supports the following class of circuits that they call **mixed-modulus circuits**:

- Every wire has a designated modulus m, and values on that wire are elements of $\mathbb{Z}_m$. We call such a wire a $\mathbb{Z}_m$**-wire**.
- Unary gates are allowed for any function $g : \mathbb{Z}_m \to \mathbb{Z}_\ell$ (note that the input/output wires may have different moduli).
- Addition-mod-m gates gates are allowed if all input/output wires have the same modulus.

- Unary multiplication-mod-m-by-constant-c gates are allowed, if the input and output wires are both $\mathbb{Z}_m$-wires and $\gcd(c, m) = 1$. Note that c is a public constant (i.e., part of the circuit description).

These kinds of circuits can be garbled at the following costs:

Theorem 1 ([BMR16]). *Assuming the existence of a mixed-modulus circular correlation robust hash function [BMR16][Definition 1] (alternatively the random oracle model), then there is a* ***garbling scheme*** *(as defined in [BHR12]) for mixed-modulus circuits, whose costs in the number of ciphertexts are as follows:*

- *Unary gates $g : \mathbb{Z}_m \to \mathbb{Z}_\ell$ cost $m - 1$ ciphertexts,*
- *Addition-mod-m gates, and multiplication-by-constant gates are free.*

Note that "multiplication-by-zero" gates can also be garbled for free by including a global "constant zero" wire in the circuit (one wire globally for each modulus). Then, whenever m is prime, we can consider any multiplication-by-constant-mod-m gate to be free, even when the constant is zero.

CRT Terminology. Starting from these basic building blocks, BMR applies the Chinese remainder theorem to construct *gadgets* for garbling higher-level operations. We use these concepts and notation extensively in our results as well.

- Let $p_1, p_2, \ldots$ denote the primes, in ascending order. $[x]_p$ denotes the residue of x in $\mathbb{Z}_p$.
- Let P_k be the product of the first k primes (i.e., the kth primordial). Hence $\mathbb{Z}_{P_k} \cong \mathbb{Z}_{p_1} \times \mathbb{Z}_{p_2} \times \cdots \times \mathbb{Z}_{p_k}$ by the CRT. We will always use k to denote the number of primes.
- When k is understood, we write $[\![x]\!]_{\mathsf{crt}}$ to denote the **CRT residue representation** of x, that is: $[\![x]\!]_{\mathsf{crt}} = (x_1, \ldots, x_k)$ where $x_i = [x]_{p_i}$.

The high-level idea of BMR is to compute arithmetic in $\mathbb{Z}_{P_k}$ by representing each logical value in the circuit as its CRT residue representation. That is, the circuit contains a "bundle" of wires with moduli $p_1, \ldots, p_k$, where the ith wire in the bundle carries $[x]_{p_i}$. Addition and multiplication mod P_k reduce to the corresponding operations mod p_i, by the CRT.

4 New Garbling Technique: Cross-Modulus Multiplication

Our improvements to the BMR garbling scheme can be split into two categories: improved garbling techniques (*i.e.*, new cryptographic constructions) for mixed-modulus circuit gates, and improved methods of expressing high-level operations (*e.g.*, neural network activation functions) as mixed-modulus circuits.

In this section, we focus on the former category of improvements. We show how to efficiently garble a multiplication $x \cdot y$ where x and y are wires with different moduli (for example, x is a bit). Later, in Sections 5 and 6 we show improved ways to express mixed-radix addition and approximate sign as mixed-modulus circuits.

4.1 Half-Gate Generalization

Zahur et al. [ZRE15] show how to garble a $\mathbb{Z}_2$-multiplication gate (namely logical AND) while supporting free-XOR (free addition in $\mathbb{Z}_2$) at a cost of two ciphertexts. In unpublished work of Malkin et al. [MPs16], this was generalized to the $\mathbb{Z}_p$ case. Their construction supports addition for free, and multiplication for a cost of $2p - 2$ ciphertexts. We summarize the construction here.

First, wire labels in the scheme are elements of $\mathbb{Z}_p^n$. On any wire, the wire label encoding $a \in \mathbb{Z}_p$ has the form $A + a\Delta$, where A and Δ are vectors, and Δ is common to all wires in the circuit. Free addition mod p is done by simply adding wire labels (component-wise mod p), so $(A + a\Delta) + (B + b\Delta) = (A + B) + (a + b)\Delta$.

The last component of the wire labels is used as a special "color digit." Suppose the zero-label A for some wire has value $r \in \mathbb{Z}_p$. If we ensure that the last component of Δ is one, then the wire label corresponding to value $a \in \mathbb{Z}_p$ has least significant digit $a + r \pmod{p}$. We call this least significant digit the "color digit" of the label, and the evaluator's behavior can depend on it.

The main idea of the half-gates construction is to write a multiplication gate as:

$$x \cdot y = x \cdot (y + r - r) = x \cdot (y + r) - x \cdot r.$$

If we take r to be the color-digit of the zero-label (on the y-wire), then the garbler knows r at garbling time, and the evaluator will know $y + r$ at evaluation time. Thus, the first term $x \cdot (y + r)$ is a multiplication between an unknown value x and a value known to the evaluator in the clear; and the second term $x \cdot r$ is a multiplication between an unknown value and a value known to the garbler at garbling time. The construction works by garbling each individual term using $p - 1$ ciphertexts; then the subtraction (mod p) is free.

Renaming the variables, consider the multiplication $a \cdot b$, when a is known to the garbler. The garbler simply constructs a unary gate $v \mapsto a \cdot v$ and garbles it in the standard way using the BMR construction.

Now consider the multiplication $a \cdot b$, when a is known to the evaluator. For every $\widetilde{a}$, the garbler uses $A + \widetilde{a}\Delta$ to encrypt the value $C - \widetilde{a}B$, where C is the zero-label of the output wire. At evaluation time, the evaluator will hold wire labels $A + a\Delta$ and $B + b\Delta$ (where a is known but b is unknown), so he can open the appropriate ciphertext to learn $C - aB$, and then compute

$$(C - aB) + a(B + b\Delta) = C + ab\Delta,$$

This is the wire label encoding ab on the output wire. Note that the evaluator must know a in the clear to perform $a(B + b\Delta)$. This approach costs p ciphertexts, but can be reduced to $p - 1$ ciphertexts with a standard row-reduction trick (choosing $C = H(A)$).

4.2 New Mixed-Modulus Half-Gate

We show how this approach can be generalized to multiplication across two different moduli. Suppose $x \in \mathbb{Z}_p$ and $y \in \mathbb{Z}_q$, where $p > q$, and we wish to compute $xy \pmod{p}$ in the circuit. For example, one way of computing a ReLU activation function is via $\mathsf{relu}(x) = x \cdot \mathsf{sgn}(x)$. Since the output of $\mathsf{sgn}(x)$ is a bit, we must multiply a $\mathbb{Z}_2$ value by a $\mathbb{Z}_p$ value.

Naively, this can be done first with a unary gate that "casts" $y \in \mathbb{Z}_q$ to $\mathbb{Z}_p$, and then a $\mathbb{Z}_p$ multiplication. Overall the cost to garble such operations is $(q-1)+(2p-2)$ ciphertexts. We show how the same operation can be done for roughly $q + p - 1$ ciphertexts.

Suppose the wire labels for x have the form $X + x\Delta_p$, and wire labels for y have the form $Y + y\Delta_q$. The naive method would apply a unary gate to the y-value, and its outputs have wire labels of the form $\widetilde{Y} + y\Delta_p$ (note the change to Δ_p). Now suppose we want to do a $\mathbb{Z}_p$ multiplication using the generalized half-gates construction.

Where will the $\widetilde{Y}$-labels be used? The half-gates trick treats the two wires in fundamentally different ways, and we can arrange for the $\widetilde{Y}$-labels to be used only in the "evaluator-half-gate." Here the evaluator uses $\widetilde{Y} + y\Delta_p$ to decrypt some value (which it will add to the X-label) and also uses the color bit of this wire label to do a scalar multiplication of the X-label. Instead of encrypting the relevant value from the half gate with the $\widetilde{Y}$-label, we encrypt it with the corresponding Y-label. Intuitively, the evaluator learns one if and only if he learns the other. This allows us to do away with the unary gate that converts Y-labels to $\widetilde{Y}$-labels. Furthermore, since we know there are only q values that the evaluator can have, we can encrypt this half gate with $q-1$ instead of $p-1$ ciphertexts. However, we still need to convey the $\mathbb{Z}_p$ color digit of the $\widetilde{Y}$-label; the $\mathbb{Z}_q$ color digit of the Y-label won't do.

