SwiftMPC: Accelerating ML under MPC with Secure-Chip-Assisted One-Way Communication

Tianze Song
Zhejiang University
Hangzhou, Zhejiang, China
stz@zju.edu.cn

Chengkun Wei[✉*]
Zhejiang University
Hangzhou, Zhejiang, China
weichengkun@zju.edu.cn

Yichen Tu
Alibaba Cloud
Beijing, China

Yangming Zhang
Alibaba Cloud
Shenzhen, Guangdong, China

Heng Cai
Alibaba Cloud
Hangzhou, Zhejiang, China

Haiyuan Gao
Alibaba Cloud
Hangzhou, Zhejiang, China

Yuyang Cui
Tianjin University
Tianjin, China
cuiyuyang@tju.edu.cn

Minghong Tang
Zhejiang University
Hangzhou, Zhejiang, China
tangmh@zju.edu.cn

Jin Shi
Zhejiang University
Hangzhou, Zhejiang, China
shijin4300@zju.edu.cn

Tong Chen
Zhejiang University
Hangzhou, Zhejiang, China
tongc727@zju.edu.cn

Yin Zhang
Zhejiang University
Hangzhou, Zhejiang, China
zhangyin98@zju.edu.cn

Wei Wang
Alibaba Group
Shenzhen, Guangdong, China

Wenzhi Chen[✉*]
Zhejiang University
Hangzhou, Zhejiang, China
chenwz@zju.edu.cn

Abstract

Secure multi-party computation (MPC) is a promising foundation for privacy-preserving machine learning. Its deployment for modern ML is limited by communication: Beaver-based multiplication requires bidirectional communication that places network latency directly on the critical path. We present SwiftMPC, a secure-chip-assisted MPC architecture built on a simple protocol insight: if one party sets its first two Beaver-triple components equal to its existing input shares, that party's masked differences become identically zero and need not be transmitted. Secure multiplication thus becomes one-way, letting communication and computation be pipelined. Such triples cannot be precomputed cryptographically, nor supplied by a trusted third party without reintroducing a round trip. SwiftMPC instead generates them with a secure chip attached to one participant. The chip plays a narrow, fixed-function role limited to triple generation, keeping it simple to verify.

We implement SwiftMPC with an FPGA secure-chip prototype and a drop-in runtime compatible with PyTorch-style programming. To better support machine-learning workloads, SwiftMPC further generates Beaver triples at matrix granularity, which reduces triple consumption and communication volume for dominant ML operators. Across representative ML workloads, SwiftMPC completes ResNet-18 inference in 2.22 s at 65 ms network latency, achieving up to 1.64× speedup over the strongest prior hardware-assisted MPC design and up to 125× over software-only MPC.

CCS Concepts: • **Security and privacy** → **Privacy-preserving protocols**; **Hardware-based security protocols**; • **Hardware** → *Hardware accelerators*; • **Computing methodologies** → Machine learning.

Keywords: Secure multi-party computation, Privacy-preserving machine learning

ACM Reference Format:
Tianze Song, Chengkun Wei, Yichen Tu, Yangming Zhang, Heng Cai, Haiyuan Gao, Yuyang Cui, Minghong Tang, Jin Shi, Tong Chen, Yin Zhang, Wei Wang, and Wenzhi Chen. 2026. SwiftMPC: Accelerating ML under MPC with Secure-Chip-Assisted One-Way Communication. In *The 2026 ACM SIGOPS Annual Technical Conference (SIGOPS ATC 2026), November 15-18, 2026, Hong Kong, China.*

[*]Corresponding authors: Wenzhi Chen and Chengkun Wei.



This work is licensed under a Creative Commons Attribution 4.0 International License.

SIGOPS ATC 2026, Hong Kong, China

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2818-1/2026/11
https://doi.org/10.1145/3828523.3857086

ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3828523.3857086

1 Introduction

Secure multi-party computation (MPC) [12, 33] is an appealing foundation for privacy-preserving machine learning because it enables multiple parties to jointly compute a function without exposing their plaintext inputs during execution. The property is especially attractive in cross-organization settings, where participants are unwilling to entrust sensitive data to a centralized execution platform. Yet, despite its strong security semantics, MPC remains difficult to deploy for modern machine-learning workloads.

The main obstacle lies in communication. In additive-secret-sharing-based MPC [8, 12, 13], additions are local, whereas multiplications rely on Beaver triples [5, 6] and require parties to exchange masked differences before the computation can proceed. Because modern ML workloads contain massive numbers of multiplications, this bidirectional communication places network latency directly on the critical path. In realistic wide-area deployments where round-trip times reach tens or hundreds of milliseconds, the resulting slowdown is prohibitive [15, 39, 55].

In this paper, we explore a different design point: *can the bidirectional communication in Beaver-based multiplication be removed altogether?* We show that the answer is yes. Our key insight is that the bidirectional communication arises solely because both parties must mask their shares and exchange the masked differences. If one party sets its first two Beaver-triple components *equal to* its existing input shares (as illustrated in Figure 1(b)), that party's masked differences become identically zero and need not be transmitted at all. Secure multiplication thus reduces from bidirectional to one-way communication, turning a latency-dominated primitive into a pipelinable dataflow that overlaps communication with computation.

Realizing this insight, however, is impossible with existing triple-generation approaches, since it requires generating the triple based on a party's live input shares. Cryptographic preprocessing, e.g., via homomorphic encryption [13] or oblivious transfer [32], produces triples ahead of time, but the live input shares are only determined during online computation, so the corresponding triple cannot be precomputed. A trusted third party [46] could generate such a triple, but only after receiving the share, which reintroduces a round trip on the critical path of every multiplication. Neither approach can realize one-way secure multiplication, calling for a new triple-generation mechanism.

To realize this insight and overcome the limitations of existing methods, we present SwiftMPC, a secure-chip-assisted MPC architecture for machine learning. A secure chip attached to only one participant generates Beaver triples locally from that participant's live input shares and pseudorandom values generated from a seed shared with the remote participant. This local generation avoids expensive cryptographic preprocessing and network round trips, preserving one-way communication and enabling pipelined execution. The chip is confined to a narrow, fixed-function role limited to triple generation, keeping its design simple and the trusted functionality small. The design also avoids introducing an external trusted third party that could collude with a participant. To support ML workloads efficiently, SwiftMPC further extends Beaver triple generation from scalar operations to matrix granularity in both software and hardware, which reduces both triple consumption and communication volume for matrix multiplication and convolution.

We implement SwiftMPC as a hardware-software co-design that combines an FPGA secure-chip prototype with a drop-in runtime compatible with PyTorch-style programming, allowing users to adopt SwiftMPC with minimal code changes. On representative convolutional and Transformer [50] workloads, SwiftMPC completes ResNet-18 [23] inference in 2.22 s at 65 ms network latency, achieving up to 1.64× speedup over the strongest prior hardware-assisted MPC design [57] and up to 125× over software-only MPC.

Overall, this paper makes the case for a different design point for practical MPC: removing the bidirectional communication by restructuring the protocol with trusted hardware. By keeping the hardware role small while preserving one-way communication and strong end-to-end performance, SwiftMPC brings MPC closer to practical privacy-preserving machine learning. In summary, this paper makes the following contributions:

- **A protocol-level redesign for one-way secure multiplication.** We redesign the Beaver-multiplication protocol so that setting one party's first two Beaver-triple components equal to its existing input shares makes that party's masked differences zero, reducing secure multiplication from bidirectional to one-way communication and making it pipelinable. We realize this redesigned protocol with a secure chip attached to one participant that generates the triple locally, serving as a narrow trust anchor in place of a trusted third party.
- **Hardware-software co-design with matrix-level triple generation.** We design SwiftMPC as a hardware-software co-design with a PyTorch-compatible drop-in runtime and an FPGA secure-chip prototype. By lifting Beaver triple generation to matrix granularity, SwiftMPC reduces triple consumption and communication volume for dominant ML operators.

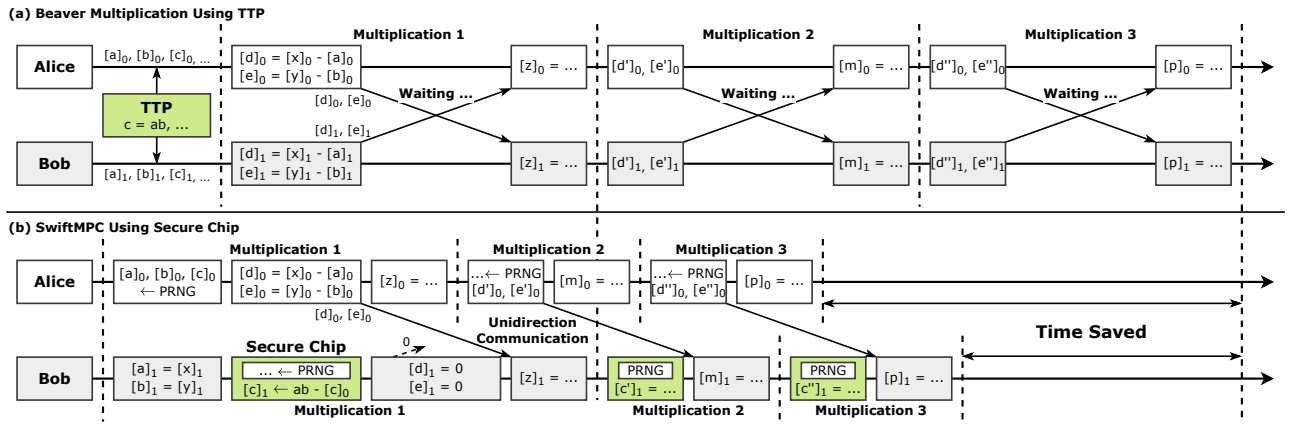


Figure 1. Comparison between Beaver multiplication (a) and SwiftMPC (b) when performing $z = x \times y$ followed by two more multiplications. In (b), by setting Bob's Beaver-triple components equal to Bob's existing input shares ($[a]_1 = [x]_1, [b]_1 = [y]_1$), the masked differences $[d]_1, [e]_1$ become zero and need not be transmitted, enabling unidirectional communication and pipelined execution. Green regions indicate components that both parties must trust.

