Secure MLaaS with TEMPER: Trusted and Efficient Model Partitioning and Enclave Reuse

Fabing Li*
Xi'an Jiaotong University
Xi'an, China
Institute for Interdisciplinary
Information Core Technology, Xi'an
Xi'an, China
lifabing@stu.xjtu.edu.cn

Xiang Li
Tsinghua University
Beijing, China
lixiang20@mails.tsinghua.edu.cn

Mingyu Gao
Tsinghua University
Beijing, China
Shanghai Artificial Intelligence Lab
Shanghai, China
Institute for Interdisciplinary
Information Core Technology, Xi'an
Xi'an, China
gaomy@tsinghua.edu.cn

ABSTRACT

Machine Learning as a Service (MLaaS) is becoming a highly available and cost-efficient way to embrace machine learning techniques in various domains. But it suffers from data privacy risks as user data must be uploaded to untrusted clouds. We propose a trusted and efficient MLaaS system, TEMPER, based on secure hardware enclaves such as Intel SGX. TEMPER significantly improves the performance without sacrificing the data security guarantees or the model inference accuracy. With the two key techniques of enclave reuse and model partitioning, it reduces the enclave initialization and model loading costs, and alleviates the secure paging overheads due to the limited hardware-protected memory capacity in SGX. We also provide rigorous security guarantees for enclave sharing and batched processing, by ensuring stateless, non-interference, and data-oblivious processing and data transfers across model partitions. TEMPER achieves on average 2.2× and 1.8× improvements over the state-of-the-art designs for latency and throughput, respectively, and within 2.1× slowdown of untrusted native execution. Its distributed paradigm provides a more scalable way for future MLaaS with large models.

ACM Reference Format:
Fabing Li, Xiang Li, and Mingyu Gao. 2023. Secure MLaaS with TEMPER: Trusted and Efficient Model Partitioning and Enclave Reuse . In *Annual Computer Security Applications Conference (ACSAC '23), December 04–08, 2023, Austin, TX, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3627106.3627145

1 INTRODUCTION

Machine learning (ML) has become an important and ubiquitous technique in modern data-driven applications [54]. However, both developing a well-tuned ML algorithm and deploying an end-to-end hardware/software ML system are difficult and burdensome, especially for general practitioners in other domains. As a result, with the benefits in availability, usability, and cost efficiency, Machine Learning as a Service (MLaaS) becomes popular [1, 3–5, 7], which provisions ML as cloud services and allows customers to process data without algorithmic expertise or local system infrastructures.

However, existing MLaaS offerings pull user data to cloud servers for storing and processing. This raises serious privacy concerns and critical security risks if the data are sensitive, such as medical or financial records. For example, the data may not be allowed to leave the user's machine due to policy restrictions; the user may worry about the cloud platforms being compromised; or even the user is not willing to trust the service provider in the first place. Any of these privacy reasons could become a showstopper for MLaaS.

To realize *secure MLaaS*, currently there are two major directions. Cryptographic algorithms such as homomorphic encryption and multi-party computation can be applied for privacy-preserving ML [11, 15, 24, 49, 59, 64, 68, 72]. But their extremely high computation cost limits the capability and the performance, making them impractical for latency-sensitive MLaaS. The alternative approach, which relies on hardware-based trusted execution environments (TEEs) like Intel SGX or ARM TrustZone [9, 20], processes private data inside trusted hardware *enclaves* on untrusted platforms [32, 34, 43, 44, 52, 53, 55, 57, 61, 68, 85, 95]. As the computations running inside the enclaves are unchanged, the performance is close to native execution, making it promising for secure MLaaS.

Nevertheless, existing TEE-based MLaaS frameworks still suffer from several performance inefficiencies. Since different users are mutually distrusted, each user must request a separate enclave and establish trust with it through remote attestation. The newly constructed enclave also needs to spend substantial time in building up the ML model and loading the weight data. These *enclave initialization* and *model loading* costs are significant, especially compared with ML inference tasks on small and medium models. On the other hand, for large ML models, the runtime data volume typically exceeds the available hardware-protected memory capacity, and results in excessive *secure paging* operations that swap the confidential data to the untrusted memory space with expensive encryption, authentication, and data copy. While model quantization [53] and careful memory planning [52, 57] may help alleviate these overheads, ML is likely to continuously use larger models [14], and eventually requires a more fundamental solution.

*Work done during the internship at Institute for Interdisciplinary Information Core Technology, Xi'an.

ACSAC '23, December 04–08, 2023, Austin, TX, USA

ACM ISBN 979-8-4007-0886-2/23/12.
https://doi.org/10.1145/3627106.3627145

In this paper, we propose Temper, a trusted and efficient MLaaS system based on Intel SGX processors, with two key techniques: *enclave reuse* and *model partitioning*. Temper optimizes the *performance* of cloud-side MLaaS processing, without sacrificing the *data security guarantees* or the *model inference accuracy*. Temper creates multiple long-running enclaves and shares them across different users, rather than repetitively launching new ones. The model is divided among these enclaves, and all partitions together process in a pipelined manner. We use efficient partitioning strategies that optimize for either throughput or latency based on system loads. Both the computation latency impact due to enclave secure paging and the communication latency overheads between separate enclaves in different machines are considered and balanced using an accurate latency estimation model and a lightweight optimizer. Temper also utilizes the state-of-the-art TVM compiler [18] to generate high-performance programs. In these ways, Temper significantly improves performance.

To guarantee security, Temper uses a customized yet lower-cost attestation protocol between the user and a special leader enclave in each model. The leader is responsible for initializing and attesting all partition enclaves, and collecting requests from the same or different mutually-distrusted users to form data batches. The partition enclaves enable faithful sharing between users through stateless, non-interference, and oblivious processing. We ensure Temper is free of common side-channel attacks, by enforcing data-oblivious processing and transfers in the partition enclaves, and making the leader enclave program not depend on private user data except for encryption/decryption which has well-studied secure implementations. We also minimize trusted components in the system by cautiously offloading request scheduling, resource management, and fault tolerance to the untrusted domain.

We evaluate Temper on a wide range of small and large deep neural network models. Compared with state-of-the-art SGX-based designs [44, 57], Temper improves the cloud-side inference latency at batch size 1 by 2.2×, and achieves within 2.1× slowdown of untrusted native execution. The throughput improves by 1.8× and 2.1× when using batch sizes of 1 and 4, respectively. These benefits come from both reduced initialization and model loading costs, as well as reduced securing paging overheads. We also demonstrate that the attestation protocol and the partitioning strategies in Temper are efficient. In summary, Temper makes a significant step towards trusted and efficient MLaaS, and the distributed nature in Temper represents a more scalable way to support even larger ML models in the future [14].

2 BACKGROUND AND MOTIVATION

This work focuses on performance optimizations of secure machine learning inference on public clouds that utilize trusted processors such as Intel SGX [20]. We first introduce relevant background to motivate our work, followed by a detailed summary of the target threat model.

2.1 Machine Learning and Neural Networks

Machine learning (ML) is one of the most rapidly developing fields today. Typical ML techniques *train* statistical models for specific applications on pre-collected datasets to update the model parameters, a.k.a., *weights*, until convergence. The trained models are then used for *inference*, i.e., predicting results for new data. During training and inference, multiple input data could be combined into *batches* to improve the convergence speed and processing throughput.

In recent years, deep learning with neural networks (NNs) has become the most widely used ML algorithm because of their supreme efficacy [54]. An NN model is a directed acyclic graph (DAG) of multiple layers. The model hyperparameters, such as the layer types and their topologies, depend on the target applications. There are usually convolution, matrix multiplication, pooling, batch normalization, and various element-wise layers, which are connected as a linear chain [79] or with complex patterns such as branches [81] and residual links [36].

The capability and the accuracy of NNs normally improve with deeper topologies and larger layers [36, 79], but come with the cost of higher execution latency. Numerous frameworks have been designed to address the performance challenge caused by larger and more complex models, to bridge the gap between productivity-centric high-level interfaces and performance-oriented low-level implementations. These frameworks include TensorFlow [8], PyTorch [71], MXNet [17], TVM [18], and many more. To support large NN models that do not fit in a single compute device, *model parallelism* partitions a model into multiple smaller parts. There are mainly two categories. Tensor partitioning splits each tensor in the model along specific dimensions and deploys them onto different devices, e.g., Tofu [90], Megatron-LM [77], and FlexFlow [47]. Graph partitioning, on the other hand, groups adjacent layers in the model and applies pipelining between them, such as PipeDream [67], GPipe [41], RaNNC [82], and DNN-Partition [83].

2.2 MLaaS and Data Privacy Concerns

While ML is now ubiquitous in nearly all application domains, neither developing a customized algorithm nor deploying a hardware/software system is easy for non-experts. To alleviate such burdens, ML as a Service (MLaaS) emerges as a new paradigm that relies on public clouds to provide training and/or inference services, currently offered by Amazon AWS [4], Google Cloud [5], Microsoft Azure [7], Alibaba Cloud [1] and Baidu AI cloud [3]. In common scenarios, the cloud service providers train the models with their intelligent properties such as private datasets, and offer the inference services to which the users can simply send their data and obtain results. MLaaS saves the users from the burdens of building ML models and maintaining computer servers locally, therefore greatly reducing the cost of embracing ML.

However, one major obstacle for MLaaS is the concern of data privacy. The service providers own the MLaaS platforms, and care about their intelligent properties such as the trained model weights. On the other hand, the users would worry about leaking their sensitive data, such as medical documents and financial records, if sending to the clouds. Even if the service providers themselves may not have direct commercial interests in these data, the public cloud platforms are currently experiencing various attacks every day [33]. Therefore the MLaaS platforms are fundamentally untrusted to the users (detailed threat model in Section 2.5).