To solve this, we replace one of the half gates ($p-1$ ciphertexts) with a truncated one that uses the $\mathbb{Z}_q$-labels as keys ($q-1$ ciphertexts), and we replace the unary gate ($q-1$ ciphertexts that encrypt entire wire labels) with $q-1$ encryptions of very short color digits. In the regimes considered in this paper, all of these color-digit ciphertexts can be packed into 128 bits, so they are equivalent in size to one "usual" ciphertext. Hence the total cost is that of only $p+q-1$ ciphertexts. In the next section, we show full detail and give proofs of our new construction.

4.3 Details & Security

Because we make only a small modification to the BMR garbling scheme (adding support for a new kind of gate), we present only the major differences. The bulk of the scheme and the security proof remain unchanged.

Notation. The notation used in the BMR garbling algorithm is as follows. Each wire i has an associated output wire label W_i^0 which represents the logical value zero on that wire. Each wire i has an associated modulus $m = i.\mathsf{domain}$, meaning that the wire carries logical values from $\mathbb{Z}_m$. Value $v \in \mathbb{Z}_m$ on the wire will be encoded by wire label $W_i^0 + v\Delta_m$, where the operation is componentwise addition modulo m and Δ_m is a global value common to all wires of this modulus. The wire labels on wire i will be interpreted as a vector of $\mathbb{Z}_m$ elements. The rightmost component of a wire-label (vector) W, which we write as $\tau(W)$, is used as a "point-and-permute digit" (which is visible to the evaluator). We assume $\tau(\Delta_m) = 1$ so that $\tau(W_i^0 + v\Delta_m) = \tau(W_i^0) + v$ (all operations mod m).

Every gate g is identified with its output wire, but also has a set of inputs. Our focus is on fan-in-2 multiplication gates, where we write $(a, b) = g.\mathsf{inputs}$ to denote that wires a and b are the input wires of gate g.

Generalized half-gates. Before showing the construction for mixed-modulus half-gates, we first describe the generalized half-gates multiplication for two wires with the same modulus.

```
Garbling a multiplication gate $g$:
  $(a, b) = g.\mathsf{inputs}$
  $p = g.\mathsf{domain}$ (must also be $a.\mathsf{domain}$ and $b.\mathsf{domain}$)
  $\tau_a = \tau(W_a^0)$
  $\tau_b = \tau(W_b^0)$

  ▷ garbler's half-gate
  $U \leftarrow$ random wire label
  for $i = 0$ to $p - 1$:
    $G_{1,i+\tau_a} = H(g; W_a^0 + i\Delta_p) + U + i\tau_b\Delta_p$

  ▷ evaluator's half-gate
  $V \leftarrow$ random wire label
  for $j = 0$ to $p - 1$:
    $G_{2,\tau_b+j} = H(g; W_b^0 + j\Delta_p) + V - (\tau_b + j)W_a^0$

  $W_g^0 = -U + V$
  the garbled gate is $G_{0,1}, \ldots, G_{1,p-1}, G_{2,0}, \ldots, G_{2,p-1}$
```

When evaluating, the evaulator has a wire label W_i^* for every wire i, which encodes an unknown value x as $W_i^* = W_i^0 + x\Delta_m$.

```
Evaluating a multiplication gate $g$:
  $(a, b) = g.\mathsf{inputs}$
  $p = g.\mathsf{domain}$ (must also be $a.\mathsf{domain}$ and $b.\mathsf{domain}$)
  $\tau_a^* = \tau(W_a^*)$
  $\tau_b^* = \tau(W_b^*)$

  ▷ garbler's half-gate
  $U^* = G_{1,\tau_a^*} - H(g; W_a^*)$

  ▷ evaluator's half-gate
  $V^* = \tau_b^* \cdot W_a^* + G_{2,\tau_b^*} - H(g; W_b^*)$

  output wire label is $-U^* + V^*$
```

Correctness follows by the following observations. Suppose the values on the input wires are x, y, so $W_a^* = W_a^0 + x\Delta_p$ and $W_b^* = W_b^0 + y\Delta_p$. Then $\tau(W_a^*) = \tau_a + x$ and $\tau(W_b^*) = \tau_b + y$. So:

$$\begin{aligned}
-U^* + V^* &= -\Big(G_{1,\tau_a^*} - H(g; W_a^*)\Big) + \Big(\tau_b^* \cdot W_a^* + G_{2,\tau_b^*} - H(g; W_b^*)\Big) \\
&= -\Big(G_{1,\tau_a+x} - H(g; W_a^0 + x\Delta_p)\Big) \\
&\qquad + \Big(\tau_b^* \cdot W_a^* + G_{2,\tau_b+y} - H(g; W_b^0 + y\Delta_p)\Big) \\
&= -(U + x\tau_b\Delta_p) + \tau_b^*(W_a^0 + x\Delta_p) + (V - \underbrace{(\tau_b + y)}_{\tau_b^*} W_a^0) \\
&= -(U + x\tau_b\Delta_p) + \tau_b^* x\Delta_p + V \\
&= -U + V + (-x\tau_b + x(\tau_b + y))\Delta_p \\
&= -U + V + xy\Delta_p \\
&= W_g^0 + xy\Delta_p
\end{aligned}$$

This is a multiplication gate with $2p$ ciphertexts. This can be further reduced by choosing U and V so that garbled gate ciphertexts $G_{1,0}$ and $G_{2,0}$ are all zeroes, rather than choosing U and V randomly. That way, those two ciphertexts do not need to be sent. This is a standard trick in garbled circuits that we omit, since it clutters the notation.

Garbling a cross-modulus multiplication gate. Now suppose the input wire a has modulus $a.\mathsf{domain} = p$ while input wire b has $b.\mathsf{domain} = q < p$. Following the discussion in Section 4, we imagine a "virtual wire" in which the value on the b-wire has been "promoted" to $\mathbb{Z}_p$ from $\mathbb{Z}_q$. We do not need wire labels on this wire, but only $\mathbb{Z}_p$ color digits.

```
Garbling a multiplication gate $g$:
  $(a, b) = g.\mathsf{inputs}$
  $p = a.\mathsf{domain}$
  $q = b.\mathsf{domain}$ $(q < p)$
  $\tau_a = \tau(W_a^0)$
  $\tau_b = \tau(W_b^0)$

  ▷ garbler's half-gate
  $U \leftarrow$ random wire label
  for $i = 0$ to $p - 1$:
    $G_{1,i+\tau_a} = H(g; W_a^0 + i\Delta_p) + U + i\tau_b\Delta_p$

  ▷ evaluator's half-gate
  $\boxed{\tilde{\tau} \leftarrow \mathbb{Z}_p}$ (color digit of "virtual wire")
  $V \leftarrow$ random wire label
  for $j = 0$ to $q - 1$:
    $G_{2,\tau_b+j} = H(g; W_b^0 + j\Delta_p) + V - (\boxed{\tilde{\tau}} + j)W_a^0$
    $\boxed{G_{3,\tau_b+j} = H'(g; W_b^0 + j\Delta_p) + \tilde{\tau} + j}$

  $W_g^0 = -U + V$
  the garbled gate is $G_{0,1}, \ldots, G_{1,p-1}, G_{2,0}, \ldots, G_{2,p-1}$
```

Note that each $G_{3,\cdot}$ is an encryption of a short $\mathbb{Z}_p$ value. Hence we use H' to denote a hash function with output domain $\mathbb{Z}_p$.

To evaluate, we decrypt the appropriate $G_{3,\cdot}$ value to get the "virtual color digit" and then proceed as in the standard half-gate evaluation:

```
Evaluating a multiplication gate $g$:
    $(a, b) = g.\mathsf{inputs}$
    $p = a.\mathsf{domain}$
    $q = b.\mathsf{domain}$ $(q < p)$
    $\tau^*_a = \tau(W^*_a)$
    $\tau^*_b = \tau(W^*_b)$

    $\triangleright$ garbler's half-gate
    $U^* = G_{1,\tau^*_a} - H(g; W^*_a)$

    $\triangleright$ evaluator's half-gate
    $\boxed{\tilde{\tau}^* = G_{3,\tau^*_b} - H'(g; W^*_b)}$
    $V^* = \boxed{\tilde{\tau}^*} \cdot W^*_a + G_{2,\tau^*} - H(g; W^*_b)$

    output wire label is $-U^* + V^*$
```

Correctness follows from a similar reasoning as before.