- **End-to-end performance improvements.** Across representative ML workloads and latency settings, SwiftMPC achieves up to 1.64× speedup over a prior secure-chip-based design and up to 125× over a software-only baseline, including 2.22 s ResNet-18 inference at 65 ms latency.

2 Background

2.1 Secure Multi-Party Computation

Secure MPC allows multiple parties to jointly compute a function over their private inputs without revealing these inputs to one another. Formally, given n parties $P_1, P_2, \ldots, P_n$ holding private inputs $x_1, x_2, \ldots, x_n$, the goal is to compute $f(x_1, x_2, \ldots, x_n)$ such that no party learns anything beyond its own input and the output of f.

A broad class of MPC protocols has been developed to balance security, efficiency, and expressiveness. The garbled circuit [7, 29, 33] is one of the earliest and most fundamental constructions for secure two-party computation. In this approach, one party (the garbler) encodes a Boolean circuit into a "garbled" form, encrypting the truth tables of each gate, while the other party (the evaluator) evaluates the circuit using encrypted input labels obtained via oblivious transfer. Secret-sharing-based protocols [45, 53] split each private value into random shares distributed among participants. A function can then be evaluated on these shares using arithmetic operations over a finite field, with correctness ensured by the linearity of addition and multiplication sharing schemes. Homomorphic encryption [10, 19, 47] allows computations directly on ciphertexts without decryption, making it attractive for outsourcing privacy-preserving computation to untrusted servers.

Among these MPC paradigms, additive secret sharing [8, 12, 13] has become a widely adopted approach due to its computational simplicity and suitability for large-scale numerical workloads. In an additive secret sharing scheme over a finite field $\mathbb{Z}_p$, a value x is split into n shares $[x]_1, [x]_2, \ldots, [x]_n$ such that $x = [x]_1 + [x]_2 + \cdots + [x]_n \pmod{p}$. Each party holds one share, and the reconstruction of x requires all shares. Additive secret sharing provides an elegant property that addition can be performed locally because $[x] + [y] = [x + y]$, where $[x]$ and $[y]$ denote shared values of an arbitrary party. However, multiplication is not locally computable and requires interactive protocols between parties. Efficiently performing secure multiplications thus becomes the central challenge in additive secret sharing-based MPC.

2.2 Using Beaver Triple to Compute Multiplication

Beaver [5, 6] introduces Beaver triples to reduce the per-multiplication communication required during the online phase. A Beaver triple is a tuple (a, b, c) of pre-computed random values satisfying $c = a \times b$. Given such a triple, parties can securely compute the product $z = x \times y$ of two secret-shared values $[x]$ and $[y]$ (also illustrated in Figure 1(a)): each party masks its shares by computing $[d] = [x] - [a]$ and $[e] = [y] - [b]$, then all parties exchange and reconstruct the plaintext d and e. Since d and e are masked by the random triple components, they reveal no information about x or y. Each party then locally derives $[z]$ from $[c]$, d, and e. Notably, this exchange of d and e requires bidirectional, blocking communication. Computation must stall until both values are received from all parties.

2.3 Generation of Beaver Triples

The generation and distribution of Beaver triples remain performance-critical in MPC systems. Because each secure multiplication consumes a triple, the triple-generation throughput must match the computational demand. Existing systems obtain triples through two primary approaches:

Cryptographic-based generation. Triples can be generated collaboratively using oblivious transfer [32] or homomorphic encryption [13, 58]. In this approach, the participants jointly execute a cryptographic preprocessing protocol to obtain shares of valid Beaver triples, which are then consumed during the subsequent Beaver-based MPC computation.

TTP-based generation. To make triple generation faster, a trusted third party can be introduced to generate Beaver triples in advance and distribute the corresponding shares to the participants [43]. In this approach, the computation workload proceeds using these pre-generated triples, while the triple generation is offloaded to the TTP. This makes TTP-based generation a straightforward and efficient way to supply triples for Beaver-based MPC.

3 Motivation and Challenges

This section motivates SwiftMPC by analyzing the trust and performance limitations of existing MPC systems and identifying the resulting design challenges.

3.1 Limitations of Existing MPC for End-User Machine Learning

Performance bottlenecks. Existing MPC systems are fundamentally limited by interaction-dominated online execution. As illustrated in Figure 1(a), each Beaver-based secure multiplication requires parties to exchange masked differences $([d]_i, [e]_i)$ and synchronize before proceeding. Since the associated local arithmetic is lightweight, the cost of secure multiplication is governed primarily by communication latency rather than computation. This bottleneck is particularly severe for machine-learning workloads, where dominant operators such as matrix multiplication and convolution translate into massive numbers of secure multiplications. The resulting interaction overhead places network latency directly on the critical path and severely limits end-to-end performance.

For example, in a representative MPC framework, ResNet-18 inference takes 63.31 s under 65 ms network latency (§7), indicating that interaction, rather than local computation, is the principal obstacle to practical deployment. A breakdown of end-to-end runtime further shows that communication and synchronization dominate execution time under realistic network latency, while local arithmetic accounts for only a small fraction of the total cost. Across all tested workloads, communication and synchronization dominate runtime under realistic WAN latency, confirming that existing Beaver-based MPC is fundamentally interaction-bound.

Trust limitations. Beyond this performance bottleneck, existing systems face a second limitation rooted in how Beaver triples are provisioned for secure multiplications. Existing systems generate these triples either cryptographically, e.g., through oblivious transfer [32] or homomorphic encryption [13, 58], or with the help of a trusted third party (TTP). The former incurs substantial computation and communication overhead, which is particularly burdensome for practical machine-learning workloads in resource-constrained end-user environments. The latter is more efficient, but enlarges the trust boundary by requiring users to trust an external party not to leak information or collude with any participant. Once this assumption fails, plaintext values may be recovered immediately [43]. Existing approaches therefore expose a fundamental tradeoff between preprocessing efficiency and trust expansion.

Taken together, these limitations reveal a fundamental tension in existing MPC systems: practical deployment requires reducing both trust expansion in preprocessing and latency-sensitive interaction in online execution.

3.2 Challenges

These limitations reveal a fundamental design tension in practical MPC for machine learning: reducing interaction and preprocessing cost often comes at the price of expanding the trust boundary. A successful design must therefore address performance, trust, and deployability jointly, rather than optimizing any one dimension in isolation.

C1: Removing latency-sensitive interaction from online execution. Standard MPC protocols require bidirectional communication for each secure multiplication or bitwise AND. Because the underlying local arithmetic is inexpensive, network latency rather than computation dominates execution time. The first challenge is therefore to eliminate mutual communication from the critical path of online execution.

C2: Avoiding trust expansion while retaining efficient triple generation. Efficient Beaver triple generation is essential, but existing methods are unsatisfactory: cryptographic preprocessing is costly, whereas TTP-based provisioning enlarges the trust boundary. More broadly, accelerating MPC through hardware assistance can easily shift more of the secure computation path into trusted hardware. The second challenge is therefore to achieve efficient triple generation and reduced interaction without relying on either an external trusted party or plaintext-style trusted execution.

C3: Matching the structure of modern ML workloads. Real machine-learning models are dominated by matrix multiplication and convolution. The third challenge is to support

these operators efficiently and expose them through interfaces compatible with modern ML frameworks.

C4: Maintaining practical deployability. Any design intended for end-user settings must keep hardware assumptions, deployment complexity, and software changes modest. Otherwise, even a performant and secure system would remain difficult to adopt in practice.

4 Secure-Chip-Assisted One-Way MPC

This section presents the design intuition behind SwiftMPC and defines the threat model underlying our secure-chip-assisted approach.

4.1 Design Rationale

The design of SwiftMPC starts from a protocol-level observation about the communication cost of Beaver multiplication. As illustrated in Figure 1(b), if Bob sets their Beaver-triple components $([a]_1, [b]_1)$ equal to their existing input shares $([x]_1, [y]_1)$, Bob's masked differences become zero and need not be transmitted to Alice. Secure multiplication thus becomes *unidirectional*: only Alice sends data, allowing communication and computation to be pipelined rather than synchronized at every multiplication.

