Customizing Trusted AI Accelerators for Efficient Privacy-Preserving Machine Learning

Peichen Xie
Peking University

Xuanle Ren
Alibaba Group

Guangyu Sun
Peking University

ABSTRACT

The use of trusted hardware has become a promising solution to enable privacy-preserving machine learning. In particular, users can upload their private data and models to a hardware-enforced trusted execution environment (e.g. an *enclave* in Intel SGX-enabled CPUs) and run machine learning tasks in it with confidentiality and integrity guaranteed. To improve performance, AI accelerators have been widely employed for modern machine learning tasks. However, how to protect privacy on an AI accelerator remains an open question. To address this question, we propose a solution for efficient privacy-preserving machine learning based on an unmodified trusted CPU and a customized trusted AI accelerator. We carefully leverage cryptographic primitives to establish trust and protect the channel between the CPU and the accelerator. As a case study, we demonstrate our solution based on the open-source versatile tensor accelerator. The result of evaluation shows that the proposed solution provides efficient privacy-preserving machine learning at a small design cost and moderate performance overhead.

1 INTRODUCTION

Privacy is a key issue in many machine learning applications such as machine learning as a service, federated learning, inference at the edge, etc. Generally, we formulate a machine learning task by

$$\text{result} = f(\text{model}, \text{data})$$

where the model and the data can be provided by two or more parties. The term *privacy-preserving machine learning* is to evaluate $f(\text{model}, \text{data})$ without disclosing their private model/data.

To this end, numerous recent studies regard it as a secure multi-party computation (MPC) problem and propose cryptography-based solutions. For example, homomorphic encryption, garbled circuits and secret sharing are leveraged for privacy-preserving inference [5, 10] and privacy-preserving training [16, 19]. However, cryptography-based solutions are very inefficient because of the intensive computation and communication of homomorphic encryption and MPC protocols. In addition, these solutions lack versatility because the operations supported by homomorphic encryption and MPC protocols are limited.

Alternatively, trusted execution environment (TEE) is a promising approach to efficient privacy-preserving machine learning. Lately, CPU designers have integrated trusted computing components into CPUs which enables hardware-based trusted execution environments. Based on trusted CPUs (e.g. Intel's SGX-enabled CPUs), the model/data owners can upload their private model/data to a secure *enclave* via encrypted communication channels. The enclave is secured such that only verified trusted software can access and decrypt the model and data. In the enclave, the CPU evaluates $f(\text{model}, \text{data})$ over the decrypted model and data. Therefore, compared with cryptography-based solutions, such approaches are more versatile and can achieve privacy-preserving machine learning with lower overhead [7, 18].

However, the computational power of CPUs is still insufficient for large-scale ML models such as deep neural networks. To improve performance, both academia and industry have designed various dedicated AI accelerators such as Eyeriss[2], TPUs[9], VTA[17], etc. Although AI accelerators can help handle such workloads, employing AI accelerators as well as protecting privacy remains an open question (Section 2.2). For example, Volos et al. [21] have proposed enabling TEEs on GPUs, but this solution relies on the assumption that recent server-class GPUs have trusted device memory (e.g. on-package HBM). This assumption, however, is not appropriate for many AI accelerators, which use off-package untrusted memory. Jiang et al. [8] propose to customize the CPU hardware to provide secure I/O paths to the GPU. However, it omits physical attacks (e.g. eavesdropping on the bus), and modifying the architecture/behavior of modern CPUs may also be difficult for most companies and institutions.

In contrast, in this paper, we propose a simple but effective solution for efficient privacy-preserving machine learning by customizing trusted AI accelerators without modifying the CPU. Specifically, we customize the accelerator by adding a security interface and a crypto engine to the AI accelerator; the CPU, the core logic of the accelerator and the buses remain unchanged. To ensure confidentiality and integrity, we first leverage the newly-added cryptographic primitives to establish trust and exchange a symmetric key between the CPU enclave and the AI accelerator. Then, we carefully protect all communication channels between the CPU enclave and the accelerator using this symmetric key. Since the key is only held by the CPU enclave and the accelerator, anyone without the key cannot obtain or tamper with the code and data. As a result, we can offload the computation to the accelerator securely, and leverage the accelerator to evaluate $f(\text{model, data})$ efficiently.

