Efficient Privacy-Preserving Machine Learning with Lightweight Trusted Hardware

Pengzhi Huang
Cornell University
ph448@cornell.edu

Thang Hoang
Virginia Tech
thanghoang@vt.edu

Yueying Li
Cornell University
yl3469@cornell.edu

Elaine Shi
Carnegie Mellon University
runting@gmail.com

G. Edward Suh
NVIDIA / Cornell University
edward.suh@cornell.edu

ABSTRACT

In this paper, we propose a new secure machine learning inference platform assisted by a small dedicated security processor, which will be easier to protect and deploy compared to today’s TEEs integrated into high-performance processors. Our platform provides three main advantages over the state-of-the-art: (i) We achieve significant performance improvements compared to state-of-the-art distributed Privacy-Preserving Machine Learning (PPML) protocols, with only a small security processor that is comparable to a discrete security chip such as the Trusted Platform Module (TPM) or on-chip security subsystems in SoCs similar to the Apple enclave processor. In the semi-honest setting with WAN/GPU, our scheme is 4×-63× faster than Falcon (PoPETs’21) and AriaNN (PoPETs’22) and 3.8×-12× more communication efficient. We achieve even higher performance improvements in the malicious setting. (ii) Our platform guarantees security with abort against malicious adversaries under honest majority assumption. (iii) Our technique is not limited by the size of secure memory in a TEE and can support high-capacity modern neural networks like ResNet18 and Transformer. While previous work investigated the use of high-performance TEEs in PPML, this work represents the first to show that even tiny secure hardware with very limited performance can be leveraged to significantly speed-up distributed PPML protocols if the protocol can be carefully designed for lightweight trusted hardware.

KEYWORDS
Multi-party computation, Secure hardware, Machine learning

1 INTRODUCTION

As the world increasingly relies on machine learning (ML) for everyday tasks, a large amount of potentially sensitive or private data need to be processed by ML learning algorithms. For example, ML models for medical applications may need to use private datasets distributed in multiple nations as inputs [39]. A cloud-based ML services process private data from users with pre-trained models to provide predictions [13, 50]. The data to be shared in these applications are often private and sensitive and must be protected from the risk of leakage. Government regulations may play an essential role as a policy, but cannot guarantee actual protection. We need technical protection for privacy-preserving machine learning (PPML) for strong confidentiality and privacy guarantees.

In this paper, we propose a new PPML framework, named STAMP (Small Trusted hardware Assisted MPc), which enables far more efficient secure multiparty computation (MPC) for machine learning through a novel use of small lightweight trusted hardware (LTH). MPC refers to a protocol that allows multiple participants to jointly evaluate a particular function while preventing their inputs from being revealed to each other. Ever since Yao’s initial studies (later called Garbled Circuit) [96, 97] which gave such a secure protocol in the case of two semi-honest parties, many studies have been conducted to improve the efficiency, to expand to more than two parties, and to ensure the feasibility against malicious behaviors. Recently, there has been significant interest in using and optimizing MPC for secure machine learning computation [42, 58, 71, 88, 89]. However, the overhead for MPC-based PPML is still significant.

For low-overhead secure computation, trusted execution environments (TEEs) in modern microprocessors such as Intel SGX [14] AMD SEV [72] aim to provide hardware-based protection for the confidentiality and integrity of data and code inside. If the TEE protection and the software inside can be trusted, secure machine learning computation can be performed directly inside a TEE with relatively low overhead [41]. The TEE can also be used to improve cryptographic protocols by accelerating bootstrapping [40, 51] or simplifying protocols [2, 12, 20, 40]. However, it is challenging to build a secure environment inside a high-performance processor due to its large trusted computing base (TCB) and complex performance optimizations such as out-of-order execution, speculation, and caching. For example, multiple attacks have been shown for SGX [24, 84, 85]. Moreover, the TEE requires adding hardware protection to each type of computing engines (CPU, GPU, and accelerators), and significant changes to the software stack. As a result, developing and deploying a TEE for a new piece of hardware requires significant effort and time.

In this paper, we propose to leverage a small dedicated security processor, another type of trusted hardware that is widely deployed today, to reduce the MPC overhead. For example, small discrete security chips such as trusted platform module (TPM), Google Titan, and Apple T1 are widely used as a platform root-of-trust. For system-on-chip (SoC) designs, on-chip security subsystems like the Apple enclave processor perform security-critical operations such as secure booting, attestation, and key management.

This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license visit https://creativecommons.org/licenses/by/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Proceedings on Privacy Enhancing Technologies 2024(4), 327–348
© 2024 Copyright held by the owner/author(s).
https://doi.org/10.56553/poptechs-2024-0119
While the high-level idea to combine trusted hardware and MPC has been explored before, we believe this work represents the first to investigate MPC acceleration using a small security processor. Clearly, such lightweight trusted hardware can only provide relatively low performance. The main question is if a low-performance trusted hardware can still be leveraged to provide meaningful speedups for MPC. In the following discussion, we refer to such small security processors as lightweight trusted hardware (LTH).

The key insight we leverage in STAMP is that non-linear operations, which can be performed very efficiently in plaintext, account for the major part of the overhead in MPC. MPC-based deep learning inference is not particularly expensive in computation but introduces large communication overhead due to multi-round data exchanges, especially when the network latency is high. This overhead leads to a very different cost distribution for MPC compared to plaintext computation. Profiling an inference task of AlexNet [44], which represents a classical deep learning model, shows that 85% of total plaintext execution time comes from linear operations such as convolution and fully-connected layers, while for MPC, this portion drops to only 5% with the remaining 95% coming from non-linear operations. Most of those non-linear operations are simple and cheap in plaintext (e.g., ReLU, MaxPooling, which are generally comparisons) with some exceptions (e.g., Softmax). This observation implies that even a lightweight trusted hardware can potentially speed up MPC-based PPML significantly if we can efficiently offload non-linear operations.

STAMP combines the advantages of MPC and trusted hardware by performing linear operations in MPC while leveraging LTH for non-linear operations. To realize this approach, we introduce new MPC protocols that efficiently offload non-linear operations while minimizing communications among multiple parties and between the LTH and an untrusted CPU/GPU. Although simple nonlinear operations can be performed inside the small LTH with sufficiently high performance, expensive operations such as Softmax require higher performance. To address the challenge, STAMP securely offloads parts of the expensive exponentiation operations to an untrusted CPU/GPU. The following describes the main technical contributions and advantages of STAMP.

Insight and performance improvement. STAMP represents the first work to investigate if tiny low-performance security processors can still be leveraged to meaningfully speed-up MPC protocols. Our results demonstrate that even with trust in a tiny piece of discrete secure hardware similar to a TPM, significant speedups can be achieved for privacy-preserving neural network inference when the MPC protocol is carefully redesigned for efficient offloading of non-linear operations. We compared STAMP with three state-of-the-art MPC protocols (Falcon [89], AriasNN [71], and CryptGPU [81]). The results show that STAMP achieves significantly lower inference overhead compared to the state-of-the-art MPC protocols on either CPUs or GPUs, under either a WAN or LAN setting, and using either a discrete security chip (LTH-chip) or a security processor on an SoC (LTH-SoC). STAMP is 4× to 63× faster in the semi-honest WAN/GPU setting, even with the tiny LTH-chip with a low-bandwidth interconnection, and reduces the inter-party communication by 7× to 10×. STAMP can also improve the performance of the MPC-based secure inference in malicious settings. Interestingly, the experimental results show that STAMP (LTH-SoC) can even outperform a protocol that leverages a high-performance TEE (Intel SGX) with secure GPU outsourcing (Goten [61]) thanks to its ability to significantly reduce the inter-party communication. While STAMP can also be used with a high-performance TEE to further improve performance, this result suggests that tiny low-performance secure hardware can indeed be sufficient if it is primarily used for non-linear operations. STAMP provides the most significant performance improvements under GPU/WAN settings when WAN communication represents more of a performance bottleneck compared to GPU-based computation.

Malicious security. STAMP provides security guarantees under the honest-majority setting similar to previous schemes [58, 89], assuming that the majority (2 out of the 3 participants) are behaving honestly. If the corrupted party behaves semi-honestly, the protocol ensures that no information is obtained by any party without reconstructing a value. If a party is actively corrupted and behaves maliciously, we guarantee detection of such a behavior and output “abort” while still keeping the confidentiality of the data with extra steps. We show the security of STAMP using the standard simulation-based paradigm in Appendix C. We implement both semi-honest and malicious protocols in our end-to-end framework.

Prototype implementation. We implemented a functional prototype of both semi-honest and malicious protocols of STAMP in C++. The compilation framework and a small number of pure MPC-based operations (see §3.2 and Appendix A) are based on [89]. The baseline framework was significantly modified to incorporate new non-linear operation protocols, GPUs support, new networks and datasets, and a better socket library. The prototype implementation supports both CPU-only and GPU-assisted settings, and adds the same GPU support to our baseline for a fair comparison.

Evaluation and analysis. We demonstrate STAMP by supporting the secure inference of various networks including AlexNet [44], VGG16 [74], ResNet18 [29] and Transformer [92], over multiple datasets including MNIST [16], CIFAR-10 [43], ImageNet [70] and Wikitext-2 [53], under both WAN and LAN, and semi-honest and malicious settings. We provide theoretical analysis of the overhead and scalability analysis. We perform detailed experimental studies against state-of-the-art MPC protocols, and also protocols leveraging high-performance TEEs for a balanced discussion on the trade-off. We show that even a very small trusted hardware reduces the overhead of MPC protocols significantly while supporting various high-capacity networks, and STAMP can support larger models without extra overhead in most cases.

2 MODEL

System Model. In our system, there are three parties who want to run a common ML model together using inputs from individuals. We assume that the model structure is publicly known. We assume that each party consists of two components: an untrusted machine (CPU/GPU) and an LTH module whose computational power is limited. LTH in each party communicates with each other through its host by establishing pairwise secure communication channels.

Threat Model. The goal of STAMP is to protect the confidentiality and the integrity of ML model inference in the presence of a malicious adversary. We capture such confidentiality and integrity through simulation-based security [7, 8, 23]:
Figure 1: STAMP system and threat model. The black local machines owned by three parties, green local buses, and black inter-party communication channels are untrusted. The blue LTHs are trusted and contain secret keys shared among LTHs.

Figure 2: The STAMP execution flow on one of the parties. Inter-party communication (wave symbol) and the local communication with the LTH (green arrows) happen during initialization and execution, with (1) or without (2) the optimization in §4.2. An adversary has complete control over the data and operations in the red zone.

Definition 1 (Simulation-based security: privacy and verifiability). A protocol $\pi_F$ is said to securely realize the ideal functionality $\mathcal{F}$ if for any probabilistic polynomial time (PPT) real-world adversary $\mathcal{A}$, there exists an ideal-world adversary $\mathcal{S}$ such that for any PPT environment $\mathcal{Z}$, there exists a negligible function $\text{negl}(\cdot)$ such that

$$\left| \Pr[\text{Real}_{\pi_F,\mathcal{A},\mathcal{Z}(\lambda)} = 1] - \Pr[\text{Ideal}_{\mathcal{F},\mathcal{S},\mathcal{Z}(\lambda)} = 1] \right| \leq \text{negl}(\lambda)$$

We consider honest majority, meaning that at most one party (except its LTH) can be malicious. The other two parties can be semi-honest, in which they may try to learn secrets (e.g., inputs or weights provided by other parties) while still following the protocol faithfully. The malicious adversary can deviate arbitrarily from the honest protocol, and its goal can be breaking the integrity of the evaluation by providing incorrect results without being noticed, or breaking the confidentiality of the data by learning the secrets. We assume that there is no collusion between any of the parties.

Figure 1 provides an overview of the STAMP system. We assume that a party or its server is untrusted except for its LTH. In other words, an adversary may control any part of the server, including a virtual machine monitor, an operating system, drivers, storage, and others except for LTH. We assume that the confidentiality and integrity of LTH are protected and an adversary cannot obtain data on an LTH or alter its execution. To ensure that only valid secure hardware can participate in the protocol, LTH contains a unique private key and is authenticated through a Certificate Authority (CA) during the initialization step. As shown in Figure 1, the three LTHs act as three trusted third parties with established correlations (secret keys). Figure 2 shows that the data flow between an untrusted server (red) and an LTH (blue) during the STAMP execution. Data should be encrypted before being sent to the red zone, and any data from or operations conducted in the red zone should be verified assuming the presence of a malicious adversary.

The security model and its detailed analysis are presented in Appendix C. Although we assume that LTH is secure, we discuss how LTH provides security benefits over TEEs in §3.3 under a hybrid MPC-trusted hardware threat model.

3 BACKGROUND

In this section, we describe our notation, and then provide some basics of MPC and trusted hardware.

