Protecting Deep Learning Models on Cloud Platforms with Trusted Execution Environments

A THESIS

SUBMITTED FOR THE DEGREE OF

Doctor of Philosophy

IN THE

Faculty of Engineering

BY

Kripa Shanker

Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012 (INDIA)

October, 2025

Declaration of Originality

I, **Kripa Shanker**, with SR No. **04-04-00-13-12-17-1-14847** hereby declare that the material presented in the thesis titled

Protecting Deep Learning Models on Cloud Platforms with Trusted Execution Environments

represents original work carried out by me in the **Department of Computer Science and Automation** at the **Indian Institute of Science** during the years **2017 - 2025**.
With my signature, I certify that:

- I have not manipulated any of the data or results.
- I have not committed any plagiarism of intellectual property. I have clearly indicated and referenced the contributions of others.
- I have explicitly acknowledged all collaborative research and discussions.
- I have understood that any false claim will result in severe disciplinary action.
- I have understood that the work may be screened for any form of academic misconduct.

Date: Student Signature

In my capacity as supervisor of the above-mentioned work, I certify that the above statements are true to the best of my knowledge, and I have carried out due diligence to ensure the originality of the report.

Advisor Name: **Prof. Vinod Ganapathy** Advisor Signature

In loving memory of my grandmother, Smt. Maharanji Devi, my grandfather, Shri Bhagwati Prasad Tiwari, and my mother, Smt. Leela Devi Tiwari.

DEDICATED TO

My father Shri Uma Shankar Tiwari

Acknowledgements

This dissertation came into existence due to the enormous support from different people who helped during the different stages of the PhD journey in their respective capacities. I want to take this opportunity to thank them and express my gratitude for their support.

First and foremost, I am deeply indebted to Prof. Vinod Ganapathy for his constant support at great lengths and encouragement throughout this journey. I want to thank him for giving me the opportunity to work under his guidance and introducing me to the research. I am always amazed by his excitement to tackle challenges. I truly enjoyed our weekly discussions and scrum meetings, and felt enlightened after every interaction with him. I have learned a lot under his guidance and wish to apply those principles in the upcoming academic, professional, and personal life.

Next, I would like to thank Prof. Aditya Kanade for his guidance during the second part of this dissertation. His support and expertise helped to expand my research horizon. I also want to express my deepest gratitude to the faculty members of the Computer Science and Automation Department, Prof. Y. Narahari, Prof. Shirish K. Shevade, Prof. Deepak D'Souza, Prof. K. V. Raghavan, Prof. Uday Reddy Bondhugula, Prof. Siddharth Barman, Prof. Anand Louis and Prof. Arkaprava Basu, for teaching courses that laid a solid foundation for my research. I am also thankful to the admission committee for granting me the opportunity to study at this prestigious institution. I initially enrolled for the M.Tech (Research) program, and later upgraded to the PhD program. I want to thank the program conversion committee, Prof. K. Gopinath and Prof. Shalabh Bhatnagar, for upgrading my enrollment to the PhD program. I also thank my comprehensive exam committee members, Prof. Deepak D'Souza, Prof. Yogesh Simmhan, Prof. Arpita Patra and Arkaprava Basu for allowing me to continue this research journey.

Next, I would like to thank my collaborators, Arun and Vivek, for their being part of my research journey. I learned a lot while working them. I especially want to thank Arun for sharing his insights and expertise since the beginning of this research journey.

I spent a significant amount of my time at the Computer Science and Automation Depart-

ment, and it has become an integral part of this journey due to the wonderful people associated with this department. I want to express my deepest gratitude to the Computer Science and Automation department and its staff, Ms. Padmavathi, Ms. Meenakshi, Ms. Kushael, Ms. Nishitha, Akshay, Aravind and Sudesh Bhaiya for providing a homely environment away from home and for the administrative help. I want to especially thank Ms. Padmavathi for playing a key role during the admission process.

This long journey was memorable and joyful due the wonderful individuals whom I met and stayed on the campus. I am thankful to my amazing Computer Systems Security lab mates, Subhendu, Aditya, Rounak, Abhishek Vijeev, Ajay, Nikita, Rakesh, Chinmay, Nikhil, Gokul, Eikansh, Isha, Vivek, Himanshu, Dhruv, Suraj, Nishit, and Kanmani, for their camaraderie and insightful discussions. Beyond the CSSL lab, I had the company of amazing masters batch mates, Akash Mishra, Anirban Biswas, Anshuman, Gaurav, Komal, Meghashyam, Mayank Singh, Mayank Gupta, Praveen, Prateek, Rakshit, Rishabh, Rishi, Sachin, Vishal, Saloni, Samadhan, Santhosh, Shivank, Tapan, Venkat, Vinay, and Vishakha. I want to thank my seniors, Stanly, Ajinkya, Mayank Ratan Bhardwaj, Ullas, Vineet, and Shubham Gupta for their mentorship. Outside IISc, I want to thank my friends from undergraduate studies, Amar, Archna, Ashish, Harshit, Himanshu, Meenakshi, Pratik, Praveen, Shivangi, and Shubham Verma for setting the right environment to begin postgraduate studies. I am thankful to Ajay Pandey and Shubham Singh for being around since the beginning of my undergraduate studies. Outside the computer science department, I especially want to thank my friend, Mahesh Kumar Singh, for his constant support and guidance from the beginning to the very end of this journey. Having him along as a co-passenger made it easier to navigate this journey.

I also want to thank my previous employer, Banyan Intelligence and Prof. Himanshu Tyagi, for providing me the opportunity to work on blockchain. I also want to Senthilkumar Bala for being the thankless manager and enabling me to complete the doctoral work. I want to thank Vishal, Arun, Pankaj, Aswin, and Archna for being amazing colleagues and providing a vibrant work environment.

During this PhD journey, I lost my loving elder grandmother, Smt. Maharanji Devi, and my grandfather, Shri Bhagwati Prasad Tiwari, who could not see me graduating with a doctoral degree. I have the most beautiful and loving childhood memories of spending time during summer vacation. She was eagerly waiting for me to finish my studies. I will always miss her.

I want to thank my extended paternal and maternal family for their wishes and prayers. I especially want to thank Nanka Bua, paternal aunt, for being there during my undergraduate studies, and Chanchal Mausi, maternal aunt, for frequent calls to find out how I am doing. I am also grateful for having Priyanka didi, my cousin sister, and her calls.

Acknowledgements

I am deeply grateful to my family for being the rock-bottom support during this journey and for going through the hardships during this journey. I have the blessings and prayers of my grandmother, Smt. Gulabkali. I am blessed to have my younger brother, Satender, for always being with me throughout the journey. I am immensely thankful to my wife, Khushboo, for the patience, support, understanding, and sacrifices she made to enable me to complete this journey.

Most of all, I owe everything to my father, Shri Uma Shankar Tiwari. I am out of words to express my gratitude for all the love, support, and sacrifices he made to make this possible. I dedicate this thesis to him.

Abstract

Deep learning is rapidly integrated into different applications, from medical imaging to financial products. Organisations are spending enormous financial resources to train deep learning models. Often, many organisations do not have sufficient resources to host, manage and scale these deep learning workloads in-house. Therefore, these organisations outsource deep learning inference workloads to public cloud platforms. However, outsourcing to public cloud platforms raises security and privacy risks for the trained models. On the cloud, the service provider controls all the software and hardware on their premises and has full access to the models deployed on their platforms. A malicious or compromised cloud provider can steal the trained model or interfere with the inference workload, which may lead to financial losses and legal troubles for the model owner. This dissertation presents solutions to secure deep learning workloads on public cloud platforms with hardware-assisted trusted execution environments.

Intel has introduced Software Guard Extensions (SGX), a hardware-based trusted execution environment, to run private computations on public cloud platforms. However, applications do not run out-of-the-box on the SGX platform due to the restrictions imposed by the SGX specifications to ensure confidentiality and integrity of the code and data. Therefore, applications need to be rewritten, or other methods should be employed to avoid executing restricted instructions within the SGX enclave that contains code and private data. To port commodity applications to SGX enclaves, the software community has developed multiple frameworks to adapt existing applications to SGX specifications. However, at the beginning of this work, it was not clear which framework should be used to port deep learning workloads to SGX enclaves. Therefore, in the first part of this dissertation, we studied various frameworks that port applications to SGX to find a suitable framework for porting deep learning workloads. The study focuses on the challenges in transitioning commodity applications to SGX enclaves.

Next, during the study, we observed that memory-intensive applications, such as deep learning workloads, incur a performance penalty when executing within the trusted execution environment offered by Intel SGX. Furthermore, SGX cannot securely use other untrusted resources, such as untrusted co-processors, that are commonly used to accelerate deep learning workloads.

Therefore, the second part of the dissertation focuses on improving the performance of deep learning workloads on TEE. It presents MazeNet, a framework to transform pre-trained models into MazeNet models and deploy them on heterogeneous execution environments based on trusted and untrusted hardware, where the trusted hardware ensures the security of the model while the untrusted hardware accelerates the deep learning workload. MazeNet employs a secure outsourcing scheme that outsources both the linear and non-linear layers of deep learning models to untrusted hardware. Our experimental evaluation demonstrates that MazeNet can improve the throughput by 30x and reduce the latency by 5x.

Publications based on this Thesis

1. **An evaluation of methods to port legacy code to SGX enclaves**
   Kripa Shanker, Arun Joseph, and Vinod Ganapathy
   In Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020.

2. **MazeNet: Protecting DNN Models on Untrusted Cloud Platforms with Trusted Hardware**
   Kripa Shanker, Vivek Kumar, Aditya Kanade, and Vinod Ganapathy
   In Proceedings of the 21st International Conference on Information Systems Security, December 16-20, 2025, Indore, India.

Tools

1. **Porpoise**: A software framework to port commodity applications to Intel SGX enclaves based on the instruction wrapper model. Archived on 16 June 2020 on Zenodo, https://doi.org/10.5281/zenodo.3895761

Abbreviations

AES	Advanced Encryption Standard
AMD ES	AMD Encrypted State
AMD SEV	AMD Secure Encrypted Virtualization
AMD SEV-SNP	AMD SEV-Secure Nested Paging
API	Application Programming Interface
Arm CCA	Arm Confidential Compute Architecture
CNN	Convolutional Neural Network
CVM	Confidential Virtual Machine
DL	Deep Learning
DNN	Deep Neural Network
DRAM	Dynamic Random-Access Memory
EPC	Enclave Page Cache
HMAC	Hash-based Message Authentication Code
Intel TDX	Intel Trust Domain Extension
LKL	Linux Kernel Library
LLM	Large Language Model
Nvidia CC	Nvidia Confidential Compute
PRM	Processor Reserved Memory
RMM	Realm Management Monitor
SDK	Software Development Kit
SGX	Software Guard Extensions
SLOC	Source Line of Code
TCB	Trusted Computing Base
TD	Trust Domain
TEE	Trusted Execution Environment
TVM	Trusted Virtual Machine
VM	Virtual Machine

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the recent advances in deep learning, commodity applications are extended with deep learning techniques to offer enhanced and tailored services to the users to accomplish their everyday tasks with ease [45]. It is evident from the fact that two of the major AI frameworks, TensorFlow and PyTorch, are used in more than 1.2 million GitHub projects in 2025 [39, 48]. A majority of these applications are developed and owned by individuals and small organisations.

Deep learning workloads are computationally expensive and require dedicated hardware accelerators to scale to real-world deployments. Many small and medium-sized organisations do not have sufficient resources to run these models in-house. Therefore, they outsource deep learning inference services to public cloud providers to free themselves from maintaining and scaling the service with user demand. It is predicted that these deep learning deployments will form a significant fraction of revenues for public cloud providers in the coming years [99].

However, moving to the cloud exposes the outsourced application and data to various security and privacy risks, as the cloud vendor controls the full software and hardware resources on its premises. As a result, an adversarial or compromised cloud vendor can steal or tamper with the applications deployed on its platform. These security challenges make public cloud computing unattractive to organisations that deal with private and sensitive data, such as healthcare institutions and financial organisations.

To tackle this problem of private computation on public cloud platforms, processor vendors have introduced hardware support for Trusted Execution Environments (TEE) that protect applications from privileged adversaries. Notable examples include Intel Software Guard Extensions (Intel SGX) [32], AMD Secure Encrypted Virtualization (AMD-SEV) [9], and Arm TrustZone [8] and Confidential Compute Architecture (ARM CCA) [93].

1.1 Trusted Execution Environments

While encryption schemes can protect the data when it is stored on disk, data-at-rest, and during transfers over untrusted channels, data-at-transit, it is available in plaintext during processing, data-in-use. An adversary can steal the data when it resides in main memory during processing. The adversary can use software-based attacks by compromising the privileged software, such as the host operating system or the hypervisor, or it can perform physical attacks, such as probing memory and PCIe interfaces.

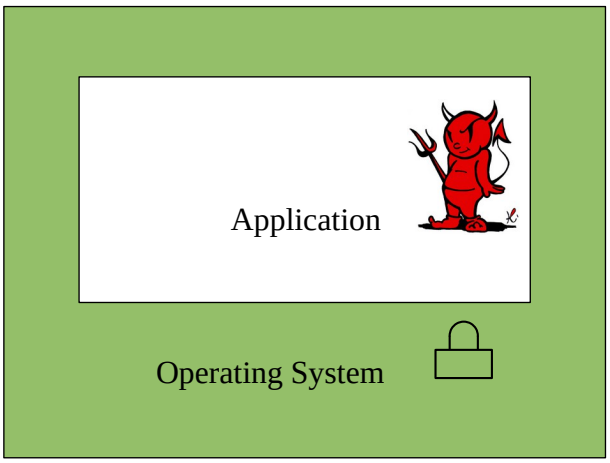
Trusted execution environments aim to protect the confidentiality and integrity of data-in-use from privileged adversaries on the cloud and edge devices. Modern computing platforms rely on hardware and software-based mechanisms such as page tables and CPU privilege levels to isolate and protect host system software and user applications. These mechanism offers protection and isolation between processes, users, kernel, user applications, and virtual machines. Traditional systems follow a hierarchical security model, where code executing at higher privilege levels has access to resources at the same or lower privilege levels. For example, a virtual machine manager executes at a higher privilege level than the guest kernel and user applications, thus it can access the resources that are allocated to them. However, those with lower privileges cannot access higher-privileged resources.

Trusted execution environments introduce a new security model where guest applications, running with lesser privileges, within an isolated and protected runtime, cannot be accessed even by the privileged software or hardware, irrespective of their privilege levels, if they are not part of the trusted computing base. However, the privileged host still controls the resource allocation, such as allocating CPU time and memory pages to the trusted execution environments.

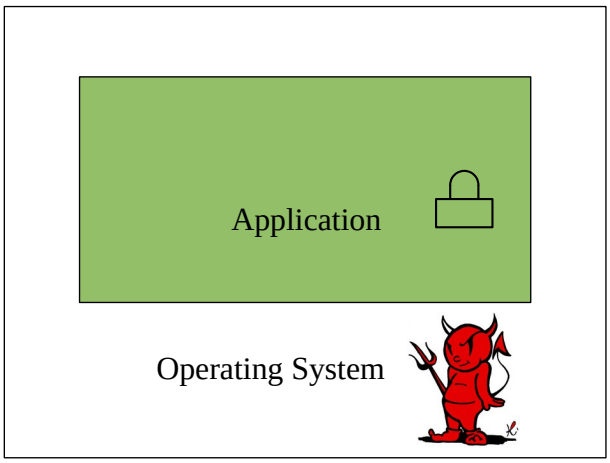
Trusted execution environments are implemented as a set of hardware and software extensions, where the responsibility of ensuring security is shared across both. The hardware extensions prevent unauthorised software access to the TEE memory, and the software extensions enable the creation and management of the trusted environment. Additionally, TEE platforms can have dedicated hardware to encrypt the contents of main memory to protect them from physical attacks [53, 72, 167, 52], e.g., memory bus snooping [142, 78].

1.1.1 Intel SGX

Intel SGX is one of the TEE implementations that enables user applications to build secure containers, referred to as enclaves, within their program address space. The code and data within these enclaves are protected from privileged adversaries, such as operating systems. Applications that wish to protect their sensitive data place their sensitive data within these



(a) Hierarchical security model aims to protect the privileged host from the malicious guest.



(b) Trusted execution environments, e.g., Intel SGX, aim to protect the guest from the host.

Figure 1.1: The Traditional or hierarchical security model aims to protect the privileged host from the guest applications. In contrast, TEEs want to protect guest applications from the host.

enclaves. An application can build one or more such enclaves to separate independent or mutually distrusting modules, where one enclave cannot access the memory of other enclaves.