Secure MLaaS can now be realized in two different ways. First, there are a wide range of solutions based on modern cryptographic

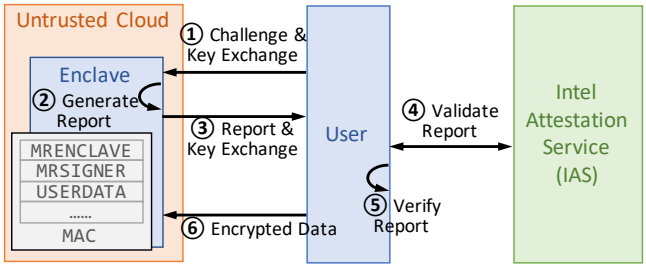

Figure 1: An SGX enclave on the untrusted cloud and the remote attestation procedure.

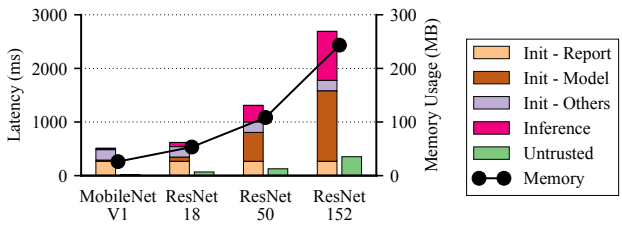

Figure 2: Performance overheads of baseline secure MLaaS.

primitives such as homomorphic encryption and multi-party computation [11, 15, 24, 49, 59, 64, 68, 72]. But their performance overheads are extremely large, usually several orders of magnitude slower than the equivalent insecure computations, making them impractical. The alternative approach leverages trusted execution environments (TEEs) provided by hardware [32, 34, 35, 43, 44, 52, 53, 55, 57, 61, 69, 85]. The TEEs are able to isolate the sensitive data from the untrusted cloud platforms. This approach can avoid the high overheads from the cryptographic primitives and achieve reasonable performance, promising for secure MLaaS. Next we introduce them in more details.

2.3 Trusted Processors: Intel SGX

Various TEE techniques have been proposed on CPUs, GPUs and FPGAs [2, 6, 9, 20, 21, 26, 42, 46, 89, 96]. Among them, Intel Software Guard Extensions (SGX) is the most prevalent, and available since the Skylake architecture [20, 45]. SGX introduces a set of instructions to create *enclaves* [38, 63], which ensure confidentiality and integrity of the code and data inside them from all external, untrusted privileged and unprivileged software, including operating systems, hypervisors, and other user programs.

SGX provides *attested*, *isolated*, and *sealed* computations on the processor, thus creating a trusted environment to the remote user on an untrusted cloud server (Figure 1). A remote *attestation* procedure first establishes the trust between a remote user and the enclave by checking whether the enclave is launched on a genuine processor and cryptographically measuring the integrity of each component inside the enclave. A secure communication channel is also constructed between the two parties. More specifically, as shown in Figure 1, the remote user starts the attestation by issuing a challenge to the processor, including the key exchange message and a nonce for freshness ①. The enclave calculates a couple of SHA-256 hash values of its initial code and data contents, and of its author (e.g., Intel). These measurements of the enclave, known as `MRENCLAVE` and `MRSIGNER`, together with the key exchange message responded by the enclave, are put into a report and signed by the processor's private attestation key ②.[1] The signed report is sent back to the user, with a certificate that authenticates the public attestation key ③. The user can verify the attestation key by querying the Intel Attestation Service (IAS) which acts as a certificate authority (CA) ④. Then the user verifies the report and checks whether the enclave has been initialized into a trusted state ⑤. If the integrity check passes, the user sends her private input to the enclave through the newly established secure communication channel, and starts the processing ⑥.

Once initialized, the enclave's execution is protected by the processor hardware, and *isolated* from the operating system (OS) and other software. The enclave uses a special hardware-guarded memory space, called *enclave page cache* (EPC). This ensures that any untrusted software, even privileged on the same physical processor, cannot read or modify the contents of the enclave. If the data exceed the EPC size, SGX uses *secure paging* to evict less often accessed pages into the untrusted memory, similar to classical OS paging. The data are automatically encrypted and authenticated when swapped out of the enclave, ensuring confidentiality and integrity. Secure paging results in large performance penalty and causes several times of slowdown [10, 91]. Hence, it is highly desired to fit the enclave data within the EPC capacity.

This work primarily focuses on Intel SGX1 with a limited EPC capacity. Newer TEE technologies start to support larger EPC sizes, but have additional security issues [56, 65, 66]. Intel TDX [6] and AMD SEV [2, 50, 51] provide basic integrity protection without data freshness checks, so they may suffer from hardware-based memory replay attacks. PENGLAI proposed a scalable memory integrity solution using hardware-assisted mountable Merkel trees [26], but relied on RISC-V processors which are not yet widely available for MLaaS clouds. Generally speaking, supporting large EPC sizes on a single machine incurs significant overheads and is not scalable to larger ML models, which may need to be distributed over multiple machines anyway.

2.4 Performance Challenges and Motivations

While trusted processors like Intel SGX provide a substrate to realize secure MLaaS with much lower cost than pure cryptographic algorithms, there are several inefficiencies that require careful performance optimizations. Since different users are mutually untrusted, a common approach in previous designs is to build a new enclave for each user separately [32, 34, 52]. Figure 2 shows the latency breakdown for the execution on the cloud server side using Myelin [44], comparing to the native, untrusted execution (detailed setup in Section 7.1). We do not consider the latencies on the user side, since we focus on optimizing cloud MLaaS performance. We highlight several critical overheads below, as our motivation to develop a new secure MLaaS system that offers the same level of security guarantees with better performance.

[1] The measurement is actually calculated and signed with the help of a special "Quoting Enclave", which is a detail not affecting our system [20].

First, **enclave initialization** is slow, but is an extra cost that must be paid for *every* user in the baseline. As Figure 2 shows, the attestation process, including the report generation in the enclave, takes a few hundreds of milliseconds.[2] This is usually several factors longer than the actual inference task to perform, if with a medium-size NN such as MobileNetV1 [39].

Second, as another step during constructing a new enclave for each user, **model loading** also introduces substantial latency [52]. Many NNs use up to hundreds of MB of model weights, which require long time to be copied into the enclave [91, 92]. For example, Figure 2 shows that for large NNs like ResNet152 [36], model loading is actually the largest latency contributor, even slower than the actual inference task. If the model weights are stored in the encrypted form, decryption would further add more overheads.

Third, the **limited EPC size** is a well-known performance issue for SGX [10, 52, 91]. If the ML model cannot fit in the EPC, excessive secure paging would lead to substantial performance degradation due to data copy and encryption [10, 91]. This is not an uncommon problem for today's MLaaS, as many large NN models already use several hundreds of MB weights [36, 79], exceeding the EPC size. In Figure 2, the inference latencies for small NNs like MobileNetV1 and ResNet18 are similar in the trusted and untrusted scenarios. However, ResNet50 and ResNet152 use over 100 MB memory and exhibit more than 3× slowdown due to secure paging when running inside the enclave. Moreover, when processed with batches, the user data and the intermediate results also take significant EPC space, making the situation even worse.

To summarize, the enclave initialization and model loading introduce huge extra cost for small ML models (Figure 2 MobileNetV1 and ResNet18), while the limited EPC size results in excessive secure paging for large models (Figure 2 ResNet50 and ResNet152). These overheads lead to 5× to 25× slowdown overall, which our work aims to alleviate.

2.5 Threat Model

Our work targets to offer the same level of security as previous MLaaS designs, and therefore uses a similar threat model [32, 52]. We assume that the ML model structure and the hyperparameters are public information. However, both the service provider's model weights and the users' data are private and sensitive. The protection of the model weights is ensured by the fact that the service provider owns the platform and controls what computations can run and what communication can happen. We do not consider algorithm-level attacks like model stealing, model inversion, backdoor, and membership inference [25, 28, 78, 86]. There exist orthogonal defenses that can be applied [13, 19, 48, 60, 74]. SGX can help alleviate some of these attacks like model replacement and gradient poisoning through attestation and isolation. We mainly focus on the protection of user data, including the input, the output, and any intermediate data through the ML models.

Relying on SGX, the processors of the MLaaS platform are trusted. Except that, the adversaries fully control any untrusted software including the OS and the hypervisor. They also physically control other hardware devices. For example, they can access data stored in the memory and the disks, or configure the network cards and switches to manipulate communication packets (e.g., replay attacks). Recent studies show that SGX is vulnerable to side-channel attacks [16, 30, 62, 76, 93, 97]. We consider the adversaries can observe and infer valuable information from the timing and volumes of data transfers between enclaves. Hence these data streams must be oblivious to defend against these side channels. We discuss the details in Section 4.3.

We do not consider denial-of-service attacks because they are contradictory to the business incentive of MLaaS providers.

[2]The end-to-end remote attestation is usually a few seconds long with communication to users. Here we only consider the server's computations.

3 TEMPER OVERVIEW

To optimize the performance of SGX-based MLaaS while not sacrificing the security, we propose Temper, which leverages *enclave reuse with solid security guarantees* to reduce the initialization and model loading overheads, and *model partitioning across multiple enclaves* to overcome the EPC size limitation on a single machine. Our goal is to improve the cloud side performance. The computations on the user side, as well as the communication among the servers, the users, and the IAS, stay unchanged. Moreover, the computations in Temper are equivalent to the original ML models when executed natively. Thus the inference accuracy is not affected.

