Matchmaker: Fast Secure Inference across Deployment Scenarios

Neha Jawalkar
Indian Institute of Science
India
jawalkarp@iisc.ac.in

Nishanth Chandran
Microsoft Research
India
nichandr@microsoft.com

Divya Gupta
Microsoft Research
India
divya.gupta@microsoft.com

Rahul Sharma
Microsoft Research
India
rahsha@microsoft.com

Arkaprava Basu
Indian Institute of Science
India
arkapravab@iisc.ac.in

Abstract

Secure Two-Party Computation (2PC) enables secure inference with cryptographic guarantees that protect the privacy of the model owner and client. However, it adds significant performance overhead. In this work, we make 2PC-based secure inference efficient *while considering important deployment scenarios.*

We observe that the hitherto unconsidered *latency of fetching keys from storage* significantly impacts performance, as does network speed. We design a Linear Secret Sharing (LSS)-based system LSS^{M} and a Function Secret Sharing (FSS)-based system FSS^{M} for secure inference, optimized for small key size and communication, respectively. Notably, our highly-optimized and hardware-aware *CPU-based* LSS^{M} outperforms prior GPU-based LSS systems by up to 50×. We then show that the best choice between LSS^{M} and FSS^{M} depends on the deployment scenario. In fact, under certain deployments, a *combination* of LSS^{M} and FSS^{M} can leverage heterogeneous processing across CPU and GPU. Such protocol-system co-design lets us outperform state-of-the-art secure inference systems by up to 21× (geomean 3.25×).

1 Introduction

Secure inference can be achieved via the cryptographic technique of secure 2-party computation (2PC). 2PC gives the formal guarantee that the client learns nothing about the model beyond the inference output and the model owner learns nothing about the client's input. While secure inference is well-explored ([20, 36, 38, 39, 42, 50, 52, 59, 61, 76] and references therein), state-of-the-art systems for secure inference via 2PC incur large performance overheads that limit its broad practical applicability. Our goal in this work is to reduce these overheads to make secure inference more useful.

The fastest systems for secure inference today [38, 42, 76] use the preprocessing model, which has two phases. In the *offline* phase, a trusted dealer gives input-independent correlated randomness, aka *keys*, to the two parties who wish to securely compute a function f. In the *online* phase parties use these keys to securely compute f on their secret inputs. Works in this model, including ours, focus on reducing online complexity.

In this work, we analyse and address the performance bottlenecks of the state-of-the-art (SOTA) 2PC systems for secure inference when deployed in real scenarios. We make the crucial observation that while the keys can be large in SOTA systems [38, 40, 42, 76] – e.g. for a reasonably sized model like VGG16 (batch size of 50), Orca [38] and CrypTen [42] need keys of size 255 GB and 580 GB, respectively – evaluation in all prior works make the unreasonable assumption that keys would be readily available in memory for fast consumption at inference time. This assumption, however, does not hold in practical systems serving high-throughput batch inference queries. In real-world settings, keys must be retrieved from storage, introducing a significant performance bottleneck. For instance, using Orca [38], the leading system for convolutional neural networks (CNNs), we observe that reading keys from storage incurs a substantial overhead of 9 minutes for a VGG16 model with a batch size of 50, whereas the online inference time, once the keys are loaded into memory, is only 20 seconds. This stark discrepancy highlights the need for a fundamental redesign of 2PC systems to address the challenges of secure inference at scale.

In particular, we explore how diverse practical deployment scenarios affect the performance of secure inferencing platforms and focus on a holistic protocol-system co-design that delivers significant speedups across varied scenarios. First, we make a critical observation that *whether the keys are available in memory or must be fetched from storage significantly impacts the performance of a secure inferencing service.* Moreover, the request arrival rate at the secure inference service can determine whether the keys can be found in the memory or must be fetched from the storage. The request arrival rate often varies widely and can be hard to predict for any webservice [66, 75], If the requests arrive intermittently, there is enough slack to fetch the keys into the memory before computation starts. However, at a high request arrival rate and/or when many requests come in a burst, the time to fetch keys from the storage would be in the critical path of execution.

The second factor that dictates performance is the network speed between the computing parties. The computing parties can reside in the same datacenter and thus be connected over a fast LAN network. They can also be located across different parts of the globe and thus be connected over a slow network, e.g., WAN. While the amount of communication needed for secure computation varies across protocols and systems, even the most communication-efficient secure inference systems [33, 38] transmit tens of GBs for reasonably large inference tasks. Consequently, the speed of the network connecting the parties has a significant bearing on the performance. [1]

We identify *four key deployment scenarios*, which arise from the combination of two crucial deployment considerations discussed above: ① whether the keys required for the online phase are readily

[1] While some of the prior works have evaluated their performance with varying network speeds, as we show later, variation in network speed alone does not provide much meaningful insight.

available in the memory or must be fetched from the storage, and ② the network speed between the parties (e.g., LAN vs. WAN). We show that each of these deployment scenarios exhibits distinct performance characteristics in the context of secure inference systems. Furthermore, two of these scenarios open up new avenues for innovation in system design (see "Hetero" below).

In creating a fast secure inferencing system that is adaptable to diverse deployment scenarios through protocol-system co-design, we start by focusing on 2PC protocols under the pre-processing model. Further, to make discussions concrete, we focus on CNN inference. However, the techniques proposed are applicable beyond CNNs, including transformers (see Appendix K) as well as to training. While CNNs have two types of layers – linear layers, e.g., matrix multiplication; and non-linear layers, e.g., ReLU, over (> 90%) of time in secure inferences is attributable to the non-linear layers. We, thus, focus on securely computing non-linear layers.

LSS vs FSS. There are two broad classes of cryptographic protocols that can be used to compute non-linear layers – Linear Secret Sharing (LSS) [31, 42, 52, 76], and Function Secret Sharing (FSS) [13, 34, 64]. LSS-based protocols communicate more bytes and more frequently (rounds) compared to FSS-based protocols. On the other hand, FSS-based protocols need more compute (AES calls) and larger keys (storage) compared to LSS-based protocols. Theoretically, we expect LSS-based protocols to stress the network and FSS-based protocols to stress the compute and storage. However, observing this in practice relies on performant implementations, which, as we will show in Section 7, is not necessarily true for prior LSS-based systems. Hence, to quantify their differences under different deployment scenarios, we first create LSS and FSS-based protocol suites for secure CNN inference (described later) that beat state-of-the-art in their respective protocol class. We call these LSS^{M} and FSS^{M} respectively. We notice that, across different models, LSS^{M} communicates $\approx$ 2.5× as much as FSS^{M} over 3× as many rounds, while FSS^{M} needs keys that 25-27× larger than LSS^{M}'s. Naturally, when the network is slow (e.g., WAN), and keys are in memory, FSS^{M} beats communication-heavy LSS^{M} by 2.2 – 2.5×. However, when the network is *fast* (e.g., LAN), and keys are in storage (previously unexplored), LSS^{M} beats FSS^{M} (and all prior works) by 18× or more. In short, we demonstrate that *one size (protocol) does not fit all (deployments)*. One must choose different protocols for different deployment scenarios.

Hetero. We then discover a hitherto unexplored opportunity to leverage *heterogeneous processing*. We notice that in certain deployment scenarios such as when parties are connected over LAN and keys are in memory, or when they are connected over WAN but keys are in storage, both LSS^{M} and FSS^{M} perform similarly (Figure 2). Further, computation-heavy FSS^{M} benefits significantly from the large computational power of Graphics Processing Units (GPUs). In contrast, LSS^{M}, being inherently communication-heavy, has limited usefulness of GPUs, once optimized to leverage advanced vectorization features of modern CPUs. Thus, while FSS^{M} can leverage GPU to compute non-linearities, LSS^{M} can rely *only* on the CPU to significantly boost the throughput of inference serving. In short, one could *simultaneously* harness both CPU and GPU computing, i.e., heterogenous processing, through carefully matching protocols with hardware capabilities to achieve throughputs that are not possible to attain using only the CPU or the GPU for computing.

Matchmaker. To ease the burden of manually choosing the *right* protocol or a combination thereof, in varying deployment scenarios, we create a software tool *Matchmaker* (MM). It uses profile-guided modeling to *automatically* divide work across LSS^{M} and FSS^{M} under any deployment in the true spirit of protocol-system co-design. By judiciously choosing protocols across all scenarios, MM beats the state-of-the-art in secure inference (Orca) by up to 21× (Section 7.3).

LSS^{M} and FSS^{M}. One of our key contributions is the creation of highly optimized *new* state-of-the-art LSS and FSS-based protocols, LSS^{M} and FSS^{M}, that form the backbone of MM. They harness both protocol and hardware-aware optimizations to attain significant speedups over their respective state-of-the-art.

We observe that the majority of time in LSS-based non-linear layers can be attributed to secure comparison. We provide a new protocol for comparison that leverages the structure of its tree-like boolean circuit [42, 76] to ① use correlated Beaver bit-triples that reduce key size, and ② optimize ANDs at the leaves to reduce communication (Section 3.1). Importantly, LSS^{M} introduces hardware-aware optimizations to speed up secure comparison. We carefully harness both vector compute and vector memory instructions on modern CPUs. Vector instructions allow simultaneous execution of the same operation (e.g., AND) on different data elements (i.e., data parallel). We leverage data parallelism *across* concurrently executing comparison circuits to fully benefit from the wide vector instructions (256/512 bits) of today's CPUs. However, this requires the reorganization of input data, which, if not performed efficiently, can eclipse the benefits of vectorizing the compute (ANDs/XORs). We then observe that input reorganization can be efficiently performed by leveraging vector memory (gather) instructions. Lastly, protocols in LSS^{M} have been designed to work with small keys such that even in a fast LAN setting, the time to fetch the keys from storage to memory can be hidden behind online computation time once keys are in memory. In particular, the key size of LSS^{M} is up to 69× smaller than prior LSS-based systems and 29× smaller than state-of-the-art in secure inference, i.e., Orca. While the key size can be reduced further using techniques from silent pre-processing literature, these are known to add significant overhead to online compute time that would be detrimental [14, 77].

Overall, LSS^{M} beats state-of-the-art LSS-based secure inference systems by up to 29× in communication (Section 7.1) and by up to 31× in latency even in the well-studied setting of LAN and keys in memory. Notice that the highly optimized realization of LSS^{M} using *only* the CPU leaves the GPU for FSS^{M}, paving the path for heterogeneous processing.

Finally, we enhance state-of-the-art FSS-based Orca [38] to use a more efficient comparison scheme [67]. We also incorporate other optimizations (Section 4) to create our FSS-based secure inference system FSS^{M}, which is faster than Orca by up to 2.2× (Section 7.1.4). To summarize, our contributions are:

- An optimized LSS-based inference system, LSS^{M}, that runs non-linearities on a CPU, and still beats the state-of the-art GPU accelerated LSS-based systems by up to 50×. LSS^{M} has 29-69× smaller

key size than state-of-the-art, resulting in significant speedups (up to 50×) in the critical high-throughput deployment.
- Isolating two critical considerations in practical deployments of secure inference systems – the location of keys (memory or storage) and network speed, we show that there is no *one size fits all* protocol between LSS^{M} and FSS^{M} across deployment scenarios.
- Recognizing the opportunity to leverage *heterogeneous processing* by simultaneously running LSS^{M} on the CPU and FSS^{M} on the GPU to boost performance.
- Matchmaker leverages our insights and automatically picks the best combination of protocols. MM beats state-of-the-art secure inference by up to 21× (geomean 3.25×).

2 Preliminaries

Notation: Let λ be the computational security parameter. For a positive integer n, let $N = 2^n$. We denote the set of n-bit unsigned integers by $\mathbb{U}_N$. We denote the set of integers by $\mathbb{Z}$. Arrays are denoted by boldface, e.g. $\mathbf{e}$, and the i^{th} element of $\mathbf{e}$ is denoted by $\mathbf{e}[i]$. We use 0-based indexing for arrays. For a predicate p, $\mathbf{1}\{p\}$ is an indicator function which returns 1 if p is true and 0 otherwise. We use $x \xleftarrow{\$} \mathbb{U}_N$ to denote that x has been sampled uniformly at random from $\mathbb{U}_N$.

Operators. For $x \in \mathbb{U}_N$, we write $\text{int}_n(x)$ when we wish to interpret x as an n-bit signed integer in 2's complement representation. We use $\text{MSB}(x)$ to denote the most significant bit of x. For $m > n$, we use $\text{extend}(x, m)$ to denote the operation of prefixing $m - n$ 0s to x. We use $\gg$ to mean logical right shift and $\gg_A$ to mean arithmetic right shift. For an array $\mathbf{e}$, we use $\mathbf{e} \ggg i$ to denote cyclically rotating the elements of $\mathbf{e}$ by i places to the right. We denote logical XOR by $\oplus$ and logical AND by $\wedge$. We use $||$ to denote concatenation.

Fixed-point representation. A real number x is converted to fixed-point representation with bitwidth n and precision f as $\lfloor x \cdot 2^f \rfloor \mod N$. A fixed-point number x with bitwidth n and precision f is converted to a real number as $\frac{\text{int}_n(x)}{2^f}$.

2.1 Linear Secret Sharing (LSS) Schemes

Arithmetic Secret Sharing. For $x \in \mathbb{U}_N$, arithmetic secret sharing randomly samples $x_0, x_1 \xleftarrow{\$} \mathbb{U}_N$ such that $x_0 + x_1 = x \mod N$. We denote the process of secret sharing x by share x. For $b \in \{0, 1\}$, the share of party P_b is denoted by x_b. We refer to the process of parties exchanging their shares and adding them to recover the underlying value by reconstruct (x_b).

Boolean secret sharing. When $x \in \{0, 1\}$ (or when $x \in \mathbb{U}_2$), we can also get *boolean* shares, i.e. random bits x_0, x_1 such that $x_0 \oplus x_1 = x$. Logical XOR is simply addition modulo 2.

2.2 Protocol Structure and Threat Model

2PC with preprocessing. We consider 2PC in the preprocessing model [10, 11, 17, 26, 37], which has been considered by many recent works on secure inference [33, 38, 76]. In this model, parties P_0 and P_1 with private inputs x_0 and x_1 securely compute, through a protocol, a publicly known function f of their inputs. In the context of secure neural network inference, x_0 refers to the private weights of the model, x_1 is the input on which to evaluate the model, and f is the structure of the neural network - however, as is common in all prior works, x_0 and x_1 will be arithmetic shares of the private weights and input of the two parties; the function securely evaluated will then first reconstruct these shares internally and then compute f on it. A protocol Π^f for a function f is a pair of algorithms $(\text{Gen}^f, \text{Eval}^f)$. Gen^f, which depends only on f and not any of the inputs, is run by a trusted dealer in a pre-processing phase and generates a pair of correlated random strings (also called *keys*) denoted by (k_0^f, k_1^f). Generic or specialized 2PC protocols can emulate the trusted dealer. In the online phase, the dealer is no longer involved, and for $b \in \{0, 1\}$, party P_b runs $\text{Eval}(b, k_b^f, x_b)$ to get $f(x)_b$, which is its share of the output $f(x)$. We denote the size of the key per party for Π^f by $\text{keysize}(\Pi^f)$, the total number of bits communicated (by both parties) by $\text{comm}(\Pi^f)$, and the number of rounds of communication by $\text{rounds}(\Pi^f)$. In this work, we focus on the online phase and *not* on Gen^f.

Security. Our protocols are proven simulation secure in the ideal/real paradigm [19, 48], with security proven against one semi-honest corruption. Informally, security implies that the protocol computation does not leak anything about x_0 to P_1 (and similarly about x_1 to P_0) beyond what is implied by the function output, $f(x_0, x_1)$, as long as P_0 (and similarly P_1) follow the protocol specification faithfully (semi-honest behaviour). We will construct protocols for various functions in which parties begin the protocol with secret shares of the inputs to the function and end the protocol with secret shares of the output to the function. This will allow us to sequentially compose different protocols. By proving the standalone security of protocols for various functions such as matrix multiplications, convolutions, ReLU, and so on, and by invoking the sequential composition theorem [19], we can prove the security of the entire end-to-end protocol for secure inference. Security of our standalone protocols can be proved in the hybrid model [19] following the template in Appendix D.

2.3 Protocols common to LSS and FSS

We consider protocols for 2PC in the pre-processing model based both on Linear Secret Sharing (LSS) [31] schemes as well as Function Secret Sharing (FSS) [13, 17] schemes. We now describe existing protocols for commonly occurring functionalities in secure ML. These protocols are the same for LSS and FSS-based 2PC protocols.

Boolean to arithmetic secret shares. For $s \in \{0, 1\}$, we define $\text{B2A}_n(s) = \text{extend}(s, n) \in \mathbb{U}_N$. It is easy to construct a protocol $\Pi_n^{\text{B2A}} = (\text{Gen}_n^{\text{B2A}}, \text{Eval}_n^{\text{B2A}})$ that given secret shares of a bit s, returns secret shares of $\text{B2A}_n(s)$ (see Appendix B). Π_n^{B2A} has keysize n and communicates 2 bits in a single round.

Matrix Multiplications and Convolutions. Matrix multiplications and convolutions can easily be realized using a generalization of Beaver triples [10]. For a bilinear function $f : \mathbb{U}_N^p \times \mathbb{U}_N^q \to \mathbb{U}_N^r$ where p, q, r are positive integers, the corresponding Beaver-triple based protocol has keysize $(p+q+r) \cdot n$. It communicates $2 \cdot (p+q) \cdot n$ bits in a single round. As a special case, to compute multiplication of secret-shared inputs $x, y \in \mathbb{U}_N$, we require a key of size $3n$ and $2n$ bits of communication in a single round. To compute AND of secret shared bits, we require a key of size 3 bits and 2 bits of communication in a single round.

Select. The functionality $\text{select}_n : \mathbb{U}_N \times \{0,1\} \rightarrow \mathbb{U}_N$ takes as input an n-bit value x and a selector bit s and returns $s \cdot x$. Orca [38] provides a protocol $\Pi_n^{\text{select}} = (\text{Gen}_n^{\text{select}}, \text{Eval}_n^{\text{select}})$ to securely realize select_n. It has keysize $3n$ and communicates $2n+2$ bits in a single round. For completeness, we present $\Pi_n^{\text{select}} = (\text{Gen}_n^{\text{select}}, \text{Eval}_n^{\text{select}})$ in Appendix C.

2.4 Secure CNN Inference

Convolutional Neural Network (CNN) inference makes use of two kinds of operations – linear operations, e.g., convolutions and matrix multiplications, and non-linear operations, e.g., ReLU and Maxpool. While plaintext ML works over floating-point numbers, secure ML works over fixed-point numbers for efficiency [38, 45, 52, 61]. Fixed-point numbers with bitwidth n can easily be mapped to the set of n-bit unsigned integers ($\mathbb{U}_N$). Thus, linear operations can be computed using the protocols outlined in Section 2.3. However, when linear operations multiply two n-bit fixed-point numbers with precision f, this results in a product with precision $2f$. To return to precision f, we require a *truncation* operation, which drops the last f bits. Truncation is a non-linear operation. CNNs also contain non-linear activations such as ReLU. For fixed-point numbers x, y, $\text{ReLU}(x) = x \cdot \mathbf{1}\{x > 0\}$, and we compute $\max(x, y)$ required in Maxpool as $\text{ReLU}(x - y) + y$. Non-linear operations have *different* protocols across LSS and FSS, and we focus on these.