As a case study, we demonstrate the implementation of our solution on an open-source AI accelerator (Section 4). The result shows that our solution can utilize the high computational power of the AI accelerator with major hardware/software designs (such as the IP core design, the instruction set architecture and the compiler) unchanged. Therefore, this solution is suitable for the scenarios where AI accelerators are customizable but CPUs/GPUs are not.

In summary, the contributions of our paper are as follows:

- We propose the solution for efficient privacy-preserving machine learning by customizing trusted AI accelerators and extending TEEs to AI accelerators.
- We propose the methods to secure the code and data used by the customized AI accelerator. Particularly, the methods are effective even if the AI accelerator does not have trusted device memory.
- We evaluate the implementation of our solution on VTA (an open-source AI accelerator) as a case study, and analyze the performance impact with cycle-accurate register-transfer-level simulation.
- We analyze the performance overhead of our solution and present several ways to reduce the overhead.

2 BACKGROUND

2.1 Trusted Execution Environment

A trusted execution environment (TEE) guarantees code and data in it are isolated and protected from the outside environment, including unauthorized users, attackers, system administrators and even the operating system and the hypervisor. Various implementations of the TEE have been proposed by industry and academia recently, e.g. Intel SGX [3], ARM TrustZone [14], Keystone [12], based on their respective CPUs. In this paper, we use Intel SGX as the CPU-side TEE because it provides comprehensive protection and it is industry-ready. In addition, because SGX has a minimal trusted computing base (TCB) [4], our solution can be adapted to other CPUs.

SGX-enabled CPUs can protect a specific memory region (i.e. processor's reserved memory) from all unauthorized memory accesses and DMA accesses thanks to the integration of the uncore (including the memory controller and the I/O controller). Besides, the SGX-enabled CPU has integrated a memory encryption engine in order to protect this memory region against physical attacks. Based on these security features, Intel provides secure containers called *enclaves* that hold private data and code in the processor's reserved memory.

For privacy-preserving machine learning, the typical solution based on SGX is to have different parties upload their confidential data/model into an enclave over an encrypted channel, attest the software running in the enclave and finally receive the encrypted result [7, 18]. However, using trusted CPUs, the performance is restricted by the CPU performance, which is not sufficient for modern machine learning tasks (e.g. training a deep neural network).

2.2 Extending TEEs to Accelerators

Recent studies have attempted several ways to leverage hardware accelerators to improve performance while ensuring data privacy and security. In this part, we will give a brief introduction to them and explain why it is still challenging to reach that goal.

Trusted I/O is a generic way to establish a trusted and secure I/O path to a target I/O device. Regretfully, SGX lacks support for generic trusted I/O; SGX does not support running privileged code in an enclave, either, which is important for managing I/O devices. Although recent works [13, 22] have proposed trusted path architectures for SGX, their methods compromise the security of SGX (e.g. they do not consider physical attacks).

Compared with the I/O devices without device memory (e.g. a keyboard), accelerators have a larger attack surface because of their own code and data stored in memory. Thus, it is difficult to use accelerators securely by just applying generic trusted I/O. Instead, designing a trusted path to the accelerator specifically is a more promising way.

HIX [8] is a hardware/software architecture to protect the I/O path between user software and an unmodified commodity GPU. To achieve this, HIX changes the CPU's hardware architecture in order to 1) provide a specific enclave for the GPU driver and 2) protect MMIO accesses to the GPU. As a result, it provides protection against privileged software. However, HIX assumes the whole graphics card (including the GPU memory) and the whole hardware system (including buses) are trusted. Therefore, the assumption suggests that HIX cannot prevent physical attacks (such as eavesdropping on the bus) yet.

Instead of customizing the CPU, Graviton [21] modifies the GPU to support TEEs on GPUs. In particular, it manages to provide secure "contexts" on the multi-kernel GPU by customizing the GPU's command processor and addresses the problem of physical snooping attacks. The design is based on the assumption that the GPU has on-package memory (e.g. HBM) which is within the trust boundary. This assumption is appropriate for modern server-class GPUs. However, many AI accelerators use off-package memory which is commonly assumed to be untrusted, so the design of Graviton cannot be directly adapted to AI accelerators.

