PPMLAC: High Performance Chipset Architecture for Secure Multi-Party Computation

Xing Zhou
zhouxing@msn.cn
Shanghai ZiXian Technology
Shanghai, China

Zhilei Xu*†
timxu@sjtu.edu.cn
Qing Yuan Research Institute, School of Electronic Information and Electrical Engineering
Shanghai Jiao Tong University
Shanghai, China

Cong Wang
wcon006@gmail.com
Shanghai ZiXian Technology
Shanghai, China

Mingyu Gao
gaomy@tsinghua.edu.cn
Institute for Interdisciplinary Information Sciences
Tsinghua University
Beijing, China

ABSTRACT

Privacy issue is a main concern restricting data sharing and cross-organization collaborations. While Privacy-Preserving Machine Learning techniques such as Multi-Party Computations (MPC), Homomorphic Encryption, and Federated Learning are proposed to solve this problem, no solution exists with both strong security and high performance to run large-scale, complex machine learning models. This paper presents PPMLAC, a novel chipset architecture to accelerate MPC, which combines MPC's strong security and hardware's high performance, eliminates the communication bottleneck from MPC, and achieves several orders of magnitudes speed up over software-based MPC. It is carefully designed to only rely on a minimum set of simple hardware components in the trusted domain, thus is robust against side-channel attacks and malicious adversaries. Our FPGA prototype can run mainstream large-scale ML models like ResNet in near real-time under a practical network environment with non-negligible latency, which is impossible for existing MPC solutions.

CCS CONCEPTS

• **Hardware → Hardware accelerators**; • **Security and privacy → Hardware-based security protocols**; **Privacy-preserving protocols**.

KEYWORDS

MPC, Secret Sharing, Security, Privacy, Privacy-preserving Machine Learning, Hardware Accelerator, Side-channel Protection

*Zhilei Xu is the corresponding author.
†Also with Shanghai ZiXian Technology.

ISCA '22, June 18–22, 2022, New York, NY, USA

ACM ISBN 978-1-4503-8610-4/22/06…$15.00
https://doi.org/10.1145/3470496.3527392

ACM Reference Format:
Xing Zhou, Zhilei Xu, Cong Wang, and Mingyu Gao. 2022. PPMLAC: High Performance Chipset Architecture for Secure Multi-Party Computation. In *The 49th Annual International Symposium on Computer Architecture (ISCA '22), June 18–22, 2022, New York, NY, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3470496.3527392

1 INTRODUCTION

Machine learning (ML) and big data technologies have profoundly changed the world, but also bring about unprecedented threats to people's privacy. *Privacy computing* are techniques that allow multiple parties to collectively compute the result of a function that depends on all parties' input while still keeping each party's own data *private* (not leaked to any other party), thus relieve people's concerns over privacy, and foster data collaborations cross organization boundaries [30, 49]. When applied to ML scenarios, privacy computing becomes *privacy-preserving machine learning* (PPML), which gains trending interest in both academia and industry [1, 2, 16, 46, 68]. Of the various privacy computing techniques, secure Multi-Party Computation (MPC) is known for its strong security guarantee and versatility, but its high performance overhead makes it unacceptable to many mainstream ML models that usually exhibit excessive computation and data amounts, thus thwarting the wider adoption of MPC in PPML use cases. In this paper, we present a novel chipset architecture called PPMLAC that performs MPC protocol without incurring its high communication and performance overhead. It combines MPC's high strength of security and hardware's high performance, and achieves orders of magnitudes speedup over software-based MPC implementations, resulting in an MPC system capable of practical ML tasks that are impossible for traditional PPML solutions. Our main contributions are:

- We invent a novel chipset architecture PPMLAC that runs secret-sharing based MPC protocol safely and efficiently. The efficiency improvement comes majorly from eliminating the communication bottleneck in MPC, so our FPGA prototype running at a low clock rate (150MHz) already achieves thousands of times speedup over traditional software MPC running on modern 40-core CPU at 2.5GHz, and tens to hundreds of times speedup over a state-of-the-art

MPC framework based on trusted third party running on NVidia V100 GPUs with 7 Trillion FLOPS performance. To our knowledge, PPMLAC is the first MPC solution that runs large neural networks like ResNet [40] within seconds under practical cross-data-center network latency.

- PPMLAC builds upon an interesting mix shift of hardware-based and algorithm-based security measures, which is novel. Like Trusted Execution Environment (TEE), we rely on chipset's non-modifiability to provide a trust root for software. But unlike TEE, our system's security comes majorly from the MPC protocol itself. The hardware-provided trust is only used for a few primitive operations such as multiplications, so the whole system exposes a much narrower attack surface, and is free from common side-channel attacks and safe against *dishonest majority* (even $n-1$ *colluding adversaries*).
- PPMLAC significantly relaxes both the network *latency* and *throughput* requirements. Most communications are removed from the critical path of the secure computation and can be done asynchronously, so PPMLAC does not require low-latency network. To its extreme, packing the messages into hard drives and sending them over a truck is doable [77]. This is in stark contrast to traditional MPC solutions that are extremely sensitive to network latency.
- To illustrate the flexibility of our new architecture, we have created two lines of prototype chips: a RISC-V CPU with extended instructions, and a specialized MPC accelerator. They are both FPGA products that runs real-world applications and ML models, and we evaluate the performance of the latter in §5.

We organize the rest of the paper as follows: §2 describes necessary background for our work, including general privacy computing and MPC concepts. §3 presents our key innovative ideas. §4 expounds the design and implementation of PPMLAC, and explains in detail how it performs MPC efficiently and securely. §5 evaluates PPMLAC's performance.

2 BACKGROUND

In this section, we first present the algorithm details of a typical MPC scheme based on additive secret sharing (§2.1, §2.2), and then compare MPC with alternative privacy-preserving computing primitives from the perspectives of system bottlenecks and hardware acceleration potentials (§2.3).

2.1 MPC: Additive Secret Sharing

A widely-used MPC scheme (e.g. [9, 24, 25, 46, 53, 60, 61, 90]) is *secret sharing*, and we particularly focus on *additive* secret sharing [25] here. We abstractly represent the function to compute as a *circuit* – a directed acyclic graph (DAG), with mainly three types of nodes: input ($Input_i$), addition (+), and multiplication ($*$). Other types of lightweight nodes, e.g. multiplied/added by a public number can also be used. Each node represents a value (input/output or intermediate) over a finite field (e.g. $\mathbb{F}_{2^{64}}$, the usual 64-bit integers), and the directed edges among nodes represent *data dependencies*. An input node $Input_i$ is the private input from party i. Without loss of generality, we assume each circuit has only one node for the final output. Fig. 1 shows an example circuit for a 2-party MPC. The circuit is topologically sorted (shown in Fig. 1 alphabetically), so computing the value of each node can be done sequentially. Addition and multiplication nodes are considered *functionally complete* in MPC schemes, and are used to approximate almost all functions using methods like Newton-Raphson [87]. In particular, most ML algorithms are dominated by additions and multiplications. We discuss non-linear operations in §4.3.

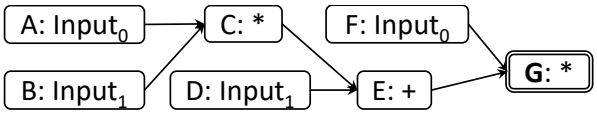


Figure 1: A simple example circuit for MPC computation with final output G.

The structure of the circuit is publicly known to all parties; however, the n parties must perform the computation while keeping *all* values secret, including the final output that is only *selectively revealed* to certain parties at the end. This is solved by *secret sharing*: for n-party MPC, every node value x is split into n random numbers $[x]_0, \ldots, [x]_{n-1}$ such that $x = \sum_{i=0}^{n-1} [x]_i$, and any $n-1$ of those random numbers are jointly independent. For party i's private input x, we let $[x]_i = x$ and $[x]_j = 0$ for $j \neq i$. Each $[x]_i$ is a *secret share* held by party i and never exposed to others. Such joint-independence offers strong security: even if $n-1$ parties collude, they cannot learn anything about the secret share of the honest party, or the original x. To reveal an output to a party j, all parties simply send their shares to party j who then adds them up.

Without exposing secret shares, MPC allows the parties to perform arithmetic computations together. With additive secret sharing, a "+" node is trivial; n-parties can compute $z = x + y$ *purely locally* — just adding their own shares of $[x]_i$ and $[y]_i$ so $[z]_i = [x]_i + [y]_i$ becomes the secret share of z. Similarly, subtracting secret-shared values and multiplied/added/divided by a *public number* can also be handled purely locally.

However, multiplying two secret-shared values x and y cannot be done locally, thus "$*$" nodes are expensive and require *inter-party communication*. The *de facto* standard approach is Beaver's protocol [6]. Each secure multiplication $z = x * y$ requires and consumes a one-time table of triples (a, b, c), such that a and b are independent random numbers and $c = a * b$ (also called a *Beaver triple*). The triple must be generated afresh and secretly shared among all parties before the multiplication can be done (e.g., offline pre-processing, §2.2): party i only holds $[a]_i$, $[b]_i$ and $[c]_i$ and cannot know anything about other parties' shares. Then $x * y$ can be computed in two steps:

Step 1: Each party computes $[d]_i = [x]_i - [a]_i$, $[e]_i = [y]_i - [b]_i$ locally; then *broadcast* their $[d]_i$ and $[e]_i$ to all parties.

Step 2: Each party receives all shares $[d]_j$ and $[e]_j$ and adds the shares up to *reveal* their true values d and e. Although $d = x - a$, $e = y - b$, random numbers a and b serve as *one-time pad* and ensures no info about x or y is leaked. Finally, each party i computes its own secret share of the product as

$$[z]_i = [x * y]_i = \begin{cases} [c]_i + e * [x]_i + d * [y]_i - e * d, & i = 0 \\ [c]_i + e * [x]_i + d * [y]_i, & i > 0 \end{cases}$$

It is easy to verify that $\sum_{i=0}^{n-1}[z]_i = z = x * y$ so $[z]_i$'s are valid secret shares. We can also rigorously prove that all $[z]_i$'s satisfy joint-independence, and this entire process does not leak anything about x, y, or z (see [6]).

The communication cost of the above protocol includes each party sending $[d]_i$ and $[e]_i$ to every other party, requiring at least one network latency. Furthermore, this communication introduces dependency between the two steps and acts as a global barrier: only after all parties finish Step 1 and the communication completes, all shares of $[d]_i$ and $[e]_i$ can be received to compute d and e in Step 2. Any compute units are idle waiting during network transmission. This may cause significant performance overheads up to $10^8\times$ slower than a plain-text multiplication, if the network connection among n-parties has a non-negligible latency or limited bandwidth.

To the best of our knowledge, existing ways to implement MPC multiplication under secret-sharing scheme all require at least one network latency *added to the critical path of every multiplication*. This is the core reason why MPC is hard to speed up in practice: GPUs and heterogeneous accelerators can help optimize local computation, but the overall time spent is lower-bounded by network latency times the depth of the longest multiplication chain (and comparisons, see §4.3).

