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[1] / 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

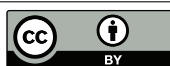
As the world increasingly relies on machine learning (ML) for everyday tasks, a large amount of potentially sensitive or private data need to be processed by ML learning algorithms. For example, ML models for medical applications may need to use private datasets distributed in multiple nations as inputs [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.

[1]This work was done while the author was at Meta.



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/popets-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], AriaNN [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]:

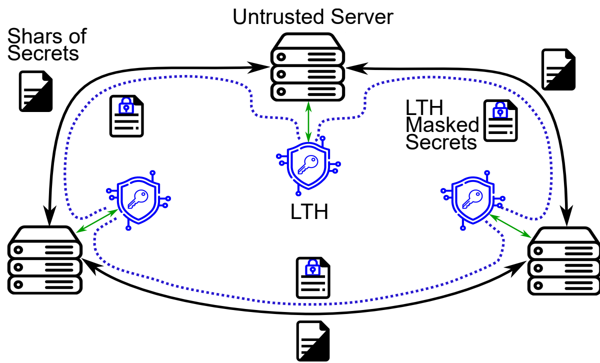


Figure 1: Stamp system and threat model. The black local machines owned by three parties, green local buses, and black inter-party communication channels are untrusted. The blue LTHs are trusted and contain secret keys shared among LTHs.

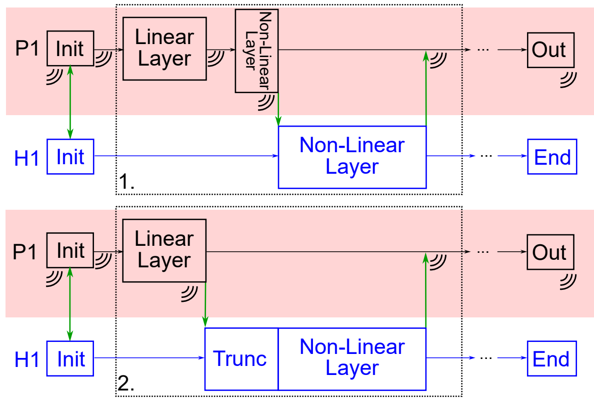


Figure 2: The Stamp execution flow on one of the parties. Inter-party communication (wave symbol) and the local communication with the LTH (green arrows) happen during initialization and execution, with (1) or without (2) the optimization in §4.2. An adversary has complete control over the data and operations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

Definition 1 (Simulation-based security: privacy and verifiability). A protocol $\pi_{\mathcal{F}}$ 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 negl such that

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

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.

Note that Stamp, similar to other secure computation techniques based on TEE, MPC and / or homomorphic encryption (HE), does not prevent attacks that poison the model through malicious inputs or extract information from the trained parameters or model outputs [9, 83]. To defend against such algorithmic attacks, a secure computing framework such as Stamp needs to be combined with other orthogonal defense techniques (e.g., Differential Privacy [1], out-of-distribution points removal [95]). Additionally, Stamp primarily targets private inference, not training, so training data poisoning attacks are not its main concern.

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. fp is the fix-point precision. We use the bold font $\mathbf{a}$ or $\mathbf{A}$ to represent a vector or a matrix. We use a_i, $(\mathbf{a})_i$ or $A_{i,j}$ to represent the ith element of the vector $\mathbf{a}$ or the element of the matrix $\mathbf{A}$ in the ith row and in the jth column. This is different from the bold $\mathbf{A}_i$, 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 $(a+b)_L$ 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 $\bar{a}$, $\overline{\exp(a)}$, to represent that a number or an output is a real number. The right-shift operation is indicated as $\gg$ (e.g., $a \gg b = a/2^b$). 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_L 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 $\lfloor \bar{a} \rceil$ 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_3 and P_2). The RSS of an integer secret $x \in \mathbb{Z}_L$ is denoted as $[\![x]\!]^L = ([\![x]\!]_1^L, [\![x]\!]_2^L, [\![x]\!]_3^L)$, where L is the size of the finite field to which the shares belong and $x = [\![x]\!]_1^L + [\![x]\!]_2^L + [\![x]\!]_3^L$. When a secret x is shared as $[\![x]\!]^L$, party P_i holds $([\![x]\!]_i^L, [\![x]\!]_{i+1}^L)$ for $i = 1, 2, 3$. To generate the integer representation x based on the real value $\bar{x}$, we use two's complement fixed-point encoding with fp bits of precision. For a positive $\bar{x}$ we have $x = \lfloor \bar{x} \cdot 2^{\text{fp}} \rfloor$, while for a negative $\bar{x}$, $x = \lfloor \bar{x} \cdot 2^{\text{fp}} \rfloor + L$, assuming that x is within the bound $[-L/2^{\text{fp}}, L/2^{\text{fp}})$.

In our experiments, we mainly use the cases of $l = 32$, with $\text{fp} = 13$ and $L = 2^l$ (which supports inputs from -262144 to $262144 - 2^{-13}$), to match the bit-width used in the baseline MPC schemes. The security of a $l = 32$ 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.

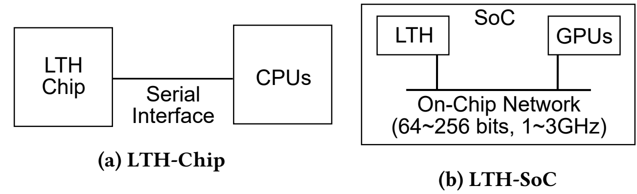
Multiplications. $[\![x \cdot y]\!]^L \leftarrow \Pi_{\text{Mul}}([\![x]\!]^L, [\![y]\!]^L)$: To get $[\![z]\!]^L = [\![x{\cdot}y]\!]^L$, P_i first computes $\hat{z}_i = [\![x_i]\!]^L [\![y_i]\!]^L + [\![x_{i+1}]\!]^L [\![y_i]\!]^L + [\![x_i]\!]^L [\![y_{i+1}]\!]^L$, then $(\hat{z}_1, \hat{z}_2, \hat{z}_3)$ is already 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 $\{\alpha_i\}$ to mask $\hat{z}_i$ as $z_i = \hat{z}_i + \alpha_i$, 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. $([\![\mathbf{AB}]\!]^L) \leftarrow \Pi_{\text{MatMul}}([\![\mathbf{A}]\!]^L, [\![\mathbf{B}]\!]^L)$: To perform matrix multiplication $[\![\mathbf{C}_{a\times c}]\!]^L = [\![\mathbf{A}_{a\times b}\mathbf{B}_{b\times c}]\!]^L$, simply applying Π_{Mul} for each multiplication leads to $O(abc)$ shares to be sent. The parties can instead perform part of the addition of shares locally (i.e., $[\![\hat{\mathbf{C}}]\!]_i^L = [\![\mathbf{A}]\!]_i^L [\![\mathbf{B}]\!]_i^L + [\![\mathbf{A}]\!]_i^L [\![\mathbf{B}]\!]_{i+1}^L + [\![\mathbf{A}]\!]_{i+1}^L [\![\mathbf{B}]\!]_i^L$) and then share $[\![\hat{\mathbf{C}}]\!]_i^L$ 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 fp bits) must follow after a multiplication to correct the fixed-point precision in 3-party MPC. We refer the readers to ABY3 [58] for more details on the 3-party truncation protocol, to prior work [22, 58] for the malicious variant of Π_{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: Synopsys 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.



(a) LTH-Chip **(b) LTH-SoC**

Figure 3: Two types of LTHs that STAMP considers.

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, NPUs, 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 $1mm^2$ of the silicon area. On the other hand, a high-performance CPU takes hundreds of mm^2 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 Icelake/TDX. AMD SEV has no replay protection against physical attacks. NVIDIA GPU 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 Π_{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 Π_{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 Π_{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, Π_{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 $\Pi_{\mathsf{LTH.GenMask}}$ and $\Pi_{\mathsf{LTH.GenMaskShare}}$ in the LTH as Protocol 2 and Protocol 3. They are very similar with only a minor difference that $\Pi_{\mathsf{LTH.GenMaskShare}}$ always generates shares of $\mathbf{0}$. Four counters $\{\mathsf{ctr}_1^i, \mathsf{ctr}_2^i, \mathsf{ctr}_3^i, \mathsf{ctr}_s^i\}$ are used in each H_i to maintain consistency among H_is in a semi-honest setting, and additional four

Protocol 1 Π_{Init} Initialization

Input. Security parameter λ.
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_is perform 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_is to obtain $O(\lambda)$-bit PRF keys $k_{i,i+1}, k_{i-1,i}$, then use $F_{k_{i,i+1}}$ to mask one key $k'_{i-1,i} \equiv k_{i-1,i} + F_{k_{i,i+1}}(0) \mod p$ and send $k'_{i-1,i}$ to H_{i+1} through P_i. H_i would receive $k'_{i+1,i-1}$ from H_{i-1} and can recover $k_{i+1,i-1} = k'_{i+1,i-1} - F_{k_{i+1,i-1}}(0)$.

Protocol 2 $\mathbf{m} \leftarrow \Pi_{\mathsf{LTH.GenMask}}(n, L, j; i, \mathsf{ctr}^i_j, k_{j,j+1})$

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, the counter ctr^i_j and the key $k_{j,j+1}$ are stored in the LTH.
Output. pseudo-random masks $\mathbf{m} \in \mathbf{Z}_L$ and updated counter ctr^i_j.
$\mathbf{m} = (F_{k_{j,j+1}}(\mathsf{ctr}^i_j), F_{k_{j,j+1}}(\mathsf{ctr}^i_j + 1), ..., F_{k_{j,j+1}}(\mathsf{ctr}^i_j + n - 1))$.
Update $\mathsf{ctr}^i_j \leftarrow \mathsf{ctr}^i_j + n$.

Protocol 3 $(\llbracket \mathbf{m}_j \rrbracket^L_i, \llbracket \mathbf{m}_j \rrbracket^L_{i+1}) \leftarrow \Pi_{\mathsf{LTH.GenMaskShare}}(n, L; i, \mathsf{ctr}^i_s, k_{i,i+1}, k_{i+1,i-1}, k_{i-1,i})$

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 $\llbracket \mathbf{m} \rrbracket^L_i, \llbracket \mathbf{m} \rrbracket^L_{i+1} \in \mathbb{Z}_L$
$\llbracket m_j \rrbracket^L_i = F_{k_{i,i+1}}(\mathsf{ctr}^i_s + j) - F_{k_{i+1,i-1}}(\mathsf{ctr}^i_s + j)$ for $j = 0, ..., n-1$
$\llbracket m_j \rrbracket^L_{i+1} = F_{k_{i+1,i-1}}(\mathsf{ctr}^i_s + j) - F_{k_{i-1,i}}(\mathsf{ctr}^i_s + j)$ for $j = 0, ..., n-1$
Update $\mathsf{ctr}^i_s \leftarrow \mathsf{ctr}^i_s + n$.

$\{\hat{\mathsf{ctr}}^i_1, \hat{\mathsf{ctr}}^i_2, \hat{\mathsf{ctr}}^i_3, \hat{\mathsf{ctr}}^i_s\}$ 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 $k_{i-1,i}$ with H_{i-1}, $k_{i,i+1}$ with H_{i+1}, and then also $k_{i+1,i-1}$ 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 RSS 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 $\Pi_{\mathsf{LTH.GenMask}}$ to get the pseudo-random masks, transmitting the masked shares, recovering the plaintext to compute inside LTHs, and then generate and distribute the psuedo-random shares of the results. We show the details of this protocol (Π_{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 §3.2, in a fixed-point setting, truncation is required after each multiplication to keep the consistency of the precision. If we apply Π_{ReLU} directly after the completion of multiplications, the communication overhead will be the multiplication / truncation overhead and the Π_{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 $\Pi_{\mathsf{MatMulReLU}}$, 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. $\Pi_{\mathsf{MatMulReLU}}$ 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 psuedorandom 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 $\Pi_{\mathsf{mal\text{-}arith\text{-}mult}}$ 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

Π_{ReLU} can be extended to $\Pi_{\mathsf{MaxPooling}}$,$\Pi_{\mathsf{BatchNorm}}$,$\Pi_{\mathsf{LayerNorm}}$ that are common non-linear operations needed in deep learning networks. $\Pi_{\mathsf{MaxPooling}}$ needs comparisons and multiplexing. $\Pi_{\mathsf{BatchNorm}}$ need about two and $\Pi_{\mathsf{LayerNorm}}$ 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 Π_{ReLU} by changing the exact plaintext function executed inside. $\Pi_{\mathsf{MatMulReLU}}$ 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 $\Pi_{\mathsf{MatMulMaxPoolReLU}}$, $\Pi_{\mathsf{MatMulBatchNormReLU}}$ 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

Protocol 4 $[\![\mathrm{ReLU}(\mathbf{A}\times\mathbf{B})]\!]^L \leftarrow \Pi_{\mathsf{MatMulReLU}}([\![\mathbf{A}]\!]^L, [\![\mathbf{B}]\!])$: Multiply **A** and **B**, then output the shares of the ReLU of the results.

Input. $\{P_i\}$ have shares of $\mathbf{A} \in \mathbf{Z}_L^{a\times b}$ and $\mathbf{B} \in \mathbf{Z}_L^{b\times c}$.

Output. $\{P_i\}$ get shares of $[\![\mathbf{Z}]\!]^L = [\![\mathrm{ReLU}(\mathbf{A}\times\mathbf{B})]\!]^L$.

(1) P_1, P_2, and P_3 locally computes $[\![\hat{\mathbf{C}}]\!]_i^L = [\![\mathbf{A}]\!]_i^L \times [\![\mathbf{B}]\!]_i^L + [\![\mathbf{A}]\!]_i^L \times [\![\mathbf{B}]\!]_{i+1}^L + [\![\mathbf{A}]\!]_{i+1}^L \times [\![\mathbf{B}]\!]_i^L$.
Malicious: Parties instead perform $\Pi_{\mathsf{mal\text{-}arith\text{-}mult}}$ of [58] to ensure that the multiplications (before truncation) were performed faithfully by parties. In the end, the 2-out-of-3 sharing $[\![\hat{\mathbf{C}}]\!]$ is distributed.