Realizing this idea, however, requires constructing a Beaver triple whose first two components for a participant equal that participant's existing input shares, which existing triple-generation methods cannot provide. Cryptographic preprocessing fixes triples ahead of time, but the corresponding input shares are only available during online computation. A trusted third party could generate such a triple, but only after receiving the share, which reintroduces a round trip on the critical path. SwiftMPC resolves this with a secure chip attached to a single participant. It exposes only a restricted fixed-function interface, protects its internal state from host inspection, and participates solely to assist triple generation, rather than executing the secure computation itself.

Before computation, Alice and Bob's secure chip establish a shared PRNG seed that remains hidden from Bob's host. For each multiplication, Alice uses the PRNG to generate her triple shares, computes the masked differences from her input shares, and sends them to Bob. Bob supplies the existing input shares to the chip, which sets Bob's first two triple components equal to those shares and computes the remaining component using the synchronized randomness, as detailed in Section 5. Bob's masked differences are therefore zero and need not be sent to Alice. Both parties then compute their output shares locally, enabling one-way communication to be pipelined with computation.

Beyond enabling one-way communication, generating triples on the chip avoids the drawbacks of both existing methods. Compared with cryptographic preprocessing, it removes the heavy computation overhead of generating triples. Compared with a trusted third party, the chip is not an external authority serving all participants. It is attached to a single participant, operates only on that participant's local secret shares, and never receives the other party's plaintext or shares. Its simple, fixed-function design keeps the trusted functionality small and well-defined.

4.2 Threat Model

We consider an MPC setting in which participants jointly evaluate a function over additively secret-shared inputs. One participant is equipped with a secure chip that assists the protocol by generating Beaver triples from local secret shares. Participants communicate over an untrusted network.

Adversary. We assume a semi-honest threat model: corrupted participants follow the protocol but attempt to infer private inputs from their local state and observed messages. We do not consider malicious participants that arbitrarily deviate from the protocol. If the corrupted party is the secure-chip holder, it may fully control the host platform attached to the chip, including the OS, drivers, user-space software, and DMA-capable peripherals. Side-channel attacks and physical tampering are outside our threat model.

Secure-chip assumptions. SwiftMPC assumes that the secure chip provides: (1) confidentiality and integrity of its internal state, including synchronized PRNG state, ephemeral registers, on-chip memory, and device-rooted secrets for authentication and seed establishment, even when the host is fully compromised; (2) correct execution of the specified fixed-function interface; (3) a restricted I/O interface that exposes only intended commands and outputs; (4) support for authenticated chip identification and secure seed establishment before protocol execution.

5 SwiftMPC Protocol Design

This section presents the protocol foundation of SwiftMPC. We formalize the protocols for arithmetic operations and Boolean AND, explain the inherent support for matrix-level operations, and analyze correctness and security of the design.

5.1 Protocol Workflow

Protocol 1 shows how SwiftMPC computes $z = x \times y$. Specifically, Bob is equipped with a secure chip, and Alice synchronizes a random seed with this chip so that both of them can generate the same pseudorandom sequence. These random numbers will later be used as part of Beaver triples. To start with, Alice generates $([a]_0, [b]_0, [c]_0)$ from the pseudorandom generator and follows the standard Beaver protocol by computing the differences $([d]_0, [e]_0)$ and sending them to Bob, and finally computes their local output $[z]_0$. On the other side, Bob provides their input shares $([x]_1, [y]_1)$ to the secure chip, which are treated as $([a]_1, [b]_1)$ in the Beaver protocol. Together with the synchronized $([a]_0, [b]_0, [c]_0)$,

Protocol 1 Two parties compute $x \otimes y \rightarrow z$ with secure chip

Input: Alice and the secure chip of Bob have a pair of synchronized PRNGs. They have shared their data to secret share $[x]_0, [x]_1$ and $[y]_0, [y]_1$, where $[x]_0 + [x]_1 = x$ and $[y]_0 + [y]_1 = y$.

Output: Alice gets $[z]_0$ and Bob gets $[z]_1$ where $[z]_0 + [z]_1 = z$ and $x \otimes y = z$. $\otimes$ here can represent scalar or element-wise vector multiplication, matrix multiplication or convolution, as long as the shape of x and y to perform $x \otimes y$ is valid, and the shapes of a, b, c correspond one-to-one with those of x, y, z. When performing convolution, each $\otimes$ must use the same stride and padding.

Procedure of Alice:

1. Generates $[a]_0, [b]_0, [c]_0$ as part of a Beaver triple from PRNG sequentially
2. $d \leftarrow [x]_0 - [a]_0$
3. $e \leftarrow [y]_0 - [b]_0$
4. Sends d, e to Bob
5. $[z]_0 = [c]_0 + e \otimes [a]_0 + d \otimes [b]_0 + e \otimes d$ (in practice, this could be simplified as $[z]_0 = [c]_0 + [x]_0 \otimes [y]_0 - [a]_0 \otimes [b]_0$)

Procedure of Bob with a Secure Chip:

1. Sends $[x]_1, [y]_1$ to secure chip (as $[a]_1, [b]_1$ in traditional Beaver multiplication)
   a. Secure chip generates $[a]_0, [b]_0, [c]_0$ identical to those of Alice from PRNG sequentially inside the chip
   b. Secure chip computes $[c]_1 = ([a]_0 + [x]_1) \otimes ([b]_0 + [y]_1) - [c]_0$ and only sends $[c]_1$ to Bob
2. Receives d, e and computes $[z]_1 = [c]_1 + e \otimes [x]_1 + d \otimes [y]_1$ outside the secure chip

the chip can compute $[c]_1$ and return it to Bob. Since $[a]_1 = [x]_1$ and $[b]_1 = [y]_1$, the differences $([d]_1, [e]_1)$ are always zero and thus do not need to be transmitted to Alice (as the dashed line in Figure 1(b) indicates). Bob can directly compute $[z]_1$ as their output share. In this procedure, Bob does not need to send any network message to Alice, so the protocol achieves unidirectional communication. As Figure 1(b) shows, removing network latency from the critical path of every multiplication enables pipelined computation that amortizes network latency across all operations, thereby resolving **C1**. It also addresses **C2** by generating Beaver triples inside the secure chip, thereby avoiding reliance on an external trusted third party. The binary AND protocol follows the same one-way structure as Protocol 1, except that the computation is carried out over binary shares, with subtraction/addition and multiplication replaced by XOR and AND, respectively. The protocol requires only one participant to hold a secure chip, keeping hardware deployment requirements modest and addressing **C4**.

Support for Matrix-Level Operations. Note that the proposed Protocol 1 also extends beyond scalar multiplication to matrix multiplication, element-wise matrix multiplication, and convolution [37]. In this case, the secure chip can generate matrix-level Beaver triples directly, e.g., matrix multiplication $C = AB$ or convolution $C = A@B$. Matrix-level Beaver triples in SwiftMPC inherently exhibit a caching property. In our matrix multiplication or convolution, each pair of multiplicands is associated with a specific entry in the pre-generated matrix triple, rather than generating a new scalar triple for every scalar multiplication. For example, an $n \times n$ matrix multiplication using scalar Beaver triples would require n^3 multiplications and hence n^3 triples, where Alice sends n^3 elements to Bob. In contrast, a matrix-level Beaver triple requires only n^2 elements, reducing both the Beaver triple generation cost and chip throughput demand. Similarly, for convolutions using a $k \times k$ kernel, the triples reduce from n^2k^2 to $n^2 + k^2$. To exploit this, we designed the secure chip with native support for matrix operations (detailed in Section 6.2). This resolves **C3** by providing efficient protocol-level support for matrix multiplication and convolution, which are the dominant operators in real-world machine-learning workloads. As a result, SwiftMPC can execute machine-learning workloads more efficiently.

5.2 Protocol Security Analysis

From a security perspective, the following analysis shows that the proposed protocol is secure, which means both Alice and Bob cannot get or infer plaintext data using the information they have. For Alice, recovering the plaintext would require knowledge of Bob's shares $([x]_1, [y]_1)$ or their output $[z]_1$. However, Bob never sends any data to Alice throughout the computation, and the pseudorandomly generated $([a]_0, [b]_0, [c]_0)$ are independent of Bob's inputs. Thus, Alice gains no information about Bob's data. For Bob, recovering the plaintext would require Alice's shares $([x]_0, [y]_0)$ or their output $[z]_0$, which may be inferred from $([d]_0, [e]_0)$ and $([a]_0, [b]_0)$. Nevertheless, Bob's control of the host does not reveal $([a]_0, [b]_0, [c]_0)$, because these values remain inside the secure chip. Even if Bob observes $([d]_0, [e]_0)$ or the chip output $[c]_1$, these values are masked by fresh random numbers $([a]_0, [b]_0, [c]_0)$ which serve as one-time pads. Consequently, Bob cannot extract meaningful information. Both parties cannot recover the plaintext data, therefore the modified Beaver protocol with secure chip maintains security. In addition, the secure chip never receives Alice's shares $([x]_0, [y]_0)$, so the chip is not capable of reconstructing their plaintext during protocol execution.

Shared Seed Establishment. Chip authentication and shared seed establishment can be implemented using TLS 1.3 [42]. The remote party authenticates the secure

chip by validating its device certificate chain to a trusted root CA, with the corresponding private key and TLS endpoint protected inside the chip. Once the authenticated TLS handshake completes, both endpoints use the TLS keying-material exporter with the same application-specific label and context to derive an identical, session-specific PRNG seed locally. This seed is distinct from the TLS traffic encryption keys and does not need to be transmitted explicitly.