2.2 One-time Table (Beaver Triple) Generation

In addition to expensive *online* multiplications mentioned above, generating one-time tables (Beaver triples) and delivering their secret shares to each party also adds significant overheads. Though it can be done *offline* as pre-processing to hide latency, in large ML computations the cost (e.g., energy consumption and server resource occupation) is still substantial because we need a fresh triple for *every* multiplication. Therefore, we argue that one-time table generation also has to be efficient to realize practical privacy-preserving ML.

Complete MPC implementations usually use other cryptographic techniques such as homomorphic encryption [34] or oblivious transfer [44] for Beaver triple generation, but they are extremely expensive in terms of computation and communication: the state-of-the-art implementation takes 0.1ms to 1ms to generate *a single* Beaver triple [44], around $10^5\times$ to $10^6\times$ slower than a plaintext multiplication. A more realistic approach widely used in practice (e.g. [24, 27, 46, 67]) is to introduce a *trusted third party* (TTP), who securely generates Beaver triples and delivers secret shares to all parties. In practice, TTP is usually provided by some entity independent of all parties involved in MPC, and the trust is based on such entity's credibility and neutrality.

For example, in a 2-party scenario, **Alice** (party 0) and **Bob** (party 1) perform a secure multiplication with the help of a TTP. As shown in Fig. 2(a), the TTP uses a pseudo random number generator (PRNG) to generate 5 random numbers $[a]_0$, $[a]_1$, $[b]_0$, $[b]_1$, $[c]_0$, and then computes $[c]_1 = ([a]_0+[a]_1)*([b]_0+[b]_1)-[c]_0$ to build the Beaver triple. Finally it securely delivers $[a]_0$, $[b]_0$, $[c]_0$ to Alice, and $[a]_1$, $[b]_1$, $[c]_1$ to Bob. To guarantee security, the TTP must satisfy both *integrity* (i.e., performing the designated operations to generate valid Beaver triples) and *confidentiality* (i.e., never leaking any information or colluding with any party).

2.3 Comparison with Alternatives and Related Work

MPC is one of the earliest cryptographic primitives to support privacy-preserving computing [85]. Securely computing Boolean circuit can be done by the Garbled Circuit based MPC [7, 35], but representing general fields (like integers) with Boolean circuits introduces unbearable overheads, making secret-sharing based MPC more favorable. There's a plethora of MPC schemes (e.g., [13, 22, 23]) with different security assumptions, some requiring honest majority or even $t < n/3$ (dishonest parties fewer than 1/3 of all parties), whereas PPMLAC implements security against up to $n-1$ corrupted adversaries.

From a system perspective, many software frameworks have been proposed to more efficiently realize MPC in real-world applications. Various MPC frameworks (e.g. [28, 43, 53, 90]) are invented, some for ML specifically (e.g. [60, 61, 84]), and some recent ones [46, 48, 76] utilize GPUs.

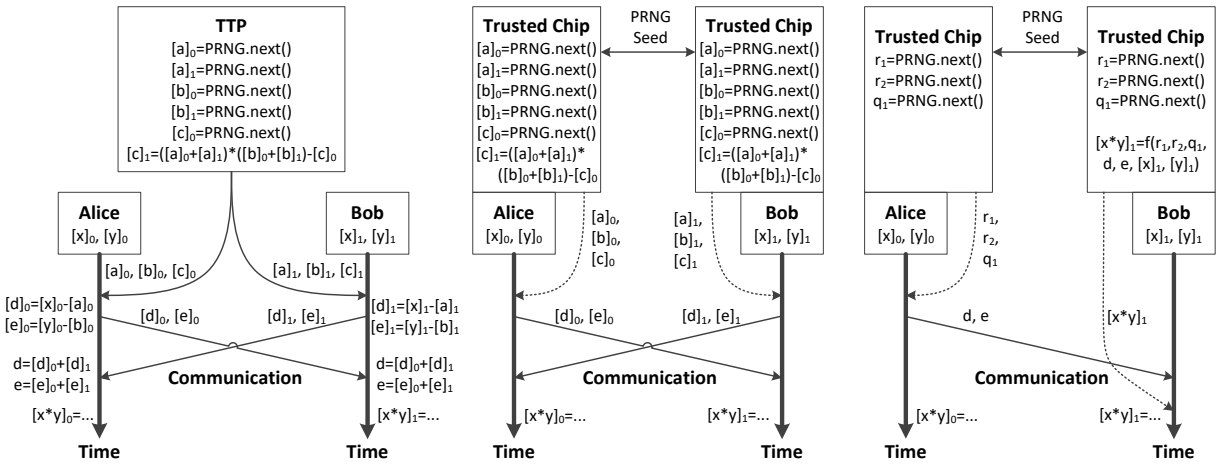
The more recent federated learning [10, 57, 86] is also a form of privacy computing, allowing multiple parties to together train an ML model without sharing data with each other. But it is limited to model training scenarios and usually requires a large number of users.

Modern cryptography also provides other alternative primitives that can be used for privacy-preserving computing. For example, homomorphic encryption (HE) allows a party to encrypt her data and send to another party, who can then conduct computations using only the cipher-text without knowing the plain-text inputs [34]. Unfortunately, HE is extremely compute-intensive, usually causing over $10^6\times$ slowdown [69]; not only the cipher-texts have to be several times larger than the original data, their computations are also orders of magnitude more expensive. Hence there have been several architectural efforts to accelerate various HE protocols using specialized hardware [66, 69, 74]. In contrast, MPC involves simple data formats and computations almost as cheap as plain-text processing, and does not require costly specific accelerated designs. However, the key bottleneck of MPC is communication, which we address in this work by reducing and tolerating its impacts.

Another approach is to build trusted execution environments (TEEs) in hardware. TEEs provide secure enclaves to remote parties on untrusted systems, so that plain-text data and computation can happen in such enclaves without leaking information. TEE examples include commercial products like Intel SGX [20] and ARM TrustZone [4], and academic prototypes like Sanctum [21], Komodo [32], Keystone [50], and Penglai [31]. While TEEs exhibit much lower performance overheads than crypto-based protocols, their security guarantees are widely criticized; plain-text data, even in the enclave, opens the door to a large body of side-channel attacks [14, 17, 47, 51, 54, 71, 75, 81–83]. In contrast, our proposal leverages rigorous MPC protocols to protect plain-text data and only relies on a minimum set of trusted hardware components, so it's much more robust to side channels (see §4.4).

3 KEY INNOVATIVE IDEAS

From §2 we see that the main performance bottlenecks of MPC are the communication overheads on the critical path of *every* multiplication, and the generation and distribution of a large number of



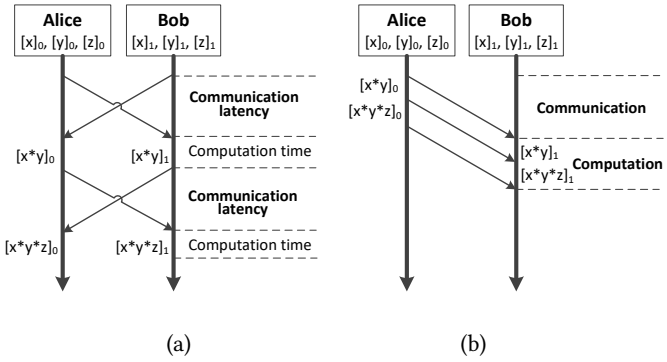
(a) existing TTP widely used in practice **(b)** chip-based TTP **(c)** better use of TTP in PPMLAC

Figure 2: Multiplication in 2-party MPC: (a) Use a trusted third party (TTP) to generate Beaver triples and deliver secret shares to each party; (b) When each party has a trusted chip with the same PRNG seed, Beaver triples can be generated and delivered locally; (c) We propose to make communication uni-direction: only Alice needs to send Bob data. Solid arrows denote remote communication via network; dashed arrows denote local data delivery with the trusted chip.

Beaver triples, *one per multiplication*. To address these inefficiencies, we propose a novel architecture named PPMLAC, which relies on a minimum set of trusted hardware units on a specially designed chip. If we trust the chip vendor, and assume the "hardened" chip cannot be broken into and only expose limited interface not leaking internal secret data, then the chip naturally offers strong integrity and confidentiality guarantees. The key idea of PPMLAC is that, such a *trusted chip* can act as a trusted third party (TTP), who can securely handle initialization and execute computations more efficiently. Furthermore, the trust chip also contains storage used as a secure cache for data that must be kept secret to each party, thus allowing for data reuse to save communication traffic.

More specifically, the trusted chips in PPMLAC are used in three ways. First, the trusted chips act as the TTP for one-time table generation and distribution, similar to [27]. Fig. 2(b) shows that, if each party involved in the MPC has a trusted chip that is initialized with the same seed (see details in §4.2), then their local PRNGs could deterministically generate the same sequence of random numbers ($[a]_0, [a]_1, [b]_0, [b]_1, [c]_0$), and also calculate $[c]_1$. The trusted chip only outputs the corresponding shares to the local party and discards the rest. Our actual design in §4.1 uses a modified form of one-time tables with three random numbers, as shown in Fig. 2(c). Using local trusted chips completely eliminates the communication with remote TTP, and the random numbers can be generated as needed as the computation goes rather than all at the beginning. Note that although the trusted chip knows all random numbers, it does not expose any interface for accessing these secrets to even the party who owns this chip.

Second, the trusted chip of a chosen party, say Bob, can also be used as a TTP to do the *actual multiplication*, after all other parties send their one-time-padded secret shares to him. If we keep all trusted chips always *synchronized*, Bob's trusted chip could exactly



(a) (b)

Figure 3: From (a) bi-directional communication that blocks computations, to (b) one-way communication in PPMLAC that is off the critical path and can be pipelined with computations.

emulate all random number generation happening in other trusted chips. This allows it to use the encrypted shares received from other parties and the random numbers generated locally to do multiplication with its own shares *without further communication* (§4.1). The resulting *one-way communication* (i.e., from other parties to Bob) is illustrated in Fig. 3(b). Alice now does not need to wait for the first multiplication $x * y$ to complete, and can immediately start the next multiplication with z. The messages to be sent to Bob for all multiplications can even be batched and transferred together, further improving communication efficiency. Using TTP this way to remove communication bottlenecks is novel.

Third, the local trusted chip also offers secure caching capabilities to reuse previous multiplicands without redundant data transfers (§4.5). For example, Alice may have previously sent her encrypted share of x to Bob's trusted chip to compute a multiplication. If later

x is needed again in another multiplication, as long as the previous x share is still cached in Bob's trusted chip, Alice does not need to send it again. Such data reuse can significantly reduce the data traffic needed to complete a large number of multiplications, e.g., in matrix multiplication and convolutions in ML workloads. Some software-based MPC systems extend beaver triples to *matrix multiplication triples* to exploit similar re-use opportunities [18, 19, 61, 84], but they require heavy offline phase or non-conventional powerful TTP [46], whereas PPMLAC only relies on standard caching algorithms.

Finally, our trusted chip design only requires standard and simple logic units (e.g., registers, random number generators, multipliers) in the trusted domain; all of them can be implemented in a security-aware manner. All operations happen deterministically and are oblivious to any secret data. Therefore, the minimum set of hardware makes the architecture much more robust to side channels and other attacks (§4.4).