In SGX, the enclaves and the host kernel are mutually distrusting, i.e., neither the enclave can access kernel memory nor the kernel can access the enclave memory. Both the enclave and the kernel can communicate through user space memory. To implement this isolation between kernel and enclaves, Intel SGX changes the memory access semantics, which are enforced through hardware to protect the kernel and enclave memory.

Within the enclaves, SGX restricts certain instructions that can compromise the confidentiality and integrity of the enclaves. Some of the restricted instructions, e.g. `syscall`, are frequently used by user applications to request the services of the operating system. Therefore, commodity applications do not run out-of-the-box on enclaves, and the application developers are responsible for ensuring that the enclave code does not execute any illegal instruction within the enclaves.

It is the responsibility of the application running within an enclave to ensure that it does not accidentally leak any sensitive data. Any data leaving the enclave must be encrypted or sanitized before leaving the enclave boundary. The application should use proper security mechanisms when communicating with the outside world, such as TLS/SSL for network communications and encrypted file systems for storage.

To protect the enclave memory, Intel SGX reserves a portion of the main memory, referred

to as Processor Reserved Memory (PRM), during the system boot to store encrypted pages of enclaves. These encrypted pages on the main memory are referred to as Enclave Page Cache (EPC), and they are encrypted by a dedicated cryptographic engine when stored on the main memory and decrypted on the fly when moved to the CPU cache. The size of the EPC is fixed and significantly smaller than the size of the main memory, e.g., 128 MB on SGXv1, due to the hardware structures required to enforce memory protection.

1.1.2 Intel TDX

After SGX, Intel introduced Trust Domain Extensions (TDX) [4] to protect the entire virtual machine instead of portions of the program address space. Intel TDX protects the guest virtual machines from the host hypervisor, BIOS, and firmware. The host hypervisor still controls the resource allocation, such as allocating memory to the guest virtual machines, but it cannot access the allocated memory. This protection is implemented through a combination of software and hardware support.

In Intel TDX, a trusted module runs within a protected region whose memory cannot be accessed from external sources, such as the host hypervisor and DMA devices. The trusted module manages the security policies and page tables of virtual machines, while the hardware enforces those policies. The TDX module is responsible for setting up and tearing down trusted virtual machines. The host hypervisor allocates a portion of main memory to the TDX module, which in turn allocates those pages among the set of trusted virtual machines. Similar to Intel SGX, Intel TDX employs main memory encryption to protect the guest virtual machines from physical attacks.

Apart from Intel, other processor vendors also support VM-based trusted execution environments. AMD offers Secure Encrypted Virtualization (SEV) [34], and Arm introduced Confidential Computing Architecture (CCA) [93] in Armv9 architecture. These solutions offer similar protections as compared to Intel TDX; however, the implementation differs across vendors, with security responsibility split across different software-hardware components.

In comparison to Intel SGX, Intel TDX, and other VM-based Trusted Execution Environments have a higher trusted computing base (TCB) as the guest kernel is part of the TCB. Therefore, VM-based TEEs are more vulnerable to attacks. In this dissertation, we have used Intel SGX for TEE as its threat model is more suitable for cloud workloads with the least amount of TCB, and it was the only commercial solution available at the beginning of this work.

1.2 Deep Neural Networks

Deep neural networks are a special category of applications that learn complex functions or input-output relationships directly from the training dataset and perform predictions on unseen data that was not part of the training dataset. A deep neural network consists of a set of layers that are connected to form a graph. Most layers contain parameters (or weights) that are learned during the training phase from the training dataset. During the training, the model learns an internal representation of the training dataset that is later used to make predictions on previously unseen data.

Once a model is trained, it can perform predictions on input samples, where the input passes through a sequence of layers to compute the prediction scores. Most of the layers consist of linear operations, such as matrix multiplications, and non-linear operations, such as ReLU or Sigmoid activation functions. As the linear operations form the bulk part of the model evaluation and are computationally expensive, specialised hardware, such as GPUs, TPUs and FPGAs, can be employed to speed up the linear and other operations to accelerate the model evaluation on the given input.

The accuracy of a deep learning model depends on multiple factors such as the size of the training dataset, the architecture or network connections between the layers and their types, and the hyperparameters. Training a well-performing model is an expensive process due to the effort required in collecting, cleaning, and labelling the training dataset, as well as the computational cost involved in training the model [114]. A well-trained model provides a competitive business advantage to model owners. Therefore, it becomes important to protect trained models.

Apart from trained model parameters, deep learning workloads involve other private and sensitive data that needs to be protected from adversaries. Often, the training dataset is sensitive, e.g., health care records [2], or private, e.g., financial data [1]. Many countries have stringent laws that require organisations to maintain the privacy and confidentiality of such records [2, 1]. In addition to the training dataset, model inputs during inference can be private or sensitive as well, and need to be protected during inference.

Deep learning models are vulnerable to different classes of attacks that were not present in earlier applications. These include membership inference attacks [137], model inversion attacks [40], model extraction attacks [109], and adversarial attacks [47]. These attacks aim to steal the trained models, training dataset, and exploit their behavior.

1.3 Contributions and Outline

This dissertation studies the challenges in protecting applications and deep learning models and workloads on public cloud platforms with Intel SGX, a hardware-based trusted execution environment, and presents systems to port applications and deep learning models to SGX enclaves. Further, it presents methods to speed up the deep learning inference workload with untrusted hardware while protecting the confidentiality of the deep learning models.

The first challenge in using Intel SGX to protect deep learning workloads is that deep learning frameworks, e.g., TensorFlow or PyTorch, do not run out-of-the-box on enclaves as SGX restricts certain instructions within the enclave. Therefore, the inference framework needs to be rewritten in an SGX-aware manner or use other methods to avoid execution on illegal instructions within the enclave.

To overcome this barrier, the software community has devised various methods to port existing applications to enclaves. However, at the beginning of this work, there was no clear answer to which framework an application developer should use to port their existing workload to enclaves. Therefore, in the first part of the dissertation, we study the challenges faced by application authors to determine which framework is most suitable for their needs.

The frameworks developed by the community can be broadly classified into three categories: the *library OS model*, the *library wrapper model*, and the *instruction wrapper model*. These models use different methods to handle restricted instructions present in application code.

The library OS model. In this model, an entire library OS executes within an enclave. To port an application to the enclave, the application developer loads the application binary along with its dependencies into an SGX-aware library OS, which is responsible for adhering to the specifications of SGX enclaves. Then, the application runs on top of the library OS, while the library OS handles illegal instructions in application code within the enclave either by delegating the execution to outside of the enclave or emulating the instruction within the enclave. Frameworks that implement the library OS model include Haven [16], Graphene-SGX [155], Occulum [134] and SGX-LKL [118].

The library OS model offers three main benefits: binary compatibility, fewer domain crossings, and a simple enclave interface. The library OS model supports executing unmodified binaries, along with support for simulating dynamic linking. The library OS emulates many services within the enclave; therefore, few domain crossings are required to request services from the host operating system. As the library OS handles many services within the enclave, the library OS and host OS interface is narrower. Thus, it is easier to protect a small set of interfaces.

The main drawback of library OS is a large TCB size, as the library OS executes within the enclave and is part of the TCB. Another drawback is low flexibility, as the entire binary is expected to run within an enclave. This becomes challenging for the developer who wishes to run non-sensitive portions of the application outside the enclave.

The library wrapper model. This model assumes that applications invoke operating system services via libraries such as the standard C library (`libc`). Normally, these libraries contain privileged instructions, which are prohibited in the enclaves. The library wrapper model provides wrappers for library functions that delegate the execution of prohibited instructions outside the enclave. This model is primarily used in Panoply [135].

The main benefit of the library wrapper model is a small TCB size, as the libraries execute outside the enclave. However, this small TCB comes at the cost of a large enclave-host interface. When the enclave requests a service from the host OS through one of these interfaces, it needs to implement checks to ensure that the untrusted host OS has correctly serviced the request. Having a large interface increases the burden of securing each interface.

Apart from the large interface, the library interface keeps changing from one release to another, as there are no standards for most library interfaces. ISO and POSIX have defined standards for function-call interfaces, but it has left many data structures unspecified.

The instruction wrapper model. In this model, wrappers are provided for the low-level instructions (such as `syscall`, `outb`, `outb`) that are prohibited within the enclave. The wrappers are responsible for outsourcing the execution of privileged instructions outside the enclave. The wrappers are responsible for encrypting the sensitive data that leaves the enclaves and decrypting the results received by the enclave. This model is implement by SCONE [13] and lxcsgx [149].

The main advantage of the instruction wrapper model is a slower-changing interface as compared to the library wrapper model. There is a defined contract between the hardware and the software, the application binary interface (ABI). Also, the interface is smaller compared to the library wrapper model, which makes it easier to secure the enclave-host interface.

The drawback arises from the somewhat larger TCB compared to the library wrapper model, as the dependent libraries execute within the enclave.

Unfortunately, at the time of the study, there was no publicly available implementation of the instruction wrapper model. Therefore, we built our own in-house instruction wrapper model, Porpoise, for the study.

1.3.1 Porpoise

First, we built Porpoise to port commodity applications with the instruction wrapper model. Porpoise is a thin intermediate layer that sits between the enclave application and the host kernel. It enables enclave applications to get services of the host operating system by invoking SGX restricted instructions outside the enclave on behalf of the enclave code. Porpoise consists of two components: a trusted shim library that executes within the TEE and an untrusted shim library that resides in the user space.

The trusted shim library in Porpoise implements wrappers around instructions that are prohibited within enclaves. Usually, the prohibited instructions are present in the standard C library (`libc`) and a few other low-level libraries. These libraries are modified to incorporate wrappers that delegate prohibited instructions to the untrusted shim layer outside the enclave.

The untrusted shim can execute instructions on behalf of enclave applications as it runs in user space. It invokes the required instructions and returns the controls to the trusted shim layer. The trusted shim copies the results to the enclave and resumes application execution.

Porpoise implements wrappers around the `syscall` instruction as it is frequently used by applications to request operating system services, such as reading/writing files, allocating/deallocating memory. The challenges in the implementation of Porpoise are presented in Chapter 3.4.

To port applications with Porpoise, a developer needs to recompile their application with Porpoise shim libraries along with the modified standard C library. Then the developer can deploy the newly compiled binary on an SGX platform. Thus, applications can be quickly ported to SGX enclaves without any modification to the application's source code.

1.3.2 An Evaluation of Methods to Port Legacy Code to SGX Enclaves

Once we had the candidate implementation for the instruction wrapper model, we resumed our in-depth study of various models to port applications to SGX enclaves. We evaluated the models on four dimensions: porting effort, application re-engineering effort, security, and runtime performance. More specifically, we ask the following research questions:-

(RQ1) Porting effort. From an application developer's point of view, what is the effort required to port the application and get it running within the enclave?

(RQ2) Application re-engineering effort. Suppose that an application developer wishes to re-engineer the application by deciding that he only wants to run a portion of the application within the enclave. After the application developer has decided what code to run within the enclave, what is the amount of effort that he needs to invest to get the code running within the

enclave?

(RQ3) Security. How much trusted code runs within the enclave, in addition to the application's own enclave code?

(RQ4) Runtime performance. What is the runtime performance overhead of each of these approaches, and how do they compare to native execution (i.e., executing the code without enclaves)?

To answer the above research questions, we selected a representative of each model. We selected Graphene-SGX [159] for the library OS model, Panoply [136] for the library wrapper model, and Porpoise for the instruction wrapper model. Then, we took a few popular applications and ported them to SGX enclaves to answer the above research questions. The selected benchmark applications consist of an in-memory key-value store (Memcached), cryptographic libraries (OpenSSL), a web server (H20), and a Python interpreter (Cpython).

The detailed results from the study are presented in Chapter 3. To summarise the results, the library OS model and the instruction wrapper model are suitable for quickly porting existing applications to SGX enclaves. Furthermore, the library OS model even provides binary compatibility that enables developers to rapidly prototype their application for SGX enclaves. Whereas the instruction wrapper model required re-compilation. Therefore, it requires more effort than the library OS model to port applications. In contrast, the library wrapper model needs extensive porting efforts as it requires writing additional wrappers for the missing library functions.

In performance evaluation, no one model emerged as a clear winner with respect to runtime performance, and the developer must choose the enclave programming model that works best for the application at hand. Library OSes can provide good performance for some applications, *e.g.,* by offering caching and avoiding domain crossings. However, because the enclave execution by itself imposes overheads, and library OSes execute entirely within the enclave, they may also offer poor performance in some cases. The instruction wrapper and library wrapper models do offer the potential for better performance if software developers have the flexibility to profile and re-engineer their applications by reducing domain crossings.

As the library OS model provides binary compatibility for porting applications, we selected the library OS model, Graphene-SGX, for porting deep learning workloads to SGX enclaves, as it provides out-of-the-box support for the Python interpreter and TensorFlow framework, one of the popular frameworks for running deep learning workloads.

1.3.3 Protecting DL Models on Public Cloud Platforms

From the previous study, we learned that memory-intensive applications – deep learning workloads are one of those applications – can incur an order of magnitude performance penalty in SGX enclaves, irrespective of the porting model, when they exceed the fixed protected memory usage offered by SGX. Further, deep learning inference runs slower on CPUs and can be accelerated by faster processors. However, SGX cannot securely access other untrusted system resources, including faster processors. Therefore, in the second part of the dissertation, we focus on the performance of deep learning models with trusted execution environments by avoiding the performance penalty imposed by SGX on memory-intensive applications, and utilise faster and untrusted processors to speed up the inference workload.

We present MazeNet in Chapter 4, a novel deep learning inference system that utilises a combination of trusted and untrusted hardware for accelerating deep learning workloads, while protecting the privacy of deep learning models on public cloud platforms. MazeNet converts pre-trained models into MazeNet models and deploys them on trusted and untrusted hardware to provide secure inference services. It uses three key techniques to avoid the performance penalty of SGX enclaves and speed up the inference service with untrusted hardware.

First, memory-intensive deep learning applications incur swapping when their memory needs are higher than the size of the Enclave Page Cache (EPC), a portion of cryptographically protected main memory. When applications exceed the size of EPC, the SGX driver in the kernel swaps out a few EPC pages to unprotected memory. As the swapping operation is computationally expensive due to cryptographic operations, applications incur a performance penalty. Therefore, MazeNet splits the given model into smaller models referred to as submodels, such that each submodel fits within the protected memory offered by Intel SGX.

Second, MazeNet outsources a subset of submodels to untrusted environments to speed up the inference process. However, outsourcing to an untrusted environment compromises the confidentiality and integrity of outsourced submodels, which were earlier protected by the trusted execution environment. Therefore, MazeNet proposes a secure outsourcing scheme to protect the confidentiality of outsourced submodels. It outsources both the linear and non-linear layers. Before outsourcing a submodel, MazeNet cloaks the outsourced submodels and deploys them on an untrusted runtime. During cloaking, synthetic layers and synthetic neurons are added to hide the original submodel architecture and weights. To recover the embedded results from the output produced by the cloaked submodels, the enclaves have access to the keys that were generated during the cloaking phase to filter the synthetic results from the embedded results.

Finally, the computations outsourced to an untrusted environment can be tampered with by the adversaries. Therefore, MazeNet employs a digital signature scheme to detect tampering in outsourced computations. The cloud vendor signs the inputs and outputs of the outsourced submodels and submits the results along with the signatures to the enclaves. The signatures are later verified by re-evaluating the computations during an auditing phase.

We have implemented MazeNet on top of the TensorFlow framework. It consists of two components: Model Builder and Model Manager. Model Builder takes models stored in TensorFlow `savedModel` format to build MazeNet models, where the cloaked submodels are exported in standard `savedModel` format, while the remaining submodels, which execute within TEEs, are exported in `TFLite` format. For inference, Model Manager deploys MazeNet models to provide secure inference services. It exposes an API for users to query the deployed models. On receiving inputs from users, it routes the inputs through a series of submodels within TEEs and cloaked submodels in untrusted environments to compute the prediction results.

To evaluate the benefits and costs of running MazeNet models, we transform popular convolutional neural networks into MazeNet models and compare their performance against a secure baseline system where the unmodified model runs within a trusted execution environment. The benchmark models used in our evaluation range from sequential models, such as VGG16 [74], models with residual connections, ResNet50 [56], to models with highly connected layers, DenseNet201 [59]. Our evaluation focuses on two key aspects of the inference workloads: throughput and latency. Experimental results demonstrate that MazeNet can improve the performance of deep learning inference workloads, up to 30x improvement in throughput and 5x improvement in latency as compared to a secure baseline.