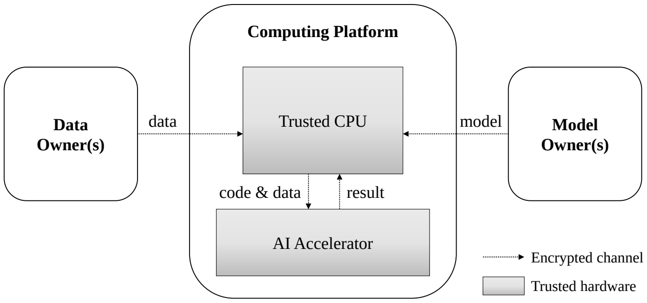

Figure 1: High-level overview of our solution for privacy-preserving machine learning.

Slalom [20] provides a novel framework to outsource matrix multiplication to an untrusted hardware accelerator, in order to accelerate deep neural network (DNN) inference, based on existing trusted CPUs. To ensure privacy and integrity, it leverages two cryptographic schemes, i.e. additive stream cipher and Freivalds' algorithm. However, Slalom has the following limitations: 1) It omits the model privacy with respect to the server. 2) As the authors acknowledge, applying Slalom to DNN training is hard because of the limitation of algorithm. 3) In terms of performance, Slalom requires a pre-processing phase but regretfully, the paper has reported neither the run time of it nor the end-to-end latency.

Ideally, we wish to leverage the AI accelerator for versatile machine learning tasks (including training) efficiently while ensuring strong security. Among the above solutions, Graviton is the closest to this goal. Inspired by Graviton, we will customize AI accelerators in order to extend TEEs to the accelerator for efficient privacy-preserving machine learning.

3 METHODOLOGY

In this section, we will detail the proposed solution for privacy-preserving machine learning, where there are data owners, model owners and a computing platform. In this scenario, the data/model owner wishes to use the computing platform (which can be a third party or one of the data owners or the model owners) to accomplish a machine learning task without disclosing their data/model. To achieve this efficiently, our solution leverages the trusted CPU and the customized AI accelerator inside the computing platform. After the data owners and the model owners send their data/model into the CPU enclave through encrypted channels, the computation is offloaded to the accelerator securely.

The whole procedure is depicted in Figure 1. In the following part, we will define our threat model first, describe the procedure of trust establishment and our protection method, and then analyze the performance overhead of our solution.

3.1 Threat Model

We consider the computing platform can be compromised by an adversary, who can control the entire software/hardware system.

For hardware, only 1) the CPU package and 2) the accelerator package are trusted. They are considered the only secure regions of the computing platform. All the other hardware, including memory, storage, peripheral devices, may be compromised. We consider the adversary can

- eavesdrop on the system bus and the PCIe bus
- directly access the main memory via a malicious device
- make man-in-the-middle attacks between the CPU and the accelerator
- eavesdrop on, tamper with or directly access the device memory of the accelerator if the AI accelerator has untrusted device memory (e.g. off-package device memory)

For software, everything except the program running in the enclave is untrusted. Malicious software is able to compromise the operating system or the hypervisor to run at the highest privilege level, so the adversary can

- directly access any part of the main memory except the PRM (processor's reserved memory) region
- access and send commands to the accelerator via MMIO
- compromise the driver and then invoke or tamper with driver APIs if the driver is not running in the enclave[1]

The adversary wishes to steal the data/model on the computing platform. The goal of our solution is to prevent the adversary from this and ensure the integrity of code and data. The size of the data/model, the run time and the memory access pattern are not considered sensitive in this paper. Since the unmodified trusted CPU is included in our solution, this work will not prevent existing vulnerabilities of the trusted CPU, so side-channel attacks and denial-of-service attacks are out of our scope.

3.2 Trust

To establish trust between two parties, there are existing methods in the practice of trusted computing (such as TPM and Intel SGX). In our solution, we adopt these methods (generally known as remote attestation [4]) to establish trust between the data owner (or the model owner) and the CPU enclave. As a result, both the data and model owners can trust 1) the enclave is a genuine secure environment provided by the trusted CPU, 2) the attested software is correctly running in this enclave and 3) their data and model have been uploaded to the enclave securely and integrally.

[1] E.g. Intel SGX does not support kernel-mode enclaves.