Network-level optimizations for fixed-point CNNs. Orca [38] showed how to change the architecture of fixed-point CNNs such that underlying functionality is identical to the original CNN, but the cost of computing it securely is reduced. At a high level, Orca is efficient because it works over bitwidths smaller than the fixed-point bitwidth n wherever possible and reuses the output of expensive computations by *fusing* functionalities. We refer the reader to [38] for details. As a result of applying Orca's optimizations to our models, we require LSS and FSS-based protocols for a new fused functionality ReLU-Extend, which takes an $(n-f)$-bit number x as input and returns $\text{ReLU}(x)$ in n-bits as output.

3 LSS$^{\text{M}}$: Optimized LSS for Matchmaker

As discussed in Section 2.4, we focus on computing non-linear functionalities. Towards this, we first describe our *novel* LSS-based protocol for the Millionaires' problem [79] i.e., comparison on secret inputs (Section 3.1). We use this as a building block for the various ML functionalities outlined in the previous section. One of our key contributions is the first *secure* LSS-based protocol for stochastic truncation (Section 3.2). We also provide the first secure LSS-based protocols for the other non-linear functionalities defined in Orca [38]. To save space, we delegate these to Appendices E-G. Notably, we provide new plaintext logic for the functionality ReLU-Extend, which improves over Orca's logic by needing fewer comparisons (Appendix G).

In Section 3.3, we discuss how we efficiently implement our protocol for the Millionaires' problem by using vectorization to accelerate computation on *CPUs* without needing to rely on GPUs unlike prior work [42, 76]). We build an end-to-end system for secure inference based on our LSS-based protocols and efficient CPU-based comparison and call it LSS$^{\text{M}}$. In Section 7.1, we compare the performance of LSS$^{\text{M}}$ with state-of-the-art systems based on LSS, Piranha [76] and CrypTen [42]. We show that LSS$^{\text{M}}$ is better by at least an order of magnitude in latency and communication.

3.1 Millionaires' and Wrap

In the Millionaires' problem, P_0 and P_1 have secret inputs $x, y \in \mathbb{U}_N$, respectively, and wish to compute boolean shares of $\text{Lt}_n(x, y) = \mathbf{1}\{x < y\}$. To compute Lt_n, we construct a tree-like boolean circuit with AND and XOR gates. While our circuit follows that of Cryptflow2 [61] and others [30], we compute the AND and XOR gates using protocols in the preprocessing model (Section 2.3). Let $x = x_1 || x_0$ and $y = y_1 || y_0$ be such that x_0, y_0 are $\lceil \frac{n}{2} \rceil$-bit strings and x_1, y_1 are $\lfloor \frac{n}{2} \rfloor$-bit strings. Then,

$$\mathbf{1}\{x < y\} = \mathbf{1}\{x_1 < y_1\} \oplus \mathbf{1}\{x_1 = y_1\} \wedge \mathbf{1}\{x_0 < y_0\}$$
$$\mathbf{1}\{x = y\} = \mathbf{1}\{x_1 = y_1\} \wedge \mathbf{1}\{x_0 = y_0\}$$

Using these relations recursively, we can reduce comparison and equality on n-bit strings to comparisons and equality on smaller strings, resulting in a tree-like circuit of depth $\lceil \log n \rceil$. When $x, y \in \{0,1\}$ (the base level of the recursion), we have,

$$\mathbf{1}\{x < y\} = (x \oplus 1) \wedge y; \mathbf{1}\{x = y\} = x \oplus y \oplus 1$$

Overall, for n-bit comparisons, we obtain a boolean circuit with $\approx 3n$ AND gates and depth $\lceil \log n \rceil$, where n AND gates are at the leaf level to compute $\mathbf{1}\{x_i < y_i\}$ for each input bit. We further optimize this circuit before realizing it with pre-processed bit-triples.

First, we observe that the AND gates at the leaf, that is, at the base of the recursion, take secret values known to each of the parties respectively as input and not secret shares of them. Hence, we can optimize our protocol for AND so the key size required per leaf node is 2 bits, and the communication required is 2 bits in 1 round. Second, for recursion steps, both comparison and equality need one AND gate each. However, one of the inputs to the AND gates is same, and hence, we can generate correlated beaver triples[2] as $\{u, v_1, w_1\}$ and $\{u, v_2, w_2\}$ such that $w_1 = u \wedge v_1$ and $w_2 = u \wedge v_2$. We also save on online communication and need 6 bits of total communication per internal node compared to 8 bits needed naively. Finally, we skip computing the equalities on the rightmost path in the tree on the least significant chunks of values (as these are never used).

We design our comparison circuit to reduce the size of the correlated randomness (keys) required to compute it securely. A practical deployment consideration drives this – we noticed that previously proposed LSS-based frameworks suffer significant slowdowns when they must fetch keys from storage in the critical path of computing (Section 7.1). Thus, we strive to design LSS$^{\text{M}}$ to have small keys. We avoid using circuits with many input AND gates, e.g., the one in ABY2.0 [56], which slightly lower communication but at the cost of a much larger key. For 64-bit comparison, ABY2.0 reduces communication by 20% but has 2× larger keys.
We summarize the cost of our Millionaire's protocol below.

Theorem 1. *There exists a protocol* $\Pi L_n^{\text{Mill}} = (\text{Gen}L_n^{\text{Mill}}, \text{Eval}L_n^{\text{Mill}})$ *that securely computes* Lt_n *with* $\text{keysize}(\Pi L_n^{\text{Mill}}) = 7n - 2\lceil \log n \rceil$, $\text{comm}(\Pi L_n^{\text{Mill}}) = 8n - 2\lceil \log n \rceil - 2$ *and* $\text{rounds}(\Pi L_n^{\text{Mill}}) = \lceil \log n \rceil + 1$.

[2] CryptFlow2 [61] made similar observation in 2PC context to reduce the cost of OTs.

Stochastic Truncate-Reduce $\Pi L^{\mathsf{stTR}}_{n,f}$

$\mathsf{Gen}L^{\mathsf{stTR}}_{n,f}$:

1: $r \xleftarrow{\$} \mathbb{U}_{2^f}$; share r
2: $w = \mathsf{extend}(\mathsf{wrap}_f(r_0, r_1), n-f)$
3: share w
4: $(k_0^{\mathsf{wrap}}, k_1^{\mathsf{wrap}}) \leftarrow \mathsf{Gen}L^{\mathsf{wrap}}_f$
5: $(k_0^{\mathsf{B2A}}, k_1^{\mathsf{B2A}}) \leftarrow \mathsf{Gen}^{\mathsf{B2A}}_{n-f}$
6: For $b \in \{0,1\}$, $k_b = r_b || k_b^{\mathsf{wrap}} || k_b^{\mathsf{B2A}} || w_b$

$\mathsf{Eval}L^{\mathsf{stTR}}_{n,f}(b, k_b, x_b)$:

1: Parse k_b as $r_b || k_b^{\mathsf{wrap}} || k_b^{\mathsf{B2A}} || w_b$
2: $z_b = x_b \mod 2^f$
3: $y_b = z_b + r_b \mod 2^f$
4: $v^{(b)} = \mathsf{TR}(x_b, f) + \mathsf{extend}(\mathsf{wrap}_f(z_b, r_b), n-f)$
5: $p_b \leftarrow \mathsf{Eval}L^{\mathsf{wrap}}_f(b, k_b^{\mathsf{wrap}}, y_b)$
6: $p'_b \leftarrow \mathsf{Eval}^{\mathsf{B2A}}_{n-f}(b, k_b^{\mathsf{B2A}}, p_b)$
7: **return** $z_b = v^{(b)} + p'_b - w_b$

Figure 1: LSS-based protocol for $\mathsf{stTR}_{n,f}$

Wrap. In subsequent protocols, we use Millionaires' protocol to compute the *wrap* bit which checks if the private input $x \in \mathbb{U}_N$ of P_0 and the private input $y \in \mathbb{U}_N$ of P_1 are such that $x + y > 2^n - 1$ over $\mathbb{Z}$. Formally, for $x, y \in \mathbb{U}_N$ we define the functionality $\mathsf{wrap}_n(x, y)$ that reduces to Lt_n as follows:

$$\mathsf{wrap}_n(x, y) = \mathbf{1}\{x + y > 2^n - 1\} = \mathsf{Lt}_n(2^n - 1 - x, y)$$

Thus, LSS-based protocol for wrap_n, denoted by $\Pi L^{\mathsf{wrap}}_n = (\mathsf{Gen}L^{\mathsf{wrap}}_n, \mathsf{Eval}L^{\mathsf{wrap}}_n)$, is simply ΠL^{Mill}_n with $(2^n - 1 - x)$ as P_0's input and y as P_1's input. We provide security proofs for our protocols for Millionaire's and Wrap in Appendix D.

3.2 Stochastic truncations

Truncations are used to reduce the scale of fixed-point values to avoid overflows after a multiplication operation. Prior works have used two kinds of truncations - faithful and stochastic. In stochastic truncations, the output is rounded up or down with a probability depending on the value of the truncated part. Prior works using LSS, such as Piranha [76], and CrypTen [42], used fast local operations to emulate stochastic truncations that have been shown to be insecure [47]. Orca [38] provided a secure *FSS*-based protocol for stochastic truncation. We provide the first secure protocol for *LSS*-based stochastic truncations. We need two kinds of operations to reduce the scale of fixed-point values – stochastic truncation (bitwidth-preserving) and stochastic truncate-reduce (bitwidth-reducing). We describe our protocol for stochastic truncate-reduce. Orca showed that stochastic truncation can be computed as stochastic truncate-reduce followed by signed-extension (Lemma 2 in [38]). Following the same, Appendix F.2 details how we build stochastic truncation based on our stochastic truncate-reduce.

3.2.1 Stochastic Truncate-reduce. Let truncate-reduce, $\mathsf{TR}_{n,f}$, be a functionality that drops the lower f bits of an n-bit value, i.e., for $x \in \mathbb{U}_N$, $\mathsf{TR}_{n,f}(x) = (x \gg f) \mod 2^{n-f} \in \mathbb{U}_{2^{n-f}}$.

Definition 1. *For $x \in \mathbb{U}_N$, $z = x \mod 2^f$, stochastic truncate-reduce by f, denoted by $\mathsf{stTR}_{n,f}(x)$ is defined as*

$$\mathsf{stTR}_{n,f}(x) = \begin{cases} \mathsf{TR}_{n,f}(x) & \text{with probability} \quad 1 - z \cdot 2^{-f} \\ \mathsf{TR}_{n,f}(x) + 1 & \text{with probability} \quad z \cdot 2^{-f} \end{cases}$$

Equivalently, stochastic truncate-reduce of x can be computed by first sampling t that is 1 with probability $z \cdot 2^{-f}$ and 0 otherwise. Then $\mathsf{stTR}_{n,f}(x) = \mathsf{TR}_{n,f}(x) + t$. Moreover, for a random $r \in \mathbb{U}_{2^f}$, $t \equiv \mathbf{1}\{z + r > 2^f - 1\} = \mathsf{wrap}_f(z, r)$. This is because there are exactly z values of r for which $z + r > 2^f - 1$.

To compute stochastic truncate reduce securely, we prove the following lemma[3] in Appendix F.1:

Lemma 1. *Let $x_0, x_1, x \in \mathbb{U}_N$ be such that $x = (x_0 + x_1) \mod N$ and $r_0, r_1, r \in \mathbb{U}_{2^f}$ be such that $r = (r_0 + r_1) \mod 2^f$. Let $z = x \mod 2^f$, and, for $b \in \{0,1\}$, let $z_b = x_b \mod 2^f$ and $y_b = z_b + r_b \mod 2^f$. Then,*

$$\mathsf{TR}_{n,f}(x) = \mathsf{TR}_{n,f}(x_0) + \mathsf{TR}_{n,f}(x_1) + \mathsf{wrap}_f(z_0, z_1) \quad \text{over } \mathbb{Z}$$
$$\mathsf{wrap}_f(z, r) = \mathsf{wrap}_f(z_0, r_0) + \mathsf{wrap}_f(z_1, r_1) + \mathsf{wrap}_f(y_0, y_1) - \mathsf{wrap}_f(z_0 + z_1) - \mathsf{wrap}_f(r_0 + r_1) \quad \text{over } \mathbb{Z}$$

Hence, it follows that, over $\mathbb{Z}$,

$$\mathsf{stTR}_{n,f}(x) = \mathsf{TR}_{n,f}(x_0) + \mathsf{wrap}_f(z_0, r_0) + \mathsf{TR}_{n,f}(x_1) + \mathsf{wrap}_f(z_1, r_1) - \mathsf{wrap}_f(r_0, r_1) + \mathsf{wrap}_f(y_0, y_1)$$

In the final expression for stTR, the sum of the first two terms can be computed locally by P_0, while the sum of the third and fourth terms can be computed locally by P_1. For an $r \in \mathbb{U}_{2^f}$ known to the dealer, $w = \mathsf{extend}(\mathsf{wrap}_f(r_0, r_1), n-f)$ is computed by the dealer and secret shared between the two parties. In the online phase, parties run ΠL^{wrap}_f to compute shares of $p = \mathsf{wrap}_f(y_0, y_1)$ and subsequently Π^{B2A}_{n-f} to compute shares of $p' = \mathsf{extend}(p, n-f)$. We describe our protocol in Figure 1 and summarize its cost below.

Theorem 2. $\Pi L^{\mathsf{stTR}}_{n,f}$ *realizes* $\mathsf{stTR}_{n,f}$ *securely with* $\mathsf{comm}(\Pi L^{\mathsf{stTR}}_{n,f}) = \mathsf{comm}(\Pi L^{\mathsf{wrap}}_f) + 2$, $\mathsf{keysize}(\Pi L^{\mathsf{stTR}}_{n,f}) = \mathsf{keysize}(\Pi L^{\mathsf{wrap}}_f) + \mathsf{keysize}(\Pi^{\mathsf{B2A}}_{n-f}) + n$, *and* $\mathsf{rounds}(\Pi L^{\mathsf{stTR}}_{n,f}) = \mathsf{rounds}(\Pi L^{\mathsf{wrap}}_f) + 1$.

3.3 Accelerating Comparison on CPU

We observe that the practical usefulness and performance potential of even a well-designed cryptographic protocol may remain unrealized without a holistic system design. Here, we demonstrate how the structure of $\mathsf{LSS}^{\mathsf{M}}$'s computation can effectively harness the wide vectorization capabilities [3, 4] of modern CPUs through a careful protocol-system co-design.

A majority of the computational cost for non-linear operations such as ReLU can be attributed to secure comparison. For example, when securely computing 1M 64-bit ReLUs on the CPU, both CrypTen [42] and MP-SPDZ [40] spend more than 90% of their

[3]This lemma is inspired from Lemma 1 in Orca [38] for FSS-based protocols but needs to be modified to work with LSS.

time on secure comparison. We describe how we map the structure of secure comparison to harness vector compute and memory instructions on the CPU.

Structure of computation for secure comparison. Consider comparing two n-bit numbers $x, y \in \mathbb{U}_N$. For $i \in \{0, \cdots, n-1\}$ and $x_i, y_i \in \{0,1\}$, parse x, y as $x = x_0||x_1||\cdots||x_{n-1}$ and $y = y_0||y_1||\cdots||y_{n-1}$. We recall that comparison has two high-level computations, less-than and equality. To compute equality, we start at the leaf level by computing equality of 1-bit inputs, then go up the comparison tree to compute equality of 2-bit inputs, then 4-bit inputs, and so on. For $i \in \{0, \cdots, n-1\}$, define $e_i = \mathbf{1}\{x_i = y_i\}$. After computing 1-bit equality, we get (shares of) a tightly-packed $(n-1)$-bit vector $\vec{e} = e_1||e_2||\cdots||e_{n-1}$. At the next level, we compute 2-bit equality as $e_1 \wedge e_2$, $e_3 \wedge e_4$, etc.

Effectively harnessing vector instructions for the above computation structure is challenging. A typical way to efficiently compute on a CPU is to process values that lie close together in the memory at the same time. This preserves memory access locality and thus, benefits from the CPU's deep cache hierarchy. However, computing on neighboring bits in the above-mentioned computation structure while also leveraging vectorization poses two challenges.

Challenge ① Insufficient parallelism in a single comparison. Vector instructions operate over 128, 256, or even 512-bit inputs. n is typically much smaller (e.g. 39), so even at the leaf level of the tree, we cannot generate enough AND/XORs to fully exploit the hardware. It reduces further as we go up the tree.

Challenge ② Reorganizing input layout. Since $\vec{e}$ is a tightly packed bit-vector, while computing 2-bit equality as $e_1 \wedge e_2$, $e_3 \wedge e_4$, etc, we are computing local ANDs of *adjacent* bits that are stored in the *same* register. Vector (compute) instructions require the left and right operands of the ANDs to be stored in *separate* registers. So if we are to compute $e_1 \wedge e_2, e_3 \wedge e_4$, etc. via the same vector AND, we need to reorganize the input and separate the odd and even elements of $\vec{e}$. This reorganization requires conditional execution whereby the even and odd bits of $\vec{e}$ are treated differently. Such irregular execution is expensive as it deters leveraging vector memory instructions for reorganizing. To quantify this cost, we implemented this reorganization and found that it took > 75% of the total time of secure comparison, severely limiting the benefit of any subsequent vectorization of computation (of ANDs/XORs).

Our technique. Since fully harnessing vector instructions within a comparison circuit is difficult, we instead vectorize *across comparisons*. In secure ML, many thousands of secure comparisons are performed in parallel (e.g., for ReLU). We exploit this to vectorize comparison. Consider computing M comparisons of n-bit numbers. This requires computing M copies of our comparison circuit. Instead of collecting ANDs from within a comparison for vectorization (e.g. $e_1 \wedge e_2$ and $e_3 \wedge e_4$), we club *corresponding* ANDs from multiple comparisons together. For equality, instead of computing $e_1 \wedge e_2$ and $e_3 \wedge e_4$ via the same vector AND, we compute $e_1 \wedge e_2$ *from different comparisons* via the same vector AND. To enable this, we first perform a bit-decomposition that clubs the 1^{st} bit of M inputs together, then the 2^{nd} bit, then the 3^{rd} bit, and so on. After bit-decomposition, instead of computing $\vec{e} = e_1||e_2||\cdots||e_{n-1}$ as a tightly-packed bit-vector, we instead compute e_1 *for all M comparisons* as a tightly-packed bit-vector. The same holds for e_2. While computing 2-bit equality, we compute $e_1 \wedge e_2$ for all M comparisons together. This computation can be vectorized as-is without needing reorganization. Additionally, there are at least M local AND/XORs for all levels of equality (from 1-bit to n-bit). Since M is very large, this allows enough parallelism to leverage vector instructions.

Unfortunately, even though this would vectorize the compute in the comparison circuit, bit decomposition is expensive. As in Challenge ①), the overhead of reorganization of the inputs (here, bit decomposition) can eclipse the benefits of vectorizing the compute (ANDs/XORs). We then observe that decomposing bits corresponds to *transposing* a bit-matrix. For M n-bit comparisons, we think of each party's input as an $M \times n$ bit-matrix, which we transpose to get an $n \times M$ bit-matrix. Fortunately, unlike the costly reorganization that would have been necessary for vectorization within a comparison, transpose is a *uniform* operation – it affects each bit in the same way without conditionals. Hence, transpose lends itself well to vectorized memory and compute instructions. Each CPU thread computes the transpose of a 32×32 sub-matrix. The size of the sub-matrix is chosen for better cache locality. For efficiently reading sub-matrices from memory, we use vector load instructions (`_mm256_i32gather_epi32`). We then vectorize the computation of the transpose using vector shift and XOR instructions (`_mm256_sllv_epi32`, `_mm256_xor_si256`). These optimizations limit the time for the transpose to < 4% of the total time.