3.1 Notation

We define $L$ as the finite field size, and $\mathbb{Z}_L$ to be the finite field we generally consider in this work. $\text{fp}$ is the fixed-point precision. We use the bold font $\textbf{a}$ or $\textbf{A}$ to represent a vector or a matrix. We use $a_i$, $(a)_i$ or $A_{i,j}$ to represent the $i^{th}$ element of the vector $\textbf{a}$ or the element of the matrix $\textbf{A}$ in the $i^{th}$ row and in the $j^{th}$ column. This is different from the bold $\bar{A}_i$, which still represents a matrix. Throughout the paper, if not specifically mentioned, all operations are carried out within the finite field $\mathbb{Z}_L$. When needed, we use $(a+b)_i$ to represent the modulo $L$ operation for the output of the integer operations in brackets. We add a bar to a variable or operation, as $\bar{a}$ or $\text{exp}(\bar{a})$, to represent that a number or an output is a real number. The right-shift operation is indicated as $\gg$ (e.g., $a \gg 2^{52}$). We will often use two signed integers $m, q$ to represent a positive real number $\bar{a}$ as $\bar{a} = 2^q \cdot (m \gg 52)$ where $m$ represents the mantissa part of 52 bits with $m \gg 52 \in [0, 1)$, and $q$ is the exponent part. This is actually the format in which floating point numbers are represented following the IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754-1985) [38], but without sign on the mantissa part. We use $\lfloor \bar{a} \rfloor$ to round a real number $\bar{a}$ down to an integer.
3.2 Multiparty Computation

**Notation.** The sharing scheme used in this work is the 2-out-of-3 replicated secret sharing scheme (RSS) modulo L. Let \( P_1, P_2, P_3 \) be the three parties participating in the evaluation. For convenience, we use \( P_{i-1}, P_{i+1} \) to refer to the previous and next party of one party (e.g., the previous and the next party of \( P_1 \) are \( P_2 \) and \( P_3 \)). The RSS of an integer secret \( x \in \mathbb{Z}_L \) is denoted as \( [x]_L = ([x]_L^1, [x]_L^2, [x]_L^3) \), where \( L \) is the size of the finite field to which the shares belong and \( x = [x]_L^1 + [x]_L^2 + [x]_L^3 \). When a secret \( x \) is shared as \([x]_L^i\), party \( P_i \) holds \(([x]_L^i, [x]_L^{i+1})\) for \( i = 1, 2, 3 \). To generate the integer representation \( x \) based on the real value \( x \), we use two’s complement fixed-point encoding with \( f_p \) bits of precision. For a positive \( x \) we have \( x = [x \cdot 2^{f_p}] \), and for a negative \( x \), \( x = [x \cdot 2^{f_p}] + L \), assuming that \( x \) is within the bound \([-L/2^{f_p}, L/2^{f_p}]\).

In our experiments, we mainly use the cases of \( L = 32 \), with \( f_p = 13 \) and \( L = 2^L \) (which supports inputs from \(-2^{26214} \) to \( 2^{26214} - 2^{-13} \)), to match the bit-width used in the baseline MPC schemes. The security of an \( L = 32 \) setting naturally comes from the random masking creating the shares. Multiple existing MPC schemes [11, 69, 71, 89] use the 32-bit secret sharing setting and already prove its security. Our protocol can also use a larger field such as a 64-bit setting if a wider range of values need to be supported.

Here, we explain how multiplications are performed under 2-out-of-3 RSS. These operations follow the protocols defined in [15, 58, 88, 89]. We describe the rest of the baseline MPC operations including share creation, reconstruction, and aggregation in Appendix A.

**Multiplications.** \( [x \cdot y]_L \leftarrow \Pi_{\text{Mul}}([x]_L, [y]_L) \). To get \([x \cdot y]_L = [xy]_L \), \( P_i \) first computes \( z_i = [x]_L^i [y]_L + [x]_L^{i+1} [y]_L^i + [x]_L^i [y]_L^{i+1} + [x]_L^{i+1} [y]_L^i \), and then \( [x \cdot y]_L = [z_i]_L \). This product is a valid 3-out-of-3 secret sharing of \( xy \) since \( z_1 + z_2 + z_3 = xy \). A reshare is needed to maintain the consistency of the 2-out-of-3 sharing scheme. To avoid any possible leakage of information, \( P_i \) uses the 3-out-of-3 randomness \( \{a_{i1}\} \) to mask \( z_i \) as \( z_i = \hat{z}_i + a_{i1} \), then share it with \( P_{i-1} \). Therefore, the parties obtain the necessary shares and build \([z]_L = [xy]_L = (z_1,z_2,z_3)\).

**Matrix Multiplications.** \( ([AB])_L \leftarrow \Pi_{\text{MatMul}}([A]_L, [B]_L) \). To perform matrix multiplication \( \Pi_{\text{MatMul}}([A]_L, [B]_L) \), simply applying \( \Pi_{\text{MatMul}} \) for each multiplication leads to \( O(ac) \) transmission overhead. The parties can instead perform part of the addition of shares locally (i.e., \( [C]_L = [A]_L [B]_L + [A]_L [B]_L + [A]_L [B]_L \), then share \([C]_L \) at once. This strategy yields only \( O(ac) \) transmission overhead. As stated in [88], convolutions can be expanded into overall larger matrix multiplications.

The above multiplication protocols work well for integer representations, but will cause errors with fixed-point representations. A truncation protocol (right-shift the results by \( f_p \) bits) must follow after a multiplication to correct the fixed-point precision in 3-party MPC. We refer the readers to ABYS [58] for more details on the 3-party truncation protocol, to prior work [22, 58] for the malicious variant of \( \Pi_{\text{MatMul}} \), and to Appendix A for other basic operations.

3.3 Trusted/Secure Hardware

Dedicated security hardware has a long history of being successfully used in many high-security use cases, starting as (co-)processors specializing in crypto operations. For example, smart cards [67] are widely used in financial transactions. Similarly, hardware security modules (HSMs) such as IBM 4758 [18] have also been used to protect critical secret keys. Discrete security chips such as TPM [64], Google Titan [33], and Apple T1 provide hardware root-of-trust on many platforms. Modern System-on-Chip (SoC) designs also typically include a dedicated security processor with crypto engines for secure booting and other high-security operations: Synopsis tRoot hardware security module [80], Rambus RT-630 programmable root-of-trust (RoT) [35], Apple secure enclave [32], Qualcomm secure processing unit [34], etc. Even though their performance is limited and their implementations may still have security vulnerabilities [6, 27, 57], the small dedicated security processors are considered to be far more secure compared to high-performance processors. The dedicated security processors are also relatively easy to deploy as a separate chip or as an IP block.

For high-performance processors, the idea of trusted hardware developed into a trusted execution environment (TEE), which adds hardware-based security protection on a shared general-purpose processor running a full software stack [30, 72, 85]. A TEE aims to protect the integrity and confidentiality of the code and data inside, even when low-level software and/or the environment cannot be trusted. TEEs on modern processors can typically provide much higher performance compared to the dedicated security hardware, but also introduce new security challenges due to the large TCB and complex optimizations in high-performance processors [4, 19, 24, 26, 47, 48, 73, 90, 94]. The high-performance TEEs also require new hardware protection for each computing engine and significant changes to a complex software stack, making their deployment for new hardware challenging.

In this work, we consider lightweight trusted hardware (LTH) with performance and complexity similar to a traditional security chip or on-chip security subsystems in modern SoCs: a dedicated low-performance security processor that supports remote attestation to validate its identity and shared key exchanges (§4.1), has hardware crypto engines, and includes a programmable processor that can run code. We consider two types of LTH designs as shown in Figure 3: 1) a discrete security chip similar to a TPM (LTH-chip) running at a low clock frequency (tens of MHz), and connected to a CPU through a low-bandwidth interface; and 2) a security subsystem on an SoC (LTH-SoC) running at a much higher SoC clock frequency (1-3GHz), and connected to other processing engines (CPUs, GPUs, NPs, etc.) on the same SoC through high-bandwidth on-chip networks. Our study suggests that even the LTH-chip can significantly improve the performance of the MPC-based PPML.

**Notation.** Each party \( P_i \) is equipped with a LTH \( H_i \), which has a built-in PRF unit \( F \) (e.g., an AES engine) for pseudo-random number generation. We assume that even malicious participants cannot break the integrity and confidentiality guarantees that LTH provides. The protocols executed in \( H_i \) will be introduced in §4.
3.3.1 LTH Benefits. While it is difficult to quantify the security, we believe that LTH provides strong security and deployment benefits over high-performance TEEs. For example, the previous survey [19] provides an overview of the vulnerabilities in Intel SGX (high-performance TEE) and countermeasures. Most vulnerability categories (address translation, CPU cache, DRAM, branch prediction, rowhammer) in the survey do not apply to LTH due to the following reasons. Here, we provide a more detailed discussion of the security of LTH and list some attacks that LTH is more robust to.

Physical isolation: LTH is dedicated to a small set of security tasks, and physically separate from main processing cores with potentially malicious software. LTH tightly controls its software using secure booting and typically does not allow user software. Because hardware is not shared with potential attack software, there is much less concern for timing-channel attacks - a major challenge in today’s TEEs.

Smaller TCB/attack surface, lower complexity: LTH uses a simple (in-order) processor with limited interfaces/commands for a small set of security tasks. Both hardware and software are much smaller and simpler compared to the main processors. For example, the dedicated security processors usually occupy less than 1mm² of the silicon area. On the other hand, a high-performance CPU takes hundreds of mm² and contains millions of lines of code (LoCs) [46], and is shared with many software components. Because there is no speculation or out-of-order execution, transient-execution attacks such as Meltdown/Spectre that can run commands or read memory without permission are not a concern for LTH. LTH does not have external memory (DRAM), and is not exposed to attacks on external memory such as DRAM probing and rowhammer attacks.

Side-channel protection: LTH such as smartcards, TPM, and others are usually equipped with dedicated crypto engines and countermeasures (e.g., tamper-resistant circuits [77] for TPM, randomized block design [56] for smart cards, etc.) against physical side channels such as power side channels, and without off-chip memory. In that sense, LTH is more robust against physical attacks.

3.3.2 LTH Limitations. The main limitation of LTH comes from its performance. LTH is typically not designed for high-performance computation. Both computation and communication on LTH are much slower compared to high-performance TEEs. As a result, the use of LTH comes with the additional challenges to support sufficient end-to-end performance. Traditionally, LTH is only used for small security-critical operations such as key management and infrequent signing. In order to leverage LTH for larger applications such as ML inference, we need to divide the workload and only offload small parts to LTH in a way that LTH does not become the performance bottleneck. In fact, STAMP had to be carefully designed to leverage low-performance LTH and our experimental results show that the overall performance still depends on the performance of LTH (LTH-chip vs. LTH-SoC). On the other hand, TEEs can closely match the performance of the underlying high-performance processors and can often be used to run the entire task such as ML inference inside, with minimal changes to the workload. If a high-performance TEE can be fully trusted, a TEE can replace the LTH in our scheme to provide higher performance or be used to run the entire ML inference without MPC.

While we believe that LTH provides stronger security compared to high-performance TEEs, we note that LTH can still have security vulnerabilities, similar to how secure cryptographic algorithms can be broken due to implementation-level vulnerabilities. For example, timing side channels and power interrupts may make TPM private key recovery possible [27, 57]. Smart cards, although practically considered secure enough and widely developed, have faced challenges including reverse engineering [65], micro probing [75], optical fault induction attacks [76], and others.

Compared to complex high-performance TEEs, LTH has far fewer vulnerabilities, making countermeasures easier to apply in terms of cost and design complexity. In practice, the main security concerns for today’s TEE come from software-exploitable vulnerabilities. In that sense, LTH provides a major security benefit by removing most timing-channel or transient-execution vulnerabilities. While physical attacks are not considered a major threat in data-center environments, LTH can also provide strong physical security. LTH has no off-chip memory to protect, and often has anti-tamper/DPA countermeasures. In contrast, recent TEEs target weaker threat models against physical attacks. Intel removed the integrity tree for replay protection in Icealke/TDX. AMD SEV has no replay protection against physical attacks. NVIDIA CPU TEE (H100) does not even encrypt its high-bandwidth memory (HBM).

4 THE STAMP PROTOCOL

This section introduces the details of the STAMP protocols for both semi-honest and malicious settings. We refer the reader to Appendix C for detailed security analysis.

4.1 Initialization phase

The initialization phase $\Pi_{\text{init}}$ is a part of the offline phase (which needs no input data or model weights) of the protocol where the LTHs will have shared keys and initial values established in them if their identities are proven. Although $\Pi_{\text{init}}$ plays an important role in our scheme, it is not where our main contribution lies, since mature remote attestation protocols already exist [3]. A simplified description of $\Pi_{\text{init}}$ is shown in Protocol 1.

The communication out of $H_i$ has to go through $P_i$, which provides a corrupted party with a natural way to observe or even alter the communication among the LTHs. For semi-honest adversaries, the Diffie–Hellman key exchange protocol already prevents them from obtaining the key with bounded computational resources. If the corrupted party behaves maliciously, $\Pi_{\text{init}}$ does not have to take extra steps to detect such actions. If a malicious $P_i$ modifies the remote attestation, a CA will not provide a certificate and $P_i$ cannot create a certificate on its own, causing an abort. If a malicious $P_i$ alters the transmission during key exchange, there will be no correct initialization established, and the protocol will abort later when data inconsistency is detected.