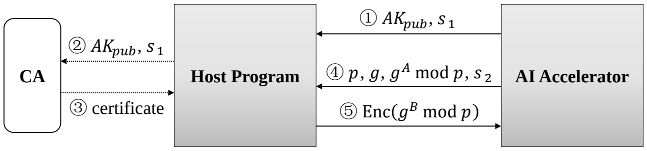

Figure 2: Trust establishment between the host program and the AI accelerator.

Next, to extend the trusted boundary from the CPU enclave to the AI accelerator, we borrow the concept of remote attestation and detail our method depicted in Figure 2.

3.2.1 Authentication. The first step is authentication, where the AI accelerator proves itself to the trusted software running in the CPU enclave (denoted by the "host program" in the following part) that it is a genuine trusted accelerator. The process of authentication is based on public-key cryptography. Specifically, each trusted accelerator is assigned with a unique pair of endorsement keys (EK_{pri}, EK_{pub}) when it is manufactured. The private key EK_{pri} is burned to the accelerator hardware and it should be only known by the accelerator. The public key EK_{pub} is maintained by the manufacturer's certificate authority (CA). Then, each time the host program wants to use the accelerator, the accelerator generates a different pair of attestation keys (AK_{pri}, AK_{pub}), and then sends the public key AK_{pub} and an EK_{pri}-signed signature $s_1 = \text{Sign}(EK_{pri}, AK_{pub})$ to the host program.

The host program should verify whether AK_{pub} is bound to a genuine trusted accelerator. For this purpose, it sends the received AK_{pub} and s_1 to the manufacturer's CA, and then the CA uses EK_{pub} to verify the signature. If s_1 matches AK_{pub}, the CA issues a certificate and returns it to the host program. Once the host program receives the certificate, it can confirm that AK_{pub} is generated by the trusted accelerator.

3.2.2 Key exchange. The second step is key exchange, the purpose of which is to establish a shared symmetric key between the AI accelerator and the host program. Based on the trust established in the first step, we use AK as the signing key for ephemeral Diffie-Hellman key exchange. Specifically, the accelerator generates a prime number p, a primitive root g and a random number A, and then transmits p, g, g^A mod p and an AK_{pri}-signed signature $s_2 = \text{Sign}(AK_{pri}, p||g||g^A \bmod p)$ to the host program. After the host program receives them and verifies the signature with AK_{pub}, it generates a random number B and then sends Encrypt($AK_{pub}, g^B \bmod p$) to the accelerator. In the end, both the host program and the accelerator can calculate g^{AB} mod p as their shared secret and can derive a symmetric key K from the shared secret by a key derivation function.

3.3 Protection

On the basis of the shared symmetric key, we propose the following data protection methods to prevent the adversary from obtaining the data/model when the host program offloads the computation to the AI accelerator.

3.3.1 Overview. First, we protect the code and data transmitted between the host program and the accelerator. Generally, the code and data are placed from the host program's memory space to the "off-chip DRAM", which denotes the memory that is not part of the CPU nor the accelerator package. Then, the accelerator fetches the code from the off-chip DRAM and accesses the off-chip DRAM to get/put run-time data. Considering the adversary can read and write the off-chip DRAM device memory and the buses can be comprised, we should not disclose the code and data in such insecure regions. Formally, we define:

Rule 1. *The code and data should be encrypted and integrity-protected during the transmission between the host program and the off-chip DRAM.*

Rule 2. *The code and data should keep encrypted and integrity-protected in the off-chip DRAM.*

Based on these rules, we employ authenticated encryption, which is a combination of encryption and message authentication, to ensure both confidentiality and integrity of the transmitted messages. In particular, the host program encrypts the code and data in the CPU enclave with the symmetric key K, calculates the message authentication code (MAC) of them with K and then writes the ciphertext and the MAC to the off-chip DRAM[2]. To use the code and data, the accelerator decrypts them and verifies their integrity with the MAC. Symmetrically, after calculating the result, the accelerator encrypts the result and calculates its MAC, and then writes the ciphertext and the MAC to the off-chip DRAM. The host program can get the result after decryption and verification.

3.3.2 Detail. However, to achieve efficiency as well as security, some issues need to be carefully considered:

- Where is the decrypted code and data placed?
- How to protect run-time data stored in the off-chip DRAM?

If the AI accelerator has its own trusted on-package device memory, the answer is simple: It can just put the decrypted code and data in the trusted device memory and directly store the run-time data without encryption or integrity protection.