In summary, it is imperative to vectorize *both* the computation of the circuit and the input reorganization to efficiently perform secure comparisons on a CPU. While we focused on comparison, these observations apply to other circuits too.

Why not use a GPU? Our vectorized comparison is communication-bound even on a fast LAN. For example, to process 1M 64-bit comparisons, 28 ms of 35 milliseconds total, i.e., 80% of the time is spent on communication. This fraction is even higher on a slow WAN. GPUs can only accelerate computation and not communication. Thus, deploying a GPU *cannot* speed up our vectorized secure comparison by > 25%. When executing LSS-based protocols for non-linear layers, our efficient comparison ensures that 52 – 65% of the time is spent on communication *even on the CPU*.

3.4 Key compression

Prior works [33, 38, 76] have assumed that keys are always available in memory (DRAM). However, in a practical deployment, if inference requests arrive rapidly or arrive in bursts, keys may need to be fetched from storage to the memory before the secure computation can proceed. Even though LSS keys are small relative to FSS, the time to read them from storage is *still* 3.6-4.5× more than the time required for computation when the parties are connected over LAN. To further reduce the size of LSS keys to improve end-to-end latency, we compress them using well-known Pseudorandom Function (PRF)-based techniques. Let F be a PRF. The dealer shares PRF keys k_0, k_1 with parties P_0, P_1 in the offline phase. The dealer then avoids explicitly sending each party its *entire* key. Instead, for $b \in \{0,1\}$, party P_b makes PRF calls that are identical to the ones made by the dealer to generate a part of its key in the online phase. We illustrate how this works for Beaver bit-triples in Appendix H. With this optimization, the key size reduces by 4.6×, 4×, up to 3.7× and up to 3.9× in Millionaires'/Wrap, ReLU/Maxpool, stochastic

truncate-reduce, and in ReLU-extend, respectively. Note that we trade a smaller key for slightly more online computation in the form of PRF calls. However, this computational overhead is small (10-16%) compared to the large reduction in key size (4.5-5.5×).

4 FSS^M : Improving FSS-based Orca

Orca [38] is the state-of-the-art in secure CNN inference in the 2PC with pre-processing model and is based on Function Secret Sharing [15–17]. We improve on Orca using ideas from our LSS-based protocols in Section 3 and improved FSS-based comparison from Grotto [67]. We refer to the resulting FSS-based 2PC as FSS^M and provide an empirical comparison with Orca in Section 7.1.4. We demonstrate that FSS^M has keys that are 7-8% smaller, requires up to 2.3× lower communication and is up to 2.2× faster.

In FSS^M, we use the same model-level optimizations as Orca. We need protocols for stochastic truncate-reduce/truncations, ReLU, ReLU-Extend, and Maxpool. In Orca [38], each of these protocols relied on secure comparisons, that were realized using *Distributed Comparison Functions* (DCF) [16]. We make four improvements in FSS^M over Orca. ① In FSS^M, we rely on *Distributed Point Function* (DPF)-based comparisons as suggested in Grotto [67]. While this switch can result in lower keysize and > 2× reduction in compute, it can only support output group $\mathbb{G}^{\text{out}} = \{0, 1\}$. All protocols in Orca except ReLU-Extend require a single-bit output from secure comparison and hence, for those, this switch is easy to make. ② Building on our ideas for LSS-based ReLU-Extend, we design a new protocol for *DPF*-based ReLU-Extend where we only require comparisons with one-bit outputs. ③ We reduce the communication of stochastic truncate-reduce by f from $2n$ to $n + f$ bits by having one party reconstruct only a part of the input. ④ We extend Orca [38] to support packing for non-power-of-2 bitwidths. This reduces communication for models, e.g., ResNet50, that requires a bitwidth of 37 to preserve accuracy. Due to space constraints, we defer the details of our FSS-based protocols to Appendix I.

5 A case for Matchmaker

In Section 7, we quantitatively establish that LSS^M and FSS^M outperform prior LSS and FSS-based secure inference systems, respectively, thanks to optimizations in Sections 3 and 4. Further, current literature suggests that FSS-based protocols [38] *always* outperform LSS-based protocols [42, 76]. However, when comparing LSS^M with FSS^M, we notice that this notion could be misplaced.

We make a novel observation that *one protocol does not fit all (deployments).* There are two primary considerations in deployments: whether the keys are available in memory or in storage, and whether the parties are connected via a fast LAN or a slow WAN. Before the online computation starts, keys are written to the storage in the pre-processing step. If the incoming requests are spread sparsely over time, there could be enough slack to fetch the keys from storage to the memory before the computation starts. However, at a high and/or bursty request arrival rate, keys must first be fetched from storage to memory in the critical path of the execution. Further, the computing parties (servers) may be hosted on the same datacenter and, thus, connected over a high-speed LAN. Parties could also be geographically distributed across datacenters or even continents, connected over slow WAN.

Model Batch=50	Key size (GB)		Comm (GB)		Rounds	
	LSS^M	FSS^M	LSS^M	FSS^M	LSS^M	FSS^M
ResNet-18	2.02	54 (27×)	6.2 (2.6×)	2.4	381 (3.3×)	116
ResNet-50	9.1	246 (27×)	26 (2.4×)	11	932 (3.3×)	279
VGG-16	9.5	236 (25×)	27 (2.5×)	11	362 (3.1×)	107

Table 1: Comparing LSS^M and FSS^M on batch inference.

LSS^M and FSS^M have inherently different characteristics. While FSS^M has lower communication and fewer rounds, it needs a much larger key size compared to LSS^M. Hence, when parties are connected over WAN and keys reside in memory, FSS^M enjoys an advantage over LSS^M. On the other hand, when parties are connected over LAN and keys are in storage, LSS^M can be more efficient.

Inspired by this, we quantitatively compare LSS^M and FSS^M under four different deployment scenarios for batch inference for three models (sub-figures) in Figure 2. For each deployment scenario, there are two bars – heights of the bars representing runtimes with LSS^M and FSS^M, respectively (lower is better). The runtimes in seconds are also mentioned at the top of each bar. The lower of the LSS^M or FSS^M inference time for a given deployment scenario is circled green, indicating the preferred protocol for the given scenario. Further, Table 1 lists the communication and key size for LSS^M and FSS^M to help us analyze the reported runtimes.

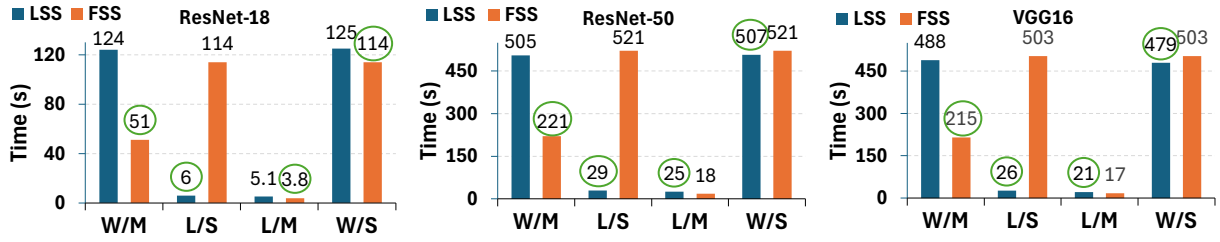
WAN, keys in memory (W/M). Here, FSS^M outperforms LSS^M by ∼ 2.5× across the models. This is expected; Table 1 shows that LSS^M communicates about 2.5× more and has about 3× more rounds than FSS^M. The runtime of LSS^M is dominated by communication, e.g., in VGG-16 inference, LSS^M spends 98% of its time communicating.

LAN, keys in storage (L/S). Here, we notice the opposite performance characteristics. LSS^M outperforms FSS^M by ∼ 19×. FSS^M must fetch keys that are 25 − 27× larger than those required by LSS^M (Table 1). Thus, FSS^M's runtime is dominated by key fetch time. Further, the fast LAN connection makes the time spent in communicating a much smaller fraction of the overall computation time for LSS^M, unlike when the connection was over WAN.

LAN, keys in memory (L/M). FSS^M is slightly faster than LSS^M across the board. While FSS-based Orca [38] was observed to beat LSS-based CrypTen by 8 − 25× in this setting, we notice that, FSS^M, which beats Orca, is only 20 − 40% faster than LSS^M. This because of our optimizations in LSS^M (Section 3).

WAN, keys in storage (W/S). Here, there is little to choose between LSS^M and FSS^M. Their runtimes are within 3-10% of the other. LSS^M suffers due to high communication, while FSS^M suffers from the time it takes to fetch keys from storage.

Opportunity to leverage heterogeneous processing: It is apparent that when the network is fast (LAN) but the keys are in memory or when the network is slow (WAN) but the keys are in storage, choosing LSS or FSS may not make a significant difference. We observe that most of the computations (up to 97%) under both LSS^M and FSS^M are attributable to non-linear layers such as ReLU and MaxPool. These layers are computed on the CPU in the case of LSS. Thanks to our careful hardware-aware optimizations in



(a) ResNet-18 (b) ResNet-50 (c) VGG16

Figure 2: One protocol does not suit all deployment scenarios – a quantitative analysis. To enhance readability, we use W/L to denote WAN/LAN, and M/S to denote that keys are in memory/storage, e.g. W/M denotes WAN with keys in memory.

realizing LSS^M (Section 3), it does not have to rely on the GPU, unlike prior works [76]). This frees the GPU when computing these non-linear layers using LSS^M. On the other hand, FSS^M, like prior work [38], relies on a GPU to accelerate computation. It leaves the CPU idle. Since the hardware requirements of the two kinds of protocols complement each other, it presents an opportunity to deploy both simultaneously for computing non-linear layers. This opens up a *first-of-its-kind* opportunity to leverage *heterogeneous processing* (simultaneous processing on CPU and GPU) to improve the throughput of secure inference.

Summary: We are the first to establish that different deployment scenarios suit different protocols. ① WAN, keys in memory: FSS^M is the preferred choice. ② LAN, keys in storage: LSS^M is the preferred choice. ③ For LAN, keys in memory and WAN, keys in storage: deploy LSS^M and FSS^M simultaneously with heterogeneous processing. Thus, we must match a protocol or a combination of protocols with a given deployment scenario – we need a *matchmaker*.

6 Design and Implementation of Matchmaker

Our tool, Matchmaker (MM), has two primary components. ① Optimized implementations of LSS- and FSS-based protocols. They are designed to be concurrently executed on the CPU and the GPU. ② Profile-guided modeling to decide the distribution of work between LSS and FSS for minimizing the latency of batch inference.

6.1 Optimized LSS^M and FSS^M

One of our contributions is the optimized implementation of LSS-based protocols on the CPU. The cornerstone of this component is the carefully vectorized implementation of secure comparison (ΠL^{Mill}). Additionally, to reduce communication overheads, we craft efficient routines to tightly pack and unpack group elements into and from bit-strings. This is especially useful when group elements have non-powers-of-2 bitwidths, e.g. 37. Our packing routine is used to tightly pack group elements before communicating them over the network. Our unpacking routine is used to store them in standard data types so they can be used in computation later. Transmitting tightly packed group elements significantly reduces communication overheads, e.g., by 15% for ResNet-50.

Besides reducing communication overheads, (un-) packing routines play a crucial role in key compression for LSS. We use AES-128 in counter mode as our PRF for compressing keys and accelerate it on the CPU with AES-NI [2] instructions. AES-128 generates a string of pseudorandom bits. These bits sometimes need to be interpreted as pseudorandom group elements with non-power-of-2 bitwidths, e.g. 20, 37, etc., in our protocols for matrix multiplication, convolution, and select. Tightly-packed non-powers-of-2 bitwidths cannot be used directly in computation. Thus, efficiently unpacking group elements with non-power-of-2 bitwidths from an AES-128 generated bit-string and storing them in standard data types is needed before they can be computed upon.

Our implementation of FSS-based protocols is built atop Orca [38] and SIGMA [33]. It extends them with $\approx$ 3000 additional lines of C++/CUDA code. We borrow the code for linear layers from Orca [38], and code for Distributed Point Functions (DPFs) and packing and unpacking non-power-of-2 bitwidths *on the GPU* from SIGMA [33]. We also write new GPU kernels for our new FSS-based protocols, e.g., for stochastic truncate-reduce and ReLU-Extend.

Our implementation allows concurrent execution of LSS-based and FSS-based protocols through multi-threading. Further, the linear layers for both LSS and FSS are computed on the same GPU concurrently. We use CUDA streams [1] to execute GPU kernels for LSS and FSS concurrently.

6.2 Profile-guided work distribution

The second component of MM is a profile-guided model to decide which protocols *or* their combination to use for computing non-linear layers under a given deployment scenario. The model takes the neural network architecture, batch size B, and the location of keys (in DRAM or in storage) as input. It outputs a fraction x, $0 \leqslant x \leqslant 1$, such that $\lfloor x \cdot B \rfloor$ images are to be computed with LSS^M. We call this fraction x MM's *configuration*. The dealer uses x to generate keys. Subsequently, parties use x to correctly parse the dealer-generated keys and use them for secure computation.

Let t_{LSS} and t_{FSS} denote the time taken by LSS^M and FSS^M, respectively, to securely compute inference of B images while running alone. Let k_{LSS} and k_{FSS} be the time to read LSS^M keys and FSS^M keys for inference of B images from storage, respectively. Let y be some MM configuration. The time to read keys from storage at configuration y is given by $y \cdot k_{LSS} + (1-y) \cdot k_{FSS}$. The computation for the current batch and the reading of the keys for the next batch happen concurrently. Thus, the time for inference of B images when keys are in storage becomes the largest of the time to read keys, LSS^M runtime and FSS^M runtime, i.e. $\max(y \cdot k_{LSS} + (1-y) \cdot k_{FSS}, y \cdot t_{LSS}, (1-y) \cdot t_{FSS})$. We want to output $x = \arg\min_y \max(y \cdot k_{LSS} + (1-y) \cdot k_{FSS}, y \cdot t_{LSS}, (1-y) \cdot t_{FSS})$. Solving this requires finding the minimum of the maximum of three

Model	PyTorch	Fixed-point $[n, f]$
ResNet-18	69.76	69.41 [32, 10]
ResNet-50	80.34	80.33 [37, 12]
VGG16	71.59	71.59 [32, 12]

Table 2: Accuracy of ImageNet-scale models. For fixed-point accuracy, n represents bitwidth and f represents scale.

lines in the interval $[0, 1]$. We fill in the values of t_{LSS}, t_{FSS}, k_{LSS} and k_{FSS} by running inference of B images with LSS$^{\text{M}}$ and FSS$^{\text{M}}$, respectively. When keys are in memory, we set $k_{LSS} = k_{FSS} = 0$.

Impact of network contention in heterogeneous processing: While LSS$^{\text{M}}$ and FSS$^{\text{M}}$ computations for non-linearities use separate processors, the CPU and the GPU, respectively, they still share the network when both run concurrently to harness heterogeneous processing. We use `fq_codel` [5] as the queuing discipline in Linux's network stack to limit contention, but it still plays a crucial role in the runtimes. Consequently, we noticed that it is possible that MM's configuration x, derived from the aforementioned equation, may lead to sub-optimal performance in practice.

Toward this, we take a two-step process to refine MM's configuration. We first determine if heterogeneous processing may yield a significant speedup under a given deployment scenario. If yes, we refine MM's configuration x to x^* that takes empirically observed network contention into account. Specifically, we first (theoretically) estimate the expected runtime at MM configuration x, yielded by the profiling-guided optimization equation described above. If this estimated runtime is at least 30% (tunable parameter) smaller than the lower of the LSS or FSS's runtime, then we consider that heterogeneous processing can yield substantial benefits and proceed on to estimate network contention as follows.

We run LSS and FSS together at configuration x and estimate the degraded bandwidth BW_{LSS} and BW_{FSS} experienced by LSS and FSS at x. We then estimate t'_{LSS} and t'_{FSS}, the time it would take for LSS and FSS to run secure inference on B images with bandwidth BW_{FSS} and BW_{LSS}. We compute $x^* = \arg\min_y \max(y \cdot k_{LSS} + (1-y) \cdot k_{FSS}, y \cdot t'_{LSS}, (1-y) \cdot t'_{FSS})$ as the *final* MM configuration.

7 Evaluation

We provide empirical evidence to justify the following claims.

- LSS$^{\text{M}}$ beats state-of-the-art LSS-based systems using GPUs, CrypTen [42] and Piranha [76], by upto 50× (Section 7.1). LSS$^{\text{M}}$ beats CPU-only MP-SPDZ by up to 1492×.
- FSS$^{\text{M}}$ beats state-of-the-art FSS-based system, Orca [38], by upto 2.2× (Section 7.1.4).
- MM beats the state-of-the-art in secure inference systems, Orca by up to 21× by judiciously leveraging LSS$^{\text{M}}$ and/or FSS$^{\text{M}}$ as appropriate in a given deployment (Section 7.3).

Evaluation setup. We use two servers to run two parties. Each server has an NVIDIA RTX A6000 GPU with 48GB of onboard memory (GDDR6). The GPU is connected to an AMD Epyc 7742 processor via a 16-lane PCIe-4 interconnect with 32GBps bandwidth. It has nearly 1TB of DRAM and is connected to two Seagate Exos 10TB disks configured as RAID0, which deliver about 400-500MBps bandwidth. This is close to the bandwidth delivered by SATA SSDs. Our servers are connected by a 9.7Gbps LAN with 0.05ms RTT. We simulate a slower WAN network with 225Mbps bandwidth and 60ms RTT using `tc` command in Linux for throttling. In all experiments, LSS$^{\text{M}}$ is run with 8 CPU threads.

Datasets and Benchmarks. Since a single inference can leave the GPU under-utilized, we focus on batch inference where the task is to label a set of images. We evaluate secure inference on the ImageNet dataset which has $224 \times 224 \times 3$ images and 1000 classes [27]. We use three ImageNet models (used previously by Orca [38]) – ResNet-18, ResNet-50, and VGG-16 which have 11.7M, 25.5M, and 138M parameters, respectively.

Fixed-point parameters. We run our LSS baselines CrypTen [42] and Piranha [76] with fixed-point bitwidth $n = 64$ and scale $f = 24$. Since both CrypTen and Piranha use (insecure) local truncations[4] that only provide probabilistic correctness, the use of large bitwidth is necessary for correctness. Like Orca [38] and MP-SPDZ [40], our protocol for truncation is secure and does not have correctness errors. This allows us to work over minimal bitwidths that suffice for preserving model accuracy. In particular, we use fixed-point parameters $[n, f] = [32, 10], [37, 12], [32, 12]$ from Orca [38] for ResNet-18, ResNet-50 and VGG-16, respectively. We use these to evaluate Orca, MP-SPDZ, LSS$^{\text{M}}$ and FSS$^{\text{M}}$. Table 2 shows that our fixed-point models match PyTorch (floating-point) accuracy. Further details about model architectures can be found in Appendix J.