Recently, Nvidia has released Confidential Compute (CC) to extend the trust boundary of TEEs from CPUs to GPUs. Nvidia CC enables VM-based TEEs to securely deploy DL models on GPUs while protecting the confidentiality and integrity of the models from privileged adversaries. The communication over the untrusted PCIe channel is protected with pre-negotiated encryption keys.

However, Nvidia Confidential Compute is limited to high-end server-grade GPUs after Hopper architecture [107]. The techniques presented in Chapter 4 can be employed to secure inference workloads running on GPUs from other manufacturers and on older GPUs where Nvidia CC is not available.

1.4 Summary of Contributions

This dissertation supports the following thesis statement:

Hardware-based trusted execution environments can be leveraged to run private deep learning inference workloads on public cloud platforms with practical runtime performance while protecting the privacy and integrity of the model.

To support the above thesis statement, this dissertation makes the following contributions:

- **Porpoise.** A framework based on the instruction wrapper model to port commodity applications to Intel SGX enclaves. To port applications, developers do not need to modify applications' source code; they only need to recompile the application with Porpoise libraries. Porpoise has successfully ported several popular applications, including a web server (H2O), cryptographic libraries (OpenSSL), a Python interpreter (CPython), and a key-value store (Memcached).

- **Evaluation of SGX porting frameworks.** A comparative study of multiple frameworks to port commodity applications to Intel SGX enclaves. The study classifies application porting frameworks into three categories: the instruction wrapper model, the library wrapper model, and the library OS model, and evaluates them on four parameters: porting effort, re-engineering effort, security, and runtime performance. This helps developers to make informed decisions when selecting a framework.

- **MazeNet**. A framework to run deep learning inference workloads on a set of trusted and untrusted hardware while protecting the privacy of the model weights from privileged adversaries. The trusted hardware protects the privacy of the model, while the untrusted hardware speeds up the inference workflow. It introduces a secure outsourcing scheme to offload both the linear and non-linear operations in deep learning models to an untrusted environment. Experimental evaluation shows that MazeNet can provide up to 30x improvement in throughput and 5x improvement in latency.

Chapter 2

Trusted Execution Environments

In this chapter, we will look into the various trusted execution environments offered by leading silicon vendors. Arm was the first silicon vendor to introduce support for a trusted execution environment with Arm TrustZone [8] in the year 2004. However, its wider adoption faced challenges, as the only device vendor and the platform owner can build applications for the TrustZone. Later, Intel announced SGX in 2013 [97] with a strong threat model suitable for cloud platforms and applications. AMD introduced Secure Encrypted Virtualization (AMD SEV) [34], which supports running an entire virtual machine within the trusted execution environment. Later, Intel and Arm also started support for virtual machine-based trusted execution environment with Intel Trust Domain Extension (TDX) [4] and Arm Confidential Compute Architecture (Arm CCA) [93]. Recently, Nvidia introduced TEE support for GPUs [107], which extends VM-based TEEs of CPUs to GPUs.

The main objective of TEEs is to protect the confidentiality and integrity of private data from adversaries. An adversary can perform the following attacks to compromise the protected region. First, it can launch software attacks, e.g., memory remapping attacks [9], from the host operating system or hypervisor to break the security isolation of the TEE. Second, it can carry out hardware attacks with DMA-capable devices to leak sensitive data from the main memory. Third, it can perform physical attacks on the system, e.g., probing physical interfaces of DRAM and PCIe [142, 78, 53, 52, 167, 95]. Lastly, the attacker can carry out side channel attacks [161, 26] to leak sensitive data on multi-tenant systems.

To protect the code and data from the above attacks, different TEEs use different mechanisms to protect code and data. The security responsibility is split across the software and hardware components of the TEEs. Some of the TEEs, e.g., Intel SGX, rely on the hardware support to protect the code and data, while some rely on trusted software or firmware, e.g., Arm CCA, to provide the security guarantees.

2.1 Intel Software Guard Extensions (SGX)

Intel SGX offers low-level instructions, e.g., `ECREATE, EADD,` and `EEXTEND`, to create an isolated and protected region in their program address space. The code and data within this protected region, referred to as an enclave, are protected from privileged adversaries, including the host operating system, BIOS, and firmware. The code and data inside the enclave can be accessed only from the accesses originating within the enclave. SGX restricts the host kernel from accessing the memory of an enclave to protect its confidentiality and integrity. Also, an enclave cannot access the kernel memory, i.e., both the kernel and the enclave are mutually distrusting. The enclave and the kernel can communicate through the memory allocated to the user space.

SGX sets aside a fixed portion of the main memory during system boot to store pages of enclaves, referred to as Enclave Page Cache (EPC). The EPC is encrypted by a dedicated hardware, Memory Encryption Engine, when data is moved from CPU cache to main memory to protect the confidentiality of data from physical attacks.

A program executing within the enclave has its own stack, heap, code and data segments. A program running in user space can enter an enclave with `EENTER` instruction, which changes the CPU mode of execution from user to enclave mode. The control enters the enclave through a set of predefined entry points. Once the enclave has completed the execution, it can return the control to the user space with the `EEXIT` instruction. In case any exception occurs during the enclave execution, the enclave exits the enclave asynchronously, where the CPU state is saved within the protected memory and registers are filled with synthetic data before returning control to the host. Upon servicing the exception, the program can re-enter the enclave with `ERESUME` instruction.

Although the host OS cannot access the memory allocated to enclaves, it is still in control of memory allocation. A malicious OS may attempt to compromise the confidentiality and integrity of an enclave through page remapping attacks as it controls the page tables. SGX uses an Enclave Page Cache Map (EPCM) to track virtual-to-physical address translations to detect tampering in address translation. External access to enclave memory generates a fault.

A program running within an enclave can use `EREPORT` instruction to get an attestation report that a remote party can use to verify the correctness of the enclave state and Intel SGX platform. The report contains measurements of code and data loaded into the enclave and signed by the hardware root of trust key, which was baked into the CPU during manufacturing.

The trusted computing base (TCB) of an enclave application consists of code and data residing within the enclave and the Intel CPU. The host OS and enclave launching user-space libraries for creating and managing the enclave are outside of the TCB.

2.2 Intel Trust Domain Extensions (TDX)

After SGX, Intel launched Trust Domain Extensions (TDX) [4] to protect virtual machines on public cloud platforms. Intel TDX introduces trust domains, a trusted execution environment that can host virtual machines. Intel TDX is a set of extensions to the Intel VMX [65] architecture to create and manage virtual machines in trust domains. It protects a trust domain from host virtual machine managers, hypervisors, legacy virtual machines and other trust domains on the host platform. It also employs main memory encryption with dedicated hardware present in Intel Total Memory Encryption Multi-key (TME-MK) [71] to protect the main memory from physical attacks.

Intel TDX introduces a new Secure-Arbitration Mode (SEAM) for trust domains in addition to the VMX mode for virtual machines in VMX architecture [65]. The SEAM mode runs an Intel-signed TDX module that manages trust domains within the SEAM memory. Intel employs Trusted Execution Technology (TXT) [18] to ensure that the TDX module is not tampered with during the launch. External memory access to the SEAM memory is prohibited by the hardware, irrespective of their privilege levels.

The TDX module manages the trust domains and virtual machines from launching, scheduling, and allocating resources to them. The host VMM controls the resources on the system and is responsible for allocating resources to trust domains. The VMM allocates a set of memory pages to the TDX module. In turn, the TDX module can allocate the assigned pages to trust domains.

The TDX module runs on the same physical CPU as the host and the guest VMs. It is also responsible for managing the security policies, such as which hardware resources to be exposed to trust domains. As the TDX module runs on the main CPU, it can enforce computationally expensive security policies.

Intel TDX offers attestation services through the TDX module with assistance from SGX to securely provision client virtual machines. The TDX module measures memory pages that were added to the trusted domain during launch. A remote owner or a user can request the trust domain to provide an attestation report to verify the initial execution state of the virtual machine. The TDX module generates an attestation quote and then sends the attestation quote to the Intel SGX TD-quoting enclave to produce an attestation report signed by the attestation key, backed by a hardware root of trust.

The TCB of a TDX trust domain consists of code and data within the trust domain, which includes the guest hypervisor, operating system, and user applications. Additionally, the TDX module, attestation software, and the Intel CPU are part of the TCB. However, it excludes

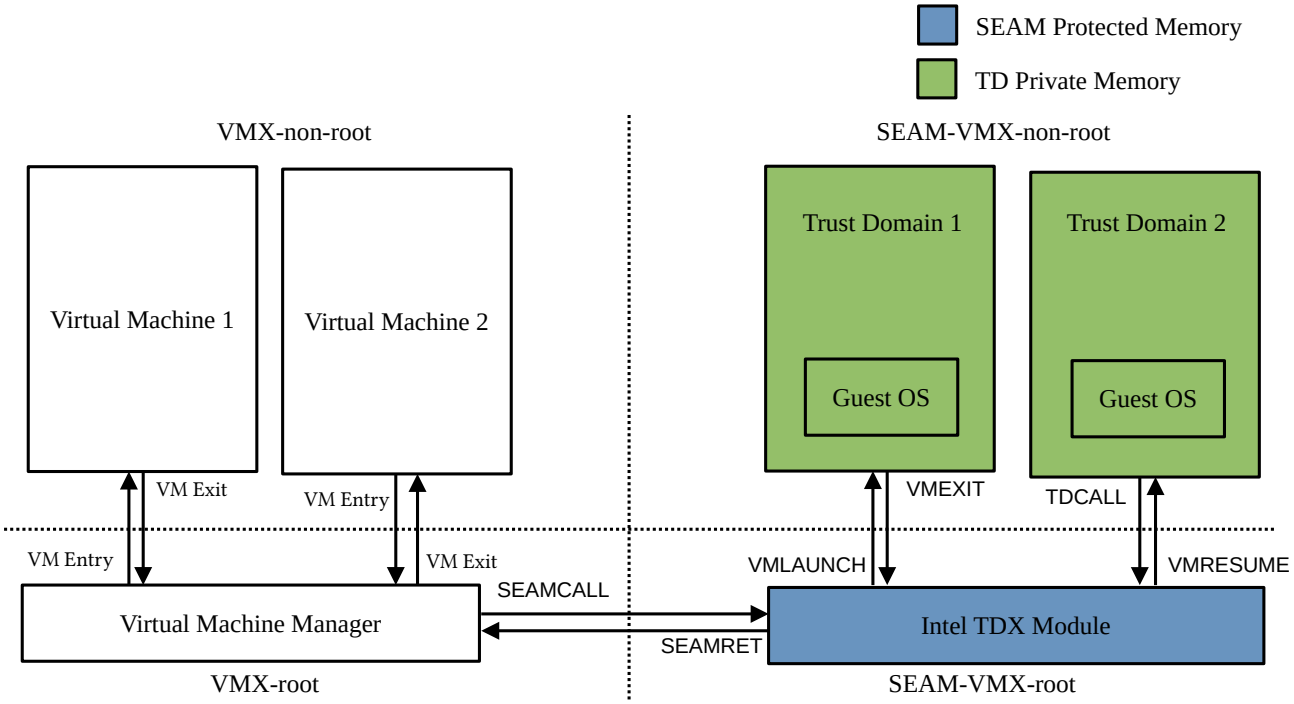


Figure 2.1: Intel TDX architecture [4]. Intel TDX introduces a new Secure-Arbitration Mode (SEAM), in addition to VMX mode, to create isolated environments called Trust Domains. In SEAM mode, a processor can be in two states: SEAM-root mode or SEAM-non-root mode. SEAM-root mode hosts an Intel-signed TDX module that manages the trusted domains executing in SEAM-non-root mode. TDX prohibits external access to the SEAM-protected memory hosting the TDX module.

BIOS, firmware, host hypervisor, operating system, system software and other hardware devices present on the system.

In comparison to Intel SGX, Intel TDX offers a strong alternative to run confidential workloads on public cloud platforms. Application developers can quickly port their applications to confidential virtual machines, whereas SGX requires applications to be rewritten in an SGX-aware manner. However, Intel TDX suffers from a higher TCB as the guest operating system is part of the TCB.

One major challenge with multiple implementations of trusted execution environments is that supporting each trusted execution environment becomes challenging for hypervisors. Recently, Cloud Hypervisor has deprecated support for Intel TDX [30], which further increases the barrier for its adoption.

2.3 AMD

AMD has introduced a series of technologies to secure private computation on public cloud platforms. Each of the technologies extends the previous ones to enhance the protection and reduce the attack surface. Initially, AMD introduced Secure Memory Encryption (SME) [34] to encrypt the contents of the main memory. Subsequently, it presented Secure Encrypted Virtualization (SEV) [34] to run encrypted virtual machines. Later, it launched Encrypted State (ES) to protect the confidentiality of the CPU state from hypervisors during context switches and interrupts. Finally, it rolled out Secure Nested Paging (SNP) [9] to preserve the integrity of encrypted virtual machines.

2.3.1 AMD Secure Memory Encryption

When the data is stored on storage disks, data-at-rest, it is encrypted to protect the confidentiality and integrity of data from unauthorized access. However, when the data is loaded into DRAM and decrypted, it is available in plaintext in the main memory and referred to as data-in-use. The plaintext data in the main memory can be compromised by privileged adversaries, e.g., malicious system administrators or maintenance staff at the data centres, through software, DRAM interface snooping [142, 78], and cold boot attacks [53, 52, 167, 95]. To protect the confidentiality of data residing in main memory, AMD introduced Secure Memory Encryption (SME), which encrypts the contents of the main memory to protect the confidentiality of the data from physical attacks.

AMD Secure Memory Encryption adds dedicated hardware in the memory controller to encrypt and decrypt the contents of DRAM on the fly. The data is decrypted when it is read from DRAM to CPU caches and encrypted when it is written back to DRAM. This encryption and decryption add an overhead to memory accesses. Therefore, only sensitive data should be identified and placed within encrypted memory to limit performance degradation. The dedicated hardware implements the Advanced Encryption Standard (AES) encryption scheme to protect the content of the main memory. AMD SME relies on a physically separate co-processor, AMD Secure Processor (AMD-SP), to manage the security. The co-processor runs a trusted firmware to manage the encryption keys. The keys are not exposed to the software to prevent accidental leakage from side channel attacks. AMD SME can protect workloads from physical attacks, such as memory probing attacks and cold boot attacks; however, the protection is insufficient if the hypervisor or the operating system is compromised or contains vulnerabilities.

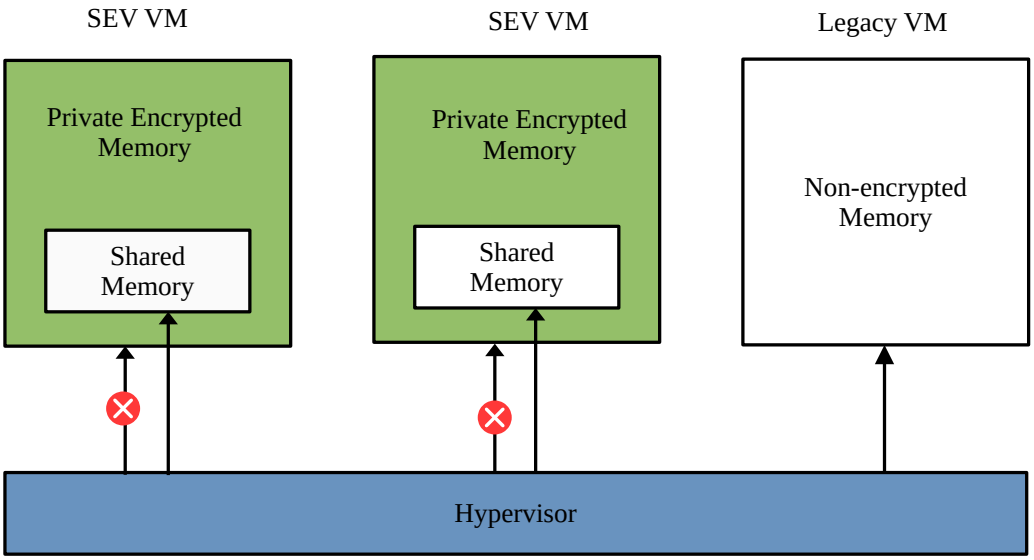


Figure 2.2: AMD SEV memory access semantics [34]. AMD SEV introduces a new security model where lower-privileged guest virtual machines are protected from the higher-privileged host hypervisor. The hypervisor still controls resource allocation to guests, but cannot access the allocated memory. The guest and the host can communicate through shared memory pages.