If not, the accelerator can never put unencrypted messages in the off-chip DRAM according to Rule 2. It is also impractical to store them in the accelerator's limited on-chip storage.

[2] The MAC can be also written to the accelerator's register.

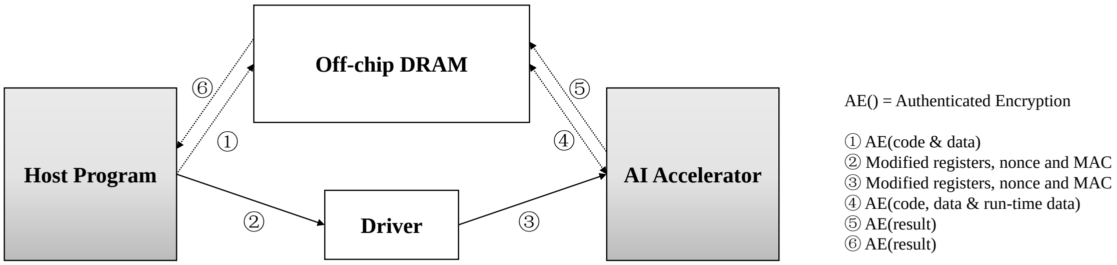
Off-chip DRAM
Host Program
Driver
AI Accelerator
①
②
③
④
⑤
⑥
AE() = Authenticated Encryption
① AE(code & data)
② Modified registers, nonce and MAC
③ Modified registers, nonce and MAC
④ AE(code, data & run-time data)
⑤ AE(result)
⑥ AE(result)

Figure 3: Methods for protecting code, data and programmer-visible registers transmitted between the host program and the AI accelerator.

Therefore, the accelerator must decrypt the code and data "on demand". In particular, each time the accelerator fetches a piece of code or data from the off-chip DRAM to the accelerator's on-chip storage (e.g. SRAM buffers or registers), it fetches the corresponding piece of ciphertext and decrypts it inside the accelerator. To achieve this efficiently, we require a counter mode–based authenticated encryption scheme such as AES-GCM and AES-CCM. Taking AES-GCM for example, after the accelerator fetches the piece of ciphertext, the ciphertext is XORed with $AES(counter)$ inside the accelerator to get the plaintext. This procedure avoids data dependency and thus it is friendly to hardware implementation.

Integrity is another challenge because the adversary can modify the code and data in the untrusted off-chip DRAM at run time. Thus, the accelerator needs to verify integrity each time it accesses the off-chip DRAM. However, it is extremely inefficient to repeatedly verify whether the whole code and data match the MAC. To tackle this problem, we propose the following scheme:

- Before the host program applies authenticated encryption to the code and data, it divides the whole code and data into m pieces, and the size of each piece does not exceed s.
- Then the host program calculates their respective ciphertext and MAC and then stores these m pieces of ciphertext and MAC in the off-chip DRAM.
- In this case, each time the accelerator accesses the off-chip DRAM, it only fetches corresponding pieces and verifies whether they match their MAC.

A smaller s means a finer granularity and causes a lower latency of the cryptographical computation. However, it also means a larger m and leads to a larger memory consumption (for the m pieces of MAC) as well as an increase in the accelerator's DRAM access. As a result, the value of m should be the trade-off between computation and memory access.

The run-time data should also be protected when the AI accelerator has no trusted device memory. Specifically, according to Rule 2, the run-time data should be encrypted by counter-mode encryption before they are written to the off-chip DRAM, and their integrity should be protected by the method described in the previous paragraph.

3.3.3 *Register state.*

So far we have completed the protection of the code and data in the off-chip DRAM to ensure that the adversary cannot learn or tamper with them. However, the information that is stored in the AI accelerator's programmer-visible registers (e.g. address registers and control registers) has not been protected. Specifically, if the host program wishes to use the AI accelerator, it invokes the accelerator's driver to write the accelerator's registers through MMIO. In our threat model, the adversary is able to directly access the registers through MMIO. Thus, we have the following rule:

Rule 3. *The integrity and freshness of the programmer-visible registers of the AI accelerator should be guaranteed during the transmission between the host program and the accelerator.*

Based on this rule, we propose to use message authentication code (MAC) to prevent the adversary from tampering with the registers and use nonces to prevent replay attacks. Note that the kernel-mode driver may not be running in the CPU enclave, so we let the runtime (also known as the user-mode driver) in the enclave to maintain the register state. Each time the host program modifies a register, it also calculates the MAC of the whole register state and the nonce with the symmetric key K. After the host program writes the modified registers, it writes the MAC (as well as the nonce) to a specific register of the accelerator. All the writes are done by invoking the kernel-mode driver. Since only the host program and the accelerator know K, the accelerator can verify the integrity of the register state and the driver cannot counterfeit the MAC. Therefore, this problem is solved even if the driver is not within the trust boundary (i.e. the driver is untrusted). To sum up, Figure 3 depicts our protection methods.