The shared keys and the PRF in the LTHs can support the pseudorandom number generation and are kept only known to the LTH, unlike the correlated randomness introduced in Appendix A. With the shared keys in §4.1 and a built-in PRF $F$, we can now construct $\Pi_{LTH.GenMask}$ and $\Pi_{LTH.GenMaskShare}$ in the LTH as Protocol 2 and Protocol 3. They are very similar with only a minor difference that $\Pi_{LTH.GenMaskShare}$ always generates shares of 0. Four counters $\{\text{ctr}_1, \text{ctr}_2, \text{ctr}_3, \text{ctr}_4\}$ are used in each $H_i$ to maintain consistency among $H_i$s in a semi-honest setting, and additional four
 Protocol 1 \(\Pi_{\text{GenMask}}\) Initialization  
**Input.** Security parameter \(\lambda\).  
**Result.** Output (Success, \(L\)) if the remote attestation succeeds and aborts if failed. After the initialization, LTHs (\(H_i\)) obtain shared keys and initial parameters.  
1. Parties first agree to a \(L\) for the finite field \(\mathbb{Z}_L\) size \(l = \log L\) bits, prime \(p\), and their order to define the previous and next party.  
2. \(P_i\) performs remote attestation on each \(H_i\) to obtain a certificate from the CA and publicly share them to validate \(H_i\). Abort if validation fails.  
3. \(H_i\) performs Diffie–Hellman key exchange with signature through the secure channel between \(P_i\) to obtain \(O(\lambda)\)-bit PRF keys \(k_{i,t+1}, k_{i-1,t}\), then use \(F_{k_{i,t+1}}\) to mask one key \(k'_{i-1,t} = k_{i-1,t} + F_{k_{i,t+1}}(0) \mod p\) and send \(k'_{i-1,t}\) to \(H_{i+1}\) through \(P_i\). \(H_i\) would receive \(k'_{i+1,t-1}\) from \(H_{i-1}\) and can recover \(k_{i+1,t-1} = k'_{i+1,t-1} - F_{k_{i,t+1}}(0)\).

 Protocol 2 \(m\rightarrow \Pi_{\text{LTH.GenMask}}(n, L; j, i, ctr_j, k_{j,j\ldots})\)  
**Input.** The number of masks to be generated \(n\), the index \(j\) for which counter, and which key to choose. The size of the finite field \(L\) and the counter \(ctr_j\) and the key \(k_{j,j\ldots}\) are stored in the LTH.  
**Output.** pseudo-random masks \(m \in \mathbb{Z}_L\) and updated counter \(ctr_j\), \(m = (F_{k_{j,j\ldots}}(ctr_j), F_{k_{j,j\ldots}}(ctr_j + 1), ..., F_{k_{j,j\ldots}}(ctr_j + n - 1))\).  
Update \(ctr_j \leftarrow ctr_j + n\).

 Protocol 3 \(\{m_j\}_{j\ldots}, \{m_j\}_{j\ldots}\}) \rightarrow \Pi_{\text{LTH.GenMaskShare}}(n, L; i, ctr_i, k_{i,i+1}, k_{i-1,i-1}, k_{i-1,i})\)  
**Input.** The number of masks to be generated \(n\). The size of the finite field \(L\) and the counter and keys are stored in the LTH.  
**Output.** pseudorandom masks \(\{m_j\}_{j\ldots}, \{m_j\}_{j\ldots}\) \(\in \mathbb{Z}_L\) \(\begin{align*} [m_{j\ldots}]_{j\ldots} &= F_{k_{i,i+1}}(ctr_j + j) - F_{k_{i-1,i-1}}(ctr_j + j) \quad \text{for } j = 0, ..., n - 1 \[m_{j\ldots}]_{j\ldots} &= F_{k_{i,i+1}}(ctr_j + j) - F_{k_{i-1,i-1}}(ctr_j + j) \quad \text{for } j = 0, ..., n - 1 \end{align*}\)  
Update \(ctr_i \leftarrow ctr_i + n\).

\(\{ctr_i, ctr_j, ctr_j, ctr_i\}\) are needed in a malicious setting for reduplicate execution for the detection of inconsistency. Notice that Protocol 2 and Protocol 3 give the outputs to \(H_i\), not \(P_i\), and \(H_i\) may be set to give partial outputs in some protocols.

A proper remote attestation protocol is commonly supported on secure hardware, such as a TPM [64], and validates the LTH’s identity and its state. This process can involve the acquisition of the certificate of a LTH from a trusted CA/Verifiers, which is usually the manufacturer of it. \(H_i\) after being verified, can perform pairwise Diffie-Hellman key exchanges with a signature to obtain the shared key \(k_{i-1,t}\) with \(H_{i-1}\) and \(k_{i,t+1}\) with \(H_{i+1}\). And then also \(k_{j,j\ldots}\) through sharing masks. The shared keys can support the pseudorandom number generation with PRF and are kept only known to the LTH.

4.2 Optimized ReLU with Matrix Multiplication

Non-linear layers used in a machine learning model are computationally light under plaintext. ReLU, for example, takes only one comparison and multiplexing. However, its complexity gets amplified significantly under the RSA scheme with more local computation steps and significant communication overhead.

Using each party’s LTH and the common randomness established in §4.1, we can significantly reduce the overhead by “offloading” the non-linear operations to the LTH. For example, ReLU can be performed by: invoking \(\Pi_{\text{LTH.GenMask}}\) to get the pseudo-random masks, transmitting the masked shares, recovering the plaintext to compute inside LTHs, and then generate and distribute the pseudo-random shares of the results. We show the details of this protocol \((\Pi_{\text{ReLU}})\) in Appendix B.

In Stamp, we further optimize ReLU by combining it with truncation. For typical ML models [29, 44, 74, 92], ReLU is applied after matrix multiplications in convolution (Conv) or fully-connected (FC) layers. As introduced in §3, in a fixed-point setting, truncation is required after each multiplication to keep the consistency of the precision. If we apply \(\Pi_{\text{ReLU}}\) directly after the completion of multiplications, the communication overhead will be the multiplication / truncation overhead and the \(\Pi_{\text{ReLU}}\) overhead summed, which is not optimal. Since the truncation itself is also a simple non-linear function in plaintext (which is just right-shift), we can exploit this common structure in deep learning models and merge the truncation with the following non-linear operations to be simply computed together in plaintext inside the trusted LTH.

The protocol \(\Pi_{\text{MatMulReLU}}\), detailed in Protocol 4, demonstrates how ReLU can be combined with truncation after matrix multiplication. The steps for a semi-honest setting are colored black, with additional steps for a malicious adversary marked blue. We use this notation in other protocols as well. \(\Pi_{\text{MatMulReLU}}\) reduces the total communication rounds of matrix multiplication and ReLU combined to 2 from at least 3, by merging the transmission needed for truncation and sharing shares masked by pseudorandom masks generated by LTHs in step 2) and 3). The malicious version generally adds replicate parallel operations and requires replicate sharing of the same values to validate the integrity. Parties compare the copies of intermediate results and final outputs from different sources to achieve malicious security with abort. We also use \(\Pi_{\text{mat-arith-mult}}\) of [58] to ensure correct 2-out-of-3 shares after local multiplication.

One may notice that the workload is not balanced among the three parties if we fix \(i\). In the protocol, the party index \(i\) can be any of \(\{1, 2, 3\}\), which means that the three parties can start the protocol simultaneously with a disjoint dataset. Therefore, when provided with a batch \(B\) of inputs for evaluation, each party can work on the \(B/3\) data and start the corresponding protocol simultaneously, balancing resource usage and reducing overall latency.

4.3 Extensions to Other Operations

\(\Pi_{\text{ReLU}}\) can be extended to \(\Pi_{\text{MaxPooling}}, \Pi_{\text{BatchNorm}}, \Pi_{\text{LayerNorm}}\) that are common non-linear operations needed in deep learning networks. \(\Pi_{\text{MaxPooling}}\) needs comparisons and multiplexing. \(\Pi_{\text{BatchNorm}}\) need about two and \(\Pi_{\text{LayerNorm}}\) needs about three multiplications for each element on average. Their low complexity allows them to be offloaded to the LTH in a similar way as \(\Pi_{\text{ReLU}}\) by changing the exact plaintext function executed inside. \(\Pi_{\text{MatMulReLU}}\) can be extended to other operations in a similar way by changing step 5) of it. To optimize neural networks in our experiments, we mainly also use \(\Pi_{\text{MatMulMaxPoolReLU}}, \Pi_{\text{MatMulBatchNormReLU}}\) which merge the truncation with different joint non-linear layers.

4.4 Softmax

Exponentiation is crucial in modern deep learning models, such as logistic and softmax functions. In this work, we focus on softmax, which is extensively used in modern models such as Transformers [92]. Classical MPC softmax implementations [42, 66] leads to a large overhead due to two main reasons: the complex protocol for
approximating exponentiation and the max function applied before softmax. A recent study [91] shows that softmax is the main source of overhead when running a Transformer network with an MPC protocol and also introduces a numerical stability problem.

The Softmax on a vector $x$ is defined as follows:

$$\text{Softmax}(x) := \exp(x) / \sum_{i=1}^{n} \exp(x_i)$$  \hspace{1cm} (1)$$

In a regular ML setting, exponentiation can easily lead to overflow, a problem exacerbated in fixed-point representations used by MPC protocols. The traditional solution is to subtract the maximum input value from every element before applying the softmax function, ensuring the maximum input value is 0 and preventing overflow. However, this additional max operation introduces significant MPC overhead as shown in a recent study [91].

A naïve extension of the previous protocol for exp would be to move exp to the LTH, similar to the other non-linear operations in $\Pi_{\text{ReLU}}$. This would not work due to the low computational power of the LTH and the large amount of computation required for exp compared to other operations. Under our assumption on the trusted hardware (details in §5), tests show that 1 million 32-bit multiplications take less than a second, while double-precision exponentiation takes over a minute. Unlike simple non-linear operations, exp needs to be done at floating point arithmetic for high precision, involving tens of multiplications per exp and creating significant overhead and a new bottleneck for our scheme on a small LTH.

Our solution is to split and “offload” the computation to the untrusted local machine. The most complex part of the exp operation is performed by the powerful but untrusted CPU/GPU, and then the results are assembled within the LTH. The protocol is based on the property $\exp(a + b + c) = \exp(a) \exp(b) \exp(c)$, allowing untrusted machines to compute exp on individual shares so that only simple multiplications are needed on LTH. However, the conversion between fixed-point representations and real-number arithmetic is non-trivial under MPC. In Protocol 5, we expand the exponent part (see §3.1) to contain all possible results of $\exp(x_i^{\Pi_{\text{ReLU}}})$, specifically for $L = 2^{32}$. Overflow would not occur after this adjustment, even without invoking the max function before Softmax.

### 4.5 Integrating Stamp into Real Systems

A full implementation of Stamp requires four main functions to be performed by LTH: attestation during the initialization phase, pseudorandom number generation for masking, communication between LTH and a host CPU, and the rest of the protocol mainly for in-LTH computation. From the functionality point of view, all these operations can be implemented in software on any security processor if it is equipped with a unique device secret key in hardware that can be used for attestation. Fortunately, most security hardware today supports attestation and meets this requirement. From the performance point of view, our prototype and experimental evaluation assume that LTH has hardware AES engines for pseudorandom number generation to match the LTH-CPU communication bandwidth, while assuming that all other LTH operations are performed in software. More specifically, the performance evaluation is based on software run-time on a tiny microcontroller (Arduino Due) with an ARM Cortex-M3 that is also used in TPM, which represents today’s low-end security processor.
Consider Apple’s Secure Enclave [31] as another example, which already includes a dedicated nonvolatile storage and a unique ID root (UID) cryptographic key to protect device-specific secrets for remote attestation. It also includes a true random number generator (TRNG), an AES engine that may be used for pseudo-random number generation, a general-purpose CPU (Secure Enclave Processor), and a communication channel with the main CPU. While Apple does not disclose the throughput of the AES engine or the performance of the Secure Enclave Processor, they are likely sufficient for Stamp, as the AES engine is designed to encrypt NAND flash storage, and the processor runs at a high SoC clock frequency. Other SoC security subsystems, such as the Synopsys Troot hardware security module [80], the Rambus RT-630 programmable root-of-trust (RoT) [35], and the Qualcomm secure processing unit [34] also support comparable hardware features, including a device-specific secret key, a hardware AES engine, and a general-purpose processor. Thus, we believe that Stamp can be realized on today’s lightweight security hardware with minimal changes.

Stamp is designed to be used even with a tiny low-performance security processor, but it can also run on a high-performance TEE such as Intel SGX and AMD SEV implemented in software inside. Although performance should be better than the low-end LTH implementation, we believe that LTH can provide stronger security protection compared to the traditional TEEs (see §3.3).

5 Evaluation

5.1 Experimental Setup

Implementation and baselines. We implemented Stamp in C++, building on Falcon [89]. We introduced new protocols, GPU support for linear layers, and we switched to ZeroMQ for networking. Falcon is the main framework we compare to, but the open-source project was not implemented to support GPUs and does not address a key protocol for Transformers: Softmax. Falcon+ introduces three main improvements: (1) GPU support for linear layers, (2) a new MPC protocol for $\Pi_{\text{softmax}}$ using the exponentiation protocol from [42] combined with Falcon’s $\Pi_{\text{Div}}$ and $\Pi_{\text{Max}}$, and (3) ZeroMQ for
networking to ensure better performance and a fair comparison with StAMP. These changes do not alter the threat model.

We also compare our scheme with AriaNN [71] and CryptGPU [81] as additional pure MPC baselines. Appendix D compares the theoretical complexities for Falcon+, AriaNN, and StAMP. AriaNN and CryptGPU show both advantages and disadvantages relative to Falcon in different settings prior to our optimizations. We discover in our experiment that AriaNN and CryptGPU consume a significant amount of memory: a server with 64GB DRAM can only process ResNet18 inference with a batch size of 8 using AriaNN, whereas 32GB suffices for a batch size of 128 with StAMP. Similarly, CryptGPU supports batch sizes of up to 8 for ResNet and 32 for VGG16. We adjusted batch sizes accordingly for these experiments, noting results with smaller batch sizes explicitly. Additionally, we compare StAMP with two high-performance TEE-based schemes: a full SGX solution, running entire inferences inside an SGX enclave; and Goten [61], which accelerates TEE-based private inference by offloading linear operations to untrusted GPUs using a secret multiplication protocol with Beaver triples. Both CryptGPU and Goten support only a semi-honest GPU setting.