Prior work has shown that authentication and secure channel establishment can be realized on chip-based trusted devices using existing hardware roots of trust [56, 57]. Commodity FPGAs such as AMD [2] and Intel [24] already provide the hardware root of trust needed for this process by securely storing device-specific secrets and verification keys in on-chip protected non-volatile storage. We measure the execution time of shared seed establishment through simulation and analyze the results in § 7.

Protocol-Level Reduction of Side-Channel Exposure. During computations performed within the secure chip, if the participant holding the chip were able to access the internal data, the plaintext could potentially be exposed. One representative attack is to perform repeated computations on the same input data (e.g., multiplying the same numbers multiple times) while collecting side-channel information, such as timing, power consumption, or voltage variations, and then analyze these patterns to infer the input values. In the secure chip of SwiftMPC, however, each multiplication $a \cdot b$ is masked by a fresh random pair $([a]_0, [b]_0)$ (both in the two-party and multi-party scenarios). Consequently, even if the same data is processed multiple times, the actual operands used in the chip's multiplication operation differ each time, which reduces the repeatability that such attacks rely on and raises the difficulty of extracting useful information.

6 SwiftMPC Implementation

This section introduces how we design the host software and hardware, and make them work together to implement SwiftMPC based on our protocol. Figure 2 illustrates the overall architecture of SwiftMPC. The software side is illustrated as the MPC machine-learning workload and its decomposition into intermediate layers and operators. SwiftMPC uses CrypTen [26] to split MPC machine-learning workload into the smallest actual executable operators. The non-local operators that require Beaver triples are processed using the protocol we proposed. On the hardware side, we implement a prototype secure chip using an FPGA. Participant holding the secure chip invokes it when handling non-local operators to generate Beaver triples for the host. All the proposed operations, including element-wise multiplication, matrix multiplication, convolution, and bitwise AND, are implemented in the secure chip to support the full functionality of the protocol.

6.1 Host Software

The host software of SwiftMPC is designed to support **C3** by allowing machine-learning models to run with minimal code changes while decomposing high-level layers into MPC-compatible operators. Built on top of PyTorch, SwiftMPC provides its own library that mirrors PyTorch's API. Users can directly load a PyTorch-exported model into SwiftMPC and run training or inference under MPC. In total, approximately ten lines of code need to be modified to migrate from plaintext execution to secure MPC execution. These modest software changes also address **C4** by reducing the effort required to adopt SwiftMPC.

When handling the machine-learning workload, layers in the workload (such as linear, convolution, and ReLU) are decomposed into the minimal operators under MPC. For example, ReLU with input x is evaluated as $x \times (x \geq 0)$, and this comparison is further computed by converting x to binary shares and testing the highest bit of x.

In this procedure, certain operators such as addition and subtraction can be performed locally by each party without Beaver triples. Other operators that require a Beaver-based multiplication (such as convolution, element-wise multiplication, and bitwise AND) are executed jointly between two parties, which is illustrated by the blue boxes in Figure 2. For non-arithmetic operators, such as comparison in ReLU (greater-than, less-than) or bitwise operators (shift, AND, OR, XOR), SwiftMPC first converts arithmetic shared tensors into binary shared tensors, where combining the two shares with XOR reveals the plaintext. Bitwise operators are performed on these binary shared tensors, which are then converted back to arithmetic shared tensors.

Beaver-Based Operators. For these Beaver-based operators, SwiftMPC uses the proposed protocol in Section 5. First, the secure chip and the other participant need to safely establish a shared random seed through a secure authenticated channel, as described in Section 5.2. After that, the participant without a secure chip can use a pseudorandom generator with the same seed as the chip to produce the random values as Beaver triples. We adopt ChaCha20 [9] as the stream cipher for random number generation, due to its high performance across various platforms and within secure chips. The participant holding the secure chip transfers matrix data via XDMA [1] to the chip and invokes the interfaces of the computing units to generate part of the Beaver triple, which is then read back from the chip. Finally, the participant computes the result share using the input share, Beaver triple and differences. A detailed procedure inside the secure chip is described in the next subsection.

The participant without the secure chip gets the triples from PRNG, computes the result share and sends differences

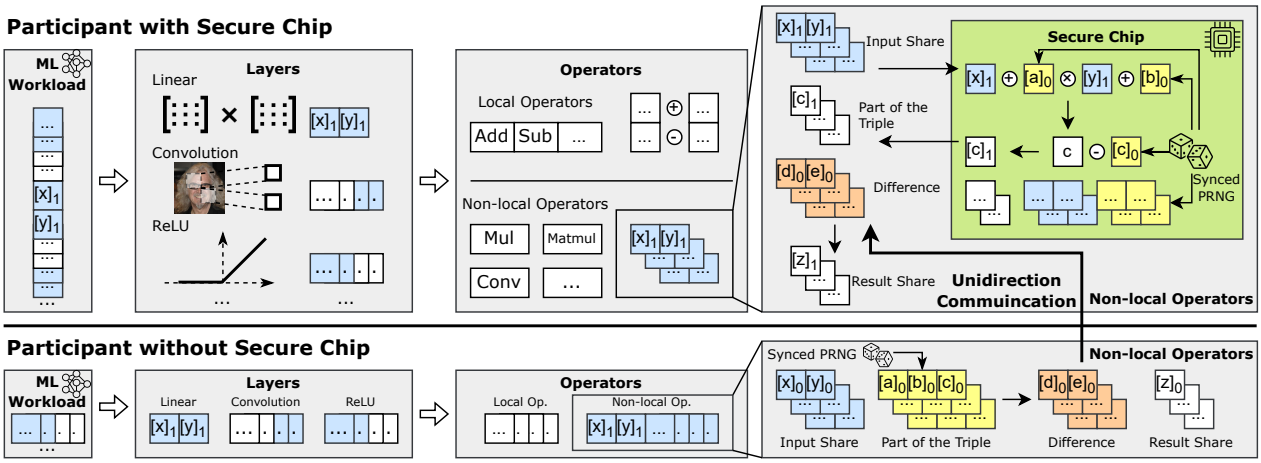


Figure 2. SwiftMPC overall architecture. Machine-learning workloads are decomposed into intermediate layers and MPC operators. Local operators are executed directly, while non-local operators are evaluated using the SwiftMPC protocol. The participant holding the secure chip invokes the secure chip to generate Beaver triple components, while the other participant supplies the differences through unidirectional communication.

d, e to the other participant through a socket connection. In the original Beaver protocol, after computing $([d], [e])$, both parties must exchange these values to reveal the actual values, requiring the parties to wait for communication to complete before continuing. In SwiftMPC, the protocol keeps the inter-party network communication unidirectional, eliminating the need to wait.

When operators involve matrices of different sizes, broadcasting may be required during computation. However, only the pre-broadcast matrix data is transmitted, and the other participant performs broadcasting locally after reception. This design further reduces communication cost. Moreover, the kernel network stack buffer also acts as a suitable parallelization mechanism here. Essentially, sending or receiving data is just copying data between the kernel and SwiftMPC. While the kernel is sending or receiving data, the participant without the secure chip can continue with subsequent computations, and the other participant can also perform computations before they actually need the data. Finally, both participants compute the result share using the triple and differences, completing our protocol.

6.2 Secure Chip Prototype

SwiftMPC relies on a secure chip to generate the Beaver triples required for MPC computation. We implemented such a prototype device on an FPGA. The hardware structure, as illustrated in Figure 3, mainly consists of the XDMA module, a random number generator and its FIFO, computing units, and triple buffer. The XDMA module is a Xilinx IP that maps an AXI bus address space on the FPGA to a character device file on the host. Both the computing units and triple buffer are connected to the XDMA module via an AXI bridge, so

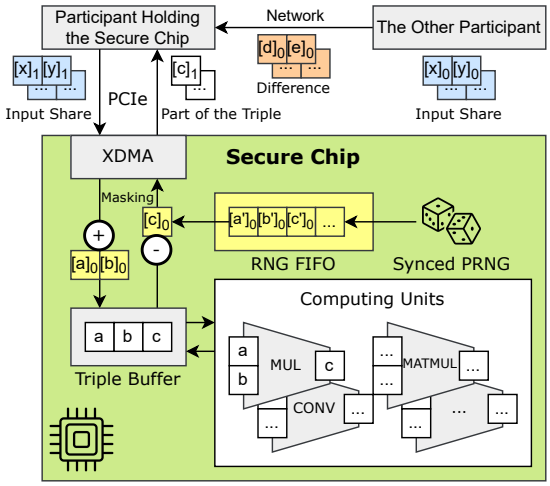


Figure 3. SwiftMPC hardware architecture. Input shares from the host and output triple shares to the host are masked by PRNG-generated triples on the fly, while the computing units generate triple components for element-wise, matrix, and convolution operators through a highly parallel design.

host read/write operations to the device under `/dev` correspond to transferring data into triple buffer or controlling the computing units.

When the secure chip is handling the SwiftMPC protocol, additions, subtractions, and XORs of Beaver triples and data shares are performed on-the-fly as data is transferred between XDMA module and triple buffer. The chip integrates a

random number generator that produces random values into a FIFO at a rate faster than data transmission. During data transfers, enabling the triple buffer also triggers consumption of one random value from the FIFO, which masks the input data. Since all masking control and computation are pipelined, it can be executed without degrading performance of data transfer.