Figure 3 illustrates the overview of Temper. The initialization of Temper happens offline without interaction with any user. It constructs a number of *model instances*. Each instance provides the inference service of a specific ML model, and one model could be deployed to multiple instances to leverage request-level parallelism. Within each instance, we partition the ML model into multiple *partitions* that run on multiple SGX-enabled processors. Each partition contains a subset of the ML model, e.g., a few layers of an NN, that fit within the processor EPC limit to avoid excessive secure paging. Such partitioning has no impact on accuracy. Different from traditional per-user enclave provisioning, the model instances in Temper will persist and reuse the same enclaves to serve different users without violating security.

At runtime, each user request is assigned to a specific model instance by the untrusted scheduler, based on the requested model type and the load balance decision. The user verifies the model instance with a special *leader* through a customized protocol. After establishing trust, the user sends the encrypted data. The leader could securely batch a pre-determined and fixed number of requests from the same or different users, and forward them to the partitions. All data transfers between the partitions are encrypted and authenticated through data-oblivious streams in a pipelined manner. In addition, we rely on the untrusted cloud infrastructure for resource management, fault tolerance, load balancing, auto-scaling, and other administration tasks, to be fully compatible with and benefit from state-of-the-art cluster management systems [12, 22, 37, 88]. These untrusted components can at most cause denial of service, but never compromise privacy (Section 4.3).

We highlight the two key innovations in Temper that overcome the three challenges described in Section 2.4.

Enclave reuse (Section 4). The enclaves of each model instance are reused to serve different users, therefore Temper avoids the high costs of **enclave initialization** and **model loading**. To also guarantee security, the leader enclave handles attestation with

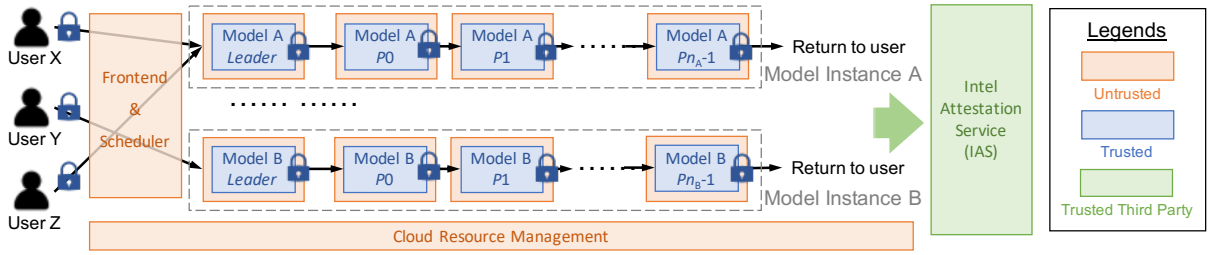

Figure 3: Overview of Temper.

the users in a customized way, and the partition enclaves ensure stateless processing and non-interference between different users. Scheduling and resource management are offloaded to the untrusted cloud platform to minimize our Trusted Computing Base (TCB).

Model partitioning (Section 5). To cope with the **limited EPC size** of an enclave, the actual ML model is partitioned into multiple enclaves to alleviate secure paging overheads. Temper supports different partitioning strategies to optimize either the overall throughput or the single-user latency, with the help of careful choices of partitioning granularities and an effective latency estimation model. Different from previous work [52, 57] that reused a shared memory space and repetitively loaded the model weights of different layers, our model partitioning technique transfers intermediate activation data between layers. In large NNs, usually the weights are much larger than the activations, and therefore Temper achieves lower communication cost.

4 SECURE ENCLAVE REUSE

While reusing the same set of enclaves for different users avoids the repetitive initialization and data loading overheads, it raises security challenges. First, in a multi-tenant scenario, each user still needs to attest these enclaves and individually establishes trust. Second, since there might be quite a few enclaves in an instance of a large ML model, we should avoid separately verifying each of them, and instead run a single combined attestation only. Third, we need to faithfully ensure that no data leakage and interference could happen across different users, regardless of whether their private data are processed in a batch or sequentially through the same enclaves. Fourth, the dataflow across enclaves should preferably be data oblivious to avoid timing side channels of communication. Finally, for long-running MLaaS, it is necessary to provide fault tolerance, by detecting failures or attacks, and reconstructing model instances if needed.

Temper addresses these issues using two types of enclaves in each model instance. The *leader* enclave handles initialization, attestation, key management and fault tolerance (Section 4.1). The multiple *partition* enclaves run stateless and non-interference inference on the private data (Section 4.2). Section 4.3 comprehensively summarizes the security guarantees of the whole system.

4.1 Leader Enclave

Each model instance has a special leader enclave with three responsibilities. First, it initializes and attests all the partition enclaves into trusted states, so that users do not need to attest them separately.

Algorithm 1: Leader program.

Input: The topology of n model partitions topo.
A list of partition measurements $M[0:n]$.

```
// Generate attestation report.
1  { k_s,pub, k_s,priv } ← KeyGen()
2  report ← GenerateReport(USERDATA = k_s,pub)
// Construct and attest each partition enclave.
3  for i ← 0 until n do
4      P[i] ← BuildEnclaveUntrusted(i)
5      if Attest(P[i], M[i]) succeeds then
6          DistributeKey(P[i])
7          BuildConnectionWithPredecessor(P[i], topo)
8      else
9          Terminate()
// Process user requests.
10 foreach user u do
       // Attestation to user.
11     SendTo(u, { report, M[0:n] })
12     BuildTrustedConnectionWith(u, k_s,priv)
       // Process data.
13     { k_sym, data } ← RecvFrom(u)
14     batch.Append({ k_sym, data })
15 SendTo(P[0], batch)
// Monitor status using a separate thread.
16 while true do
17     for i ← 0 until n do
18         if Heartbeat(P[i]) times out then Terminate()
19         if predefined time duration elapsed then
20             RevocateKey(P[i])
21             DistributeKey(P[i])
```

Second, it establishes users' trust in both itself and all the partition enclaves. Third, it continuously monitors the status of the entire model instance, periodically revocates and distributes the symmetric keys between the partition enclaves, and handles reconstruction after failure. Algorithm 1 describes the leader program.

The leader enclave starts by generating a pair of asymmetric keys $\{k_{s,pub}, k_{s,priv}\}$ that are later used to communicate with users. It then generates a normal attestation report containing the measurements of its own states. Particularly, it sets USERDATA in the report to the public key $k_{s,pub}$ (Lines 1 to 2). It then requests the untrusted cloud platform to build the partition enclaves, and attests them using a list of expected measurements. If the attestation passes, the leader distributes symmetric keys to the partitions and asks them to connect with each other according to the model topology, with secure communication channels (Lines 3 to 9).

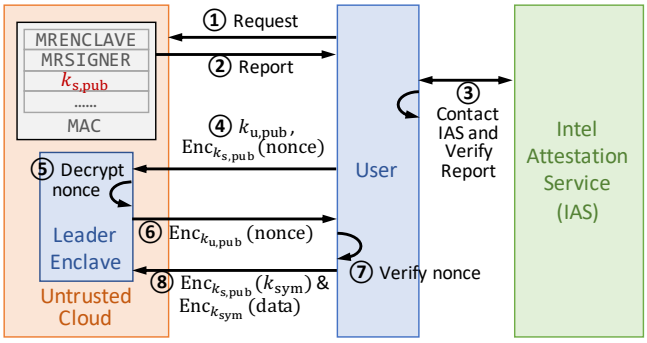

Figure 4: Building trust with the leader.

When a user is assigned to the model instance, the leader follows the procedure in Figure 4 and Algorithm 1 Lines 10 to 15. It directly replies to the user with its attestation report ①②, which the user can verify using the IAS ③. Since the report is pre-generated, it can be stored in the untrusted environment and returned without entering the enclave, with the help of a nonce check described shortly. This is much faster than the original report generation. Note that different from the standard attestation in Figure 1, the report does not include the user's nonce, so the user needs to further verify that the enclave is indeed the one that generates the report, rather than an adversary. The user uses $k_{s,pub}$ extracted from the report to encrypt a nonce, and sends to the leader together with her public key $k_{u,pub}$ ④. An authentic leader can use its private key to decrypt the nonce ⑤. It then returns the encrypted nonce under the user's public key to the user ⑥. If the returned nonce is correct, the user can trust the leader ⑦. At the same time, the leader also sends the list of partition enclave measurements to the user, so that the user can verify the partition enclave contents. Finally, the user sends the encrypted data to the leader to add into the current batch ⑧, and the leader periodically forwards the batched data to the first partition enclave and subsequently to other partitions.

The leader enclave may batch multiple data from different users. The leader receives the data from users, decrypts the data and forms the batch, which is forwarded to the later partition enclaves through their secure channels. After the batch results are returned, they are decomposed into individual results and forwarded back to the users separately. The enclave protection and the leader program integrity guarantee neither the individual users nor the outside attackers can access the user keys stored in the leader enclave.

The leader also continuously tracks the status of the entire model instance (Lines 16 to 21). After initialization, the leader keeps the communication channels with all the partition enclaves and receives regular heartbeat messages, to prevent any attacker from replacing them. The leader also regularly updates the symmetric keys between the partitions to enhance security.

We adopt a simple and flexible recovery strategy. If the leader shuts down or fails, the entire model instance is closed and waits to be restarted, because all credentials and sensitive data are gone. If any partition enclave becomes unreachable, the leader stops accepting requests, closes the connections to users, and notifies the scheduler, which tears down the entire instance. The scheduler should re-dispatch the incomplete request to another instance. Note that it is also possible to keep the good enclaves in the instance and only restart the failed ones. However we find that the saved initialization and attestation cost is insignificant compared to the sufficiently long time of normal processing. Moreover, it would be difficult for the stateless partition enclaves to track and restore the partially processed data from the failed enclaves.

To summarize, the use of a leader improves performance by (1) reusing the report to different users to avoid the report generation latency; (2) allowing users to only attest once for all the enclaves.