2.3.2 AMD Secure Encrypted (SEV)

In cloud environments, the host and the hypervisor have full access to the guest virtual machines. A compromised host can inspect and modify the guest state with memory scanning and debugging utilities [148, 120] to steal cryptographic keys or compromise execution integrity. AMD Secure Encrypted Virtualization extends AMD virtualization techniques (AMD-V) to protect guest virtual machines from the compromised host by introducing a new security model for CPU execution.

Traditional security models follow a hierarchical approach where the code executing with higher privilege levels can access lower-privileged resources. In contrast, AMD SEV introduces a new security model where the guest virtual machine running with lower privileges does not trust the host hypervisor running at a higher privilege level. AMD SEV modifies the memory access semantics that prohibit privileged hypervisors from accessing the private memory allocated to guest virtual machines, as illustrated in Figure 2.2. The host still controls the resource allocation, such as allocating memory pages to guest virtual machines. However, they cannot inspect or modify private memory pages allocated to guests. The guest can share a few memory pages with the host hypervisor to enable data transfers between the host and the guest. Further, external hardware access from DMA devices to the private pages of a guest is prohibited. DMA devices can use the shared pages to communicate with the guest virtual machines.

AMD SEV relies on AMD-SP to provide measurements that are signed by a hardware root of trust to prove to the remote owner that the virtual machine was launched correctly on a SEV-enabled platform. The attestation report also contains the security version number of components that are part of the trusted computing base to detect outdated platforms that may contain security vulnerabilities.

To summarise, AMD SEV provides strong memory confidentiality guarantees, similar to Intel SGX, for workloads running within the encrypted virtual machines. It relies on a co-processor and a trusted firmware to manage the security of encrypted virtual machines. However, similar to Intel TDX, the trusted computing base of a workload running in an encrypted VM is much higher than an SGX enclave, as the operating system and other services, in addition to the application, are part of the trusted computing base.

AMD SEV had a few limitations, such as the CPU state being visible in cleartext to the hypervisor and missing support for memory integrity. AMD later fixed these limitations in the following releases, which are discussed in the next section.

2.3.3 AMD Encrypted State (ES)

In AMD-SEV, discussed in the previous section, the CPU register state is visible in plaintext to the hypervisor on certain events, e.g., VM exit, even if the guest is running within an SEV-enabled virtual machine. The registers may contain sensitive data such as encryption keys, which can be leaked or tampered with by a malicious hypervisor. Further, the hypervisor can compromise the integrity through replay attacks by restoring the registers to a previous state or by modifying the RIP (instruction pointer) register.

AMD Encrypted State (AMD-ES) secures the CPU register state during program execution and VM exit events to prevent the above attacks. To protect CPU state, it encrypts the state of the registers and stores them in the private memory of the VM during exit events for interrupt processing or hypervisor emulation before transferring control to the host hypervisor. On the VM resume, the hardware decrypts the register state and restores the CPU state.

However, the hypervisor needs certain register values to process the interrupt. For example, if the guest VM has invoked the CPUID instruction, then the hypervisor needs the RAX register values to service the CPUID instruction. AMD-ES enables a guest to selectively expose the required registers to the hypervisor.

On VM exits, the hypervisor informs the guest VM that it has invoked an instructions that require hypervisor assistance. Then, the guest places the necessary register values in plaintext that are required by the hypervisor to service the interrupt. Once the hypervisor has processed the request, control returns to the guest OS, which verifies the results and resumes execution.

2.3.4 AMD SEV Secure Nested Paging (SEV-SNP)

AMD SEV Secure Nested Paging (SEV-SNP) [9] further strengthens the security of guest VMs with integrity protection. In the earlier AMD SEV implementation, Secure Memory Encryption protects the confidentiality of guest virtual machines on main memory, and AMD Encrypted State (AMD SEV-ES) protects the confidentiality of the CPU state for guest virtual machines.

However, the hypervisor still manages the memory allocation and page tables for the guest. A malicious hypervisor can use software-based attacks, e.g., memory remapping, to compromise the integrity of the guest. Although the hypervisor cannot read plaintext data of the guest data due to memory encryption, it can write arbitrary ciphertext to main memory, which corrupts the guest state. During guest execution, the guest will observe arbitrary data at those locations. Further, the host can perform replay attacks where it changes the encrypted pages in main memory with past ciphertext texts, which causes the guest to read stale data. As the hypervisor controls the page tables as well, it can perform memory aliasing and memory remapping attacks to compromise the guest. In memory aliasing attacks, the hypervisor maps multiple virtual pages to a single host physical page. Whereas, in memory remapping attacks, the hypervisor maps a single guest page to multiple system physical pages.

In the earlier SEV threat model, the hypervisor was considered vulnerable but benign. In contrast, SEV-SNP assumes the hypervisor to be malicious. Thus, SEV-SNP further reduces the attack surface by removing the hypervisor from the trusted computing base. A program running on an encrypted guest trusts the code and data within the virtual machine, along with AMD Secure Processor (AMD-SP) and accompanying trusted firmware.

AMD SEV-SNP uses a Reverse Map Table (RMP) [9], as shown in Figure 2.3, to enforce integrity protection. An RMP table is a system-wide data structure to track ownership of each physical page. As RMP is privileged data, software access is prohibited to modify RMP entries. Instead, SEV-SNP introduces a set of instructions for updating the table, allowing the guest VMs to validate the pages that are allocated to them. Once pages are validated, the hypervisor cannot remap those pages; otherwise, the mapping will become invalid, which will be detected by the guest VM.

AMD SEV-SNP offers both confidentiality and integrity protection to the guest virtual machines, similar to Intel SGX security guarantees for user enclaves. However, the level of isolation is different in both implementations. AMD SEV-SNP offers isolation at the VM level, whereas SGX protects code and data at the process level.

Due to VM-level isolation, client applications running in SNP-enabled virtual machines will have a higher trusted computing base as the guest kernel and other services within the VM

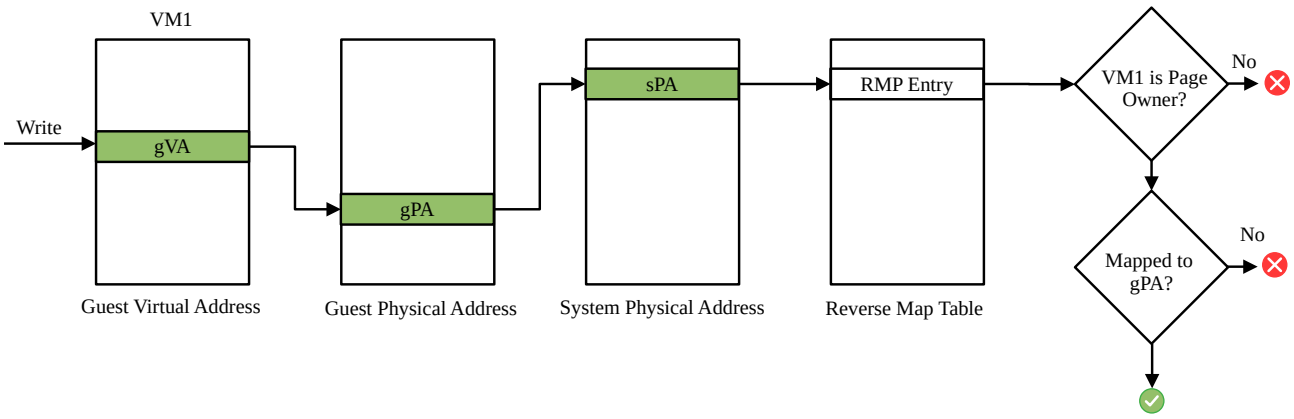


Figure 2.3: AMD SEV-SNP uses a Reverse Map table [9] to track ownership of each physical page on main memory. Write access to the private pages is granted only to the owners, guest VM, hypervisor, or the AMD Secure Processor.

are included in the TCB. For instance, a standard virtual machine with a Linux kernel loads around 5MB of privileged kernel code, and any vulnerability in the privileged code compromises the entire VM. In contrast, SGX requires minimal SGX-related code, in addition to the user application, which minimizes the attack surface.

As SEV-SNP offers VM-based TEEs, existing workloads can run unmodified within the virtual machines, which reduces the barrier to transitioning existing workloads to trusted execution environments. In contrast, SGX requires applications to be rewritten in an SGX-aware manner, which adds development cost and time to move workloads to SGX enclaves or use specialised library OSes.

A major advantage of virtual machine-based TEE over Intel SGX is that it does not require specialised software stacks, e.g., library OS [159], or modifications to the application source code [136] to run existing applications. Despite years of development effort poured into library OSes and specialised kernels have not achieved full binary compatibility with existing workloads [51] due to which niche workloads either do not run or incur a performance penalty because of missing optimisations that are incorporated in standard operating systems over the decades.

AMD SEV-SNP is an alternative VM-based trusted execution environment and offers similar features and functionality to Intel TDX. In contrast to Intel TDX, it runs the trusted firmware on a separate co-processor rather than the host CPU, as in Intel TDX, which offers some protections from side channel attacks. However, similar to Intel TDX, it suffers from a larger TCB due to the guest kernel being part of the TCB.

2.4 Arm

Arm was the first silicon vendor to introduce support for a trusted execution environment, TrustZone [8, 81, 82], that was commercially and widely available on inexpensive mobile devices. Arm introduced TrustZone in Armv6 architecture [92] with ARM1176JZF-S processor [91]. ARM TrustZone provides a hardware-based secure and isolated execution environment that protects applications from privileged software-based attacks arising from hypervisors and operating systems. It extends the Arm architecture with hardware-based security features to protect applications in the trusted execution environment.

However, the wide adoption of TrustZone by the application developers always remained a cause of concern due to the locked-in nature of TrustZone. An application developer has to rely on the platform owner or device manufacturer to include their application in the set of trusted applications that can run within the trusted execution environment [49, 31, 115]. As each application that is added to the set of trusted applications increases the trusted computing base, any vulnerability in one of the trusted applications can compromise the security of the whole system. Therefore, the use of TrustZone was limited to a small set of applications that provide trusted services such as biometric authentication, DRM media applications, and cryptographic key management to other applications on the device. Some of the device manufacturers have used TrustZone to provide additional security, as in Samsung Knox [37].

To overcome the limitations of TrustZone, Arm introduced Arm Confidential Computing Architecture (CCA) [93] in Armv9 architecture [94] to run multiple isolated execution environments within a system. ARM CCA enables independent application developers to create an isolated and protected environment to run their sensitive applications without relying on OEM vendors. It provides stronger, hardware-backed confidentiality and integrity guarantees for the applications running within the trusted environment.

2.4.1 Arm TrustZone

Arm TrustZone extends the Arm architecture to create a hardware-enforced isolated environment, referred to as a secure world, that protects trusted applications and sensitive data in the secure world from malicious or compromised software in the normal world. The secure world runs applications with a separate software stack consisting of a trusted operating system, independent from the normal world.

ARM TrustZone protects the secure world against an adversary who controls the privileged software running in the normal world. The adversary can tamper with the host hypervisor or the operating system to compromise the security of trusted services and data within the secure

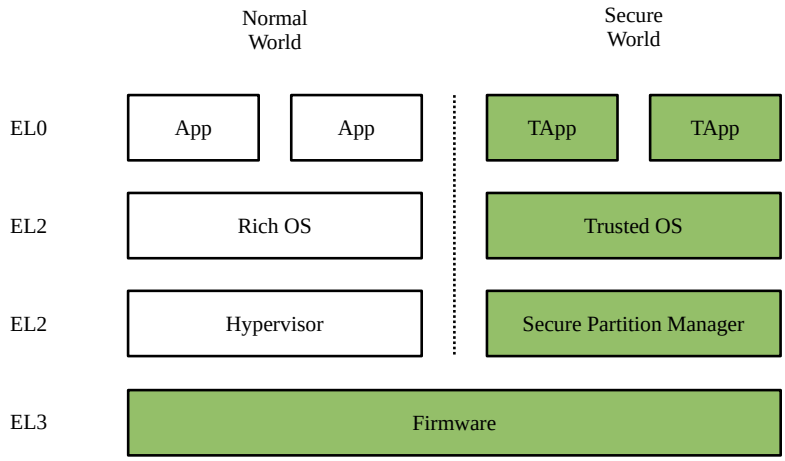


Figure 2.4: Arm TrustZone architecture [8]. Arm TrustZone partitions the physical resources into two logical worlds, secure and normal. The secure world hosts trusted applications along with a small trusted kernel, whereas the normal world runs a feature-rich kernel and extensive applications. The secure world has higher privileges and can access the resources allocated to the normal world, whereas the normal world does not have access to the secure world.

world.

To protect the secure world, TrustZone partitions hardware resources into secure and non-secure components. The CPU cores, cache, memory, and system bus are partitioned into logical components to isolate the code and data in the secure world from the normal world. The code and data in the secure world are not accessible to the software running outside the secure world. However, the secure world can map normal world memory to access its contents in the secure world. The secure world typically runs a small trusted OS (e.g., OP-TEE [31] on Linux, Trusty TEE based on Little Kernel for Android[49]), and hosts a fixed set of trusted applications that provide trusted services to the non-secure world. These trusted applications mostly deal with sensitive, private data.

On the other hand, the normal world hosts a full-fledged operating system referred to as Rich OS, e.g., Android or Linux, to provide services to the user. It runs feature-rich applications such as messaging, banking apps, browsers, and a rich graphical user interface.

ARM TrustZone offers four different privilege levels from Exception Level 0, `EL0`, being the lowest and Exception Level 3, `EL3`, being the highest, as shown in Figure 2.4. The processor can be any one of the privilege levels in both worlds. However, it always executes in secure mode at `EL3` privilege level. A secure monitor or trusted firmware runs at `EL3` that handles transitions between both worlds. A processor running in the non-secure world can switch to the secure world with the Secure Monitor Call (`SMC`) instruction that traps the CPU into the highest privilege level, Exception Level 3 (EL3). The secure monitor saves the CPU context

and performs a world switch.

ARM TrustZone can protect trusted applications running in the secure world from the normal world. However, the trusted applications and the trusted operating systems running in the secure world are still part of the trusted computing base. Any vulnerability in the trusted operating system or trusted applications can compromise the security of the secure world and applications running in the normal world [124, 133]. Although the size of trusted operating systems and trusted apps is small, vulnerabilities have been reported in the past in trusted operating systems and trusted apps [24, 3].

An optional secure partition manager, introduced in ARMv8.4-A architecture, in the secure world with `EL2` privileges can further partition the secure world into isolated environments such that trusted applications in the secure world do not need to rely on a common kernel, i.e., each application can have its own kernel. Therefore, any vulnerability in any one of the kernels does not affect the trusted applications of the remaining kernels. Furthermore, some of the functionalities of secure monitor which does not need the highest privileges can be moved to an isolated environment running with lower `EL1` privileges in the secure world.

Although Arm TrustZone provides a good starting point for securing applications and services on mobile devices, it lacks sufficient protection to defend against physical attacks due to the lack of memory encryption. An adversary can probe system buses to leak sensitive data such as model weights. Further, the dependency on the platform owner to include additional applications increases the barrier for Arm TrustZone adoption. Due to insufficient security guarantees, TrustZone is not suitable for securing deep learning models.

Arm fixed the limitations of TrustZone in the following Confidential Computing Architecture, which enables large-scale deployment of trusted applications and private data on the Arm platform.

2.4.2 Arm Confidential Computing Architecture (CCA)

ARM Confidential Computing Architecture (CCA) enables third-party application developers, independent of the platform owner and OEM vendor, to run confidential computations on Armv9-A processors.

Arm introduced Realm Management Extensions (RME) in the Armv9 architecture [83]. RME is a set of hardware and software extensions to create isolated execution environments called Realms. It introduces two new worlds – realm and root – in addition to the existing secure and normal world in the Arm TrustZone, as illustrated in Figure 2.5. The realm world is mutually distrusting with the secure world, i.e., it cannot access the resources of the secure world, and the secure world cannot access the resources of the realm world.