(2) P_i calls H_i to execute $\Pi_{\mathsf{LTH.GenMaskShare}}(a\times c, L)$ to obtain the masks $[\![\mathbf{M}]\!]_i \in \mathbf{Z}_L^{a\times c}$, then compute $\hat{\mathbf{C}}'_i = [\![\hat{\mathbf{C}}]\!]_i^L + [\![\mathbf{M}]\!]_i$. P_{i-1} also calls H_{i-1} to perform $\Pi_{\mathsf{LTH.GenMaskShare}}(a\times c, L)$ to obtain $[\![\mathbf{M}]\!]_{i-1} \in \mathbf{Z}_L^{a\times c}$ and $\hat{\mathbf{C}}'_{i-1} = [\![\hat{\mathbf{C}}]\!]_{i-1}^L + [\![\mathbf{M}]\!]_{i-1}$.
Malicious: Instead of doing the semi-honest protocol, P_{i+1}, P_{i-1} generates $[\![\mathbf{M}]\!]_{i+1} \leftarrow \Pi_{\mathsf{LTH.GenMask}}(a\times c, L, i+1)$ and $\hat{\mathbf{C}}'_{i+1} = [\![\hat{\mathbf{C}}']\!]_{i+1}^L + [\![\mathbf{M}]\!]_{i+1}$. P_{i-1}, P_i also generates $[\![\mathbf{M}]\!]_{i-1}$ invoking $\Pi'_{\mathsf{LTH.GenMask}}(a\times c, L, i-1)$, $\hat{\mathbf{C}}'_{i-1} = [\![\hat{\mathbf{C}}']\!]_{i-1}^L + [\![\mathbf{M}]\!]_{i-1}$.

(3) P_i and P_{i-1} send $\hat{\mathbf{C}}'_i$ and $\hat{\mathbf{C}}'_{i-1}$to P_{i+1}.
Malicious: Instead of doing the semi-honest protocol, P_{i+1}, P_{i-1} send $\hat{\mathbf{C}}'_{i+1}$ to P_i; P_{i-1}, P_i send $\mathbf{C}'_{i-1}$ to P_{i+1}.

(4) P_{i+1} computes $\hat{\mathbf{C}}' = \hat{\mathbf{C}}'_i + \hat{\mathbf{C}}'_{i-1} + [\![\hat{\mathbf{C}}']\!]_{i+1}^L$ and passes it to H_{i+1}.
Malicious: Instead of doing the semi-honest protocol, P_i, P_{i+1} compare the two received copies and abort if inconsistency is found. P_i computes $\hat{\mathbf{C}}' = \hat{\mathbf{C}}'_{i+1} + [\![\hat{\mathbf{C}}]\!]_{i-1}^L + [\![\hat{\mathbf{C}}]\!]_i^L$ and passes it to H_i, P_{i+1} computes $\hat{\mathbf{C}}' = \hat{\mathbf{C}}'_{i-1} + [\![\hat{\mathbf{C}}]\!]_i^L + [\![\hat{\mathbf{C}}]\!]_{i+1}^L$ and passes it to H_{i+1}.

(5) **LTH Only** : H_{i+1} generates the masks $[\![\mathbf{M}]\!]_{i+1}$ by invoking $\Pi_{\mathsf{LTH.GenMaskShare}}(a\times c, L)$ and recovers the plaintext through truncation: $\hat{\mathbf{C}} = \lfloor(\hat{\mathbf{C}}' + [\![\mathbf{M}]\!]_{i+1})\rfloor \gg \mathrm{fp}$. Note that $[\![\mathbf{M}]\!]_{i+1} + [\![\mathbf{M}]\!]_i + [\![\mathbf{M}]\!]_{i-1} = \mathbf{0}$.
Set $\mathbf{D} = (\hat{\mathbf{C}} > \mathbf{0})$. Then H_{i+1} invokes $\Pi_{\mathsf{LTH.GenMaskShare}}(a\times c, L)$ to get $[\![\mathbf{Z}^*]\!]_i^L \in \mathbb{Z}_L^{a\times c}$, $[\![\mathbf{Z}^*]\!]_{i+1}^L \in \mathbb{Z}_L^{a\times c}$, and compute:
$([\![Z_{j,k}]\!]_i^L, [\![Z_{j,k}]\!]_{i+1}^L) = ((D_{j,k}?C_{j,k}:0) + [\![Z^*_{j,k}]\!]_i^L, [\![Z^*_{j,k}]\!]_{i+1}^L)$. Return them to P_{i+1}.
P_i and P_{i-1} call H_i and H_{i-1} to invoke $\Pi_{\mathsf{LTH.GenMaskShare}}(a\times c, L)$ to get $[\![\mathbf{Z}]\!]_{i-1}^L \in \mathbb{Z}_L^{a\times c}$.
Malicious: Instead, H_{i+1} generates masks with $\Pi_{\mathsf{LTH.GenMask}}(a\times c, L, i-1)$ to recover the plaintext $\hat{\mathbf{C}} = \lfloor(\hat{\mathbf{C}}' - [\![\mathbf{M}]\!]_{i-1})\rfloor \gg \mathrm{fp}$, with the remaining being the same. H_i does as above respectively with index i replacing index $i+1$, replacing $\Pi_{\mathsf{LTH.GenMask}}$ with $\Pi'_{\mathsf{LTH.GenMask}}$, and fixes that the plaintext results are added to mask share i. $\Pi_{\mathsf{LTH.GenMaskShare}}(a\times c, L)$ is invoked at last to generate the shares. H_{i-1} also generates the remaining share for P_{i-1}.

(6) P_{i+1} send $[\![\mathbf{Z}]\!]_i^L, [\![\mathbf{D}]\!]_i^L$ to P_i, send $[\![\mathbf{Z}]\!]_{i+1}^L, [\![\mathbf{D}]\!]_{i+1}^L$ to P_{i-1}. Now, $[\![\mathbf{Z}]\!]^L$ and $[\![\mathbf{D}]\!]^L$ are calculated and shared with each party.
Malicious: P_i shares $[\![\mathbf{Z}]\!]_{i-1}^L, [\![\mathbf{D}]\!]_{i-1}^L$ and $[\![\mathbf{Z}]\!]_i^L, [\![\mathbf{D}]\!]_i^L$ to P_{i-1} and P_{i+1} respectively. Each party checks the results from the two parallel computations and aborts if an inconsistency is found.

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 $\mathbf{x}$ is defined as follows:

$$\mathrm{Softmax}(\mathbf{x}) := \exp(\mathbf{x}) / \sum_{i=1}^{n} \exp(\mathbf{x}_i) \quad (1)$$

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 value of the input vector $\mathbf{x}$ 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 Π_{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 with 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 $\exp([\![x]\!]_i^L)$, 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 Cortext-M3 that is also used in TPM, which represents today’s low-end security processor.

Protocol 5 $\Pi_{\mathsf{Softmax}}(\mathbf{x})$ compute softmax

Input. $\{P_i\}$ have replicative shares of $\mathbf{x} \in \mathbb{Z}_L^n$.

Output. $\{P_i\}$ get $\llbracket \exp(\mathbf{x}) \rrbracket^L$

Initial values. The LTHs save (q_L, m_L) for later use, setting $(m_L \gg 52) = \lfloor \overline{\exp(L \gg \mathrm{fp}) \cdot 2^{-q_L}} \rfloor$ with fp $= 13$ is the fixed-point precision under $L = 2^{32}$, where we save $q_L \in \mathbb{Z}_{2^{32}}$, $m_L \in \mathbb{Z}_{2^{52}}$, so $(m_L \gg 52) \in [0, 1)$ and q_L will not overflow. We note $\bar{\mathbf{x}}$ to be the real values that $\mathbf{x}$ represents.

1. For each $\llbracket x_j \rrbracket^L$, P_i computes $\bar{r} = \overline{\exp(\llbracket x_j \rrbracket_{i-1}^L \gg \mathrm{fp})}$. Let $\bar{r} = \overline{\exp(\llbracket x_j \rrbracket_{i-1}^L \gg \mathrm{fp})} = 2^{q_j} \cdot (m_j \gg 52)$ where m_j has no sign, since it is always positive. (Notice that $|q_j| = |\lfloor \log_2(\overline{\exp(\llbracket x_j \rrbracket_{i-1}^L \gg \mathrm{fp})}) \rfloor| = \lfloor |\llbracket x_j \rrbracket_{i-1}^L \gg \mathrm{fp}| \cdot \log_2(e)) \rfloor < 2^{30}$, so 32 bits are enough to store q_j and support additions without overflow).
   Invoke $\Pi_{\mathsf{LTH.GenMask}}$ from H_i to generate two masks $\alpha_j \in \mathbb{Z}_{2^{52}}$ and $\beta_j \in \mathbb{Z}_{2^{32}}$ for $j = 1, \ldots, n$ with the corresponding dimensions, and send $\{m_j^* = (m_j + \alpha_j)_{2^{52}}, q_j^* = (q_j + \beta_j)_{2^{32}}\}$ for $i = 1, \ldots, n$ to P_{i+1}.

   Malicious: P_{i-1} follows the same computation to get $\{m_j^*, q_j^*\}$ to P_{i+1}. P_{i-1}, P_{i+1} additionally compute $'\bar{r} = \overline{\exp(\llbracket x_j \rrbracket_{i+1}^L \gg \mathrm{fp})}$ and obtain $\{'m_j^*, ' q_j^*\}$ by masking the mantissa and exponent part with masks from $\Pi'_{\mathsf{LTH.GenMask}}$, send them to P_i.
2. P_{i+1} receives $\{q_j^*, m_j^*\}$ and computes $2^{\hat{q}_j} \cdot \hat{m}_j := \overline{\exp((\llbracket x_j \rrbracket_{i+1}^L + \llbracket x_j \rrbracket_i^L)_L \gg \mathrm{fp})}$. Send $\{\mathbf{q}^* + \hat{\mathbf{q}}, \mathbf{m}^*, \hat{\mathbf{m}}\}$ to H_{i+1}.
   Malicious: P_{i+1} and P_i compare the two received copies and abort if an inconsistency is found. P_i do the same computation as above with index i replacing index $i-1$, and send the obtained $\{'\mathbf{q}^* +' \hat{\mathbf{q}}, ' \mathbf{m}^*, ' \hat{\mathbf{m}}\}$ to H_i.
3. **LTH Only**: H_{i+1} Generate α_j and β_j, Compute $q'_j = (q_j^* + \hat{q}_j) - \beta_j = q_j + \hat{q}_j$, $m'_j = (m_j^* - \alpha_j) \cdot \hat{m}_j = m_j \cdot \hat{m}_j$. Define the results
   $2^{\hat{q}'_j} \cdot m'_j := \overline{\exp(\llbracket x_j \rrbracket_i^L \gg \mathrm{fp})} \cdot \overline{\exp((\llbracket x_j \rrbracket_{i-1}^L + \llbracket x_j \rrbracket_{i+1}^L)_L \gg \mathrm{fp})} = \overline{\exp(((\llbracket x_j \rrbracket_{i-1}^L + \llbracket x_j \rrbracket_{i+1}^L)_L + \llbracket x_j \rrbracket_i^L) \gg \mathrm{fp})}$
   For the fixed-point representation $x_j \in [0, L)$, the real value it represents $\bar{x}_j \in [-L/2 \gg \mathrm{fp}, L/2 \gg \mathrm{fp})$,
   $\lceil \log_2(\exp(x)) \rceil \in [-(L/2) \gg \mathrm{fp} \cdot \log_2(e), (L/2) \gg \mathrm{fp} \cdot \log_2(e)]$. Set the q-bound: $\mathrm{qb} = ((L/2) \gg \mathrm{fp}) \cdot \log_2(e)$. Define
   $\overline{\exp(\bar{x}_j)} := 2^{q''_j} \cdot (m''_j \gg 52)$, then q''_j should be in $[-\mathrm{qb}, \mathrm{qb}]$.
   q'_j and m'_j may alter from the correct q''_j and m''_j for two possible reasons: We are missing an L to be subtracted if
   $(\llbracket x_j \rrbracket_{i-1}^L + \llbracket x_j \rrbracket_{i+1}^L)_L + \llbracket x_j \rrbracket_i^L \neq (\llbracket x_j \rrbracket_i^L + \llbracket x_j \rrbracket_{i+1}^L + \llbracket x_j \rrbracket_{i-1}^L)_L = x_j$; or $\bar{x}_j$, or the real value x_j represents is actually negative, so
   $\bar{x}_j = ((\llbracket x_j \rrbracket_i^L + \llbracket x_j \rrbracket_{i+1}^L + \llbracket x_j \rrbracket_{i-1}^L)_L - L) \gg \mathrm{fp}$. H_{i+1} runs:
   (a) If $q'_j \in (0, \mathrm{qb}]$, $q''_j = q'_j$, $m''_j = m'_j$.
   (b) If $q'_j \in (\mathrm{qb}, 3 * \mathrm{qb}]$, we are missing one $\exp(-L \gg \mathrm{fp})$ to be multiplied for either reason mentioned above. Compute $q''_j = q'_j - q_L$, $m''_j = (m_j^* - \alpha_j) \cdot \hat{m}_j / m_L$.
   (c) If $q'_j \in (3 * \mathrm{qb}, 5 * \mathrm{qb}]$, we are missing $\exp(-2L \gg \mathrm{fp})$ to be multiplied for both reasons. Compute $q''_j = q'_j - 2 * q_L$, $m''_j = (m_j^* - \alpha_j) \cdot \hat{m}_j / (m_L)^2$.
   Now H_{i+1} obtains the corrected $\exp(\bar{x}_j) = 2^{q''_j} (m''_j \gg 52)$. H_{i+1}.
   •Then compute softmax of the real values directly by $\mathrm{Softmax}(\bar{\mathbf{x}}) = \exp(\bar{\mathbf{x}}) / \sum(\exp(\bar{\mathbf{x}}))$ which involves only $O(n)$ additions, $O(1)$ multiplications and divisions. Then convert the results to fixed-point representations, and invoke $\Pi_{\mathsf{LTH.GenMaskShare}}$ for masks $\llbracket \mathbf{m} \rrbracket_{i+1}^L, \llbracket \mathbf{m} \rrbracket_i^L$ to output $(\llbracket \mathbf{y} \rrbracket_i^L, \llbracket \mathbf{y} \rrbracket_{i+1}^L) = (\mathrm{Softmax}(\mathbf{x}) + \llbracket \mathbf{m} \rrbracket_i^L, \llbracket \mathbf{m} \rrbracket_{i+1}^L)$ to P_{i+1}.
   P_i and P_{i-1} call H_i and H_{i-1} to generate $\llbracket \mathbf{m} \rrbracket_i^L = \Pi_{\mathsf{LTH.GenMaskShare}}(n)$ as $\llbracket \mathbf{y} \rrbracket_{i-1}^L$.
   Malicious: H_i does the computation accordingly, while also masking the results with the index masks i. P_{i-1} calls H_{i-1} to generate $(\llbracket \mathbf{m} \rrbracket_{i+1}^L, \llbracket \mathbf{m} \rrbracket_{i-1}^L) = \Pi_{\mathsf{LTH.GenMaskShare}}(n, L)$ as $(\llbracket \mathbf{y} \rrbracket_{i+1}^L, \llbracket \mathbf{y} \rrbracket_{i-1}^L)$.
4. P_{i+1} shares $\llbracket \mathbf{y} \rrbracket_i^L$ with P_i and $\llbracket \mathbf{y} \rrbracket_{i+1}^L$ with P_{i-1}
   Malicious: P_i shares $\llbracket \mathbf{y} \rrbracket_{i-1}^L$ to P_{i-1} and $\llbracket \mathbf{y} \rrbracket_i^L$ to P_{i+1} respectively. Each party checks the results of the two parallel computations and aborts if an inconsistency is found.