4.2 Partition Enclaves

The partition enclaves of a model instance are organized as a DAG following the original model topology, and sequentially process the private user data batch in a pipelined manner. The model weights are preloaded and stay in the enclaves. While eliminating the data loading overheads for each user, enclave reuse has security implications. The enclave programs must be *stateless* and satisfy *non-interference*, so that processing different users' data in a parallel batch or sequentially through the same enclaves does not leak any sensitive information. Also, the data communication between enclaves should be *oblivious* to avoid timing side channels.

Fortunately, such requirements are naturally satisfied by most ML algorithms, including NNs. Typical inference tasks treat the model weights as read-only, and the private input data are only used to compute the output results without any side effects (stateless). Different data in a batch are also fully independent (non-interference). The amount of data to communicate across partitions are determined by the model hyperparameters, such as the number of channels and the feature map dimensions, which are all public constant values. The data streams flow between enclaves with deterministic sizes and at fixed time, regardless of the data values (oblivious). Nevertheless, we still need to ensure that the actual *implementation* does not have covert channels and backdoors. After computations, we carefully zero out all registers and memories before reuse or deallocation to prevent use-after-free attacks. All messages across enclaves are encrypted and authenticated, with unique sequence numbers for freshness.

4.3 Security Arguments

Establishing trust in multiple enclaves. We emphasize that the security guarantees of the multi-enclave Temper are equivalent to a single-enclave scenario provided by Intel SGX. We start by arguing that the user can establish trust with the leader enclave using the above design. The returned report follows the same format as in the standard SGX remote attestation in Figure 1 and can be verified by the IAS. The `USERDATA` field can be freely set without impacting security. The report convinces the user that the leader enclave is genuine and the program running inside it follows Algorithm 1. The leader program is designed to not leak secrets intentionally, and also to cleanse sensitive data after processing. The additional round of nonce verification in Figure 4 ensures that only the enclave that generates the report and knows $k_{s,priv}$ can pass the check, protecting against man-in-the-middle attacks.

To also trust the partition enclaves, the trusted and legitimate leader program attests all the partition enclaves on behalf of the

user. The list of partition enclave measurements is returned to the user, and can be verified against the open-source programs or a trusted public repository at the user side.

Side-channel attacks at runtime. Side-channel attacks are well known but hard to defend, stealing private information at the microarchitectural [16, 76, 93] (e.g., TLB, cache, DRAM, or page table) and physical (e.g., temperature [62], power [97], acoustical emanations [30], or electromagnetic signals [29]) levels. Comprehensive defense against side-channel vulnerabilities remains an open problem that is orthogonal to the Temper design. Nevertheless, we notice that typical NN layers processed in the partition enclaves are data oblivious, with no data access pattern leakage. For the leader enclave, the program in Algorithm 1 only involves encryption/decryption that has well-studied secure implementations. We also see from Section 4.2 that the communication between enclaves is oblivious, with deterministic sizes and at fixed time [42]. Therefore, Temper is free of these common side channels.

Impacts of malicious schedulers and managers. It is worth noting that in Temper, the global scheduler and the resource manager are untrusted. An attacker controlling those components may at most deny service to users, but never compromises sensitive data. Specifically, a corrupted scheduler may direct a user to a malicious model instance leader, but the attestation process is able to discover such hazards. The resource manager may kill any leader or partition enclave at any time, which would cause the entire model instance to be teared down, and the sensitive data inside the enclaves are cleared without leaking. When a new model instance is created, it will use newly generated keys. So replay attacks are not possible.

Minimizing TCB. Temper is designed to minimize the necessary TCB. Compared with other hardware TEE techniques such as ARM TrustZone [9], the TCB of Intel SGX does not include the OS. Furthermore, all cloud platform software, including the frontend, the scheduler, and the resource manager, does not need to be trusted. In Temper, other than the ML models themselves, the only additional trusted code is the leader program (Algorithm 1). We keep the leader to the minimal functionalities and exclude more complex tasks.

5 EFFICIENT MODEL PARTITIONING

To eliminate the large overheads of SGX secure paging without affecting the inference accuracy, Temper divides the many layers in an ML model into multiple smaller subsets that each fits in the EPC size limit of a single enclave. This section discusses the partitioning strategies in Temper to achieve high throughput and/or low latency.

5.1 Comparing with Prior Approaches

As introduced in Section 2.1, there have been a large body of model parallelism methods, including tensor partitioning [47, 77, 90] and graph partitioning [41, 67, 82, 83]. For our scenario, tensor partitioning is less suitable because it may introduce significant data duplication or communication across partitions. Therefore, we mostly focus on graph partitioning.

However, previous graph partitioning algorithms are not directly applicable to Temper (a quantitative comparison is in Section 7.4). First, existing approaches either assume unlimited memory space (e.g., CPUs) or strict capacity constraints (e.g., GPUs and accelerators) for each partition. In contrast, with SGX, performance exhibits a drastic change once crossing the EPC limit. Nevertheless, we actually may allow sometimes exceeding the EPC limit and tolerating the slowdown, especially when there are too many partitions dominated by the communication cost. Therefore, the partition sizes should not be treated as constraints, but they affect the computation latencies, which are more difficult to model accurately and require specific estimation models. Second, existing methods mostly focus on fine-grained partitioning that aims to maximize parallelism and to improve pipeline efficiency. In Temper, however, we are concerned about the EPC size limit, and thus only require *sufficient* numbers of partitions rather than the maximum. This requires us to use different optimization goals of throughput and latency, which lead to different solution strategies. Third, communication across distributed enclaves involves data encryption/decryption, data copying into and out of enclaves, and data transfers across networks. These all have much higher cost than traditional settings where partitioning happens mostly across multiple GPUs on a single machine with high-bandwidth PCIe or NVLinks. Our partitioning strategy needs to focus more on communication.

5.2 Optimization Goals

In typical cloud scenarios like MLaaS, the service providers would like to optimize the overall throughput of the system under a fixed budget of resources (e.g., number of servers), while ensuring that the user latencies are within a preset service-level objective (SLO) [88]. The partitioning strategies of Temper focus on providing high performance for each model instance. A large body of work on cluster management, including load balancing, auto-scaling, resource scheduling, etc. [12, 22, 37, 88], can be applied on top of Temper by the untrusted platform software.

We support two different optimization goals in Temper, *high-throughput* and *low-latency*. An ML model is partitioned to maximize the overall throughput under a fixed total number of servers using the high-throughput strategy, or to minimize the latency for a single user under the low-latency strategy. The two strategies for each ML model are solved offline and stored. At runtime, the cloud resource manager can auto-scale the numbers of instances under the two strategies based on the request load and the user requirements. For example, we can launch more high-throughput instances if the system load is high, and use low-latency instances when the load reduces. Temper does not support directly switching an instance between the two modes; the resource manager has to end the old instance and launch a new one. Such a coarse-grained, sub-minute-level switch is acceptable for MLaaS and comparable to typical cloud resource elasticity granularities.

More specifically, a partitioning strategy divides a model into n partitions, each with computation and communication latencies of $t_{\text{comp},i}$ and $t_{\text{comm},i}$, $i = 0, \ldots, n-1$. We then have

$$\min \left\{ n \times \max_i \{t_{\text{comp},i}, t_{\text{comm},i}\} \right\} \quad \leftarrow \text{high-throughput} \tag{1}$$

$$\min \left\{ \sum_i t_{\text{comp},i} + t_{\text{comm},i} \right\} \quad \leftarrow \text{low-latency} \tag{2}$$

Notice that under the high-throughput strategy, we assume fully pipelined processing across the n enclaves, and also between computation and communication of each enclave. So the overall throughput is determined by the slowest stage.

Model partitioning has no security impact. Different partitioning goals may result in different computation and communication latencies, but this does not exhibit any timing side channel; these latencies are still oblivious to input data. The model structure is public, so it is fine to expose the partitioning scheme.

5.3 Basic Units for Partitioning

When partitioning an ML model, a too coarse granularity may miss potential opportunities and fail to bring the memory usage below the EPC size, while overly fine-grained partitioning makes it slow to search for the best strategy. In NNs, it is natural to choose the *layer* as the basic granularity. Each partition contains one or more layers with their total memory usage no more than the EPC size.

Furthermore, we make some special adjustments as needed. We fuse small layers, such as pooling, batch normalization, and ReLU activation, with their neighbor layers [18, 84], to reduce the number of basic units and thus the partitioning complexity. We also split very large layers (e.g., a matrix multiplication with numerous weights) into multiple smaller sublayers that could be separately processed by different enclaves [52]. For complex NN topologies that contain branches, Temper marks each branch as an individual partition unit, i.e., each partition enclave can contain one or multiple branches. When calculating the performance, we slightly modify Equations (1) and (2) so that the latencies of these parallel branch units depend on the slowest one.

5.4 Latency Estimation Model

From Equations (1) and (2) we see that, in order to solve the best partitioning, it is crucial to obtain accurate estimation of the computation and communication latencies for each possible combination of layers that may run together in the same enclave. Such latencies are highly sensitive to the actual memory usage due to potential secure paging, and the input/output data volumes due to transfer, copy, and encryption/decryption. Traditional NN model partitioning approaches usually profile and measure each layer in isolation and assume the execution time stays unchanged [47, 67]. However, for in-enclave execution, the execution time of a layer highly depends on which other layers (and thus their data) are in the same enclave, and so cannot be measured independently.