For the triple buffer, we configure it as two dual-port RAMs. The XDMA module accesses them through 128-bit ports, while the computing units access them through 4096-bit wide ports. This design maximizes parallelism, as each data access by the computing units can simultaneously access two groups, each containing up to 64 elements (64-bit each). Because Beaver triple generation requires multiplication and convolution operations, large-scale data can only achieve high performance through parallel execution.

To support the diverse arithmetic patterns required by SwiftMPC, several dedicated computing units are implemented inside the secure chip. These units optimized for matrix operations further support **C3** by accelerating Beaver-triple generation for these critical computations in machine learning.

- **Element-wise multiplication and AND.** These two operations are straightforward to parallelize. The FPGA reads two groups of 64 elements from triple buffer and processes them in parallel, achieving up to 64 operations per cycle.
- **Matrix multiplication.** The B matrix is transposed before being written into triple buffer, enabling each triple buffer access to load 64 elements from a row of A and 64 elements from a column of B, which are then multiplied and accumulated in parallel to obtain the result.
- **Convolution.** Two variants of the operation are designed. In one mode, parallelism is applied across input channels: A provides up to 64 elements from different input channels at the same matrix location, which are multiplied with a single element in kernel B. In the other, parallelism is applied across output channels: B provides up to 64 kernel elements from different output channels at the same location, which are multiplied with a single input element from A. Depending on whether input or output channels dominate, the host selects the corresponding mode for the computing unit. To support this, the host also preprocesses tensors via the `permute` operation so that the parallel dimension is aligned as the last dimension before transfer into triple buffer.

6.3 Other implementation forms of SwiftMPC

Secure chips can also be implemented in other forms, including TEE enclaves, portable USB keys, or embedded modules. Since the secure chip requires authentication, shared seed establishment between participants, and protection of intermediate results, it can also be implemented in a TEE enclave. We implement SwiftMPC in Intel SGX by running the protocol purely in software inside an SGX enclave. Compared with this SGX-based implementation, our hardware design achieves much higher performance and also offers plug-and-play deployment even on platforms without a usable TEE. A detailed comparison between our hardware design and the TEE-based implementation is provided in Section 7.2.

Implementing our current prototype on a PCIe-based FPGA card does not contradict our goal of developing a portable secure chip device. In our implementation, the PCIe 2.0 ×8 interface has a maximum bandwidth of 4 GB/s, which is comparable to the bandwidth with emerging high-speed interfaces such as USB4 (up to 40 Gbps). When such interfaces become widely adopted in portable hardware, a compact USB-based secure chip can deliver performance equivalent to our current FPGA-based prototype. Therefore, the bandwidth requirement of SwiftMPC does not pose a barrier to realizing a portable secure hardware device.

Host–chip bandwidth directly affects performance in such standalone deployments. Our bottleneck analysis (§7.3) shows that PCIe communication between the chip and host accounts for about 20–40% of the execution time on most workloads. This overhead can be reduced through architecture-level integration, such as extending the instruction set to natively support MPC operations or integrating the secure module directly into CPU cores, which lowers host–chip communication latency and raises interaction throughput.

6.4 Secure Chip Security Analysis

Like other systems with a hardware root of trust, SwiftMPC relies on the secure chip to preserve the confidentiality and integrity of its internal state. If the chip is fully compromised and its internal secrets are extracted, the privacy may no longer hold. Therefore, we design the secure chip to minimize its attack surface as much as possible. The secure chip implements a fixed-function datapath with a restricted interface. Compared with a more general TEE-style execution environment, this implementation yields a narrower side-channel surface and a smaller amount of security-critical state inside the chip. In our FPGA prototype, intermediate values are kept in on-chip RAMs and registers instead of off-chip DRAMs. The synchronized seed is used on-chip as a starting PRNG state, rather than being repeatedly stored or transmitted as a static secret. In addition, because the PRNG in our implementation is instantiated with ChaCha20 [9], exposure of the current internal state does not automatically reveal previously consumed pseudorandom values. Thus, even though a full chip compromise breaks the trust assumption for ongoing computation, it does not necessarily imply retrospective disclosure of all past masks.

7 Evaluation

7.1 Experimental Setup

We implement and test SwiftMPC in a real system environment. The host machine is equipped with an AMD Ryzen 9700X CPU and 32 GB of memory, running Linux kernel version 6.14.0. The secure chip prototype is implemented on a Xilinx XC7K480T FPGA operating at 250MHz for bus and 125MHz for computing units. All computing units are implemented using HLS. Two convolution variants employ 8-way and 32-way parallelism, respectively. The matrix-multiplication, element-wise multiplication, and AND modules all employ 32-way parallelism. These modules share a large portion of DSPs, so the overall DSP utilization is lower than the sum of individual module utilizations. The remaining logic, including the XDMA interface and dataflow management, is implemented in Verilog and block diagrams. The whole design uses 111,866 LUTs (37%), 184,584 flip-flops (31%), 391 DSPs (20%) and 824.5 BRAMs (86%).

We additionally implement a SGX-based realization of the secure chip for SwiftMPC on a server with Intel Xeon Gold 6354 CPU, so as to better evaluate the impact of different implementation forms on the performance of SwiftMPC. In the following experiments, this SGX-based realization is denoted as SwiftMPC-SGX, while SwiftMPC represents our hardware-optimized design using FPGA.

Baselines. For baselines, we compare SwiftMPC against two state-of-the-art MPC systems on machine learning, namely CrypTen [26] and PPMLAC [57]. CrypTen adopts additive secret sharing with Beaver triples and is tightly integrated with PyTorch, thereby benefiting from GPU acceleration. We therefore run CrypTen on a server with AMD EPYC 7763 CPU and NVIDIA A10 GPU.

Network Latency. We test two-party collaborative MPC workloads under three different RTT (Round-Trip Time) delays:

- **Latency of 0.3 ms:** Simulating the case where both parties are located within the same data center with minimal network delay. Although in practice two MPC parties rarely colocate, it shows the maximum achievable performance of the tested framework, where the influence of network latency is minimized and the computational capability of the framework is more important.
- **Latency of 65 ms:** Simulating a cross-data-center deployment by adding a 65 ms RTT delay between the two parties. This reflects a realistic wide-area setting for collaborative computation across geographically separated organizations [34].
- **Latency of 200 ms:** Simulating an intercontinental deployment by adding a 200 ms RTT delay between the two parties. This setting is intended to simulate realistic long-distance collaboration through intercontinental networks [34].

Workloads. We evaluate the performance of SwiftMPC on both basic operators and representative machine-learning workloads. The basic operators include element-wise multiplication (MUL), comparison (COMP), and bitwise AND (AND) on randomly generated vectors with 10 million operations each, using freshly generated Beaver triples. All arithmetic is executed on 64-bit integers, and floating-point values are converted to fixed-point integers. For machine-learning workloads, we conduct four comprehensive tests, adopting the same configurations as PPMLAC for the LR, MLP, and ResNet benchmarks.

- **LR:** A logistic regression model with a 10^6-dimensional input and a sigmoid layer, processing 24 samples per batch during inference.
- **MLP:** A multilayer perceptron consisting of five linear layers (with dimensions 1024×512, 512×256, 256×128, 128×64, 64×32) and four ReLU layers between them. Each inference processes 24 samples per batch.
- **ResNet:** Inference for the ResNet-18 [23] model with input size 224×224, which features multiple convolutional and residual modules and is one of the most widely used CNN architectures for image classification.
- **Transformer:** To demonstrate better compatibility with LLM-style architectures, we also perform experiments on Transformer [50] models. The model can be modified to make it more suitable for MPC [40], but we still use the standard Transformer architecture to better align with our goal of requiring minimal program changes. To keep the evaluation computationally tractable, we use two compact Transformer configurations with different parameter scales, denoted as TFM-S and TFM-M in the following.

7.2 Performance

Basic Operators. As shown in Table 1, at 0.3 ms latency, SwiftMPC achieves up to 1.6× the performance of PPMLAC. These results demonstrate the efficiency of SwiftMPC in performing basic MPC operators. At 65 ms latency, SwiftMPC achieves up to 1.9× the performance of PPMLAC and 17× the performance of CrypTen. At 200 ms latency, SwiftMPC still achieves up to 1.9× the performance of PPMLAC and 46× the performance of CrypTen, confirming that its performance remains robust even under high network latency.

Compared with the SwiftMPC-SGX, the SwiftMPC using our FPGA prototype is 17× faster on average, demonstrating the performance gains enabled by our hardware design that is specifically optimized for the SwiftMPC protocol, and showing that this approach is actually more efficient than TEE-based implementations.

Comparing each workload across the latency settings in Table 1, the advantage of unidirectional communication becomes clear. Among these methods, only CrypTen relies on bidirectional communication, and its performance degrades substantially as network latency increases. In contrast, both

Table 1. Execution time (in seconds) under different network round-trip times (RTT, in ms) for basic operators and representative machine-learning workloads. Each entry is followed by its runtime ratio relative to SwiftMPC under the same workload and RTT. "Swift" and "Swift-SGX" denote SwiftMPC and SwiftMPC-SGX, respectively. Timing spans from the first party starting computation to the other party finishing, excluding initialization and data preprocessing.