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 Π_{softmax} using the exponentiation protocol from [42] combined with Falcon's Π_{Div} and Π_{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.

	Network-A				Network-B				Network-C			
Framework	LAN GPU	LAN CPU	WAN GPU	WAN CPU	LAN GPU	LAN CPU	WAN GPU	WAN CPU	LAN GPU	LAN CPU	WAN GPU	WAN CPU
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	-	5.589	-
Full SGX	0.462				0.461				0.461			

	LeNet				AlexNet				Transformer			
Framework	LAN GPU	LAN CPU	WAN GPU	WAN CPU	LAN GPU	LAN CPU	WAN GPU	WAN CPU	LAN GPU	LAN CPU	WAN GPU	WAN CPU
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	8.215	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				0.563			

	VGG16				ResNet18				Word2Vec			
Framework	LAN GPU	LAN CPU	WAN GPU	WAN CPU	LAN GPU	LAN CPU	WAN GPU	WAN CPU	LAN GPU	LAN CPU	WAN GPU	WAN CPU
Falcon+	49.36	-	122.1	-	545.9	-	1439		1.631	8.687	81.93	90.82
AriaNN	198.4	-	-	-	1779(8)	-	-	-	-	-	-	-
CryptGPU	11.31(32)	-	55.43(32)	-	45.20(8)	-	600.3(8)	-	1.469	-	47.19	-
Stamp-chip	**26.55**	-	**30.05**	-	**309.7**	-	**350.7**	-	**0.628**	**7.556**	**1.727**	**8.869**
speed-up	1.85×	-	4.06×	-	1.76×	-	4.10×	-	2.33×	1.14×	47.4×	10.2×
Stamp-SoC	13.06	-	16.56	-	106.9	-	148.0	-	0.593	7.521	1.692	8.835
speed-up	3.78×	-	7.37×	-	5.10×	-	9.72×	-	2.74×	1.15×	27.32×	10.2×
Goten	6.208	-	25.71	-	-	-	-	-	-	-	-	-
Full SGX	32.422				8.156				2.23			

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.

performance overhead of a TEE. On the other hand, MPC is generally considered to be more secure compared to a high-performance TEE such as Intel SGX. As discussed in §3.3, we believe that LTH is easier to protect and deploy compared to high-performance TEEs.

When compared to Goten, which also relies on MPC for secure outsourcing of linear layers, the experimental results suggest that Stamp is faster in most cases even though LTH has much lower performance compared to Intel SGX. Goten is relatively slow for small networks, again partialy due to the SGX initialization overhead. For VGG16, Goten outperforms Stamp with a discrete security chip (LTH-chip), mainly due to the large performance gap between the low-end LTH and a high-performance TEE (SGX) used by Goten. However, Stamp outperforms Goten in the WAN/GPU setting when running on a more powerful LTH (LTH-SoC), which has high local communication bandwidth and runs at a higher clock frequency.[1] These results confirm the main intuition behind the Stamp design, that small high-security hardware can be sufficient when primarily used to perform non-linear operations.

In Figure 4, we show the time breakdown of semi-honest inference over AlexNet in the WAN / GPU setting. In both Stamp and Falcon+, linear layers take a similar amount of time. However, they contribute only 5.1% of the execution time in Falcon+, and over 36% in Stamp, because the non-linear layers' runtime is significantly reduced from about 94.9% to 18.6% (63.6% if we roughly consider all operations on LTH are related to non-linear layers. There is some overhead, such as the local transmission in step 4 of Protocol 4, which cannot be assigned to be only linear or non-linear operations). Table 5 shows another breakdown of the execution time in the WAN/GPU setting: CPU/GPU, communication, and LTH. Note

[1] We 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.

Table 2: Latency (s) of running the entire batch of size 128 in a malicious setting.

	Network-A				Network-B				Network-C			
Framework	LAN GPU	LAN CPU	WAN GPU	WAN CPU	LAN GPU	LAN CPU	WAN GPU	WAN CPU	LAN GPU	LAN CPU	WAN GPU	WAN CPU
Falcon+	0.1921	0.3293	3.359	3.3701	0.6279	0.6416	3.504	3.0743	5.639	6.758	23.26	20.8988
Stamp-chip	**0.0913**	**0.2567**	**0.5150**	**0.7307**	**0.1594**	**0.2860**	**0.5803**	**0.7246**	**1.157**	**3.661**	**2.196**	**4.258**
speed-up	2.10×	1.51×	6.52×	4.61×	3.94×	2.82×	6.04×	4.24×	4.87×	2.12×	10.6×	4.91×
Stamp-SoC	0.0200	0.1854	0.4437	0.6594	0.0357	0.1623	0.4566	0.6010	0.0325	2.536	1.071	3.134
speed-up	9.59×	2.09×	7.57×	5.11×	17.5×	4.98×	7.67×	5.12×	173×	3.06×	21.7×	6.67×

	LeNet				AlexNet				Transformer			
Framework	LAN GPU	LAN CPU	WAN GPU	WAN CPU	LAN GPU	LAN CPU	WAN GPU	WAN CPU	LAN GPU	LAN CPU	WAN GPU	WAN CPU
Falcon+	7.492	15.38	28.82	32.57	13.89	41.88	100.9	123.2	10.99	56.10	777.9	821.3
Stamp-chip	**2.106**	**10.64**	**2.929**	**10.80**	**3.653**	**36.80**	**5.538**	**36.10**	**2.073**	**47.37**	**13.03**	**59.30**
speed-up	3.56×	1.55×	9.84×	3.01×	3.80×	1.27×	18.2×	3.41×	5.30×	-	59.6×	-
Stamp-SoC	0.5136	9.052	1.337	9.213	0.7738	33.92	2.659	33.22	1.720	47.02	12.68	58.95
speed-up	14.5×	1.82×	21.5×	3.54×	17.9×	1.37×	37.9×	3.71×	6.39×	1.19×	61.3×	13.9×

	VGG16				ResNet18				Word2Vec			
Framework	LAN GPU	LAN CPU	WAN GPU	WAN CPU	LAN GPU	LAN CPU	WAN GPU	WAN CPU	LAN GPU	LAN CPU	WAN GPU	WAN CPU
Falcon+	136.9	-	407.8	-	1550	-	4993	-	3.985	32.80	202.5	231.2
Stamp-chip	**51.54**	-	**68.09**	-	**639.3**	-	**772.2**	-	**1.327**	**30.44**	**2.209**	**31.44**
speed-up	2.66×	-	5.99×	-	2.43×	-	6.47×	-	3.00×	1.08×	91.6×	7.35×
Stamp-SoC	20.06	-	36.62	-	166.2	-	299.1	-	1.258	30.37	2.140	31.37
speed-up	6.82×	-	11.1×	-	9.33×	-	16.6×	-	3.17×	1.08×	94.6×	7.37×

Table 3: Communication (MB) for the entire batch of size 128 in a semi-honest setting. Brackets indicate an altered batch size.

Framework		Network-A	Network-B	Network-C	LeNet	AlexNet	Transformer	VGG16	ResNet18	Word2Vec
Falcon+	Inter-party	1.536	6.272	64.87	95.33	173.5	72.21	1730	22933	12.61
AriaNN	Inter-party	2.816	-	38.54	55.04	121.6	-	1161	18944	-
CryptGPU	Inter-party	3.012	9.911	33.35	122.7	99.05	-	1714(32)	2729(8)	95.46
Stamp	Inter-party	0.2058	0.8371	5.328	7.931	12.31	19.21	187.7	2106	0.4624
	LTH-CPU	0.4585	0.7958	7.235	10.24	18.53	2.270	202.5	3044	0.412

Table 4: Inference communication (MB) of the entire batch of size 128 in a malicious setting.

Framework		Network-A	Network-B	Network-C	LeNet	AlexNet	Transformer	VGG16	ResNet18	Word2Vec
Falcon+	Inter-party	10.51	41.33	423.4	620.1	1135	340.0	11543	139287	99.49
Stamp	Inter-party	0.8443	2.0704	21.02	28.80	48.68	131.2	838.6	7500	25.08
	LTH-CPU	1.070	1.856	16.88	23.90	43.23	5.296	472.5	7103	0.826

Table 5: Inference time (s) breakdown of a batch of size 128 in a semi-honest WAN/GPU setting, comparing with Falcon+.

Framework	Component	Network-A		Network-B		Network-C		LeNet		AlexNet		Transformer		VGG16		ResNet18		Word2Vec	
		Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio
Stamp	CPU/GPU	0.04	17%	0.06	23%	0.19	19%	0.28	21%	0.29	12%	0.30	6%	12.8	43%	98.0	28%	0.47	27%
	Comm.	0.18	70%	0.16	58%	0.35	34%	0.35	27%	0.94	38%	4.56	91%	3.34	11%	44.0	13%	1.22	71%
	LTH	0.03	13%	0.05	19%	0.48	47%	0.68	52%	1.24	50%	0.16	3%	13.8	46%	207	59%	0.03	2%
Falcon+	CPU/GPU	0.08	6%	0.20	14%	1.80	25%	2.50	28%	4.09	11%	3.95	1%	47.6	39%	523	36%	1.55	2%
	Comm.	1.40	94%	1.27	86%	5.52	75%	6.37	72%	34.4	89%	317	99%	74.4	61%	916	64%	89.3	99%

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 Softmax in the final layer of Word2Vec to demonstrate its impact). 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 relying 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.

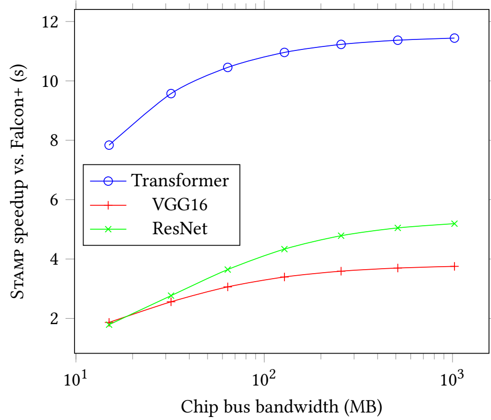


Figure 5: The speedup with different bus bandwidth in the semi-honest setting under LAN/GPU.

STAMP trades off the inter-party communication with the local communication between LTH and the untrusted CPU/GPU. However, as shown in Table 3 and Table 4, for LTH-chip, the low (15 MBps) bus bandwidth becomes a bottleneck of our performance with large networks, especially in the LAN setting. 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.

5.3 Accuracy

The precision of the inference using the plaintext computation and model weights is shown in Table 6, where the model weights are from the plaintext training. For Network-A, B, C and LeNet, which are measured by Falcon [89], STAMP has the same accuracy that Falcon achieved. STAMP optimizes overhead, but does not change computation precision with the same L and fp. In most schemes, for each batch of 128 elements, only one more sample would be classified incorrectly compared with the plaintext results, mainly due to the quantization when converting data from floating-point precision to fixed-point precision. However if some weights and activations are outside of the fix-point representation range, the accuracy may degrade more significantly. Carefully capping the values during training can potentially help avoid this issue.

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 $0.40mm^2$ area for the ARM SecurCore SC300. The microcontroller has a peak power consumption of $12mW$.

Table 6: Accuracy on different networks. The weights of ResNet18 are from Torchvision [52].

Network	Plaintext Accuracy	STAMP Accuracy
Network-A	98.18%	97.42%
Network-B	98.93%	97.81%
Network-C	99.16%	98.64%
LeNet	99.76%	99.15%
ResNet18@1	84.76%	84.37%
ResNet18@5	95.80%	95.50%

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

The combined overhead of $0.53mm^2$ 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: $322mm^2 \sim 698mm^2$, Intel Sapphire Rapids: $\sim 400mm^2$) is much larger than the size of LTH ($0.53mm^2$).

For the software TCB, our LTH software implementation has ~13k LoC. On the other hand, the software TCB for the Intel SGX experiment includes Gramine (~50k LoC) and PyTorch (~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 GPU 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.

Trusted hardware-based privacy-preserving machine learning. The previous work investigated performing and optimizing machine learning computation inside a CPU TEE (Intel SGX) [41], and providing stronger side-channel protection through data-oblivious computation [62]. The performance of a TEE can be further improved by introducing the TEE capabilities to GPUs [36, 87] and domain-specific accelerators [30]. While TEE on a high-performance CPU/GPU/accelerator is capable of providing much higher performance compared to MPC-based machine learning computation, the approach comes with the challenges in securing complex high-performance hardware as well as the cost of developing and deploying new hardware and software. STAMP is the first to combine a small security processor with MPC for privacy-preserving machine learning, introducing a new security and performance trade-off.

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 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.

REFERENCES

[1] Martin Abadi, Andy Chu, Ian Goodfellow, H Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. 2016. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 308–318.

[2] Raad Bahmani, Manuel Barbosa, Ferdinand Brasser, Bernardo Portela, Ahmad-Reza Sadeghi, Guillaume Scerri, and Bogdan Warinschi. 2017. Secure multiparty computation from SGX. In *International Conference on Financial Cryptography and Data Security*. Springer, 477–497.

[3] Alexander Sprogø Banks, Marek Kisiel, and Philip Korsholm. 2021. Remote attestation: a literature review. *arXiv preprint arXiv:2105.02466* (2021).

[4] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software grand exposure:SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*.

[5] Brandon Broadnax, Alexander Koch, Jeremias Mechler, Tobias Müller, Jörn Müller-Quade, and Matthias Nagel. 2021. Fortified Multi-Party Computation: Taking Advantage of Simple Secure Hardware Modules. *Proceedings on Privacy Enhancing Technologies* 2021, 4 (2021), 312–338.

[6] John Butterworth, Corey Kallenberg, Xeno Kovah, and Amy Herzog. 2013. Bios chronomancy: Fixing the core root of trust for measurement. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications Security*. 25–36.

[7] Ran Canetti. 2000. Security and composition of multiparty cryptographic protocols. *Journal of CRYPTOLOGY* 13, 1 (2000), 143–202.

[8] Ran Canetti. 2001. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE, 136–145.

[9] Nicolas Carlini, Jamie Hayes, Milad Nasr, Matthew Jagielski, Vikash Sehwag, Florian Tramer, Borja Balle, Daphne Ippolito, and Eric Wallace. 2023. Extracting training data from diffusion models. In *32nd USENIX Security Symposium (USENIX Security 23)*. 5253–5270.

[10] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. 2017. EzPC: programmable, efficient, and scalable secure two-party computation for machine learning. *Cryptology ePrint Archive* (2017).

[11] Harsh Chaudhari, Ashish Choudhury, Arpita Patra, and Ajith Suresh. 2019. Astra: High throughput 3pc over rings with application to secure prediction. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*. 81–92.

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

[13] Microsoft Corporation. 2023. Azure Machine Learning Studio. https://azure.microsoft.com/en-us/services/machine-learning/.