For computation latencies, we make a key empirical observation that, if the total memory usage of a group of layers exceeds the EPC size, then *every* layer in this group will suffer from significant secure paging. In another word, the performance degradation happens for all the layers rather than a subset. This allows us to record only two latency profiles for each layer, i.e., with and without secure paging. First, we separately compile and measure the performance of each partitioning unit (Section 5.3) in two situations: alone (no secure paging), and with a simple memory ballooning module that eats up most of the EPC space (with secure paging). Second, we calculate the memory usage of each unit. NN layers have very regular program semantics. The memory layout is mainly composed of model weights, intermediate activation results, and inputs/outputs, where the model weights account for the largest share. Third, for any given combination of units, we sum up their memory usage, and compare with the enclave EPC size to determine whether secure paging happens for *all* or *none* of them. The corresponding latencies are then summed up as the total computation latency.

The communication latencies mainly involve decrypting and encrypting data before and after processing, copying data into and out from the enclaves, and transferring data over the network. All these overheads are roughly proportional to the corresponding input/output data volumes. So we use simple linear regression to calculate $t_{\text{comm}} = \alpha \times \text{data size}$. The parameter α is empirically measured, and is typically around 17 ms/MB on our system (Table 2).

5.5 Solving Optimized Partitioning

With the determined basic units in Section 5.3 and the latency estimation in Section 5.4, solving for the best partitioning strategy for a given model under the goals in Section 5.2 becomes straightforward. According to Equations (1) and (2), the key tradeoff is to balance between the computation and communication latencies. More partitions (a larger n) reduce secure paging overheads and thus the computation time, but also introduce more communication.

The complexity of these optimizations is modest. Typically, we partition ℓ (10s to 100s) units into n (1 to 10) enclaves. The number of possible strategies is $O(\ell^2)$. It is worth noting that evaluating each strategy does not involve any compilation or execution; we simply accumulate the pre-collected memory sizes and latencies of the basic units. Such a very fast procedure is enabled by our novel latency estimation approach. Therefore for simplicity, we use a simple exhaustive search.

6 IMPLEMENTATION

Temper is implemented in Rust [73], which is memory safe and avoids vulnerabilities like buffer overflow. We use the Fortanix Rust enclave development platform (EDP) [27] with Intel SGX SDK v2.9 [45] on Linux. We manually write the leader program as in Algorithm 1, with about 410 lines of Rust code. Such a small size of trusted code allows thorough inspection to eliminate potential bugs and vulnerabilities. We use TVM v0.7 [18] to compile high-performance NN kernels, leveraging both of its graph-level and operator-level optimizations. The Fortanix Rust EDP is able to integrate the TVM kernels into the partition enclaves. Communication between enclaves happens through sockets. Data are encrypted and authenticated with AES-256-GCM.

Model partitioning. We import the NNs using PyTorch, and implement the partitioning algorithms in Section 5 in Python. We output the partitioned models in the ONNX format [70], which are accepted as input to TVM. We use 32-bit floating-point precision. Both the basic unit memory and latency estimation and the final code generation of partitioned models follow this procedure.

Leader enclave. The leader enclave does not process the data, and occupies minimum EPC space. To reduce resource consumption and communication overheads, we co-locate the leader with the first partition on the same server, but they still use separate enclaves.

Multi-threading. We enable multi-threading in TVM to improve performance. On our eight-core processors, we can achieve almost linear scaling up to six threads. We suspect that when the

number of threads approaches or exceeds the number of physical cores, more frequent context switches would occur, which are particularly expensive for enclave execution due to swapping in/out sensitive contexts.

7 EVALUATION

We now evaluate the overall performance of Temper in terms of latency and throughput, as well as the effectiveness of its attestation protocol and partitioning strategies.

7.1 Experimental Setup

Platform. We evaluate Temper on four desktop machines, each with an Intel Core i7-9700 3.0 GHz CPU and 32 GB main memory running Ubuntu 18.04, and connected through 1 Gbps Ethernet. The SGX available EPC size to user applications is 93.5 MB. While 1 Gbps bandwidth is low, the data encryption/decryption and the data transfers between enclaves and untrusted memory also contribute to the communication cost besides the network bandwidth (Table 2). Also, this configuration biases the baselines which use no networking. When executing models requiring more than four machines, we fold them onto our testbed and run each group of enclaves individually one after another, and conservatively aggregate the results.

Workloads. We evaluate MobileNetV1 [39], ResNet18/50/152 [36], VGG19 [79], InceptionV3 [81], and DenseNet201 [40], all with the ImageNet dataset [23]. The default batch size is 1, and we also evaluate larger batch sizes of 4 and 16 in Figure 6 when optimizing Temper for high-throughput.

Baselines. We use three prior SGX-based MLaaS designs as our baselines, which adopt the similar threat model. TensorSCONE [53] uses a shim layer for OS support [10], which is more efficient than libOS approaches. Myelin [44] also uses TVM to generate efficient kernels, but does not apply any other optimizations to deal with the model loading overheads or the limited EPC size. With a similar principle of careful scheduling on model parameters, Lasagna [57] is the state-of-the-art. It uses a local task scheduler with three ring buffers to pipeline the stages of on-demand data loading, layer initialization, and layer processing. This design allows it to reduce the overall memory footprint and also (partially) hide the data loading cost, thus effectively alleviating the secure paging cost. Due to lack of open source and for fair comparisons, we reimplement the Lasagna techniques in our system, so it can also benefit from the high-performance TVM-generated kernels. We also include untrusted inference with TVM as the ideal upper bound. This untrusted baseline does not partition the NN models. The full model runs completely in the untrusted environment of a single machine, because there is sufficient memory.

7.2 Overall Performance

We first compare the latencies across all the systems at batch size 1, when optimizing for low-latency in Temper (Equation (2)). Figure 5 illustrates the results. We exclude the attestation latencies here, and evaluate them in Section 7.3. TensorSCONE (S) not only increases the TCB, but also causes substantial performance degradation of over 10×. For Myelin (M), Lasagna (L), and Temper (T), we break

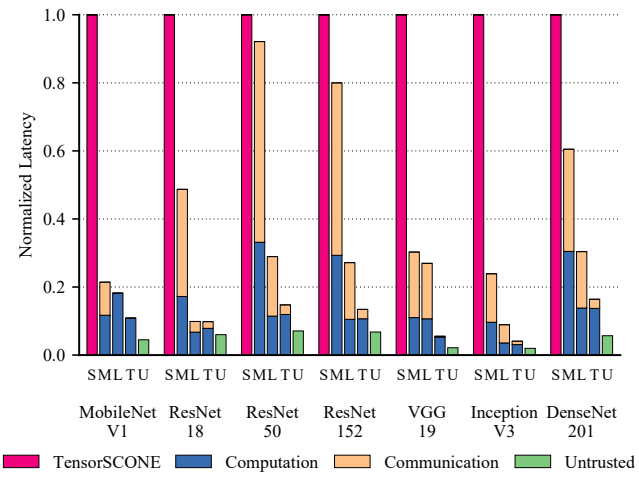

Figure 5: Comparison when optimizing for low-latency.

down the latencies into computation and communication. Communication includes loading and decrypting model weights in Myelin and Lasagna, and data transfers between enclaves through networking in Temper. Note that the communication cost of Lasagna is partially hidden by its pipeline, and the communication parts in the figure only show the portions that cannot be fully hidden. For the computation parts, Myelin suffers from serious secure paging for large models. Lasagna can effectively reduce this cost and result in lower latencies than Myelin in general, but introduces extra overheads with its shared memory buffers for MobileNetV1, which is a small model designed for mobile platforms and entirely fits within one enclave. However, both Myelin and Lasagna suffer from large data loading costs. The on-demand data loading in Lasagna cannot be hidden perfectly, since the pipeline stages are not balanced and weight loading can sometimes be the dominant bottleneck. In contrast, Temper can alleviate the secure paging issues and get similar computation latencies to Lasagna, while avoiding repetitive data loading for each user through enclave reuse. The network transfers between enclaves have only small overheads, typically on the order of tens of milliseconds, and can be overlapped with computations. Moreover, in large NNs, many layers would have much more weight data (repetitively loaded by Lasagna) than intermediate activation results (transferred across network in Temper), so the communication cost is smaller in Temper. On average, Temper achieves 4.9× and 2.2× latency reduction against Myelin and Lasagna across all the evaluated ML models. Even compared to the untrusted upper bounds U, Temper only introduces an average 2.1× slowdown, which is a reasonable price for privacy.

Next, we evaluate Temper in terms of throughput. Figure 6 compares Temper against Lasagna under the high-throughput goal (Equation (1)). We omit the other baselines as they are constantly inferior to Lasagna. Although Temper requires multiple enclaves on separate machines, partitioning ensures balanced pipeline stages with little resource idleness. Overall, Temper achieves 1.8×, 2.1×, and 1.2× higher throughput on average over Lasagna with batch size 1, 4, and 16. It is also interesting to see that, different from

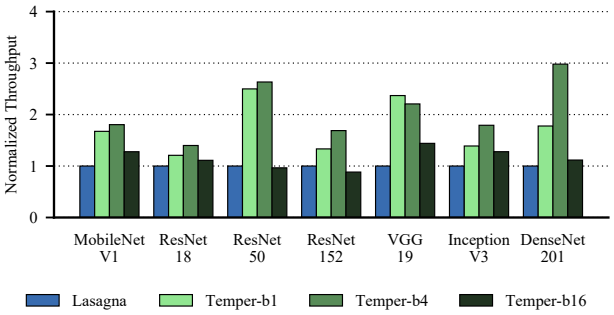

Figure 6: Comparison when optimizing for high throughput. Temper uses batch sizes 1, 4, and 16.

Table 1: Absolute performance summary of Temper.