To summarize, PPMLAC (1) uses lightweight remote attestation and initialization to *set up all parties' trusted chips in a synchronized state* (§4.2), allowing them to locally generate pseudo random numbers rather than relying on **expensive Beaver triple generation and distribution**; (2) eliminates the all-to-all communication barrier that blocks the critical path of every multiplication, and replaces with *one-way communication* (§4.1), which can be fully pipelined with computations to hide the long **communication latency**; (3) equips with *secure cache storage for secret data* inside the trusted domain (§4.5), which enables data reuse to save **communication bandwidth**; (4) only uses *a minimum set of hardware units for simple operations* in the trusted domain, thus allowing for easy implementations free from various **side channels and other attacks** (§4.4).

4 DESIGN AND IMPLEMENTATION

This section illustrates the design details of PPMLAC. §4.1 and §4.2 describe the secure multiplication and initialization protocols as well as their implementations in the trusted chip. §4.3 extends the basic design to support more operations, more data types, and many parties. §4.4 discusses the security arguments of our trusted chip, and §4.5 elaborates the caching optimization to reduce communication traffic. Finally §4.6 puts everything together to summarize the overall architecture of PPMLAC.

4.1 Architectural Support for Secure Multiplications

We first describe how the (simplified) PPMLAC architecture accelerates secure multiplications, under a 2-party scenario for ease of explanation. Suppose **Alice** (party 0) and **Bob** (party 1) each has a trusted chip, in addition to their normal CPUs, memory hierarchies, and network stacks. The trusted chip has three *operand registers* `reg[1,2,3]` and three *random registers* `rnd[1,2,3]`, each holding one 64-bit word. The chip also has a cryptographically-secure pseudo-random number generator (`CSPRNG`) that can deterministically generate a stream of random numbers from an initial seed. Assume for now a mechanism (detailed in §4.2) exists to securely initialize both Alice's and Bob's chips with the same seed, and no information about the seed is leaked to Bob so Bob can never know or predict the random numbers. Conceptually, each chip generates *two independent streams* of random numbers $q_1, q_2, \ldots$ and $r_1, r_2, \ldots$; this is emulated by deterministically splitting the stream generated by a CSPRNG into two, e.g., force every third number to be stream q and the rest to be r.

```
ld reg, addr   Load a word from memory to operand register
st reg, addr   Store a word from operand register to memory
setRnd rnd   Let the CSPRNG generate a new random number r_j
 (from stream r); set random register rnd to be r_j
outRnd memAddr   Let CSPRNG generate a new random number
 r_j; copy r_j to memory memAddr. outRnd is disabled for Bob.
outQnd memAddr   Let CSPRNG generate a new random number
 q_k; copy q_k to memory memAddr. outQnd is disabled for Bob.
mul reg_1, rnd_1, reg_2, rnd_2, reg_3   Let the CSPRNG generate a new
 random number q_k; set reg_3 to (reg_1 + rnd_1) * (reg_2 + rnd_2) - q_k
```

Listing 1: Trusted chip ISA and instruction semantics (for computation).

Listing 1 shows the trusted chip's instructions to support multiplications on secret-shared values. It uses normal `ld` and `st` for data transfers between operand registers and memory, but has *no way to load or store the random registers* by design. The `setRnd` instruction asks the CSPRNG to generate the next random number r_j from stream r and put it into the designated random register `rnd` (inaccessible from outside). The `outRnd` or `outQnd` instruction lets the CSPRNG generate the next random number r_j or q_k from stream r or q, but stores the number to the memory. These two instructions expose the random numbers and thus violate the security of MPC, so they are *disabled for Bob* (see §4.2 for details). Because Alice never receives any information from Bob with our one-way communication protocol, she cannot learn anything even she knows about these random numbers.

The `mul` instruction actually does a multiplication. Its semantic[1] is best illustrated by how it is used in **Protocol 1**, *which is the key algorithm of PPMLAC*. Inline comments starting with ";" follow each step in Protocol 1. Note: steps like assignment (←) and send/receive do not involve either operand registers or random registers, so they are purely performed by the normal CPU. The protocol's correctness is trivial to see: if the precondition holds, it is easy to verify that the output $[z]_0$ on Alice and $[z]_1$ on Bob add up to $z = x * y$, and they constitute a valid secret sharing of z. Also, both Alice and Bob invoke their CSPRNGs for the same times (twice for r stream, once for q stream), so the CSPRNGs on both sides move in tandem, generating the same sequence of random numbers and ending up in the same states. Thus the postcondition holds. We also emphasize that this multiplication protocol only uses one round of *one-way communication* from Alice to Bob.

We next show Protocol 1 is indeed *secure*: neither party should know anything about the other party's secret share. This is trivially ensured for Alice, as she never receives anything and thus cannot learn information of Bob. For Bob, we need to prove anything he receives or may infer from Alice is indistinguishable from pure random numbers. Note that Bob gets information from two sources: network messages from Alice, and instruction results from his local trusted chip. Below we explain all interesting points (highlighted in Protocol 1).

Step 3. Bob receives $d = [x]_0 - r_1$ (the same argument holds for e). Here the random number r_1 plays as a one-time pad encryption for

[1] The real `mul` instruction uses even more complex formula, see §4.4

Protocol 1 Multiply two secret-shared numbers $x * y \rightarrow z$

Notation: For a variable *var* stored in memory, $\langle var \rangle$ denotes the memory address of *var*.

Precondition: The CSPRNGs' states on both parties' chips are the same. $\{[x]_0, [x]_1\}$ form a valid secret sharing of value x; $\{[y]_0, [y]_1\}$ form a valid secret sharing of value y.

Postcondition: $\{[z]_0, [z]_1\}$ form a valid secret sharing of value $z = x * y$. The CSPRNGs' states are still the same.

Alice's procedure:

Variables in memory: two inputs $[x]_0, [y]_0$, the output $[z]_0$, four temporary variables a, b, d, e.

Steps:

(1) `outRnd` $\langle a \rangle$; so $a = r_1$, a new random number
(2) `outRnd` $\langle b \rangle$; so $b = r_2$, a new random number
(3) $d \leftarrow [x]_0 - a$; so $d = [x]_0 - r_1$
(4) $e \leftarrow [y]_0 - b$; so $e = [y]_0 - r_2$
(5) Send (d, e) to Bob ; *one-way* communication.
(6) `outQnd` $\langle [z]_0 \rangle$; so $[z]_0 = q_1$, a new random number.

Bob's procedure:

Variables in memory: two inputs $[x]_1, [y]_1$, the output $[z]_1$, four temporary variables d, e, u, v.

Steps:

(1) `setRnd` rnd_1 ; so $rnd_1 = r_1$, a new random number. **Note** that this r_1 is exactly the same r_1 as in Alice's procedure, because Bob's CSPRNG is synchronized with Alice's. Same for r_2 and q_1 in the steps below.
(2) `setRnd` rnd_2 ; so $rnd_2 = r_2$, a new random number.
(3) Receive (d, e) from Alice. **Note** that (d, e) here have the same values as (d, e) in Alice's procedure, because they come from Alice, so $d = [x]_0 - r_1, e = [y]_0 - r_2$
(4) $u \leftarrow [x]_1 + d$; so $u = [x]_1 + ([x]_0 - r_1) = x - r_1$
(5) $v \leftarrow [y]_1 + e$; so $v = [y]_1 + ([y]_0 - r_2) = y - r_2$
(6) `ld` reg_1, $\langle u \rangle$; so $reg_1 = u = x - r_1$
(7) `ld` reg_2, $\langle v \rangle$; so $reg_2 = v = y - r_2$
(8) `mul` reg_1, rnd_1, reg_2, rnd_2, reg_3 ; so $reg_3 = (reg_1 + rnd_1) * (reg_2 + rnd_2) - q_1 = (x - r_1 + r_1) * (y - r_2 + r_2) - q_1 = x * y - q_1$; q_1 is a new random number.
(9) `st` reg_3, $\langle [z]_1 \rangle$; so the output is $[z]_1 = x * y - q_1$

$[x]_0$ to *mask* any information Bob might infer about Alice's secret share, and makes d indistinguishable from a random number [72]. The key point is that r_1 is confined inside Bob's trusted chip and cannot be read out in any way.

Steps 4 and 5. Bob calculates $u \leftarrow [x]_1 + d$ *using its normal CPU and memory.* The two input values are Bob's own value $[x]_1$, and d which we already proved to leak no information about Alice. The output result, u, equals to $x - r_1$, which is again masked by r_1's randomness and indistinguishable from a random number to Bob. So Bob cannot infer anything from u, either. The same argument holds for v.

Step 9 transfers data from reg_3 inside the trusted chip to $[z]_1$ in the normal memory accessible by Bob. We argue that $[z]_1$ does not leak any information about x or y. This follows from the fact that $[z]_1 = x * y - q_1$, and the *freshly generated* random number q_1 acts as a one-time pad mask. $[z]_1$ is indistinguishable from a random number to Bob.

So far we have proved Protocol 1 is secure in a semi-honest setting (i.e., all plays strictly follow the protocol but only try to learn secrets). In §4.4 we further show it is secure even if Bob deviates from the protocol and calls arbitrary instructions to hack on the messages received from Alice.

4.2 Secure Initialization and Remote Attestation

Protocol 1 heavily relies on the fact that both parties' CSPRNGs are synchronized, i.e., moving in tandem with always the same states. Note that we only need to ensure that the CSPRNGs are *initialized with the same seed* into the same states before the MPC begins. By induction, the postcondition of Protocol 1 guarantees the CSPRNGs would stay synchronized across multiplications. Other operators (e.g., additions) in MPC are performed locally without involving CSPRNGs. However, such initialization cannot be done directly. Bob must be kept from knowing anything about the seed; otherwise he can predict the generated random numbers and thus recover secrets from messages received from Alice.

To address the challenge, we design a protocol based on *asymmetric cryptography*. Each party i's trusted chip has a pair of *public* key PK_i and *hidden* (a.k.a. *private*) key HK_i. PK_i is public while HK_i is hidden inside the trusted chip, not exposed to outside including the host CPU [3, 92].

With the public/hidden key pair, there seems to be a simple and standard initialization method: we let Alice choose a seed, encrypt it with Bob's public key PK_1, and send to Bob. Bob receives the encrypted seed, and passes it to his trusted chip, which uses the hidden key HK_1 to decrypt and retrieve the seed inside the chip. Because HK_1 is inaccessible to Bob, he would not be able to learn the seed.

However, the above simple approach actually has a serious vulnerability that allows Bob to perform *replay attacks* to learn Alice's secrets even without knowing the seed. To illustrate the attack, suppose Alice has two private numbers x, y, Bob has two private numbers a, b, and they want to jointly compute $c = a * x + b * y$ and reveal c to Bob. Note that x, y, a, b here are the *original* private inputs that are to be secret shared by the two parties (e.g., x is split into $[x]_0$ and $[x]_1$). MPC should guarantee that eventually Bob only knows information that can be deduced from his own private inputs and the revealed final result c, i.e., a, b, and c. This means that Bob knows only that x and y satisfy the equation $a * x + b * y = c$ (a *line* in the 2D Cartesian space) but not their concrete values. Recall that our MPC protocol only requires Alice to send one-way messages to Bob, and Bob computes his part of the protocol from the received messages using his trusted chip. Now Bob can save all the messages from Alice, and *replay* the entire protocol without Alice's awareness, but this time feed a different pair of a', b' values, to obtain another line $a' * x + b' * y = c'$. From these two equations Bob can solve the values of x, y and thus learn Alice's secret inputs.