[14] Victor Costan and Srinivas Devadas. 2016. Intel SGX explained. *Cryptology ePrint Archive* (2016).

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

[16] Li Deng. 2012. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine* 29, 6 (2012), 141–142.

[17] Pham-Khoi Dong, Hung K Nguyen, and Xuan-Tu Tran. 2019. A 45nm high-throughput and low latency aes encryption for real-time applications. In *2019 19th International Symposium on Communications and Information Technologies (ISCIT)*. IEEE, 196–200.

[18] Joan G Dyer, Mark Lindemann, Ronald Perez, Reiner Sailer, Leendert Van Doorn, and Sean W Smith. 2001. Building the IBM 4758 secure coprocessor. *Computer* 34, 10 (2001), 57–66.

[19] Shufan Fei, Zheng Yan, Wenxiu Ding, and Haomeng Xie. 2021. Security vulnerabilities of SGX and countermeasures: A survey. *ACM Computing Surveys (CSUR)* 54, 6 (2021), 1–36.

[20] Susanne Felsen, Ágnes Kiss, Thomas Schneider, and Christian Weinert. 2019. Secure and private function evaluation with Intel SGX. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*. 165–181.

[21] Ben Fisch, Dhinakaran Vinayagamurthy, Dan Boneh, and Sergey Gorbunov. 2017. Iron: functional encryption using Intel SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 765–782.

[22] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. 2017. High-throughput secure three-party computation for malicious adversaries and an honest majority. In *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 225–255.

[23] Oded Goldreich, Silvio Micali, and Avi Wigderson. 2019. How to play any mental game, or a completeness theorem for protocols with honest majority. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*. 307–328.

[24] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*. 1–6.

[25] Debayan Gupta, Benjamin Mood, Joan Feigenbaum, Kevin Butler, and Patrick Traynor. 2016. Using intel software guard extensions for efficient two-party secure function evaluation. In *International Conference on Financial Cryptography and Data Security*. Springer, 302–318.

[26] Jago Gyselinck, Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2018. Off-limits: Abusing legacy x86 memory segmentation to spy on enclaved execution. In *International Symposium on Engineering Secure Software and Systems*. Springer, 44–60.

[27] Seunghun Han, Wook Shin, Jun-Hyeok Park, and HyoungChun Kim. 2018. A bad dream: Subverting trusted platform module while you are sleeping. In *27th USENIX Security Symposium (USENIX Security 18)*. 1229–1246.

[28] Hanieh Hashemi, Yongqin Wang, and Murali Annavaram. 2020. Darknight: A data privacy scheme for training and inference of deep neural networks. *arXiv preprint arXiv:2006.01300* (2020).

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

[30] Weizhe Hua, Muhammad Umar, Zhiru Zhang, and G Edward Suh. 2020. Guardnn: Secure dnn accelerator for privacy-preserving deep learning. *arXiv preprint arXiv:2008.11632* (2020).

[31] Apple Inc. 2022. *Apple Platform Security*. https://help.apple.com/pdf/security/en_GB/apple-platform-security-guide-b.pdf

[32] Apple Inc. 2023. *Apple Secure Enclave*. https://support.apple.com/guide/security/secure-enclave-sec59b0b31ff/web

[33] Google Inc. 2023. *Google Titan Key*. https://cloud.google.com/titan-security-key/

[34] Qualcomm Incorporated. 2019. *Qualcomm Secure Processing Unit*. https://www.qualcomm.com/news/releases/2019/06/qualcomm-snapdragon-855-becomes-first-mobile-soc-receive-smart-card

[35] Rambus Incorporated. 2023. *RT-630 Programmable Root of Trust*. https://www.rambus.com/security/root-of-trust/rt-630/

[36] Insu Jang, Adrian Tang, Taehoon Kim, Simha Sethumadhavan, and Jaehyuk Huh. 2019. Heterogeneous isolated execution for commodity gpus. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 455–468.

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

[38] William Kahan. 1996. IEEE standard 754 for binary floating-point arithmetic. *Lecture Notes on the Status of IEEE* 754, 94720-1776 (1996), 11.

[39] Georgios Kaissis, Alexander Ziller, Jonathan Passerat-Palmbach, Théo Ryffel, Dmitrii Usynin, Andrew Trask, Ionésio Lima, Jason Mancuso, Friederike Jungmann, Marc-Matthias Steinborn, et al. 2021. End-to-end privacy preserving deep learning on multi-institutional medical imaging. *Nature Machine Intelligence* 3, 6 (2021), 473–484.

[40] Jonathan Katz. 2007. Universally composable multi-party computation using tamper-proof hardware. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 115–128.

[41] Kyungtae Kim, Chung Hwan Kim, Junghwan" John" Rhee, Xiao Yu, Haifeng Chen, Dave Tian, and Byoungyoung Lee. 2020. Vessels: Efficient and scalable deep learning prediction on trusted processors. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 462–476.

[42] B. Knott, S. Venkataraman, A.Y. Hannun, S. Sengupta, M. Ibrahim, and L.J.P. van der Maaten. 2021. CrypTen: Secure Multi-Party Computation Meets Machine Learning. In *arXiv 2109.00984*.

[43] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. 2014. The CIFAR-10 dataset. *online: http://www.cs.toronto.edu/kriz/cifar.html* 55, 5 (2014).

[44] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* 25 (2012).

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

[46] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. 2020. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[47] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. 2017. Hacking in darkness: Return-oriented programming against secure enclaves. In *26th USENIX Security Symposium (USENIX Security 17)*. 523–539.

[48] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium (USENIX Security 17)*. 557–574.

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

[50] Google LLC. 2023. Google Cloud AI. https://cloud.google.com/products/.

[51] Yibiao Lu, Bingsheng Zhang, Hong-Sheng Zhou, Weiran Liu, Lei Zhang, and Kui Ren. 2021. Correlated Randomness Teleportation via Semi-trusted Hardware—Enabling Silent Multi-party Computation. In *European Symposium on Research in Computer Security*. Springer, 699–720.

[52] Sébastien Marcel and Yann Rodriguez. 2010. Torchvision the machine-vision package of torch. In *Proceedings of the 18th ACM international conference on Multimedia*. 1485–1488.

[53] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843* (2016).

[54] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[55] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. 2020. Delphi: A cryptographic inference service for neural networks. In *29th USENIX Security Symposium (USENIX Security 20)*. 2505–2522.

[56] Samer Moein, T Aaron Gulliver, Fayez Gebali, and Abdulrahman Alkandari. 2017. Hardware attack mitigation techniques analysis. *International Journal on Cryptography and Information Security (IJCIS)* 7, 1 (2017), 9–28.

[57] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. 2020. TPM-FAIL : TPM meets Timing and Lattice Attacks. In *29th USENIX Security Symposium (USENIX Security 20)*. 2057–2073.

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

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

[60] Lucien KL Ng and Sherman SM Chow. 2021. GForce : GPU-Friendly Oblivious and Rapid Neural Network Inference. In *30th USENIX Security Symposium (USENIX Security 21)*. 2147–2164.

[61] Lucien KL Ng, Sherman SM Chow, Anna PY Woo, Donald PH Wong, and Yongjun Zhao. 2021. Goten: Gpu-outsourcing trusted execution of neural network training. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 14876–14883.

[62] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious Multi-Party Machine Learning on Trusted Processors. In *25th USENIX Security Symposium (USENIX Security 16)*. 619–636.

[63] Arpita Patra and Ajith Suresh. 2020. BLAZE: blazing fast privacy-preserving machine learning. *arXiv preprint arXiv:2005.09042* (2020).

[64] Siani Pearson and Boris Balacheff. 2003. *Trusted computing platforms: TCPA technology in context*. Prentice Hall Professional.

[65] Shahed E Quadir, Junlin Chen, Domenic Forte, Navid Asadizanjani, Sina Shahbazmohamadi, Lei Wang, John Chandy, and Mark Tehranipoor. 2016. A survey on chip to system reverse engineering. *ACM journal on emerging technologies in computing systems (JETC)* 13, 1 (2016), 1–34.

[66] Prashanthi Ramachandran, Shivam Agarwal, Arup Mondal, Aastha Shah, and Debayan Gupta. 2021. S++: A fast and deployable secure-computation framework for privacy-preserving neural network training. *arXiv preprint arXiv:2101.12078* (2021).

[67] Wolfgang Rankl and Wolfgang Effing. 2004. *Smart card handbook*. John Wiley & Sons.

[68] M Sadegh Riazi, Mohammad Samragh, Hao Chen, Kim Laine, Kristin Lauter, and Farinaz Koushanfar. 2019. XONN:XNOR-based Oblivious Deep Neural Network Inference. In *28th USENIX Security Symposium (USENIX Security 19)*. 1501–1518.

[69] M Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M Songhori, Thomas Schneider, and Farinaz Koushanfar. 2018. Chameleon: A hybrid secure computation framework for machine learning applications. In *Proceedings of the 2018 on Asia conference on computer and communications security*. 707–721.

[70] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)* 115, 3 (2015), 211–252. https://doi.org/10.1007/s11263-015-0816-y

[71] Théo Ryffel, Pierre Tholoniat, David Pointcheval, and Francis Bach. 2020. Ariann: Low-interaction privacy-preserving deep learning via function secret sharing. *Proceedings on Privacy Enhancing Technologies* 2022, 1 (2020), 291–316.

[72] AMD SEV-SNP. 2020. Strengthening VM isolation with integrity protection and more. *White Paper, January* (2020).

[73] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. 2015. Preventing your faults from telling your secrets: Defenses against pigeonhole attacks. *arXiv preprint arXiv:1506.04832* (2015).

[74] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[75] Sergei Skorobogatov. 2017. How microprobing can attack encrypted memory. In *2017 Euromicro Conference on Digital System Design (DSD)*. IEEE, 244–251.

[76] Sergei P Skorobogatov and Ross J Anderson. 2003. Optical fault induction attacks. In *Cryptographic Hardware and Embedded Systems-CHES 2002: 4th International Workshop Redwood Shores, CA, USA, August 13–15, 2002 Revised Papers 4*. Springer, 2–12.

[77] Evan R Sparks and Evan R Sparks. 2007. A security assessment of trusted platform modules computer science technical report TR2007-597. *Dept. Comput. Sci., Dartmouth College, Hanover, NH, USA, Tech. Rep., TR2007-597* (2007).

[78] STMicroelectronics. 2013. Trusted Platform Module with SPI based on 32-bit ARM® SecurCore® SC300™ CPU. https://datasheet.octopart.com/ST33TPM12SPI-STMicroelectronics-datasheet-62334860.pdf.

[79] Alin Suciu and Tudor Carean. 2010. Benchmarking the true random number generator of TPM chips. *arXiv preprint arXiv:1008.2223* (2010).

[80] Inc. Synopsys. 2023. *Synopsys tRoot Hardware Secure Modules*. https://www.synopsys.com/designware-ip/security-ip/root-of-trust.html

[81] Sijun Tan, Brian Knott, Yuan Tian, and David J Wu. 2021. CryptGPU: Fast privacy-preserving machine learning on the GPU. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1021–1038.

[82] Florian Tramer and Dan Boneh. 2018. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. *arXiv preprint arXiv:1806.03287* (2018).

[83] Florian Tramèr, Reza Shokri, Ayrton San Joaquin, Hoang Le, Matthew Jagielski, Sanghyun Hong, and Nicholas Carlini. 2022. Truth serum: Poisoning machine learning models to reveal their secrets. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2779–2792.

[84] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium (USENIX Security 18)*. 991–1008.

[85] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2017. SGX-Step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*. 1–6.

[86] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[87] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. 2018. Graviton: Trusted Execution Environments on GPUs. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 681–696.

[88] Sameer Wagh, Divya Gupta, and Nishanth Chandran. 2019. SecureNN: 3-Party Secure Computation for Neural Network Training. *Proc. Priv. Enhancing Technol.* 2019, 3 (2019), 26–49.

[89] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. 2020. Falcon: Honest-majority maliciously secure framework for private deep learning. *arXiv preprint arXiv:2004.02229* (2020).

[90] Jinwen Wang, Yueqiang Cheng, Qi Li, and Yong Jiang. 2018. Interface-based side channel attack against intel SGX. *arXiv preprint arXiv:1811.05378* (2018).

[91] Yongqin Wang, Edward Suh, Wenjie Xiong, Brian Knott, Benjamin Lefaudeux, Murali Annavaram, and Hsien-Hsin Lee. 2021. Characterizing and Improving MPC-based Private Inference for Transformer-based Models. In *NeurIPS 2021 Workshop Privacy in Machine Learning*.

[92] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Online, 38–45. https://www.aclweb.org/anthology/2020.emnlp-demos.6

[93] Pengfei Wu, Jianting Ning, Jiamin Shen, Hongbing Wang, and Ee-Chien Chang. 2022. Hybrid trust multi-party computation with trusted execution environment. In *The Network and Distributed System Security (NDSS) Symposium*.

[94] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 640–656.

[95] Jingkang Yang, Kaiyang Zhou, Yixuan Li, and Ziwei Liu. 2021. Generalized out-of-distribution detection: A survey. *arXiv preprint arXiv:2110.11334* (2021).

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

[97] Andrew Chi-Chih Yao. 1986. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*. IEEE, 162–167.

[98] Wenting Zheng, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. 2019. Helen: Maliciously secure coopetitive learning for linear models. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 724–738.

[99] Xing Zhou, Zhilei Xu, Cong Wang, and Mingyu Gao. 2022. PPMLAC: high performance chipset architecture for secure multi-party computation. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 87–101.

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 offline phase. The 3-out-of-3 randomness is defined as each P_i holding a share of 0: $\alpha = [\![\alpha]\!]_1^L + [\![\alpha]\!]_2^L + [\![\alpha]\!]_3^L$ where $\alpha = 0$ and P_i holds $[\![\alpha]\!]_i^L$. 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 k_i 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 $[\![\alpha]\!]_i^L$ as $[\![\alpha]\!]_i^L = \hat{F}_{k_i}(\mathrm{ctr}) - \hat{F}_{k_i}(\mathrm{ctr})$ with increments of ctr each time this process is invoked.