Model	High-Throughput		Low-Latency
	Throughput (img/sec/server)	Latency (msec)	Latency (msec)
MobileNetV1	41.57	24.06	24.06
ResNet18	23.96	55.20	50.17
ResNet50	9.42	177.85	135.25
ResNet152	1.89	394.35	349.27
VGG19	0.60	1246.28	815.51
InceptionV3	4.39	203.95	144.82
DenseNet201	4.62	218.32	207.75

native execution, increasing batch sizes in Temper will not always result in throughput improvements. Larger data batches consume more EPC space, either increasing secure paging in each enclave, or being partitioned across more machines with higher communication cost and less balanced pipelines. Lasagna also suffers from such inevitable secure paging when increasing the batch size.

Table 1 summarizes the absolute performance of Temper under the two modes. Although the high-throughput strategy tends to use deeper pipelines, the latencies are still reasonable, within 2× of the low-latency results.

We also compare Temper with LoLa [15], a recent cryptographic approach for MLaaS that uses homomorphic encryption. LoLa designs a customized yet very small model for the MNIST dataset, which trades off accuracy for performance. We port this model into Temper, and see a 1607× speedup. Note that the main bottlenecks of LoLa are the sizes of weight matrices and data vectors. Therefore, its performance will be even worse if we evaluate it using complex NNs like those in Temper.

7.3 Attestation and Communication Cost

In Temper, the attestation with the leader enclave reuses the pre-generated report for all users to reduce the initial trust setup latency. Table 2 compares this customized protocol with the standard SGX attestation process. We exclude the communication delay between the user and the IAS. On the server side, the report generation inside the enclave takes non-negligible time, close to half a second. Temper eliminates this cost, but has another round of public key exchange (Figure 4). This key exchange is lightweight and only costs 30 ms. On the other hand, the report verification by the user remains the same. Overall, Temper attestation is about 4× faster.

Table 2: Attestation and communication delays in Temper.

Attestation (msec)	Server	User	Total
Standard	462.43	111.25	573.68
Temper	30.48	112.97	143.45
Communication (ms/MB)	Encryption	Transfer	Decryption
	2.40	12.82	2.19

Table 3: Comparison of Temper and DNN-Partition, both under high-throughput partitioning.

Model	Temper (img/sec/server)	DNN-Partition [83] (img/sec/server)
MobileNetV1	41.57	41.53
ResNet18	23.96	15.37
ResNet50	9.41	2.47
ResNet152	1.89	2.67
VGG19	0.60	0.39
InceptionV3	4.39	1.06
DenseNet201	4.62	2.74

One potential overhead in Temper is the data transfer latencies between partition enclaves, including encrypting and decrypting data as well as copying into and out of the enclave and transferring data over the network. Table 2 shows each MB of data needs about 17 ms to transfer on our 1 Gbps network, where the actual network transfer takes about 13 ms. With typical intermediate data across partitions of a few MB, the communication latencies are much lower than those for computation, which are usually several hundreds of milliseconds. High-speed networks at 10 to 100 Gbps may further bring the cost down. But note that the data copies into and out of enclaves and the encrypting/decrypting cost depend on the CPUs instead of the network.

7.4 Partitioning Strategies

We compare our partitioning strategy against a recent NN model parallelism approach, DNN-Partition [83]. The same pre-processing like operator fusion and splitting in Section 5.3 is applied to both. Table 3 shows the results. Temper substantially outperforms DNN-Partition in large NN models like InceptionV3 and DenseNet201. The main reason is that DNN-Partition assumes relatively cheap communication across heterogeneous devices in the same machine, aims to optimize for maximum parallelism, and enforces more strict partition size constraints. Hence, it has too many partitions for large models, resulting in higher data communication overheads.

When searching for the best partitioning strategies, the dominant cost is to build the latency estimation model using 2ℓ times of compilation and execution with and without secure paging, for each of the ℓ basic units in the model (Section 5.4). Each compilation and execution usually need tens of seconds. Table 4 shows the overheads for several representative models, where the total

Table 4: Compilation and execution time to build the latency estimation model for partitioning.

Model	MobileNetV1	ResNet50	ResNet152
Time (sec)	1363.4	1843.1	5269.5

overheads are measured to be within half an hour for MobileNetV1 and ResNet50, and up to 1.5 hours for ResNet152. The actual solving process (Section 5.5) only takes several seconds, since it directly uses the estimated latency values without additional measurements. Nevertheless, both of them are just offline, one-time overheads.

8 RELATED WORK

In this section we present related work about secure ML inference with cloud computing. Note that there also exist other proposals that deploy ML models on the user side while protecting the intellectual property (IP) of the service provider [58, 80]. This approach ships the performance problem to the users, who may not have enough resources or their computing resources may not be as cost-efficient as cloud computing. We believe the two approaches are orthogonal and each could be preferred in different cases depending on the requirements and constraints.

Cryptographic approaches. Modern cryptographic theories have enabled privacy-preserving computations. Fully homomorphic encryption (FHE) [31] allows us to do operations like addition and multiplication on ciphertext equivalently to those on plaintext. Multi-party computation (MPC) [94] enables mutually-distrusted parties to complete a joint computation while protecting the privacy of each other's input. NN models can be seen as a collection of linear and non-linear functions. CryptoNets [24] was the first to use FHE to build a nine-layer NN. However, FHE has several fundamental issues limiting the capability and the performance. First, it usually works with integers but not floating-point numbers. Second, ciphertext multiplication sharply increases the noise and limits the model depth. Third, non-linear activation functions are hard to realize. LoLa [15], as the successor of CryptoNets, solved part of the problems above and achieved higher accuracy and efficiency. It was even faster than HCNN [11], a GPU-accelerated FHE solution. Nevertheless, Temper achieves over 1600× speedup over LoLa. Other proposals like MiniONN [59] and Chameleon [72] used MPC for ML inference. Gazelle [49] and DELPHI [64] combined FHE and MPC. GForce [68] designed special ciphertext transformations to compute non-linear layers without approximation, and accelerated linear layers by offline computations.

TEE-based approaches. Hardware-based TEEs, such as Intel SGX [20] and ARM TrustZone [9], provide strongly isolated environments and guarantee the confidentiality and/or integrity of the code and data inside. Many designs like SCONE [10], Occlum [75], Graphene-SGX [87] integrated libraries into Intel SGX to ease application porting and development. This method can be applied to MLaaS, but often comes with substantial performance degradation, e.g., TensorSCONE [53], S3ML [61], PPML [55]. So it is desired to directly develop NN kernels in SGX for better efficiency. Privado [32] realized input-oblivous ML inference with SGX. Chiron [43] enabled distributed ML training. MLCapsule [34] instead ran inference on the user side and offered privacy guarantees to the service provider. Myelin [44] used TVM to reduce the runtime memory usage and the computation latency. Vessels [52] and Lasagna [57] aimed to address secure paging under the limited EPC size, by reusing a shared memory pool for the weights across different parts of the model, or smartly pipelining data loading with layer processing and hiding the overheads of the former. Instead, Temper alleviates secure paging by partitioning a model into multiple enclaves, and is demonstrated to be more efficient.

There are also TEE proposals on GPUs [42, 46, 89] and FPGAs [96], but not yet commercially available. Slalom [85] and DarKnight [35] used encrypted and verifiable linear computation outsourcing from CPU TEEs to untrusted GPUs, which significantly reduced the computation cost but at the expense of high communication overheads.

9 CONCLUSIONS

The privacy and security concerns now become a practical obstacle for MLaaS systems. Although existing TEE-based approaches offer better performance than cryptographic algorithms, they still sacrifice substantial performance for security due to the initialization overhead and the limited memory resources. We propose Temper, a secure MLaaS framework that optimizes performance and keeps security and accuracy guarantees. Temper integrates hardware TEEs like Intel SGX and high-performance ML software like TVM, and leverages novel techniques including enclave reuse and model partitioning. Temper outperforms the state-of-the-art baseline by over 2× in terms of latency and throughput.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their valuable suggestions, and the Tsinghua IDEAL group members for constructive discussion. This work was supported by the National Natural Science Foundation of China (62072262) and Shanghai Artificial Intelligence Lab. Mingyu Gao is the corresponding author.

REFERENCES

[1] [n. d.]. Alibaba Cloud. https://ai.aliyun.com/. Accessed: August 2021.
[2] [n. d.]. AMD Secure Encrypted Virtualization (SEV). https://developer.amd.com/sev/. Accessed: November 2021.
[3] [n. d.]. Baidu AI cloud. https://intl.cloud.baidu.com/. Accessed: August 2021.
[4] [n. d.]. Deep Learning on AWS. https://aws.amazon.com/deep-learning/. Accessed: August 2021.
[5] [n. d.]. Deep Learning VM, Google Cloud. https://cloud.google.com/deep-learning-vm/. Accessed: August 2021.
[6] [n. d.]. Intel® Trust Domain Extensions (Intel® TDX). https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html. Accessed: November 2021.
[7] [n. d.]. Machine Learning Service, Microsoft Azure. https://azure.microsoft.com/en-us/services/machine-learning/. Accessed: August 2021.
[8] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 265–283.
[9] ARM. 2009. ARM Security Technology Building a Secure System using TrustZone Technology. https://developer.arm.com/documentation/genc009492/c.
[10] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *12th*

USENIX Symposium on Operating Systems Design and Implementation (OSDI). 689–703.

[11] Ahmad Al Badawi, Jin Chao, Jie Lin, Chan Fook Mun, Sim Jun Jie, Benjamin Hong Meng Tan, Xiao Nan, Khin Mi Mi Aung, and Vijay Ramaseshan Chandrasekhar. 2018. Towards the AlexNet Moment for Homomorphic Encryption: HCNN, the First Homomorphic CNN on Encrypted Data with GPUs. *arXiv preprint arXiv:1811.00778* (2018).