**Hardware and network.** We conducted our experiments on Cloudlab c240g5 machines with Ubuntu 20.04 LTS, equipped with an Intel Xeon Silver 4114 10-core CPU (2.20 GHz) and an NVIDIA 12GB P100 GPU. The network setup mirrors previous studies [59, 71, 88, 89], with a LAN bandwidth of 625 MBps and a ping time of 0.2 ms, and a WAN bandwidth of 40 MBps and a ping time of 70 ms. Both semi-honest and malicious settings were tested. For the LTH-chip, we used an Arduino Due with an Atmel SAM3X8E ARM Cortex-M3 CPU (84MHz, 512 KB of Flash, and up to 96 KB of SRAM), which is used for a commercial implementation of TPM [78], to evaluate the LTH runtime. The LTH assumes a low-pin-count (LPC) bus, resulting in a 15MBps bandwidth limit. A maximum of 3MBps of random number generation can be achieved in a TPM [79], which is enough for its original use case, but not for our scheme. We assume an additional low-cost hardware AES engine [17] achieving a throughput of 14 GBps, making the LTH’s pseudo-random number generation time negligible compared to data transmission time. Other details of the hardware can be seen in §5.4. For the LTH-SoC, we assume the same Cortex-M3 processor running at 1GHz and the 128-bit on-chip network (16GBps). The performance is estimated by scaling the execution time of the discrete LTH. We additionally provide a memory usage analysis of LTH in Appendix E, showing that StAMP can handle most models with our current LTH setting, and can be modified to handle even larger models with increased LTH local communication cost.

**Neural networks and dataset.** We use 8 neural networks: a small 3-layer fully-connected network with ReLU activations (Network-A, as in SecureML [59]), a small convolutional network with ReLU activation (Network-B, as in [69]), a small convolutional network with ReLU activation and maxpooling (Network-C, as in [49]), AlexNet [44], VGG16 [74], ResNet18 [29], a small Transformer [86] and a small Word2Vec [54]. We use a small Transformer and reduce the size of the last layer in Word2Vec to manage the computational expense of Softmax in pure MPC, especially in a WAN setting. The datasets used are MNIST [16] for the first four networks, CIFAR-10 [43] for AlexNet and VGG16, ImageNet [70] for ResNet18, and WikiText-2 [53] for the Transformer and Word2Vec.

**Parameter choice.** As mentioned in §4.4, we choose $L = 2^{32}$ and $fp = 13$ in our implementation. We pick the group size to be 2048 bits in Diffie–Hellman key exchange and use AES-128 for the pseudo-random number generation.

### 5.2 Performance

Table 1 and Table 2 show the end-to-end latency (in seconds) of the inference on inputs of batch size 128 in semi-honest and malicious settings, respectively. Table 3 and Table 4 report the amount of data transmitted compared to baselines with traffic analysis tools or the data reported in the papers. '*' in the cells indicates that the implementation is missing or the network is too large for CPU evaluation. The brackets in the tables indicate an altered batch size for the cases when a large batch size did not fit into our machine.

Only Falcon implemented its work in a malicious setting. In the tables, we compare StAMP with Falcon [89], AriaNN [71], and CryptGPU [81], three of the state-of-the-art MPC frameworks implementing different optimizations for non-linear layer inference. AriaNN does not implement the execution of different parties on separate machines, but instead uses the local simulation of the network for performance evaluation. AriaNN does not support Transformer and Word2Vec because it does not support Softmax, and we could not use the open-sourced CryptGPU to run Transformer, because the provided version raised error when generating the secret sharing model due to compatibility issues. Depending on the structure of the machine learning network, StAMP with LTH-chip is 4× to 63× or 6× to 59× faster than the state-of-the-art MPC results with semi-honest or malicious settings in WAN/GPU environments. The advantage that we obtain under a LAN or CPU environment is smaller compared to a WAN/GPU environment. In the LAN, communication overhead is significantly reduced. With a CPU, the computation accounts for a larger portion of the execution time. These factors reduce the speedup, which is mainly accomplished by reducing the communication overhead of non-linear functions. StAMP with LTH-SoC achieves even higher speedup because LTH-SoC has higher performance compared to LTH-chip due to its higher clock frequency and the data movement between an LTH and a CPU/GPU is also faster on an SoC. The performance gap between LTH-SoC and LTH-chip is the largest in LAN/GPU environments where the local communication overhead is large. Also, for smaller networks, a GPU can be slower than a CPU. For a small amount of data and a small model, initialization and data movement may take more time than operating directly on a CPU.

We also compare StAMP with other schemes that rely on a high-performance TEE (Intel SGX): the full SGX solution and Goten [61]. The full SGX solution assumes that semi-honest parties can securely share their data with one party’s SGX for evaluation. The experiments are run on SGX V1 with 16GB enclave memory on Azure Standard DC4s v3. For smaller networks such as Network-B, the full SGX solution is slightly slower (0.46 seconds) compared to StAMP (0.12), mainly due to initialization overhead. However, for larger networks such as ResNet18, the full SGX solution only takes 8.15 seconds, while StAMP (LTH-SoC) takes 148 seconds. This result is expected as the performance overhead of MPC-based secure computation is known to be substantially higher compared to the
Table 1: Inference time (s) of the entire batch of size 128 in a semi-honest setting. AriaNN has a reduced batch size of 64 and 8 for VGG16 and ResNet18 due to memory consumption, which also applies to other tables. Brackets indicate an altered batch size.

| Framework | Network-A | Network-B | Network-C |
|-----------|-----------|-----------|-----------|
|           | LAN | WAN | WAN | WAN | LAN | WAN | WAN | WAN | LAN | WAN | WAN |
| Falcon+   | 0.082 | 0.112 | 1.478 | 1.393 | 0.203 | 0.2618 | 1.464 | 1.292 | 1.866 | 2.191 | 7.321 | 7.288 |
| AriaNN    | 0.256 | 0.512 | - | 5.504 | - | - | - | - | 3.072 | 5.248 | - | 17.02 |
| CryptGPU  | 0.449 | - | 13.49 | - | 0.333 | 9.592 | - | - | 0.752 | - | 19.45 |
| STAMP-chip | 0.073 | 0.077 | 0.251 | 0.294 | 0.117 | 0.114 | 0.278 | 0.293 | 0.680 | 1.298 | 1.024 | 1.494 |
| speed-up  | 1.11× | 1.43× | 5.87× | 4.72× | 1.73× | 2.29× | 5.25× | 4.39× | 1.10× | 1.68× | 7.14× | 4.87× |
| STAMP-SoC | 0.0432 | 0.0472 | 0.2211 | 0.2641 | 0.0643 | 0.0613 | 0.1994 | 0.2408 | 0.1990 | 0.8163 | 0.5424 | 1.012 |
| speed-up  | 1.91× | 2.37× | 6.69× | 5.28× | 3.17× | 4.28× | 7.34× | 5.37× | 3.78× | 3.10× | 13.5× | 7.20× |
| Goten     | 0.261 | - | 3.304 | - | 0.376 | - | 4.097 | - | 0.602 | - | 5.389 | - |
| Full SGX  | 0.462 | - | 0.461 | - | 0.461 | - | - | - | - | - | - | - |
|           | LAN | LAN | WAN | WAN | LAN | LAN | WAN | WAN | LAN | LAN | WAN | WAN |
| Framework | Falcon+ | AriaNN | CryptGPU | STAMP-chip | STAMP-SoC | speed-up | Goten | Full SGX |
|           | LeNet | AlexNet | Transformer | VGG16 | ResNet18 | Word2Vec |
|           | LAN | WAN | WAN | LAN | WAN | WAN | LAN | WAN | WAN | LAN | WAN | WAN | LAN | WAN | WAN | LAN | WAN | WAN |
| Falcon+   | 2.592 | 4.603 | 8.867 | 9.563 | 4.276 | 11.78 | 38.56 | 43.06 | 4.026 | 16.20 | 321.0 | 334.7 |
| AriaNN    | 4.480 | 7.040 | - | 18.30 | 9.984 | 19.20 | - | 43.52 | - | - | - | - |
| CryptGPU  | 1.337 | - | 19.12 | - | 1.918 | - | 35.90 | - | - | - | - | - |
| STAMP-chip | 0.969 | 3.075 | 1.315 | 3.255 | 1.564 | 9.263 | 2.463 | 9.449 | 0.5130 | 12.18 | 5.024 | 16.66 |
| speed-up  | 1.38× | 1.49× | 6.74× | 2.93× | 1.22× | 1.27× | 14.6× | 4.55× | 7.84× | 1.32× | 63.8× | 20.08× |
| STAMP-SoC | 0.2869 | 2.392 | 0.6328 | 2.573 | 0.305 | 8.029 | 1.229 | 2.125 | 0.3618 | 12.03 | 4.873 | 16.51 |
| speed-up  | 4.66× | 2.02× | 6.27× | 3.72× | 14.0× | 1.52× | 29.2× | 5.24× | 11.1× | 1.34× | 65.8× | 20.26× |
| Goten     | 0.944 | - | 6.233 | - | 0.778 | - | 9.127 | - | - | - | - | - |
| Full SGX  | 0.507 | - | 5.031 | - | 0.507 | - | - | - | - | - | - | - |

Figure 4: The breakdown of local machine execution time: linear layers, non-linear layers, and LTH-Chip bus communication & computation time. STAMP (left) and Falcon+ (right) on semi-honest inference over AlexNet under WAN/GPU.

1We believe STAMP outperforms in this case because Goten requires more communication for its secure multiplication. As the communication cost analysis for Goten is not available, we estimate the costs using analytical results in their paper (Table 1); and as an example, Goten’s communication for VGG16 is estimated to be 273MB compared to STAMP’s 188 MB.
that the CPU/GPU execution time is also reduced because major parts of most non-linear computations are moved to LTH.

**Discussion.** The acceleration achieved by STAMP varies significantly with the architectural design of the model. Convolutional neural networks (CNNs), such as AlexNet and VGG16, exhibit less pronounced speed improvements compared to language models like the Transformer and Word2Vec. This discrepancy aligns with the observation that language models employ computationally intensive non-linear operations more frequently, notably Softmax in our case. For instance, the Transformer model applies Softmax within each of its multiple attention heads. Word2Vec, despite its simplicity and consisting of only two linear layers, incurs a high computation cost for non-linear operations due to the inclusion of Softmax (we also keep the computational benefits for models that extensively leverage more complex non-linear operations). Consequently, STAMP tends to offer greater benefits for models that extensively leverage more complex non-linear operations. In contrast, models such as ResNet18 or VGG16, which are computationally heavy for linear operations but rely on simpler non-linear activation functions like ReLU, do not exhibit as significant speed-ups. This observation explains the higher speed-up numbers in the language models compared to the CNNs.
5.3 Accuracy

The precision of the inference using the plaintext computation and model weights during training can potentially help avoid this issue. In Table 3, almost 3GB of data is transmitted through the LTH-CPU bus for the ResNet18 reference, causing more than 200 seconds of communication time, which is about 60% of the total execution time of our scheme. LTH-SoC provides a much higher LTH-CPU bandwidth and significantly alleviates this bottleneck.

Figure 5 shows how the speedup over Falcon+ can change if we use a higher-bandwidth interconnect for the LTH. In this figure, we choose two computation-heavy networks, VGG16 and ResNet18, and a communication-heavy Transformer network (due to frequently used Softmax) as examples. We can observe a considerable boost in performance with a higher LTH bandwidth.

5.4 Hardware Overhead of LTH

The LTH in STAMP consists of two parts:

1. The core microcontroller with the same capability as the entire TPM. We refer to the design of ST33TPM12SPI [78] as a baseline with 0.40mm² area for the ARM SecurCore SC300. The microcontroller has a peak power consumption of 12mW.

2. An AES engine performing pseudo-random number generation. A previous study [17] reports a cost of 0.13mm² and 56mW in area and peak power consumption. The AES engine serves as the PRF \( F \) in \( \Pi_{LTH,GenMask} \) and \( \Pi_{LTH,GenMaskShare} \).

The combined overhead of 0.53mm² and 68mW is quite small, suggesting that LTH is cheaper and easier to deploy compared to adding a TEE to a high-performance processor. LTH may even be implemented as a simple extension of the existing TPM hardware or the on-chip SoC security subsystem. Furthermore, our protocol can be deployed with existing or future hardware platforms without integrating new TEE features directly into them. Note that the LTH overhead here does not represent the full power consumption of STAMP, which also runs an MPC protocol on an untrusted CPU/GPU.

5.5 Trusted Computing Base (TCB)

As the security of a system is difficult to quantify, the TCB size is often used as a proxy when comparing system designs. Our estimates suggest that LTH has a much smaller TCB compared to a high-performance TEE. For the hardware TCB, open-source microcontrollers whose complexity is comparable to LTH that we use have <20k Lines-of-Code (LoC) (OpenRISC: 16k LoC + AES: 1k LoC). While the LoC for commercial TEE hardware is not publicly available, the area of high-performance processors (Intel Skylake: \( \sim 322mm^2 \) ~ 698mm², Intel Sapphire Rapids: \( \sim 400mm^2 \) ) is much larger than the size of LTH (0.53mm²).