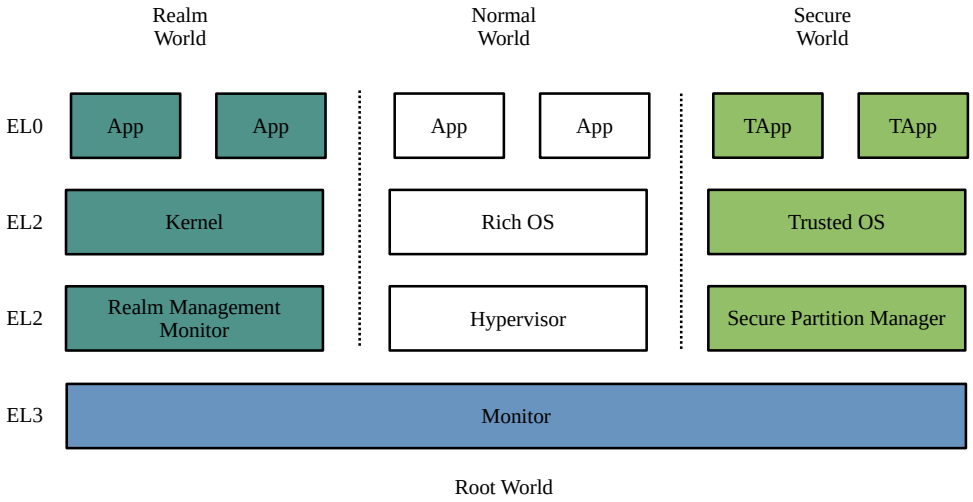


Figure 2.5: ARM CCA architecture [83]. It introduces two new worlds, Realm and Root, in addition to the secure world and the normal world of the Arm TrustZone architecture. The secure world and the realm world are mutually distrusting, and they communicate through a secure monitor that runs in a new root world with the highest privileges, `EL3`.

The Realm world has three privilege levels `EL0`, `EL1`, and `EL2`, similar to privilege levels in the secure and normal world. All three worlds, secure, normal and Realm, interact through a secure monitor running at the highest privilege level `EL3`. As the three worlds need to trust the secure monitor, it is moved from the secure world in ARM TrustZone architecture to a new root world of its own.

A Realm Management Monitor (RMM) is a trusted firmware that runs at `EL2` privilege level in the realm world. It creates isolated execution environments referred to as realms. Realms are confidential virtual machines where kernels run with `EL1` privileges and applications with `EL0` privileges in the realm world. RMM is responsible for maintaining confidentiality and integrity between different realms. It exposes a Realm Management Interface (RMI) to the hypervisor in the normal world to create and manage realm VMs. RMM only ensures the security of realms, and other management tasks such as CPU scheduling and resource allocation are under the control of the host hypervisor. The RMM is kept minimal, and all the device virtualisation and emulation tasks are delegated to the host hypervisor. A hypervisor in the normal world uses RMI to manage memory and other resources for realms. As the host hypervisor controls the memory allocation, it may attempt to compromise the confidentiality and integrity of realm VMs through memory mapping attacks. RMM protects against such attacks by maintaining a nested page table and verifying RMI requests from the host hypervisor.

To protect the contents of the main memory from physical attacks [167], RME encrypts the realm world, the secure world and the root world with different encryption keys. It also

offers an optional Memory Encryption Context (MEC) [84] that can encrypt each Realm with a separate key.

Arm CCA provides attestation capabilities for the remote application owner to verify the integrity and confidentiality of the host platform. An attestation report signed by a hardware root of trust contains measurements of software components that may influence the security of a realm, such as the secure monitor running at `EL3`, the RMM running at `EL2`, and the VM image. The attestation of the initial realm state provides assurance to the remote owner that the initial VM image was not tampered with during the deployment. Once the owner has verified the authenticity of the platform, it can provision sensitive data to the realm VM required to proceed further.

Further, Arm CCA extends TCB beyond the CPU to include on-SoC peripheral devices in the trust boundary after authentication. A realm VM can use TEE Device Interface Security Protocol (TDISP) to authenticate devices [93]. On successful authentication, the device is assigned to the realm VM, while other software access, e.g., host hypervisor and other realm VMs, to the device is prohibited. The attested device can DMA into the assigned realm memory. However, devices outside the SoC must use encryption when communicating with realm VMs to protect against confidentiality, integrity, and memory replay attacks.

The trusted computing base of a realm consists of the kernel and applications code and data within the Realm, RMM firmware, secure monitor firmware, Arm CPU and RME hardware extensions. Other realm VMs and non-realm software, such as firmware of untrusted devices on the system, privileged code in the secure world, are outside the trusted computing base.

Arm CCA offers another alternative implementation for running confidential virtual machines similar to AMD SEV and Intel TDX. It relies extensively on the trusted monitor running at the highest privilege level to ensure the confidentiality and integrity of the realms. Similar to Intel TDX, it runs the trusted monitor on the main CPU and can implement computationally expensive security policies.

Arm CCA opens an exciting opportunity for application developers to leverage trusted execution environments on mobile devices to protect the sensitive portions of their application from privileged adversaries. It will be interesting to see how realms are incorporated into the vast array of commodity applications. However, running computationally expensive deep learning workloads on mobile devices may not be suitable, as it dramatically reduces the battery life.

2.5 Nvidia Confidential Compute (CC)

Nvidia Confidential Compute (CC) [107] extends the trust boundary of trusted execution environments on CPUs to the GPUs. Nvidia introduced Confidential Compute in the Nvidia Hopper architecture with H100 GPUs. It supports selected CPU-based TEEs, Intel Trust Domain eXtensions (TDX), AMD SEV-SNP and Arm Confidential Computing Architecture, which offers support for confidential virtual machines. With Nvidia CC, a confidential virtual machine can securely outsource confidential computing workloads to supported Nvidia GPUs. Nvidia CC protects the confidentiality and integrity of code and data when it runs on the GPUs.

Nvidia Confidential Compute provides hardware and software features to protect the confidentiality and integrity of confidential workloads on the GPU. It protects the data from unauthorised software and hardware access from the privileged hosts, hypervisors, DMAs, and other attached peripheral devices. It employs mechanisms for GPU TEEs to protect against some of the side-channel attacks from other confidential virtual machines on the same host, hypervisors, and physical attacks.

Nvidia Confidential Compute assumes that the adversary has limited physical access to the GPUs. The adversary can snoop hardware interfaces, PCIe bus, NVLink, and DRAM. The adversary can read and modify data on non-secure shared memory and perform replay attacks and denial of service attacks.

The host is responsible for managing the resources on the system, including the Confidential Compute on the GPUs, but it cannot read or write to the protected memory that belongs to either CPU TEEs or GPU TEEs. The host is responsible for creating and managing both the TEEs, but cannot access private memory allocated to the TEEs.

The CPU TEE would need to transfer data to the GPU TEE over untrusted channels, such as PCIe. A malicious host can compromise the confidentiality and integrity of the data when it is passing through an untrusted channel. To ensure confidentiality, Nvidia CC encrypts the data before transferring it over an untrusted channel. The CPU and the GPU TEE establish a symmetric encryption key and place the encrypted data on bounce buffers in untrusted memory, which can be accessed by the CPU TEE, the GPU TEE, and the host. For integrity, Nvidia CC uses digital signatures to detect integrity violations during transmission over untrusted channels.

Nvidia CC offers secure boot and remote attestation similar to CPU TEEs. The GPU firmware is verified during the GPU boot process to ensure that non-tampered firmware and microcode that was released and signed by Nvidia is loaded. If the verification fails, the boot process is halted. During the boot, the GPU attests its authenticity to the host through an

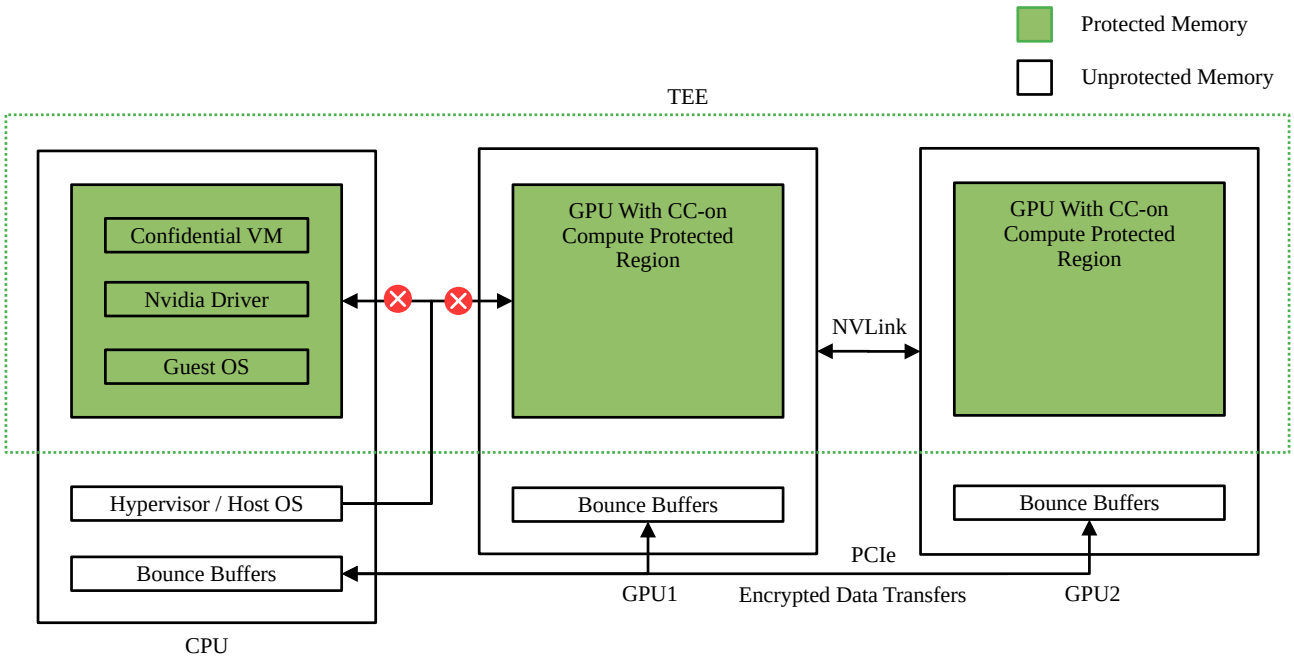


Figure 2.6: Nvidia Confidential Compute Architecture [107]. Nvidia Confidential Compute enables support to create trusted execution environments on GPU devices. A confidential virtual machine running on the CPU can securely communicate with the GPU TEE, and unauthorised access to GPU TEE memory from the privileged host kernel and hypervisor is prohibited by the hardware. When the data is sent over untrusted channels, such as PCIe, it is encrypted to protect against physical attacks.

attestation report. The host can verify that the attestation was signed by the Nvidia GPU through a hardware-backed key.

A confidential virtual machine or the users must verify the authenticity of the GPU with confidential compute before running any confidential workload on the GPU. Nvidia provides remote attestation services, similar to remote attestation services offered by CPU-based TEEs, for CPU-TEEs and remote owners to verify that the host has correctly set the confidential compute environment on Nvidia GPUs.

Nvidia CC provides the required security guarantees for running private deep learning workloads on public cloud platforms. However, Nvidia CC is available only on server-grade GPUs with recent architectures starting from Hopper. This leaves earlier GPUs and other hardware accelerators, which do not support a trusted execution environment, unsuitable for privacy-preserving deep learning.

2.6 Conclusion

TEEs are expanding in commercial availability and scope from CPU to peripheral devices. Each TEE offers different security guarantees due to different trusted computing bases, hardware and software-based protection mechanisms, and the targeted platform.

However, due to diverse security models and software support, it becomes challenging for cloud tenants to select the right security solution. For small security-sensitive applications and code that require maximum protection, such as cryptographic libraries and wallets, biometric data, authentication modules, and policy decisions can be deployed on Intel SGX due to the least trusted computing base. However, the application needs to be rewritten or adapted to be SGX-aware. Existing large business logic and applications, such as databases, web services, logging, and monitoring applications, can be deployed on confidential virtual machines supported by Intel TDX, AMD SEV-SNP, and Arm CCA, as they support unmodified client workloads. For deep learning applications and workloads that require hardware acceleration, Nvidia Confidential Compute is the only commercially available option. Based on the recent advances and commercial availability of TEE on diverse devices, it can be expected that TEE will be supported by other hardware accelerators in the future.

Currently, not all devices support trusted execution environments, and legacy devices cannot be retrofitted to include the hardware support required for trusted execution environments. Therefore, we need innovative solutions to utilise those devices.

This dissertation focuses on Intel SGX as it offers a strong threat model suitable for running workloads on public cloud platforms and was the only commercial solution available at the beginning of this work.

Chapter 3

An Evalutation of Methods to Port Legacy Code to SGX Enclaves

3.1 Introduction

Intel's Software Guard Extensions (SGX) [96, 57] technology has now been commercially available since microprocessors using the Skylake micro-architecture were launched in August 2016. Using the facilities of the SGX, user-level applications can create *enclaves* within which they can place their sensitive code and data. Enclaves are cryptographically secured by the hardware so that an adversary cannot observe the data or the computations with in the enclave. SGX's threat model accommodates a powerful set of adversaries, including the most privileged software running on the system, *i.e.,* the operating system or the hypervisor.

This powerful threat model makes SGX attractive for use in public cloud computing platforms. On such platforms, the cloud provider controls the system software. An adversarial cloud provider (or a benign one acting under government subpoena) can leverage this control to completely subvert the confidentiality and integrity of a cloud client. The cloud provider can peek into and arbitrarily modify the state of a client's virtual machines or containers. This makes public cloud computing environments unattractive to clients in several domains that handle sensitive data, such as healthcare and banking. To accommodate clients with such sensitive computing needs, a number of public cloud providers have begun to deploy SGX hardware in their data centres, and offer solutions that allow clients to leverage the capabilities of the SGX to build applications.

A client that wishes to leverage SGX must write its applications to be SGX-aware. An SGX-aware application will place its sensitive data in enclaves, and ensure that the code that operates on this data is also placed in the enclave. The SGX hardware places certain restrictions on the

kinds of instructions that can execute within enclaves, *e.g.,* system calls cannot be executed within an enclave. Enclave code must be written to respect these restrictions, *e.g.,* by having the application that created the enclave make the system call on behalf of the enclave. The enclave code must also take care to ensure that it does not inadvertently leak sensitive data outside the enclave, and that any sensitive data written outside the enclave is cryptographically-protected using keys stored within the enclave. Thus, while the SGX hardware offers powerful primitives, much of the responsibility of ensuring the confidentiality and integrity of enclave data falls on the application authors.

A number of techniques have been proposed in the literature to allow application authors build secure enclave applications. These techniques range from those that statically verify the absence of information leaks from enclave applications [139], programming-aids and libraries to allow enclave applications to be written easily with encryption of any egress data being handled by the library [140], and techniques that use programmer annotations on sensitive data structures to automatically split applications into enclave/non-enclave portions [85]. The focus of these techniques is to aid authors of enclave applications, writing *new* code tailored to use the features of the SGX.

The focus of this work is on frameworks that have been developed to allow *legacy* code to execute within enclaves. While several applications have been tailor-built for enclaves (*e.g.,* [131, 117]), this is a resource-intensive process, and application developers may wish to enjoy the benefits of the SGX without the upfront investment needed to build enclave code from scratch. These frameworks provide the necessary in-enclave support to allow legacy code to operate within the constraints imposed by the enclave programming model, *e.g.,* inability to perform certain operations such as system calls within the enclave. These frameworks broadly follow three different models:

① ***Library OS model.*** In this model, an entire library OS executes within the enclave. To port an application within the enclave, the application developer simply loads the application binary, together with any libraries that it uses, and can execute the resulting binary within the enclave. As a result, these techniques are able to provide *binary compatibility*, *i.e.,* unmodified binaries (or only modified to link with the library OS) can execute within the enclave. Frameworks that implement this model can additionally implement protections against IAGO attacks [25] (*e.g.,* attacks against the enclave application implemented by adversarial system software by tampering with return values) by including a suitable shim layer within the library OS that checks return values. Examples of frameworks that implement this model include Haven [16], Graphene-SGX [155], and SGX-LKL [118].

② ***Library wrapper model.*** This model, implemented by Panoply [135], assumes that applications invoke system services via libraries such as the standard C library (`libc`). Normally, these libraries contain the low-level system calls and other sensitive instructions that cannot be executed within the enclave. Panoply provides library wrappers that enclave-based applications can link against. An application author can use Panoply by simply modifying the code that uses the standard C library to instead use Panoply wrappers, which in turn provides the necessary machinery to cross the enclave boundary.

③ ***Instruction wrapper model.*** In this model, wrappers are provided for the low-level instructions (such as `syscall`, `inb`, `outb`) that are not permitted within enclaves. The wrappers contain the machinery to cross the enclave boundary, and take care of data protection—they automatically encrypt all data leaving the enclave boundary and decrypt the ciphertext data received by the enclave.