The key vulnerability that enables Bob's replay attack is that Bob can reuse the same (encrypted) seed for multiple times and thus redo the protocol with the *same* inputs from Alice but *different* inputs

of himself. To break this, we choose a seed $s = m + TR$ consisting of two parts: a number m chosen by Alice, and an unpredictable *truly random number* TR generated by Bob's trusted chip each time. To do that, the trusted chip must also contain a *true random number generator* (TRNG) that makes use of physical noises [12, 41, 58, 79]. In order for Alice to initialize her CSPRNG with the same seed, Bob must send out the true random number TR to Alice in the initialization phase.

```
trueRnd mAddr Generate the next true random number,
  and copy its value to the memory mAddr.
initA mAddr, yAddr Read from memory mAddr to get a
 value m; read memory yAddr to get a value y; use
 my hidden key to decrypt y, get TR = Decrypt(HK, y)
 and initialize my GenA with seed m+TR.
initB xAddr, yAddr Read from memory xAddr to get a
 ciphered message x; use my hidden key to decrypt
 x and get the original message x' = {m, P} where P
 is a public key; generate the next true random
 number TR; initialize GenB with seed m+TR; use
 P to encrypt TR and get y; copy y to memory yAddr.

To generate random numbers, outRnd and outQnd always
use GenA, while setRnd and mul always use GenB.
```

Listing 2: Trusted chip ISA and instruction semantics (for secure initialization).

Listing 2 shows the trusted chip's instructions related to initialization, and Protocol 2 formalizes the initialization procedure discussed above. Each chip has a true random number generator TRNG and *two* CSPRNGs *GenA* and *GenB*: Alice only uses *GenA*, Bob only uses *GenB*. They correspond to the limitations in §4.1, where `outRnd` and `outQnd` are disabled for Bob. In our current settings, `outRnd` and `outQnd` only use *GenA*, and Bob has no way to get a correct *GenA* — he could call *initA*, but he knows neither the value m (chosen by Alice) nor TR (encrypted) to make his *GenA* match Alice's, therefore his `outRnd` and `outQnd` are useless. We could have used one CSPRNG with a single bit to select different "modes", e.g., 0 for Alice and 1 for Bob; but using separate CSPRNGs is more secure in practice, in case a physical attack or a random failure may flip the bit. Other implementations, e.g., using more bits or more complex logic to encode the chip mode, are also possible.

One may argue that the seed can be just the true random number TR without an Alice-chosen part m, which already ensures unpredictability and is protected by the trusted chips. This is theoretically correct, but we let Alice also contribute a part to implement *remote attestation*: Alice needs to first verify Bob's public key (and thus his trusted chip) is authenticated and trusted, only after which she participates in the MPC. Only the intended recipient (a trusted chip) can decrypt Alice's seed contribution with the corresponding hidden key.

4.3 Extensions

Binary secret sharing is essentially additive secret sharing on field $\mathbb{F}_2$ so each value is just 0 or 1 [35]. Now, addition becomes `XOR` ($\oplus$), and multiplication becomes `AND`. We group $W = 64$ bits into a word. A binary-secret-shared word x satisfies $x = [x]_0 \oplus \cdots \oplus [x]_{n-1}$ where each party i holds $[x]_i$. `XOR` and bit shift operators can be done locally, but `AND` needs communication almost identical to Protocol 1 (multiplication).

Protocol 2 Securely initialize CSPRNGs on both sides

Notation: For a variable *var* stored in memory, $\langle var \rangle$ denotes the memory address of *var*.

Precondition: Alice has a public key PK_0 in memory and a hidden key HK_0 inside her trusted chip, and temporary variables m, x, x', y, PK_1 in memory. Bob has a public key PK_1 in memory and a hidden key HK_1 inside his trusted chip, and temporary variables x, y in memory.

Postcondition: If succeeded, Alice's CSPRNG *GenA* and Bob's CSPRNG *GenB* are initialized with the same seed.

Procedure (Alice and Bob steps interleaved):

(1) **Bob** sends PK_1 (his public key) to Alice. **Alice** receives.
(2) **Alice** does *remote attestation* to verify PK_1 is a trusted public key, and fails the whole protocol if it is not.
(3) **Alice** calls `trueRnd` $\langle m \rangle$; m is set to a random number.
(4) **Alice** packs m, PK_0 (her public key) into a message $x' = \{m, PK_0\}$ and encrypts x' with PK_1 (Bob's public key) to get ciphertext $x = Encrypt(PK_1, \{m, PK_0\})$
(5) **Alice** sends message x to Bob. Bob receives x.
(6) **Bob** calls `initB` $\langle x \rangle$, $\langle y \rangle$; refer to `initB` semantics: internally, Bob's trusted chip decrypts x to get $x' = \{m, PK_0\}$ (PK_0 is Alice's public key), its TRNG generates a new true random number TR, its *GenB* is initialized with seed $m + TR$, and y is set to $Encrypt(PK_0, TR)$.
(7) **Bob** sends message y to Alice. Alice receives y.
(8) **Alice** calls `initA` $\langle m \rangle$, $\langle y \rangle$; refer to `initA` semantics: internally, Alice's trusted chip decrypts y to get $TR = Decrypt(HK_0, y)$, and her *GenA* is initialized with seed $m + TR$, same as Bob's *GenB*.

We convert between additive and binary secret sharing schemes (with algorithm similar to [28, 46]) to conduct *comparisons* on additive secret shared values. To test if $x < 0$, we (1) convert x to a binary shared value y; (2) right shift y by 63 bits to get the original sign bit; (3) convert the sign bit back to an additive shared value z, so $z = 1$ if $x < 0$ and 0 otherwise. Note that z is also secret shared, so no party can directly inspect its actual value; but z can participate in further computations, for example $x * (1 - z)$ is how we implement ReLU in neural networks — it equals x when $x \geq 0$ and 0 otherwise. To compare two secret shared values, we first obtain their difference and use the above protocol. Like other MPC tools [53, 55, 90], we do not support branching on a secret condition, but it can be emulated by executing both branches to get both results, and multiplying them with the 0-1 branch condition value to select the final output. Techniques like Oblivious RAM [36, 73, 88] can be combined with MPC to efficiently emulate certain secret-value-dependent control flows, which we leave as future work.

Beyond integers. It is easy to extend integers to fixed-point numbers. Only each multiplication needs to be followed by a divided-by-2^f, where f is the number of fractional bits. This divide-by-constant operation can be performed locally at each party, thus incurring negligible overheads. Our prototype PPMLAC chip uses 1 bit for sign, 31 bits for integer part, and 32 bits for fractional.

Fixed-point arithmetics are sufficient for our ML workloads, while general floating-point support is quite expensive using MPC and left as future work.

Beyond two parties. *Many-party MPC* can be supported by treating party 0 as Bob, and all other $n-1$ parties act as individual Alice. Each Alice only sends messages to Bob, and only Bob receives messages, so this is still *one-way communication*. Each Alice$_i$-Bob pair has its own synchronized CSPRNG pair, so different Alices are mutually distrusted. The hardware overhead is actually acceptable in most practical situations as there are not too many parties. An optimization is to time-multiplex the CSPRNG in Bob-side chip, by storing internal states for each Alice separately and doing context switches. Different Alices do not share seeds or states. More implementation care is needed to ensure security.

For a multiplication $z = x * y$, each Alice$_i$ generates random numbers $r_{i,1}$ and $r_{i,2}$, sends $d_i = [x]_i - r_{i,1}$ and $e_i = [y]_i - r_{i,2}$ to Bob. Bob calculates $u = [x]_0 + \sum_{i=1}^{n-1} d_i = x - \sum_{i=1}^{n-1} r_{i,1}$ and $v = [y]_0 + \sum_{i=1}^{n-1} e_i = y - \sum_{i=1}^{n-1} r_{i,2}$. Note that Bob's trusted chip can generate all $r_{i,1}$'s and $r_{i,2}$'s. Here $\sum_{i=1}^{n-1} r_{i,1}$ is just like r_1 in Protocol 1 and $\sum_{i=1}^{n-1} r_{i,2}$ is like r_2, and the multiplication can be done accordingly. For security, we guarantee that as long as at least one party is honest, even if all other parties collude, they cannot recover the honest party's secret, which is sent out as $x - r_i$ and r_i is never exposed.

4.4 Security Guarantees

Malicious adversaries. A *malicious* adversary may deviate from the protocols and call arbitrary instructions. It is trivial to see that Alice does not receive any information from Bob so her malicity does not break security. We next prove a malicious Bob still cannot infer anything about Alice's data. In Protocol 1, every message that Bob receives is just Alice's secret share minus a random number (e.g., $[x]_0 - r_1$). As long as the random number does not leak to Bob, the whole message is indistinguishable from a pure random number, and leak nothing to Bob (this argument is the foundation of Beaver's protocol [6] and one-time pad in general). These random numbers (e.g., r_1) are only present inside the trusted chip's *random registers* (generated by `setRnd` instructions on Bob's CSPRNG), which are inaccessible from outside. The only instruction that transfers information from *random registers* to *operand registers* is `mul`; but the result of `mul` is in the form of $\Phi - q$ where q is a freshly generated random number from a different stream than r_1, thus information from *rnd* to Φ is securely masked by the randomness of q. More protections like MAC ([8, 25, 26, 38, 39, 45, 62]) can be added to achieve *active security*, left as future work.

Side-channel attacks. It is straightforward to see that our initialization, local operations, and `mul` all only involve deterministic control flow and memory accesses *independent of secret data*. Higher-level control flow *across* operators are realized as in §4.3. All branches are obliviously executed regardless of data values, and the final result selection is a deterministic sequence of MPC operations. Thus our design is free from control flow and memory access side channels.

Most instructions are random number generation (Listing 1) and encryption/decryption (Listing 2), which are well studied security-sensitive operations that can benefit from a large body of side-channel-resist hardware implementations [52, 78, 80, 89]. The most critical one would be the `mul` instruction. Instead of the simplified version discussed in §4.1, the real `mul` generates *three* new random numbers q_1, q_2, q_3 on Bob side, and sets reg_3 (Line 8 of Protocol 1) as $reg_3 = q_3 + (reg_1 - q_1) * (rnd_2 + q_2) + (reg_2 - q_2) * (rnd_1 + q_1) + (reg_1 - q_1) * (reg_2 - q_2) = q_3 + x * y - (r_1 + q_1) * (r_2 + q_2)$. Correspondingly, Alice also calls three `outQnd` to get q_1, q_2, q_3, and sets $[z]_0$ (Line 6) to $[z]_0 = (a+q_1) * (b+q_2) - q_3 = (r_1+q_1) * (r_2 + q_2) - q_3$. One can easily verify its correctness. This more complex `mul` ensures that every intermediate term is fully random (always masked with a new random number), so the plain-text values of x and y *never exist physically*. Such a design prevents most physical side channels including power consumption and electromagnetic signal analysis [33, 70, 91].