Work load	RTT	**Swift**	Swift-SGX	PPMLAC	CrypTen
MUL	0.3	0.22	4.17 (19×)	0.35 (1.6×)	**0.13** (0.6×)
	65	**0.24**	4.18 (17×)	0.42 (1.8×)	3.42 (14×)
	200	**0.30**	4.23 (14×)	0.56 (1.9×)	10.8 (36×)
COMP	0.3	1.51	27.5 (18×)	2.11 (1.4×)	**0.90** (0.6×)
	65	**1.54**	28.0 (18×)	2.17 (1.4×)	22.6 (15×)
	200	**1.56**	28.2 (18×)	2.32 (1.5×)	71.5 (46×)
AND	0.3	0.20	4.13 (21×)	0.30 (1.5×)	**0.12** (0.6×)
	65	**0.22**	4.15 (19×)	0.41 (1.9×)	3.71 (17×)
	200	**0.31**	4.21 (14×)	0.55 (1.8×)	10.8 (35×)
MLP	0.3	0.03	0.31 (10×)	**0.02** (0.7×)	0.73 (24×)
	65	0.10	0.37 (3.7×)	**0.09** (0.9×)	5.55 (55×)
	200	**0.19**	0.41 (2.2×)	0.24 (1.3×)	15.8 (83×)
LR	0.3	**0.33**	4.43 (13×)	0.40 (1.2×)	2.05 (6.2×)
	65	**0.34**	4.50 (13×)	0.46 (1.4×)	20.6 (61×)
	200	**0.39**	4.70 (12×)	0.65 (1.7×)	62.2 (159×)
ResNet	0.3	**2.21**	43.4 (20×)	3.54 (1.6×)	3.64 (1.6×)
	65	**2.22**	43.5 (20×)	3.64 (1.6×)	63.3 (29×)
	200	**2.25**	43.6 (19×)	3.86 (1.7×)	193 (86×)
TFM-S	0.3	**3.79**	40.8 (11×)	-	15.9 (4.2×)
	65	**3.81**	41.2 (11×)	-	477 (125×)
	200	**3.89**	41.7 (11×)	-	1423 (366×)
TFM-M	0.3	**12.2**	168 (14×)	-	34.0 (2.8×)
	65	**12.2**	169 (14×)	-	957 (78×)
	200	**12.3**	169 (14×)	-	2876 (233×)

SwiftMPC and PPMLAC employ unidirectional communication, and therefore exhibit only minor increases in execution time as latency rises.

Machine Learning and Neural Network. At 0.3 ms latency, SwiftMPC achieves up to 1.6× the performance of PPMLAC and 24× the performance of CrypTen, demonstrating the efficiency of both the SwiftMPC protocol and its implementation in computation-intensive machine-learning workloads. At 65 ms latency, SwiftMPC achieves up to 1.6× the performance of PPMLAC and 125× the performance of CrypTen, validating our claim that it is less sensitive to network latency. At 200 ms latency, SwiftMPC still achieves up to 1.7× the performance of PPMLAC and 366× the performance of CrypTen. On average, SwiftMPC is 11× faster than SwiftMPC-SGX, which is consistent with the results observed in the basic operators. Given that these workloads resemble practical applications, SwiftMPC shows potential for real-world tasks such as financial or payment verification systems, where results are required within a few seconds.

It should be noted that, due to hardware limitations, the FPGA prototype operates under PCIe 2.0 x8 (with 4 GB/s maximum bandwidth), which represents a relatively low throughput requirement compared to PPMLAC and CrypTen. Using devices with higher bandwidth would further improve the performance. A detailed analysis is provided in the next subsection.

7.3 Performance Breakdown and Bottleneck Analysis

We further decompose the total execution time of the participant holding the secure chip into separate components

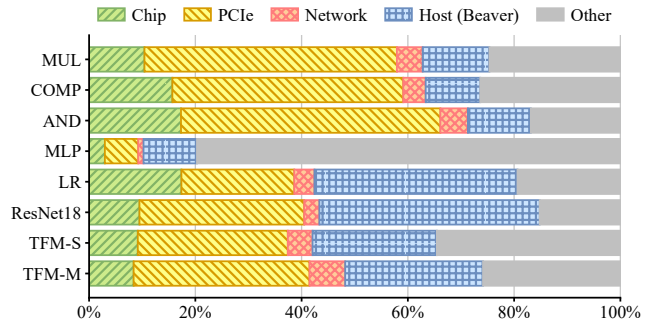


Figure 4. Proportion of time spent on different operations in various workloads under 65ms network latency.

to identify opportunities for further optimization. The network latency is set to 65 ms. During the whole execution, four kinds of operations are related to Beaver computation, including secure-chip computation for generating the triples (Chip), PCIe communication between the secure chip and host (PCIe), network communication between participants (Network), and host computation on shares and triples to generate results (Host). "Other" represents processes other than Beaver computation, including computations that can be performed locally and the execution overhead of the framework itself.

As listed in Figure 4, the "Chip" component accounts for less than 20% of the total execution time, leaving relatively limited room for further optimization on the secure chip side. For workloads other than MLP, the PCIe communication between the secure chip and host incurs a relatively significant

Table 2. Network transmission volume in MB for SwiftMPC and baseline frameworks across representative workloads.

Model	LR	MLP	ResNet-18	TFM-S	TFM-M
SwiftMPC	**191**	9.9	**814**	**1263**	**5200**
CrypTen	**191**	**8.16**	993	1321	5431
PPMLAC	210	10	921	-	-

overhead, accounting for about 20–40% of the execution time. Given the actual data transfer volume and the available PCIe bandwidth, this overhead is clearly higher than optimal conditions, indicating further opportunities to optimize chip-host data transfer. Since our design employs unidirectional communication, network transmission contributes only a small fraction of the time. Host-side computations including the "Host" and "Other" components account for around 40–60% of the execution time. The "Other" component remains relatively small across most workloads, indicating that Beaver computations are actually the dominant bottleneck in MPC execution. Further optimization of MPC workloads should continue to focus primarily on Beaver computation and its associated data movement. The overall result shows that SwiftMPC effectively handles the Beaver computations, and suggests that future performance gains should focus on improving the chip-host communication and host-side processing efficiency.

For shared seed establishment, our simulation measures a total initialization time of 1.35 ms from initiating TLS session establishment to deriving the shared PRNG seed. This initialization is required only once per session, and its cost can be amortized across executions of different models within the same session. Therefore, the initialization overhead would have a negligible impact on the reported performance benefits.

7.4 Network Transmission Volume

Since SwiftMPC generates and computes Beaver triples in a matrix form, both the total number of Beaver triples required and the amount of data transmitted between participants are reduced compared with scalar-level triple generation for each multiplication in matrix or convolution operations (see §5 for analysis). As shown in Table 2, SwiftMPC achieves lower or comparable network transmission volume across the evaluated workloads. The reduction is most visible on the larger models, where SwiftMPC transmits 814 MB on ResNet-18 versus 993 MB for CrypTen and 921 MB for PPMLAC, and stays consistently lower on both transformer workloads. The exception is MLP, whose layers involve small matrices for which matrix-granularity triple generation offers little advantage. Compared with PPMLAC, which sometimes requires manual annotation to cache matrix shares, SwiftMPC achieves generally better results without any manual labeling, since generating Beaver triples at the scale of the matrix itself already provides good caching behavior. For repeated inference on the same model, caching the Beaver triples for the weight matrices could further reduce triple generation and communication overhead, which we leave as a possible optimization.

8 Related work

Trusted Hardware Assisted MPC. A line of work explores using trusted hardware to accelerate MPC by offloading cryptographic preprocessing. Demmler et al. [14] extend the GMW protocol with a trusted hardware token (e.g., TPM), enabling offline preprocessing that completes within seconds, demonstrating the benefits of small trusted devices for mobile MPC. HPCG [16] similarly uses lightweight TEE chips exclusively in the offline phase to reduce online latency while remaining compatible with standard secret-sharing semantics. Chameleon [43] replaces ABY's OT-based offline phase with a Semi-honest Third Party for seed expansion, but retains bidirectional communication in its online phase. PPMLAC [57] proposes a hardware chipset that implements secret-sharing-based MPC with minimal trusted hardware, eliminating communication bottlenecks to achieve several orders of magnitude speedup over software-only MPC. However, it computes directly on plaintext within the hardware, resembling a TEE-style design. HYBRTC [54] integrates TEEs with MPC to form a hybrid-trust abstraction that adapts to varied trust levels. It demonstrates secure distributed SQL operations and achieves malicious-security guarantees. HAAC [36] targets garbled circuits and introduces a co-designed compiler, ISA, and microarchitecture that dramatically accelerates GC evaluation. By balancing hardware-software responsibilities, it introduces a high-throughput gate engine, tailored memory structures, and reuse-aware optimizations.

These works demonstrate that trusted hardware can improve MPC performance, but they generally require mutual communication or assume that every participant possesses a trusted device. In contrast, SwiftMPC adopts unidirectional communication to reduce the impact of network latency and requires only a single secure chip for the full system.