On the surface, this model may appear conceptually similar to the library wrapper model; however, the key difference is the *level at which the wrappers are implemented*. Because applications rarely use the low-level instructions such as `syscall`, `inb`, `outb` that are forbidden for use within the enclave in their raw form, and instead rely on libraries to perform these calls on their behalf, only the libraries that use these calls need to be modified to use the wrappers. The application code that invokes these libraries remains unmodified. SCONE [13], lxcsgx [149], and Porpoise, which is described in this work, use this model.

We present these models in more detail in Section 3.3. Note that while the example frameworks discussed above have been developed with legacy applications in mind, their main goal is to implement the heavy-lifting needed to get applications to conform to the constraints imposed by enclave programming. The same frameworks can also be leveraged as supporting infrastructure by new enclave-based applications.

The primary contribution of this work is to evaluate the relative merits of the three methods above in porting legacy code to SGX enclaves from a software engineering perspective. The criteria on which we wish to evaluate the methods are:

- How much effort is required to port application code to enclaves in each of these methods? This criterion measures the amount of effort that it takes a software developer to deploy a first-cut of an application within the enclave.
- How much flexibility does each method offer the application developer in engineering the enclave? For example, suppose that the application developer decides to execute some code outside the enclave for performance reasons, how much effort does the developer need to invest to port the code in that way using each of the methods?

- How much trusted code (in addition to the application's own code) must execute within the enclave?
- What are the performance overheads imposed by each approach?

We discuss these questions in more detail in Section 3.5. To study these questions, we ported a number of popular applications (including OpenSSL, Memcached, a Python interpreter, and a Web server) to SGX enclaves using representative frameworks that implement each of the models described above: Graphene-SGX [155] representing the library OS model, Panoply [135] representing the library wrapper model, and Porpoise, representing the instruction wrapper model. We studied the benefits and costs of each of these methods to port legacy applications.

In addition to answering the above questions, which is the primary contribution of this work, we also consider the Porpoise prototype as a secondary contribution of this work.

3.2 Background on SGX

SGX enables confidentiality and integrity-protected execution of trusted code in untrusted software environments [57, 96]. The primary end-user-visible artifact in an SGX system is the concept of an *enclave*. An enclave is a linear region of a process's virtual address space, the contents of which are protected by SGX from even the most privileged software running on that hardware platform.

A process creates and initializes an enclave via a set of instructions exported by the SGX ISA. To create an enclave, the process provides a pointer to a binary executable and instructs the hardware to initialize the enclave with this binary (which contains the code and data with which the enclave must start executing). The hardware reserves a region of the virtual address space for the enclave, and loads up the enclave with this binary. The pages for this portion of the address space are drawn from a reserved region of physical memory (called the encrypted page cache). The hardware then obtains a measurement of the enclave (for attesting it to the entity that started the enclave) and seals the enclave so that any further modifications are not possible [11].

SGX introduces a new *enclave-mode* in which the hardware can execute when executing the code of the enclave. The SGX hardware ensures that the contents of the enclave are visible in the clear only when the processor is in enclave mode, and the control is within that enclave. When the processor is in kernel-mode or in user-mode, any code that attempts to access the enclave will be unable to access the cleartext contents of the enclave. It accomplishes this protection by encrypting the contents of the enclave with hardware-generated keys, and ensuring that decryption only happens when the there is a memory access from code executing within the

Instruction	Comment
`cpuid`, `getsec`, `rdpmc`, `sgdt`, `sidt`, `sldt`, `str`, `vmcall`, `vmfunc`	Might cause VM exit
`in`, `ins/insb/insw/insd`, `out`, `outs/outsb/outsw/outsd`	I/O fault may not safely recover. May require emulation.
Far call, Far jump, Far Ret, `int n/into`, `iret`, `lds/les/lfs/lgs/lss`, `mov` to `DS/ES/SS/FS/GS`, `pop` of `DS/ES/SS/FS/GS`, `syscall`, `sysenter` workhline `lar`, `verr`, `verw`	Accessing the segment register could change privilege level. Might provide access to kernel information.
`enclu[eenter]`, `enclu[eresume]`	Cannot enter an enclave from within an enclave.

Table 3.1: Set of instructions forbidden for use within the enclave. Adapted from the Intel SGX programming manual [64].
enclave.

The process enters enclave mode by executing an `EENTER` instruction exported by the SGX ISA. Once the processor is in enclave-mode, the enclave code can freely access data stored both within the enclave, as well as other user-space memory within the process's address space.

However, SGX places several restrictions on the instructions that can execute when the processor is in enclave mode. Table 3.1 lists the set of instructions that are forbidden in enclave mode. These instructions can either be executed when the process is in user-mode (*e.g.,* `syscall`, `enclu`), or by the privileged system software (*e.g.,* the OS or the hypervisor) on behalf of the process (*e.g.,* `encls`, modifications to the registers `DS/ES/SS/FS/GS`). Thus, applications written for execution within the enclave must not include these instructions.

The main goal of the enclave-execution frameworks discussed in this work is to enable to execution of legacy applications within the enclave. Legacy applications are not written with enclaves in mind, and may include many of these instructions, *e.g.,* instructions such as `syscall`, `sysenter` and `int` are routinely used within user-space applications to invoke kernel services. While these instructions are permitted for execution in the processor's user-mode, they are not permitted when the processor is in enclave-mode. The main goal of the enclave-execution frameworks is therefore to enable user-space applications that use these instructions to execute within the enclave by suitably wrapping the forbidden instructions and forwarding them for execution outside the enclave.

Once the enclave has completed execution, it exits using the `EEXIT` instruction, transferring control back to the user process that entered the enclave. Enclave exits can also happen asynchronously (called an `AEX` in SGX). In both cases, the hardware saves the state of the enclave, scrubs registers, and returns the processor to user-mode.

Threat Model. As is standard with SGX, we assume that the enclave contains sensitive code and data that must be protected from adversaries. SGX admits a powerful adversary model

in which even the code of the user-space process that launches the enclave and the privileged system software (*e.g.,* OS or hypervisor) are untrusted. SGX protects against these adversaries by encrypting the enclave contents with hardware-managed keys, and decrypting the contents only when the access is from within the enclave. Decryption is done within the cache-hierarchy to prevent cold-boot and bus-snooping attacks on the contents of the enclave.

The attacker can attempt to attack the enclave in a variety of ways to compromise confidentiality and integrity. SGX provides confidentiality and integrity protection against such adversaries using standard cryptographic techniques (albeit implemented in hardware). The adversary can also attempt to subvert the execution of the enclave by feeding it malformed input, *e.g.,* to exploit a memory error in the enclave code itself, or by suitably modifying return values when enclave code interacts with non-enclave code. For example, in IAGO-like attacks [25], the attacker (OS) convinces the victim (enclave application) to act against itself. When the enclave application requests random numbers, e.g., reading `/dev/random` file, the operating system may return a fixed pattern, e.g., `0x00000000`, to mislead cryptographic libraries into generating weak keys. Similarly, the operating system may return incorrect or fixed system time or process ID to alter the enclave execution. Such attacks are a realistic threat, especially the enclave contains a lot of low-level trusted code. Systems such as Haven [16] attempt to protect against some such attacks (in particular, IAGO attacks) by implementing a shim layer that checks the values that cross the enclave boundary. Nevertheless, it is important to minimize the amount of trusted code running within the enclave. Indeed, this is one of the metrics on which we evaluate the various enclave-execution frameworks that we consider.

For the purposes of this work, however, we do not consider within our threat model recent work on hardware-based side channels to subvert SGX (*e.g.,* ForeShadow [160]). While these attacks constitute a serious threat to SGX, we consider them out of scope for this work, whose main goal is to evaluate the merits and costs of various approaches to enclave code development.

3.3 Enclave-execution Models

In this section, we present the technical details of the three models that have been proposed to date in the research community to support enclave-based applications. All the three models referenced in this section are illustrated in Figure 3.1. For each model, we also qualitatively discuss the benefits and costs of each model.

3.3.1 Library OS Model

The library OS model for enclave execution was pioneered by Haven [16], and has been followed by other open-source prototypes such as Graphene-SGX [155] and LKL-SGX [118]. In

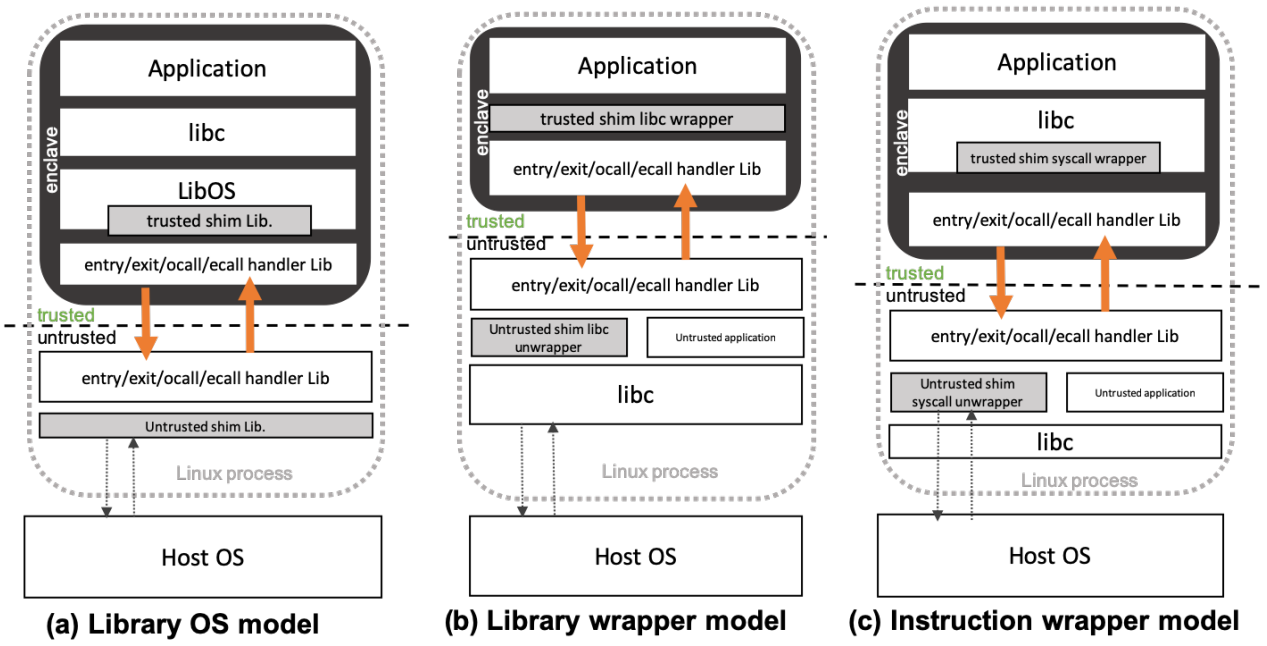


Figure 3.1: Models to support enclave-based applications.

this model, the enclave consists of the application to be protected and a library OS (*e.g.,* Drawbridge [116], Graphene [157], or the Linux kernel library (LKL) [119]).

A library OS implements the abstractions that a traditional OS exposes to applications, but does so completely within user space. Because the library OS executes in user-space, it cannot execute privileged operations that can only be executed in the processor's supervisor mode (*e.g.,* operations related to protection and isolation, such as switching page tables upon a context switch). Thus, the library OS interfaces with a small privileged software layer that implements these privileged operations. This interface between the library OS and the privileged software layer is typically narrower than the system call layer exposed by the OS to applications, *e.g.,* 38 distinct operations in the interface of Graphene-SGX, 24 in Haven, and 7 in LKL-SGX [118]. The library OS includes wrappers that redirects control to the library OS handlers before it reaches any of the instructions that cannot be executed in enclave mode (Table 3.1).

From the perspective of an application developer, the experience of running an application with this model works as follows. The application developer specifies the binary executable that must be executed within the enclave. This binary executable need not be statically linked with the libraries that it uses. The application developer specifies the list of all libraries that the application may potentially use (together with their code). The enclave is initialized with the code of the library OS, the code of the application, as well as all libraries that the application may use. Because the library OS contains all the supporting code needed by the application, it

can link dynamically against any libraries it uses (that are already pre-loaded into the enclave). The dynamic linker, which also runs inside the library OS, patches up the appropriate symbol tables at runtime.

From an end-user's standpoint, the execution of the application proceeds in a manner very similar to executing the application on a traditional desktop (*i.e.,* without enclaves). It is important to note that no new code is loaded into the enclave after initialization (SGX's attestation model disallows this) even though the end-user gets the illusion of dynamic linking. This is because all the code is loaded into the enclave prior to attestation, and the dynamic linker simply patches up symbol tables as the application references the libraries that it needs at runtime.

Benefits of the model. We now list the benefits of this model:

• *Binary compatibility.* This approach offers binary compatibility. The application binary interface is unchanged, so this approach can simply take any standard binary application (*e.g.,* one that is POSIX compatible), and execute it in the enclave "out of the box." Binaries need not even be statically linked, because the approach simulates dynamic linking; the application developer need only specify and initialize the enclave with the list of all libraries that the application may potentially use.

Note that library OS-based prototypes also rely on wrapping low-level instructions (*i.e.,* those forbidden in enclave execution) that appear within library code. They also rely on the observation that applications typically invoke the functionality implemented by these low-level instructions via library calls, which themselves contain these low-level instructions. Thus, an application developer needs to include enclave-compatible libraries when initiating the enclave.

• *Fewer domain crossings.* Because the library OS runs within the enclave, several operations that traditionally execute within the OS can be fulfilled by the library OS itself. For example, system calls such as `getpwd`, `fcntl`, `dup` and `brk` require a user/kernel domain crossing on a traditional system. This is because these operations modify OS kernel data structures that are shared among multiple applications. In this case, the library OS executes together with the application that it serves, and this is the only application that it has to serve. Thus, the operations represented by these system calls can simply be implemented as modifications to data structures within the library OS, *e.g.,* to an in-enclave file system. These calls do not modify security-sensitive state of any other applications, and thus do not need to be executed by privileged system software.

For some system calls, it is important *not* to cross over from the enclave. For example, applications typically rely on the OS to provide a source of randomness. Since the OS is

untrusted, the application can no longer simply trust the results of a `getrandom` system call executed by the OS; the OS could simply cheat by providing a poor set of random values, weakening any cryptographic keys that the application may then generate using these random values. Instead, the underlying enclave execution framework must leverage other mechanisms to obtain randomness, *e.g.,* the `rdrand` x86-64 instruction, which sources randomness from the hardware.

• *Simple enclave interface.* On the SGX, all code running outside the enclave is untrusted. This includes the user-space application within which the enclave is initiated, as well as the privileged system software layer. As a result, enclaves must typically guard against IAGO-like attacks [25], in which the untrusted code attempts to compromise the security and privacy of enclave code by passing malicious return values to calls that cross the enclave boundary.

In the library OS model, the interface to the enclave is conceptually simpler than even the system call interface on traditional OSes. For example, as discussed previously, among the three library OS execution models discussed in the literature, Graphene-SGX has the widest interface, and even that interface consists of only 38 interfaces, as opposed to a few hundred in a typical system call interface. As a result, on systems that use the library OS model, it is also easier to design shim layers to protect against IAGO-like attacks.

Costs of the model. The main costs of the library OS model arise from the large TCB size that executes within the enclave, and the relative inflexibility available to the application developer to restructure the application:

• *Large TCB size.* The entire library OS runs within the enclave, amounting to code that is a few hundreds of thousands of lines within the TCB (we provide concrete numbers in Section 3.5). As a result, an attacker who targets the enclave-based application has a larger attack surface to work with, and any vulnerabilities within the library OS or other supporting code become a liability for the enclave.

• *Low flexibility.* As proposed and illustrated in the research literature on library OS prototypes, entire applications execute within the enclave. This offers little flexibility to a potential application developer who wishes to execute only part of the application within the enclave. For example, consider an in-enclave Python interpreter—the developer may wish to execute only the core interpreter loop within the enclave, leaving all the other functionality of the interpreter outside the enclave. This may either be for performance reasons, or to reduce the amount of trusted code in the application.

In the library OS model, re-engineering application code is cumbersome at best. The portion of the application code that executes will need to communicate with the rest of the application,

but this will involve communicating across the layers of the library OS. This model also complicates the case where two enclave-based applications need to interact with each other. For example, suppose that a Python interpreter, executing within an enclave, needs to communicate with a key-value store, also executing within an enclave. The two are mutually-distrusting, and therefore cannot execute as applications within the same library OS (and hence the same enclave). Thus, they execute in two different enclaves, and each message in a cross-enclave communication must go through the library OS layers in both enclaves. Graphene-SGX implements this idea in the notion of *enclave groups*, which are a group of mutually-distrusting, yet interacting applications, that may have been derived from a common parent process using a `fork`.