Security. Security also follows a similar reasoning as in BMR. The hardness assumption used in that proof is that H is a kind of circular correlation-robust hash function. In short, this means that expressions of the form

$$H(g; W + \alpha\Delta_p) + \beta\Delta_q$$

are pseudorandom, when W, α, β, p, q are chosen by the adversary (with $a \neq 0$), and Δ_p and Δ_q are secret. The additions are with respect to the appropriate moduli.

Making the same assumption about H' allows the new security proof to proceed. At a high level, the proof proceeds by performing a "perspective shift" in the garbling algorithm, from the garbler's point of view (in terms of W^0 and τ values) to the evaluator's view (in terms of W^* and τ^* values).

After rewriting the garbling algorithm in these terms, we see that the $G_{\cdot,\cdot}$ values that are **not** accessed by the evaluation algorithm are written in the form $G_{\cdot,\cdot} = H(g; W + \alpha\Delta) + \beta\Delta' + Z$, where $\alpha \neq 0$ and W, Z, α, β are known. Hence, these terms are pseudorandom. In short, the garbled gate ciphertexts can be replaced by random values in the security proof. In doing so, the simulation no longer uses the truth values on the wires (they were only used in choosing the α, β values in the expression above). Hence, the simulation generates a garbled circuit without knowledge of the circuit input.

5 Improved Mixed-Radix Addition

Consider the problem of adding k terms, which are expressed in a mixed radix number system (Section 2.1). More specifically, each of the k values is represented in $\mathsf{MRS}[d_1, d_2, \ldots, d_n]$ in the circuit by a collection of n wires with corresponding moduli d_1, d_2, etc. We wish to efficiently compute the sum (also in the same mixed-radix representation) of k such values.

5.1 Background: Binary Addition

To add binary numbers (and to deal with any $\mathbb{Z}_2$-digits in a mixed-radix system), we are not aware of a more efficient approach than the straight-forward use of fan-in-2 full adders. A full adder takes in inputs x, y, c_{in} and gives output s, c_{out}, where s is the sum in this digit and c_{out} is the carry out. Using free-XOR [KS08] (to which BMR garbling collapses for $\mathbb{Z}_2$-wires), computing s is free. Using a folklore construction, the carry-out computation requires only one AND gate — 2 ciphertexts using the half-gates technique [ZRE15]:

$$c_{out} = \big[(x \oplus y) \wedge (x \oplus c_{in})\big] \oplus x$$

5.2 Improved Base-m Addition

Now consider a full-adder for $\mathbb{Z}_m$ digits. The sum $s = x + y + c_{in}$ is free to garble if all inputs are given on $\mathbb{Z}_m$-wires. To compute the carry-out, we propose the following. Let us first assume that $c_{in} \in \{0, 1\}$, which would be the case in a normal addition of two numbers.

1. With three unary gates, transfer x, y, c_{in} from $\mathbb{Z}_m$-wires into $\mathbb{Z}_{2m}$ wires. The cost to garble 3 such unary gates is $3(m - 1)$ ciphertexts.

2. With x, y, c_{in} now represented in $\mathbb{Z}_{2m}$, add them (for free) over $\mathbb{Z}_{2m}$. Note that the largest possible sum is $(m - 1) + (m - 1) + 1 = 2m - 1$, which does not wrap around.

3. Compute the carry-out with a unary gate, via $(x + y + c_{in}) \mapsto \lfloor \frac{x+y+c_{in}}{m} \rfloor$. The cost of this unary gate is $2m - 1$ since the input wire is a $\mathbb{Z}_{2m}$-wire.

The total cost of the full adder is $3(m - 1) + (2m - 1) = 5m - 4$.

However, if we are adding more than two terms, we can do better. Consider adding up three numbers in the mixed-radix system. The naïve approach is to use two copies of a fan-in-two adder. The total cost for a $\mathbb{Z}_m$ digit will be $2(5m - 4) = 10m - 8$. We can do better by adding all three values in one step, as a fan-in-three adder.

1. Let x, y, z and c_{in} be the inputs, given on $\mathbb{Z}_m$-wires. We compute the sum $s = x + y + z + c_{in}$ for free, as usual.

2. Let c_{max} be the largest possible value of c_{in}.[4] With four unary gates, transfer each $\mathbb{Z}_m$ to $\mathbb{Z}_{3m+c_{max}-1}$. Total garbling cost is $4(m - 1)$.

3. Add all input values mod $3m + c_{max} - 1$ for free. Again, this sum does not overflow.

4. With one unary gate, compute the carry-out as $\lfloor \frac{x+y+z+c_{in}}{m} \rfloor$. Total garbling cost is $3m + c_{max} - 2$.

Now the total cost is only $4(m - 1) + (3m + c_{max} - 2) = 7m - 6 + c_{max}$, a significant improvement.

5.3 Generalization

In general, we need to add k values represented in mixed-radix. Rather than adding numbers two-at-a-time, we add digit-by-digit, processing all k values in each digit at once. Each digit has a native modulus, but also a modulus that it uses to compute the carry-out (e.g., in the above example, this auxiliary modulus was $3m + c_{max} - 2$). Each digit provides carry-out to its neighbor in both moduli. A few edge cases are worth pointing out:

[4] When adding three numbers, the neighboring digit could have provided a carry larger than one.

- The most-significant digit does not compute a carry-out, so the preceding digit does not need to provide its carry in the carry-computation modulus.

- We handle $\mathbb{Z}_2$-digits using the half-gates fan-in-two adder. When a $\mathbb{Z}_2$-digit gives carry-out to a non-$\mathbb{Z}_2$-digit, we must collect all $k-1$ individual carry-outs with some extra logic.

- Suppose some $\mathbb{Z}_m$ digit computes its own carry-out under modulus $3m$. It must give this carry-out to the neighboring digit in two different moduli. This can be done with two unary gates, each with garbling cost $3m-1$. But suppose the neighboring digit is $\mathbb{Z}_\ell$ and the carry-out is guaranteed to not overflow mod ℓ. Then we can compute carry-out with a unary gate $\mathbb{Z}_m \to \mathbb{Z}_\ell$ as before, but then "copy" this value (with another unary gate) from $\mathbb{Z}_\ell$ to the neighboring digit's carry-modulus. The total cost is $(3m-1)+(\ell-1)$ which is nearly always less than $2(3m-1)$.

Finally, our neural-network applications do not require the full result of the addition. Rather, they **only require the most-significant digit** of the result. When this is the case, we can save even further by computing only the carry-out for all but the most-significant digits.

Now, every digit computes just a single sum. For most digits this is a sum over a modulus chosen so that the addition doesn't overflow (so we can compute the carry with a unary gate). For the most significant digit d_1, this is the sum mod d_1 that actually computes the most significant digit of the final answer. Then every digit except the most-significant one computes its carry-out via a single unary gate, whose output modulus is the appropriate modulus for the next digit's sum computation.

Overall, the cost of computing the most significant digit of MRS addition is as follows. In all but the most significant digit, we use k unary gates to convert the digits of the k summands to the correct modulus. For a $\mathbb{Z}_d$-digit, the cost is $k(d-1)$ ciphertexts. These values are added for free over the appropriate modulus. Then the carry-out is computed with a single unary gate over the "carry-modulus" for that digit. The carry modulus is $k(d-1)+m$ where m is the maximum carry-in value. The overall cost for $\mathsf{MRS}[d_1, d_2, \cdots, d_n]$ is

$$2k\sum_{i=2}^{n}(d_i-1) + [\text{sum of maximum carry values}]$$

An upper bound on each maximum carry value is $k-1$, so an upper bound on the total cost is $2k\sum(d_i-1)+(k-1)^2$.

6 (Approximate) Garbled Sign

In this section we discuss new approaches for garbling the function $\mathsf{sgn} : \mathbb{Z}_{P_k} \to \{0,1\}$ where

$$\mathsf{sgn}(x) = \begin{cases} 0 & \text{if } x < P_k/2 \\ 1 & \text{if } x \geq P_k/2 \end{cases}$$

This corresponds to the natural concept of sign, when we interpret $\mathbb{Z}_{P_k}$ as $\{-P_k/2+1, \ldots, P_k/2\}$ rather than $\{0, \ldots, P_k-1\}$.