MPC Frameworks. Several MPC frameworks aim to make secure computation practical by abstracting cryptographic operations. MP-SPDZ [25] provides over 30 protocol variants under a unified Python interface. FALCON [52] proposes an honest-majority, maliciously secure three-party framework for private deep learning that improves efficiency by designing arithmetic-secret-sharing-based protocols for nonlinear neural network operations, enabling private training and inference of large models. SPU/SecretFlow [30] presents a performant and user-friendly privacy-preserving machine

learning framework that compiles programs from mainstream machine-learning frameworks into MPC execution, enabling efficient private training with only minor code modifications. CrypTen [26] provides a user-friendly framework for machine learning under MPC, offering PyTorch-like abstractions and GPU acceleration. Inspired by these design, SwiftMPC also provides a user-friendly framework for machine learning under MPC while targeting stronger performance by addressing the communication bottleneck.

Other Privacy-Preserving Machine Learning Methods. Beyond secret-sharing MPC, homomorphic encryption (HE) occupies a different point in the computation-communication tradeoff. Cheddar [11] accelerates CKKS-based FHE on GPUs, reducing encrypted ResNet-20 (CIFAR-10) inference to 0.72 s on an RTX 5090. CHOCO-TACO [48] is a client-aided HE approach where a resource-constrained client repeatedly decrypts, applies non-linear operations, and re-encrypts with an untrusted server between layers. It uses a dedicated ASIC to address the resulting encryption/decryption bottleneck and employs algorithmic optimizations that reduce ciphertext size. SwiftMPC instead targets the mutual communication overhead of Beaver multiplication, addressing an orthogonal bottleneck from a different computation model.

9 Discussion

Possible Mitigations against Side-Channel and Physical Attacks. Although side-channel attacks and physical tampering are outside our threat model, SwiftMPC allows conventional countermeasures to be integrated without changing the protocol. For side-channel attacks [4, 38], prior research has proposed mitigations such as hiding and masking [31]. Hiding [21, 27, 28] reduces the information leaked to attackers, while masking [17, 41, 44] makes the leaked signals less dependent on the original data. For physical attacks [51], representative mechanisms include tamper-detecting conductive meshes [3, 49], clock/voltage monitoring [20, 35], and physical-unclonable-function-based verification [18, 22]. These techniques can be incorporated as implementation-level hardening without changing the SwiftMPC protocol.

Although our prototype does not implement these mitigations, they need not negate the performance benefits of SwiftMPC. As shown in Section 7.2, secure-chip computation accounts for less than 20% of the total execution time in our prototype. Therefore, even if chip-side protections introduce additional resource or latency overheads, their effect on end-to-end runtime is likely to remain limited compared with host-side computation and chip-host communication costs.

Generalization and protocol choice. The broader idea behind SwiftMPC is to use simple, fixed-function trusted hardware to simplify complex and time-consuming parts of cryptographic protocols. This principle could potentially extend to other protocols to simplify expensive cryptographic operations or reduce communication between participants. For example, a trusted chip might assist with oblivious-transfer correlation generation in a garbled-circuit-based design. Using a seed shared with the sender, the chip could generate pairs of random values locally and release only the chosen value of the receiver per transfer. Such a construction would require determining which secrets the chip holds, which outputs it exposes, and whether the resulting protocol actually reduces computation or communication costs.

We choose secret sharing because it fits both our numerical ML workloads and our goal of keeping trusted hardware simple. Arithmetic shares preserve the tensor representation of ML data, allowing matrix multiplication and convolution to use standard numerical kernels on the hosts. In our design, the chip handles only pseudorandomness and triple generation through a small interface that is independent of the model structure. This makes secret sharing a suitable starting point for hardware-assisted MPC protocols with a simple trusted component.

10 Conclusion

SwiftMPC builds on a simple protocol insight: if one party sets its first two Beaver-triple components equal to its existing input shares, that party's masked differences vanish and secure multiplication reduces from bidirectional to one-way communication, letting communication and computation be pipelined. Such input-bound triples cannot be precomputed cryptographically nor supplied by a trusted third party without reintroducing a round trip, so SwiftMPC generates them with a secure chip attached to one participant, a narrow trust anchor in place of an external trusted party. Our implementation and FPGA prototype show that this design yields significant acceleration on machine-learning workloads with only a single secure chip and minimal code changes. Notably, SwiftMPC's performance remains nearly flat as network latency grows, confirming that the one-way communication structure effectively decouples execution time from network conditions. These results highlight a viable architectural path toward fast and deployable MPC systems.

Acknowledgments

We thank the anonymous reviewers and our shepherd for their constructive feedback and insightful suggestions, which significantly improved the quality of this paper. This research was supported by the National Natural Science Foundation of China (Grant No. 62302441) and the Alibaba Cloud. This work was partially supported by the Shandong Provincial Higher Education Institutions Youth Innovation Team Development Plan (No. 2025KJH023) and the ZJU Kunpeng & Ascend Center of Excellence. We also gratefully acknowledge the support of the Qizhen Scholar Foundation of the Zhejiang University Education Foundation.

References

[1] AMD. 2025. DMA/Bridge Subsystem for PCI Express Product Guide (PG195).

[2] AMD. 2025. *Spartan UltraScale+ FPGAs Configuration User Guide (UG860).* Technical Report.

[3] Naveed Anjum, Saifullah Hammad, and Muhammad Faheem. 2018. Passive Design Technique for Tamper-Resistant Embedded System. In *2018 15th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*. 427–430. doi:10.1109/IBCAST.2018.8312259

[4] Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. 2019. Poster: Recovering the Input of Neural Networks via Single Shot Side-Channel Attacks. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. Association for Computing Machinery, New York, NY, USA, 2657–2659. doi:10.1145/3319535.3363280

[5] Donald Beaver. 1991. Secure Multiparty Protocols and Zero-Knowledge Proof Systems Tolerating a Faulty Minority. *Journal of Cryptology* 4, 2 (Jan. 1991), 75–122. doi:10.1007/BF00196771

[6] Donald Beaver. 1992. Efficient Multiparty Protocols Using Circuit Randomization. In *Advances in Cryptology — CRYPTO '91*, Joan Feigenbaum (Ed.). Springer, Berlin, Heidelberg, 420–432. doi:10.1007/3-540-46766-1_34

[7] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. 2012. Foundations of Garbled Circuits. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. Association for Computing Machinery, New York, NY, USA, 784–796. doi:10.1145/2382196.2382279

[8] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. 2011. Semi-Homomorphic Encryption and Multiparty Computation. In *Advances in Cryptology – EUROCRYPT 2011*, Kenneth G. Paterson (Ed.). Springer, Berlin, Heidelberg, 169–188. doi:10.1007/978-3-642-20465-4_11

[9] Daniel J Bernstein. 2008. ChaCha, a Variant of Salsa20. *Workshop Record of SASC 2008* (2008).

[10] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2020. TFHE: Fast Fully Homomorphic Encryption Over the Torus. *Journal of Cryptology* 33, 1 (Jan. 2020), 34–91. doi:10.1007/s00145-019-09319-x

[11] Wonseok Choi, Jongmin Kim, and Jung Ho Ahn. 2025. Cheddar: A Swift Fully Homomorphic Encryption Library Designed for GPU Architectures. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '26)*. Association for Computing Machinery, New York, NY, USA, 35–49. doi:10.1145/3760250.3762223

[12] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. 2018. SPDZ2k: Efficient MPC Mod 2^k for Dishonest Majority. In *Advances in Cryptology – CRYPTO 2018*, Hovav Shacham and Alexandra Boldyreva (Eds.). Springer International Publishing, Cham, 769–798. doi:10.1007/978-3-319-96881-0_26

[13] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. 2012. Multiparty Computation from Somewhat Homomorphic Encryption. In *Advances in Cryptology – CRYPTO 2012*, Reihaneh Safavi-Naini and Ran Canetti (Eds.). Springer, Berlin, Heidelberg, 643–662. doi:10.1007/978-3-642-32009-5_38

[14] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2014. {Ad-Hoc} Secure {Two-Party} Computation on Mobile Devices Using Hardware Tokens. In *23rd USENIX Security Symposium (USENIX Security 14)*. 893–908.

[15] Shlomi Dolev, Niv Gilboa, and Ximing Li. 2019. Accumulating Automata and Cascaded Equations Automata for Communicationless Information Theoretically Secure Multi-Party Computation. *Theoretical Computer Science* 795 (Nov. 2019), 81–99. doi:10.1016/j.tcs.2019.06.005

[16] Wentao Dong and Cong Wang. 2023. Poster: Towards Lightweight TEE-Assisted MPC. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23)*. Association for Computing Machinery, New York, NY, USA, 3609–3611. doi:10.1145/3576915.3624398

[17] Anuj Dubey, Rosario Cammarota, and Aydin Aysu. 2020. MaskedNet: The First Hardware Inference Engine Aiming Power Side-Channel Protection. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 197–208. doi:10.1109/HOST45689.2020.9300276

[18] Rainer Falk. 2014. Tamper Protection Device for Protecting a Field Device against Tampering.