7.1 LSS$^{\text{M}}$ and FSS$^{\text{M}}$: The new state-of-the-art

We first empirically substantiate our claim that LSS$^{\text{M}}$ and FSS$^{\text{M}}$ are the new state-of-the-art LSS and FSS-based secure inferencing systems, respectively. We notice that *no prior LSS-based systems realized the theoretical promise of small key size of LSS-based protocols.* In fact, the existing LSS-based systems have keys larger than even the *FSS*-based ones (which in theory are expected to have larger keys in order to reduce communication). With the introduction of LSS$^{\text{M}}$, we fix this gap between theoretical expectation and actual system behavior through a series of protocol *and* system optimizations (Section 3). This is critical in making LSS$^{\text{M}}$ performant when keys are in storage; an important scenario that was previously ignored.

We compare LSS$^{\text{M}}$ against three LSS-based systems – CrypTen [42], MP-SPDZ [40] and Piranha [76]. CrypTen and MP-SDPZ support ImageNet-scale models, while Piranha only supports smaller models for the MNIST and CIFAR-10 datasets[5]. For uniformity, we only consider ImageNet-scale models here. We compare briefly against Piranha in Section 7.1.3 and defer a detailed analysis to Appendix A. We compare FSS$^{\text{M}}$ against FSS-based state-of-the-art Orca.

Table 3 reports key size (GB), communication (GB), and runtimes (s) for all four scenarios – LAN with keys in memory (K/M) and storage (K/S), and WAN with keys in memory (K/M) and storage (K/S). Each of the three models have five rows in the table – one for each of MP-SPDZ, CrypTen, Orca, LSS$^{\text{M}}$ and FSS$^{\text{M}}$.

7.1.1 Comparing LSS$^{\text{M}}$ with CrypTen. We first detail how LSS$^{\text{M}}$ fares against CrypTen when parties are connected over LAN and keys are in memory. We then examine how runtime changes when keys must be fetched from storage.

Keys in memory (K/M). When keys are in memory, LSS$^{\text{M}}$'s speedups over CrypTen (11-31×) closely mirror improvements over CrypTen

[4] [47] proved that local truncations are insecure.
[5] These datasets have 49-192× smaller images than the images in ImageNet and thus require simpler models.

Model	Framework	Keysize (GB)	Comm (GB)	LAN, K/M (s)	LAN, K/S (s)	WAN, K/M (s)	WAN, K/S (s)
ResNet18	MP-SPDZ	950	1350	8250	8250	-	-
	CrypTen	138	179	142	298	3012	3012
	Orca	58	3.3	5	123	66	123
	LSS$^{\mathbf{M}}$	2	6.2	5.1	6	124	125
	FSS$^{\mathbf{M}}$	54	2.4	3.8	114	51	114
ResNet50	MP-SPDZ	2600	3900	11900	11900	-	-
	CrypTen	264	343	286	582	5728	5728
	Orca	265	25	28	561	477	561
	LSS$^{\mathbf{M}}$	9.1	26	25	29	505	507
	FSS$^{\mathbf{M}}$	246	11	18	521	221	521
VGG16	MP-SPDZ	8450	11750	38050	38050	-	-
	CrypTen	580	744	646	1278	13386	13386
	Orca	255	16	23	543	290	543
	LSS$^{\mathbf{M}}$	9.5	27	21	26	488	479
	FSS$^{\mathbf{M}}$	236	11	17	503	215	503

Table 3: Comparing MP-SPDZ, CrypTen, Orca, LSS$^{\mathbf{M}}$ and FSS$^{\mathbf{M}}$ on inference benchmarks with batch 50. MP-SDPZ only supports batch 1 so we multiply reported metrics by 50. - indicates that MP-SPDZ did not finish running even after one day.

in communication (13-29×). Even though CrypTen relies on GPU, LSS$^{\mathrm{M}}$'s vectorized implementation of ΠL^{Mill} on the CPU makes the compute efficient, enabling a reduction in communication over CrypTen to be reflected in end-to-end speedups.

LSS$^{\mathrm{M}}$ lowers communication over CrypTen in three ways. ① The optimized implementation of ΠL^{Mill} communicates tightly-packed bits (Section 3.3). While CrypTen computes a boolean circuit similar to LSS$^{\mathrm{M}}$ for comparison, it wastefully communicates 64 bits per party for *each* level of a single 64-bit comparison even though the number of bits *halves* at each successive level of the tree (circuit). ② LSS$^{\mathrm{M}}$ uses *optimal* fixed-point bitwidth n for all benchmarks. As discussed earlier, while CrypTen is forced to use large bitwidth (64) for correctness due to probabilistically correct truncation, LSS$^{\mathrm{M}}$ can use much smaller bitwidths, e.g., 32 for VGG-16 (see paragraph on "Fixed-point parameters"). This reduces communication since the number of bits communicated is linear in bitwidth. ③ LSS$^{\mathrm{M}}$ leverages the benefits of Orca's network-level optimizations that enable even smaller bitwidths for non-linear functions. For example, Maxpool uses comparison over $(n - f)$ bits instead of n bits. For VGG-16, we use $n = 32$ and $f = 12$. Thus, LSS$^{\mathrm{M}}$ computes Maxpool over 20 bits instead of 64 bits as in CrypTen.

Keys in storage (K/S). CrypTen assumes that keys are always in memory. To simulate its performance when keys are in storage, we instrumented CrypTen's code to measure the size of the key material used. We then measured the time taken to read that amount of key material from the storage in our server. We report the *larger* of the key read time and the online compute time as the expected runtime of CrypTen when keys are in storage.

As noted in Table 3, CrypTen needs large keys ranging from 138-580 GB. This is even larger than FSS-based Orca, contrary to the theoretical expectation that LSS-based protocols should need smaller keys than FSS-based ones. Mirroring its inefficiencies in communication, CrypTen uses the same amount of key material for each level of the comparison tree, amplifying key size by 8×. Further, as is the case with communication, CrypTen's inability to use smaller bitwidth increases the size of its key. Consequently, the time to read keys eclipses the online compute time, making inference ∼ 2× slower than when keys are in memory (LAN, K/M).

Thanks to bit packing (Section 3.3), and key compression (Section 3.4), LSS$^{\mathrm{M}}$'s keys are 30-70× smaller than CrypTen's. Crucially, they are also 27-29× smaller than FSS-based Orca's, as one would expect following theory. The de-compression of keys during online computation only adds 10-16% overhead. Thus, LSS$^{\mathrm{M}}$'s speedup over CrypTen increases to 20-50× when keys were in storage versus 11-31× when keys were in memory.

Comparison in WAN. Over a slow network, both Crypten and LSS$^{\mathrm{M}}$ are bottlenecked by communication, as expected. LSS$^{\mathrm{M}}$'s speedup over CrypTen reflects its improvement over CrypTen in communication (11-28×). This is independent of the location of the keys.

7.1.2 Comparing LSS^{M} with MP-SPDZ. We compare LSS$^{\mathrm{M}}$ and MP-SPDZ when parties are connected on LAN, since MP-SPDZ does not finish running over WAN even after one day. MP-SPDZ does not support batch inference for the evaluated models. Thus, we report all its metrics for a single inference multiplied by the batch size (here, 50) in Table 3. To estimate key size, we use the output of MP-SPDZ's online phase to collect the amount of key material used (e.g., Beaver triples). We convert this to bytes (GBs) by assuming that bits and *all* ring elements (even ones with non-power-of-2 bitwidths) are tightly packed. We present this key size in Table 3. We measure the time it would take to read that much data from our storage and report the larger of the key read and online time as time in LAN when keys are on storage (LAN, K/S). As Table 3 shows, we improve over MP-SPDZ by 271 – 1492× in the LAN setting. LSS$^{\mathrm{M}}$'s improvement mainly stems from MP-SPDZ's inefficient implementation of linear layers and Maxpool. MP-SPDZ uses one Beaver triple for *each* multiplication while multiplying matrices, which amplifies the compute, communication, and key size. Moreover, it is a CPU-*only* framework. Hence, its local computation for linear layers is significantly slower than LSS$^{\mathrm{M}}$'s (which uses the GPU for linear layers but CPU for non-linear). Using one triple per multiplication makes MP-SPDZ's keys 217 – 384× larger and its communication 149 – 438× higher.

7.1.3 Comparing LSS^{M} with Piranha. LSS$^{\mathrm{M}}$ reduces keysize by 4.5-11× and improves performance by 1.8-7.7× over Piranha on Piranha's benchmarks. It does so while ensuring end-to-end security,

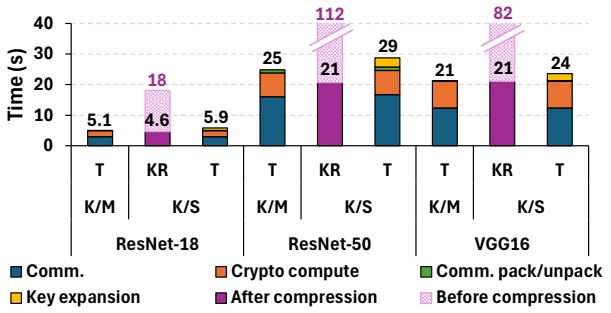


Figure 3: Breakdown of LSS^M's runtime under LAN. K/M denotes keys in memory and K/S denotes keys in storage. T denotes online time. KR denotes key read time from storage.

unlike Piranha. Further, Piranha only supports smaller models for MNIST and CIFAR-10 datasets, which limits headroom for speedups. Due to differences in benchmarks, we defer a detailed comparison with Piranha to Appendix A.

7.1.4 Comparing FSS^M with Orca. We detail how FSS^M improves upon state-of-art for FSS-based inferencing systems, Orca [38] when keys are in memory. FSS^M's optimized FSS protocols with bit packing beat Orca's communication by 1.4-2.3× (Table 3). This is beneficial in the slow WAN, where communication directly impacts performance. On LAN, FSS^M's lower compute needs (> 2×, Section 4) and smaller keys (by 7-8%) help it outperform Orca by 1.3-1.5×. Interestingly, while key size does *not* affect LSS^M when keys are in memory, it has an impact on FSS^M. This is because FSS^M uses GPUs for non-linear layers and needs to move large FSS keys from CPU to GPU memory over the PCIe.

When keys are in storage, both FSS^M and Orca are bottlenecked by the key read time, irrespective of whether the parties are connected over the LAN or WAN. FSS^M marginally speeds up over Orca (7-8%) due to smaller key size.

Orca is best amongst prior works. From the table, we notice that Orca outperforms prior secure inferencing systems, such as MP-SPDZ, and CrypTen, for all models and under all deployment scenarios. Thus, in Section 7.3, we will compare Matchmaker, our secure inference system, with Orca.

7.2 Performance breakdown of LSS^M and FSS^M

We break down LSS^M and FSS^M's runtimes for a deeper analysis.

Performance Breakdown of LSS^M. We first consider the case when parties are connected over LAN [6] in Figure 3. We consider two scenarios: keys in memory (K/M) and keys in storage (K/S).

K/M has a single stacked bar per model, which provides a breakdown of online time (T). We report the time spent on communication (blue), cryptographic (protocol) computation (orange), communication packing and unpacking (green) and key expansion (yellow). K/S has two stacked bars. The first bar shows the key read time from storage (KR). The dark purple bar shows the actual key read time in LSS^M, and the light purple colored extension shows what the key read time *would have been*, without key compression. This captures the benefit of key compression. The second bar shows a breakdown of online time. The numbers above all the bars denote the total time for the corresponding bar in seconds.

From the T bars for both K/M and K/S, we see that LSS^M spends > 50% of time on communication (blue stack), even with a fast LAN network. It spends 58-65% of time on communication when keys are in memory and 51-58% when keys are in storage. This demonstrates that after an optimized implementation on the CPU, there is little room for further optimizing the computation of non-linear layers [7], e.g., by using a GPU. It is already communication-bound.

Next, we notice that LSS^M pays 10-16% performance overhead for expanding the compressed keys when keys are in storage (yellow stack on the T bar for K/S). However, without compression, the time to read the keys from storage (total height of KR bars under K/S) would have eclipsed the compute time (T bars under K/M). This would have slowed down LSS^M by 3.6-4.5× when keys are in the storage (difference between heights of KR and T, K/M bars). Finally, communication packing and unpacking take up a small fraction (< 4%) of overall execution time across all models and settings.

Performance Breakdown of FSS^M. In the LAN, once FSS^M optimizes the heavy FSS computation on the GPU, a majority of the time is spent on moving (large) FSS keys from host (CPU) memory to GPU memory. Across all models, FSS^M spends 60% of it time on CPU-GPU data transfer, 30% of its time on communication, and only 10% of its time on actual computation. In the WAN, FSS^M spends more than 90% of its time on communication.

7.3 Putting It All Together: Matchmaker under different deployment scenarios

Recall that a primary contribution of our work is Matchmaker (MM), which *automatically* chooses LSS^M, FSS^M, or a combination of both, as appropriate, for a given deployment scenario. We demonstrate this unique adaptability of MM by reporting its performance against the state-of-the-art secure inferencing system, Orca [38] under four diverse deployment scenarios – LAN with keys in memory/storage, and WAN with keys in memory/storage.

Table 4a reports the performance of MM and Orca under the four deployment scenarios while 4b reports the relevant secondary metrics such as key size and communication.

In Table 4a, *each* scenario has three sub-columns – the first two list the time (in seconds) that it takes for Orca and MM to execute in that given scenario, while the last column shows the configuration chosen by MM. Specifically, it reports the fraction of non-linearities executed with LSS^M. MM uses FSS^M for the rest. For example, an entry 0.42 means that MM uses LSS^M for secure computation of 42% of the non-linearities and uses FSS^M for the remaining 58%.

From Table 4a, we observe that MM *always* outperforms Orca under all deployment scenarios demonstrating MM's adaptability. It speeds up secure inferencing by 1.3-21× depending upon the scenario. MM's greatest improvement over Orca (over 20×) comes under LAN when keys are in storage. This is expected – Orca's FSS-based protocol with large keys is bottlenecked by time to read the keys from storage. MM, instead, deploys LSS^M under this scenario that uses 27-29× smaller keys than Orca (Table 4b) and is also not constrained by communication overheads due to a fast LAN.

[6] We do not provide a breakdown when parties are connected over WAN since more than 95% time there is spent on communicating due to slow network.

[7] Note that linear layers always execute on GPUs.

Model Batch=50	LAN, keys in mem.			LAN, keys in storage			WAN, keys in mem.			WAN, keys in storage		
	Orca	MM	MM config.	Orca	MM	MM config.	Orca	MM	MM config.	Orca	MM	MM config.
ResNet-18	5 (2×)	2.5	0.44	123 (21×)	5.9	1	66 (1.3×)	50	0	123 (1.8×)	70	0.42
ResNet-50	28 (2.3×)	12.4	0.40	561 (19×)	29	1	477 (2.2×)	220	0	561 (1.9×)	300	0.44
VGG-16	23 (2×)	11.4	0.44	543 (21×)	26	1	290 (1.4×)	209	0	543 (1.9×)	293	0.44

(a) Performance of MM compared to Orca. We report the runtime in seconds. MM config. denotes fraction of non-linearities run with LSS^M.

Model Batch=50	Keysize (GB)					Comm (GB)				
	Orca	MM				Orca	MM			
		LAN, keys in mem.	LAN, keys in storage	WAN, keys in mem.	WAN, keys in storage		LAN, keys in mem.	LAN, keys in storage	WAN, keys in mem.	WAN, keys in storage
ResNet-18	58	34	2	54	32	3.3	4.1	6.2	2.4	4
ResNet-50	265	146	9.1	246	142	25	17.1	26	11	17.7
VGG-16	255	149	9.5	236	137	16	18.1	27	11	18.1

(b) Keysize and communication of Orca and MM in different settings.

Table 4: Comparing Matchmaker (MM) with Orca (state-of-the-art) under different deployment scenarios.

On the other extreme, when keys are in memory and the network is slow (i.e., in the WAN), MM correctly chooses to deploy FSS^M since deploying LSS^M would have made it hamstrung by communication overheads. Here, MM's improvements over Orca [38] are exactly the improvements of FSS^M over Orca (1.3-2.2×).

In two scenarios, MM employs novel *heterogeneous processing*. i.e. runs LSS^M and FSS^M simultaneously on the CPU and the GPU, respectively. When the keys are in memory and the network is fast (LAN), both LSS^M and FSS^M perform similarly (Section 5). While FSS^M uses GPU for secure computation of non-linearities, MM simultaneously deploys LSS^M on the CPU to increase the inference throughput. Table 4a reports 40-44% of non-linearities are computed using LSS^M while FSS^M is responsible for the rest.

MM also employs heterogeneous processing when keys are in storage and the parties are connected over WAN. While FSS^M may seem like the best choice in WAN due to its low communication, it needs large amounts of keys. For example, it takes 503 seconds to read FSS keys for VGG16 from storage, while the compute takes 215 seconds. On the other hand, LSS^M needs just 21 seconds to read (compressed) LSS keys for VGG16, but needs 479 seconds to compute. MM then judiciously partitions the work for securely computing non-linearities between LSS^M and FSS^M to strike a fine balance between computation time and the key read time such that both are roughly equal. It does so automatically without manual intervention, thanks to its profile-guided modeling (Section 6.2). MM improves performance by 1.8-1.9× over Orca under this scenario.

Table 4b also captures how MM trades off keys size and communication overhead based on the deployment scenario. Notice that while the key size and amount of communication remain the same for Orca across the deployment scenarios, for MM changes as it adapts to different deployment scenarios. For example, under LAN and when keys are in storage, FSS^M slightly increases the amount of communication over Orca (up to 87%) but significantly lowers the amount of keys needed (by 26-29×). Thanks to a fast LAN network, a slight increase in the amount of communication has little impact on performance but needing to fetch much smaller amounts of keys from the slow storage helps performance. On the other hand, when the parties are connected over WAN and keys are in memory, MM prioritizes limiting the communication. In summary, MM adapts to the characteristics of the given deployment scenario.

8 Related work

Secure Inference/2PC with preprocessing. SecureML [52] introduced the problem of secure inference in the preprocessing model. Following a long line of works [34, 39, 63, 64], GPU-accelerated Piranha [76] and CrypTen [42] are the state-of-the-art in LSS-based secure inference of CNNs with preprocessing. GForce [53] and Delphi [50] also use GPUs but rely on training MPC-friendly ML models. MP-SPDZ [6, 40] and ABY2.0 [7, 56] implement LSS-based 2PC protocols with preprocessing on CPUs. Orca [38] and SIGMA [33] build upon prior FSS works [17, 34, 64, 67, 71] and are the state-of-the-art secure inference systems for CNNs and LLMs, respectively.

Secure Inference/2PC without preprocessing. The 2-party secure CNN inference (without preprocessing) is a well-studied problem [36, 39, 50, 52, 53, 60, 61]. They use different techniques, e.g., oblivious transfer and/or homomorphic encryption, that are naturally more expensive than works with preprocessing. Works such as [9, 24, 35, 49, 55] consider secure LLM inference. Some of these works modify the underlying ML algorithm to be more 2PC-friendly. They [50, 53, 55] also accelerate computation using GPUs.

Secure ML under other models. Finally, several works have also considered secure computation of ML algorithms amongst > 2 parties [23, 25, 28, 41, 42, 44, 51, 57, 68, 72, 74, 81]. Due to a different setup, the performance of these protocols are incomparable to 2PC protocols in the preprocessing model. A few works have also explored malicious secure 2/MPC for ML [20, 46, 51, 74, 81]. We leave the exploration of such protocols in our context to future work.

Other privacy/security approaches. Another approach to secure inference is via Trusted Execution Environments (TEEs) that make assumptions on the hardware to provide security [32, 54, 58, 69, 70]. Works on Federated learning [43] and those that improve the security/privacy guarantees in federated learning [12, 62, 65, 78] aim to limit the amount of information shared between the participants

during ML training while works on differential privacy [8, 73] provide *privacy* guarantees to individual data records in the training data. Both these lines of works are orthogonal to secure computation of training algorithms and are further inapplicable to inference.