Forward security. Our design supports an even stronger guarantee known as *forward security*: secure conversations are protected against future key leaks [11]. While the hidden key HK embedded in the trusted chip is assumed to be extremely difficult to retrieve, there may exist methods (despite being expensive) to break it, e.g., by freezing the chip to ultra low temperature and using some futuristic device to get HK. We reasonably assume that one cannot break into a trusted chip while the chip is powered on and working normally. So, we can additionally require the trusted chip to communicate with some central attestation server periodically, by signing a timestamp message with its hidden key. If the attestation server fails to receive an up-to-date timestamp, it marks the chip as "broken" and fails its future remote attestations. Thus all other MPC participants will refuse to talk to that chip.

Nevertheless, a malicious Bob can still record all history messages from previous MPC sessions, and then break the chip to retrieve the hidden key, trying to discover the historical secrets. The secure initialization (Protocol 2) is carefully designed to protect against this issue. TR generated by Bob's trusted chip is sent out not as plain-text, but encrypted under Alice's public key PK_0. Even if Bob is able to retrieve his chip's hidden key HK_1 as described above, and use it to decrypt and obtain the m values received from Alice, Bob is unable to know TR as he cannot know Alice's chip's hidden key HK_0. Thus Bob still cannot replay the history MPC sessions to sniff out anything without the seeds $m + TR$.

Man-in-the-middle attacks. We also need to ensure Bob cannot play any man-in-the-middle attack between Alice and his own trusted chip. The `initB` instruction (used in Protocol 2) is carefully designed to read a packed cipher-text message x from $xAddr$ and decrypt x to $x' = \{m, PK_0\}$ where PK_0 is Alice's public key. Otherwise, if `initB` takes PK_0 separately, Bob may replace it with another *fake* public key PK_2 whose hidden key HK_2 is known to Bob. The `initB` instruction would thus use PK_2 to encrypt TR and return it to Bob. Bob decrypts it using HK_2 and learns TR, and re-encrypts it using PK_0 before sending back to Alice. Alice would not be aware of anything in this case, but Bob can now break the aforementioned forward security protection. Combining and encrypting $\{m, PK_0\}$ together in our design prevents Bob from replacing the associated public key.

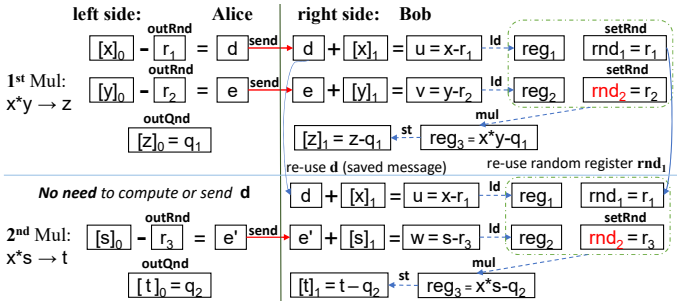


Figure 4: Cache multiplicands to save communication. Dashed arrows indicate data flow for chip-executed instructions; blue curvy solid arrows indicate reusing numbers; red straight solid arrows indicate network transmissions.

4.5 Reducing Communication by Caching Multiplicands

The basic PPMLAC architecture in §4.1 eliminates the communication *latency* bottleneck with one-way communication. We next present an optimization that further reduce the communication *bandwidth* consumption, through caching and reusing previous multiplicands inside the trusted chip.

In the basic Protocol 1, each multiplication requires Alice to send two numbers to Bob, but actually one or both of them can be saved. As shown in Fig. 4 top, suppose the two parties have just jointly computed a multiplication $z = x * y$. Alice sent $d = [x]_0 - r_1$ and $e = [y]_0 - r_2$ to Bob, and Bob called `setRnd` rnd_1 and `setRnd` rnd_2 so the random registers $rnd_1 = r_1$, $rnd_2 = r_2$, where r_1, r_2 are the two random numbers generated by both sides' CSPRNGs. Bob also computed $u = [x]_1 + d$ and $v = [y]_1 + e$ in the operand registers reg_1 and reg_2. Note that $reg_1 + rnd_1 = x$ and $reg_2 + rnd_2 = y$ — this is actually an invariant in our design: $reg_i + rnd_i$ equals to an multiplicand.

Now if they want to compute another multiplication $t = x * s$, Alice *does not need* to generate a new random number for $[x]_0$ and send their difference again, because Bob has already saved $d = [x]_0 - r_1$, which this second multiplication can reuse. This is illustrated in Fig. 4 bottom. Alice only needs to call `outRnd` $\langle b' \rangle$ to generate *one* random number r_3 and send $e' = [s]_0 - r_3$ to Bob. Once Bob receives e', he treats (d, e') as the message that should be received, and follows the rest of Protocol 1. This includes calling `setRnd` rnd_2 (step 2) so random register $rnd_2 = r_3$, computing $w = [s]_1 + e'$ (step 5) and calling `ld` reg_2, $\langle w \rangle$ (step 7) so operand register $reg_2 = w = s - r_3$. These steps ensure the invariant that $reg_2 + rnd_2 = s$ for the new multiplicand. For the other multiplicand, random register rnd_1 still equals to r_1, and operand register $reg_1 = [x]_1 + d = x - r_1$ can be re-produced from Bob's own secret share $[x]_1$ and the saved message d. Therefore, these states satisfy all requirements for `mul` and calling `mul` now would result in the correct output $[t]_1 = x * s - q_2$ for Bob; Alice just calls `outQnd` $\langle [t]_0 \rangle$ so $[t]_0 = q_2$.

This caching optimization does not exhibit any security issue. s and t are protected using newly generated random numbers r_3 and q_2. x is protected by the same r_1 from the previous multiplication, but this does not break the one-time pad requirement, as r_1 is never used to mask *different* plain-texts.

Clearly, the potential communication saving depends on how many multiplicands we can cache, i.e., the size of the register files in the trusted chip. In the previous example we only have three pairs of operand and random registers, limiting reuse opportunities. For example, if next the two parties want to compute a third multiplication $q = y * p$, they cannot reuse the previous data of y, because r_2 has already been overwritten, and there is no way to restore it (cannot be stored anywhere outside, and CSPRNGs cannot roll back). So now the two parties have to follow the complete protocol to transfer two numbers. In general, as long as a previously used random register associated with a multiplicand is not overwritten, this multiplicand can directly participate in new multiplications without requiring communication. But once a random register has been overwritten, its associated multiplicand cannot be reused further. This is a lot like the *cache eviction* problem, where evicted data must be fetched again from the remote side. The actual PPMLAC chip builds more (e.g. 384) random registers that is allowed by available hardware resource.

We currently leverage a compiler-based, modified LRU policy to statically optimize the register usage and minimize evictions. Specifically, our compiler book-keeps the usages of all random registers to perform standard LRU evictions, but provides a *soft pinning* facility to programmers, who can soft pin critical multiplicands such that their random registers can only be evicted by other soft-pinned multiplicands. Non-pinned multiplicands can only evict non-pinned random registers. Such a policy works well for our main target – ML and statistics workloads, which use matrices and tensors heavily; the programmer can pin a small block in one matrix that is multiplied by hundreds of blocks in the other matrix, so their random registers are always cached and re-used, saving over 99% of total data transfer (see §5 for quantitative evaluation). We leave fully-automated pinning support as future work.

Recall that MPC programs are always deterministic, without any control flow depending on secret data. Therefore static compiler analysis can be used. This property also eases soft pinning heuristics specified by programmers, because all tensor sizes are pre-determined *public numbers*. Note that Alice never uses random registers, yet the compiler also has to generate her local instructions to be consistent with Bob's operations, and to determine which multiplicands need message transfers. The compiler guarantees Alice and Bob always use the same register allocation/eviction.

4.6 Put It Altogether: The PPMLAC Architecture

Fig. 5 summarize the overall PPMLAC architecture for Bob side only. The incoming initial m is decrypted by the trusted chip's hidden key, and combined with TR generated from TRNG to get the final seed, to initialize the CSPRNG. At runtime, the generated random number sequences are put into the random registers to be used by `mul` instructions in the secret multipliers, or added to mask the multiplier output.

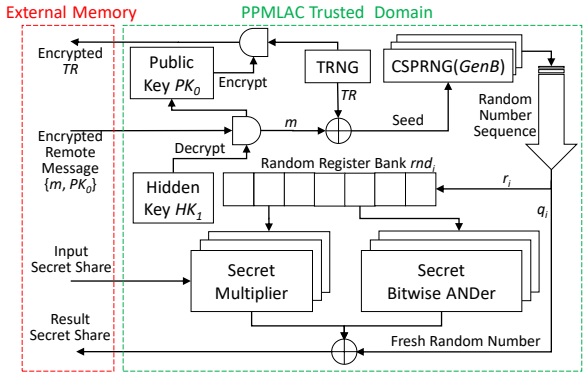


Figure 5: The overall PPMLAC architecture (focus on Bob's side).

Vectorization. To better support ML workloads, PPMLAC is inherently *vectorized* to exploit batched transfers and data-parallel execution. Multiple secret multipliers work on a vector of operand/random registers simultaneously, accessed from/to the banked register files. In our prototype, we use 24 lanes.

System integration. We discuss several potential integrations of our trusted chip with the host system. One is to build a *discrete heterogeneous device*, similar to accelerators like GPUs [63] and TPUs [42]. Besides the central components, we also add an on-device memory hierarchy to locally store data and avoid frequent transfers with the host CPU. This includes a small instruction buffer for the MPC program, a FIFO to buffer messages that are sent to or received from the other party (through the proxy of host CPU network stack), and data caches and global memory to hold the working data set. The host CPU initially loads the input data into the chip's global memory, and all intermediate data uses the on-device memory hierarchy rather than the host CPU memory whenever possible. Any remaining necessary data transfers are maximally batched by software to reduce overheads, similar to conventional CPU-GPU interactions. The entire memory hierarchy is *not* in the trusted domain; attackers can freely access them without violating security. Therefore although operand registers are backed up by the data caches/memory, random registers can *never* be spilled to the public memory.

We also enhance the chip with minimum functionalities to support other MPC computations besides the central multiplication and initialization protocols. These include normal arithmetic operations on operand registers (e.g., add, multiply/divide by a constant) and instructions to access memory and communication FIFO. This way, a complete MPC program (a DAG in Fig. 1) can run on-device as locally as possible. Note that all these components are *not* in the trusted domain, and their implementations are security-insensitive.

Alternatively, PPMLAC can be *tightly integrated with the CPU as an ISA extension*. The additional components would be TRNGs and CSPRNGs, attestation logic and public/hidden keys, random registers, and dedicated multiplier logic to support `mul`. Other parts, including operand registers, loads/stores, secret-sharing adds, message send/receive, can simply reuse the existing CPU functionalities. We implemented such an extension to the RISC-V CPU on an FPGA as a proof-of-concept, but we do not further evaluate its performance.

Programming model. The PPMLAC architecture is easy to use. It has a simple and well defined ISA (Listings 1 and 2). For discrete device integration, a separate MPC program is generated and loaded to the chip; for CPU extension, the MPC instructions are embedded into the original CPU program. We can let programmers manually write short code pieces or leverage existing MPC program compilers (e.g., [5, 37, 53, 90]) and add our multiplicand caching optimization (§4.5).