[19] Craig Gentry. 2009. Fully Homomorphic Encryption Using Ideal Lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing (STOC '09)*. Association for Computing Machinery, New York, NY, USA, 169–178. doi:10.1145/1536414.1536440

[20] Dennis R. E. Gnad, Fabian Oboril, and Mehdi B. Tahoori. 2017. Voltage Drop-Based Fault Attacks on FPGAs Using Valid Bitstreams. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. 1–7. doi:10.23919/FPL.2017.8056840

[21] Tim Güneysu and Amir Moradi. 2011. Generic Side-Channel Countermeasures for Reconfigurable Devices. In *Cryptographic Hardware and Embedded Systems – CHES 2011*, Bart Preneel and Tsuyoshi Takagi (Eds.). Springer, Berlin, Heidelberg, 33–48. doi:10.1007/978-3-642-23951-9_3

[22] Afm Rezaul HASAN and Russell REZAIAN. 2016. Method and Apparatus for Tamper Detection.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. doi:10.1109/CVPR.2016.90

[24] Intel. 2023. *Intel Stratix 10 Device Security User Guide.* Technical Report.

[25] Marcel Keller. 2020. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20)*. Association for Computing Machinery, New York, NY, USA, 1575–1590. doi:10.1145/3372297.3417872

[26] Brian Knott, Shobha Venkataraman, Awni Hannun, Shubho Sengupta, Mark Ibrahim, and Laurens van der Maaten. 2021. CrypTen: Secure Multi-Party Computation Meets Machine Learning. In *Advances in Neural Information Processing Systems*, Vol. 34. Curran Associates, Inc., 4961–4973.

[27] Jonas Krautter, Dennis R.E. Gnad, Falk Schellenberg, Amir Moradi, and Mehdi B. Tahoori. 2019. Active Fences against Voltage-Based Side Channels in Multi-Tenant FPGAs. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. doi:10.1109/ICCAD45719.2019.8942094

[28] Adrien Le Masle, Gary C. T. Chow, and Wayne Luk. 2011. Constant Power Reconfigurable Computing. In *2011 International Conference on Field-Programmable Technology*. 1–8. doi:10.1109/FPT.2011.6132682

[29] Ruiyang Li, Yiteng Sun, Chun Guo, François-Xavier Standaert, Weijia Wang, and Xiao Wang. 2024. Leakage-Resilient Circuit Garbling. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security (CCS '24)*. Association for Computing Machinery, New York, NY, USA, 780–794. doi:10.1145/3658644.3690204

[30] Junming Ma, Yancheng Zheng, Jun Feng, Derun Zhao, Haoqi Wu, Wenjing Fang, Jin Tan, Chaofan Yu, Benyu Zhang, and Lei Wang. 2023. {SecretFlow-SPU}: A Performant and {User-Friendly} Framework for {Privacy-Preserving} Machine Learning. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 17–33.

[31] Dina G. Mahmoud, Vincent Lenders, and Mirjana Stojilović. 2022. Electrical-Level Attacks on CPUs, FPGAs, and GPUs: Survey and Implications in the Heterogeneous Era. *Comput. Surveys* 55, 3 (Feb. 2022), 58:1–58:40. doi:10.1145/3498337

[32] Marcel Keller, Emmanuela Orsini, and Peter Scholl. 2016. MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer.

[33] Silvio Micali, Oded Goldreich, and Avi Wigderson. 1987. How to Play Any Mental Game. In *Proceedings of the Nineteenth ACM Symp. on Theory of Computing, STOC*. ACM New York, 218–229.

[34] Microsoft. 2025. Azure Network Round-Trip Latency Statistics. https://learn.microsoft.com/en-us/azure/networking/azure-network-latency.

[35] Seyedeh Sharareh Mirzargar, Gaiëtan Renault, Andrea Guerrieri, and Mirjana Stojilović. 2020. Nonintrusive and Adaptive Monitoring for Locating Voltage Attacks in Virtualized FPGAs. In *2020 International Conference on Field-Programmable Technology (ICFPT)*. 288–289. doi:10.1109/ICFPT51103.2020.00050

[36] Jianqiao Mo, Jayanth Gopinath, and Brandon Reagen. 2023. HAAC: A Hardware-Software Co-Design to Accelerate Garbled Circuits. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23)*. Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/3579371.3589045

[37] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *2017 IEEE Symposium on Security and Privacy (SP)*. 19–38. doi:10.1109/SP.2017.12

[38] Jungmin Park, Xiaolin Xu, Yier Jin, Domenic Forte, and Mark Tehranipoor. 2018. Power-Based Side-Channel Instruction-Level Disassembler. In *Proceedings of the 55th Annual Design Automation Conference (DAC '18)*. Association for Computing Machinery, New York, NY, USA, 1–6. doi:10.1145/3195970.3196094

[39] H. U. Peng, S. U. N. Lei, H. U. Cuiyun, G. U. O. Song, Wang Jingwen, Wang Zhihong, and Y. a. O. Jingyi. 2025. Survey of Research on Efficient Secure Inference for Deep Learning. *Tongxin xuebao* 46 (May 2025), 238–257.

[40] Deevashwer Rathee, Dacheng Li, Ion Stoica, Hao Zhang, and Raluca Popa. 2024. MPC-Minimized Secure LLM Inference. arXiv:2408.03561 doi:10.48550/arXiv.2408.03561

[41] Francesco Regazzoni, Yi Wang, and François-Xavier Standaert. 2011. FPGA Implementations of the AES Masked against Power Analysis Attacks. In *Proceedings of COSADE*, Vol. 2011. 56–66.

[42] E. Rescorla. 2026. The Transport Layer Security (TLS) Protocol Version 1.3. rfc9846 (July 2026). doi:10.17487/RFC9846

[43] 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 (ASIACCS '18)*. Association for Computing Machinery, New York, NY, USA, 707–721. doi:10.1145/3196494.3196522

[44] Pascal Sasdrich, Amir Moradi, Oliver Mischke, and Tim Güneysu. 2015. Achieving Side-Channel Protection with Dynamic Logic Reconfiguration on Modern FPGAs. In *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 130–136. doi:10.1109/HST.2015.7140251

[45] Adi Shamir. 1979. How to Share a Secret. *Commun. ACM* 22, 11 (Nov. 1979), 612–613. doi:10.1145/359168.359176

[46] Nigel P. Smart and Titouan Tanguy. 2019. TaaS: Commodity MPC via Triples-as-a-Service. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop (CCSW'19)*. Association for Computing Machinery, New York, NY, USA, 105–116. doi:10.1145/3338466.3358918

[47] Shamsher Ullah, Jianqiang Li, Jie Chen, Ikram Ali, Salabat Khan, Muhammad Tanveer Hussain, Farhan Ullah, and Victor C. M. Leung. 2025. Homomorphic Encryption Applications for IoT and Light-Weighted Environments: A Review. *IEEE Internet of Things Journal* 12, 2 (Jan. 2025), 1222–1246. doi:10.1109/JIOT.2024.3472029

[48] McKenzie van der Hagen and Brandon Lucia. 2022. Client-Optimized Algorithms and Acceleration for Encrypted Compute Offloading. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 683–696. doi:10.1145/3503222.3507737

[49] Daniel-Ciprian Vasile, Sorin Chițu, and Paul Svasta. 2020. Cryptographic Key Derivation from an Anti-Tamper Solution. In *2020 IEEE 8th Electronics System-Integration Technology Conference (ESTC)*. 1–6. doi:10.1109/ESTC48849.2020.9229844

[50] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. In *Advances in Neural Information Processing Systems*, Vol. 30. Curran Associates, Inc.

[51] Marin Vidaković and Davor Vinko. 2023. Hardware-Based Methods for Electronic Device Protection against Invasive and Non-Invasive Attacks. *Electronics* 12, 21 (Jan. 2023), 4507. doi:10.3390/electronics12214507

[52] 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:2004.02229 doi:10.48550/arXiv.2004.02229

[53] Avi Wigderson, M. B. Or, and S. Goldwasser. 1988. Completeness Theorems for Noncryptographic Fault-Tolerant Distributed Computations. In *Proceedings of the 20th Annual Symposium on the Theory of Computing (STOC'88)*. 1–10.

[54] Pengfei Wu, Jianting Ning, Jiamin Shen, Hongbing Wang, and Ee-Chien Chang. 2022. Hybrid Trust Multi-Party Computation with Trusted Execution Environment.. In *NDSS*.

[55] Chuan Zhao, Shengnan Zhao, Minghao Zhao, Zhenxiang Chen, Chong-Zhi Gao, Hongwei Li, and Yu-an Tan. 2019. Secure Multi-Party Computation: Theory, Practice and Applications. *Information Sciences* 476 (Feb. 2019), 357–372. doi:10.1016/j.ins.2018.10.024

[56] Mark Zhao, Mingyu Gao, and Christos Kozyrakis. 2022. ShEF: Shielded Enclaves for Cloud FPGAs. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2022)*. Association for Computing Machinery, New York, NY, USA, 1070–1085. doi:10.1145/3503222.3507733

[57] 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 (ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 87–101. doi:10.1145/3470496.3527392

[58] Huafei Zhu and Wee Keong Ng. 2022. Highly Scalable Beaver Triple Generator from Additively Homomorphic Encryption. In *Advanced Information Networking and Applications*, Leonard Barolli, Farookh Hussain, and Tomoya Enokido (Eds.). Springer International Publishing, Cham, 504–514. doi:10.1007/978-3-030-99619-2_48