The sgn function can be used in its own right as an activation function in neural networks, or as a component in other activation functions. For example, the ReLU activation function is defined as $\mathsf{relu}(x) = \max\{0, x\}$, and computed as $\mathsf{relu}(x) = \mathsf{sgn}(x) \cdot x$.

How sgn is handled in BMR. The approach for sgn in BMR is to first convert from residue representation $[\![x]\!]_{\mathsf{crt}}$ to another representation called *primorial mixed-radix* (PMR). In the notation of Section 2.1, this is the $\mathsf{MRS}[2, 3, 5, 7, 11, \ldots]$ representation. BMR show a technique to convert from residue representation to PMR representation. Once in PMR, the sign can be computed for free by checking whether the most significant digit is 1 — *i.e.*, items 0 through $P_k/2 - 1$ have most significant digit 0 and items $P_k/2$ through $P_k - 1$ have most significant digit 1.

6.1 Conceptual Overview

We start with a common technique for residue number systems (*e.g.*, [HP94, BEPP99]). It is well-known that the reconstruction of a value $x \in \mathbb{Z}_{P_k}$ from its residue representation $[\![x]\!]_{\mathsf{crt}}$ is a **linear** operation. That is, there exist integers $\alpha_1, \ldots, \alpha_k$ (which depend only on $p_1, \ldots, p_k$) such that

$$x \equiv \textstyle\sum_i \alpha_i x_i \pmod{P_k}$$

Over the integers, the sum becomes:

$$x = q \cdot P_k + \textstyle\sum_i \alpha_i x_i$$

for some integer q. Divide both sides by P_k and we get:

$$\begin{aligned} \frac{x}{P_k} &= q + \textstyle\sum_i \frac{\alpha_i x_i}{P_k} \\ \implies \left[\text{fractional part of } \frac{x}{P_k}\right] &= \left[\text{fractional part of } \textstyle\sum_i \frac{\alpha_i x_i}{P_k}\right] \end{aligned}$$

And therefore:

$$\begin{aligned} \mathsf{sgn}(x) = 1 &\iff x \geq P_k/2 \\ &\iff \frac{x}{P_k} \geq 1/2 \\ &\iff \left[\text{fractional part of } \textstyle\sum_i \frac{\alpha_i x_i}{P_k}\right] \geq 1/2 \end{aligned}$$

These observations lead to the following algorithm for computing $\mathsf{sgn}(x)$. Later we will discuss how to carry out this algorithm efficiently within a mixed-modulus circuit.

1. First convert each x_i to $\alpha_i x_i / P_k$, **represented as fixed-point approximation**. In more detail, for some discretization level M (whose selection is discussed below), round the rational number $\alpha_i x_i / P_k$ to the nearest fraction with denominator M. This approximation d/M will be represented simply as $d \in \mathbb{Z}_M$. The overall conversion of x_i to d can be computed as a simple lookup table, as the range of values for each x_i is small.

2. Add these fractional approximations, ignoring the integral part. This corresponds to adding their representations (numerators) mod M. This gives an approximation of the fractional part of $\sum_i \alpha_i x_i / P_k$.

3. Finally, compare the resulting sum to $M/2$.

Correctness, error, precision. Each term $\alpha_i x_i/P_k$ is approximated by a fixed-point value d/M to within error $1/2M$. The sum has k terms, so the total error is at most $k/2M$. If this error is less than $1/P_k$ then the result is correct. Hence $M > kP_k/2$ will guarantee a correct computation. Smaller values of M can also lead to correct results for all of $\mathbb{Z}_{P_k}$, which we discuss below. As we will also see, choosing a smaller M may lead to a significantly less expensive computation, which is correct on most inputs (e.g., 99.9% of inputs).

6.2 Garbling Costs for the Sign Function

As mentioned above, this general approach appears in other works dealing with residue number systems (*e.g.*, [HP94]). In this work we explore more of the design space, informed by how the approach translates to a mixed-modulus circuit suitable for garbling. Specifically: What M should be chosen, and how do we represent $\mathbb{Z}_M$ in a way that admits efficient addition (Step 2) and also comparison (Step 3)?

A simple choice is to represent these fixed-point values via a single $\mathbb{Z}_M$-wire in the circuit. This causes addition-mod-M to be free, but the comparison against $M/2$ is expensive — a unary gate that costs $M-1$ ciphertexts to garble. Using a recursive construction to let M be a smaller primordial allows us to use residue representation for $\mathbb{Z}_M$ (hence free addition), but overall the approach is expensive.

A better choice is to give up on free addition mod M. Suppose we choose $M = 2^t$ and represent values $\mathbb{Z}_M$ as t boolean wires in the circuit. This choice has the following effect on the costs of garbling:

- In Step 1 (converting each x_i to an approximation $d \in \mathbb{Z}_M$), we now have t unary gates for each x_i — one for each binary digit (wire) of d. Overall, garbling this step requires $t\sum_i(p_i-1)$ ciphertexts.

- In Step 2, we add values mod M. This is no longer free, but requires binary addition circuits (ignoring the carry-out). Using the free-XOR method (which BMR collapses to in the case of $\mathbb{Z}_2$-wires), the cost of adding two t-digit binary numbers is $2(t-1)$ ciphertexts. There are $k-1$ such additions, to sum k values, for a total garbling cost of $2(k-1)(t-1)$ ciphertexts.

- In Step 3, we need to compare the sum against $M/2$. Since the sum is represented in binary, the result of this comparison already exists as the most-significant-bit of the sum. So this step is free!

Generalizing even further. There is nothing particularly special about representing $\mathbb{Z}_M$-values in binary. We could use almost any **mixed-radix system** (see Section 2.1). Suppose we choose $M = m_1 \cdots m_2 \cdots m_t$, and we represent numbers in $\mathsf{MRS}[m_1, \ldots, m_t]$. Numbers in this base system can be added by adding the least significant digits mod m_t, then taking carry-over into the next digit ($\mathbb{Z}_{m_{t-1}}$), and so on.

Then the cost of garbling the sgn function is:

- In Step 1 (converting each x_i to its approximation $d \in \mathbb{Z}_M$) consists of t unary gates per prime p_i, one for each digit of the $\mathsf{MRS}[m_1, \ldots, m_t]$-representation. Total cost $= t\sum_i(p_i-1)$

- In Step 2, the cost is that of adding k values represented in $\mathsf{MRS}[m_1, \ldots, m_t]$. We use the mixed-radix addition ideas described in Section 5. The total cost of this step is at most $2k\sum_{i=2}^{t}(m_i-1) + (k-1)^2$.

- In Step 3, we compare the sum against $M/2$. Provided that the **most significant digit** m_1 **is even**, this can be done by simply inspecting the most-significant digit. We simply check whether the most significant digit is greater than or equal to $m_1/2$, using a unary gate of cost $m_1 - 1$ ciphertexts. Note that this implies we only need to compute the most-significant digit of the summation in step 2.

This flexibility gives us a wide design space to choose different values for M (and their factorizations) in an effort to evenly balance cost across these three contributors. Usually, the best choices of $M = m_1 \cdots m_t$ are when m_1 is somewhat large (larger than 50), and $m_2, \dots, m_t$ are relatively small (less than 10).