Modeling-based protocol selection. Prior works [18, 21, 22] considered using modeling to select the protocol best suited to a given deployment scenario. However, these works did not consider the preprocessing model and neither did they consider GPU acceleration. Furthermore, they also do not consider simultaneously executing different protocols with varying performance characteristics.

References

[1] CUDA Streams. https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/.

[2] Intel Advanced Encryption Standard Instructions . https://www.intel.com/content/www/us/en/developer/articles/technical/advanced-encryption-standard-instructions-aes-ni.html.

[3] Intel Advanced Vector Extensions 512. https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html.

[4] Intrinsics for Intel Advanced Vector Extensions. https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-8/intrinsics-for-intel-advanced-vector-extensions.html.

[5] Linux FQ_CoDel . https://man7.org/linux/man-pages/man8/tc-fq_codel.8.html.

[6] Multi-Protocol SPDZ: Versatile framework for multi-party computation, 2019.

[7] MOTION2NX – A Framework for Generic Hybrid Two-Party Computation and Private Inference with Neural Networks. https://github.com/encryptogroup/MOTION2NX, 2022.

[8] Martin Abadi, Andy Chu, Ian Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[9] Y. Akimoto, K. Fukuchi, Y. Akimoto, and J. Sakuma. Privformer: Privacy-preserving transformer with mpc. In *EuroS&P*, 2023.

[10] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO*, '91.

[11] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In *EUROCRYPT*, 2011.

[12] Kallista A. Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for privacy-preserving machine learning. In Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1175–1191. ACM, 2017.

[13] Elette Boyle, Nishanth Chandran, Niv Gilboa, Divya Gupta, Yuval Ishai, Nishant Kumar, and Mayank Rathee. Function secret sharing for mixed-mode and fixed-point secure computation. In *EUROCRYPT*, 2020.

[14] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. Efficient two-round ot extension and silent non-interactive secure computation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 291–308, New York, NY, USA, 2019. Association for Computing Machinery.

[15] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In *EUROCRYPT*, 2015.

[16] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *CCS*, 2016.

[17] Elette Boyle, Niv Gilboa, and Yuval Ishai. Secure computation with preprocessing via function secret sharing. In *TCC*, 2019.

[18] Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, and Thomas Schneider. HyCC: Compilation of Hybrid Protocols for Practical Secure Computation. In *CCS*, 2018.

[19] Ran Canetti. Security and Composition of Multiparty Cryptographic Protocols. *J. Cryptology*, 2000.

[20] Nishanth Chandran, Divya Gupta, Sai Lakshmi Bhavana Obbattu, and Akash Shah. SIMC: ML Inference Secure Against Malicious Clients at Semi-Honest Cost. In *USENIX Security Symposium*, 2022.

[21] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. Ezpc: Programmable and efficient secure two-party computation for machine learning. In *IEEE European Symposium on Security and Privacy, EuroS&P 2019, Stockholm, Sweden, June 17-19, 2019*, pages 496–511. IEEE, 2019.

[22] Edward Chen, Jinhao Zhu, Alex Ozdemir, Riad S. Wahby, Fraser Brown, and Wenting Zheng. Silph: A framework for scalable and accurate generation of hybrid mpc protocols. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 848–863, 2023.

[23] Hao Chen, Miran Kim, Ilya Razenshteyn, Dragos Rotaru, Yongsoo Song, and Sameer Wagh. Maliciously secure matrix multiplication with applications to private deep learning. In *ASIACRYPT*, 2020.

[24] Tianyu Chen, Hangbo Bao, Shaohan Huang, Li Dong, Binxing Jiao, Daxin Jiang, Haoyi Zhou, Jianxin Li, and Furu Wei. THE-X: privacy-preserving transformer inference with homomorphic encryption. In *ACL*, 2022.

[25] Anders Dalskov, Daniel E. Escudero, and Marcel Keller. Fantastic four: Honest-majority four-party secure computation with malicious security. *IACR Cryptol. ePrint Arch.*, 2020:1330, 2020.

[26] Ivan Damgård and Sarah Zakarias. Constant-overhead secure computation of boolean circuits using preprocessing. In *TCC*, 2013.

[27] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *CVPR*, 2009.

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

[29] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In *CRYPTO*, 2020.

[30] Juan Garay, Berry Schoenmakers, and José Villegas. Practical and secure solutions for integer comparison. In Tatsuaki Okamoto and Xiaoyun Wang, editors, *Public Key Cryptography – PKC 2007*, pages 330–342, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[31] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *STOC*, 1987.

[32] Jinnan Guo, Peter R. Pietzuch, Andrew Paverd, and Kapil Vaswani. Trustworthy AI using confidential federated learning: Federated learning and confidential computing are not competing technologies. *ACM Queue*, *22*(2), 2024.

[33] Kanav Gupta, Neha Jawalkar, Ananta Mukherjee, Nishanth Chandran, Divya Gupta, Ashish Panwar, and Rahul Sharma. Sigma: Secure gpt inference with function secret sharing. In *Proc. Priv. Enhancing Technol.*, 2024.

[34] Kanav Gupta, Deepak Kumaraswamy, Nishanth Chandran, and Divya Gupta. Llama: A low latency math library for secure inference. In *PETS*, 2022.

[35] Meng Hao, Hongwei Li, Hanxiao Chen, Pengzhi Xing, Guowen Xu, and Tianwei Zhang. Iron: Private inference on transformers. In *NeurIPS*, 2022.

[36] Zhicong Huang, Wen jie Lu, Cheng Hong, and Jiansheng Ding. Cheetah: Lean and fast secure two-party deep neural network inference. In *USENIX Security Symposium*, 2022.

[37] Yuval Ishai, Eyal Kushilevitz, Sigurd Meldgaard, Claudio Orlandi, and Anat Paskin-Cherniavsky. On the power of correlated randomness in secure computation. In *TCC*, 2013.

[38] N. Jawalkar, K. Gupta, A. Basu, N. Chandran, D. Gupta, and R. Sharma. Orca: Fss-based secure training and inference with gpus. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 62–62, Los Alamitos, CA, USA, may 2024. IEEE Computer Society.

[39] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. Gazelle: A low latency framework for secure neural network inference. In *USENIX Security Symposium*, 2018.

[40] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In *CCS*, 2020.

[41] Marcel Keller and Ke Sun. Secure quantized training for deep learning. In *ICML*, 2022.

[42] Brian Knott, Shobha Venkataraman, Awni Hannun, Shubhabrata Sengupta, Mark Ibrahim, and Laurens van der Maaten. CrypTen: Secure multi-party computation meets machine learning. In *NeurIPS*, 2021.

[43] Jakub Konečný, Brendan McMahan, and Daniel Ramage. Federated optimization: Distributed optimization beyond the datacenter. *CoRR*, abs/1511.03575, 2015.

[44] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. SWIFT: super-fast and robust privacy-preserving machine learning. In *USENIX Security Symposium*, 2021.

[45] Nishant Kumar, Mayank Rathee, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. CrypTFlow: Secure tensorflow inference. In *IEEE S&P*, 2020.

[46] Ryan Lehmkuhl, Pratyush Mishra, Akshayaram Srinivasan, and Raluca Ada Popa. Muse: Secure inference resilient to malicious clients. In *USENIX Security Symposium*, 2021.

[47] Yun Li, Yufei Duan, Zhicong Huang, Cheng Hong, Chao Zhang, and Yifan Song. Efficient 3PC for Binary Circuits with Application to Maliciously-Secure DNN Inference. In *USENIX Security Symposium*, 2023.

[48] Yehuda Lindell. How to simulate it – a tutorial on the simulation proof technique. *Tutorials on the Foundations of Cryptography*, 2017.

[49] Wen-jie Lu, Zhicong Huang, Zhen Gu, Jingyu Li, Jian Liu, Kui Ren, Cheng Hong, Tao Wei, and Wenguang Chen. Bumblebee: Secure two-party inference framework for large transformers. In *NDSS 2025*, 2025.

[50] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. Delphi: A cryptographic inference service for neural networks. In *USENIX Security Symposium*, 2020.

[51] Payman Mohassel and Peter Rindal. ABY^3: A Mixed Protocol Framework for Machine Learning. In *CCS*, 2018.

[52] Payman Mohassel and Yupeng Zhang. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *IEEE S&P*, 2017.

[53] Lucien K. L. Ng and Sherman S. M. Chow. Gforce: Gpu-friendly oblivious and rapid neural network inference. In *USENIX Security Symposium*, 2021.

[54] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 619–636. USENIX Association, 2016.

[55] Q. Pang, J. Zhu, H. Möllering, W. Zheng, and T. Schneider. Bolt: Privacy-preserving, accurate and efficient inference for transformers. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 133–133, Los Alamitos, CA, USA, may 2024. IEEE Computer Society.

[56] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. ABY2.0: Improved Mixed-Protocol secure Two-Party computation. In *USENIX Security Symposium*, 2021.

[57] Arpita Patra and Ajith Suresh. Blaze: Blazing fast privacy-preserving machine learning. In *NDSS*, 2020.

[58] Rishabh Poddar, Ganesh Ananthanarayanan, Srinath Setty, Stavros Volos, and Raluca Ada Popa. Visor: Privacy-preserving video analytics as a cloud service. In *USENIX Security Symposium*, 2020.

[59] Deevashwer Rathee, Anwesh Bhattacharya, Rahul Sharma, Divya Gupta, Nishanth Chandran, and Aseem Rastogi. SecFloat: Accurate Floating-Point meets Secure 2-Party Computation. In *IEEE S&P*, 2022.

[60] Deevashwer Rathee, Mayank Rathee, Rahul Kranti Kiran Goli, Divya Gupta, Rahul Sharma, Nishanth Chandran, and Aseem Rastogi. SIRNN: A math library for secure inference of RNNs. In *IEEE S&P*, 2021.

[61] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. CrypTFlow2: Practical 2-Party Secure Inference. In *CCS*, 2020.

[62] M. Rathee, C. Shen, S. Wagh, and R. Popa. Elsa: Secure aggregation for federated learning with malicious actors. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1961–1979, Los Alamitos, CA, USA, may 2023. IEEE Computer Society.

[63] 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 *ASIACCS*, 2018.

[64] Théo Ryffel, David Pointcheval, and Francis Bach. ARIANN: Low-interaction privacy-preserving deep learning via function secret sharing. In *PETS*, 2022.

[65] Sinem Sav, Apostolos Pyrgelis, Juan Ramón Troncoso-Pastoriza, David Froelicher, Jean-Philippe Bossuat, Joao Sa Sousa, and Jean-Pierre Hubaux. POSEIDON: privacy-preserving federated neural network learning. In *NDSS*, 2021.

[66] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218. USENIX Association, July 2020.

[67] Kyle Storrier, Adithya Vadapalli, Allan Lyons, and Ryan Henry. Grotto: Screaming fast $(2+1)$-pc for $\mathbb{Z}_{2^n}$ via (2, 2)-dpfs. Cryptology ePrint Archive, Paper 2023/108, 2023.

[68] Sijun Tan, Brian Knott, Yuan Tian, and David J. Wu. Cryptgpu: Fast privacy-preserving machine learning on the GPU. In *IEEE S&P*, 2021.

[69] Florian Tramèr and Dan Boneh. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. In *ICLR*, 2019.

[70] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted execution environments on gpus. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 681–696. USENIX Association, 2018.

[71] Sameer Wagh. Pika: Secure Computation using Function Secret Sharing over Rings. *PoPETs*, 2022.

[72] Sameer Wagh, Divya Gupta, and Nishanth Chandran. SecureNN: 3-party secure computation for neural network training. *PoPETs*, 2019.

[73] Sameer Wagh, Xi He, Ashwin Machanavajjhala, and Prateek Mittal. Dp-cryptography: Marrying differential privacy and cryptography in emerging applications. *Commun. ACM*, 2021.

[74] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. Falcon: Honest-majority maliciously secure framework for private deep learning. *PoPETs*, 2021.

[75] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 443–457. USENIX Association, July 2021.

[76] Jean-Luc Watson, Sameer Wagh, and Raluca Ada Popa. Piranha: A GPU Platform for Secure Computation. In *USENIX Security Symposium*, 2022.

[77] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. Ferret: Fast extension for correlated OT with small communication. In *CCS*, 2020.

[78] Yuchen Yang, Bo Hui, Haolin Yuan, Neil Gong, and Yinzhi Cao. PrivateFL: Accurate, differentially private federated learning via personalized data transformation. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1595–1612, Anaheim, CA, August 2023. USENIX Association.

[79] Andrew C. Yao. Protocols for secure computations. In *FOCS*, 1982.

[80] Edouard Yvinec, Arnaud Dapogny, and Kevin Bailly. To fold or not to fold: a necessary and sufficient condition on batch-normalization layers folding, 2022.

[81] Wenting Zheng, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Helen: Maliciously secure coopetitive learning for linear models. In *IEEE S&P*, 2019.

[82] Victor Quétu Van-Tam Nguyen Enzo Tartaglione Zhu Liao, Nour Hezbri. Till the layers collapse: Compressing a deep neural network through the lenses of batch normalization layers.

Boolean to Arithmetic Π_n^{B2A}

$\mathsf{Gen}_n^{\mathsf{B2A}}$:

1: $t \xleftarrow{\$} \{0,1\}$
2: $t' = \mathsf{extend}(t, n)$
3: share t'
4: For $b \in \{0,1\}$, $k_b = t'_b$

$\mathsf{Eval}_n^{\mathsf{B2A}}(b, k_b, s_b)$:

1: Parse k_b as t'_b
2: $t_b = t'_b \mod 2$; $\hat{s}_b = s_b \oplus t_b$
3: $\hat{s} \leftarrow$ reconstruct $(\hat{s}_b)$
4: $\hat{s}' = \mathsf{extend}(\hat{s}, n)$
5: **return** $z_b = b \cdot \hat{s}' + (1 - 2\hat{s}') \cdot t'_b$

Figure 4: Protocol for $\mathsf{B2A}_n$

A Comparing LSS$^{\mathrm{M}}$ with Piranha

Datasets. We perform secure inference on the MNIST and CIFAR-10 datasets. MNIST is a 10-class dataset with 28×28 images and 10 classes. CIFAR-10 is also a 10-class dataset, but has $32 \times 32 \times 3$ images ($\approx 4\times$ as large as MNIST).

Benchmarks. Piranha uses two MNIST-scale models – P-SecureML and P-LeNet, and two CIFAR-10-scale models – P-AlexNet and P-VGG16. Piranha modifies plaintext models, e.g. by replacing Maxpool with Averagepool, so we use the prefix P to separate them from the corresponding floating-point plaintext models. The accuracy of these models for our choice of fixed-point parameters ($n = 64, f = 24$) can be found in Figure 5 in [76].

Performance. Table 5a compares the performance of LSS$^{\mathrm{M}}$ with Piranha in all four settings – in the LAN when keys are in memory/storage, and in the WAN when keys are in memory/storage. We provide keysize, communication and rounds in Table 5b to help with analysis. We split our analysis into two parts.

LAN, keys in memory/storage. LSS$^{\mathrm{M}}$ is $2.9 - 6.3\times$ faster than Piranha when keys are in memory, mirroring improvement in communication. Notably we are faster than Piranha and communicate $2 - 6\times$ less data while employing a more expensive (but *secure*) protocol for stochastic truncation. Our lower communication can be attributed to two sources. First, our efficient implementation of ΠL_n^{Mill} (Section 3.3) that communicates tightly packed bits. Piranha computes a comparison circuit similar to ours, but embeds each bit of the comparison input into a Byte, and then treats the Bytes as logical bits. This amplifies communication by 8×. Second, we apply Orca's network-level optimizations to Piranha's networks.

Piranha does not faithfully generate keys and write them to storage. It assumes that keys are always in memory. To simulate keys in storage, we instrument its code to output the size of the preprocessing material used. We then measure the time it takes to read that much data from our disk. We report Piranha's expected runtime as the larger of disk read and online compute. Table 5a shows that LSS$^{\mathrm{M}}$ is $2.4 - 7.7\times$ faster than Piranha when keys are on disk. For smaller networks P-SecureML, P-LeNet and P-AlexNet (< 4 million parameters), Piranha's runtime degrades only slightly ($1 - 1.2\times$) from having to read keys from disk. LSS$^{\mathrm{M}}$ is able to hide keyread behind online time, but online time itself is $1.08 - 1.3\times$ larger than when keys are in memory since key expansion is in the critical path. LSS$^{\mathrm{M}}$ suffers slightly more than Piranha when keys are in storage for smaller networks, so its improvement over Piranha decreases slightly to $2.4 - 4.3\times$ compared to the case when keys are in memory. For P-VGG16, which is a larger network (~ 15 million parameters), Piranha takes $1.7\times$ longer to read keys from disk than online computation. LSS$^{\mathrm{M}}$'s keysize-specific optimizations keep keyread time smaller than online time and key expansion costs relatively low ($1.4\times$ longer online time compared to when keys are in memory). Thus, LSS$^{\mathrm{M}}$'s improvement over Piranha increases to $7.7\times$ for P-VGG16, compared to $6.3\times$ when keys are in memory.

WAN, keys in memory/storage. In the slow WAN, LSS$^{\mathrm{M}}$ and Piranha are bottlenecked by communication and rounds, regardless of where the keys are located. LSS$^{\mathrm{M}}$'s improvement over Piranha in performance ($1.8 - 4.1\times$) mostly mirrors its improvement over Piranha in communication ($2.2 - 4.5\times$). The only anomaly is P-SecureML. Here, we see the impact of LSS$^{\mathrm{M}}$'s muted improvement over Piranha in rounds ($1.4 - 1.7\times$), arising from its use of *secure* stochastic truncation. Rounds do not impact performance when the network is fast (LAN), but they do impact performance in the WAN when the amount of data being communicated is small (tens of MBs). Both Piranha and LSS$^{\mathrm{M}}$ communicate very little data for P-SecureML. Consequently, LSS$^{\mathrm{M}}$ spends 1.2 seconds on rounds, which makes up 92% of its runtime. Piranha spends 1.98 seconds on rounds (86% of its runtime). Since the time for rounds dominates, LSS$^{\mathrm{M}}$'s improvement over Piranha for P-SecureML closely mirrors its improvement over Piranha in rounds ($1.7\times$).

B Boolean shares to arithmetic shares

We rely on the following observation (which was also made in Orca [38]). For $s, t \in \{0,1\}$ such that $\hat{s} = s \oplus t$ and $s' = \mathsf{extend}(s, n)$, $t' = \mathsf{extend}(t, n)$, $\hat{s}' = \mathsf{extend}(\hat{s}, n)$,

$$\begin{aligned}\mathsf{B2A}_n(s) &= \mathsf{extend}(s, n)\\ &= \hat{s}' - t' + 2 \cdot \mathbf{1}\{\hat{s} < t\}\\ &= \hat{s}' - t' + 2 \cdot (1 - \hat{s}') \cdot t'\\ &= \hat{s}' + (1 - 2\hat{s}') \cdot t'\end{aligned}$$

Our protocol Π_n^{B2A} uses the above expression and is described in Figure 4.