For the software TCB, our LTH software implementation has \( \sim 13k \) LoC. On the other hand, the software TCB for the Intel SGX experiment includes Gramine (\( \sim 50k \) LoC) and PyTorch (\( \sim 166k \) LoC) inside a TEE. For, virtual machine (VM) based TEEs such as Intel TDX and AMD SEV the software TCB can be much larger as the entire operating system (OS), drivers, and ML software stack (PyTorch) all need to run inside a TEE (millions of LoC for Linux).

6 RELATED WORK

Encrypted computation (MPC/HE) for machine learning. Cryptographic techniques such as garbled circuits [10, 68], secret sharing [58, 71, 88], homomorphic encryption [59, 98] have been applied for privacy-preserving inference. Gazelle and Delphi [37, 55] combine homomorphic encryption and garbled circuits for their advantages in linear and non-linear operations, respectively. Falcon [89] implements a 3-party malicious secure protocol, combining techniques from SecureNN [88] and ABY3 [58]. Blaze [63] achieves not only 3-party malicious security but also fairness in an honest majority setting. AriaNN [71] leverages function secret sharing to reduce communication rounds for specific functions, but at the cost of increasing the total amount of communication data in some cases. CrypTen [42] provides a general software framework that makes secure MPC primitives more easily used by integrating them into
a popular ML framework, PyTorch. GForce [60] proposed fusing layers in MPC, and more specifically combined dequantization and quantization layers into a truncation before and after ReLU and MaxPooling. Our protocol also applies layer fusing when applicable, but in the context of reducing overhead for non-linear operations in LTH. Our work leverages the recent developments in MPC for PPML, but shows that a simple security processor can significantly reduce the high overhead of today’s MPC-based PPML methods.

Combination of trusted hardware and crypto-based secure computation. Recent studies explored multiple approaches to improve MPC/HE for machine learning using trusted hardware. However, the previous work typically assumes a high-performance TEE such as Intel SGX and relies on the TEE to perform a significant amount of computation, which will be too slow on a small security processor. To the best of our knowledge, our work is the first to show that even a small low-performance security processor can significantly improve the performance of MPC if the protocol can be carefully designed for lightweight trusted hardware.

For performance improvements, the previous studied proposed using a TEE (Intel SGX) to accelerate bootstrapping [40, 51], perform faster functional encryption [21], and simplify certain protocols [12, 20, 40]. The previous work also investigated splitting the work between a TEE (Intel SGX) and MPC. For example, Gupta et al. [25] propose splitting secure computation between garbled circuits and Intel SGX. Zhou et al. [99] introduce a two-party TEE-aided MPC scheme that focuses on improving multiplication overhead by moving part of the linear operations to a TEE. HYBRTC [93] decides where the computation should be run based on whether or not the parties trust a TEE; a hybrid protocol moves the computation to the TEE or just performs an MPC protocol. While the high-level approach of offloading computation from MPC to trusted hardware is similar, the previous work offloaded heavy computation to a high-performance TEE while our work studies how to leverage a low-performance security processor.

Slalom [82] and Darknight [28] propose to run a private machine learning computation on an untrusted CPU by securely outsourcing linear operations from the CPU TEE (SGX) to the GPU using secret sharing, and later Goten [61] proposed an improved scheme compared with Slalom by introducing “dynamic quantization” for training. While the use of pseudorandom masks is similar to our protocol in Slalom, Slalom uses masking only for outsourcing linear operations, as the other two papers. As a result, non-linear operations cannot be offloaded, and the CPU TEE still needs to perform as many linear operations as a GPU in an offline phase. These approaches require a high-performance TEE, and the TEE performance limits the overall secure computation performance. Stamp, on the other hand, only requires small low-performance trusted hardware for non-linear operations by performing linear operations on untrusted CPUs/GPUs using MPC. Also, Slalom utilizes its pseudorandom masks with the pure additive linear homomorphism of functions. Our approach of computing Softmax has a similar idea but involves multiplicative homomorphism as shown in §4.4.

Trusted hardware can also be used to improve the security of an MPC protocol. For example, CryptFlow [45] runs MPC protocols on Intel SGX and leverages SGX’s integrity protection to achieve malicious security. Another work [5] uses Intel SGX to protect the data of parties in MPC even if they are remotely compromised.

7 CONCLUSION AND FUTURE WORK

This paper introduces a new PPML system which significantly reduces the overhead of MPC with the assistance of an LTH. Stamp can guarantee security against malicious parties in an honest-majority 3-party setting. Theoretical analysis and experimental results show that Stamp achieves significantly higher performance over state-of-the-art MPC protocols in various environments, even with an LTH whose performance is comparable to a TPM.

While Stamp provides significant speed-ups, we believe that this work represents the first step in exploring a broad design space of combining cryptographic protection and small high-security hardware to unlock a better security-efficiency trade-off, opening up interesting future directions. From the system’s point of view, while today’s SoC security subsystems such as Apple Secure Enclave is closed and its software is tightly controlled by the SoC vendors, it will be valuable if we can integrate Stamp into today’s SoCs to more fully understand the performance, security, and functionality of today’s LTH. It will also be interesting to broaden the applicability of Stamp to a wider array of modern ML models in practice, including Large Language Models (LLMs) and diffusion models. In particular, previous MPC studies found that the polynomial approximation of softmax can cause serious accuracy challenges in large Transformers. We leave a study on practical MPC-based private LLM inference with sufficient performance and accuracy for future work.

The experiments in this paper, while showing promising speed-ups, also show the challenges from the limited performance of LTH. In that sense, further optimizations of the protocol to reduce the overhead, extending the protocols to other types of MPC protocols, and the use of LTH is other types of operations beyond non-linear layers will be all promising future directions.

ACKNOWLEDGMENTS

This material is based upon work supported by the U.S. National Science Foundation under award No. CCF-2118709. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. Thang Hoang was supported by an unrestricted gift from Robert Bosch, 4-VA, and the Commonwealth Cyber Initiative (CCI), an investment in the advancement of cyber R&D, innovation, and workforce development. For more information about CCI, visit www.cyberinitiative.org.
A BASIC MPC PROTOCOLS

Correlated Randomness. A large number of random shares have to be obtained by the parties during the offline phase to reduce the communication cost during the online phase. The 3-out-of-3 randomness is defined as each $P_i$ holding a share of $0: \alpha = [a]_{L_i}^3 + [a]_{L_i}^3 + [a]_{L_i}^3$ where $\alpha = 0$ and $P_i$ holds $[a]_{L_i}^3$. They can be efficiently generated locally by a pseudo-random function (PRF).

The security of the PRF function indicates that the output of a PRF is computationally indistinguishable (indistinguishable by a computationally bounded adversary) from the output of a truly random function. Given $\xi$ as the key that $P_i$ and $P_{i+1}$ share through a key exchange protocol for each $i$ and $\hat{F}$ as the PRF that is public to all parties, each $P_i$ can generate shares $[\alpha]_{L_i}^3$ as $[\alpha]_{L_i}^3 = \hat{F}_k(\xi)\cdot (\text{ctr})$ with increments of ctr each time this process is invoked.

Input Phase. To construct $[x]^L_i$ from the generated 3-out-of-3 randomness and $x$ which is provided by $P_i$, $P_i$ compute $[x]_{L_i}^3 = x + [\alpha]_{L_i}^3$ and share it with $P_{i-1}$. $P_{i-1}$ will send $[\alpha]_{L_i}^3$ to $P_{i+1}$ and $P_{i+1}$ will send $[\alpha]_{L_i}^3$ to $P_i$. In an honest majority malicious setting, additionally $P_{i+1}$ should compute $[\alpha]_{L_i}^3 + [\alpha]_{L_i}^3$ and send to $P_i$, then $P_i$ confirm $[\alpha]_{L_i}^3 + [\alpha]_{L_i}^3 = -[\alpha]_{L_i}^3$, therefore $P_i$ behaves honestly; $P_i$ should compute $[\alpha]_{L_i}^3 + [\alpha]_{L_i}^3$ and send to $P_{i-1}$, then $P_{i-1}$ confirm $[\alpha]_{L_i}^3 + [\alpha]_{L_i}^3 = -[\alpha]_{L_i}^3$, therefore $P_{i-1}$ behaves honestly. Since $P_i$ is allowed to send an arbitrary input $x$, we do not need to check the share it sends.

Linear Operations. For RSS shared secrets, most linear operations can be performed locally. For shares $[x]_i^L$, $[y]_i^L$, and the public scalar $c$, we have the following:

- $[x]_i^L + c = ([x]_i^L + c, [x]_j^L, [x]_k^L])$
- $c \cdot [x]_i^L = (c \cdot [x]_i^L, c \cdot [x]_j^L, c \cdot [x]_k^L)$
- $[x]_i^L + [y]_i^L = ([x]_i^L + [y]_i^L, [y]_j^L, [y]_k^L)$

That can be done without any communication. However, multiplication between shared secrets cannot be done locally.

Reconstruction. $x \leftarrow \Pi_{\text{Recon}}([x]^L)$: To reconstruct the plaintext $x$ from the shares $[x]_i^L$, each party $P_i$ will send $[x]_i^L$ to $P_{i+1}$ in the semi-honest setting. After completion, all parties have all 3 shares to reconstruct $x$. In a malicious setting, $P_i$ will also send $[x]_{i+1}^L$ to $P_{i-1}$. Then, under the honest majority assumption, at most one of the two copies of the share they receive is altered. A party can compare the values received from the other two and abort if an inconsistency occurs.

B DETAILED STAMP PROTOCOLS FOR RELU

In this section, we introduce $\Pi_{\text{ReLU}}$, the protocol to offload ReLU operations under MPC to trusted hardware. The steps we take are as follows: First, $P_i$ invokes $\Pi_{\text{LTH.GenMask}}$ to get the pseudo-random masks, and sends $[x]_i^L$ to $P_{i+1}$ after adding them with the masks; Second, $P_{i+1}$ adds the received value with the two shares it holds, then sends the results to the LTH; Third, $H_{i+1}$ recovers the plaintext value by invoking $\Pi_{\text{LTH.GenMask}}$ using the same key and the counter with the same recorded number, and computes ReLU in plaintext, re-masks the result with $\Pi_{\text{LTH.GenMaskShare}}$, and sends them back to $P_{i+1}$. The other two parties will also generate their common share in the meantime; Fourth, $P_{i+1}$ re-shares the received values to complete the construction of the RSS of the outputs.

After the protocol, $\{\text{ctr}_{i+2}^i, \text{ctr}_{i+3}^i, \text{ctr}_{i+1}^i\}$ for $i = 1, 2, 3$ all increase by $n$, and $\{\text{ctr}_{i+2}^i, \text{ctr}_{i+3}^i, \text{ctr}_{i+1}^i\}$ for $i = 1, 2, 3$ all increase by $n$ too in a malicious setting. The synchronization of the counters together with shared keys guarantees the correlated randomness among the LTHs.

C SECURITY ANALYSIS

We claim the following three theorems hold:

**Theorem 1.** Under the assumption of secure PRF and LTH, $\Pi_{\text{ReLU}}$ (in Protocol 6) securely realizes the ideal functionality $\mathcal{F}_{\text{ReLU}}$ (in Figure 6) against any non-uniform PPT malicious adversary that can corrupt up to 1 out of 3 parties with static corruption.

**Theorem 2.** Under the assumption of secure PRF and LTH, $\Pi_{\text{MatMulReLU}}$ (in Protocol 4) securely realizes the ideal functionality $\mathcal{F}_{\text{MatMulReLU}}$ against any non-uniform PPT malicious adversary that can corrupt up to 1 out of 3 parties with static corruption.

**Theorem 3.** Under the assumption of secure PRF and LTH, $\Pi_{\text{Softmax}}$ (in Protocol 5) securely realizes the ideal functionality $\mathcal{F}_{\text{Softmax}}$ against any non-uniform PPT malicious adversary that can corrupt up to 1 out of 3 parties with static corruption.

In this section, we analyze the security of our proposed techniques. The extra or different steps done for a malicious setting than in a semi-honest setting are marked in blue.

\[ \mathcal{F}_{\text{ReLU}} \]

1. Upon receiving inputs $(x_{j,1}, x_{j,1})$, $(x_{j,2}, x_{j,2})$, $(x_{j,3}, x_{j,3})$ from $P_j$, $P_i$ respectively, check if $x_{j,j} = x_{j,j+1}$ for $j = 1, 2, 3$. If not, notify abort. Otherwise, compute $x = x_{j,1} + x_{j,2} + x_{j,3}$, $z = x > 0$. Then, send a signal (ReLU, $x_{j-1}, x_{j/i, n, L, i}$) to $S_{\text{Stamp}}$ that includes the inputs of $P_j$, the size of inputs of $P_{j+1}$ and $P_{j-1}$, and the index $i$ of the corrupted party.

2. Upon receiving (ReLUEnd, $i, z_{i-1}, z_i$) from $S_{\text{Stamp}}$, let $z_{i+1} = z - z_{i-1} - z_i$. Return $(z_i, z_{i+1})$ to the $P_f$ for $i = 1, 2, 3$.

\[ \mathcal{F}_{\text{ReLU}} \]

3. Upon receiving signal with inputs $x'$, masks $m$ and party index $i$, compute the following:

1. $x = x' - m$, $a = x > 0$.

2. Generate random $z_1, z_2, z_3$ such that $z_1 + z_2 + z_3 = x$. Returns $(z_i, z_{i+1})$ to $P_i$ (as mentioned in notation, for party index $i+1$ means the next party).

\[ \mathcal{F}_{\text{ReLU}} \]

4. Upon receiving with inputs $x'$, masks $m$ and party index $i$, compute the following:

1. $x = x' - m$, $a = x > 0$.