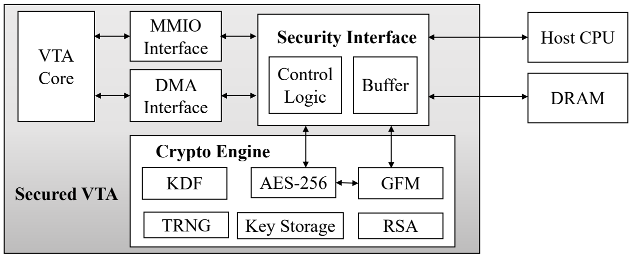

Figure 4: We customize the VTA by adding a security interface and a crypto engine.

3.4 Overhead

Analytically, the total run time of a machine learning task is composed of computation and memory access. Compared with conventional AI accelerators, our methods do not affect the computation (e.g. matrix multiplications). In contrast, our protection methods additionally introduce encryption, decryption, and message authentication, all of which are bound to memory access. Each time the trusted accelerator fetches a piece of code/data from the off-chip DRAM, the accelerator has to decrypt it and verify its MAC before using it. Each time the trusted accelerator aims to write a piece of data to the off-chip DRAM, the accelerator has to encrypt it and calculate its MAC. As a result, the slowdown will be significant if the workload involves a large amount of memory access. On the contrary, a small amount of memory access will lead to slight slowdowns. Therefore, the extent of the performance overhead depends on the memory access intensity of the workload.

4 CASE STUDY: VTA

In this section, we demonstrate the process of customizing a trusted AI accelerator using the proposed methods. In particular, we implement the customization on the open-source versatile tensor accelerator (VTA) [17] and evaluate its impact on the performance of the VTA.

4.1 Architecture

Referring to Figure 1, we deploy a trusted CPU and a customized VTA in the computing platform, and we have the model/data owners upload their private model and data to a CPU enclave securely. Then, the host program (a trusted and attested piece of software running in the enclave) compiles the model into VTA code just in time with the TVM deep learning compilation stack [1], and then offloads the code and data to the VTA and waits until the result is calculated following the proposed protection methods. To implement support for the protection methods as well as the trust establishment methods on the VTA, both hardware and software of the VTA are modified.

In terms of hardware, we do not change the design of the VTA core[3]. Instead, a security layer, including a security interface and a crypto engine, is added between the VTA core and the host CPU/DRAM (Figure 4) to handle the cryptographic functionalities. The security interface buffers the received data in an on-chip buffer (2 KB), communicates the data to the crypto engine, and then sends the encrypted/decrypted data to the DRAM/VTA. The crypto engine contains components (such as AES, RSA, and TRNG) that are used for authenticated encryption and trust establishment. In this experiment, we implement the AES-256 module that employs a pipelined structure in which encryption/decryption of a 128-bit plaintext/ciphertext takes 29 clock cycles[4], and the GFM (Galois-Field Multiplication) module that incurs 8 clock cycles for the authentication of each 128-bit text[5].

The modification of software only involves the VTA runtime, while the instruction set architecture of VTA or the just-in-time compiler does not need to change in our solution. The VTA runtime (`vta/src/runtime.cc`) is part of the host program which acts as an interface to the untrusted off-chip DRAM and the kernel-mode driver. Specifically, the original VTA runtime is responsible for allocating a few buffers in the off-chip DRAM, putting the code (namely "kernel" in VTA) and data into corresponding buffers, and invoking the driver to launch the kernel. In our solution, the runtime is modified such that it can 1) apply authenticated encryption to the code and data, and 2) maintain the register state to calculate MAC of the registers and nonces. The memory layout remains the same because counter-mode encryption does not change the size of messages, but authenticated encryption results in extra metadata (i.e., nonces and GMAC) in addition to the original message. The metadata is stored in a newly-allocated buffer in the off-chip DRAM.