C Select

Orca [38] writes select as a mixed-bitwidth multiplication. Let $x, r, \hat{x} \in \mathbb{U}_N$ be such that $\hat{x} = x + r \mod N$. Let $s, t, \hat{s} \in \{0,1\}$ be such that $\hat{s} = s \oplus t$ and $\hat{s}' = \mathsf{extend}(\hat{s}, n)$. Let $t' = \mathsf{extend}(t, n)$. Then, from [34] and the expression in Appendix B we have that

$$\begin{aligned}\mathsf{select}_n(s, x) &= \mathsf{extend}(s, n) \cdot x\\ &= (\hat{s}' + (1 - 2\hat{s}') \cdot t') \cdot (\hat{x} - r)\\ &= \hat{s}' \cdot \hat{x} - \hat{s}' \cdot r + (1 - 2\hat{s}') \cdot \hat{x} \cdot t' - (1 - 2\hat{s}') \cdot t' \cdot r\end{aligned}$$

We use the above expression to describe our protocol for select_n in Figure 5.

D Security proof of Millionaire's and Wrap

D.1 Security proof of ΠL_n^{Mill}

Model	LAN, keys in mem. (s)		LAN, keys in storage (s)		WAN, keys in mem. (s)		WAN, keys in storage (s)	
	Piranha	LSS^M	Piranha	LSS^M	Piranha	LSS^M	Piranha	LSS^M
P-SecureML	0.057 (4.8×)	0.012	0.057 (3.4×)	0.017	2.3 (1.8×)	1.3	2.3 (1.8×)	1.3
P-LeNet	0.402 (3×)	0.136	0.48 (3×)	0.159	8.6 (1.8×)	4.9	8.6 (1.8×)	4.9
P-AlexNet	0.424 (2.6×)	0.162	0.432 (2.3×)	0.19	12.2 (1.5×)	7.9	12.2 (1.5×)	7.9
P-VGG16	14 (5.7×)	2.47	23.726 (8.2×)	2.9	259 (3.8×)	67	259 (3.8×)	67

(a) Comparing the performance of LSS^M with Piranha.

Model	Key size (MB)		Comm (MB)		Rounds	
	Piranha	LSS^M	Piranha	LSS^M	Piranha	LSS^M
P-SecureML	15 (11×)	1.4	21 (3.3×)	6.3	66 (1.6×)	42
P-LeNet	265 (4.6×)	58	335 (2.1×)	157	108 (1.3×)	82
P-AlexNet	283 (5×)	57	324 (1.8×)	183	223 (1.4×)	162
P-VGG16	10163 (9.5×)	1071	13589 (4.3×)	3191	474 (1.4×)	343

(b) Comparing keysize and communication of LSS^M with Piranha.

Table 5: Comparing LSS^M with Piranha on inference benchmarks with batch 128.

Select Π_n^{select}

$\text{Gen}_n^{\text{select}}$:

1: $t \xleftarrow{\$} \{0,1\}$; $r \xleftarrow{\$} \mathbb{U}_N$
2: $t' = \text{extend}(t, n)$; $u = t' \cdot r$
3: share r, t', u
4: For $b \in \{0,1\}$, $k_b = r_b || t'_b || u_b$

$\text{Eval}_n^{\text{select}}(b, k_b, s_b, x_b)$:

1: Parse k_b as $r_b || t'_b || u_b$
2: $\hat{x}_b = x_b + r_b \mod N$;
3: $t_b = t'_b \mod 2$
4: $\hat{s}_b = s_b \oplus t_b$
5: $(\hat{x}, \hat{s}) \leftarrow$ reconstruct $(\hat{x}_b, \hat{s}_b)$
6: $\hat{s}' = \text{extend}(\hat{s}, n)$
7: **return** $z_b = b \cdot \hat{s}' \cdot \hat{x} - \hat{s}' \cdot r_b + (1 - 2\hat{s}') \cdot \hat{x} \cdot t'_b - (1 - 2 \cdot \hat{s}') \cdot u_b$

Figure 5: Protocol for select_n

The ideal functionality for logical AND, $\mathcal{F}^{\text{AND}}$, takes secret shares of bits p, q as input and returns secret shares of the bit $p \wedge q$ as output. We define a related functionality $\mathcal{F}^{\text{AND}'}$ that takes a bit p as P_0's private input and a bit q as P_1's private input and returns secret shares of $p \wedge q$. We additionally define the gate ANDCorr as taking bits p, q, r as input and returning $p \wedge q$ and $q \wedge r$ as output. The functionality $\mathcal{F}^{\text{ANDCorr}}$ takes shares of bits p, q, r as input and returns shares of ANDCorr's outputs.

Let C denote the plaintext comparison logic described in Section 3.1. C can be written as a circuit with AND, ANDCorr and XOR gates. The protocols Π^{AND}, $\Pi^{\text{AND}'}$ and Π^{ANDCorr} that securely realize $\mathcal{F}^{\text{AND}}$, $\mathcal{F}^{\text{AND}'}$ and $\mathcal{F}^{\text{ANDCorr}}$ can be trivially constructed using Beaver triples.

The ideal functionality $\mathcal{F}_n^{\text{Mill}}$ for the Millionaire's problem takes inputs x, y from P_0, P_1 respectively and returns secret shares of $\mathbf{1}\{x < y\}$. Intuitively, the security of ΠL_n^{Mill} follows from the correctness of C and the security of the AND protocol (that uses Beaver bit-triples). Note that since we can decompose ΠL_n^{Mill} into calls to Π^{AND}, $\Pi^{\text{AND}'}$ and Π^{ANDCorr} and local XOR operations, we prove the security of ΠL_n^{Mill} in the $\mathcal{F}^{\text{BB}}$-hybrid model, where $\mathcal{F}^{\text{BB}}$ is the set of ideal functionalities of our building blocks, i.e., $\mathcal{F}^{\text{BB}} = \{\mathcal{F}^{\text{AND}}, \mathcal{F}^{\text{AND}'}, \mathcal{F}^{\text{ANDCorr}}\}$. We replace all calls to Π^{AND}, $\Pi^{\text{AND}'}$ and Π^{ANDCorr} in ΠL_n^{Mill} with calls to their ideal functionalities to get a new protocol $\widehat{\Pi L}_n^{\text{Mill}}$. $\widehat{\Pi L}_n^{\text{Mill}}$ computes a comparison circuit $\hat{C}$ consisting of $\mathcal{F}^{\text{AND}}$, $\mathcal{F}^{\text{AND}'}$, $\mathcal{F}^{\text{ANDCorr}}$ and XOR gates (which only require a local XOR). Every wire in $\hat{C}$ either ① holds a value that is *only* a function of P_0/P_1's private input x/y and can thus be computed locally by P_0/P_1, or ② holds a secret share of the corresponding wire in C. This follows from ① the definition of ideal functionalities in $\mathcal{F}^{\text{BB}}$, and ② if the inputs to the local XOR operations are secret shares, then the outputs are also secret shares. Thus, $\hat{C}$ outputs secret shares of $\mathbf{1}\{x < y\}$ and so does $\widehat{\Pi L}_n^{\text{Mill}}$.

We now show the security of $\widehat{\Pi L}_n^{\text{Mill}}$ with respect to $\mathcal{F}_n^{\text{Mill}}$. In $\widehat{\Pi L}_n^{\text{Mill}}$, P_0's view consists of random bits that it receives as outputs of calls to ideal functionalities in $\mathcal{F}^{\text{BB}}$. Parse $x = x_1 || x_0$ and $y = y_1 || y_0$, where x_1, y_1, x_0, y_0 all have the same length. We recall our

DReLU ΠL_n^{DReLU}

$\mathsf{Gen}L_n^{\mathsf{DReLU}}$:

1: $(k_0^{\mathsf{wrap}}, k_1^{\mathsf{wrap}}) \leftarrow \mathsf{Gen}L_{n-1}^{\mathsf{wrap}}$
2: For $b \in \{0,1\}, k_b = k_b^{\mathsf{wrap}}$

$\mathsf{Eval}L_{n-f,n}^{\mathsf{ReLUExt}}(b, k_b, x_b)$:

1: Parse $k_b = k_b^{\mathsf{wrap}}$
2: $c_b \leftarrow \mathsf{Eval}L_{n-1}^{\mathsf{wrap}}(x_b \mod 2^{n-1})$
3: **return** $d_b = \mathsf{MSB}(x_b) \oplus c_b \oplus b$

Figure 6: LSS protocol for DReLU_n.

ReLU ΠL_n^{ReLU}

$\mathsf{Gen}L_n^{\mathsf{ReLU}}$:

1: $(k_0^{\mathsf{DReLU}}, k_1^{\mathsf{DReLU}}) \leftarrow \mathsf{Gen}L_n^{\mathsf{DReLU}}$
2: $(k_0^{\mathsf{sel}}, k_1^{\mathsf{sel}}) \leftarrow \mathsf{Gen}_n^{\mathsf{select}}$
3: For $b \in \{0,1\}, k_b = k_b^{\mathsf{DReLU}} || k_b^{\mathsf{sel}}$

$\mathsf{Eval}L_{n-f,n}^{\mathsf{ReLUExt}}(b, k_b, x_b)$:

1: Parse $k_b = k_b^{\mathsf{DReLU}} || k_b^{\mathsf{sel}}$
2: $d_b \leftarrow \mathsf{Eval}L_n^{\mathsf{DReLU}}(b, k_b^{\mathsf{DReLU}}, x_b)$
3: **return** $z_b \leftarrow \mathsf{Eval}_n^{\mathsf{select}}(b, k_b^{\mathsf{sel}}, d_b, x_b)$

Figure 7: LSS protocol for ReLU_n.

equation for $\mathbf{1}\{x < y\}$ from Section 3.1.

$$\mathbf{1}\{x < y\} = \mathbf{1}\{x_1 < y_1\} \oplus \mathbf{1}\{x_1 = y_1\} \wedge \mathbf{1}\{x_0 < y_0\}$$

Let $r = \mathbf{1}\{x_1 < y_1\}$, $s = \mathbf{1}\{x_1 = y_1\}$ and $t = \mathbf{1}\{x_0 < y_0\}$. At the very top of the comparison tree, following the equation above, P_0 securely computes $r \oplus s \wedge t$. Let r_0, s_0, t_0 denote the shares of r, s, t held by P_0. P_0 makes a call to $\mathcal{F}^{\mathsf{AND}}$ with s_0 and t_0 as input. It XORs $\mathcal{F}^{\mathsf{AND}}$'s output u_0 with r_0 to get the final output of the protocol.

We construct a simulator $\hat{\mathcal{S}}_0^{\mathsf{Mill}}$ to simulate P_0's view in $\widehat{\Pi L}_n^{\mathsf{Mill}}$ given x and z_0, which is $\mathcal{F}^{\mathsf{Mill}}$'s output for P_0 (an identical simulator can be constructed for P_1). To simulate P_0's view as described above, $\hat{\mathcal{S}}_0^{\mathsf{Mill}}$ first samples random bits as outputs of all calls to ideal functionalities in $\mathcal{F}^{\mathsf{BB}}$ *except* the last call to $\mathcal{F}^{\mathsf{AND}}$. To set $\mathcal{F}^{\mathsf{AND}}$'s output, $\hat{\mathcal{S}}_0^{\mathsf{Mill}}$ faithfully computes the bit r_0 as P_0 would in a real execution of $\widehat{\Pi L}_n^{\mathsf{Mill}}$. To do this it uses P_0's private input x and the view simulated thus far. It then sets the output of $\mathcal{F}^{\mathsf{AND}}$ to $r_0 \oplus z_0$. Since z_0 is a random bit from the definition of $\mathcal{F}_n^{\mathsf{Mill}}$, $r_0 \oplus z_0$ is also a random bit. This exactly mimics the output of $\mathcal{F}^{\mathsf{AND}}$. With this, P_0's output in the simulated view matches P_0's output from $\mathcal{F}_n^{\mathsf{Mill}}$. Indistinguishability of the joint distribution of the simulated view and the outputs of $\mathcal{F}_n^{\mathsf{Mill}}$ and the joint distribution of P_0's view in $\widehat{\Pi L}_n^{\mathsf{Mill}}$ and the outputs of $\widehat{\Pi L}_n^{\mathsf{Mill}}$ follows from the fact that $\widehat{\Pi L}_n^{\mathsf{Mill}}$ outputs secret shares of $\mathbf{1}\{x < y\}$ as argued previously. Thus, $\widehat{\Pi L}_n^{\mathsf{Mill}}$ is secure in the $\mathcal{F}^{\mathsf{BB}}$-hybrid model. Security of ΠL_n^{Mill} in the standard model follows from initializing the ideal functionalities in $\mathcal{F}^{\mathsf{BB}}$ with their corresponding secure protocols and invoking the sequential composition theorem [19, 48].

D.2 Security proof of ΠL_n^{wrap}

The ideal functionality $\mathcal{F}_n^{\mathsf{wrap}}$ takes inputs x, y from P_0, P_1 respectively and returns shares of $\mathsf{wrap}_n(x, y)$. We obtain $\widehat{\Pi L}_n^{\mathsf{wrap}}$ by replacing the call to ΠL_n^{Mill} in ΠL_n^{wrap} with a call to $\mathcal{F}_n^{\mathsf{Mill}}$. Security of $\widehat{\Pi L}_n^{\mathsf{wrap}}$ in the $\mathcal{F}_n^{\mathsf{Mill}}$-hybrid model follows from the correctness of the expression for wrap_n in Section 3.1 and the definition of $\mathcal{F}_n^{\mathsf{Mill}}$ (which returns secret shares). Security of ΠL_n^{wrap} follows from securely instantiating $\mathcal{F}_n^{\mathsf{Mill}}$ with ΠL_n^{Mill}.

E LSS-based ReLU

Over reals, ReLU is defined as $\mathsf{ReLU}(x) = \max(x, 0)$. When $x \in \mathbb{U}_N$ is interpreted as a signed value in 2's complement representation, $\mathsf{ReLU}_n(x) = x \cdot \mathsf{DReLU}_n(x)$, where $\mathsf{DReLU}_n(x) = \mathbf{1}\{x < 2^{n-1}\}$. To compute DReLU_n, we use the approach followed by CryptFlow2 [61]. Let $\mathsf{MSB}(\cdot)$ denote the most significant bit. Consider $x, x_0, x_1 \in \mathbb{U}_N$ such that $x = x_0 + x_1 \mod N$. For $b \in \{0,1\}$, parse $x_b = \mathsf{MSB}(x_b)||y_b$ where $\mathsf{MSB}(x_b) \in \{0,1\}$ and $y_b \in \{0,1\}^{n-1}$. Define $\mathsf{carry} := \mathbf{1}\{y_0 + y_1 > 2^{n-1} - 1\} = \mathsf{wrap}_{n-1}(y_0, y_1)$. Then,

$$\mathsf{MSB}(x) = \mathsf{MSB}(x_0) \oplus \mathsf{MSB}(x_1) \oplus \mathsf{carry}$$
$$\mathsf{DReLU}_n(x) = 1 \oplus \mathsf{MSB}(x)$$

In our context, parties P_0 and P_1 hold secret shares x_0, x_1 of x. Using above equations, to securely compute DReLU, it suffices to securely compute carry that can be computed using $\Pi L_{n-1}^{\mathsf{wrap}}$. Given boolean shares of $\mathsf{DReLU}_n(x)$, we can select between 0 and x using Π_n^{select}. We describe our protocols for DReLU_n and ReLU_n formally in Figures 6 and 7 that achieve the following cost.

Theorem 3. *The protocol* ΠL_n^{DReLU} *in Figure 6 securely computes* DReLU_n *with* $\Gamma(\Pi L_n^{\mathsf{DReLU}}) = \Gamma(\Pi L_{n-1}^{\mathsf{wrap}})$ *for* $\Gamma \in \{\mathsf{keysize}, \mathsf{comm}, \mathsf{rounds}\}$. *Moreover, the protocol* ΠL_n^{ReLU} *in Figure 7 securely computes* ReLU_n *with* $\Gamma(\Pi L_n^{\mathsf{ReLU}}) = \Gamma(\Pi L_n^{\mathsf{DReLU}}) + \Gamma(\Pi_n^{\mathsf{select}})$ *for* $\Gamma \in \{\mathsf{keysize}, \mathsf{comm}, \mathsf{rounds}\}$.

F LSS-based Stochastic Truncation

F.1 Proof of Lemma 1

Proof. For $b \in \{0,1\}$, let $w_b \in \mathbb{U}_{2^{n-f}}$ be such that $x_b = w_b \cdot 2^f + y_b$. Alternately, $w_b = \mathsf{TR}_{n,f}(x_b)$. Then

$$\begin{aligned}
\mathsf{TR}_{n,f}(x) &= \frac{x_0 + x_1 - 2^n \cdot \mathsf{wrap}_n(x_0, x_1)}{2^f} \mod 2^{n-f} \\
&= \frac{x_0 + x_1}{2^f} - 2^{n-f} \cdot \mathsf{wrap}_n(x_0, x_1) \mod 2^{n-f} \\
&= \frac{w_0 \cdot 2^f + z_0 + w_1 \cdot 2^f + z_1}{2^f} \mod 2^{n-f} \\
&= w_0 + w_1 + \frac{z_0 + z_1}{2^f} \mod 2^{n-f} \\
&= w_0 + w_1 + \mathsf{wrap}_f(z_0, z_1)
\end{aligned}$$

To prove the second part of the lemma, we use the fact that addition modulo 2^f is commutative and associative, and so $z + r \mod 2^f$ can be computed in one of two ways, both of which give

Signed Extension $\Pi L^{\text{SignExt}}_{n-f,n}$

$\text{Gen}L^{\text{SignExt}}_{n-f,n}$:

1: $(k_0^{\text{wrap}}, k_1^{\text{wrap}}) \leftarrow \text{Gen}L^{\text{wrap}}_{n-f}$
2: $(k_0^{\text{B2A}}, k_1^{\text{B2A}}) \leftarrow \text{Gen}^{\text{B2A}}_n$
3: For $b \in \{0,1\}$, $k_b = k_b^{\text{wrap}} || k_b^{\text{B2A}}$

$\text{Eval}L^{\text{SignExt}}_{n-f,n}(b, k_b, y_b)$:

1: Parse k_b as $k_b^{\text{wrap}} || k_b^{\text{B2A}}$
2: $u_b = y_b + b \cdot 2^{n-f-1} \mod 2^{n-f}$
3: $w_b \leftarrow \text{Eval}L^{\text{wrap}}_{n-f}(b, k_b^{\text{wrap}}, u_b)$
4: $w'_b \leftarrow \text{Eval}^{\text{B2A}}_n(b, k_b^{\text{B2A}}, w_b)$
5: **return** $z_b = \text{extend}(u_b, n) - 2^{n-f} \cdot w'_b - b \cdot 2^{n-f-1}$

Figure 8: LSS protocol for $\text{SignExt}_{n-f,n}$.

us identical results. Thus,

$$
\begin{aligned}
& z + r \mod 2^f \\
&= ((z_0 + z_1) \mod 2^f + (r_0 + r_1) \mod 2^f) \mod 2^f \\
&= ((z_0 + r_0) \mod 2^f + (z_1 + r_1) \mod 2^f) \mod 2^f \\
&\implies z_0 + z_1 - 2^f \cdot \text{wrap}_f(z_0, z_1) + r_0 + r_1 - 2^f \cdot \text{wrap}_f(r_0, r_1) \\
&\quad -2^f \cdot \text{wrap}_f(z, r) \\
&= z_0 + r_0 - 2^f \cdot \text{wrap}_f(z_0, r_0) + z_1 + r_1 - 2^f \cdot \text{wrap}_f(z_1, r_1) \\
&\quad -2^f \cdot \text{wrap}_f(y_0, y_1) \\
&\implies \text{wrap}_f(z, r) = \text{wrap}_f(z_0, r_0) + \text{wrap}_f(z_1, r_1) + \text{wrap}_f(y_0, y_1) \\
&\quad -\text{wrap}_f(z_0, z_1) - \text{wrap}_f(r_0, r_1)
\end{aligned}
$$

□

F.2 LSS-based protocol for Stochastic Truncation

Definition 2. *For $x \in \mathbb{U}_N$ and $z = x \mod 2^f$, we define stochastic truncation as*

$$
\text{StTrunc}_{n,f}(x) = \begin{cases} (x \gg_A f) & \text{with probability } 1 - z \cdot 2^{-f} \\ (x \gg_A f) + 1 & \text{with probability } z \cdot 2^{-f} \end{cases}
$$

Orca showed that stochastic truncation of $x \in \mathbb{U}_N$ by f can be computed as stochastic truncate-reduce by f followed by signed-extension to n bits (see Lemma 2 in [38]). More formally, let signed-extension $\text{SignExt}_{n-f,n}$ be a functionality that takes a $y \in \mathbb{U}_{2^{n-f}}$ as input and returns $z \in \mathbb{U}_N$ such that $\text{int}_n(z) = \text{int}_{n-f}(y)$. Then, for $x \in \mathbb{U}_N$ such that $\text{int}_n(x) \leqslant 2^{n-1} - 2^f$, we have

$$\text{StTrunc}_{n,f}(x) = \text{SignExt}_{n-f,n}(\text{stTR}_{n,f}(x))$$

We describe our protocol for signed-extension in Appendix F.3 that results in the following cost for our protocol $\Pi L^{\text{StTrunc}}_{n,f}$ for stochastic truncation that invokes the protocol for stochastic truncate-reduce followed by the protocol for signed-extension.