Input Phase. To construct $[\![x]\!]^L$ from the generated 3-out-of-3 randomness and $\bar{x}$ which is provided by P_i, P_i compute $[\![x]\!]_i^L = x + [\![\alpha]\!]_i^L$ and share it with P_{i-1}. P_{i-1} will send $[\![\alpha]\!]_{i-1}^L$ to P_{i+1} and P_{i+1} will send $[\![\alpha]\!]_{i+1}^L$ to P_i. In an honest majority malicious setting, additionally P_{i+1} should compute $[\![\alpha]\!]_{i+1}^L + [\![\alpha]\!]_{i-1}^L$ and send to P_i, then P_i confirm $[\![\alpha]\!]_{i+1}^L + [\![\alpha]\!]_{i-1}^L = -[\![\alpha]\!]_i^L$, therefore P_{i_1} behaves honestly; P_i should compute $[\![\alpha]\!]_{i+1}^L + [\![\alpha]\!]_i^L$ and send to P_{i-1}, then P_{i-1} confirm by $[\![\alpha]\!]_{i+1}^L + [\![\alpha]\!]_i^L = -[\![\alpha]\!]_{i-1}^L$, therefore P_{i+1} behaves honestly. Since P_i is allowed to send an arbitrary input $\bar{x}$, 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 $[\![x]\!]^L$, $[\![y]\!]^L$, and the public scalar c, we have the following:

- $[\![x]\!]^L + c = ([\![x]\!]_1^L + c, [\![x]\!]_2^L, [\![x]\!]_3^L)$
- $c \cdot [\![x]\!]^L = (c \cdot [\![x]\!]_1^L, c \cdot [\![x]\!]_2^L, c \cdot [\![x]\!]_3^L)$
- $[\![x]\!]^L + [\![y]\!]^L = ([\![x]\!]_1^L + [\![y]\!]_1^L, [\![x]\!]_2^L + [\![y]\!]_2^L, [\![x]\!]_3^L + [\![y]\!]_3^L)$

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

Reconstruction. $x \leftarrow \Pi_{\mathrm{Reconst}}([\![x]\!]^L)$: To reconstruct the plaintext x from the shares $[\![x]\!]^L$, 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 Π_{ReLU}, the protocol to offload ReLU operations under MPC to trusted hardware. The steps we take are as follows: First, P_i invokes $\Pi_{\mathrm{LTH.GenMask}}$ to get the pseudo-random masks, and sends $[\![x]\!]_i$ 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 $\Pi_{\mathrm{LTH.GenMask}}$ using the same key and the counter with the same recorded number, and computes ReLU in plaintext, re-masks the result with $\Pi_{\mathrm{LTH.GenMaskShare}}$, 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, $\{\mathrm{ctr}_{1,2}^i, \mathrm{ctr}_{2,3}^i, \mathrm{ctr}_{3,1}^i, \mathrm{ctr}_s^i\}$ for $i = 1, 2, 3$ all increase by n, and $\{\hat{\mathrm{ctr}}_{1,2}^i, \hat{\mathrm{ctr}}_{2,3}^i, \hat{\mathrm{ctr}}_{3,1}^i, \hat{\mathrm{ctr}}_s^i\}$ 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

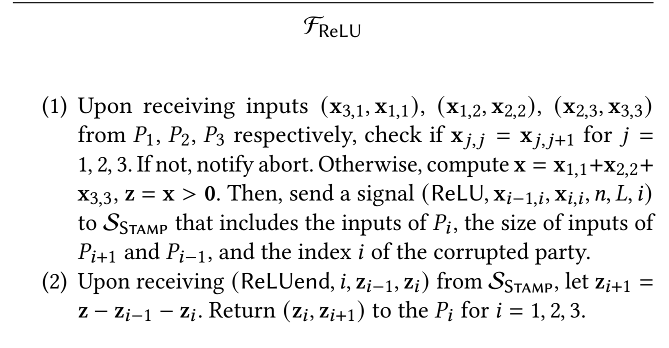
We claim the following three theorems hold:

Theorem 1. *Under the assumption of secure PRF and LTH, Π_{ReLU} (in Protocol 6) securely realizes the ideal functionality $\mathcal{F}_{ReLU}$ (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, $\Pi_{MatMalReLU}$ (in Protocol 4) securely realizes the ideal functionality $\mathcal{F}_{MatMulReLU}$ 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, $\Pi_{Softmax}$ (in Protocol 5) securely realizes the ideal functionality $\mathcal{F}_{Softmax}$ 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.

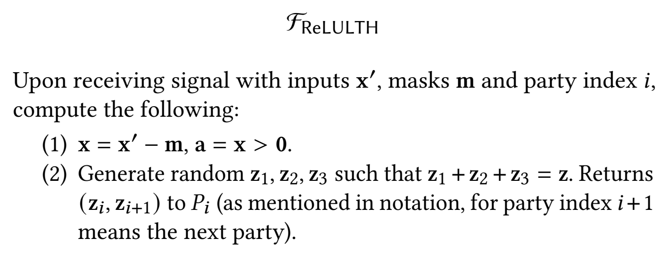


$\mathcal{F}_{\mathrm{ReLU}}$

(1) Upon receiving inputs $(\mathbf{x}_{3,1}, \mathbf{x}_{1,1})$, $(\mathbf{x}_{1,2}, \mathbf{x}_{2,2})$, $(\mathbf{x}_{2,3}, \mathbf{x}_{3,3})$ from P_1, P_2, P_3 respectively, check if $\mathbf{x}_{j,j} = \mathbf{x}_{j,j+1}$ for $j = 1, 2, 3$. If not, notify abort. Otherwise, compute $\mathbf{x} = \mathbf{x}_{1,1} + \mathbf{x}_{2,2} + \mathbf{x}_{3,3}$, $\mathbf{z} = \mathbf{x} > \mathbf{0}$. Then, send a signal $(\mathsf{ReLU}, \mathbf{x}_{i-1,i}, \mathbf{x}_{i,i}, n, L, i)$ to $\mathcal{S}_{\mathsf{STAMP}}$ 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 $(\mathsf{ReLUend}, i, \mathbf{z}_{i-1}, \mathbf{z}_i)$ from $\mathcal{S}_{\mathsf{STAMP}}$, let $\mathbf{z}_{i+1} = \mathbf{z} - \mathbf{z}_{i-1} - \mathbf{z}_i$. Return $(\mathbf{z}_i, \mathbf{z}_{i+1})$ to the P_i for $i = 1, 2, 3$.

Figure 6: Ideal functionality for $\Pi_{\mathbf{ReLU}}$.



$\mathcal{F}_{\mathrm{ReLULTH}}$

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

(1) $\mathbf{x} = \mathbf{x}' - \mathbf{m}$, $\mathbf{a} = \mathbf{x} > \mathbf{0}$.

(2) Generate random $\mathbf{z}_1, \mathbf{z}_2, \mathbf{z}_3$ such that $\mathbf{z}_1 + \mathbf{z}_2 + \mathbf{z}_3 = \mathbf{z}$. Returns $(\mathbf{z}_i, \mathbf{z}_{i+1})$ to P_i (as mentioned in notation, for party index $i+1$ means the next party).

Figure 7: Ideal functionality for the LTH part of $\Pi_{\mathbf{ReLU}}$.

Protocol 6 $[\![\mathsf{ReLU}(\mathbf{x})]\!]^L \leftarrow \Pi_{\mathsf{ReLU}}([\![\mathbf{x}]\!]^L)$ Do ReLU on shares of vector $\mathbf{x}$

Input. Each $\{P_i\}$ owns $[\![\mathbf{x}]\!]^L$.
Output. Each $\{P_i\}$ gets $[\![\mathbf{z}]\!]^L = [\![\mathsf{ReLU}(\mathbf{x})]\!]^L$.

(1) P_i calls H_i to execute $\Pi_{\mathsf{LTH.GenMask}}(n, L, i-1)$ to obtain the masks $\mathbf{m}_{i-1} \in \mathbb{Z}_L^n$ then compute $\mathbf{x}'_{i-1} = [\![\mathbf{x}]\!]_{i-1}^L + \mathbf{m}_{i-1}$.
Malicious: P_{i-1} generates $\mathbf{m}_{i-1} = \Pi_{\mathsf{LTH.GenMask}}(n, L, i-1)$ and $\mathbf{x}'_{i-1} = [\![\mathbf{x}]\!]_{i-1}^L + \mathbf{m}_{i-1}$, and P_{i-1}, P_{i+1} also generates $\mathbf{m}_{i+1} = \Pi_{\mathsf{LTH.GenMask}}(n, L, i+1)$, $\mathbf{x}'_{i+1} = [\![\mathbf{x}]\!]_{i+1}^L + \mathbf{m}_{i+1}$.
(2) P_i send $\mathbf{x}'_{i-1}$ to P_{i+1}.
Malicious: P_{i-1} send $\mathbf{x}'_{i-1}$ to P_{i+1}; P_{i-1} and P_{i+1} send $\mathbf{x}'_{i+1}$ to P_i.
(3) P_{i+1}, after receiving $\mathbf{x}'_{i-1}$ add it to $[\![\mathbf{x}]\!]_i^L, [\![\mathbf{x}]\!]_{i+1}^L$ and pass it to H_{i+1}.
Malicious: P_i, P_{i+1} check the two copies received and abort if any inconsistency is found. P_i execute as above with index $i-1$ replacing index i.
(4) **LTH Only** : H_{i+1} recovers the plaintext $\mathbf{x} = (\mathbf{x}'_{i-1} + [\![\mathbf{x}]\!]_i^L + [\![\mathbf{x}]\!]_{i+1}^L) - \mathbf{m}_{i-1}$ where $\mathbf{m}_{i-1}$ is generated with $\Pi_{\mathsf{LTH.GenMask}}(n, L, i-1)$. Set $\mathbf{b} = (\mathbf{x} > \mathbf{0})$. Then H_{i+1} invokes $\Pi_{\mathsf{LTH.GenMaskShare}}(n, L)$ to get $[\![\mathbf{z}^*]\!]_i^L, [\![\mathbf{z}^*]\!]_{i+1}^L \in \mathbb{Z}_L^n$, and compute: $([\![z_j]\!]_i^L, [\![z_j]\!]_{i+1}^L) = ((b_j?x_j : 0) + [\![z_j^*]\!]_i^L, [\![z_j^*]\!]_{i+1}^L)$. Return their values to P_{i+1}.
H_i and H_{i-1} invokes $\Pi_{\mathsf{LTH.GenMaskShare}}(n, L)$ to get $[\![\mathbf{z}]\!]_{i-1}^L \in \mathbb{Z}_L^n$, and invokes $\Pi_{\mathsf{LTH.GenMaskShare}}(n, 2)$ to get $[\![\mathbf{b}]\!]_{i-1}^L \in \mathbb{Z}_2^n$. Return them respectively to P_i and P_{i-1}.
Malicious: H_i perform the same step as above with index $i-1$ replacing index i, replacing $\Pi_{\mathsf{LTH.GenMaskShare}}$ with $\Pi'_{\mathsf{LTH.GenMaskShare}}$, while also masking the results with the masks of index i (i.e., $([\![z_j]\!]_{i-1}^L, [\![z_j]\!]_i^L) = ([\![z_j^*]\!]_{i-1}^L, (b_j?x_j : 0) + [\![z_j^*]\!]_i^L)$). H_{i-1} and H_{i+1} generate the remaining share $[\![z_j]\!]_{i+1}^L$ for P_{i-1} and P_{i+1} respectively.
(5) P_{i+1} send $[\![\mathbf{z}]\!]_i^L$ to P_i, send $[\![\mathbf{z}]\!]_{i+1}^L$ to P_{i-1}. Now $[\![\mathbf{z}]\!]^L$ and $[\![\mathbf{b}]\!]^L$ are calculated and shared to each party.
Malicious: P_i shares to P_{i-1} and P_{i+1} respectively. Each party checks the copies they receive and aborts if an inconsistency is found.

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

The Environment $\mathcal{E}$. The environment $\mathcal{E}$ provides inputs $(\mathbf{x}_{3,1}, \mathbf{x}_{1,1})$ to P_1, $(\mathbf{x}_{1,2}, \mathbf{x}_{2,2})$ to P_2, $(\mathbf{x}_{2,3}, \mathbf{x}_{3,3})$ to P_3, which are forwarded to the ideal functionality $\mathcal{F}_{\mathsf{ReLU}}$ in Figure 6. The environment $\mathcal{E}$ 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. $\mathcal{S}_{\mathsf{STAMP}}$ simulates the following interactions on receiving the signal from $\mathcal{F}_{\mathsf{ReLU}}$:

- Upon receiving $(\mathsf{ReLU}, 1, \mathbf{x}_{3,1}, \mathbf{x}_{1,1}, n, L)$ from $\mathcal{F}_{\mathsf{ReLU}}$, $\mathcal{S}_{\mathsf{STAMP}}$ generates a random $\hat{\mathbf{x}}_2$, and use $(\mathbf{x}_{1,1}, \hat{\mathbf{x}}_2)$ and $(\hat{\mathbf{x}}_2, \mathbf{x}_{3,1})$ as dummy inputs for P_2 and P_3, respectively.
- $\mathcal{S}_{\mathsf{STAMP}}$ acts as $\mathcal{F}_{\mathsf{GenMask}}$ to generate random masks $\mathbf{m}_3$ for P_1 and P_3, then computes $\mathbf{x}'_3 = \mathbf{x}_{3,1} + \mathbf{m}_3$ and sends $(\mathbf{m}_3, \mathbf{x}'_3)$ to P_1, computes $\mathbf{x}''_3 = \mathbf{x}_{3,1} + \mathbf{m}_3$ and sends $(\mathbf{m}_3, \mathbf{x}''_3)$ to P_3 then also sends $\tilde{\mathbf{x}}'_3$ (as adversary input for P_1) and $\mathbf{x}''_3$ to P_2 on behalf of P_1 and P_3, respectively.
$\mathcal{S}_{\mathsf{STAMP}}$ also acts as $\mathcal{F}_{\mathsf{GenMask}}$ to generate random masks $\mathbf{m}_2$ for P_3 and P_2, then computes $\mathbf{x}'_2 = \hat{\mathbf{x}}_2 + \mathbf{m}_2$ and sends $(\mathbf{m}_2, \mathbf{x}'_2)$ to P_3, computes $\mathbf{x}''_2 = \hat{\mathbf{x}}_2 + \mathbf{m}_2$ and sends $(\mathbf{m}_2, \mathbf{x}''_2)$ to P_2 then also sends $\mathbf{x}'_2$ and $\mathbf{x}''_2$ to P_1 on behalf of P_3 and P_2, respectively.
- $\mathcal{S}_{\mathsf{STAMP}}$ check $\mathbf{x}''_3 = \tilde{\mathbf{x}}'_3$, $\mathbf{x}''_2 = \mathbf{x}'_2$ on behalf of P_2 and P_1 respectively, signal abort to $\mathcal{F}_{\mathsf{ReLU}}$ if inconsistency found.
- $\mathcal{S}_{\mathsf{STAMP}}$ acts as $\mathcal{F}_{\mathsf{LTHReLU}}$ with P_2's input $\hat{\mathbf{x}} = \mathbf{x}'_3 + \hat{\mathbf{x}}_2 + \mathbf{x}_{1,1}$ to (re)generate $\mathbf{m}_3$, randoms $\mathbf{z}_1^*, \mathbf{z}_2^*, \mathbf{z}_3$ such that $\mathbf{z}_1^* + \mathbf{z}_2^* + \mathbf{z}_3 = 0$, and then compute $(\mathbf{z}_1, \mathbf{z}_2) = (\mathsf{ReLU}(\hat{\mathbf{x}} - \mathbf{m}_3) + \mathbf{z}_1^*, \mathbf{z}_2^*)$. $\mathcal{S}_{\mathsf{STAMP}}$ sends $(\mathbf{z}_1, \mathbf{z}_2)$ to P_2. $\mathcal{S}_{\mathsf{STAMP}}$ sends $\tilde{\mathbf{z}}_1$ to P_1 and sends $\mathbf{z}_2$ to P_3 (on behalf P_2).
$\mathcal{S}_{\mathsf{STAMP}}$ acts as $\mathcal{F}_{\mathsf{LTHReLU}}$ with P_1's input $\hat{\mathbf{x}}' = \mathbf{x}'_2 + \mathbf{x}_{3,1} + \mathbf{x}_{1,1}$ to (re)generate $\mathbf{m}_2$, randoms $\mathbf{z}_1^*, \mathbf{z}_2^*, \mathbf{z}_3$ such that $\mathbf{z}_1^* + \mathbf{z}_2^* + \mathbf{z}_3 = 0$, and then compute $(\mathbf{z}_3, \mathbf{z}'_1) = (\mathbf{z}_3, \mathsf{ReLU}(\hat{\mathbf{x}}' - \mathbf{m}_3) + \mathbf{z}_1^*)$. $\mathcal{S}_{\mathsf{STAMP}}$ sends $(\mathbf{z}_3, \mathbf{z}'_1)$ to P_1. $\mathcal{S}_{\mathsf{STAMP}}$ sends $\tilde{\mathbf{z}}_3$ to P_3 and sends $\tilde{\mathbf{z}}'_1$ to P_2 (as adversary inputs on behalf P_1).
- $\mathcal{S}_{\mathsf{STAMP}}$ acts as $\mathcal{F}_{\mathsf{GenMaskShr}}$ for P_1 to (re)generate random $\mathbf{z}_3$ and sends it to P_1. Similarly, $\mathcal{S}_{\mathsf{STAMP}}$ acts as $\mathcal{F}_{\mathsf{GenMaskShr}}$ for P_3 to (re)generate random as $\mathbf{z}_3$ and sends it to P_3. $\mathcal{S}_{\mathsf{STAMP}}$ also generate $\mathbf{z}_2^*$ for P_2 and P_3.
- $\mathcal{S}_{\mathsf{STAMP}}$ checks if $\mathbf{z}'_1 = \mathbf{z}_1$ received by P_1, also P_2; if same $\mathbf{z}_2^*$ received by P_2, also P_3; if same $\mathbf{z}_3$ received by P_3, also P_1. Signal abort to $\mathcal{F}_{\mathsf{ReLU}}$ if an inconsistency is found. If no abort is signaled, $\mathcal{S}_{\mathsf{STAMP}}$ signals $(\mathsf{ReLUend}, 1, \mathbf{z}_3, \mathbf{z}_1)$ to $\mathcal{F}_{\mathsf{ReLU}}$.