3.3.2 Library Wrapper Model

To our knowledge, Panoply [135] is the only system that implements the library wrapper model. Driven by the goal of minimizing the amount of trusted code executing in the system, Panoply implements library wrappers for in-enclave applications. For example, it implements wrappers for the standard C library. Applications are compiled/linked against the library wrappers provided by Panoply. The library wrappers implement the task of marshalling data and passing it to the library. In Panoply, the library itself executes outside the enclave, and is untrusted.

Benefits of the model. As discussed, the primary benefits of the model are that the enclave code is extremely lightweight.

• *Small TCB size.* Among the three models discussed in this work, Panoply offers the smallest TCB size. However, this comes at the cost of running the library outside the enclave.

• *Flexibility to application developer.* The application developer has the flexibility to decide which portion of the application executes within the enclave. The task of splitting an application is as simple as executing all the enclave code as a separate module, and making a cross-module call to a function within that code. The enclave code itself is easy to produce, by simply linking the module against the Panoply library.

Costs of the model. There are three key costs associated with the library wrapper model:

• *Securing library execution.* In Panoply, the code of all supporting libraries used by the application executes outside the enclave. This code is untrusted, and can be used by the attacker to subvert the enclave application, *e.g.,* via IAGO-like attacks. Because the library wrapper is much larger than the library OS interface, it becomes more challenging to defend against such attacks.

• *Application-level modifications.* The Panoply prototype requires applications to be modified

to leverage its library wrappers. All calls to the library within the application must be modified by calls to the Panoply library instead. The Panoply authors report modifications on the order of about a 1000 lines for the benchmarks that they report in their paper.

Conceptually, such changes do not require access to source code, as long as the code is compiled to be position-independent (which SGX requires). It is possible to implement a library wrapper prototype that can be incorporated with the enclave application at link time (or by binary rewriting of the application). Our focus in this work is on the Panoply prototype, which has a few limitations that necessitate source-level modifications. For example, Panoply does not support the `FILE` structure of the standard C library. Any application code that uses the `FILE` structure must be rewritten to use a placeholder variable of type `int`, which is then translated to the corresponding `FILE` structure in the library (outside the enclave) via a table lookup.

Although modifying applications does impose a burden on a software developer wishing to quickly prototype an enclave-protected version of the application, the resulting engineering effort may sometimes be used to improve application performance as well. For example, we observed that the authors of Panoply had significantly modified the code of `OpenSSL` in the process of porting it to work within the enclave. For example, aside from the modifications to use Panoply library wrappers, the Panoply version of `OpenSSL` also replaces the random number generation code with calls to the Intel SDK's random number generator, which in turn calls the `rdrand` x86-64 instruction to obtain random numbers. This prevents a domain crossing, and is also more secure than relying on the underlying untrusted OS to provide random numbers.

• *Large and evolving interface.* The standard C library has an interface spanning several thousand functions. Moreover, this interface has also been changing as the library implementation evolves.

To understand the impact of this evolution, we studied the `glibc` repository and observed the number of API calls across ten versions of the library, and the number of APIs added or removed from each version. Table 3.2 presents our results. As this table shows, the library wrapper interface is over two orders of magnitude larger than the library OS enclave interface, and evolves significantly over the ten generations that we studied. Consequently, a library wrapper implementation such as Panoply must also be modified and tailored for each version of the library.

It is also important to note that standards such as POSIX and ISO only specify the library's function-call interface, but leave unspecified the definitions of data structures (*e.g.,* the `FILE` data structure). These data structures can change from one library version to another even if

#Version	# API size	# added/removed
2.20	2021	+1/-2
2.21	2023	+2/-0
2.22	2032	+9/-0
2.23	2043	+12/-1
2.24	2046	+3/-0
2.25	2058	+13/-1
2.26	2073	+32/-17
2.27	2110	+37/-0
2.28	2126	+18/-2
2.29	2128	+2/-0

Table 3.2: The evolution of the standard C library (glibc) interface across several versions. This table shows the number of API calls in each version, and the number of API calls added (+) or removed (-) from the prior version.

the interface remains POSIX or ISO-compliant. Changes to these data structures will require modifications to the library wrappers to suitably marshal/unmarshal the data as it crosses the enclave boundary.

3.3.3 Instruction Wrapper Model

The instruction wrapper model works by providing wrappers for instructions that are forbidden for use within the enclave, *i.e.,* the instructions in Table 3.1. The wrappers perform marshaling of arguments, and forward them to supporting code outside the enclave, which executes the instructions on behalf of the enclave. In contrast to the library wrapper model, the marshaling happens at a much lower level of abstraction, *i.e.,* at the level of registers and memory.

In theory, this approach can be made to work on arbitrary binaries by replacing all occurrences of the instruction in the enclave code with the wrapper. However, practical implementations of the instruction wrapper model, including SCONE [13] and Porpoise make the observation that applications rarely use these instructions in their raw form. Rather, the applications are programmed to use libraries, which in turn execute these low-level instructions on their behalf. Thus, they wrap the occurrences of these instructions within the library. Applications simply link against these libraries to leverage the instruction wrappers.

Benefits of the model. This approach is conceptually similar to the library wrapper model and shares some of its benefits. The main difference between this models and the Panoply prototype is that the standard C library executes within the enclave. As a result these approaches have a somewhat larger TCB than Panoply.

The primary benefit of this model over Panoply is that it is that by implementing wrappers at a much lower level (*i.e.,* wrapping instructions rather than providing library call wrappers), it works on a much slower changing interface. To take an example, we consider the `syscall`

#version	# system calls	# system calls added
v4.15	333	0
v4.16	333	0
v4.17	333	0
v4.18	335	2
v4.19	335	0
v4.20	335	0
v5.0	335	0
v5.1	339	4
v5.2	345	6

Table 3.3: Evolution of the system call interface across versions of the Linux kernel

instruction, which is used to implement system calls. Naturally, the wrappers for the `syscall` instruction depend on which system call is being invoked (*e.g.,* because the number of arguments to each system call are different). As Table 3.3 shows, the system call interface on Linux is both much narrower and much stabler over kernel versions as compared to the glibc interface (which was shown in Table 3.2).

The instruction wrapper model executes libraries within the enclave. Thus, unlike with Panoply, libraries can be trusted. However, this model must still implement a shim to protect against IAGO-like attacks on the (much narrower) instruction return interface. This model does not require invasive application-level modifications for rapidly creating an in-enclave prototype. However, application-level modifications may be necessary to optimize its performance, and as our evaluation shows, the instruction-wrapper model also lends itself well to any future re-engineering of the application.

Costs of the model. The primary costs of the model are that it has a somewhat larger TCB than Panoply (although a much smaller TCB than the library OS model). Applications also need to be re-linked to use libraries in which the low-level instructions forbidden within SGX enclaves are wrapped.

3.4 Porpoise: An Instruction Wrapper Prototype

Our goal in this work is to quantitatively evaluate the benefits and costs of the library OS, library wrapping and instruction wrapping approaches to port code to enclaves. While we could find open-source prototypes representing the library OS and library wrapping approaches, the source code for SCONE [13], which implements instruction wrapping, was not available (SCONE is the basis for a commercial offering from `scontain.com`). We therefore built our in-house prototype, called Porpoise, for our evaluation. In this section, we describe, in brief, the implementation of Porpoise and some key challenges we had to overcome in its implementation.

Porpoise is implemented as a set of modifications to `musl-libc` [106] version 1.1.9. As

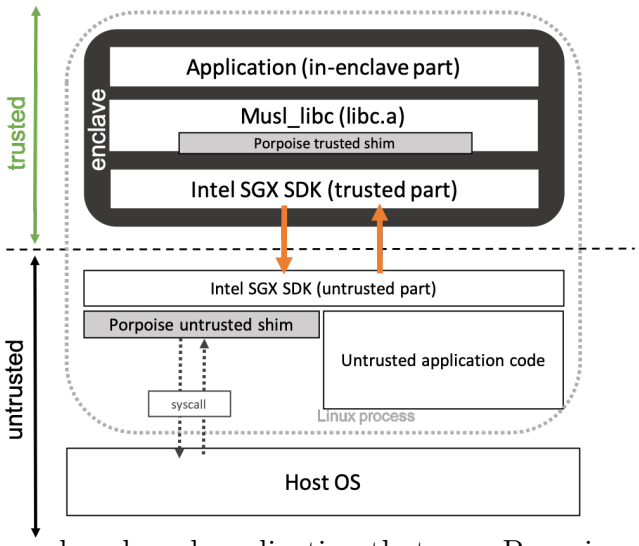


Figure 3.2: Design of an enclave-based application that uses Porpoise. Porpoise consists of the trusted in-enclave shim, and an untrusted shim outside the enclave (shown in gray).

discussed in Section 3.3.3, we make the assumption that applications do not directly invoke the raw low-level instructions, but rather rely on libraries for doing so. `musl-libc` is an API-compatible implementation of the standard C library that is much more modular and easier to modify than its counterparts such as glibc.

Figure 3.2 depicts an enclave that uses Porpoise to support an enclave application. Porpoise relies on the Intel SGX SDK (version 2.7.1) [66] for various standard tasks such as enclave initialization, `ecalls` (calls to the enclave from outside), `ocalls` (calls from the enclave to the outside), maintaining the thread-control structures, the state save area and other structures that are part of the SGX's hardware/software interface. The hardware uses these structures to store the state of registers when it exits the enclave, so as to protect them from untrusted code outside the enclave. The Intel SGX SDK also has some untrusted support code outside the enclave that the untrusted code in the process uses to interact with the enclave. The core functionality of Porpoise is implemented in two shim layers: a *trusted shim* that runs within the enclave, and an *untrusted shim* that facilitates the interaction of the enclave with the rest of the user process.

The trusted shim layer is implemented as a set of wrappers around the evaluation of `syscall` instructions in `musl-libc`. The shim is responsible for marshaling data outside the enclave. It creates a copy of the system call's arguments to buffers in the untrusted shim layer, and transfer control to the untrusted shim layer. Correspondingly, the untrusted shim unmarshals the data outside the enclave, and performs the system call on behalf of the enclave. Once the system call returns, the untrusted shim returns control to the enclave with any return values from the system call stored in buffers outside the enclave. The trusted shim copies data back from the

untrusted shim layer, and unmarshals the data for consumption by the enclave. Logically, our shim is structured as wrapper around occurrences of the `syscall` instruction in `musl-libc`, with a case analysis based on the system call number, to determine the number of arguments to be copied.

Because the trusted shim is the portion of the enclave that interacts with the untrusted world, it is also logically the place where filters that detect IAGO-style attacks can be implemented. Our Porpoise prototype currently only has wrappers for IAGO-style attacks largely similar to the file-system shield and network shield described in the SCONE paper [13].

Porpoise's trusted shim provides encryption by default for data that exits the enclave. Each piece of data that is not required for executing the system call is encrypted with keys managed by the trusted shim. However, we cannot encrypt all the arguments to the `syscall` instruction, *e.g.,* the system call number itself cannot be encrypted. Similarly, an enclave that interacts with the file system outside must be able to name the file, which must be sent in the clear (although the bytes sent to the file can be encrypted). For each system call, Porpoise's trusted shim encrypts the arguments that are not needed for the execution of the system call outside the enclave (*e.g.,* data blocks are not modified by the system call, and are therefore sent encrypted.

We note here that a number of papers have proposed oblivious file systems for enclaves [100, 6] that even hide the name of the file from the untrusted code. This is required to prevent the untrusted code from making inferences about the file accesses made from within the enclave. We do not consider these techniques to be within the scope of Porpoise for the purposes of the present work, which is to evaluate the merits and costs of different ways of porting code to the enclave. However, Porpoise is extensible, and such algorithms can be incorporated within Porpoise as well.

We structured our implementation of the trusted shim by creating a simple `send`/`recv` interface (as was also discussed in prior work [140]). Each argument of the system call is wrapped with a `send_user` call, whose API is as follows:

```
send_user  (void *enclbuf, void *userbuf,
                ssize_t size, int prot)
```

The first two arguments point to the in-enclave source buffer containing the data and the buffer within the untrusted shim to receive the data, respectively, `size` denotes the amount of data to be copied, and `prot` determines whether the data in the buffer should be encrypted on its way out. We use AES in CTR mode with 128 bit keys for encryption. A corresponding `recv_user` call obtains data on the return path.

Porpoise incorporates support for 145 system calls (of the total of 325 system calls in Linux-4.4.0-169). While we have plans to add support for the remaining system calls, prior

work [158] indicates that system calls vary in terms of importance (based on their usage in real-world packages). Indeed, we did not encounter these system calls in any of the application benchmarks that we studied. We now discuss the technical challenges that Porpoise overcomes in the implementation of some system calls.

• *In-enclave threading.* Porpoise supports multi-threaded applications via the `pthreads` API. Although Intel SGX supports multiple threads of execution within the enclave, it does not allow thread creation from within the enclave. Rather, an application has to pre-declare the number of enclave threads that it would like to support, and the application creates this number of threads in untrusted code. Each of these threads can enter the enclave in fresh thread context, thereby creating the illusion of a multi-threaded enclave. Thus, each enclave thread will have an associated counterpart thread in the untrusted user-space process.

Porpoise's `pthread`-compatible threading model has to work within the constraints of SGX. The `pthread` library uses the `clone` and `arch_prctl` system calls to create new threads. The `clone` system call is used to create a thread, while the `arch_prctl` is used to modify the `%fs` and `%gs` registers, which point to structures that store the thread state. For instance, each thread has its thread-local structure (that we will call `thread_data`) that stores a pointer to the base of the thread's stack, the thread ID, signal mask, canaries, and so on. The `%fs` register is used to point to this structure of the currently executing thread. Both the `%fs` and `%gs` registers can only be modified when the processor is in supervisor mode, and they cannot be modified in user- or enclave-mode. They can be read irrespective of processor mode.

Within a traditional process, `pthread` uses the `arch_prctl` system call to set the `%fs` register to point to the `thread_data` of the thread that is currently executing. This `thread_data` structure resides in a memory location within the process's address space. The key difficulty with wrapping `arch_prctl` is that if the wrapper simply performs the equivalent operation outside the enclave within the user process, the resulting call will set the `%fs` register within the user-space. The kernel cannot access enclave memory, and therefore cannot change the pointer to the thread's state inside the enclave.

We address this problem as follows. As discussed earlier, the Intel SGX does not allow threads to be created within the enclave itself. Rather, the application must pre-declare the number of enclave threads it intends to use, and the enclave is initialized accordingly. Internally, the Intel SGX SDK maintains an in-enclave *thread control structure* for each thread . This is akin to the `thread_data` structure for traditional user-space threads, but is required for in-enclave bookkeeping of the thread.

Porpoise maintains a table that associates the in-enclave thread-control structure for each

Application	Description
`bzip2`-1.0.6	File compression utility
`memcached`-1.5.20	Key-value store
`openssl`-1.0.1m	OpenSSL cryptography library
`h2o`-2.0.0	HTTP Web server
`cpython`-3.7	Python interpreter in C

Table 3.4: Applications used in our evaluation.

enclave thread with the corresponding `thread_data` structure of that enclave thread's user-space counterpart. As the wrapped `arch_prctl` call updates the pointer to the `thread_data` structure in the process (by modifying the `%fs` register), Porpoise identifies the corresponding in-enclave thread to which this `%fs` corresponds, and updates the thread-control structure to resume executing that thread during enclave entry (without exiting the enclave).

A related problem happens with other data structures of the `pthread` library, where the kernel directly modifies data structures. For example, the kernel modifies the `detach_state` data structure in the `pthread` library to denote the current state of a thread (*e.g.,* `EXITED`, `JOINABLE`, ...). On an enclave-based system, this data structure cannot be stored within the enclave, because it will not be accessible to the kernel. Porpoise addresses this problem by maintaining two copies of the data structure: one within the enclave, and one in the user-space process. As the kernel modifies the data structure within the process, Porpoise modifies the corresponding copy within the enclave.

• ***`brk`*** *and dynamic memory allocation.* The current version of Intel SGX does not allow dynamic memory allocation within enclaves, although this has been proposed for future versions of SGX [165]. Instead, the Intel SGX SDK implements functionality that simulates the effect of dynamic memory allocation. It pre-allocates a certain amount of memory for use by the enclave, and maintains an internal break point to denote the top of the heap. This break point is modified by a "`malloc`" call that simply returns memory by modifying this break point. We redirect `brk` system calls to this implementation to offer the illusion of dynamic memory allocation for legacy applications.