4.2 Evaluation

We build a cycle-accurate simulation environment based on VTA's TSIM[6], which can compile the register-transfer-level design into `libvta_hw.so` and integrate this simulation library into the TVM stack to evaluate run time in clock cycles.

Since the VTA is mainly designed for computer-vision tasks [17], we first test the computation of 2D convolution layers and fully connected layers, which represent the dominant operators in most computer vision deep neural networks. Specifically, we use two convolutional layers and two fully connected layers of AlexNet [11], which have similar amount of computation, as our benchmarks: Conv4 (384

[3] https://github.com/apache/incubator-tvm/tree/v0.6/vta
[4] https://opencores.org/projects/tiny_aes
[5] https://opencores.org/projects/gcm-aes
[6] https://github.com/apache/incubator-tvm/tree/v0.6/vta/apps/tsim_example

Table 1: Latency in clock cycles (and slowdowns compared with the original VTA).

	VTA	VTA-trusted	VTA-ctr
Conv4	2 782 962	2 988 247 (1.074×)	2 872 727 (1.032×)
Conv5	1 879 117	2 083 659 (1.109×)	1 969 399 (1.048×)
FC1	5 418 983	29 300 635 (5.407×)	6 016 817 (1.110×)
FC2	2 412 609	13 034 043 (5.402×)	2 682 866 (1.112×)
ResNet-18	29 964 469	32 338 145 (1.079×)	30 238 890 (1.009×)

input channels, 256 output channels), Conv5 (256 input channels, 256 output channels), FC1 (9216 inputs, 4096 outputs) and FC2 (4096 inputs, 4096 outputs). For each benchmark, the run time of three configurations are evaluated: 1) the original VTA without any protection (denoted by VTA), 2) the VTA with full protection (including confidentiality, integrity and freshness, denoted by VTA-trusted), and 3) the VTA with the protection of only confidentiality (implemented by AES-CTR, denoted by VTA-ctr), which is used for profiling.

The experimental result is shown in Table 1. Compared with the original VTA, VTA-trusted results in 1.074×–1.109× slowdowns for convolutional layers but results in 5.402×–5.407× slowdowns with respect to fully connected layers. It is noticed that the slowdowns with respect to different types of layers vary significantly. We will explain this as follows.

There are two main factors that lead to the significant overhead of computing fully connected layers. The first one is memory access intensity. As we discussed in Section 3.4, the extent of the performance overhead depends on the memory access intensity of the workload. For a fully connected layer (with only 1 batch), the dominant operator is vector-matrix multiplication. However, the memory access intensity of the vector-matrix multiplication is about 1 word/FLOP, which is notably higher than 2D convolution (approximately $\frac{1}{H_o \times W_o}$ word/FLOP).

The second factor stems from the implementation of the GFM module which is used for calculating GMAC. Specifically, we tested a configuration without the GFM module (i.e. VTA-ctr) for profiling. As the result shows, removing the GFM module results in a much better performance for fully connected layers compared with VTA-trusted. The slowdowns against the original VTA are reduced from 5.402×–5.407× to 1.110×–1.112×. We notice that the implementation of the AES module is pipelined and counter-mode encryption avoids data dependency, so it is feasible to achieve high throughput. As a result, VTA-ctr only incurs 1.032×–1.112× slowdowns. However, our implementation of the GFM module is not pipelined and the calculation of GMAC has strong data dependency. Thus, authenticating a piece of data of s bits needs $\lceil \frac{s}{128} \rceil \times 8$ clock cycles in our implementation.

4.3 Potential Improvement

Based on the observations above, we propose several potential ways to improve the trusted VTA in order to reduce slowdowns.

- To reduce the memory access intensity, we can optimize the machine learning model at algorithm level, such as reducing/avoiding fully connected layers, increasing the batch size, using sparse neural networks, etc.
- To reduce the memory access intensity, we may also optimize the computation at the compiler level. For example, recent deep learning compilers are exploiting dedicated optimization to achieve efficient model-to-hardware mapping (e.g. TVM).
- To reduce the overhead of message authentication, optimizing hardware implementation of the authentication module (e.g. designing a low-latency GFM module [15]) is a straightforward way.
- To reduce the overhead of message authentication, using other authentication schemes may be a potential direction. For example, the Merkle tree can reduce the computation complexity of authenticating s-bit data from $O(s)$ to $O(\log s)$ because of its potential for parallelism. Theoretically, the best effect (i.e. upperbound) that this method can achieve is the result of VTA-ctr, which has been shown in Table 1.