2. Generate random $z_1, z_2, z_3$ such that $z_1 + z_2 + z_3 = x$. Returns $(z_i, z_{i+1})$ to $P_i$ (as mentioned in notation, for party index $i+1$ means the next party).
**Protocol 6** \[ \text{ReLU}(x)^{L'} \leftarrow \Pi_{\text{ReLU}}(\|x\|^{L}) \] Do ReLU on shares of vector x

**Input.** Each \( P_i \) owns \( \|x_i\|^{L} \).

**Output.** Each \( P_i \) gets \( \|x_i\|^{L'} \).

1. \( P_i \) calls \( H_i \) to execute \( \Pi_{\text{LTH.GenMask}}(n, L, i-1) \) to obtain the masks \( m_{i-1} \in \mathbb{Z}_n^L \) and compute \( x_i^{(0)} = [\|x_i\|^{L} + m_{i-1}] \).

2. \( P_i \) sends \( x_i^{(0)} \) to \( P_{i+1} \).

3. \( P_{i+1} \), after receiving \( x_i^{(0)} \), adds it to \( \|x_i\|^{L} \) and pass it to \( H_{i+1} \).

4. \( H_{i+1} \) recovers the plaintext \( x = (x_i^{(1)} + \|x\|^{L} + \|x\|^{L}) \), where \( m_{i-1} \) is generated with \( \Pi_{\text{LTH.GenMask}}(n, L, i-1) \).

5. \( P_{i+1} \) and \( P_i \) share the two copies received and abort if any inconsistency is found.

**Proof for Theorem 1.** We first prove Theorem 1 of \( \Pi_{\text{ReLU}} \) by constructing a simulator \( S_{\text{STAMP}} \) such that no non-uniform PPT environment \( E \) can distinguish between: (i) the execution of the real protocol \( \text{EXEC}_{\text{ReLU}, S, E} \) where parties \( P_1, P_2, P_3 \) run \( \Pi_{\text{ReLU}} \) and the corrupted parties are controlled by a dummy adversary \( A \) who simply forward messages from/to \( E \), and (ii) the ideal execution \( \text{EXEC}_{\text{ReLU}, S_{\text{STAMP}}, E} \) where parties interact with \( F_{\text{ReLU}} \), while the simulator \( S_{\text{STAMP}} \) has the control over the corrupted party. Compared to the semi-honest scheme, the changed actions are written in blue.

**The Environment \( E \).** The environment \( E \) provides inputs \( (x_1, x_{1.1}) \) to \( P_1 \), \( (x_{3.2}, x_{3.2}) \) to \( P_2 \), \( (x_{3.3}, x_{3.3}) \) to \( P_3 \), which are forwarded to the ideal functionality \( F_{\text{ReLU}} \) in Figure 6. The environment \( E \) also indicates which party is corrupted to the ideal functionality.

**Case 1:** \( P_1 \) is corrupted \((i = 1)\) and \( P_2, P_3 \) are honest.

**The Simulator.** \( S_{\text{STAMP}} \) simulates the following interactions on receiving the signal from \( F_{\text{ReLU}} \):

- Upon receiving \( F_{\text{ReLU}}, 1, x_{3.3}, x_{3.3}, n, L \) from \( F_{\text{ReLU}} \), \( S_{\text{STAMP}} \) generates a random \( z_1 \), and use \( (x_{1.1}, \hat{x}_2) \) and \( (x_{2.2}, x_{3.3}) \) as dummy inputs for \( P_1, P_2, P_3 \), respectively.

- \( S_{\text{STAMP}} \) acts as \( F_{\text{GenMask}} \) to generate random masks \( m_3 \) for \( P_1, P_2, P_3 \) and then computes \( x_i^{(1)} = x_{3.1} + m_3 \) and sends \( (m_3, x_i^{(1)}) \) to \( P_1 \), computes \( x_i^{(2)} = x_{3.2} + m_3 \) and sends \( (m_3, x_i^{(2)}) \) to \( P_2 \), also sends \( x_i^{(1)} \) (as adversary input for \( P_1 \)) and \( x_i^{(2)} \) to \( P_2 \) on behalf of \( P_1, P_2, P_3 \), respectively.

- \( S_{\text{STAMP}} \) acts as \( F_{\text{GenMask}} \) to generate random masks \( m_2 \) for \( P_3 \) and \( P_2 \) and then computes \( x_i^{(0)} = x_{2.2} + m_2 \) and sends \( (m_2, x_i^{(0)}) \) to \( P_2 \), computes \( x_i^{(0)} = x_{2.3} + m_2 \) and sends \( (m_2, x_i^{(0)}) \) to \( P_2 \), also then sends \( x_i^{(0)} \) to \( P_2 \) on behalf of \( P_2 \) and \( P_3 \), respectively.

- \( S_{\text{STAMP}} \) acts as \( F_{\text{GenMask}} \) to generate random masks \( m_2 \) for \( P_3 \) and \( P_2 \) and then computes \( x_i^{(0)} = x_{2.3} + m_2 \) and sends \( (m_2, x_i^{(0)}) \) to \( P_2 \), computes \( x_i^{(0)} = x_{2.3} + m_2 \) and sends \( (m_2, x_i^{(0)}) \) to \( P_2 \), also then sends \( x_i^{(0)} \) to \( P_2 \) on behalf of \( P_2 \) and \( P_3 \), respectively.

- \( S_{\text{STAMP}} \) check \( x_i^{(0)} = x_i^{(1)} = x_i^{(1)} \), on behalf of \( P_2 \) and \( P_3 \) respectively, signal abort to \( F_{\text{ReLU}} \) if inconsistency found.

- \( S_{\text{STAMP}} \) acts as \( F_{\text{LTH.ReLU}} \) with \( P_2 \)'s input \( \hat{x} = x_i^{(1)} + \hat{x}_2 + x_{1.1} \) to (re)generate \( m_3 \), randoms \( z_1^*, z_2, z_3 \) such that \( z_1^* + z_2 + z_3 = 0 \), and then compute \( (z_1, z_2) = (\text{ReLU}(\hat{x} - m_3) + z_1^*, z_1^*) \).

**Indistinguishability.** We prove the indistinguishability argument by constructing a sequence of hybrid games as follows.

**Hybrid \( H_0 \):** This is the real protocol execution.

**Hybrid \( H_1 \):** \( H_1 \) is the same as \( H_0 \) except that \( \Pi_{\text{LTH.GenMask}} \) is replaced with simulated \( F_{\text{GenMask}} \), which outputs random \( m_3, m_2 \) for both step 1) and 4).

We claim that \( H_0 \) and \( H_1 \) are computationally indistinguishable. This is because \( \Pi_{\text{LTH.GenMask}} \) generates pseudo-random \( m_3, m_2 \) to \( P_1 \) using LTH. Due to the secure hardware assumption, there exists a simulator that is indistinguishable from the real hardware protocol execution. Moreover, due to the security of PRF used in LTH, the random \( m_3, m_2 \) produced by LTH is computationally indistinguishable from the random \( m_3, m_2 \) generated by the simulator. Therefore, \( H_0 \) and \( H_1 \) are computationally indistinguishable.

**Hybrid \( H_2 \):** \( H_2 \) is the same as \( H_1 \), except that we replace step 4) with the simulated \( F_{\text{LTH.ReLU}} \).

We claim that \( H_1 \) and \( H_2 \) are computationally indistinguishable using the same argument on the trusted hardware and PRF security as in \( H_1 \). Specifically, the random vectors generated by \( \Pi_{\text{LTH.GenMaskShare}} \) in the LTH are based on PRF and therefore, they
are computationally indistinguishable from the random vectors generated by the simulator. Therefore, \( \mathcal{H}_1 \) and \( \mathcal{H}_2 \) are computationally indistinguishable.

**Hybrid \( \mathcal{H}_3 \):** \( \mathcal{H}_3 \) is the same as \( \mathcal{H}_2 \), except that \( P_2, P_3 \) use dummy inputs for interaction, instead of the ones provided by the environment. In this hybrid, we introduce an ideal functionality \( F_{ReLU} \) that takes the environments’ actual inputs and returns the corresponding outputs.

We claim that \( \mathcal{H}_2 \) and \( \mathcal{H}_3 \) are indistinguishable. Since the corrupted party is \( P_1, S_{Stamp} \) knows \( x_{1,1} = x_{3,3}, x_{1,1} = x_{3,3} \). The dummy inputs would be \( x_{2,2} = x_{2,3} \) (represented by \( x_{2,1} \) in \( S_{Stamp} \)). The distribution of the computation result, \( x' = x_1' + x_2 + x_1 = (x_1 + m_1) + x_2 + x_1,1 = (x_1 + m_1) + x_2 + x_1,1 = (x_2 + m_2) + x_3 + x_1,1 \) is uniformly random since \( m_2, m_2 \) are random. Therefore, \( \mathcal{H}_2 \) and \( \mathcal{H}_3 \) are indistinguishable.

The adversary’s view of \( \mathcal{H}_3 \) is identical to \( EXEC_{F,S_{Stamp},E} \). Therefore, in Case 1 the view of \( A \) and \( E \) are indistinguishable in the real and the simulated world.

Putting it all together, we have that \( \mathcal{H}_0 \approx \mathcal{H}_1 \approx \mathcal{H}_2 \approx \mathcal{H}_3 = S_{Stamp} \).

**Case 2:** \( P_2 \) is corrupted (\( i = 2 \)) and \( P_1, P_3 \) are honest.

**The Simulator, \( S_{Stamp} \):** simulates the following interactions on receiving the signal from \( F_{ReLU} \):

- Upon receiving \( (ReLU, 2, x_{1,2}, x_{2,2}, n, l) \) from \( F_{ReLU}, S_{Stamp} \) generates a random \( x_1 \), and use \( (x_{2,2}, x_1, x_{3,1,2}) \) as dummy inputs for \( P_1 \) and \( P_1 \), respectively.

- \( S_{Stamp} \) acts as \( F_{GenMask} \) to generate random masks \( m_3 \) for \( P_1 \) and \( P_3 \), then computes \( x'_3 = x_3 + m_3 \) and sends \( (m_3, x'_3) \) to \( P_1 \). If it then also sends \( x'_3 \) and \( x''_3 \) to \( P_2 \), on behalf of \( P_1 \) and \( P_3 \), respectively.

- \( S_{Stamp} \) acts as \( F_{GenMask} \) to generate random masks \( m_2 \) for \( P_1 \), then computes \( x'_2 = x_2 + m_2 \) and sends \( (m_2, x'_2) \) to \( P_1 \). If it sends \( x'_2 \) and \( x''_2 \) as (adversary input for \( P_1 \)) to \( P_1 \), on behalf of \( P_1 \) and \( P_2 \), respectively.

- \( S_{Stamp} \) checks \( x''_1 = x'_1, x''_2 \), \( x''_3 \) equal to \( x''_2 \) on behalf of \( P_2 \) and \( P_1 \) respectively, signal abort to \( F_{ReLU} \) if inconsistent found.

- \( S_{Stamp} \) acts as \( F_{LTHReLU} \) with \( P_2 \)’s input \( x = x'_1 + x'_2 + x_{1,2} \) to (re)generate \( m_2 \), randoms \( z'_1, z_1, z'_2 \) such that \( z'_2 + z_2 + z'_2 = 0 \), and then compute \( (z_1, z_2) = (ReLU(x - m_2), z'_2, z_2) \). \( S_{Stamp} \) sends \( (z_1, z_2) \) to \( P_1 \). If it then also sends \( z'_2 \) to \( P_2 \) as (adversary inputs for \( P_1 \)).

- \( S_{Stamp} \) acts as \( F_{LTHReLU} \) with \( P_1 \)’s input \( x = x'_1 + x'_1 + x_{1,2} \) to (re)generate \( m_3 \), randoms \( z'_1, z_1, z'_2 \) such that \( z'_1 + z_1 + z'_2 = 0 \), and then compute \( (z_1, z_2) = \) \( (ReLU(x - m_3), z'_2, z_2) \). \( S_{Stamp} \) sends \( (z_1, z_2) \) to \( P_1 \). If it then also sends \( z'_2 \) to \( P_2 \) on behalf of \( P_1 \).

- \( S_{Stamp} \) acts as \( F_{GenMaskShr} \) for \( P_2 \) (to regenerate random \( z_2 \) and sends it to \( P_1 \). Similarly, \( S_{Stamp} \) acts as \( F_{GenMaskShr} \) for \( P_3 \) (to regenerate random \( z_3 \) and sends it to \( P_3 \). \( S_{Stamp} \) also generate \( z_2 \) for \( P_2 \) and \( P_1 \).

- \( S_{Stamp} \) checks if \( x'_1 = x_1 \) received by \( P_1 \), also \( P_2 \); if same \( z_2 \) received by \( P_2 \), also \( P_3 \); if same \( z_2 \) received by \( P_3 \), also \( P_1 \). Signal abort to \( F_{ReLU} \) if inconsistent found.

- Upon receiving \( (ReLU, 3, x_{2,3}, x_{3,3}, n, l) \) from \( F_{ReLU}, S_{Stamp} \) generates a random \( x_1 \), and use \( (x_{2,3}, x_1, x_{3,2,3}) \) as dummy inputs for \( P_1 \) and \( P_2 \), respectively.