6.3 Concrete Costs

				garbling cost (ctxts)			
k	$\log_2(P_k)$		M = mixed-radix	1	2	3	total
3	4.9	BMR	-	-	-	-	55
		us	$14 = 14$	7	0	13	**20**
		us	$32 = 2^5$	35	16	0	51
4	7.7	BMR	-	-	-	-	130
		us	$68 = 68$	13	0	67	80
		us	$128 = 2^7$	91	36	0	127
		us	$78 = 26 \cdot 3$	26	16	25	**67**
5	11.2	BMR	-	-	-	-	269
		us	$538 = 538$	23	0	537	560
		us	$2048 = 2^{12}$	276	88	0	364
		us	$648 = 54 \cdot 4 \cdot 3$	69	53	53	**175**
6	14.9	BMR	-	-	-	-	476
		us	$6070 = 6070$	35	0	6069	6104
		us	$16384 = 2^{14}$	490	130	0	620
		us	$7500 = 60 \cdot 5^3$	140	153	49	**352**
7	19.0	BMR	-	-	-	-	787
		us	$524288 = 2^{19}$	969	216	0	1185
		us	$108360 = 86 \cdot 7 \cdot 6^2 \cdot 5$	255	297	85	**637**
8	23.2	BMR	-	-	-	-	1198
		us	$16777216 = 2^{24}$	1656	322	0	1978
		us	$1932000 = 92 \cdot 7 \cdot 6 \cdot 5^3 \cdot 4$	483	450	91	**1024**
9	27.7	BMR	-	-	-	-	1753
		us	$268435456 = 2^{28}$	2548	432	0	2980
		us	$31933300 = 76 \cdot 7^5 \cdot 5^2$	728	731	75	**1534**
10	32.6	BMR	-	-	-	-	2512
		us	$8589934592 = 2^{33}$	3927	576	0	4503
		us	$791920800 = 202 \cdot 11^2 \cdot 6^4 \cdot 5^2$	1071	1022	201	**2294**
11	37.5	BMR	-	-	-	-	3431
		us	$137438953472 = 2^{37}$	5513	720	0	6233
		us	$39690000000 = 150 \cdot 8 \cdot 7^2 \cdot 6^3 \cdot 5^5$	1788	1286	149	**3223**

Figure 1: Garbling cost of **exact** sign computation. This table illustrates our approach of choosing M to balance the costs of steps 1,2,3 in the overall sgn algorithm described in Section 6.1. k is the number of primes in the CRT representation. P_k is the corresponding primorial modulus (product of first k primes), so $\log_2(P_k)$ is the equivalent number of bits to represent numbers in $\mathbb{Z}_{P_k}$.

The correctness of the sgn computation depends only on the choice of M and not its mixed-radix representation. Step 2 is done in $\mathbb{Z}_M$ no matter how $\mathbb{Z}_M$ is represented in mixed-radix.

Interestingly, increasing M does not always decrease the overall number of errors. For example, with $k = 5$ primes, our sgn construction has perfect correctness only for $M \in \{538, 648, 678, 688, \ldots\}$. We do not understand these patterns. Instead, we empirically test a candidate M for its correctness. Recall that our sgn gadget can be incorrect only on numbers within $P_k k/2M$ of one of the discontinuities of the sgn function (which are at 0 and $P_k/2$). Hence, to check the correctness of a candidate M, it suffices to check the behavior of the gadget on numbers in this range.

For $k \leq 11$ primes, checking the entire relevant range of M-candidates (and their mixed-radix representations) is feasible for an exhaustive search. We now report on such an exhaustive search.

Exact sign computation. In Figure 1, we show the cost of garbling an exact sgn function, under different approaches for choosing M and its mixed-radix representation. We show (1) representing $\mathbb{Z}_M$ as a single wire, with the smallest M that yields perfect correctness; (2) choosing M as the smallest power of two that yields perfect correctness; and (3) the best possible M considering all mixed-radix representations.

We also compare to the exact sgn function described in BMR garbling. Interestingly, ours is cheaper for $k \leq 11$. We are not sure whether ours continues to be cheaper for $k \geq 12$, as that is the limit of our present exhaustive search capabilities.

Approximate sign computation. Our approach shines when we are willing to trade a tiny fraction of correctness errors for a significant decrease in garbling costs. In Figure 2 we show the garbling cost for various choices of M leading to correctness $\geq \tau$ for $\tau \in \{0.99, 0.999, 0.9999, 0.9999, 1\}$.

As is clear from the table, even a small degradation in correctness can result in a significant reduction of cost.

6.4 Other Activation/Pooling Functions

As mentioned previously, ReLU activation can be written as $\mathsf{relu}(x) = \mathsf{sgn}(x) \cdot x$. If x is encoded in CRT as $[\![x]\!]_{\mathsf{crt}} = (x_1, \ldots, x_k)$, we compute $[\![\mathsf{relu}(x)]\!]_{\mathsf{crt}}$ as $(\mathsf{sgn}(x) \cdot x_1, \ldots, \mathsf{sgn}(x) \cdot x_k)$. Each term here is the product of between a $\mathbb{Z}_2$ value and $\mathbb{Z}_{p_i}$ value, and we use the optimization from Section 4.

Similarly, we can compute a max (for max pooling layers) as $\mathsf{max}(x, y) = x + \mathsf{relu}(y - x)$. This is a combination of free addition/subtraction (for CRT-encoded values) and efficient components we have already described.

7 Implementation & Optimizations

We implemented BMR garbling in Rust. Our library, fancy-garbling, is available as open source[5]. Our library consists of tools for producing a garbling scheme only. If the user wishes to implement MPC, they must provide their own oblivious transfer implementation and so on. Finally, our implementation is optimized to reduce the cost of the main trade-off of BMR: converting wire-labels back and forth between representations. In this section, we highlight the main optimizations of our implementation.

7.1 Major Implementation-Level Optimizations

Unpacking: conversion from bitstring to mod-q digits. In BMR garbling, a wire-label is a list of digits modulo some small prime q (see Section 3). In order to encrypt a garbled gate,

[5] https://github.com/GaloisInc/fancy-garbling

k	$\log_2(P_k)$	M = mixed-base	correct	cost
4	7.7	$78 = 26 \cdot 3$	= 100%	67
		$54 = 18 \cdot 3$	≥ 99%	59
5	11.2	$648 = 54 \cdot 4 \cdot 3$	= 100%	175
		$450 = 30 \cdot 5 \cdot 3$	≥ 99.9%	161
		$108 = 36 \cdot 3$	≥ 99%	101
6	14.9	$7500 = 60 \cdot 5^3$	= 100%	352
		$5250 = 42 \cdot 5^3$	≥ 99.99%	334
		$960 = 48 \cdot 5 \cdot 4$	≥ 99.9%	240
		$120 = 40 \cdot 3$	≥ 99%	133
7	19.0	$108360 = 86 \cdot 7 \cdot 6^2 \cdot 5$	= 100%	637
		$10560 = 88 \cdot 6 \cdot 5 \cdot 4$	≥ 99.99%	470
		$1200 = 60 \cdot 5 \cdot 4$	≥ 99.9%	315
		$120 = 40 \cdot 3$	≥ 99%	169
8	23.2	$1975680 = 98 \cdot 9 \cdot 8^2 \cdot 7 \cdot 5$	= 100%	1078
		$107100 = 102 \cdot 7 \cdot 6 \cdot 5^2$	≥ 99.999%	770
		$10920 = 78 \cdot 7 \cdot 5 \cdot 4$	≥ 99.99%	574
		$1170 = 78 \cdot 5 \cdot 3$	≥ 99.9%	385
		$126 = 126$	≥ 99%	194
9	27.7	$31933300 = 76 \cdot 7^5 \cdot 5^2$	= 100%	1534
		$119700 = 114 \cdot 7 \cdot 6 \cdot 5^2$	≥ 99.999%	933
		$12600 = 84 \cdot 6 \cdot 5^2$	≥ 99.99%	696
		$1260 = 140 \cdot 9$	≥ 99.9%	465
		$138 = 138$	≥ 99%	228
10	32.6	$791920800 = 202 \cdot 11^2 \cdot 6^4 \cdot 5^2$	= 100%	2294
		$128520 = 102 \cdot 7 \cdot 6^2 \cdot 5$	≥ 99.999%	1122
		$13440 = 112 \cdot 6 \cdot 5 \cdot 4$	≥ 99.99%	843
		$1330 = 190 \cdot 7$	≥ 99.9%	547
		$140 = 140$	≥ 99%	258
11	37.5	$39690000000 = 150 \cdot 8 \cdot 7^2 \cdot 6^3 \cdot 5^5$	= 100%	3223
		$136500 = 130 \cdot 7 \cdot 6 \cdot 5^2$	≥ 99.999%	1320
		$13398 = 174 \cdot 11 \cdot 7$	≥ 99.99%	981

Figure 2: Garbling cost of **approximate** sign computation, measured in ciphertexts. k is the number of primes in the CRT representation. P_k is the corresponding primorial modulus (product of first k primes), so $\log_2(P_k)$ is the equivalent number of bits to represent numbers in $\mathbb{Z}_{P_k}$.

these wire-labels must be converted into a string of 128 bits in order to send to fixed-key AES (used as a hash function). We call this operation "packing." Packing is cheap. Packing takes about 20 nanoseconds, using Horner's method (adding each digit and multiplying by q, one by one). Unfortunately the other direction — "unpacking" a list of mod-q digits from a bitstring — is much more expensive. It takes about 4 microseconds to naively unpack a bitstring by dividing off each mod-q digit.