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.
Hybrid $\mathcal{H}_1$: $\mathcal{H}_1$ is the same as $\mathcal{H}_0$, except that $\Pi_{\mathsf{LTH.GenMask}}$ is replaced with simulated $\mathcal{F}_{\mathsf{GenMask}}$ that outputs random $\hat{\mathbf{m}}_3, \hat{\mathbf{m}}_2$ for both step 1) and 4).

We claim that $\mathcal{H}_0$ and $\mathcal{H}_1$ are computationally indistinguishable. This is because $\Pi_{\mathsf{LTH.GenMask}}$ generates pseudo-random $\mathbf{m}_3, \mathbf{m}_2$ 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 $\mathbf{m}_3, \mathbf{m}_2$ produced by LTH is computationally indistinguishable from the random $\hat{\mathbf{m}}_3, \hat{\mathbf{m}}_2$ generated by the simulator. Therefore, $\mathcal{H}_0$ and $\mathcal{H}_1$ are computationally indistinguishable.
Hybrid $\mathcal{H}_2$: $\mathcal{H}_2$ is the same as $\mathcal{H}_1$, except that we replace step 4) with the simulated $\mathcal{F}_{\mathsf{LTHReLU}}$.

We claim that $\mathcal{H}_1$ and $\mathcal{H}_2$ are computationally indistinguishable using the same argument on the trusted hardware and PRF security as in $\mathcal{H}_1$. Specifically, the random vectors generated by $\Pi_{\mathsf{LTH.GenMaskShare}}$ 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 $\mathcal{F}_{\mathsf{ReLU}}$ 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, $\mathcal{S}_{\mathsf{STAMP}}$ knows $\mathbf{x}_{3,1} = \mathbf{x}_{3,3}, \mathbf{x}_{1,1} = \mathbf{x}_{1,2}$. The dummy inputs would be $\mathbf{x}_{2,2} = \mathbf{x}_{2,3}$ (represented by $\hat{\mathbf{x}}_2$ in $\mathcal{S}_{\mathsf{STAMP}}$). The distribution of the computation result, $\hat{\mathbf{x}} = \mathbf{x}'_3 + \hat{\mathbf{x}}_2 + \mathbf{x}_{1,1} = (\mathbf{x}_{3,1}+\mathbf{m}_3)+\hat{\mathbf{x}}_2+\mathbf{x}_{1,1}$ and $\hat{\mathbf{x}}' = \mathbf{x}'_2+\mathbf{x}_{3,1}+\mathbf{x}_{1,1} = (\hat{\mathbf{x}}_2+\mathbf{m}_2)+\mathbf{x}_{3,1}+\mathbf{x}_{1,1}$ are uniformly random since $\mathbf{m}_3, \mathbf{m}_2$ are random. Therefore, $\mathcal{H}_2$ and $\mathcal{H}_3$ are indistinguishable.

The adversary's view of $\mathcal{H}_3$ is identical to $\mathsf{EXEC}_{F,\mathcal{S}_{\mathsf{STAMP}},\mathcal{E}}$. 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 $\mathcal{H}_0 \approx \mathcal{H}_1 \approx \mathcal{H}_2 \approx \mathcal{H}_3 = \mathcal{S}_{\mathsf{STAMP}}$.

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

The Simulator. $\mathcal{S}_{\mathsf{STAMP}}$ simulates the following interactions on receiving the signal from $\mathcal{F}_{\mathsf{ReLU}}$:

- Upon receiving $(\mathsf{ReLU}, 2, \mathbf{x}_{1,2}, \mathbf{x}_{2,2}, n, L)$ from $\mathcal{F}_{\mathsf{ReLU}}$, $\mathcal{S}_{\mathsf{STAMP}}$ generates a random $\hat{\mathbf{x}}_3$, and use $(\mathbf{x}_{2,2}, \hat{\mathbf{x}}_3)$ and $(\hat{\mathbf{x}}_3, \mathbf{x}_{1,2})$ as dummy inputs for P_3 and P_1, respectively.
- $\mathcal{S}_{\mathsf{STAMP}}$ acts as $\mathcal{F}_{\mathsf{GenMask}}$ to generate random masks $\mathbf{m}_3$ for P_1 and P_3, then computes $\mathbf{x}'_3 = \hat{\mathbf{x}}_3 + \mathbf{m}_3$ and sends $(\mathbf{m}_3, \mathbf{x}'_3)$ to P_1, computes $\mathbf{x}''_3 = \hat{\mathbf{x}}_3+\mathbf{m}_3$ and sends $(\mathbf{m}_3, \mathbf{x}''_3)$ to P_3 then also sends $\mathbf{x}'_3$ and $\mathbf{x}''_3$ to P_2 on behalf of P_1 and P_3, respectively. $\mathcal{S}_{\mathsf{STAMP}}$ also acts as $\mathcal{F}_{\mathsf{GenMask}}$ to generate random masks $\mathbf{m}_2$ for P_3 and P_2, then computes $\mathbf{x}'_2 = \mathbf{x}_{2,2} + \mathbf{m}_2$ and sends $(\mathbf{m}_2, \mathbf{x}'_2)$ to P_3, computes $\mathbf{x}''_2 = \mathbf{x}_{2,2} + \mathbf{m}_2$ and sends $(\mathbf{m}_2, \mathbf{x}''_2)$ to P_2 then also sends $\mathbf{x}'_2$ and $\tilde{\mathbf{x}}''_2$ (as adversary input for P_3) to P_1 on behalf of P_3 and P_2, respectively.
- $\mathcal{S}_{\mathsf{STAMP}}$ check $\mathbf{x}''_3 = \mathbf{x}'_3$, $\tilde{\mathbf{x}}''_2 = \mathbf{x}'_2$ on behalf of P_2 and P_1 respectively, signal abort to $\mathcal{F}_{\mathsf{ReLU}}$if inconsistency found.
- $\mathcal{S}_{\mathsf{STAMP}}$ acts as $\mathcal{F}_{\mathsf{LTHReLU}}$ with P_2's input $\hat{\mathbf{x}} = \mathbf{x}'_3 + \mathbf{x}_{2,2} + \mathbf{x}_{1,2}$ to (re)generate $\mathbf{m}_3$, randoms $\mathbf{z}^*_1, \mathbf{z}_2, \mathbf{z}^*_3$ such that $\mathbf{z}^*_1 + \mathbf{z}_2 + \mathbf{z}^*_3 = 0$, and then compute $(\mathbf{z}_1, \mathbf{z}_2) = (\mathsf{ReLU}(\hat{\mathbf{x}} - \mathbf{m}_3) + \mathbf{z}^*_1, \mathbf{z}_2)$. $\mathcal{S}_{\mathsf{STAMP}}$ sends $(\mathbf{z}_1, \mathbf{z}_2)$ to P_2. $\mathcal{S}_{\mathsf{STAMP}}$ sends $\tilde{\mathbf{z}}_1$ to P_1 and sends $\tilde{\mathbf{z}}_2$ to P_3 (as adversary inputs for P_2) . $\mathcal{S}_{\mathsf{STAMP}}$ acts as $\mathcal{F}_{\mathsf{LTHReLU}}$ with P_1's input $\hat{\mathbf{x}}' = \mathbf{x}'_2 + \hat{\mathbf{x}}_3 + \mathbf{x}_{1,2}$, (re)generate $\mathbf{m}_2$, generate randoms $\mathbf{z}^*_1, \mathbf{z}_2, \mathbf{z}^*_3$ such that $\mathbf{z}^*_1+\mathbf{z}_2+\mathbf{z}^*_3 = 0$, and then compute $(\mathbf{z}_3, \mathbf{z}'_1) = (\mathbf{z}^*_3, \mathsf{ReLU}(\hat{\mathbf{x}}' - \mathbf{m}_3)+\mathbf{z}^*_1)$. $\mathcal{S}_{\mathsf{STAMP}}$ sends $(\mathbf{z}_3, \mathbf{z}'_1)$ to P_1. $\mathcal{S}_{\mathsf{STAMP}}$ sends $\mathbf{z}_3$ to P_3 and sends $\mathbf{z}'_1$ to P_2 (on behalf of P_1).
- $\mathcal{S}_{\mathsf{STAMP}}$ acts as $\mathcal{F}_{\mathsf{GenMaskShr}}$ for P_1 to (re)generate random $\mathbf{z}^*_3$ and sends it to P_1. Similarly, $\mathcal{S}_{\mathsf{STAMP}}$ acts as $\mathcal{F}_{\mathsf{GenMaskShr}}$ for P_3 to (re)generate random $\mathbf{z}^*_3$ and sends it to P_3. $\mathcal{S}_{\mathsf{STAMP}}$ also generate $\mathbf{z}_2$ for P_2 and P_3.
- $\mathcal{S}_{\mathsf{STAMP}}$ checks if $\mathbf{z}'_1 = \tilde{\mathbf{z}}_1$ received by P_1, also P_2; if same $\mathbf{z}_2$ received by P_2, also P_3; if same $\mathbf{z}^*_3$ received by P_3, also P_1. Signal abort to $\mathcal{F}_{\mathsf{ReLU}}$if an inconsistency is found. If no abort is signaled, $\mathcal{S}_{\mathsf{STAMP}}$ signals $(\mathsf{ReLUend}, 2, \mathbf{z}_1, \mathbf{z}_2)$ to $\mathcal{F}_{\mathsf{ReLU}}$.

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 $\mathcal{H}_3$: $\mathcal{H}_3$ 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 $\mathcal{F}_{\mathsf{ReLU}}$ 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, $\mathcal{S}_{\mathsf{STAMP}}$ knows $\mathbf{x}_{2,2} = \mathbf{x}_{2,3}, \mathbf{x}_{1,2} = \mathbf{x}_{1,1}$. The dummy inputs would be $\mathbf{x}_{3,3} = \mathbf{x}_{1,3}$ (represented by $\hat{\mathbf{x}}_3$ in $\mathcal{S}_{\mathsf{STAMP}}$). The distribution of the computation result, $\hat{\mathbf{x}} = \mathbf{x}'_3 + \mathbf{x}_{2,2} + \mathbf{x}_{1,2} = (\hat{\mathbf{x}}_3 + \mathbf{m}_3) + \mathbf{x}_{2,2} + \mathbf{x}_{1,2}$ and $\hat{\mathbf{x}}' = \mathbf{x}'_2 + \hat{\mathbf{x}}_3 + \mathbf{x}_{1,2} = (\hat{\mathbf{x}}_{2,2} + \mathbf{m}_3) + \hat{\mathbf{x}}_3 + \mathbf{x}_{1,2}$ are uniformly random since $\mathbf{m}_3, \mathbf{m}_2$ are random. Therefore, $\mathcal{H}_2$ and $\mathcal{H}_3$ are indistinguishable.

The adversary's view of $\mathcal{H}_3$ is identical to $\mathsf{EXEC}_{F,\mathcal{S}_{\mathsf{STAMP}},\mathcal{E}}$. Therefore, in **Case 2** the view of $\mathcal{A}$ and $\mathcal{E}$are indistinguishable in the real and the simulated world.

Putting it all together, we have that $\mathcal{H}_0 \approx \mathcal{H}_1 \approx \mathcal{H}_2 \approx \mathcal{H}_3 = \mathcal{S}_{\mathsf{STAMP}}$.

Case 3: P_3 is corrupted ($i = 3$) and P_1, P_2 are honest.