- \( S_{Stamp} \) acts as \( F_{GenMask} \) to generate random masks \( m_1 \) for \( P_1 \) and \( P_3 \), then computes \( x'_3 = x_3 + m_3 \) and sends \( (m_3, x'_3) \) to \( P_1 \). Computes \( x'_3 \) also and (\( m_2, x'_2 \)) to \( P_2 \) then also sends \( x'_3 \) and \( x''_3 \) to \( P_2 \) on behalf of \( P_1 \) and \( P_3 \), respectively.

- \( S_{Stamp} \) acts as \( F_{GenMask} \) to generate random masks \( m_2 \) for \( P_3 \), then computes \( x'_2 = x_2 + m_2 \) and sends \( (m_2, x'_2) \) to \( P_3 \). Computes \( x'_2 \) also and (\( m_3, x'_3 \)) to \( P_3 \). Also sends \( x'_3 \) and \( x''_3 \) to \( P_2 \) as (adversary input for \( P_2 \)) to \( P_1 \), on behalf of \( P_1 \) and \( P_3 \), respectively.

- \( S_{Stamp} \) checks \( x''_1 = x'_1, x''_2 = x'_2 \) on behalf of \( P_2 \) and \( P_1 \) respectively, signal abort to \( F_{ReLU} \) if inconsistent found.

- \( S_{Stamp} \) acts as \( F_{LTHReLU} \) with \( P_2 \)’s input \( x = x'_3 + x_{2,3} + x_1 \) to (re)generate \( m_3 \), randoms \( z'_1, z_1, z'_2 \) such that \( z'_1 + z_1 + z'_2 = 0 \), and then compute \( (z_1, z_2) = (ReLU(x - m_3), z'_2, z_2) \). \( S_{Stamp} \) sends \( (z_1, z_2) \) to \( P_2 \). If it then also sends \( z'_2 \) to \( P_2 \) on behalf of \( P_1 \).

- \( S_{Stamp} \) acts as \( F_{LTHReLU} \) with \( P_1 \)’s input \( x = x'_3 + x_1 + x_{3,3,3} \) (re)generate \( m_3 \) to generate randoms \( z'_3, z_3, z_2 \) such that \( z'_3 + z_3 + z_2 = 0 \), and then compute \( (z_1, z_2) = (ReLU(x - m_3), z'_2, z_2) \). \( S_{Stamp} \) sends \( (z_1, z_2) \) to \( P_1 \). If it then also sends \( z'_2 \) to \( P_2 \) on behalf of \( P_1 \).

- \( S_{Stamp} \) acts as \( F_{GenMaskShr} \) for \( P_1 \) (to (re)generate random \( z_3 \) and sends it to \( P_1 \). Similarly, \( S_{Stamp} \) acts as \( F_{GenMaskShr} \) for \( P_2 \) (to (re)generate random \( z_3 \) and sends it to \( P_3 \). \( S_{Stamp} \) also generate \( z_2 \) for \( P_2 \) and \( P_3 \).
• $S_{\text{STAMP}}$ checks if $z'_i = z_i^*$ received by $P_1$, also $P_2$; if same $z_2$ received by $P_2$, also $P_3$; if same $z_3$ received by $P_3$, also $P_1$. Signal abort to $\mathcal{F}_{\text{RelU}}$ if an inconsistency is found. If no abort is signaled, $S_{\text{STAMP}}$ signals (RelUend, 3, z2, z3) to $\mathcal{F}_{\text{RelU}}$.

Indistinguishability. We prove the indistinguishability argument by constructing a sequence of hybrid games as follows. Notice that the first 3 games, Hybrid $\mathcal{H}_0$, Hybrid $\mathcal{H}_1$ and Hybrid $\mathcal{H}_2$ are identical as Case 1’s. The proofs between Hybrid $\mathcal{H}_0$ and Hybrid $\mathcal{H}_1$, Hybrid $\mathcal{H}_1$ and Hybrid $\mathcal{H}_2$ are exactly the same. Hybrid $\mathcal{H}_2$, $\mathcal{H}_3$ is the same as $\mathcal{H}_2$, except that $P_1, P_3$ use dummy inputs for interaction, instead of the ones provided by the environment. In this hybrid, we introduce an ideal functionality $\mathcal{F}_{\text{RelU}}$ that takes the environments’ actual inputs and returns the corresponding outputs.

We claim that $\mathcal{H}_2$ and $\mathcal{H}_3$ are indistinguishable. Since the corrupted party is $P_2$, $S_{\text{STAMP}}$ knows $x_{2,3} = x_{2,2} x_{3,3} = x_{3,1}$. The dummy inputs would be $x_{1,1} = x_{1,2}$ (represented by $\hat{x}_1$ in $S_{\text{STAMP}}$). The distribution of the computation result, $x = x_{1,2} x_{2,3} = x_{1,1} x_{2,3} x_{3,3} + x_{2,3} + x_{1,1} x_{1,1} x_{2,3} + x_{2,3} + x_{1,1} x_{2,3} x_{3,3} = (x_{3,3} + m_3) x_{2,3} + x_{1,1} x_{2,3} + 3$ are uniformly random since $m_3, m_2$ are random. Therefore, $\mathcal{H}_2$ and $\mathcal{H}_3$ are indistinguishable.

The adversary’s view of $\mathcal{H}_3$ is identical to EXEC$_F, S_{\text{STAMP}}$, $E$. Therefore, in Case 1 the view of $\mathcal{A}$ and $\mathcal{E}$ are indistinguishable in the real and the simulated world.

Putting it all together, we have that $\mathcal{H}_0 \approx \mathcal{H}_1 \approx H_2 \approx H_3 = S_{\text{STAMP}}$ and this completes the proof. □

Notice that as discussed in §4.3, the above proof works identically for MaxPooling and BatchNorm since only the plaintext computations after subtracting the masks are different.

Next, we will prove the Theorem 2. We provide the complete proof for case 1 (where $P_1$ is corrupted) due to the space limit, and the proof of the other two cases are similar as provided in the proof for $\Pi_{\text{RelU}}$.

\[ \mathcal{F}_{\text{MatMulReLU}} \]

(1) Upon receiving inputs $\langle A_{3,1}, A_{2,1}, B_{3,1}, B_{2,1}, A_{2,2}, B_{2,2}, A_{2,3}, B_{2,3}, B_{3,3} \rangle$ from $\mathcal{F}_{\text{MatReLU}}$, $S_{\text{STAMP}}$ generates random $A_{3,2}, B_{3,2}$, and use $\langle A_{4,1}, A_2, B_{1,1}, B_{2,2} \rangle$ and $\langle A_{2,3}, B_{3,1}, B_{3,2} \rangle$ as dummy inputs for $P_2$ and $P_3$, respectively.

$S_{\text{STAMP}}$ invokes the simulator of the secure multiplication protocol $\Pi_{\text{Mat-Mult}}$ [58] without the truncation. In the end, $\langle C_{3,1}, C_{1,1} \rangle, \langle C_{2,2}, C_{2,1} \rangle, \langle C_{3,2}, C_{3,1} \rangle, \langle C_{3,3}, C_{1,1} \rangle$ are distributed accordingly and $S_{\text{STAMP}}$ will signal abort to $\mathcal{F}_{\text{MatReLU}}$ if inconsistency was found in the distributed shares.

$S_{\text{STAMP}}$ acts as $\mathcal{F}_{\text{GenMask}}$ to generate random masks $M_2$ for $P_2$ and $P_3$, $M_3$ for $P_3$ and $P_1$ so that $M_1 + M_2 + M_3 = 0$. Then $S_{\text{STAMP}}$ computes $C_{2,2} = C_{2,2} + M_2$ as $P_2$ and $C_{3,3} = C_{2,3} + M_2$ as $P_3$; $C_{3,3}' = C_{3,3} + M_3$ as $P_3$ and $C_{3,3}' = C_{3,3} + M_1$ as $P_1$.

$s_{\text{STAMP}}$ sends $C_{2,2}'$, $C_{3,3}'$ to $P_1$ as $P_2$ and $P_3$ sends $C_{3,3}'$, $C_{3,3}'$ to $P_2$ as $P_3$ and $P_1$. $S_{\text{STAMP}}$ aborts if any inconsistency is found on the pairs.

$S_{\text{STAMP}}$ computes $C_{1,1}' = C_{1,1}, C_{3,1}' + C_{3,1}', C_{1,2}' = C_{1,2}, C_{1,2}' + C_{3,2}$ as $P_2$.

$S_{\text{STAMP}}$ acts as $\mathcal{F}_{\text{THReLU}} (H_1)$ with $P_1$’s input $C_{1,1}'$ to (re)generate $M_2$, computes $\hat{C} = (C_{1,1}' - M_2) \gg sp_{\text{STAMP}}$ computes $D = \hat{C} > 0$, generates $Z_{1}' + Z_{2}' + Z_{3}' = 0$ (as $\mathcal{F}_{\text{GenMaskShr}}$), then obtain $(Z_{1}', Z_{2}', Z_{3})$ as $(Z_{1,3}, Z_{1,3})$ for $P_1$, $S_{\text{STAMP}}$ acts as $\mathcal{F}_{\text{THReLU}} (H_2)$ with $P_2$’s input $C_{2,2}'$ to (re)generate $M_3$, computes $\hat{C} = (C_{2,2}' - M_3) \gg sp_{\text{STAMP}}$ computes $D = \hat{C} > 0$ (as $\mathcal{F}_{\text{GenMaskShr}}$), generates $Z_{1}' + Z_{2}' + Z_{3}' = 0$, then obtain $(Z_{1}', Z_{2}', Z_{3})$ as $(Z_{1,2}, Z_{2,2})$ for $P_1$.

$S_{\text{STAMP}}$ acts as $\mathcal{F}_{\text{GenMaskShr}}$ for $P_3$ to (re)generate $(Z_{2,3}, Z_{3,3}) = (Z_{2,3}, Z_{3,3})$ and sends it to $P_2$. $S_{\text{STAMP}}$ compare $Z_{1,1}' = Z_{1,2}'$ as $P_1$ and $P_2$; $Z_{2,2}' = Z_{2,3}'$; $Z_{3,3}' = Z_{3,1}'$ as $P_3$. Signal abort to $\mathcal{F}_{\text{MatReLU}}$ if inconsistency was found.

Indistinguishability. We prove the indistinguishability argument by constructing a sequence of hybrid games as follows.

Hybrid $\mathcal{H}_0$: This is the real protocol execution.

\[ \mathcal{F}_{\text{MatMulReLU}} \]

Upon receiving signal with inputs $C'$, masks $M$ and party index $i$, compute the following:

1. $D = C' - M; E = D > 0$.

2. Generate random $C_1, C_2, C_3$ such that $C_1 + C_2 + C_3 = C$. Returns $(C_1, C_{i+1})$ to $P_1$ (as mentioned in notation, for party index $i + 1$ means the next party).

Figure 9: Ideal functionality for the LTH part of $\Pi_{\text{RelU}}$. 

Proof for Theorem 2. We prove Theorem 2 by constructing a simulator and a series of hybrid games similar to the Proof of Theorem 1, with $E$ providing inputs to parties.

Case 1: $P_1$ is corrupted ($i = 1$) and $P_2, P_3$ are honest.

The Simulator. $S_{\text{STAMP}}$ simulates the following interactions on receiving the signal from $\mathcal{F}_{\text{MatReLU}}$:
Hybrid $H_1$: $H_1$ is the same as $H_0$, except that $Π_{LTHGenMask}$ is replaced with simulated $F_{GenMask}$ that outputs random $M_2, M_3$ for both step 2) and 5).

Hybrid $H_2$: $H_2$ is the same as $H_1$, except that we replace step 5) with the simulated $F_{LTHMatMulReLU}$.

Proofs of $H_0$ and $H_1$, $H_2$ and $H_5$ being computationally indistinguishable are similar to the proof of Theorem 1, with the actual random masks replaced by $M_2, M_3$.

Hybrid $H_3$: $H_3$ is the same as $H_2$, except that $P_2, P_3$ use dummy inputs for interaction, instead of the ones provided by the environment. Also $Π_{mal-arith-mult}$ of [58] is replaced by its corresponding simulator. In this hybrid, we introduce an ideal functionality $F_{ReLU}$ that takes the environments’ actual inputs and returns the corresponding outputs.

We claim that $H_2$ and $H_3$ are indistinguishable, $H_2$ and $H_5$ are only different in inputs and step 1), and their indistinguishability directly comes from the security of the simulator of $Π_{mal-arith-mult}$. Again we refer the reader to [58] for more details. Since the view of $P_1$ does not change after step 1), it remains the same for the whole protocol since later steps remain the same in both hybrids. Therefore, $H_2$ and $H_3$ are indistinguishable.

The adversary’s view of $H_3$ is identical to EXEC$_F, S_{ Stamp}, E$. Therefore, in Case 1 the view of $A$ and $E$ are indistinguishable in the real and the simulated world.

Putting it all together, we have that $H_0 \equiv H_1 \equiv H_2 \equiv H_3 = S_{ Stamp}$.

Notice that as discussed in §4.3, the above proof works identically for MatMulBatchNormReLU and MatMulMaxPoolReLU since only the plaintext computations after subtracting the masks are different.

Next, we will prove the Theorem 3. We provide the complete proof for case 1 (where $P_1$ is corrupted) due to the space limit, and the proof of the other two cases is similar to what we did previously.