[12] Ricardo Bianchini, Marcus Fontoura, Eli Cortez, Anand Bonde, Alexandre Muzio, Ana-Maria Constantin, Thomas Moscibroda, Gabriel Magalhaes, Girish Bablani, and Mark Russinovich. 2020. Toward ML-Centric Cloud Platforms. *Commun. ACM* 63, 2 (2020), 50–59.

[13] Peva Blanchard, El Mahdi El Mhamdi, Rachid Guerraoui, and Julien Stainer. 2017. Machine Learning with Adversaries: Byzantine Tolerant Gradient Descent. In *Advances in Neural Information Processing Systems (NeurIPS)*. 118–128.

[14] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. *arXiv preprint arXiv:2005.14165* (2020).

[15] Alon Brutzkus, Ran Gilad-Bachrach, and Oren Elisha. 2019. Low Latency Privacy Preserving Inference. In *36th International Conference on Machine Learning (ICML)*. 812–821.

[16] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. 2019. SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. 142–157.

[17] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *arXiv preprint arXiv:1512.01274* (2015).

[18] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie 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 *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 578–594.

[19] Edward Chou, Florian Tramer, and Giancarlo Pellegrino. 2020. SentiNet: Detecting Localized Universal Attacks Against Deep Learning Systems. In *2020 IEEE Security and Privacy Workshops (SPW)*. IEEE, 48–54.

[20] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016, 086 (2016), 1–118.

[21] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *25th USENIX Security Symposium (USENIX Security)*. 857–874.

[22] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 127–144.

[23] Jia Deng, Wei Dong, Richard Socher, Li Jia Li, and Fei Fei Li. 2009. ImageNet: A Large-Scale Hierarchical Image Database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 248–255.

[24] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. 2016. CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy. In *33rd International Conference on Machine Learning (ICML)*. 201–210.

[25] Minghong Fang, Xiaoyu Cao, Jinyuan Jia, and Neil Gong. 2020. Local Model Poisoning Attacks to Byzantine-Robust Federated Learning. In *29th USENIX Security Symposium (USENIX Security)*. 1605–1622.

[26] 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)*. 275–294.

[27] Fortanix. [n. d.]. Fortanix Enclave Development Platform - Rust EDP. https://edp.fortanix.com/. Accessed: January 2021.

[28] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. 2015. Model Inversion Attacks That Exploit Confidence Information and Basic Countermeasures. In *22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1322–1333.

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

[30] Daniel Genkin, Adi Shamir, and Eran Tromer. 2017. Acoustic Cryptanalysis. *Journal of Cryptology* 30, 2 (2017), 392–443.

[31] Craig Gentry. 2009. Fully Homomorphic Encryption Using Ideal Lattices. In *41st Annual ACM Symposium on Theory of Computing (STOC)*. 169–178.

[32] Karan Grover, Shruti Tople, Shweta Shinde, Ranjita Bhagwan, and Ramachandran Ramjee. 2019. PRIVADO: Practical and Secure DNN Inference with Enclaves. *arXiv preprint arXiv:1810.00602* (2019).

[33] Nils Gruschka and Meiko Jensen. 2010. Attack Surfaces: A Taxonomy for Attacks on Cloud Services. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. 276–279.

[34] Lucjan Hanzlik, Yang Zhang, Kathrin Grosse, Ahmed Salem, Max Augustin, Michael Backes, and Mario Fritz. 2018. MLCapsule: Guarded Offline Deployment of Machine Learning as a Service. *arXiv preprint arXiv:1808.00590* (2018).

[35] Hanieh Hashemi, Yongqin Wang, and Murali Annavaram. 2021. DarKnight: An Accelerated Framework for Privacy and Integrity Preserving Deep Learning Using Trusted Hardware. In *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 212–224.

[36] 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.

[37] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 295–308.

[38] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. 2013. Using Innovative Instructions to Create Trustworthy Software Solutions. In *2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*.

[39] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv preprint arXiv:1704.04861* (2017).

[40] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely Connected Convolutional Networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 4700–4708.

[41] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism.. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[42] Tyler Hunt, Zhipeng Jia, Vance Miller, Ariel Szekely, Yige Hu, Christopher J Rossbach, and Emmett Witchel. 2020. Telekine: Secure Computing with Cloud GPUs. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 817–833.

[43] Tyler Hunt, Congzheng Song, Reza Shokri, Vitaly Shmatikov, and Emmett Witchel. 2018. Chiron: Privacy-preserving Machine Learning as a Service. *arXiv preprint arXiv:1803.05961* (2018).

[44] Nick Hynes, Raymond Cheng, and Dawn Song. 2018. Efficient Deep Learning on Multi-Source Private Data. *arXiv preprint arXiv:1807.06689* (2018).

[45] Intel. 2018. Intel Software Guard Extensions (Intel SGX) Developer Guide. https://software.intel.com/content/www/us/en/develop/download/intel-software-guard-extensions-intel-sgx-developer-guide.html.

[46] Insu Jang, Adrian Tang, Taehoon Kim, Simha Sethumadhavan, and Jaehyuk Huh. 2019. Heterogeneous Isolated Execution for Commodity GPUs. In *24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 455–468.

[47] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks. In *Proceedings of the 2nd Conference on Systems and Machine Learning (MLSys)*.

[48] Mika Juuti, Sebastian Szyller, Samuel Marchal, and N. Asokan. 2019. PRADA: Protecting against DNN Model Stealing Attacks. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. 512–527.

[49] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. 2018. GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In *27th USENIX Security Symposium (USENIX Security)*. 1651–1669.

[50] David Kaplan, Jeremy Powell, and Tom Woller. 2016. AMD Memory Encryption. *White paper* (2016), 13.

[51] David Kaplan, Jeremy Powell, and Tom Woller. 2020. *AMD SEV-SNP: Strengthening VM Isolationwith Integrity Protection and More.* Technical Report. AMD Inc.

[52] Kyungtae Kim, Chung Hwan Kim, Junghwan "John" Rhee, Xiao Yu, Haifeng Chen, Dave Tian, and Byoungyoung Lee. 2020. Vessels: Efficient and Scalable Deep Learning Prediction on Trusted Processors. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. 462–476.

[53] Roland Kunkel, Do Le Quoc, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. 2019. TensorSCONE: A Secure TensorFlow Framework using Intel SGX. *arXiv preprint arXiv:1902.04413* (2019).

[54] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep Learning. *Nature* 521, 7553 (2015), 436–444.

[55] Dayeol Lee, Dmitrii Kuvaiskii, Anjo Vahldiek-Oberwagner, and Mona Vij. 2020. Privacy-Preserving Machine Learning in Untrusted Clouds Made Simple. *arXiv preprint arXiv:2009.04390* (2020).

[56] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. 2019. Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization. In *28th USENIX Security Symposium (USENIX Security)*. 1257–1272.

[57] Yuepeng Li, Deze Zeng, Lin Gu, Quan Chen, Song Guo, Albert Zomaya, and Minyi Guo. 2021. Lasagna: Accelerating Secure Deep Learning Inference in SGX-enabled Edge Cloud. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. 533–545.

[58] Ning Lin, Xiaoming Chen, Hang Lu, and Xiaowei Li. 2021. Chaotic Weights: A Novel Approach to Protect Intellectual Property of Deep Neural Networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 40, 7 (2021), 1327–1339. https://doi.org/10.1109/TCAD.2020.3018403

[59] Jian Liu, Mika Juuti, Yao Lu, and Nadarajah Asokan. 2017. Oblivious Neural Network Predictions via MiniONN Transformations. In *2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 619–631.

[60] Kang Liu, Brendan Dolan-Gavitt, and Siddharth Garg. 2018. Fine-Pruning: Defending Against Backdooring Attacks on Deep Neural Networks. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 273–294.

[61] Junming Ma, Chaofan Yu, Aihui Zhou, Bingzhe Wu, Xibin Wu, Xingyu Chen, Xiangqun Chen, Lei Wang, and Donggang Cao. 2020. S3ML: A Secure Serving System for Machine Learning Inference. *arXiv preprint arXiv:2010.06212* (October 2020).

[62] Ramya Jayaram Masti, Devendra Rai, Aanjhan Ranganathan, Christian Müller, Lothar Thiele, and Srdjan Capkun. 2015. Thermal Covert Channels on Multi-Core Platforms. In *24th USENIX Security Symposium (USENIX Security)*. 865–880.

[63] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*.

[64] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. 2020. Delphi: A Cryptographic Inference Service for Neural Networks. In *29th USENIX Security Symposium (USENIX Security)*. 2505–2522.

[65] Mathias Morbitzer, Manuel Huber, and Julian Horsch. 2019. Extracting Secrets from Encrypted Virtual Machines. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy (CODASPY)*. 221–230.

[66] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. 2018. SEVered: Subverting AMD's Virtual Machine Encryption. In *Proceedings of the 11th European Workshop on Systems Security (EuroSec)*. 1–6.

[67] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*. 1–15.

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

[69] Lucien KL Ng, Sherman SM Chow, Anna PY Woo, Donald PH Wong, and Yongjun Zhao. 2021. Goten: GPU-Outsourcing Trusted Execution of Neural Network Training. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, Vol. 35. 14876–14883.

[70] ONNX team. [n. d.]. Open Neural Network Exchange (ONNX). https://onnx.ai/. Accessed: January 2021.

[71] 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 (NeurIPS)*. 8026–8037.

[72] 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 *2018 ACM Asia Conference on Computer and Communications Security (AsiaCCS)*. 707–721.

[73] Rust team. [n. d.]. Rust Programming Language. https://www.rust-lang.org/. Accessed: January 2021.

[74] Shiqi Shen, Shruti Tople, and Prateek Saxena. 2016. Auror: Defending Against Poisoning Attacks in Collaborative Deep Learning Systems. In *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC)*. 508–519.