Theorem 4. *Let $x \in \mathbb{U}_N$ with $\text{int}_n(x) \leqslant 2^{n-1} - 2^f$. There exists a protocol $\Pi L^{\text{StTrunc}}_{n,f}$ that securely computes $\text{StTrunc}_{n,f}(x)$ with $\Gamma(\Pi L^{\text{StTrunc}}_{n,f}) = \Gamma(\Pi L^{\text{stTR}}_{n,f}) + \Gamma(\Pi L^{\text{wrap}}_{n-f}) + \Gamma(\Pi^{\text{B2A}}_n)$ for $\Gamma \in$ {keysize, comm, rounds}.*

F.3 LSS-based Signed Extension

To perform signed-extension, we use Lemma 4 which was proved in [29].

Lemma 2. *Let $y, y_0, y_1, u_0, u_1 \in \mathbb{U}_{2^{n-f}}$ be such that $y = (y_0 + y_1) \mod 2^{n-f}$ and $u_b = y_b + b \cdot 2^{n-f-1} \mod 2^{n-f}$ for $b \in \{0,1\}$. Let $w = \text{wrap}_{n-f}(u_0, u_1)$. Then $\text{SignExt}_{n-f,n}(y) = \text{extend}(u_0, n) + \text{extend}(u_1, n) - 2^{n-f} \cdot \text{extend}(w, n) - 2^{n-f-1}$.*

Following the above lemma, let $y_0, y_1 \in \mathbb{U}_{2^{n-f}}$ be the secret shares held by P_0, P_1 of some underlying value $y \in \mathbb{U}_{2^{n-f}}$. For $b \in \{0,1\}$, we have P_b compute $u_b = y_b + b \cdot 2^{n-f-1} \mod 2^{n-f}$. Parties then run $\Pi L^{\text{wrap}}_{n-f}$ to compute boolean shares of $w = \text{wrap}_f(u_0, u_1)$. They subsequently run Π^{B2A}_n on the shares of w to get shares of $w' = \text{extend}(w, n)$. Finally, P_b sets $z_b = \text{extend}(u_b, n) - 2^{n-f} \cdot w_b - b \cdot 2^{n-f-1}$ as its output. Our protocol $\Pi L^{\text{SignExt}}_{n-f,n}$ that executes these steps is shown in Figure 8 and its costs are summarized in the following theorem.

Theorem 5. *The protocol $\Pi L^{\text{SignExt}}_{n-f,n}$ in Figure 8 securely computes $\text{SignExt}_{n-f,n}$ with $\Gamma(\Pi L^{\text{SignExt}}_{n-f,n}) = \Gamma(\Pi L^{\text{wrap}}_{n-f}) + \Gamma(\Pi^{\text{B2A}}_n)$ for $\Gamma \in$ {keysize, comm, rounds}.*

G LSS-based ReLU-Extend

In CNNs, linear layers are often followed by an activation such as ReLU. In fixed-point computation, the output of a linear layer needs to be truncated (to scale down). To reduce cost of truncation followed by ReLU, Orca [38] re-wrote the computation as truncate-reduce followed by ReLUExt, which is defined as $\text{ReLUExt}_{n-f,n}(x) = \text{SignExt}_{n-f,n}(\text{ReLU}_{n-f}(x)) = \text{extend}(\text{ReLU}_{n-f}(x), n)$ for $x \in \mathbb{U}_{2^{n-f}}$. Note that this keeps the functionality intact. Moreover, if the linear layer is followed by Maxpool and ReLU, Orca computes truncate-reduce followed by Maxpool on lower bitwidth $(n-f)$ followed by ReLUExt that outputs in n-bits.

To compute ReLUExt securely, we prove the following lemma (see Appendix G.1) that expresses $\text{ReLUExt}(x)$ as computations on secret shares of x.

Lemma 3. *For $x_0, x_1, x \in \mathbb{U}_{2^{n-f}}$ such that $x = x_0 + x_1 \mod 2^{n-f}$, let $d = \mathbf{1}\{x < 2^{n-f-1}\}$, $w = \text{wrap}_{n-f}(x_0, x_1)$ and $w' = \text{extend}(w, n)$. Let $y := \text{extend}(x_0, n) + \text{extend}(x_1, n) - 2^{n-f} \cdot w'$. Then $\text{ReLUExt}_{n-f,n}(x) = \text{select}_n(d, y)$.*

Using the above lemma, we can get an LSS-based secure protocol trivially for ReLUExt that does 2 secure comparisons, one each for DReLU bit d and wrap computation w. We optimize this further to only require a single secure comparison as follows[8]: We build on our construction for ReLU in Appendix E. Consider $x, x_0, x_1 \in \mathbb{U}_{2^{n-f}}$ such that $x = x_0 + x_1 \mod 2^{n-f}$. For $b \in \{0,1\}$, $x_b = m_b || y_b$, where $m_b = \text{MSB}(x_b)$. Define $\text{carry} = \text{wrap}_{n-f-1}(y_0, y_1)$. Then, we compute $d = \text{DReLU}(x) = \mathbf{1}\{x < 2^{n-f-1}\} = m_0 \oplus m_1 \oplus \text{carry} \oplus 1$.

[8]This idea is similar in spirit to MSB-to-Wrap optimization in [60].

ReLU-Extend $\Pi L^{\mathsf{ReLUExt}}_{n-f,n}$

$\mathsf{Gen}L^{\mathsf{ReLUExt}}_{n-f,n}$:

1: $(k_0^{\mathsf{wrap}}, k_1^{\mathsf{wrap}}) \leftarrow \mathsf{Gen}L^{\mathsf{wrap}}_{n-f-1}$
2: $r^{(c)}, r^{(0)}, r^{(1)}, \xleftarrow{\$} \{0,1\}$
3: $t = r^{(0)} \oplus r^{(1)}$
4: $s = (r^{(0)} \wedge r^{(1)}) \oplus (r^{(c)} \wedge t)$; $r^{(d)} = r^{(c)} \oplus t$
5: share $r^{(c)}$, s
6: $(k_0^{\mathsf{B2A}}, k_1^{\mathsf{B2A}}) \leftarrow \mathsf{Gen}_n^{\mathsf{B2A}}$
7: $(k_0^{\mathsf{select}}, k_1^{\mathsf{select}}) \leftarrow \mathsf{Gen}_n^{\mathsf{select}}$
8: For $b \in \{0,1\}$, $k_b = k_b^{\mathsf{wrap}} || r_b^{(c)} || r^{(b)} || s_b || k_b^{\mathsf{B2A}} || k_b^{\mathsf{select}}$

$\mathsf{Eval}L^{\mathsf{ReLUExt}}_{n-f,n}(b, k_b, x_b)$:

1: Parse $k_b = k_b^{\mathsf{wrap}} || r_b^{(c)} || r^{(b)} || s_b || k_b^{\mathsf{B2A}} || k_b^{\mathsf{select}}$
2: $c_b \leftarrow \mathsf{Eval}L^{\mathsf{wrap}}_{n-f-1}(b, k_b^{\mathsf{wrap}}, x_b \mod 2^{n-f-1})$
3: $\hat{m}^{(b)} = \mathsf{MSB}(x_b) \oplus r^{(b)}$
4: $\hat{c}_b = c_b \oplus r_b^{(c)}$
5: $\hat{c} \leftarrow$ reconstruct $(\hat{c}_b)$; Send $\hat{m}^{(b)}$ to P_{1-b} and receive $\hat{m}^{(1-b)}$ from P_{1-b}.
6: $q^{(b)} = \hat{c} \oplus \hat{m}^{(1-b)}$, $u = \hat{m}^{(0)} \oplus \hat{m}^{(1)}$
7: $v = (\hat{m}^{(0)} \wedge \hat{m}^{(1)}) \oplus (\hat{c} \wedge u)$
8: $w_b = b \cdot v \oplus (r^{(b)} \wedge q^{(b)}) \oplus (r_b^{(c)} \wedge u) \oplus s_b$
9: $w'_b \leftarrow \mathsf{Eval}_n^{\mathsf{B2A}}(b, k_b^{\mathsf{B2A}}, w_b)$
10: $d_b = m^{(b)} \oplus c_b \oplus b$
11: $y_b = \mathsf{extend}(x_b, n) - 2^{n-f} \cdot w'_b$
12: **return** $z_b \leftarrow \mathsf{Eval}_n^{\mathsf{select}}(b, k_b^{\mathsf{select}}, d_b, y_b)$

Figure 9: LSS protocol for $\mathsf{ReLUExt}_{n-f,n}$

Hence, to compute d it suffices to compute carry, which requires a single invocation of wrap_{n-f-1}. Next, we reduce computation of $w = \mathsf{wrap}_{n-f}(x_0, x_1)$ to carry by observing

$$w = \mathsf{wrap}_{n-f}(x_0, x_1) = (m_0 \wedge m_1) \oplus (\mathsf{carry} \wedge m_0) \oplus (\mathsf{carry} \wedge m_1)$$

Given the above equation, w can be computed using bitwise AND operations resulting in boolean shares of w. These can be converted to arithmetic shares over $\mathbb{U}_N$ using Π_n^{B2A}. We describe the protocol formally in Appendix G.2 that achieves the cost summarized below.

Theorem 6. $\Pi^{\mathsf{ReLUExt}}_{n-f,n}$ *securely computes* $\mathsf{ReLUExt}_{n-f,n}$ *with* $\mathsf{keysize}(\Pi^{\mathsf{ReLUExt}}_{n-f,n}) = \mathsf{keysize}(\Pi L^{\mathsf{wrap}}_{n-f-1}) + 4n + 3$, $\mathsf{comm}(\Pi^{\mathsf{ReLUExt}}_{n-f,n}) = \mathsf{comm}(\Pi L^{\mathsf{wrap}}_{n-f-1}) + 2n + 4$ *and* $\mathsf{rounds}(\Pi^{\mathsf{ReLUExt}}_{n-f,n}) = \mathsf{rounds}(\Pi L^{\mathsf{wrap}}_{n-f-1}) + 3$.

G.1 Proof of Lemma 3

We start by showing how to compute *zero-extension*. The zero-extension functionality $\mathsf{ZeroExt}_{n-f,n}$ takes $x \in \mathbb{U}_{2^{n-f}}$ as input and returns $\mathsf{extend}(x, n) \in \mathbb{U}_N$ as output. To compute $\mathsf{ZeroExt}_{n-f,n}$, we rely on the following lemma (proved in [29]).

Lemma 4. *Let* $x, x_0, x_1 \in \mathbb{U}_{2^{n-f}}$ *be such that* $x = (x_0 + x_1) \mod 2^{n-f}$. *Let* $w = \mathsf{wrap}_{n-f}(x_0, x_1)$ *and* $w' = \mathsf{extend}(w, n)$. *Then* $\mathsf{ZeroExt}_{n-f,n}(x) = \mathsf{extend}(x_0, n) + \mathsf{extend}(x_1, n) - 2^{n-f} \cdot w'$.

We now present our proof of Lemma 3.

Proof. When the DReLU bit $d = 0$, the output of $\mathsf{ReLU}_{n-f}(x)$, and thus $\mathsf{ReLUExt}_{n-f,n}(x)$ is 0. When $d = 1$, $\mathsf{ReLUExt}_{n-f,n}(x) = \mathsf{ZeroExt}_{n-f,n}(x)$. Thus, when $d = 1$, we have from Lemma 4 that $\mathsf{ReLUExt}_{n-f,n}(x) = x_0 + x_1 - 2^{n-f} \cdot w'$. This concludes the proof. □

G.2 Protocol Description

Our protocol is given in Figure 9. We compute carry using our protocol for wrap over $n - f - 1$ bits which is used to compute both d and w. Computation of d is local with XORs. The boolean formula for w requires us to compute bitwise XOR operations, which are local, and bitwise AND operations, which we compute using Beaver bit-triples (Section 2.3). Once we have boolean shares of w (output by our protocol for bitwise AND), we feed them into Π_n^{B2A} to get shares w'_0, w'_1 of $w' = \mathsf{extend}(w, n)$. This allows us to compute shares of $y = \mathsf{extend}(x_0, n) + \mathsf{extend}(x_1, n) - 2^{n-f} \cdot w'$ with, for $b \in \{0,1\}$, party P_b computing $y_b = \mathsf{extend}(x_b, n) - 2^{n-f} \cdot w'_b$. Once we have shares of y, we use the DReLU bit to choose between y and 0 with Π_n^{select}.

H Details of LSS Key Compression

We illustrate the standard technique of compressing Beaver triples that results in significant keysize reduction while computing secure AND. Secure AND is used liberally within our comparison protocol ΠL^{Mill}. Let $u, v, w \in \{0,1\}$ be a bit-triple such that $w = u \wedge v$. Ordinarily, P_0 gets shares u_0, v_0, w_0 and P_1 gets shares u_1, v_1, w_1. Without compression, both parties need to store 3 bits of keys. Let F be a pseudorandom function (PRF). The dealer shares PRF keys k_0, k_1 with P_0, P_1 in the offline phase. Now, the dealer picks a (publicly known) value i and a party b, and sets the bits $u_{1-b}, v_{1-b}, w_{1-b}$ to be the output of $F(k_{1-b}, i)$. These bits can be computed by P_{1-b} in the online phase and need not be sent explicitly by the dealer. Similarly, u_b, v_b are set to be the output of $F(k_b, i)$ and can be computed by P_b instead of being sent by the dealer. With this, the dealer only needs to send w_b to P_b. Thus, out of the overall 6 bits of correlation, only 1 bit needs to be stored explicitly (by party P_b). We can share keys for several ANDs by simply having the dealer, P_0 and P_1 increment i when needed. We additionally do load balancing between P_0 and P_1, i.e. to share the key for m ANDs, the dealer picks $b = 0$ for $\frac{m}{2}$ ANDs, and $b = 1$ for $\frac{m}{2}$ ANDs. This means that in contrast to $3m$ bits before, each party now only needs to store $\frac{m}{2}$ bits. Thus, we can get a key size reduction of 6× compared to naively sharing Beaver triples. This basic idea can be extended to all our protocols.

I Details of $\mathsf{FSS}^{\mathsf{M}}$

I.1 2PC with pre-processing based on FSS

Here we provide a brief description of FSS-based 2PC in the pre-processing model and refer the reader to Orca [38] for a detailed explanation.

I.1.1 Function Secret Sharing. For a function f, a function secret sharing (FSS) [15,16] scheme provides a pair of algorithms (Gen, Eval) such that Gen splits f into function shares f_0, f_1, and Eval on input $b \in \{0,1\}$, f_b and x produces y_b. The correctness guarantee requires

that $y_0 + y_1 = f(x)$. Security property requires that each function share f_b hides f. Here, f_0, f_1 are referred to as the function keys, and size of each of them is referred to as the *keysize*.

I.1.2 2PC with FSS. Boyle et al. [17] described how FSS can be used to construct 2PC protocols with an offline/pre-processing phase and an online phase. At a high level, consider a circuit with gates $\{g_i\}$. To realize the circuit securely, it suffices if we have protocols for each gate where parties start with secret shares of input to g_i and generate secret shares of output of g_i. For each gate consider the corresponding offset gate $g_i^{[r_i]}(\hat{x}) = g(\hat{x} - r_i)$. Then, in the offline phase, the dealer gives out shares of r_i and FSS keys for $g_i^{[r_i]}$. In the online phase, P_0 and P_1 hold secret shares of x. Given secret shares of r_i, they compute shares of $\hat{x} = x + r_i$, and reconstruct $\hat{x}$. Then, they locally evaluate their FSS key on $\hat{x}$ to learn shares of $g_i(x)$.

I.1.3 FSS for Comparison. A key FSS scheme that we will use in all our protocols is the FSS scheme for comparison. We define the comparison function $f^{<}_{\alpha,\beta} : \mathbb{U}_N \to \mathbb{G}$ that, for input $x \in \mathbb{U}_N$, returns $\beta \in \mathbb{G}$ if $x < \alpha$ and 0 otherwise. When $\mathbb{G}^{\text{out}} = \{0,1\}$ and $\beta = 1$ (as in our case), Grotto [67] shows how to construct an FSS scheme $\mathsf{LtFSS}_n = (\mathsf{Gen}^{<}_n, \mathsf{Eval}^{<}_n)$ for $f^{<}_{\alpha,\beta}$ that is based on Distributed Point Functions (DPFs) [16] and has the following cost.

Theorem 7 (FSS scheme for comparison [16,67]). *Let λ be the computational security parameter. Let $G : \{0,1\}^\lambda \to \{0,1\}^{2\lambda+2}$ be a PRG. Let $\nu = \log(\lambda+1)$. When $n > \nu$, there exists an FSS scheme $\mathsf{LtFSS}_n = (\mathsf{Gen}^{<}_n, \mathsf{Eval}^{<}_n)$ such that $\forall x, \alpha \in \mathbb{U}_N$:*

$$(k^{<}_0, k^{<}_1) \leftarrow \mathsf{Gen}^{<}_n(\alpha)$$
$$\implies \mathsf{Eval}^{<}_n(0, x, k^{<}_0) + \mathsf{Eval}^{<}_n(1, x, k^{<}_1) = f^{<}_{\alpha,1}(x)$$

LtFSS_n has key size $(n-\nu)\cdot(\lambda+2)+2\lambda$. $\mathsf{Gen}^{<}_n$ invokes the PRG $2\cdot(n-\nu)$ times and $\mathsf{Eval}^{<}_n$ invokes the PRG $(n-\nu)$ times.