The Simulator. $\mathcal{S}_{\mathsf{STAMP}}$ simulates the following interactions on receiving the signal from $\mathcal{F}_{\mathsf{ReLU}}$:

- Upon receiving $(\mathsf{ReLU}, 3, \mathbf{x}_{2,3}, \mathbf{x}_{3,3}, n, L)$ from $\mathcal{F}_{\mathsf{ReLU}}$, $\mathcal{S}_{\mathsf{STAMP}}$ generates a random $\hat{\mathbf{x}}_1$, and use $(\mathbf{x}_{3,3}, \hat{\mathbf{x}}_1)$ and $(\hat{\mathbf{x}}_1, \mathbf{x}_{2,3})$ as dummy inputs for P_1 and P_2, respectively.
- $\mathcal{S}_{\mathsf{STAMP}}$ acts as $\mathcal{F}_{\mathsf{GenMask}}$ to generate random masks $\mathbf{m}_3$ for P_1 and P_3, then computes $\mathbf{x}'_3 = \mathbf{x}_{3,3} + \mathbf{m}_3$ and sends $(\mathbf{m}_3, \mathbf{x}'_3)$ to P_1, computes $\mathbf{x}''_3 = \mathbf{x}_{3,3}+\mathbf{m}_3$ and sends $(\mathbf{m}_3, \mathbf{x}''_3)$ to P_3 then also sends $\mathbf{x}'_3$ and arbitrary $\tilde{\mathbf{x}}''_3$ to P_2 on behalf of P_1 and P_3, respectively. $\mathcal{S}_{\mathsf{STAMP}}$ also acts as $\mathcal{F}_{\mathsf{GenMask}}$ to generate random masks $\mathbf{m}_2$ for P_3 and P_2, then computes $\mathbf{x}'_2 = \mathbf{x}_{2,3} + \mathbf{m}_2$ and sends $(\mathbf{m}_2, \mathbf{x}'_2)$ to P_3, computes $\mathbf{x}''_2 = \mathbf{x}_{2,3} + \mathbf{m}_2$ and sends $(\mathbf{m}_2, \mathbf{x}''_2)$ to P_2 then also sends $\tilde{\mathbf{x}}'_2$ (as adversary input for P_3) and $\mathbf{x}''_2$ to P_1 on behalf of P_3 and P_2, respectively.
- $\mathcal{S}_{\mathsf{STAMP}}$ check $\tilde{\mathbf{x}}''_3 = \mathbf{x}'_3$, $\mathbf{x}''_2 = \tilde{\mathbf{x}}'_2$ on behalf of P_2 and P_1 respectively, signal abort to $\mathcal{F}_{\mathsf{ReLU}}$if inconsistency found.
- $\mathcal{S}_{\mathsf{STAMP}}$ acts as $\mathcal{F}_{\mathsf{LTHReLU}}$ with P_2's input $\hat{\mathbf{x}} = \mathbf{x}'_3 + \mathbf{x}_{2,3} + \hat{\mathbf{x}}_1$ to (re)generate $\mathbf{m}_3$, randoms $\mathbf{z}^*_1, \mathbf{z}_2, \mathbf{z}_3$ such that $\mathbf{z}^*_1 + \mathbf{z}_2 + \mathbf{z}_3 = 0$, and then compute $(\mathbf{z}^{**}_1, \mathbf{z}_2) = (\mathsf{ReLU}(\hat{\mathbf{x}} - \mathbf{m}_3) + \mathbf{z}^*_1, \mathbf{z}_2)$. $\mathcal{S}_{\mathsf{STAMP}}$ sends $(\mathbf{z}^{**}_1, \mathbf{z}_2)$ to P_2. $\mathcal{S}_{\mathsf{STAMP}}$ sends $\mathbf{z}^{**}_1$ to P_1 and sends $\mathbf{z}_2$ to P_3 (on behalf P_2). $\mathcal{S}_{\mathsf{STAMP}}$ acts as $\mathcal{F}_{\mathsf{LTHReLU}}$ with P_1's input $\hat{\mathbf{x}}' = \mathbf{x}'_2 + \hat{\mathbf{x}}_1 + \mathbf{x}_{3,3}$, (re)generate $\mathbf{m}_2$, generate randoms $\mathbf{z}^*_1, \mathbf{z}_2, \mathbf{z}_3$ such that $\mathbf{z}^*_1+\mathbf{z}^*_2+\mathbf{z}_3 = 0$, and then compute $(\mathbf{z}_3, \mathbf{z}'_1) = (\mathbf{z}_3, \mathsf{ReLU}(\hat{\mathbf{x}}' - \mathbf{m}_3)+\mathbf{z}^*_1)$. $\mathcal{S}_{\mathsf{STAMP}}$ sends $(\mathbf{z}_3, \mathbf{z}'_1)$ to P_1. $\mathcal{S}_{\mathsf{STAMP}}$ sends $\mathbf{z}_3$ to P_3 and sends $\mathbf{z}'_1$ to P_2 (on behalf P_1).
- $\mathcal{S}_{\mathsf{STAMP}}$ acts as $\mathcal{F}_{\mathsf{GenMaskShr}}$ for P_1 to (re)generate random $\mathbf{z}_3$ and sends it to P_1. Similarly, $\mathcal{S}_{\mathsf{STAMP}}$ acts as $\mathcal{F}_{\mathsf{GenMaskShr}}$ for P_3 to (re)generate random $\mathbf{z}_3$ and sends it to P_3. $\mathcal{S}_{\mathsf{STAMP}}$ also generate $\mathbf{z}_2$ for P_2 and P_3.

- $\mathcal{S}_{\text{STAMP}}$ checks if $\mathbf{z}'_1 = \mathbf{z}^{**}_1$ received by P_1, also P_2; if same $\mathbf{z}_2$ received by P_2, also P_3; if same $\mathbf{z}_3$ received by P_3, also P_1. Signal abort to $\mathcal{F}_{\text{ReLU}}$ if an inconsistency is found. If no abort is signaled, $\mathcal{S}_{\text{STAMP}}$ signals $(\text{ReLUend}, 3, \mathbf{z}_2, \mathbf{z}_3)$ 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 $\mathcal{H}_3$: $\mathcal{H}_3$ 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, $\mathcal{S}_{\text{STAMP}}$ knows $\mathbf{x}_{2,3} = \mathbf{x}_{2,2}, \mathbf{x}_{3,3} = \mathbf{x}_{3,1}$. The dummy inputs would be $\mathbf{x}_{1,1} = \mathbf{x}_{1,2}$ (represented by $\hat{\mathbf{x}}_1$ in $\mathcal{S}_{\text{STAMP}}$). The distribution of the computation result, $\hat{\mathbf{x}} = \mathbf{x}'_3 + \mathbf{x}_{2,3} + \hat{\mathbf{x}}_1 = (\mathbf{x}_{3,3} + \mathbf{m}_3) + \mathbf{x}_{2,3} + \hat{\mathbf{x}}_1$ and $\hat{\mathbf{x}}' = \mathbf{x}'_2 + \hat{\mathbf{x}}_1 + \mathbf{x}_{3,3} = (\hat{\mathbf{x}}_{2,3} + \mathbf{m}_3) + \hat{\mathbf{x}}_1 + \mathbf{x}_{3,3}$ are uniformly random since $\mathbf{m}_3, \mathbf{m}_2$ are random. Therefore, $\mathcal{H}_2$ and $\mathcal{H}_3$ are indistinguishable.

The adversary's view of $\mathcal{H}_3$ is identical to $\text{EXEC}_{F,\mathcal{S}_{\text{STAMP}},\mathcal{E}}$. 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 $\mathcal{H}_0 \approx \mathcal{H}_1 \approx \mathcal{H}_2 \approx \mathcal{H}_3 = \mathcal{S}_{\text{STAMP}}$ 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 Π_{ReLU}.

$\mathcal{F}_{\text{MatMulReLU}}$

(1) Upon receiving inputs $(\mathbf{A}_{3,1}, \mathbf{A}_{1,1}, \mathbf{B}_{3,1}, \mathbf{B}_{1,1})$, $(\mathbf{A}_{1,2}, \mathbf{A}_{2,2}, \mathbf{B}_{1,2}, \mathbf{B}_{2,2})$, $(\mathbf{A}_{2,3}, \mathbf{A}_{3,3}, \mathbf{B}_{2,3}, \mathbf{B}_{3,3})$ from P_1, P_2, P_3 respectively, check if $\mathbf{A}_{j,j} = \mathbf{A}_{j,j+1}, \mathbf{B}_{j,j} = \mathbf{B}_{j,j+1}$ for $j = 1,2,3$. If not, notify abort. Otherwise, compute $\mathbf{C}' = (\mathbf{A}_{1,1} + \mathbf{A}_{2,2} + \mathbf{B}_{3,3}) \times (\mathbf{B}_{1,1} + \mathbf{B}_{2,2} + \mathbf{B}_{3,3}) \gg \text{fp}$, $\mathbf{C} = \mathbf{C}' > \mathbf{0}$. Then, send a signal $(\text{MatMalReLU}, i, \mathbf{A}_{i-1,i}, \mathbf{A}_{i,i}, \mathbf{B}_{i-1,i}, \mathbf{B}_{i,i}, a, b, c, L)$ to $\mathcal{S}_{\text{STAMP}}$ 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 $(\text{MatMalReLUend}, i, \mathbf{C}_{i-1}, \mathbf{C}_i)$ from $\mathcal{S}_{\text{STAMP}}$, let $\mathbf{C}_{i+1} = \mathbf{z} - \mathbf{C}_{i-1} - \mathbf{C}_i$. Return $(\mathbf{C}_i, \mathbf{C}_{i+1})$ to the P_i for $i = 1,2,3$.

Figure 8: Ideal functionality for $\Pi_{\text{MatMulReLU}}$.

$\mathcal{F}_{\text{MatMulReLULTH}}$

Upon receiving signal with inputs $\hat{\mathbf{C}}'$, masks $\mathbf{M}$ and party index i, compute the following:

(1) $\mathbf{D} = \hat{\mathbf{C}}' - \mathbf{M}$, $\mathbf{E} = \mathbf{D} > \mathbf{0}$.
(2) Generate random $\mathbf{C}_1, \mathbf{C}_2, \mathbf{C}_3$ such that $\mathbf{C}_1 + \mathbf{C}_2 + \mathbf{C}_3 = \mathbf{C}$. Returns $(\mathbf{C}_i, \mathbf{C}_{i+1})$ to P_i (as mentioned in notation, for party index $i + 1$ means the next party).

Figure 9: Ideal functionality for the LTH part of Π_{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 $\mathcal{E}$ providing inputs to parties.
Case 1: P_1 is corrupted ($i = 1$) and P_2, P_3 are honest.

The Simulator. $\mathcal{S}_{\text{STAMP}}$ simulates the following interactions on receiving the signal from $\mathcal{F}_{\text{MatMalReLU}}$:

- Upon receiving $(\text{MatMalReLU}, 1, \mathbf{A}_{3,1}, \mathbf{A}_{1,1}, \mathbf{B}_{3,1}, \mathbf{B}_{1,1}, a, b, c, L)$ from $\mathcal{F}_{\text{MatMalReLU}}$, $\mathcal{S}_{\text{STAMP}}$ generates random $\hat{\mathbf{A}}_2, \hat{\mathbf{B}}_2$, and use $(\mathbf{A}_{1,1}, \hat{\mathbf{A}}_2, \mathbf{B}_{1,1}, \hat{\mathbf{B}}_2)$ and $(\hat{\mathbf{A}}_2, \mathbf{A}_{3,1}, \hat{\mathbf{B}}_2, \mathbf{B}_{3,1})$ as dummy inputs for P_2 and P_3, respectively.
- $\mathcal{S}_{\text{STAMP}}$ invokes the simulator of the secure multiplication protocol $\Pi_{\text{mal-arith-mult}}$ of [58] without the truncation. In the end, $(\hat{\mathbf{C}}_{3,1}, \hat{\mathbf{C}}1, 1), (\hat{\mathbf{C}}_{1,2}, \hat{\mathbf{C}}2, 2), (\hat{\mathbf{C}}_{2,3}, \hat{\mathbf{C}}3, 3)$ are distributed accordingly and $\mathcal{S}_{\text{STAMP}}$ will signal abort to $\mathcal{F}_{\text{MatMulReLU}}$if inconsistency was found in the distributed shares.
- $\mathcal{S}_{\text{STAMP}}$ acts as $\mathcal{F}_{\text{GenMask}}$ to generate random masks $\mathbf{M}_2$ for P_2 and P_3, $\mathbf{M}_3$ for P_3 and P_1 so that $\mathbf{M}_1 + \mathbf{M}_2 + \mathbf{M}_3 = \mathbf{0}$. Then $\mathcal{S}_{\text{STAMP}}$ computes $\hat{\mathbf{C}}'_{2,2} = \hat{\mathbf{C}}_{2,2} + \mathbf{M}_2$ as P_2 and $\hat{\mathbf{C}}'_{2,3} = \hat{\mathbf{C}}_{2,3} + \mathbf{M}_2$ as P_3; $\hat{\mathbf{C}}'_{3,3} = \hat{\mathbf{C}}_{3,3} + \mathbf{M}_3$ as P_3 and $\hat{\mathbf{C}}'_{3,1} = \hat{\mathbf{C}}_{3,1} + \mathbf{M}_3$ as P_1. $\mathcal{S}_{\text{STAMP}}$ sends $\hat{\mathbf{C}}'_{2,2}, \hat{\mathbf{C}}'_{2,3}$ to P_1 as P_2 and P_3, sends $\hat{\mathbf{C}}'_{3,3}, \hat{\mathbf{C}}'_{3,1}$ to P_2 as P_3 and P_1. $\mathcal{S}_{\text{STAMP}}$ aborts if any inconsistency is found on the pairs..
- $\mathcal{S}_{\text{STAMP}}$ computes $\hat{\mathbf{C}}'_1 = \hat{\mathbf{C}}'_{2,2} + \hat{\mathbf{C}}^L_{3,1} + \hat{\mathbf{C}}^L_{1,1}$ as P_1, $\hat{\mathbf{C}}'_2 = \hat{\mathbf{C}}'_{3,3} + \hat{\mathbf{C}}^L_{1,2} + \hat{\mathbf{C}}^L_{2,2}$ as P_2.
- $\mathcal{S}_{\text{STAMP}}$ acts as $\mathcal{F}_{\text{LTHReLU}}$ (H_1) with P_1's input $\hat{\mathbf{C}}'_1$ to (re)generate $\mathbf{M}_2$, computes $\hat{\mathbf{C}} = (\hat{\mathbf{C}}'_1 - \mathbf{M}_2) \gg \text{fp}$. $\mathcal{S}_{\text{STAMP}}$ computes $\mathbf{D} = \hat{\mathbf{C}} > \mathbf{0}$, generates $\mathbf{Z}'_1 + \mathbf{Z}'_2 + \mathbf{Z}'_3 = \mathbf{0}$ (as $\mathcal{F}_{\text{GenMaskShr}}$), then obtain $(\mathbf{Z}'_3, \mathbf{Z}'_1 + \mathbf{D})$ as $(\mathbf{Z}_{3,1}, \mathbf{Z}_{1,1})$ for P_1. $\mathcal{S}_{\text{STAMP}}$ acts as $\mathcal{F}_{\text{LTHReLU}}$ (H_2) with P_2's input $\hat{\mathbf{C}}'_2$ to (re)generate $\mathbf{M}_3$, computes $\hat{\mathbf{C}} = (\hat{\mathbf{C}}'_2 - \mathbf{M}_3) \gg \text{fp}$. $\mathcal{S}_{\text{STAMP}}$ computes $\mathbf{D} = \hat{\mathbf{C}} > \mathbf{0}$ (as $\mathcal{F}_{\text{GenMaskShr}}$), generates $\mathbf{Z}'_1 + \mathbf{Z}'_2 + \mathbf{Z}'_3 = \mathbf{0}$, then obtain $(\mathbf{Z}'_1 + \mathbf{D}, \mathbf{Z}'_2)$ as $(\mathbf{Z}_{1,2}, \mathbf{Z}_{2,2})$ for P_1.
- $\mathcal{S}_{\text{STAMP}}$ acts as $\mathcal{F}_{\text{GenMaskShr}}$ for P_3 to (re)generate $(\mathbf{Z}_{2,3}, \mathbf{Z}_{3,3}) = (\mathbf{Z}'_2, \mathbf{Z}'_3)$ and sends it to P_3. $\mathcal{S}_{\text{STAMP}}$ compare $\mathbf{Z}_{1,1} = \mathbf{Z}_{1,2}$ as P_1 and P_2; $\mathbf{Z}_{2,2} = \mathbf{Z}_{2,3}$, $\mathbf{Z}_{3,3} = \mathbf{Z}_{3,1}$ as P_3. Signal abort to $\mathcal{F}_{\text{MatMulReLU}}$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.