We improve the efficiency of unpacking base-q digits by using lookup tables. The idea is as follows: we first first break up the 128 bit string into 16 bit chunks. Then, precompute a lookup table of the 2^{16} *shifted* base-q numbers it could correspond to. For instance the first chunk is not shifted at all, the second chunk gets shifted by 16 bits before converted into base-q, the third chunk gets shifted by 32 bits before converted into base-q, and so on. This shifting allows us to avoid multiplication in base-q, instead precomputing it in binary. Then to unpack a bitstring into base-q digits, look up each 16-bit chunk in the table, using base-q addition to add the results together. This technique reduces the cost of unpacking to about 400 nanoseconds, a 10× improvement.

Streaming garbler & evaluator. The circuits for convolutional neural networks are very large, easily using more than 16GB of memory. To get around this issue, we implemented the existing technique of streaming. Instead of generating a circuit first and then garbling or evaluating it, the garbler and evaluator directly *execute* BMR instructions – adding, multiplying, and projecting wire-labels – and sending garbled gates to the evaluator as they are generated. Our streaming method is implemented as a Rust trait, a generic interface that various classes can implement. We call our trait Fancy. It contains the basic BMR operations such as `add`, `mul`, and `project`. In addition, the activation gadgets described in Section 6 are implemented in terms of Fancy, allowing them to be used by the garbler or evaluator directly, with no special coding required.[6] See Figure 3 for more details on the architecture of fancy-garbling.

Parallelizable garbling & evaluation. Parallelization occurs at the Fancy level. In order to support parallel garbling and evaluation, we designed a method to ensure potentially out-of-order garbled gates were delivered to the right thread of the evaluator. This works through a special *sync* mode, where all Fancy operations take an additional index argument, which often corresponds to the thread number. The garbler produces garbled gates which have an associated index. The evaluator uses a special "postman" thread which coordinates delivering garbled gates to the threads that need them. The overhead for this coordination is expensive enough that it makes sense only for very large computations like convolutional neural networks.

7.2 Fancy Implementation of Neural Networks

We implemented convolutional neural networks for fancy-garbling. Our implementation works by reading a TensorFlow model exported directly from Keras [C+15a] as JSON, along with weights, biases, test data, and test labels. The neural network itself is divided into a series of layers, as in TensorFlow. Each layer may be Convolutional, Dense, Flatten, or MaxPooling. Each layer has an "as-fancy-computation" method, which computes the given layer using an arbitrary object that implements Fancy. This allows us to both test the correctness of the layer using the Dummy object, but also evaluate the layer as a streaming garbled circuit using a Garbler and Evaluator.

Inputs and weights to the neural network are encoded in CRT, with the minimum number of prime residues necessary to fit the intermediate values and preserve the accuracy of the neural network. Our activation functions are direct implementations of the methods described in Section 6. Finally, we parallelize at this level, splitting the computation of a layer into eight threads.

By default, a neural network layer evaluates its input using *public weights and biases.* Since the weights are public, we can use scalar multiplication to multiply them with the input, which is free in BMR. This is the cause of our low communication costs with public weights in Section 8.

We also support secret weights and biases. To do this, we use projection gates. Projection gates can be programmed by the garbler using truth tables that are oblivious to the evaluator, in the style of classic Yao garbling. Then, in order to multiply the input by a secret weight w, we simply compute the truth table consisting of $x \cdot w$ for every $x \in [q]$, and use the result as a BMR projection gate. This technique results in higher communication costs than public weights, since projection gates are not free (the ciphertext cost of a single projection gate in BMR is $q - 1$ per CRT residue). Note that this method is cheaper than treating the weight as a garbler input and using a multiplication gate, a method which has a base cost of $2q - 2$ per residue.

[6] fancy-garbling provides a number of other objects which implement Fancy besides Garbler and Evaluator. Dummy evaluates the computation in the clear for debugging. Informer is used to calculate ciphertext size and performance characteristics. Finally, CircuitBuilder is used to build a static circuit, which can be saved.

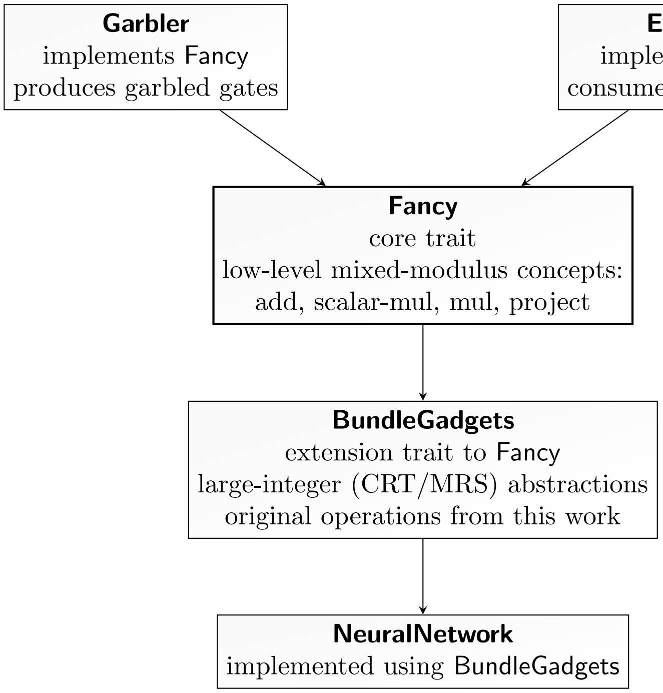

Figure 3: **Architecture of fancy-garbling for neural networks.** The core of fancy-garbling is the Fancy trait, which contains the basic low-level mixed-modulus circuit operations of BMR addition, multiplication, and projection gates. The Garbler and Evaluator both provide implementations of Fancy. This allows us to use the *exact same code* for both the Garbler and Evaluator, increasing confidence in our implementation and reducing the surface area of hard-to-understand bugs inherent in building circuits for MPC. It also means that we only have to implement the neural network in terms of Fancy, and we get a Garbler and Evaluator for neural networks for free. Other implementors of Fancy such as Dummy allow us to check its correctness by evaluating it in the clear. This design also allows us to forgo using a circuit at all, simply evaluating the neural network directly, streaming garbled gates from the Garbler to the Evaluator.

Our neural network implementation also supports Boolean garbling. We strove to provide as optimal an implementation as possible, despite not including the techniques described in DeepSecure [RRK17], due to implementation effort. We implement public weights using bit-shift (free) and binary addition (cheap), shifting the value to implement cheap multiplication, e.g. $7x = 4x+2x+x$. We implement secret weights by treating the weights as garbler inputs and using binary multiplication to multiply them with the input. This is quite expensive, unfortunately. Finally, activations are straightforward and cheap/free in Boolean: we simply output the most significant bit to obtain sgn, and use same bit as a mask to implement relu.

8 Experimental Results

All experiments are executed on a machine using an eight-core 3.7Ghz AMD CPU with 32GB RAM. Neural network classifiers are trained using the Keras library [C^{+}15b] in Python running on top of TensorFlow. These models are trained to classify two datasets: MNIST and CIFAR-10.

The MNIST dataset is a collection of 70,000 labeled images of handwritten digits [LC98]. Each grayscale image is a 28×28 matrix of integers in the range $[0, 255]$ with a corresponding label in the range $[0, 9]$. The standard training set consists of 60,000 images with a test set of 10,000 images.

The CIFAR-10 dataset consists of 60,000 color images in 10 classes (airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks) [KH09]. Each image is represented as a $32 \times 32 \times 3$ matrix of integers in the range $[0, 255]$. There are 50,000 images in the training set and 10,000 in the test set.

After training, each model is discretized in a similar manner as the FHE-DiNN work [BMMP17]. Specifically, the weights and biases are rounded to the nearest integer after being scaled by a factor that does not significantly reduce accuracy. In some cases, we find the best accuracy by training with a tanh activation function for models which, after being discretized, use the sign activation function. In the following section, we describe the neural network models we used for our experiments.

8.1 Neural Network Experimental Models

MNIST Model A. This model consists of three fully connected layers with 128 neurons in the first two layers and ten in the last. We use the ReLU activation function in the first two layers for training as was done originally in SecureML [MZ17b]. Evaluation results are in Table 1; fancy-garbling uses the more common ReLU activations for testing where the other frameworks use a square activation function. This results in significantly higher accuracy and matches standard practice in neural networks (square is not commonly used).