Security assumptions from whole-system perspective. We only consider the PPMLAC chip to be in trusted domain. The host CPU, memory, and software running outside the trusted domain are all untrusted. The stand-alone trusted chip *does not share* components like internal buffers and caches with host CPU, so it is protected from *microarchitectural attacks* from host CPU threads. The random numbers inside the chip are never exposed to outside, so they are safe against *memory bus snooping*. When extending existing ISA with PPMLAC, there's a PPMLAC core (trusted domain) and a host core (untrusted domain) on the same CPU, so extra care is needed to avoid microarchitectural sharing between the two, to prevent cross-core side-channel attacks like CrossTalk [65]. We do assume attackers cannot directly and physically break into the trusted chip to read its random numbers. Periodic timestamp messages (see §4.4 "Forward Security") can be used to detect such attacks. Nevertheless, PPMLAC is still resilient to many known *indirect* approaches (including side channels), explained as follows. PPMLAC's random numbers only participate in simple operations like multiplications. It is much easier to make such simple operations resilient against time-based and power-based analysis. Each multiplication is constant-time, and involves fresh random numbers that randomize per-operation power consumption. We can further improve side-channel protection by using redundant randomized operations, active power fences, etc. PPMLAC *forbids replay*, thus prevents attacks that rely on re-running programs multiple times. Fault-injection attacks may skip certain branching instructions to bypass guards. PPMLAC ISA has no branching instructions. If Alice or Bob skips some PPMLAC instructions, they are simply treated as malicious adversaries as §4.4, and still gain nothing. All these make PPMLAC safer than TEE.

Comparison with TEE. TEE adds enclave-related features to a full-fledged CPU that has complex microarchitecture like speculation. The complexity of the CPU brings a wide attack surface to TEE. In comparison, PPMLAC is designed just for the purpose of MPC multiplications, so it has a much narrower attack surface. To some extent, PPMLAC can be viewed as a minimal enclave that supports secure multiplications for MPC, which is more robust against side-channel and other attacks.

5 EVALUATION

We evaluate the performance of a PPMLAC prototype as a discrete accelerator implemented in FPGA with 256 banks of random registers (24 lanes per bank) and 100KB fast local memory cache. It is implemented using the High-Level Synthesis development environment, and runs on a Xilinx Virtex UltraScale+ VU9P FPGA of

an AWS F1 instance. It runs at only 150MHz clock rate due to the limitation of AWS.

We evaluate the scenario of 2-Party MPC under 3 different settings: (1) 1-DC: Alice and Bob are two nodes in one data center, with network round-trip latency less than 0.3ms. This is unsafe and unrealistic, given that the main motivation of MPC is for cross-organization data collaboration. Nevertheless, this setup allows us to see the ideal MPC performance at minimal network latency. (2) Cross-DC: We use `tc` command to add 65ms round-trip latency between Alice and Bob to emulate a connection between two data centers [59]. This is a realistic setting. (3) Trans-Pacific: We add 200ms round-trip latency to emulate a connection from East Asia to the US. This is also realistic, simulating cross-country data collaboration.

We compare with two state-of-the-art software MPC frameworks: MP-SPDZ [43] and CrypTen [46]. MP-SPDZ extends the classical SPDZ [25] protocol. We use its "semi-honest, 2^{64} ring" mode and run it on a server with 80-core Intel Xeon Gold 6145 CPU at 2.5GHz and 192GB RAM. Two MP-SPDZ processes communicate through a loopback network device, allowing tremendous network bandwidth, and each party still has 40 cores for data parallellism. CrypTen leverages the matrix multiplication optimizations [18, 61] to reduce communication and exploits GPU acceleration aggressively, but uses a very sophisticated TTP that is assumed to handle the heavy matrix multiplications and convolutions, instead of pure algorithmic safety measures like Oblivious Transfers. Therefore CrypTen improves performance significantly at the cost of more expensive hardware and more relaxed security setting. We run CrypTen on 3 servers (Alice, Bob, TTP) each with NVidia V100 GPU (16GB RAM) and Intel Xeon E5-2686 CPU at 2.30GHz. We exclude the initialization phases before the actual MPC from measurements, which favors the baselines as they need heavier initialization than PPMLAC.

Micro-benchmarks. Fig. 6-(a)(b)(c)(d) show how long it takes to run 10 million basic operations (multiply, comparison, bit-AND, exponential) by PPMLAC (P) versus CrypTen (C) and MP-SPDZ (M). The 10M operations are carefully written to allow full vectorization. Two factors favor the software-based MPC greatly: (1) CrypTen runs on V100 GPUs with 7.8 Trillion-OPS (64-bit operations per second) performance, and MP-SPDZ leverages multithreading and AVX512 instructions to utilize the many-core CPU, providing at least 100 GOPS. PPMLAC only has 3.6 GOPS peak throughput; (2) there is no data-dependence among operations, so software MPC is less impacted by the communication bottleneck.

Even under such unfair conditions, PPMLAC is still the fastest, even in the unrealistic 1-DC setting that dwarfs the communication latency issue. Note that PPMLAC does not focus on improving computations and thus this is the worst case for it. Under the more realistic cross-DC and Trans-Pac settings, PPMLAC achieves 14× to 280× speedup over CrypTen and 100× to 300000× speedup over MP-SPDZ. These results demonstrate the effectiveness of the optimizations in PPMLAC that eliminate the communication bottleneck and result in higher speedups with longer network latencies.

Real-world applications. We also evaluate four widely used real-world applications: (1) **LR**: inference for a Logistic Regression model with input dimension 10^6 and a Sigmoid layer, batching 24 samples per inference — a standard statistics task; (2) **SVM**: train a linear Support Vector Machine model with 1000 examples, 1000 features, and 100 training epochs — a classic machine learning training task; (3) **MLP**: inference for a Multilayer Perceptron model with five linear layers (dimensions 1024×512, 512×256, 256×128, 128×64, 64×32) and four ReLU layers in between, batching 24 samples per inference — a typical neural network used in recommendation systems; (4) **ResNet**: inference for the ResNet18 model [40] with deep architecture and residual blocks — a mainstream deep learning model for object recognition.

For **SVM**, Alice and Bob each has part of the training dataset, and they collectively train the model parameters; for other cases, Bob has the model parameters, Alice has the input data, and they collectively compute model inference results. MPC is flexible regarding data partitioning schemes and other partitioning schemes are also supported.

Fig. 6-(e)(f)(g)(h) compare the running times of PPMLAC (P) with CrypTen (C) and MP-SPDZ (M): under realistic settings (Cross-DC and Trans-Pac), PPMLAC achieves 40× to 40000× speedups over the software-based MPC implementations. Even under the unrealistic setting (1-DC), PPMLAC still exhibits performance improvements. We highlight the case with the heaviest workload **ResNet** over long-latency network **Trans-Pac**, where PPMLAC just runs for less than 4 seconds, which is *close to interactive* and can be acceptable for meaningful real-world use cases like financial Know-Your-Client checking or physical access control, whereas the fastest software-based MPC runs over two minutes cross-DC. Moreover, it is worth mentioning that PPMLAC exhibits the above great advantages despite using much weaker hardware (3.6 GOPS) than CrypTen (7.8 TOPS) and MP-SPDZ (100 GOPS). If we were building a real PPMLAC ASIC chip, another 10× performance would be possible, and further brings down the runtime for **ResNet** to within 0.5 seconds. In contrast, software-based MPC is restricted by the communication bottleneck and cannot benefit much from increasing computational capabilities.

We also ran the four ML models with *plain-text* computations on Intel Xeon 6145 (2.5GHz) CPU, just to see how much slowdown PPMLAC (under Cross-DC setting) causes against plain-text computation. The results are listed in Fig. 7. We can see that PPMLAC still brings some slowdown over plain-text computation, but they are all smaller than 100×, and much better than software-based MPC by far.

Model accuracy. Converting floating point numbers to fixed point used in MPC (§4.3) inherently loses precision and may affect *model accuracy*, so we verify the accuracy of our ResNet-18 model inference against PyTorch [64] with *plain-text, floating-point* computations, using the same pre-trained model and the ImageNet dataset [29]. We ran 1000 images and compare the Top-1 and Top-5 accuracy, shown in Fig. 8. The differences are tiny. Improving model accuracy in MPC is itself an interesting topic [15, 56]. We plan to incorporate techniques in the literature as our future work.

Impacts of multiplicand caching optimization (§4.5) are investigated in Fig. 9. For applications like MLP and ResNet that are dominated by matrix/tensor multiplications, the optimization saves 96% and 99% traffic. For vector-dominant cases, there are still over 45% savings.

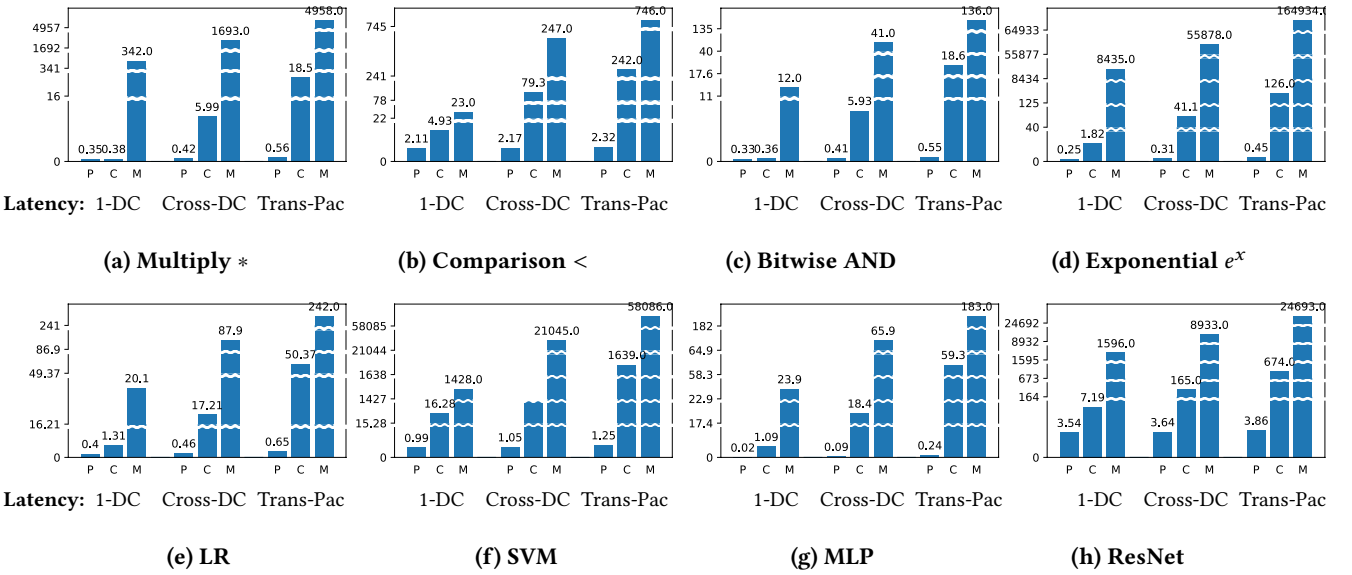


Figure 6: Running time in seconds by each MPC system with different latency setups: 1-DC, Cross-DC and Trans-Pac. MPC systems evaluated: P - PPMLAC; C - CrypTen; M - MP-SPDZ. (a) - (d) show micro-benchmarks, each performing 10^7 basic operations. (e) - (h) show results of four real-world applications.