---

$F_{SoftmaxMax}$

(1) Upon receiving inputs $(x_{3,1}, x_{1,1}), (x_{1,2}, x_{2,2}), (x_{2,3}, x_{3,3})$ from $P_1, P_2, P_3$ respectively, check if $x_{j,j} = x_{j,j+1}$ for $j = 1, 2, 3$. If not, notify abort. Otherwise, compute $x = x_{1,1} + x_{2,2} + x_{3,3}, z = \exp(x \gg fp) \ll fp$. Then, send a signal (Softmax, $x_{i-1,i}, x_{i,i}, n, L, i$) to $S_{ Stamp}$ that includes the inputs of $P_i$, the size of inputs of $P_{i+1}$ and $P_{i-1}$, and the index $i$ of the corrupted party.

(2) Upon receiving (Softmaxend, $i, z_{i-1}, z_i$) from $S_{ Stamp}$, let $z_{i+1} = z - z_i - 1$. Return $(z_i, z_{i+1})$ to the $P_i$ for $i = 1, 2, 3$.

---

Figure 10: Ideal functionality for $Π_{Softmax}$

Proof for Theorem 3. We prove Theorem 3 by constructing a simulator and a series of hybrid games similar to the Proof of Theorem 1, with $E$ providing inputs to parties.

Case 1: $P_1$ is corrupted ($i = 1$) and $P_2, P_3$ are honest.

$F_{SoftmaxMax}$

Upon receiving signal with inputs $x'$, masks $m$ and party index $i$, compute the following:

1. $x = x' - m, a = \lfloor \exp(x \gg fp) \rfloor \ll fp$.
2. Generate random $z_1, z_2, z_3$ such that $z_1 + z_2 + z_3 = z$. Returns $(z_i, z_{i+1})$ to $P_i$ (as mentioned in notation, for party index $i + 1$ means the next party).

Figure 11: Ideal functionality for the LTH part of $Π_{Softmax}$.

The Simulator. $S_{ Stamp}$ simulates the following interactions on receiving the signal from $F_{ReLU}$:

- Upon receiving (Softmax, $1, x_{3,1}, x_{1,1}, n, L$) from $F_{Softmax}$, $S_{ Stamp}$ generates a random $x_2$, and use $(x_{1,1}, x_2)$ and $(x_{2,1}, x_{3,1})$ as dummy inputs for $P_2$ and $P_3$, respectively.
- $S_{ Stamp}$ acts as $F_{GenMask}$ to generate random masks $α_j, β_j \in Z_{256}$ for $j = 1, ..., n$ for $P_1$, then computes $r = \exp((x_{1,1})_j \gg fp)$ for $j = 1, ..., n$. Let $r = \exp((x_{1,1})_j \gg fp) = 2^{q_j} \cdot (m_j \gg 52)$ as noted. $S_{ Stamp}$ computes $\{m'_j, q'_j\} = \{q_j + β_j, 2^{q_j}\}$ for $j = 1, ..., n$ as $P_1$ sending to $P_2$, $S_{ Stamp}$ repeat above for $P_2$, replacing $x_{3,1}$ with $x_{3,1}$; $S_{ Stamp}$ again repeat above as $P_2$ and $P_3$, each replacing $x_{1,1}$ with $x_{1,2}$ and $x_{2,2}$, with $Π'_{LTH}$ generating $(α', β')$, and get $\{m''_j, q''_j\}$ respectively. $\{m'_j, q'_j\}$ are for $P_1$ provided by $P_1$ and $P_2$, and $(m''_j, q''_j)$ are for $P_1$ provided by $P_2$ and $P_3$.
- $S_{ Stamp}$ compare the received copies as $P_2$ and $P_1$, and signal abort to $F_{Softmax}$ if an inconsistency is found. $S_{ Stamp}$ computes as $P_1: 2^{(q'_j / l)} \cdot (m_j)_j = \exp((x_{1,3})_j + (x_{1,1})_j \gg fp)$ for $j = 1, ..., n$. $S_{ Stamp}$ computes as $P_1: 2^{(q''_j / l)} \cdot (m_j)_j = \exp((x_{1,3})_j + (x_{1,1})_j \gg fp)$ for $j = 1, ..., n$. $S_{ Stamp}$ acts as $F_{LTHSoftmax}$ with $P_2'$'s input $\{q''_j, q'_j, m''_j, m'_j\}$ to $H_1$ to (re)generate $(α', β')$, then locally compute $y$ as in step 4) $S_{ Stamp}$ then generate $m_1 + m_3 = 0$ and $(m_1 + y, m_2)$ as $(y_{y, y}, y_2)$ for $P_2$. $S_{ Stamp}$ acts as $F_{LTHSoftmax}$ with $P_1'$'s input $\{q''_j, q'_j, m''_j, m'_j\}$ to $H_1$ to (re)generate $(α', β')$, then locally compute $y$ as in step 4). $S_{ Stamp}$ then generate $m_1 + m_3 = 0$ and $(m_3 + m_1)$ as $(y_{y, y}, y_1)$ for $P_1$. $S_{ Stamp}$ signal $S_{ Stamp}$ checks $S_{ Stamp}$ compare $y_{y, y} = y_{y, y}$ as $P_1$ and $P_2$: $y_{y, y} = y_{y, y}, y_{y, y} = y_{y, y}$ as $P_3$. Signal abort to $F_{Softmax}$ if an inconsistency is found. If no abort is signaled, $S_{ Stamp}$ signals (Softmaxend, $1, y_{y, y}, 1, y_{1, y}$) to $F_{Softmax}$.

Indistinguishability. We prove the indistinguishability argument by constructing a sequence of hybrid games as follows.

Hybrid $H_0$ This is the real protocol execution.

Hybrid $H_1$: $H_1$ is the same as $H_0$, except that $Π_{LTHGenMask}$ is replaced with simulated $F_{GenMask}$ that outputs random $(α, β)$ and $(α', β')$ for both step 1) and 3).

Hybrid $H_2$: $H_2$ is the same as $H_1$, except that we replace step 4) with the simulated $F_{LTHSoftmax}$.
Table 7: Analytical cost analysis of the network communication rounds and amount (in Bytes) under the semi-honest setting, and the local bus communication with the LEE. Here, \( n = m \times m \) is the input size, \( s \) is the stride, \( w \times w \) is the filter size, and \( e \) is the precision parameter for the exponent (\( \exp(x) \approx (1 + \frac{x}{5})^x \)). We did not include CryptGPU and Goten because they did not focus on optimization of their non-linear layer protocols and provide no analytical cost analysis.

| Communication Type | Framework | RelU | MaxPool | BatchNorm | Softmax |
|--------------------|-----------|------|---------|-----------|---------|
| Network comm. rounds | Falcon+ | 10 | 12\((w^2 - 1)\) | 335 | 12\(n + p + 317\) |
| | AriaNN | 2 | 3 | 9 | - |
| | STAMP | 2 | 2 | 2 | - |
| Network comm. data | Falcon+ | 16\(n\) | \((20 + w^2)(\frac{w}{2} - 1)^2\) | 224\(n\) | \((\frac{n}{5} - 1)^2(n + 20) + (110 + e)n\) |
| | AriaNN | 12\(n\) | \((\frac{w}{2} + 1)^2(w^4 + 1)\) | 72\(n\) | - |
| | STAMP | 5\(n\) | \(\frac{2}{3}(2n + 2w^2 + 5(w + 1)^2)\) | 4\(n\) | \(\frac{8}{7}n\) |
| LTH comm. | STAMP | \(\frac{23}{11}n\) | \(\frac{1}{4}(8m^2 + 8w^2 + 15(\frac{m}{2} + 1)^2)\) | \(\frac{15}{11}n\) | \(\frac{29}{23}n\) |

Proofs of \( H_0 \) and \( H_1, H_2 \) and \( H_3 \) being computationally indistinguishable are similar to the proof of Theorem 1, with the actual random masks replaced by \((\alpha, \beta)\) and \((\alpha', \beta')\).

**Hybrid** \( H_1 \) and \( H_2 \) is the same as \( H_2 \), except that \( P_2, P_3 \) use dummy inputs for interaction, instead of the ones provided by the environment. In this hybrid, we introduce an ideal functionality \( F_{\text{Softmax}} \) that takes the environments' actual inputs and returns the corresponding outputs.

We claim that \( H_2 \) and \( H_3 \) are indistinguishable. Since the corrupted party is \( P_1 \), \( S_{\text{STAMP}} \) knows \( x_{3,1} = x_{3,2}, x_{4,1} = x_{4,2} \). The dummy inputs would be \( x_{3,2} = x_{3,3}, x_{4,1} = x_{4,2} \) (represented by \( x_3 \) in \( S_{\text{STAMP}} \)). The computation result sent to \( P_3 \) by \( P_3 \) in step 1) used the dummy inputs, and \( \{m_i' = (m_i + \alpha')_{3st}, q_i' = (q_i + \beta')_{3st}\} \) for \( i = 1, \ldots, n \) are uniformly random since \((\alpha', \beta')\) are random. Therefore the views of adversary in step 1) are not distinguishable in both hybrids, and its views of step 3) and 4) are also not distinguishable in both hybrids due the uniformly masked output. Therefore, \( H_2 \) and \( H_3 \) are indistinguishable.

The adversary’s view of \( H_3 \) is identical to EXEC\(_{F, S_{\text{STAMP}}, E}\). Therefore, in **Case 1** the view of \( A \) and \( E \) are indistinguishable in the real and the simulated world.

Putting it all together, we have that \( H_0 \approx H_1 \approx H_2 \approx H_3 \approx S_{\text{STAMP}} \).

**D ANALYTICAL COST ANALYSIS**

The cost analysis of our protocol is shown in Table 7, compared with baselines with analytical results provided in their work. The byte size of the finite field is chosen to be 4 and we count the exponent and the mantissa part in Protocol 5 as 4 and 8 bytes. We see significant improvements in inter-party communication rounds compared to Falcon, and significant theoretical reduction in the amount of communication data compared to both Falcon and AriaNN.

The actual speedup of a particular neural network depends on its structure including the ratio between linear and non-linear operations, the order of linear/non-linear operations/layers (which determines if protocols like Protocol 4 can be applied), the input dimensions, etc. The communication setting and the computational power also matter. We discuss the performance in §5.

**E MEMORY USAGE ANALYSIS**

As LTH typically does not support off-chip DRAM, STAMP needs to be able to run using a small on-chip SRAM. Here, we analyze the LTH memory usage of STAMP and show that the small on-chip SRAM is sufficient even for large ML models. For our experiments, our implementation runs on a Arduino Due microcontroller, as discussed in §5.1. In this prototype, the code occupies 23 KB of flash memory, which is less than 4% of the total capacity (512 KB) of Arduino Due’s flash memory. The LTH code uses up to 43 KB of on-chip SRAM during the execution, including the buffers for variables, space used by Arduino’s libraries and middleware (e.g., SerialUSB functions), and other usage like the function call stack.

In order to more fully understand the LTH memory requirement for larger models that were not run in our experiments, we provide analytical memory usage numbers of different non-linear operations. We list all the SRAM memory usage of the non-linear operations in Table 8. Note that the memory usage is constant for ReLU and BatchNorm. This is because they are scalar-wise operations during the inference phase. Therefore, each individual scalar of an input vector can be processed independently. The MaxPool operation has a dependency on the window size, which is rather small in all the models used in practice. All of the above three operations need no more than 0.5 KB of SRAM and, therefore, will not pose any memory usage issue even for larger models. As a reference point, our prototype has 96 KB SRAM for LTH.

Table 8: Analytical analysis of the minimum SRAM usage for each operations. Here, \( n \) is the plaintext input vector size, \( w \times w \times h \) is the maxpool window size, and \( l \) is the normal variable size (in our case 4 bytes). Dynamic buffer reuse is considered.

| Operations | Parameters | LTH Least SRAM Usage |
|------------|-----------|----------------------|
| ReLU       | -         | 2\(l\)               |
| MaxPool    | \(\{w, h\}\) | 2\(wH\)               |
| BatchNorm  | -         | 2\(l\)               |
| LayerNorm  | \(n\)     | \((2n + 1)l\)         |
| Softmax    | \(n\)     | 6\(nl\)              |

However, the minimum memory usage of the Softmax and LayerNorm operation depends on the input vector length, which can be large in some model structure. In our prototype, LTH cannot host all the variables needed within its SRAM with \( n \) greater than 3925. For the models in our experiments, the largest vector size is 200. Even modern Transformer models such as GPT-3 2.7B have
the maximum $\mathbf{n}$ not larger than 2560 in its model structure. In that sense, LTH will be able to support many modern ML models even with a relatively small SRAM capacity without protocol changes.

In a case when an input vector size is too large to fit into the LTH SRAM, the Stamp protocol can be slightly modified to break down the input vector into multiple smaller chunks, and perform non-linear operations in multiple rounds. Here, we show this approach using Softmax. The party first cuts the input of step (2) of Protocol 5 into chunks in the host CPU, and sends them to the LTH. The LTH recovers and computes the exponent of each input chunk as in step (3) of Protocol 5 until the $\mathbf{\bullet}$ accumulates the exponents locally and then sends the exponent results back to the host CPU with temporal generated masks calling Protocol 2; After all chunks are summed, the party sends again the masked exponent results and LTH will continue step (3) of Protocol 5. If such a multi-round Softmax is used, the communication cost of LTH will increase from $\frac{20}{3} \mathbf{n}$ to $\frac{48}{7} \mathbf{n}$. Similar steps can be taken for LayerNorm.

In summary, Stamp should be able to handle most models with our current LTH setting, and can be modified to handle even larger models with some increase in the LTH local communication cost.