	Runtime (s)		Com. (MB)		Acc.
	Offline	Total	Offline	Total	%
SecureML	4.7	4.88	-	-	93.1
MiniONN	0.9	1.04	3.8	47.6	97.6
GAZELLE	-	0.03	-	0.5	-
	Garbling	Total	Offline	Total	%
boolean garbling	8.8	53	-	618	96.8
boolean garbling (sw)	45	-	-	3407	96.8
fancy-garbling	0.06	0.12	-	4.43	96.8
fancy-garbling (sw)	0.54	1.98	-	128	96.8
fancy-garbling (99.99%)	0.05	0.1	-	2.77	95.7
fancy-garbling (99.99%)(sw)	0.47	1.98	-	127	95.7

Table 1: MNIST Model A. For our results, both fancy-garbling and boolean garbling, the notation "(sw)" means the weights are kept secret. Otherwise, weights are public. The notation "(x%)" means the activation functions have x% approximate correctness, otherwise they are exact. We divide our runtime into garbling only ("Garbling") and streaming garbler to an evaluator ("Total"). We note that garbled circuits have no offline mode, so we report no offline communication. Our neural network requires 22 bits (or the first 8 primes in CRT-mode) to evaluate. We include standard Boolean garbling as a baseline. Note that DeepSecure is also based on Boolean garbling but contains optimizations we did not include, hence it has better performance than our Boolean baseline. A dash (-) indicates that either results were not reported, not applicable (such as offline mode in fancy-garbling), or the runtime was too large to run to completion.

MNIST Model B. A convolutional neural network with 6 layers originally described by the CryptoNets work [GBDL+16]. Their results in Table 2 are based on optimizations of the following model:

1. Convolutional layer with 5 kernels and square activation,

2. Mean pooling layer,
3. Convolutional layer with 10 kernels and square activation,
4. Mean pooling layer,
5. Fully connected layer with 100 neurons and a square activation,
6. Fully connected layer with 10 neurons and a sigmoid activation.

Our implementation of this model uses the more common ReLU activation function (rather than square activation) and max pooling (rather than mean pooling).

	Runtime (s)		Com. (MB)		Acc.
	Offline	Total	Offline	Total	%
CryptoNets	-	297.5	-	372.2	98.95
MiniONN	0.88	1.28	3.6	15.8	98.95
GAZELLE	0	0.03	0	0.5	-
	Garbling	Total	Offline	Total	%
boolean garbling	9.6	74	-	877	86.72
boolean garbling (sw)	49	-	-	4717	86.72
fancy-garbling	0.67	2.19	-	160	86.72
fancy-garbling (sw)	1.17	3.87	-	290	86.72

Table 2: MNIST Model B. See Table 1 for label descriptions. We were not able to get high accuracy using approximate activations on this network. Our neural network requires 26 bits (or the first 9 primes in CRT-mode) to evaluate.

MNIST Model C. A convolutional neural network with three layers described in DeepSecure [RRK18]. The results in Table 3 are based on this model. This model consists of the following layers:

1. Convolutional layer with 5 kernels and ReLU activation,
2. Fully connected layer with 100 neurons and a ReLU activation,
3. Fully connected layer with 10 neurons and a softmax activation.

	Runtime (s)		Com. (MB)		Acc.
	Offline	Total	Offline	Total	%
DeepSecure	-	9.67	-	791	99.0
GAZELLE	0.15	0.20	5.9	8.0	-
	Garbling	Total	Offline	Total	%
boolean garbling	6.25	44	-	453	97.21
boolean garbling (sw)	37	-	-	3410	97.21
fancy-garbling	0.17	0.38	-	23	97.21
fancy-garbling (sw)	0.63	2.27	-	161	97.21

Table 3: MNIST Model C. See Table 1 for label descriptions. We were not able to get high accuracy using approximate activations on this network. Our neural network requires 24 bits (or the first 9 primes in CRT-mode) to evaluate.

MNIST Model D. A convolutional neural network with six layers described in MiniONN [LJLA17]. The results in Table 4 are based this model. This model consists of:

1. Convolutional layer with 16 kernels and ReLU activation,
2. Max pooling layer,

3. Convolutional layer with 16 kernels and ReLU activation,
4. Max pooling layer,
5. Fully connected layer with 100 neurons and a ReLU activation,
6. Fully connected layer with 10 neurons.

	Runtime (s)		Com. (MB)		Acc.
	Offline	Total	Offline	Total	%
MiniONN	3.58	9.32	20.9	657.5	99.0
ExPC	-	5.1	-	501	99.0
GAZELLE	0.481	0.81	47.5	70.0	-
	Garbling	Total	Offline	Total	%
fancy-garbling	1.3	4.64	-	321	96.44
fancy-garbling (sw)	3.8	17	-	1023	96.44
fancy-garbling (99.99%)	1.01	3.13	-	190	87
fancy-garbling (99.99%)(sw)	3.66	15.58	-	892	87

Table 4: MNIST Model D. See Table 1 for label descriptions. Our neural network requires 20 bits (or the first 8 primes in CRT-mode) to evaluate.

MNIST Model E. Two fully connected layers with 30 neurons in the first layer and 10 in the last (we also test a model with 100 neurons in the first layer). We use the tanh activation function in the first layer for training and the sign activation function for testing, as was done originally in [BMMP17]. Evaluation results are in Table 5.

30 Neurons	Runtime (s)		Com. (MB)	Acc.
	Enc	Eval	Total	%
FHE-DiNN30	0.000168	0.49	8.2 kB	93.71
	Garbling	Total	Total	%
fancy-garbling	0.004	0.007	0.08	93.42
fancy-garbling (sw)	0.026	0.09	2.88	93.42
fancy-garbling (99%)	0.004	0.007	0.05	88.84
fancy-garbling (99%)(sw)	0.024	0.09	2.85	88.84

100 Neurons	Runtime (s)		Com. (MB)	Acc.
	Enc	Eval	Total	%
FHE-DiNN100	0.000168	1.65	8.2 kB	96.35
	Garbling	Total	Total	%
fancy-garbling	0.009	0.016	0.27	95.6
fancy-garbling (sw)	0.074	0.286	9.61	95.6
fancy-garbling (99%)	0.009	0.013	0.16	92.8
fancy-garbling (99%)(sw)	0.075	0.278	9.5	92.8

Table 5: MNIST Model E. See Table 1 for label descriptions. Our neural networks require 9 bits (or the first 5 primes in CRT-mode) to evaluate.

CIFAR-10 Model. A convolutional neural network model similar to the one originally described in MiniONN [LJLA17]. We use the tanh activation function in some layers for training and sign for testing. Model details follow.

1. Convolutional layer with 32 kernels and ReLU activation,
2. Convolutional layer with 32 kernels and tanh activation,

3. Mean pooling layer,
4. Convolutional layer with 64 kernels and ReLU activation,
5. Convolutional layer with 64 kernels and tanh activation,
6. Mean pooling layer,
7. Convolutional layer with 128 kernels and ReLU activation,
8. Convolutional layer with 128 kernels and tanh activation,
9. Fully connected layer with 10 neurons and a softmax activation.

Our implementation of this model uses max pooling in layers 3 and 6. Results are found in Table 6.

	Runtime (s)		Com. (MB)		Acc.
	Offline	Total	Offline	Total	%
MiniONN	472	544	3046	9272	81.61
GAZELLE	9.34	12.9	940	1236	-
	Garbling	Total	Offline	Total	%
fancy-garbling	64	161	-	2718	73.74
fancy-garbling (sw)	286	1162	-	43429	73.73

Table 6: CIFAR-10. See Table 1 for label descriptions. Our neural network requires 23 bits for the first two layers, then 12 for all remaining layers. In CRT-mode, this requires the first 8 primes for the first two layers, then the first 6 for all remaining layers.

8.2 Experimental Observations

Performance. Note that the reported results show that our running time outperforms every other scheme which has a constant number of rounds (those that are GC based, and even more so FHE based). We are slower than schemes with linearly many rounds, although our running time is still reasonable in many cases, and we expect to outperform these schemes in terms of latency (which we did not measure), which becomes more important the deeper the neural network is.