Hybrid $\mathcal{H}_1$: $\mathcal{H}_1$ is the same as $\mathcal{H}_0$, except that $\Pi_{\mathsf{LTH.GenMask}}$ is replaced with simulated $\mathcal{F}_{\mathsf{GenMask}}$ that outputs random $\mathbf{M}_2, \mathbf{M}_3$ for both step 2) and 5).
Hybrid $\mathcal{H}_2$: $\mathcal{H}_2$ is the same as $\mathcal{H}_1$, except that we replace step 5) with the simulated $\mathcal{F}_{\mathsf{LTHMatMulReLU}}$.

Proofs of $\mathcal{H}_0$ and $\mathcal{H}_1$, $\mathcal{H}_1$ and $\mathcal{H}_2$ being computationally indistinguishable are similar to the proof of Theorem 1, with the actual random masks repalced by $\mathbf{M}_2, \mathbf{M}_3$.
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. Also $\Pi_{\mathsf{mal\text{-}arith\text{-}mult}}$ of [58] is replaced by its corresponding simulator. In this hybrid, we introduce an ideal functionality $\mathcal{F}_{\mathsf{ReLU}}$ that takes the environments' actual inputs and returns the corresponding outputs.

We claim that $\mathcal{H}_2$ and $\mathcal{H}_3$ are indistinguishable. $\mathcal{H}_2$ and $\mathcal{H}_3$ are only different in inputs and step 1), and their indistinguishability directly comes from the security of the simulator of $\Pi_{\mathsf{mal\text{-}arith\text{-}mult}}$. 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, $\mathcal{H}_2$ and $\mathcal{H}_3$ are indistinguishable.

The adversary's view of $\mathcal{H}_3$ is identical to $\mathsf{EXEC}_{F,\mathcal{S}_{\mathsf{STAMP}},\mathcal{E}}$. 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 $\mathcal{H}_0 \approx \mathcal{H}_1 \approx \mathcal{H}_2 \approx \mathcal{H}_3 = \mathcal{S}_{\mathsf{STAMP}}$.

□

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.

$\mathcal{F}_{\mathsf{Softmax}}$

(1) Upon receiving inputs $(\mathbf{x}_{3,1}, \mathbf{x}_{1,1})$, $(\mathbf{x}_{1,2}, \mathbf{x}_{2,2})$, $(\mathbf{x}_{2,3}, \mathbf{x}_{3,3})$ from P_1, P_2, P_3 respectively, check if $\mathbf{x}_{j,j} = \mathbf{x}_{j,j+1}$ for $j = 1, 2, 3$. If not, notify abort. Otherwise, compute $\mathbf{x} = \mathbf{x}_{1,1} + \mathbf{x}_{2,2} + \mathbf{x}_{3,3}$, $\mathbf{z} = \lfloor \exp(\mathbf{x} \gg \mathsf{fp}) \ll \mathsf{fp} \rfloor$. Then, send a signal $(\mathsf{Softmax}, \mathbf{x}_{i-1,i}, \mathbf{x}_{i,i}, n, L, i)$ to $\mathcal{S}_{\mathsf{STAMP}}$ 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 $(\mathsf{Softmaxend}, i, \mathbf{z}_{i-1}, \mathbf{z}_i)$ from $\mathcal{S}_{\mathsf{STAMP}}$, let $\mathbf{z}_{i+1} = \mathbf{z} - \mathbf{z}_{i-1} - \mathbf{z}_i$. Return $(\mathbf{z}_i, \mathbf{z}_{i+1})$ 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 $\mathcal{E}$ providing inputs to parties.
Case 1: P_1 is corrupted ($i = 1$) and P_2, P_3 are honest.

$\mathcal{F}_{\mathsf{SoftmaxLTH}}$

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

(1) $\mathbf{x} = \mathbf{x}' - \mathbf{m}$, $\mathbf{a} = \lfloor \exp(\mathbf{x} \gg \mathsf{fp}) \ll \mathsf{fp} \rfloor$.
(2) Generate random $\mathbf{z}_1, \mathbf{z}_2, \mathbf{z}_3$ such that $\mathbf{z}_1 + \mathbf{z}_2 + \mathbf{z}_3 = \mathbf{z}$. Returns $(\mathbf{z}_i, \mathbf{z}_{i+1})$ 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. $\mathcal{S}_{\mathsf{STAMP}}$ simulates the following interactions on receiving the signal from $\mathcal{F}_{\mathsf{ReLU}}$:

- Upon receiving $(\mathsf{Softmax}, 1, \mathbf{x}_{3,1}, \mathbf{x}_{1,1}, n, L)$ from $\mathcal{F}_{\mathsf{Softmax}}$, $\mathcal{S}_{\mathsf{STAMP}}$ generates a random $\hat{\mathbf{x}}_2$, and use $(\mathbf{x}_{1,1}, \hat{\mathbf{x}}_2)$ and $(\hat{\mathbf{x}}_2, \mathbf{x}_{3,1})$ as dummy inputs for P_2 and P_3, respectively.
- $\mathcal{S}_{\mathsf{STAMP}}$ acts as $\mathcal{F}_{\mathsf{GenMask}}$ to generate random masks $\alpha_j \in \mathbb{Z}_{2^{52}}$ and $\beta_j \in \mathbb{Z}_{2^{32}}$ for $j = 1, \ldots, n$ for P_1, then computes $\bar{r} = \overline{\exp((\mathbf{x}_{3,1})_j \gg \mathsf{fp})}$ for $j = 1, \ldots, n$. Let $\bar{r} = \overline{\exp((\mathbf{x}_{3,1})_j \gg \mathsf{fp})} = 2^{q_j} \cdot (m_j \gg 52)$ as noted. $\mathcal{S}_{\mathsf{STAMP}}$ computes $\{m_j^* = (m_j + \alpha_j)_{2^{52}}, q_j^* = (q_j + \beta_j)_{2^{32}}\}$ for $i = 1, \ldots, n$ as P_1 sending to P_2. $\mathcal{S}_{\mathsf{STAMP}}$ repeat above for P_3, replacing $\mathbf{x}_{3,1}$ with $\mathbf{x}_{3,3}$; $\mathcal{S}_{\mathsf{STAMP}}$ again repeat above as P_2 and P_3, each replacing $\mathbf{x}_{3,1}$ with $\mathbf{x}_{2,2}$ and $\mathbf{x}_{2,3}$, with $\Pi'_{\mathsf{LTH.}}$ generating $(\boldsymbol{\alpha}', \boldsymbol{\beta}')$, and get $\{'m_j^*,' q_j^*\}$ respectively. $\{m_j^*, q_j^*\}$ are for P_2 provided by P_1 and P_2, and $\{'m_j^*,' q_j^*\}$ are for P_1 provided by P_2 and P_3
- $\mathcal{S}_{\mathsf{STAMP}}$ compare the received copies as P_2 and P_1, and signal abort to $\mathcal{F}_{\mathsf{Softmax}}$if an inconsistency is found. $\mathcal{S}_{\mathsf{STAMP}}$ computes as P_2: $2^{\hat{(q_2)}_j} \cdot \hat{(m_2)}_j := \overline{\exp(((\mathbf{x}_{2,2})_j + (\mathbf{x}_{1,2})_j) \gg \mathsf{fp})}$ for $j = 1, \ldots, n$. $\mathcal{S}_{\mathsf{STAMP}}$ computes as P_1: $2^{\hat{(q_1)}_j} \cdot \hat{(m_1)}_j := \overline{\exp(((\mathbf{x}_{1,1})_j + (\mathbf{x}_{3,1})_j) \gg \mathsf{fp})}$ for $j = 1, \ldots, n$.
- $\mathcal{S}_{\mathsf{STAMP}}$ acts as $\mathcal{F}_{\mathsf{LTHSoftmax}}$ with P_2's input $\{\mathbf{q}^* + \hat{\mathbf{q}}_2, \mathbf{m}^*, \hat{\mathbf{m}}_2\}$ to H_1, to (re)generate $(\boldsymbol{\alpha}, \boldsymbol{\beta})$, then locally compute $\mathbf{y}$ as in step 4) $\mathcal{S}_{\mathsf{STAMP}}$ then generate $\mathbf{m}_1 + \mathbf{m}_2 + \mathbf{m}_3 = \mathbf{0}$ and $(\mathbf{m}_1 + \mathbf{y}, \mathbf{m}_2)$ as $(\mathbf{y}_{1,2}, \mathbf{y}_{1,2})$ for P_2. $\mathcal{S}_{\mathsf{STAMP}}$ acts as $\mathcal{F}_{\mathsf{LTHSoftmax}}$ with P_1's input $\{'\mathbf{q}^* + \hat{\mathbf{q}}_1,' \mathbf{m}^*, \hat{\mathbf{m}}_1\}$ to H_1, to (re)generate $(\boldsymbol{\alpha}', \boldsymbol{\beta}')$, then locally compute $\mathbf{y}$ as in step 4). $\mathcal{S}_{\mathsf{STAMP}}$ then generate $\mathbf{m}_1 + \mathbf{m}_2 + \mathbf{m}_3 = \mathbf{0}$ and $(\mathbf{m}_3, \mathbf{y} + \mathbf{m}_1)$ as $(\mathbf{y}_{3,1}, \mathbf{y}_{1,1})$ for P_1. $\mathcal{S}_{\mathsf{STAMP}}$ send
- $\mathcal{S}_{\mathsf{STAMP}}$ checks $\mathcal{S}_{\mathsf{STAMP}}$ compare $\mathbf{y}_{1,1} = \mathbf{y}_{1,2}$ as P_1 and P_2; $\mathbf{y}_{2,2} = \mathbf{y}_{2,3}$, $\mathbf{y}_{3,3} = \mathbf{y}_{3,1}$ as P_3. Signal abort to $\mathcal{F}_{\mathsf{Softmax}}$if an inconsistency is found. If no abort is signaled, $\mathcal{S}_{\mathsf{STAMP}}$ signals $(\mathsf{Softmaxend}, 1, \mathbf{y}_{3,1}, \mathbf{y}_{1,1})$ to $\mathcal{F}_{\mathsf{Softmax}}$.

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.
Hybrid $\mathcal{H}_1$: $\mathcal{H}_1$ is the same as $\mathcal{H}_0$, except that$\Pi_{\mathsf{LTH.GenMask}}$ is replaced with simulated $\mathcal{F}_{\mathsf{GenMask}}$ that outputs random $(\boldsymbol{\alpha}, \boldsymbol{\beta})$ and $(\boldsymbol{\alpha}', \boldsymbol{\beta}')$ for both step 1) and 3).
Hybrid $\mathcal{H}_2$: $\mathcal{H}_2$ is the same as $\mathcal{H}_1$, except that we replace step 4) with the simulated $\mathcal{F}_{\mathsf{LTHSoftmax}}$.

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 ($\exp(x) \approx (1 + \frac{x}{2^e})^{\frac{x}{2^e}}$). 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	$12n + p + 317$
	AriaNN	2	3	9	-
	STAMP	2	2	2	2
Network comm. data	Falcon+	$16n$	$(20 + w^2)(\frac{m}{s} - 1)^2$	$224n$	$(\frac{m}{s} - 1)^2(n + 20) + (110 + e)n$
	AriaNN	$12n$	$(\frac{m}{s} + 1)^2(w^4 + 1)$	$72n$ -	-
	STAMP	$5n$	$\frac{2}{3}(2n + 2w^2 + 5(\frac{m}{s} + 1)^2)$	$4n$	$\frac{20}{3}n$
LTH comm.	STAMP	$\frac{25}{3}n$	$\frac{1}{3}(8m^2 + 8w^2 + 15(\frac{m}{s} + 1)^2)$	$\frac{16}{3}n$	$\frac{44}{3}n$

Proofs of $\mathcal{H}_0$ and $\mathcal{H}_1$, $\mathcal{H}_1$ and $\mathcal{H}_2$ being computationally indistinguishable are similar to the proof of Theorem 1, with the actual random masks repalced by $(\boldsymbol{\alpha}, \boldsymbol{\beta})$ and $(\boldsymbol{\alpha}', \boldsymbol{\beta}')$.

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 $\mathcal{F}_{\mathsf{Softmax}}$ 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, $\mathcal{S}_{\text{STAMP}}$ knows $\mathbf{x}_{3,1} = \mathbf{x}_{3,3}, \mathbf{x}_{1,1} = \mathbf{x}_{1,2}$. The dummy inputs would be $\mathbf{x}_{2,2} = \mathbf{x}_{2,3}$ (represented by $\hat{\mathbf{x}}_2$ in $\mathcal{S}_{\text{STAMP}}$). The computation result sent to P_1 by P_3 in step 1) used the dummy inputs, and $\{'m^*_j = ('m_j + \alpha'_j)_{2^{52}}, 'q^*_j = ('q_j + \beta'_j)_{2^{32}}\}$ for $i = 1, ..., n$ are uniformly random since $(\boldsymbol{\alpha}', \boldsymbol{\beta}')$ 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, $\mathcal{H}_2$ and $\mathcal{H}_3$ are indistinguishable.

The adversary's view of $\mathcal{H}_3$ is identical to $\mathsf{EXEC}_{F, \mathcal{S}_{\text{STAMP}}, \mathcal{E}}$. 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 $\mathcal{H}_0 \approx \mathcal{H}_1 \approx \mathcal{H}_2 \approx \mathcal{H}_3 = \mathcal{S}_{\text{STAMP}}$.

□

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, $w \times h$ 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	-	$2l$
MaxPool	$\{w, h\}$	$2whl$
BatchNorm	-	$2l$
LayerNorm	n	$(2n + 1)l$
Softmax	n	$6nl$

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 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 •, 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 $\frac{20}{3}n$ to $\frac{68}{3}n$. 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.