We set $\lambda = 127$ and use two calls to AES-128 in counter mode to realize the slightly more than length-doubling PRG. When $\mathsf{Eval}^{<}_n$ invokes the PRG, it only needs a single AES call since it only uses either the first or second half of the PRG output. We refer to this as a *half-PRG call*. Orca [38] uses a different FSS scheme called the Distributed Comparison Function (or DCF) for comparison. For n-bit inputs, LtFSS_n is more efficient than DCF_n when $\mathbb{G}^{\text{out}} = \{0,1\}$. LtFSS_n requires 2× fewer AES calls in the online phase compared to DCF_n. LtFSS_n also has a $1.02 - 1.2\times$ smaller keysize than DCF_n (depending on n).

I.2 FSS-based protocols for secure ML

We now elaborate on our FSS protocols for stochastic truncate-reduce and ReLU-Extend. As previously stated, the rest of our protocols can be trivially obtained by replacing DCF in Orca [38]'s protocols with LtFSS.

I.2.1 Stochastic truncate-reduce. We exactly follow Orca's [38] mathematical logic for computing stochastic truncate-reduce with two changes.

Lemma 5 ([38]). *Let $x, r^{(x)}, \hat{x} \in \mathbb{U}_N$ be such that $\hat{x} = x + r^{(x)} \mod N$. Let $z = x \mod 2^f$, $\hat{z} = \hat{x} \mod 2^f$ and $r^{(z)} = r^{(x)} \mod 2^f$. For $s \xleftarrow{\$} \mathbb{U}_{2^f}$, let $\hat{s} = s + r^{(z)} \mod 2^f$. Then,*

$$\mathsf{stTR}_{n,f}(x) = \underbrace{\mathsf{TR}_{n,f}(\hat{x})}_{P_1} + \underbrace{\mathbf{1}\{\hat{z} > \hat{s}\}}_{FSS} - \underbrace{\mathsf{TR}_{n,f}(r^{(x)}) - \mathbf{1}\{\hat{s} < r^{(z)}\}}_{Dealer}$$

Orca computed the second term using a DCF-like FSS scheme for greater-than comparison, $f^{>}_{\hat{s},1}$. While the ideas behind LtFSS can potentially be extended (in a non-black-box manner) to compute greater-than as well, we do something simpler. We note that

$$\mathbf{1}\{\hat{z} > \hat{s}\} = 1 - \mathbf{1}\{\hat{z} \leqslant \hat{s}\}$$
$$= 1 - \underbrace{\mathbf{1}\{\hat{z} < (\hat{s}+1) \mod 2^f\}}_{\text{Computed via FSS}} - \underbrace{\mathbf{1}\{\hat{s} = 2^f - 1\}}_{\text{Computed by Dealer}})$$

Next, we reduce the communication of Orca as follows: Orca reconstructs $\hat{x}$ using $2n$ bits of communication, and parties locally compute $\hat{z} = \hat{x} \mod 2^f$. We note that while both parties need to learn $\hat{z}$, only P_1 needs to learn $\hat{x}$. We can achieve this with only $(n+f)$ bits of communication as follows: P_0 computes $\hat{x}_0 = x_0 + r_0^{(x)}$ and sends it to P_1. Also, P_1 computes $\hat{z}_1 = x_1 + r^{(x)} \mod 2^f$ and sends to P_0. Then, P_0 computes $\hat{z} = \hat{x}_0 + \hat{z}_1 \mod 2^f$. Also, P_1 computes $\hat{x} = x_1 + r_1^{(x)} + \hat{x}_0$ and $\hat{z} = \hat{x} \mod 2^f$.

Our protocol for stochastic truncate-reduce is given in Figure 10 that satisfies the following theorem.

Theorem 8. *There exists a protocol $\Pi F^{\mathsf{stTR}}_{n,f}$ that securely computes $\mathsf{stTR}_{n,f}$ with $\mathsf{keysize}(\Pi F^{\mathsf{stTR}}_{n,f}) = \mathsf{keysize}(\mathsf{LtFSS}_f) + \mathsf{keysize}(\Pi^{\mathsf{B2A}}_{n-f}) + 2n - f$, $\mathsf{comm}(\Pi F^{\mathsf{stTR}}_{n,f}) = \mathsf{comm}(\Pi^{\mathsf{B2A}}_{n-f}) + n + f$ and $\mathsf{rounds}(\Pi F^{\mathsf{stTR}}_{n,f}) = \mathsf{rounds}(\Pi^{\mathsf{B2A}}_{n-f}) + 1$.*

I.2.2 ReLU-Extend. Let $x, r, \hat{x} \in \mathbb{U}_{2^{n-f}}$ be such that $\hat{x} = x + r \mod 2^{n-f}$. Let $d = \mathsf{DReLU}(x)$ and $\tilde{w} = \mathbf{1}\{\hat{x} < r\}$. Orca uses one DCF key and two evaluations of DCF to compute the DReLU bit d and the bit $\tilde{w}$ for the secret value x. Then, it uses $(d, \tilde{w})$ to perform a selection from a table of 4 values. To enable this, $d, \tilde{w} \in \mathbb{U}_4$, that is both d and $\tilde{w}$ are 2-bit outputs of secure comparisons done with DCF. One straightforward way to modify Orca's protocol to use comparisons with 1-bit outputs (so we can use LtFSS) is to first obtain d and $\tilde{w}$ as single bit values, and then use our protocol for Boolean-to-Arithmetic to convert $(d, \tilde{w})$ to values in $\mathbb{U}_4$. While this would work, it requires an additional round of interaction and 4 bits of online communication over Orca. Instead, below, we build on the ideas described in Section G to require a single evaluation of LtFSS (instead of 2 evaluations of DCF) and also avoid the above overhead of extension by re-designing the logic of ReLU-Extend to work directly with one-bit comparison outputs. Overall compared to ReLU-Extend in Orca, for $n = 64$ and $f = 24$, we have a marginally (1.05×) lower keysize, 4× fewer PRG calls, the same number of rounds, and 6 fewer bits of communication.

We reuse our ideas from Section G to design our new FSS-based protocol for ReLU-Extend. Let $x, r^{(x)}, \hat{x} \in \mathbb{U}_{2^{n-f}}$ be such that $x = \hat{x} - r^{(x)} \mod 2^{n-f}$. We interpret x as being shared between the dealer in the offline phase with share $-r^{(x)}$ and the two parties in the online phase with share $\hat{x}$. As we did in Section G, we define $y_0 = \hat{x} \mod 2^{n-f-1}$, $y_1 = -r^{(x)} \mod 2^{n-f-1}$ and carry $= \mathbf{1}\{y_0 + y_1 > 2^{n-f-1} - 1\}$. We compute carry as $\mathbf{1}\{2^{n-f-1} - 1 - y_0 < y_1\}$

Stochastic Truncate-Reduce $\Pi F^{\mathsf{stTR}}_{n,f}$

$\mathsf{Gen}F^{\mathsf{stTR}}_{n,f}$:

1: $r^{(x)} \xleftarrow{\$} \mathbb{U}_{2^n}$; share $r^{(x)}$
2: $r^{(z)} = r^{(x)} \mod 2^f$
3: $s \xleftarrow{\$} \mathbb{U}_{2^f}$; $\hat{s} = s + r^{(z)} \mod 2^f$
4: $t = \hat{s} + 1 \mod 2^f$
5: $(k_0^<, k_1^<) \leftarrow \mathsf{Gen}^<_f(t)$
6: $q = 1 - \mathsf{TR}_{n,f}(r^{(x)}) - \mathbf{1}\{\hat{s} < r^{(z)}\} - \mathbf{1}\{\hat{s} = 2^f - 1\} \mod 2^{n-f}$
7: share q
8: $(k_0^{\mathsf{B2A}}, k_1^{\mathsf{B2A}}) \leftarrow \mathsf{Gen}^{\mathsf{B2A}}_{n-f}$
9: For $b \in \{0,1\}$, $k_b = r_b^{(x)} || k_b^< || k_b^{\mathsf{B2A}} || q_b$

$\mathsf{Eval}F^{\mathsf{stTR}}_{n,f}(b, k_b, x_b)$:

1: Parse k_b as $r_b^{(x)} || k_b^< || k_b^{\mathsf{B2A}} || q_b$
2: $\hat{x}_b = x_b + r_b^{(x)} \mod 2^n$; $\hat{z}_b = \hat{x}_b \mod 2^f$
3: P_0 sends $\hat{x}_0$ to P_1. P_1 sends $\hat{z}_1$ to P_0.
4: P_0 computes $\hat{z} = \hat{z}_0 + \hat{z}_1 \mod 2^f$. P_1 computes $\hat{x} = \hat{x}_0 + \hat{x}_1 \mod 2^n$ and $\hat{z} = \hat{x} \mod 2^f$.
5: $p_b \leftarrow \mathsf{Eval}^<_f(b, k_b^<, \hat{z})$
6: $p'_b \leftarrow \mathsf{Eval}^{\mathsf{B2A}}_{n-f}(b, k_b^{\mathsf{B2A}}, p_b)$
7: **return** $z_b = b \cdot \mathsf{TR}_{n,f}(\hat{x}) - p'_b + q_b$

Figure 10: FSS protocol for $\mathsf{stTR}_{n,f}$

using LtFSS_{n-f-1}. Let $g = -r^{(x)} \mod 2^{n-f}$ and $h = \mathsf{MSB}(g)$. The dealer gives out boolean shares of $r^{(c)} \xleftarrow{\$} \{0,1\}$, and parties reconstruct $\hat{c}$, which is the carry bit masked by $r^{(c)}$. Once parties have $\hat{x}$ and $\hat{c}$, they compute $\hat{d} = \hat{x} \oplus \hat{c}$, which we interpret as the DReLU bit d masked by $r^{(c)} \oplus h \oplus 1$. Now, in Section G, we explicitly compute the wrap bit $w = \mathbf{1}\{\hat{x} + g > 2^{n-f} - 1\}$ via a boolean formula. This boolean formula, when mapped to the current setting, takes $\mathsf{MSB}(\hat{x})$, h and the carry bit as input. In contrast to LSS, where all three inputs of the boolean formula were secret, here, $\mathsf{MSB}(\hat{x})$ is known to the parties, and h is known to the dealer in the offline phase. Only the carry bit is secret. We exploit this to avoid computing w explicitly (since that costs one round and 2 bits of communication). Let $d' = \mathsf{extend}(d, n)$, $w' = \mathsf{extend}(w, n)$, $u = \mathsf{extend}(g, n)$ and $\hat{x}' = \mathsf{extend}(\hat{x}, n)$. Recall that $\mathsf{ReLUExt}_{n-f,n}(x) = \mathsf{select}_n(d, \hat{x}' + u - 2^{n-f} \cdot w') = d' \cdot \hat{x}' + \underline{d' \cdot u - d' \cdot w' \cdot 2^{n-f}}$ (this follows from Lemma 3). We now use the following lemma, which computes the underlined part directly as a function of $\mathsf{MSB}(\hat{x})$, u and carry. We can thus compute $\mathsf{ReLUExt}_{n-f,n}$ with no further interaction after computing carry.

Lemma 6. *Let* $x, r^{(x)}, \hat{x} \in \mathbb{U}_{2^{n-f}}$ *be such that* $\hat{x} = x + r^{(x)} \mod 2^{n-f}$. *Let* $u = \mathsf{extend}(-r^{(x)} \mod 2^{n-f}, n)$, $s = \mathsf{extend}(\mathsf{MSB}(-r^{(x)}), n)$, $s' = 1 - s$, $t = u \cdot s$ *and* $t' = u \cdot s'$. *Let* $d' = \mathsf{extend}(\mathsf{DReLU}(x), n)$, $\mathsf{carry} = \mathbf{1}\{\hat{x} - r^{(x)} > 2^{n-f} - 1\}$ *and* $\hat{y} = \mathsf{MSB}(\hat{x})$. *Let* $\hat{x}' = \mathsf{extend}(\hat{x}, n)$. *Then,* $\mathsf{ReLUExt}_{n-f,n}(x) = d' \cdot \hat{x}' + F_{s,t}(\mathsf{carry}, \hat{y})$, *where* $F_{s,t}$ *is given by*

$$F_{s,t}(\mathsf{carry}, \hat{y}) = \begin{cases} t' & \mathsf{carry} = 0, \hat{y} = 0 \\ t - 2^{n-f} \cdot s & \mathsf{carry} = 0, \hat{y} = 1 \\ t - 2^{n-f} \cdot s & \mathsf{carry} = 1, \hat{y} = 0 \\ t' - 2^{n-f} \cdot s' & \mathsf{carry} = 1, \hat{y} = 1 \end{cases}$$

Proof. Let $g = -r^{(x)} \mod 2^{n-f}$, $w = \mathbf{1}\{\hat{x} + g > 2^{n-f} - 1\}$ and $w' = \mathsf{extend}(w, n)$. We have from Lemma 3 that

$$\mathsf{ReLUExt}_{n-f,n}(x) = d' \cdot \hat{x}' + \underline{d' \cdot u - d' \cdot w' \cdot 2^{n-f}} \quad (1)$$

Thus, it suffices to show that $F_{s,t}$ correctly computes the underlined part. Given $\hat{y}$ and carry, the DReLU bit d' can be computed as

$$d' = \begin{cases} 1 - s & \mathsf{carry} = 0, \hat{y} = 0 \\ s & \mathsf{carry} = 0, \hat{y} = 1 \\ s & \mathsf{carry} = 1, \hat{y} = 0 \\ 1 - s & \mathsf{carry} = 1, \hat{y} = 1 \end{cases}$$

Similarly, $d' \cdot w'$ can be computed as

$$d' \cdot w' = \begin{cases} 0 & \mathsf{carry} = 0, \hat{y} = 0 \\ s & \mathsf{carry} = 0, \hat{y} = 1 \\ s & \mathsf{carry} = 1, \hat{y} = 0 \\ 1 - s & \mathsf{carry} = 1, \hat{y} = 1 \end{cases}$$

By substituting the above expressions for d' and $d' \cdot w'$ in the underlined part of Equation 1, the lemma follows. □

Following the above lemma, we compute arithmetic shares of d' from the masked DReLU bit (with underlying mask $r^{(c)} \oplus h \oplus 1$) using the expression in Appendix B for Π^{B2A}_n. We compute $F_{s,t}$ by having the dealer secret share a look-up table indexed by the carry and $\hat{y}$ bits. Parties have access to the masked carry bit, so the dealer appropriately rotates the look-up table to ensure that parties look up the correct entry of look-up table in the online phase. Our protocol is given in Figure 11.

Theorem 9. $\Pi^{\mathsf{ReLUExt}}_{n-f,n}$ *securely computes* $\mathsf{ReLUExt}_{n-f,n}$ *with* $\mathsf{keysize}(\Pi^{\mathsf{ReLUExt}}_{n-f,n}) = \mathsf{keysize}(\mathsf{LtFSS}_{n-f-1}) + 6n - f + 1$, $\mathsf{comm}(\Pi^{\mathsf{ReLUExt}}_{n-f,n}) = 2n - 2f + 2$ *and* $\mathsf{rounds}(\Pi^{\mathsf{ReLUExt}}_{n-f,n}) = 2$. *It requires one evaluation of* LtFSS_{n-f-1} *in the online phase.*

J Model details

The architecture of VGG16 is identical to the corresponding plaintext model. Plaintext ResNet-18 and ResNet-50 have batch normalization after convolution. A common optimization implemented during inference (in PyTorch, and in CrypTen) is to fold the weights/biases of batch normalization into the weights/biases of the preceding convolution [80, 82]. This reduces the number of operations required during inference and improves efficiency. Since multiplication is commutative and associative over reals, the function computed after merging the two (linear) layers is *identical* to the one computed before merging, and so merging does *not* affect accuracy. In Table 2, PyTorch accuracy is *with* batch normalization, and fixed-point accuracy is after merging. Since merging causes no accuracy loss, we follow PyTorch and CrypTen and merge convolution and batch normalization in ResNet-18 and ResNet-50. For fairness, we use the same architecture for ResNet-18 and ResNet-50 across all our baselines (CrypTen supports it by default).

ReLU-Extend $\Pi F^{\mathsf{ReLUExt}}_{n-f,n}$

$\mathsf{Gen}F^{\mathsf{ReLUExt}}_{n-f,n}$:

1: $r^{(x)} \xleftarrow{\$} \mathbb{U}_{2^{n-f}}$; share $r^{(x)}$
2: $y_1 = -r^{(x)} \mod 2^{n-f-1}$
3: $(k^<_0, k^<_1) \leftarrow \mathsf{Gen}^<_{n-f-1}(y_1)$
4: $r^{(c)} \xleftarrow{\$} \{0,1\}$; share $r^{(c)}$
5: $g = -r^{(x)} \mod 2^{n-f}$; $h = \mathsf{MSB}(a)$
6: $u = \mathsf{extend}(g, n)$; $s = \mathsf{extend}(h, n)$
7: $r^{(d)} = r^{(c)} \oplus f \oplus 1$
8: $r' = \mathsf{extend}(r^{(d)}, n)$; share r'
9: $s' = 1 - s$
10: $t = u \cdot s$; $t' = u \cdot s'$
11: $\mathbf{e} = \{(t', t - 2^{n-f} \cdot s), (t - 2^{n-f} \cdot s, t' - 2^{n-f} \cdot s')\} \ggg r^{(c)}$
12: For $b \in \{0,1\}$, $k_b = r^{(x)}_b || k^<_b || r^{(c)}_b || r'_b || \mathbf{e}_b$

$\mathsf{Eval}F^{\mathsf{ReLUExt}}_{n-f,n}(b, k_b, x_b)$:

1: Parse $k_b = r^{(x)}_b || k^<_b || r^{(c)}_b || r'_b || \mathbf{e}_b$
2: $\hat{x}_b = x_b + r^{(x)}_b \mod 2^{n-f}$
3: $\hat{x} \leftarrow$ reconstruct $(\hat{x}_b)$
4: $y_0 = \hat{x} \mod 2^{n-f-1}$
5: $c_b \leftarrow \mathsf{Eval}^<_{n-f}(b, k^<_b, 2^{n-f-1} - 1 - y_0)$
6: $\hat{c}_b = c_b \oplus r^{(c)}_b$; $\hat{c} \leftarrow$ reconstruct $(\hat{c}_b)$
7: $\hat{y} = \mathsf{MSB}(\hat{x})$; $\hat{d} = \hat{y} \oplus \hat{c}$
8: $\hat{d}' = \mathsf{extend}(\hat{d}, n)$
9: $d'_b = b \cdot \hat{d}' + (1 - 2\hat{d}') \cdot r'_b$
10: Parse $\mathbf{e}_b[\hat{c}]$ as $(q^{(0)}, q^{(1)})$
11: $\hat{x}' = \mathsf{extend}(\hat{x}, n)$
12: **return** $z_b = d'_b \cdot \hat{x}' + q^{(\hat{y})}$

Figure 11: FSS protocol for $\mathsf{ReLUExt}_{n-f,n}$

K Extending MM to transformers

The ideas driving MM can be extended to secure transformer inference as well. To illustrate, we consider SIGMA [33], which is the current state-of-the-art in secure transformer inference in the preprocessing model. SIGMA designs accuracy-preserving approximations of the complex non-linearities in transformers (e.g. GeLU) and realizes them securely via FSS-based protocols. These approximations use comparisons, linear functions and small look-up tables (LUTs). We provide efficient LSS-based comparison in this work. SIGMA's LUTs are small (most have 2^8 entries) and only need boolean secret-shared vectors. Thus, we have all the building blocks we need to construct an LSS-based protocol suite for secure transformer inference. We can use the techniques outlined in this paper to choose between LSS and FSS based on the deployment scenario and to further employ heterogeneity (mixing LSS and FSS) whenever it is useful.