Accuracy. Our models are based on the descriptions reported in other works. If we could obtain the exact trained model weights from those other papers, we could discretize them and apply our methods. If we did this, we could better compare the effects of the discretization process on model accuracy. As it stands, accuracy is a haphazard metric to compare our work with others. For instance, it is possible to increase the accuracy of some of our models by putting more effort into the training process. In addition, loss of accuracy when weights are discretized depends on the values of the weights. That is, two models with the same accuracy but different weights may have different accuracy after discretization.

Instead of focusing on accuracy, our experiments highlight the difference in runtime and communication cost. Our models match previous work as closely as possible in number of neurons, layers, activations, etc. This means the *cost* difference will be as accurate as possible, even if the model accuracy is not a reliable metric to compare cryptographic protocols by.

We note that, in principle, our methods can be applied to ResNet architecture of neural networks and others. To date, our implementation supports convolutional neural networks using ReLU and sign activation functions.

Finally, we note that our discretization process is simple (scale and round-to-nearest-integer) and negatively affects the accuracy of a trained model. However, it has the trade-off in that it can be applied to a pre-trained model. Training a model directly over the integers rather than the real numbers would significantly improve accuracy.

Acknowledgements

Thanks to Dave Archer and Alex Malozemoff for guidance and comments on this work.

The first and third authors are supported in part by NSF grant CCF1423306 and the Leona M. & Harry B. Helmsley Charitable Trust. The first author is additionally supported in part by an IBM Research PhD Fellowship.

The second and fifth authors are supported in part by the ARO and DARPA under Contract No. W911NF-15-C-0227. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the ARO and DARPA.

The fourth author is supported by NSF grant 1617197.

References

[AEM08] Shai Avidan, Ariel Elbaz, and Tal Malkin. Privacy preserving pattern classification. In *Proceedings of the International Conference on Image Processing, ICIP 2008, October 12-15, 2008, San Diego, California, USA*, pages 1684–1687, 2008.

[BDK+18] Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, and Thomas Schneider. Hycc: Compilation of hybrid protocols for practical secure computation. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security,CCS*, pages 847–861, 2018.

[BEPP99] Hervé Brönnimann, Ioannis Emiris, Victor Y Pan, and Sylvain Pion. Sign determination in residue number systems. *Theoretical Computer Science*, 210:173–197, 1999.

[BFL+11] Mauro Barni, Pierluigi Failla, Riccardo Lazzeretti, Ahmad-Reza Sadeghi, and Thomas Schneider. Privacy-preserving ECG classification with branching programs and neural networks. *IEEE Trans. Information Forensics and Security*, 6(2):452–468, 2011.

[BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 784–796. ACM Press, October 2012.

[BLN14] Joppe W. Bos, Kristin E. Lauter, and Michael Naehrig. Private predictive analysis on encrypted medical data. *Journal of biomedical informatics*, 50:234–43, 2014.

[BMMP17] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. Fast homomorphic evaluation of deep discretized neural networks. Cryptology ePrint Archive, Report 2017/1114, 2017. https://eprint.iacr.org/2017/1114.

[BMMP18] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. Fast homomorphic evaluation of deep discretized neural networks. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 483–512. Springer, Heidelberg, August 2018.

[BMR16] Marshall Ball, Tal Malkin, and Mike Rosulek. Garbling gadgets for boolean and arithmetic circuits. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 565–577. ACM Press, October 2016.

[BOP06] Mauro Barni, Claudio Orlandi, and Alessandro Piva. A privacy-preserving protocol for neural-network-based computation. In *Proceedings of the 8th workshop on Multimedia & Security, MM&Sec 2006, Geneva, Switzerland, September 26-27, 2006*, pages 146–151, 2006.

[BPTG15] Raphael Bost, Raluca Ada Popa, Stephen Tu, and Shafi Goldwasser. Machine learning classification over encrypted data. In *NDSS 2015*. The Internet Society, February 2015.

[C$^+$15a] François Chollet et al. Keras. https://github.com/fchollet/keras, 2015.

[C$^+$15b] François Chollet et al. Keras. https://keras.io, 2015.

[CL05] Yan-Cheng Chang and Chi-Jen Lu. Oblivious polynomial evaluation and oblivious neural learning. *Theor. Comput. Sci.*, 341(1-3):39–54, 2005.

[GBDL$^+$16] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning*, pages 201–210, 2016.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.

[HP94] CY Hung and B Parhami. An approximate sign detection method for residue numbers and its application to rns division. *Computers & Mathematics with Applications*, 27(4):23–35, 1994.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[JKO13] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 955–966. ACM Press, November 2013.

[JVC18] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. Cryptology ePrint Archive, Report 2018/073, 2018. https://eprint.iacr.org/2018/073.

[KH09] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP 2008, Part II*, volume 5126 of *LNCS*, pages 486–498. Springer, Heidelberg, July 2008.

[LC98] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. http://yann.lecun.com/exdb/mnist/, 1998.

[LJLA17] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious neural network predictions via minionn transformations. Cryptology ePrint Archive, Report 2017/452, 2017. https://eprint.iacr.org/2017/452.

[MPs16] Tal Malkin, Valerio Pastro, and abhi shelat. An algebraic approach to garbling. Unpublished manuscript. Presented at Simons Institute workshop on securing computation: https://simons.berkeley.edu/talks/tal-malkin-2015-06-10, 2016.

[MZ17a] Payman Mohassel and Yupeng Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy*, pages 19–38. IEEE Computer Society Press, May 2017.

[MZ17b] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. Cryptology ePrint Archive, Report 2017/396, 2017. https://eprint.iacr.org/2017/396.

[NWI+13] Valeria Nikolaenko, Udi Weinsberg, Stratis Ioannidis, Marc Joye, Dan Boneh, and Nina Taft. Privacy-preserving ridge regression on hundreds of millions of records. *2013 IEEE Symposium on Security and Privacy*, pages 334–348, 2013.

[OPB07] Claudio Orlandi, Alessandro Piva, and Mauro Barni. Oblivious neural network computing via homomorphic encryption. *EURASIP Journal on Information Security*, 2007, 01 2007.

[OPJM10] Margarita Osadchy, Benny Pinkas, Ayman Jarrous, and Boaz Moskovich. SCiFI - a system for secure face identification. In *2010 IEEE Symposium on Security and Privacy*, pages 239–254. IEEE Computer Society Press, May 2010.

[PMG+17] Nicolas Papernot, Patrick D. McDaniel, Ian J. Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In Ramesh Karri, Ozgur Sinanoglu, Ahmad-Reza Sadeghi, and Xun Yi, editors, *ASIACCS 17*, pages 506–519. ACM Press, April 2017.

[RRK17] Bita Darvish Rouhani, M. Sadegh Riazi, and Farinaz Koushanfar. DeepSecure: Scalable provably-secure deep learning. Cryptology ePrint Archive, Report 2017/502, 2017. http://eprint.iacr.org/2017/502.

[RRK18] Bita Darvish Rouhani, M. Sadegh Riazi, and Farinaz Koushanfar. Deepsecure: Scalable provably-secure deep learning. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, pages 2:1–2:6, New York, NY, USA, 2018. ACM.

[RSC+19] M. Sadegh Riazi, Mohammad Samragh, Hao Chen, Kim Laine, Kristin Lauter, and Farinaz Koushanfar. Xonn: Xnor-based oblivious deep neural network inference. Cryptology ePrint Archive, Report 2019/171, 2019. https://eprint.iacr.org/2019/171.

[RWT+18] M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In Jong Kim, Gail-Joon Ahn, Seungjoo Kim, Yongdae Kim, Javier López, and Taesoo Kim, editors, *ASIACCS 18*, pages 707–721. ACM Press, April 2018.

[SHS$^+$15] Ebrahim M. Songhori, Siam U. Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. TinyGarble: Highly compressed and scalable sequential garbled circuits. In *2015 IEEE Symposium on Security and Privacy*, pages 411–428. IEEE Computer Society Press, May 2015.

[SS09] Ahmad-Reza Sadeghi and Thomas Schneider. Generalized universal circuits for secure evaluation of private functions with application to data classification. In Pil Joong Lee and Jung Hee Cheon, editors, *ICISC 08*, volume 5461 of *LNCS*, pages 336–353. Springer, Heidelberg, December 2009.

[TZJ$^+$16] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction apis. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 601–618, 2016.

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 220–250. Springer, Heidelberg, April 2015.