Fortunately, the current version of TVM (v0.6) has supported end-to-end compilation for ResNet-18 [6] on VTA with compilation optimization (support for other models is still under development). Thus, we take ResNet-18 as an additional benchmark, whose result is shown in Table 1. As we can see, the slowdown of VTA-trusted is satisfactorily small (1.079×), which is attributed to the low memory access intensity after compilation optimization. In addition, the slowdown of VTA-ctr (1.009×) indicates a large space for accelerating message authentication.

5 CONCLUSION

This paper proposes an efficient solution for privacy-preserving machine learning with AI accelerators. The key innovation lies in incorporating a customized AI accelerator into the computing platform and ensuring its security. With the help of the proposed methods, the users can upload their private model and data to the computing platform securely and calculates the result efficiently within the AI accelerator. The case study shows this solution is effective on the versatile tensor accelerator at a small design cost and moderate performance overhead, and it is promising to use this solution in the industry with lower overhead.

REFERENCES

[1] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 578–594.

[2] Yu-Hsin Chen, Joel S. Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *International Symposium on Computer Architecture (ISCA)*. 367–379.

[3] Intel Corporation. 2016. *Intel® 64 and IA-32 Architectures Software Developer Manuals*.

[4] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016 (2016), 86.

[5] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin E. Lauter, Michael Naehrig, and John Wernsing. 2016. CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy. In *International Conference on Machine Learning (ICML)*, Vol. 48. 201–210.

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

[7] Tyler Hunt, Congzheng Song, Reza Shokri, Vitaly Shmatikov, and Emmett Witchel. 2018. Chiron: Privacy-Preserving Machine Learning as a Service. *CoRR* abs/1803.05961 (2018).

[8] Insu Jang, Adrian Tang, Taehoon Kim, Simha Sethumadhavan, and Jaehyuk Huh. 2019. Heterogeneous Isolated Execution for Commodity GPUs. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 455–468.

[9] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. 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 *International Symposium on Computer Architecture (ISCA)*. 1–12.

[10] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. 2018. GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In *USENIX Security Symposium*. 1651–1669.

[11] Alex Krizhevsky. 2014. One Weird Trick for Parallelizing Convolutional Neural Networks. *CoRR* abs/1404.5997 (2014).

[12] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanovic, and Dawn Song. 2020. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *European Conference on Computer Systems (EuroSys)*.

[13] Hongliang Liang, Mingyu Li, Yixiu Chen, Lin Jiang, Zhuosi Xie, and Tianqi Yang. 2020. Establishing Trusted I/O Paths for SGX Client Systems With Aurora. *IEEE Transactions on Information Forensics and Security* 15 (2020), 1589–1600.

[14] Arm Limited. 2009. *ARM Security Technology Building a Secure System using TrustZone Technology.*

[15] Yang Lu, Guochu Shou, Yihong Hu, and Zhigang Guo. 2009. The Research and Efficient FPGA Implementation of Ghash Core for GMAC. In *International Conference on E-Business and Information System Security (EBISS)*. 1–5.

[16] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *IEEE Symposium on Security and Privacy (S&P)*. 19–38.

[17] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Eddie Q. Yan, Lianmin Zheng, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2019. A Hardware-Software Blueprint for Flexible Deep Learning Specialization. *IEEE Micro* 39, 5 (2019), 8–16.

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

[19] Bita Darvish Rouhani, M. Sadegh Riazi, and Farinaz Koushanfar. 2018. Deepsecure: Scalable Provably-Secure Deep Learning. In *Design Automation Conference (DAC)*. 2:1–2:6.

[20] Florian Tramèr and Dan Boneh. 2019. Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware. In *International Conference on Learning Representations (ICLR)*.

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

[22] Samuel Weiser and Mario Werner. 2017. SGXIO: Generic Trusted I/O Path for Intel SGX. In *Conference on Data and Application Security and Privacy (CODASPY)*. 261–268.