	LR	SVM	MLP	ResNet
Plain-text computation on CPU	0.02s	0.09s	0.001s	0.09s
MPC on PPMLAC	0.46s	1.05s	0.09s	3.64s
Slowdown	23×	12×	90×	40×

Figure 7: Compare PPMLAC's MPC performance against plain-text computations on CPU.

Framework	Top-1 accuracy	Top-5 accuracy
PPMLAC	84.6%	95.3%
PyTorch	84.8%	95.9%

Figure 8: Model inference accuracy comparison: PPMLAC v.s. PyTorch

	LR	SVM	MLP	ResNet
with caching	210 MB	161 MB	10 MB	921 MB
without caching	384 MB	321 MB	276 MB	106 GB
savings percent	45.3%	49.8%	96.4%	99.1%

Figure 9: Effect of multiplicand caching: communication amount, run with v.s. without caching.

Generality. Our benchmark applications range from classical statistics to state-of-the-art deep learning models, and exercise various non-linear functions like Sigmoid, Exponential, ReLU, and Hinge Loss. PPMLAC, like other MPC tools, is suitable for general computation tasks instead of being restricted to training-only case like Federated Learning. Nevertheless, training needs higher programming efforts than inference. We manually implemented the training algorithm for SVM here to show PPMLAC can do both *training* and *inference*, but it is natural to adapt existing secure ML frameworks like [46, 53, 61] to use PPMLAC as the backend and get a full-fledged high-performance system. We leave such integration as our future work.

6 CONCLUSION

We have shown that Multi-Party Computation can be implemented with high performance and strong security by adding a minimum hardware trust root. Our chipset (PPMLAC) runs the secret-sharing based MPC protocol while eliminating its communication bottlenecks, thus achieves several orders of magnitude speed-up over software-based MPC. PPMLAC uses a minimum set of hardware units for simple operations in the trusted domain, allowing for easy implementations free from various side channels and other attacks. It is safe against dishonest majority (up to $n-1$ corrupted parties), and runs large-scale complex machine learning models with unprecedented high performance under realistic network environments.

ACKNOWLEDGMENTS

We thank the anonymous reviewers of ISCA 2022 for their insightful comments and suggestions on our paper. The authors are grateful to Zhiyao Li and Jiangbin Dong who have helped us with test server setup. We also want to thank Yan Huang, who have discussed with us and helped us gain clearer understanding on many MPC topics. Mingyu Gao is partially supported by the National Natural Science Foundation of China (62072262).

REFERENCES

[1] ACM CCS. 2021. PRIVACY PRESERVING MACHINE LEARNING, an ACM CCS 2021 Workshop. https://ppml-workshop.github.io/

[2] Mohammad Al-Rubaie and J. Morris Chang. 2019. Privacy-Preserving Machine Learning: Threats and Solutions. *IEEE Security and Privacy* 17, 2 (2019), 49–58. https://doi.org/10.1109/MSEC.2018.2888775

[3] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, Vol. 13. ACM New York, NY, USA, 7.

[4] ARM. 2022. TrustZone. https://www.arm.com/technologies/trustzone-for-cortex-a. Accessed: 2022-04-10.

[5] Yuyan Bao, Kirshanthan Sundararajah, Raghav Malik, Qianchuan Ye, Christopher Wagner, Fei Wang, Mohammad Hassan Ameri, Donghang Lu, Alexander Seto, Benjamin Delaware, Roopsha Samanta, Aniket Kate, Christina Garman, Jeremiah Blocki, Pierre-David Letourneau, Benoît Meister, Jonathan Springer, Tiark Rompf, and Milind Kulkarni. 2020. HACCLE: An Ecosystem for Building Secure Multi-Party Computations. *CoRR* abs/2009.01489 (2020). arXiv:2009.01489 https://arxiv.org/abs/2009.01489

[6] Donald Beaver. 1992. Efficient Multiparty Protocols Using Circuit Randomization. In *Advances in Cryptology — CRYPTO '91*, Joan Feigenbaum (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 420–432.

[7] D. Beaver, S. Micali, and P. Rogaway. 1990. The Round Complexity of Secure Protocols. In *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing* (Baltimore, Maryland, USA) *(STOC '90)*. Association for Computing Machinery, New York, NY, USA, 503–513. https://doi.org/10.1145/100216.100287

[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, Berlin, Heidelberg, 169–188.

[9] Dan Bogdanov, Sven Laur, and Jan Willemson. 2008. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *Computer Security - ESORICS 2008*, Sushil Jajodia and Javier Lopez (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 192–206.

[10] K. A. Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. 2016. Practical Secure Aggregation for Federated Learning on User-Held Data. In *NIPS Workshop on Private Multi-Party Machine Learning*. https://arxiv.org/abs/1611.04482

[11] Colin Boyd and Kai Gellert. 2019. A Modern View on Forward Security. Cryptology ePrint Archive, Report 2019/1362. https://ia.cr/2019/1362.

[12] R. Brederlow, R. Prakash, C. Paulus, and R. Thewes. 2006. A low-power true random number generator using random telegraph noise of single oxide-traps. In *2006 IEEE International Solid State Circuits Conference - Digest of Technical Papers*. 1666–1675. https://doi.org/10.1109/ISSCC.2006.1696222

[13] Ernest F. Brickell. 1990. Some Ideal Secret Sharing Schemes. In *Advances in Cryptology — EUROCRYPT '89*, Jean-Jacques Quisquater and Joos Vandewalle (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 468–475.

[14] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. 2019. Fallout: Leaking Data on Meltdown-resistant CPUs. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.

[15] Octavian Catrina and Amitabh Saxena. 2010. Secure Computation with Fixed-Point Numbers. In *Financial Cryptography and Data Security*, Radu Sion (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 35–50.

[16] J.M. Chang, D. Zhuang, and G.D. Samaraweera. 2022. *Privacy-Preserving Machine Learning*. Manning.

[17] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. 2018. SgxPectre Attacks: Leaking Enclave Secrets via Speculative Execution. *CoRR* abs/1802.09085 (2018). arXiv:1802.09085 http://arxiv.org/abs/1802.09085

[18] Hao Chen, Miran Kim, Ilya Razenshteyn, Dragos Rotaru, Yongsoo Song, and Sameer Wagh. 2020. Maliciously Secure Matrix Multiplication with Applications to Private Deep Learning. In *Advances in Cryptology – ASIACRYPT 2020*, Shiho Moriai and Huaxiong Wang (Eds.). Springer International Publishing, Cham, 31–59.

[19] Martine de Cock, Rafael Dowsley, Anderson C.A. Nascimento, and Stacey C. Newman. 2015. Fast, Privacy Preserving Linear Regression over Distributed Datasets Based on Pre-Distributed Data. In *Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security* (Denver, Colorado, USA) *(AISec '15)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/2808769.2808774

[20] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. Cryptology ePrint Archive, Report 2016/086. https://eprint.iacr.org/2016/086.

[21] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *Proceedings of the 25th USENIX Conference on Security Symposium* (Austin, TX, USA) *(SEC'16)*. USENIX Association, USA, 857–874.

[22] Ronald Cramer, Ivan Damgård, and Yuval Ishai. 2005. Share Conversion, Pseudorandom Secret-Sharing and Applications to Secure Computation. In *Theory of Cryptography*, Joe Kilian (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 342–362.

[23] Ronald Cramer, Ivan Damgård, and Ueli Maurer. 2000. General Secure Multi-party Computation from any Linear Secret-Sharing Scheme. In *Advances in Cryptology — EUROCRYPT 2000*, Bart Preneel (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 316–334.

[24] Morten Dahl, Jason Mancuso, Yann Dupis, Ben Decoste, Morgan Giraud, Ian Livingstone, Justin Patriquin, and Gavin Uhma. 2018. Private Machine Learning in TensorFlow using Secure Computation. arXiv:1810.08130 [cs.CR]

[25] 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, Berlin, Heidelberg, 643–662.

[26] Ivan Damgård, Marcel Keller, Enrique Larraia, Christian Miles, and Nigel P. Smart. 2012. Implementing AES via an Actively/Covertly Secure Dishonest-Majority MPC Protocol. In *Proceedings of the 8th International Conference on Security and Cryptography for Networks* (Amalfi, Italy) *(SCN'12)*. Springer-Verlag, Berlin, Heidelberg, 241–263. https://doi.org/10.1007/978-3-642-32928-9_14

[27] 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)*. USENIX Association, San Diego, CA, 893–908. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/demmler

[28] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society. https://www.ndss-symposium.org/ndss2015/aby---framework-efficient-mixed-protocol-secure-two-party-computation

[29] Jia Deng, Alex Berg, Sanjeev Satheesh, H Su, Aditya Khosla, and Li Fei-Fei. 2012. Imagenet large scale visual recognition competition 2012 (ILSVRC2012). *See net. org/challenges/LSVRC* 41 (2012).

[30] Tamara Dugan and Xukai Zou. 2016. A Survey of Secure Multiparty Computation Protocols for Privacy Preserving Genetic Tests. In *2016 IEEE First International Conference on Connected Health: Applications, Systems and Engineering Technologies (CHASE)*. 173–182. https://doi.org/10.1109/CHASE.2016.71

[31] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. 2021. Scalable Memory Protection in the PENGLAI Enclave. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 275–294. https://www.usenix.org/conference/osdi21/presentation/feng

[32] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. 2017. Komodo: Using Verification to Disentangle Secure-Enclave Hardware from Software. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 287–305. https://doi.org/10.1145/3132747.3132782

[33] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. 2001. Electromagnetic Analysis: Concrete Results. In *Cryptographic Hardware and Embedded Systems — CHES 2001*. Springer Berlin Heidelberg, Berlin, Heidelberg, 251–261.

[34] Craig Gentry. 2009. *A Fully Homomorphic Encryption Scheme*. Ph. D. Dissertation. Stanford, CA, USA. Advisor(s) Boneh, Dan. AAI3382729.

[35] O. Goldreich, S. Micali, and A. Wigderson. 1987. How to Play ANY Mental Game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing* (New York, New York, USA) *(STOC '87)*. Association for Computing Machinery, New York, NY, USA, 218–229. https://doi.org/10.1145/28395.28420

[36] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. 2012. Secure Two-Party Computation in Sublinear (Amortized) Time. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (Raleigh, North Carolina, USA) *(CCS '12)*. Association for Computing Machinery, New York, NY, USA, 513–524. https://doi.org/10.1145/2382196.2382251

[37] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. 2019. SoK: General Purpose Compilers for Secure Multi-Party Computation. *2019 IEEE Symposium on Security and Privacy (SP)* (2019), 1220–1237.

[38] Carmit Hazay, Emmanuela Orsini, Peter Scholl, and Eduardo Soria-Vazquez. 2018. Concretely Efficient Large-Scale MPC with Active Security (or, TinyKeys for TinyOT). In *Advances in Cryptology – ASIACRYPT 2018*, Thomas Peyrin and Steven Galbraith (Eds.). Springer International Publishing, Cham, 86–117.