[75] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. 2020. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In *25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 955–970.

[76] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. 2016. Preventing Page Faults from Telling Your Secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (AsiaCCS)*. 317–328.

[77] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *arXiv preprint arXiv:1909.08053* (2019).

[78] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. 2017. Membership Inference Attacks Against Machine Learning Models. In *2017 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 3–18.

[79] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv preprint arXiv:1409.1556* (2014).

[80] Ziwei Song, Wei Jiang, Jinyu Zhan, Xiangyu Wen, and Chen Bian. 2021. Critical-Weight Based Locking Scheme for DNN IP Protection in Edge Computing: Work-in-Progress. In *Proceedings of the 2021 International Conference on Hardware/Software Codesign and System Synthesis (CODES/ISSS)*. 33–34.

[81] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the Inception Architecture for Computer Vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2818–2826.

[82] Masahiro Tanaka, Kenjiro Taura, Toshihiro Hanawa, and Kentaro Torisawa. 2021. Automatic Graph Partitioning for Very Large-scale Deep Learning. *arXiv preprint arXiv:2103.16063* (2021).

[83] Jakub M Tarnawski, Amar Phanishayee, Nikhil Devanur, Divya Mahajan, and Fanny Nina Paravecino. 2020. Efficient Algorithms for Device Placement of DNN Graph Operators. In *Advances in Neural Information Processing Systems (NeurIPS)*. 15451–15463.

[84] TensorFlow. 2020. XLA: Optimizing Compiler for Machine Learning. https://www.tensorflow.org/xla.

[85] Florian Tramer and Dan Boneh. 2019. Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware. In *7th International Conference on Learning Representations (ICLR)*.

[86] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2016. Stealing Machine Learning Models via Prediction APIs. In *25th USENIX Security Symposium (USENIX Security)*. 601–618.

[87] Chia-Che Tsai, Donald E Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *2017 USENIX Annual Technical Conference (ATC)*. 645–658.

[88] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-Scale Cluster Management at Google with Borg. In *10th European Conference on Computer Systems (EuroSys)*. Article 18.

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

[90] Minjie Wang, Chien-chin Huang, and Jinyang Li. 2019. Supporting Very Large Models using Automatic Dataflow Graph Partitioning. In *14th European Conference on Computer Systems (EuroSys)*. 1–17.

[91] Nico Weichbrodt, Pierre-Louis Aublin, and Rüdiger Kapitza. 2018. sgx-perf: A Performance Analysis Tool for Intel SGX Enclaves. In *19th International Middleware Conference (Middleware)*. 201–213.

[92] Ofir Weisse, Valeria Bertacco, and Todd Austin. 2017. Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves. In *44th Annual International Symposium on Computer Architecture (ISCA)*. 81–93.

[93] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *2015 IEEE Symposium on Security and Privacy (S&P)*. 640–656.

[94] Andrew C Yao. 1982. Protocols for Secure Computations. In *23rd Annual Symposium on Foundations of Computer Science (SFCS)*. 160–164.

[95] Chengliang Zhang, Junzhe Xia, Baichen Yang, Huancheng Puyang, Wei Wang, Ruichuan Chen, Istemi Ekin Akkus, Paarijaat Aditya, and Feng Yan. 2021. Citadel: Protecting Data Privacy and Model Confidentiality for Collaborative Learning. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. 546–561.

[96] Mark Zhao, Mingyu Gao, and Christos Kozyrakis. 2022. ShEF: Shielded Enclaves for Cloud FPGAs. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 1070–1085.

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

A APPENDIX

A.1 Abstract

We provide the source code of TEMPER in this artifact, including the real runtime of partitioned models (Section 4) and the partition and search algorithms (Section 5).

A.2 Artifact check-list (meta-information)

(1) Runtime environment: Linux Debian 5.4.0-54-generic; rust toolchain: nightly-2021-04-15-x86_64-unknown-linux-gnu, Python 3.6.9
(2) Hardware: Intel Core i7-9700 3.0 GHz CPU, 32 GB main memory, and 1 Gbps Ethernet
(3) Experiments: Described in Section 7
(4) Workflow frameworks used: Pytorch, ONNX, Fortanix, TVM
(5) How much time is needed to prepare workflow: The model partition is time-consuming due to the execution of the basic units of the networks, which may take several hours. Therefore, we provide some of the discovered best partition strategies under our settings. The compilation of the enclaves will take some minutes.
(6) How much time is needed to complete experiments (approximately)?: The execution time is at the level of seconds.

A.3 Description

A.3.1 How to access. The source code for the end-to-end TEMPER workflow, as well as the benchmarks, is archived at https://github.com/tsinghua-ideal/TEMPER-Secure-MLaaS.

A.3.2 Hardware dependencies. This artifact depends on the Intel CPU platform with SGX. We need SGX v1 which has limited EPC. Refer to https://github.com/intel/linux-sgx for more information.

A.3.3 Software dependencies. This artifact depends on the following software libraries:

- Intel SGX SDK 2.9
- Python 3 and the following packages are required:
  - torch
  - ONNX
- TVM 0.7
- Fortanix
- Clang 3.8 or older for building rust-mbedtls.

A.4 Installation

Follow the instructions below to install and set up the artifact:

(1) Install the required Python packages listed in the file of `requirements.txt`. Use the following command:

```
pip3 install -r requirements.txt
```

If some packages cause errors, use this command instead:

```
while read requirement;
    do sudo pip3 install $requirement;
done < requirements.txt
```

Note that the TVM packages should be installed separately as compiled packages in the next step.

(2) Install TVM v0.7 from the GitHub repository: https://github.com/grief8/tvm.git. You can refer to the TVM documentation at https://tvm.apache.org/docs/install/index.html for installation instructions. You can also refer to the following commands:

```
git clone --recursive \
    https://github.com/grief8/tvm.git tvm
sudo apt-get update
sudo apt-get install -y \
    python3 python3-dev \
    python3-setuptools gcc \
    libtinfo-dev zlib1g-dev \
    build-essential cmake \
    libedit-dev libxml2-dev

mkdir build
cp cmake/config.cmake build
cd build
cmake ..
make -j4

cd ../python
python setup.py install --user
cd ..
```

After compilation, install the required Python packages. We prepare the script of `install_tvm.sh` for it.

(3) Prepare the Rust environment by executing the following commands:

```
sudo apt install -y build-essential
curl --proto '=https' \
    --tlsv1.2 https: //sh.rustup.rs \
    -sSf | sh
```

This will download a script and start installing the `rustup` tool, which installs the latest stable version of Rust. If the installation succeeds, you will see a message: "Rust is installed now. Great!"
Then switch the `rustup` toolchain to `nightly` by installing the `nightly` version:

```
rustup install nightly
```

Finally, switch to the `nightly` version of `cargo`. We recommend the following one:

```
rustup default \
   nightly-2021-04-15-x86_64-unknown-linux-gnu
```

(4) Install Fortanix by following the official documentation at https://edp.fortanix.com/docs/installation/guide/. Note that Intel SGX SDK is required for this installation. You can also install it by `install_fortanix.sh`. Additionally, run the following command to get `llvm-ar` and `llvm-objcopy`:

```
rustup component add llvm-tools-preview
```

A.5 Experiment workflow

To evaluate the model partition and inference process, follow the instructions below:

A.5.1 Model Partition. To run model partition, execute the following command:

```
python auto_model_partition.py \
    --model <your_model> \
    --input_size <data_size> \
    --build_dir <generated_model_path>
```

Replace <your_model> with the model name, <data_size> with the desired input size, and <generated_model_path> with the directory where the enclave libraries will be stored. The model will be partitioned into several TVM submodels, and the submodels will be compiled into libraries and parameters which are compatible for SGX. The names include the models in Table 1, including ResNet18, ResNet50, ResNet152, VGG19, DenseNet201 and InceptionV3. The data size is set to 3×224×224 by default.

A.5.2 Model Inference. In the directory of `cluster-inference/`, there are dependencies: `attest-client` as the user, `ra-sp` as the attestation agent, `sgx-task-enclave` as the template of **worker enclaves**, and `scheduler` as the **leader enclave**. A worker generator script `worker_generator.py` will be used to generate actual worker enclaves on demand.

First, configure the inference instances as below.

Attestation: Set the contents or paths of SPID, keys, and certificates under `ra-sp/examples/data/setting.json`. The details are in https://github.com/ndokmai/rust-sgx-remote-attestation. We have already set the default configurations.

Network: For easy testing, we set the default configurations available for a single machine, i.e., localhost. You can modify the configurations in each enclave project to run the instance across multiple machines. For worker enclaves, you will have to modify `worker_generator.py` and re-generate the instances to make the changes take effect.

Worker generation: Use `worker_generator.py` to generate worker enclaves with different partitioned models after the configuration. The following command imports the submodels from <generated_model_path> in A.5.1 and produces the worker enclave projects to <target_instance_dir>.

```
python worker_generator.py \
    <generated_model_path> \
    <target_instance_dir>
```

Now, we can perform model inference by executing the following commands:

```
# Setup environment
cd cluster-inference
source environment.sh

# Generate instances
python worker_generator.py \
    <the path of generated models> \
    <the path of target instance dir>

# Build and run
./clean.sh <the path of target instance dir>
./build.sh <the path of target instance dir>
./run.sh <the path of target instance dir>
```

If you test the benchmark on a single machine, just run `run.sh` to build and run all the modules above. The results will be shown on the terminal.

A.6 Evaluation and expected results

The client will output the total cost, and every worker enclave will also output the latency of the submodels. The results will be printed with text explanations on the terminal, and we can collect them to calculate the throughput and other metrics. For example, we can collect the latency and latency breakdown data of different models to get similar results of Figures 5 and 6, and Table 1.