[39] Carmit Hazay, Emmanuela Orsini, Peter Scholl, and Eduardo Soria-Vazquez. 2018. TinyKeys: A New Approach to Efficient Multi-Party Computation. In *Advances in Cryptology – CRYPTO 2018*, Hovav Shacham and Alexandra Boldyreva (Eds.). Springer International Publishing, Cham, 3–33.

[40] 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. https://doi.org/10.1109/CVPR.2016.90

[41] Daniel E. Holcomb, Wayne P. Burleson, and Kevin Fu. 2009. Power-Up SRAM State as an Identifying Fingerprint and Source of True Random Numbers. *IEEE Trans. Comput.* 58, 9 (2009), 1198–1210. https://doi.org/10.1109/TC.2008.212

[42] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *44th International Symposium on Computer Architecture (ISCA)*. 1–12.

[43] 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* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 1575–1590. https://doi.org/10.1145/3372297.3417872

[44] Marcel Keller, Emmanuela Orsini, and Peter Scholl. 2016. MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 830–842. https://doi.org/10.1145/2976749.2978357

[45] Marcel Keller, Peter Scholl, and Nigel P. Smart. 2013. An Architecture for Practical Actively Secure MPC with Dishonest Majority. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer Communications Security* (Berlin, Germany) *(CCS '13)*. Association for Computing Machinery, New York, NY, USA, 549–560. https://doi.org/10.1145/2508859.2516744

[46] Brian Knott, Shobha Venkataraman, Awni Hannun, Shubho Sengupta, Mark Ibrahim, and Laurens van der Maaten. 2021. CrypTen: Secure Multi-Party Computation Meets Machine Learning. arXiv:2109.00984 [cs.LG]

[47] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[48] 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 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 336–353. https://doi.org/10.1109/SP40000.2020.00092

[49] Andrei Lapets, Nikolaj Volgushev, Azer Bestavros, Frederick Jansen, and Mayank Varia. 2016. Secure multi-party computation for analytics deployed as a lightweight web application. https://open.bu.edu/handle/2144/21786

[50] 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* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 38, 16 pages. https://doi.org/10.1145/3342195.3387532

[51] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. 2021. CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 717–732. https://www.usenix.org/conference/usenixsecurity21/presentation/li-mengyuan

[52] Xun Li, Mohit Tiwari, Jason K. Oberg, Vineeth Kashyap, Frederic T. Chong, Timothy Sherwood, and Ben Hardekopf. 2011. Caisson: A Hardware Description Language for Secure Information Flow. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 109–120. https://doi.org/10.1145/1993498.1993512

[53] Yi Li and Wei Xu. 2019. PrivPy: General and Scalable Privacy-Preserving Data Mining. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery Data Mining* (Anchorage, AK, USA) *(KDD '19)*. Association for Computing Machinery, New York, NY, USA, 1299–1307. https://doi.org/10.1145/3292500.3330920

[54] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*.

[55] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. 2015. ObliVM: A Programming Framework for Secure Computation. In *2015 IEEE Symposium on Security and Privacy*. 359–376. https://doi.org/10.1109/SP.2015.29

[56] Ziyao Liu, Ivan Tjuawinata, Chaoping Xing, and Kwok-Yan Lam. 2020. MPC-enabled Privacy-Preserving Neural Network Training against Malicious Attack. *CoRR* abs/2007.12557 (2020). arXiv:2007.12557 https://arxiv.org/abs/2007.12557

[57] Lingjuan Lyu, Han Yu, Xingjun Ma, Lichao Sun, Jun Zhao, Qiang Yang, and Philip S. Yu. 2020. Privacy and Robustness in Federated Learning: Attacks and Defenses. *CoRR* abs/2012.06337 (2020). arXiv:2012.06337 https://arxiv.org/abs/2012.06337

[58] Sanu K. Mathew, Suresh Srinivasan, Mark A. Anders, Himanshu Kaul, Steven K. Hsu, Farhana Sheikh, Amit Agarwal, Sudhir Satpathy, and Ram K. Krishnamurthy. 2012. 2.4 Gbps, 7 mW All-Digital PVT-Variation Tolerant True Random Number Generator for 45 nm CMOS High-Performance Microprocessors. *IEEE Journal of Solid-State Circuits* 47, 11 (2012), 2807–2821. https://doi.org/10.1109/JSSC.2012.2217631

[59] Microsoft. 2021. Azure network round-trip latency statistics. https://docs.microsoft.com/en-us/azure/networking/azure-network-latency.

[60] 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* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 35–52. https://doi.org/10.1145/3243734.3243760

[61] 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. https://doi.org/10.1109/SP.2017.12

[62] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. 2012. A New Approach to Practical Active-Secure Two-Party Computation. In *Advances in Cryptology – CRYPTO 2012*, Reihaneh Safavi-Naini and Ran Canetti (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 681–700.

[63] NVIDIA. 2022. CUDA GPUs. https://developer.nvidia.com/cuda-gpus. Accessed: 2022-04-10.

[64] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

[65] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2021. CrossTalk: Speculative Data Leaks Across Cores Are Real. In *2021 IEEE Symposium on Security and Privacy (SP)*. 1852–1867. https://doi.org/10.1109/SP40001.2021.00020

[66] M. Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. 2020. HEAX: An Architecture for Computing on Encrypted Data. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1295–1309. https://doi.org/10.1145/3373376.3378523

[67] 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* (Incheon, Republic of Korea) *(ASIACCS '18)*. Association for Computing Machinery, New York, NY, USA, 707–721. https://doi.org/10.1145/3196494.3196522

[68] Victor Ruehle, Robert Sim, Sergey Yekhanin, Nishanth Chandran, Melissa Chase, Daniel Jones, Kim Laine, Boris Köpf, Jaime Teevan, Jim Kleewein, and Saravan Rajmohan. 2021. Privacy Preserving Machine Learning: Maintaining confidentiality and preserving trust. https://www.microsoft.com/en-us/research/blog/privacy-preserving-machine-learning-maintaining-confidentiality-and-preserving-trust/

[69] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. 2021. F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 238–252. https://doi.org/10.1145/3466752.3480070

[70] F. Schellenberg, D. R. E. Gnad, A. Moradi, and M. B. Tahoori. 2018. An inside job: Remote power analysis attacks on FPGAs. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1111–1116. https://doi.org/10.23919/DATE.2018.8342177

[71] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS*.

[72] Claude E Shannon. 1949. Communication theory of secrecy systems. *The Bell system technical journal* 28, 4 (1949), 656–715.

[73] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. 2011. Oblivious RAM with O((Logn)3) Worst-Case Cost. In *Proceedings of the 17th International Conference on The Theory and Application of Cryptology and Information Security* (Seoul, South Korea) *(ASIACRYPT'11)*. Springer-Verlag, Berlin, Heidelberg, 197–214. https://doi.org/10.1007/978-3-642-25385-0_11

[74] Sujoy Sinha Roy, Furkan Turan, Kimmo Jarvinen, Frederik Vercauteren, and Ingrid Verbauwhede. 2019. FPGA-Based High-Performance Parallel Architecture for Homomorphic Computing on Encrypted Data. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 387–398. https://doi.org/10.1109/HPCA.2019.00052

[75] Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W. Fletcher. 2019. MicroScope: Enabling Microarchitectural Replay Attacks. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 318–331. https://doi.org/10.1145/3307650.3322228

[76] Sijun Tan, Brian Knott, Yuan Tian, and David J. Wu. 2021. CryptGPU: Fast Privacy-Preserving Machine Learning on the GPU. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 1021–1038. https://doi.org/10.1109/SP40001.2021.00098

[77] Andrew S Tanenbaum. 1989. Never underestimate the bandwidth of a station wagon full of tapes hurtling down the highway.. In *Computer Networks*. Prentice-Hall, New Jersey, USA, 51.

[78] Mohit Tiwari, Jason K. Oberg, Xun Li, Jonathan Valamehr, Timothy Levin, Ben Hardekopf, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. 2011. Crafting a Usable Microkernel, Processor, and I/O System with Strict and Provable Information Flow Security. In *Proceedings of the 38th Annual International Symposium on Computer Architecture* (San Jose, California, USA) *(ISCA '11)*. Association for Computing Machinery, New York, NY, USA, 189–200. https://doi.org/10.1145/2000064.2000087

[79] Carlos Tokunaga, David Blaauw, and Trevor Mudge. 2008. True Random Number Generator With a Metastability-Based Quality Control. *IEEE Journal of Solid-State Circuits* 43, 1 (2008), 78–85. https://doi.org/10.1109/JSSC.2007.910965

[80] Klaus v. Gleissenthall, Rami Gökhan Kıcı, Deian Stefan, and Ranjit Jhala. 2019. IODINE: Verifying Constant-Time Execution of Hardware. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1411–1428. https://www.usenix.org/conference/usenixsecurity19/presentation/vongleissenthall

[81] 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 *Proceedings of the 27th USENIX Security Symposium*. USENIX Association.

[82] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. 2020. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *41th IEEE Symposium on Security and Privacy (S&P'20)*.

[83] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-flight Data Load. In *S&P*.

[84] Sameer Wagh, Divya Gupta, and Nishanth Chandran. 2018. SecureNN: Efficient and Private Neural Network Training. Cryptology ePrint Archive, Report 2018/442. https://ia.cr/2018/442.

[85] Andrew C. Yao. 1982. Protocols for secure computations. In *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*. 160–164. https://doi.org/10.1109/SFCS.1982.38

[86] Xuefei Yin, Yanming Zhu, and Jiankun Hu. 2021. A Comprehensive Survey of Privacy-Preserving Federated Learning: A Taxonomy, Review, and Future Directions. *ACM Comput. Surv.* 54, 6, Article 131 (jul 2021), 36 pages. https://doi.org/10.1145/3460427

[87] Tjalling J. Ypma. 1995. Historical Development of the Newton-Raphson Method. *SIAM Rev.* 37, 4 (1995), 531–551. http://www.jstor.org/stable/2132904

[88] Samee Zahur and David Evans. 2015. Obliv-C: A Language for Extensible Data-Oblivious Computation. *IACR Cryptol. ePrint Arch.* 2015 (2015), 1153.

[89] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. 2015. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS '15)*. Association for Computing Machinery, New York, NY, USA, 503–516. https://doi.org/10.1145/2694344.2694372

[90] Yihua Zhang, Aaron Steele, and Marina Blanton. 2013. PICCO: A General-Purpose Compiler for Private Distributed Computation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer Communications Security* (Berlin, Germany) *(CCS '13)*. Association for Computing Machinery, New York, NY, USA, 813–826. https://doi.org/10.1145/2508859.2516752

[91] M. Zhao and G. E. Suh. 2018. FPGA-Based Remote Power Side-Channel Attacks. In *2018 IEEE Symposium on Security and Privacy (SP)*. 229–244. https://doi.org/10.1109/SP.2018.00049

[92] Shijun Zhao, Qianying Zhang, Guangyao Hu, Yu Qin, and Dengguo Feng. 2014. Providing Root of Trust for ARM TrustZone Using On-Chip SRAM. In *Proceedings of the 4th International Workshop on Trustworthy Embedded Devices* (Scottsdale, Arizona, USA) *(TrustED '14)*. Association for Computing Machinery, New York, NY, USA, 25–36. https://doi.org/10.1145/2666141.2666145