3.5 Evaluation

We now quantitatively evaluate the costs and benefits of the three models discussed in the previous section. Our methodology is as follows: We consider one concrete prototype as a representative of each model—Graphene-SGX for the library OS model, Panoply for the library wrapper model, and Porpoise for the instruction wrapper model, and port a suite of applications (see Table 3.4) to enclaves using each of these models (not all applications work in all settings, as we will see) to answer the following research questions:

Application	Graphene-SGX	Panoply	Porpoise
`Bzip2`	✔	✔	✔
`Memcached`	✔	✖	✔
`OpenSSL`	✔	✔	✔
`H2O`	✔	✔	✔
`Cpython`	✔	✖	✔

Table 3.5: Evaluating the ability to port applications to enclaves using each framework (RQ1).

(RQ1) Porting effort. From an end-user's point of view, what is the effort required to port the application and get it running within the enclave?

(RQ2) Application re-engineering effort. Suppose that an application developer wishes to re-engineer the application by deciding that he only wants to run a portion of the application within the enclave. After the application developer has decided what code to run within the enclave, what is the amount that he needs to invest to get the code running within the enclave?

(RQ3) TCB size. How much trusted code runs within the enclave, in addition to the application's own enclave code?

(RQ4) Runtime performance. What is the runtime performance overhead of each of these approaches, and how do they compare to native execution (*i.e.,* executing the code without enclaves)?

3.5.1 RQ1: Porting Effort

To answer RQ1, we attempted to port the applications from Table 3.4 using each of the three methods. For this research question, we ported the *entire* application to run within the enclave. The application that starts the enclave is simply a dummy `main` function, that starts the application, but then has no further role to play in the execution of the application. Other than this `main` function, only the untrusted portion of the Intel SGX SDK, and any other code required by the framework (*e.g.,* the untrusted shim of Porpoise) run in user-space. Table 3.5 presents the results of our experiments.

Of the three methods, Graphene-SGX provides the smoothest porting experience. We simply wrote a manifest file that describes (among other things) the set of libraries used by the benchmark, and Graphene-SGX is able to execute the applications.

The effort to port these applications to run with Porpoise is comparable to that of Graphene. We only had to compile the application with Porpoise's version of `musl-libc`, and build it as a statically-linked, position-independent binary. We could port all five applications successfully to Porpoise.

Panoply is the most cumbersome of the three approaches to which to port applications. In

fact, the authors of Panoply themselves ported `OpenSSL` (version 1.0.1) and `H2O` (version 2.0.0), and reported having to modify 307 SLOC and 154 SLOC in these applications, respectively. In our experiments, we used the same code-base for `OpenSSL` and `H2O` as provided by the authors of Panoply. As a result, we chose `OpenSSL` version 1.0.1 and `H2O` version 2.0.0 for our experiments with Graphene-SGX and Porpoise (so that we could compare them across the same versions for our research questions), even though both Graphene-SGX and Porpoise can execute the latest versions of both `OpenSSL` and `H2O`.

To understand the complexity of porting an application afresh to Panoply, we attempted to port `Bzip2` from scratch. We found that `Bzip2` uses 10 API calls from the standard C library that were not wrapped in the Panoply prototype available to us (Panoply currently has about 250 wrappers implemented); we therefore implemented these 10 wrappers. However, we were still unable to successfully run `Bzip2`. Further investigation revealed that `Bzip2` uses the `FILE` structure in its code. The `FILE` structure includes a pointer. Panoply does not currently support such structures that have pointers in them as part of its library wrapper interface. Instead, it maps each `FILE` structure to an index (of type `int`), and uses the index as part of the library wrapper. The Panoply code outside the enclave uses the index to look up the `FILE` structure, now maintained outside the enclave, and performs the operation. We therefore had to modify `Bzip2` to replace all occurrences of the `FILE` structure with an index instead. In all, this and other changes required 149 SLOC of modifications to `Bzip2`. Given this rather cumbersome and time-consuming experience attempting to port `Bzip2` to Panoply, and the number of invasive changes needed to its source code, we did not attempt to port `Cpython` and `Memcached` to Panoply, because they use many more interfaces from the standard C library.

Summary (RQ1): The library OS and instruction-wrapping approaches provide a seamless enclave-porting experience. The library-wrapping approach, as implemented by Panoply, requires modifying a few hundred lines of code within the application.

3.5.2 RQ2: Application Re-engineering Effort

While RQ1 concerned the effort to port an application in its entirety into the enclave, RQ2 concerns the effort that an application developer would invest to port an application after deciding to re-engineer it. For example, an application developer may decide that it is not necessary to execute the entire application inside the enclave, and that it suffices to execute just certain security-critical parts of it within the enclave. RQ2 asks the following question: how different is the experience in building enclave code with each of these frameworks after such application re-engineering?

Application	#Interfaces	SLOC added
`Bzip2`	3	29
`OpenSSL`	1	8
`Cpython`	24	277

Table 3.6: Number of new interfaces required and code added for application re-engineering with Porpoise (RQ2)

· The research literature does contain examples of tools that assist with such porting, notably Glamdring [85] and the gcc-based tool described by the authors of lxcsgx [149]. These tools assist the programmer with the core task of re-engineering the application. Given a specification of sensitive data that must be protected (*e.g.,* in the form of annotations), and hence execute within the enclave, these techniques use static taint analysis to identify the other dependent code that must also execute within the enclave. Note that these tools assume the existence of an enclave execution framework.

Our goal in RQ2 is *not* to identify the sensitive data that must execute within the enclave or assess the difficulty of splitting the application. Rather, assuming that a suitable split has been identified, we wish to determine how much effort it is to re-engineer the application after such a split has been identified. We assume that the split is identified at the function-level of granularity, *i.e.,* certain functions have been identified to execute within the enclave, while the rest execute within the untrusted code. Because we manually analyzed the applications to identify this split for RQ2, we restricted ourselves to three of the application benchmarks, *viz.,* `Bzip2`, `OpenSSL`, and `Cpython`, as described below.

• `Bzip2`. We split `Bzip2` so that only the main file compression algorithm executes inside the enclave. This results in an enclave interface of three functions, `BZ_compressInit`, `BZ2_compress` and `BZ2_compressEnd`, with which the the re-engineered `Bzip2` application interacts with the compression algorithm

• `OpenSSL`. We moved the functionality that generates RSA keys to the enclave. The enclave interface to do so consists of just one function, (`genrsa_main`).

• `Cpython`. The `Cpython` application consists of code to parse, compile and then interpret an input python program. In real-world settings, the interpreter is responsible for running the python code on sensitive data. Therefore, we decided to port only the interpreter to the enclave. The interpreter's enclave interface has 24 functions.

We only re-engineer our benchmarks to execute atop Porpoise and Panoply (only atop Porpoise for `cpython`), where the effort to build the enclave part of the code after splitting is comparable. This is because both Porpoise and Panoply have a similar enclave/non-enclave

```
filename: enclave.edl
public ecall_PyArena* PyArena_New(void);

filename: pythonrun.c
PyArena * ecall_PyArena_New(void);
#define PyArena_New ecall_PyArena_New

filename: function_wrapper.c
PyArena* ecall_PyArena_New(void){
    PyArena *ret = NULL;
    sgx_status_t status = SGX_SUCCESS;
    status = ecall_PyArena_New(enclave_id, &ret);
    // ret will point to a buffer that stores
    // the return value from the enclave
    assert(status == SGX_SUCCESS);
    return ret;
}
```

Figure 3.3: Example of new code required to introduce an enclave interface in `Cpython` with Porpoise (RQ2).

interaction interface.

Figure 3.3 shows the new code that we write to create a new enclave interface in Porpoise for the `Cpython` application. As this code shows, an application that wishes to invoke a function in the enclave interface must simply perform an `ecall` to that function together with its arguments, and the enclave ID. A new entry for this interface is also included as part of the enclave's interface definition file. Table 3.6 shows the number of lines of such code that we had to write in total to create enclaves for each of the re-engineered applications.

We did not attempt to re-engineer our benchmarks to run atop Graphene-SGX. This is because Graphene-SGX was originally designed to run applications in their entirety within the enclave. This is reflected in the design of their enclave interface, which is a low-level interface that communicates with a platform-specific adaptation layer (called the Graphene-PAL). By design, the Graphene-PAL invokes a fixed entrypoint inside the enclave, typically the equivalent of the `_start` function in a traditional application. As a result, re-engineering an application to work atop Graphene-SGX, with part of its code running in the enclave, would require invasive changes to the Graphene-SGX platform itself—an activity that we did not wish to undertake, since our goal is to understand the platforms as-is.

It would be possible to re-engineer an application atop Graphene-SGX, so that the sensitive portion runs in its own process (with its own enclave), and interacts over IPC with the remaining parts of the application. However, this would require a fundamental rewrite of the application to make it a distributed client/server system. We view this change as being rather invasive to

Graphene		Panoply		Porpoise	
Component	**SLOC**	**Component**	**SLOC**	**Component**	**SLOC**
Trusted Code					
LibOS	31,742	Panoply shim	14,506	Porpoise shim	1,934
`glibc`-2.27	1,222,912	-		`musl`-1.1.9	82,978
-		Intel SDK	119,545	Intel SDK	119,545
Untrusted Code					
Graphene PAL	40,493	Panoply shim	3,004	Porpoise shim	1,209

Table 3.7: Amount of trusted (and untrusted) code that executes within each of the frameworks (RQ3).

the application's code base, and therefore do not evaluate this method.

Summary (RQ2): The effort required to re-engineer applications with the library-wrapping and instruction-wrapping models is similar. The enclave interface exposed by library OSes does not facilitate easy re-engineering of the application into an enclave and non-enclave portion.

3.5.3 RQ3: TCB Size

We evaluated the amount of trusted code that must execute within the enclave for each of the three frameworks. For RQ3, we did not consider the trusted code of the application itself (*i.e.,* its enclave code), because that number would depend on the application itself, and how the developer has decided to engineer the enclave. Rather, we only consider the code that is core to the framework itself. In addition, both Panoply and Porpoise use the Intel SGX SDK to bootstrap basic enclave functionality (Graphene-SGX does not), and this code is therefore part of their trusted code base. SCONE [13], an instruction-wrapping framework, also does not use the Intel SGX SDK within the enclave, relying instead on a home-grown library for basic enclave functionality. The source code for SCONE is not publicly available, but their paper reports a TCB of size of approximately 187,000 lines of code for the version of SCONE that implements shielding against IAGO-style attacks.

Table 3.7 presents the results of our evaluation. It shows the number of lines of trusted code that executes in each of these frameworks (measured using the `sloccount` utility). It also shows the amount of untrusted support code provided by the infrastructure (*i.e.,* the code that executes outside the enclave, and interacts with the enclave).

We see that Panoply emerges as the framework that reduces the amount of code that executes within the TCB. However, they do this at the cost of implementing wrappers at the library level of abstraction, which means that many more wrappers have to be written and

Application	Workload
`Bzip2`	zipping and unzipping files of various sizes
`Memcached`	memtier_benchmark [98]
`OpenSSL`	HMAC (md5), DES-CBC, AES-256-CBC, SHA-256, MD5, RSA-2048-sign and RSA-2048-verify from `OpenSSL` speed benchmark
`H2O`	wrk2 http workload generator [164]
`Cpython`	benchmarks from `pyperformance` using `timeit`

Table 3.8: Workloads used to run applications (RQ4).

that these wrappers have to evolve as the libraries evolve. Recall from Table 3.2 and Table 3.3 that the library API evolves much more than the relatively-stable system-call API. Thus, while Porpoise has more trusted code (in particular, the `musl-libc` library, which it modifies), its interface is more stable and requires fewer changes as the code evolves. Graphene-SGX requires the most trusted code, because it includes the library OS in the enclave.

Summary (RQ3): Library-wrappers, as implemented in Panoply, require the least amount of support code within the enclave. However, this code must also evolve to support changes to the library API. The instruction-wrapper approach requires more code within the enclave, but is likely to be stabler with respect to code evolution. The library OS approach requires the largest amount of trusted code within the enclave.

3.5.4 RQ4: Runtime Performance

We conducted experiments to understand the runtime performance implications of the three models. We studied both the overall performance impact on the applications that we ported, as well as microbenchmarks to stress the enclave/non-enclave interface.

We conducted all our experiments on an Intel(R) Core(TM) i7-7700 CPU (3.60GHz) with 4 cores and 2 threads per core (8 hyper-core) and an 8192KB cache and 16GB of RAM. We used Ubuntu 16.04 LTS (Linux 4.4.0-169) as the underlying OS for Graphene-SGX, Panoply and Porpoise. (Panoply works only atop Linux 4.4.0-169; we therefore used it for our evaluation. However, Porpoise works even on the latest version of the Linux kernel (5.3.8).) For experiments with our application benchmarks, we run the entire application within the enclave under each of the frameworks. However, we are unable to report performance numbers for each application on all frameworks, because we were unable to port all the applications to Panoply. Table 3.8 shows the workloads with which we ran the applications, while Table 3.9 reports the results of our experiments

Table 3.9 shows the performance of each of the applications, running the benchmarks from Table 3.8 on Graphene-SGX, Panoply (where applicable), and Porpoise. It also shows the native performance of the application, *i.e.,* when run outside the enclave. We find that across

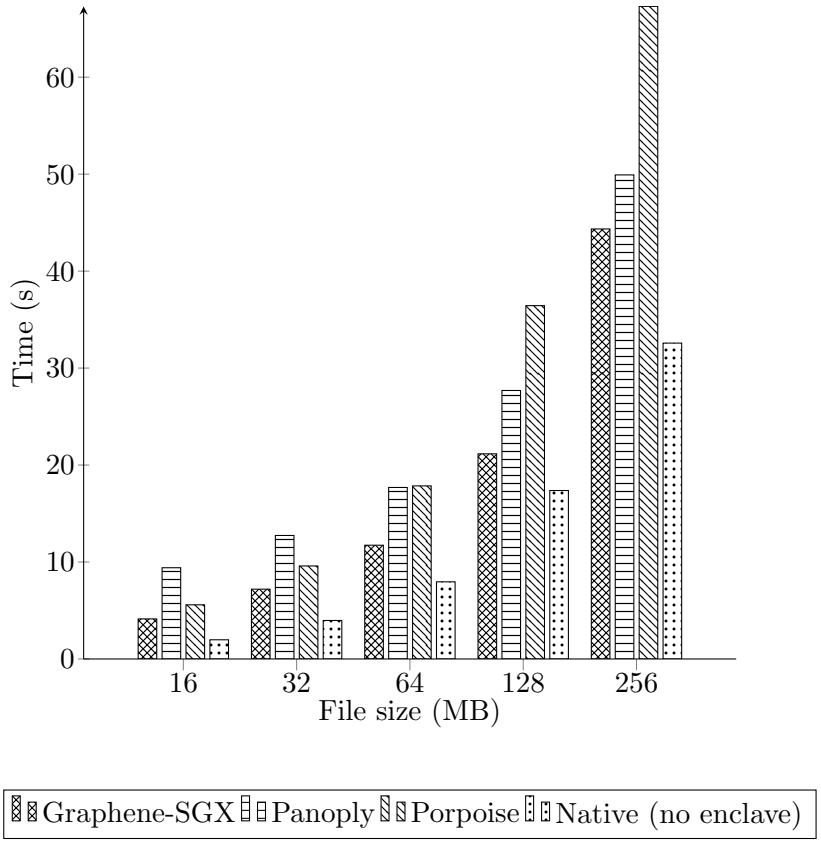


Figure 3.4: Time taken by Bzip2 to compress and decompress files of various file sizes with each framework.

the benchmarks, neither Graphene-SGX, Panoply nor Porpoise consistently outperforms the other. For example, on `Memcached` and `H2O`, Porpoise provides higher throughput and lower latency than Graphene-SGX. However, Graphene-SGX outperforms Porpoise on `Bzip2` and `Cpython`.

The `Bzip2` benchmark reads files in fixed-size chunks as it passes the file contents to the enclave. Thus, as the file size increases, the number of enclave/non-enclave interactions increases. However, Graphene-SGX, being a library OS implements file caching techniques, which fulfil some of the read requests, thereby ameliorating the number of domain crossings required. For the `Bzip2` benchmark, Panoply offers performance roughly comparable to Graphene-SGX, both outperforming Porpoise, especially as file sizes increase, Figure 3.4. This is because Porpoise executes the standard C library inside the enclave, in contrast to Panoply, which executes it outside the enclave. A number of library calls such as `fopen`, `read`, and so on, result in multiple domain crossings for Porpoise, and the number of such domain crossings increases with file size.

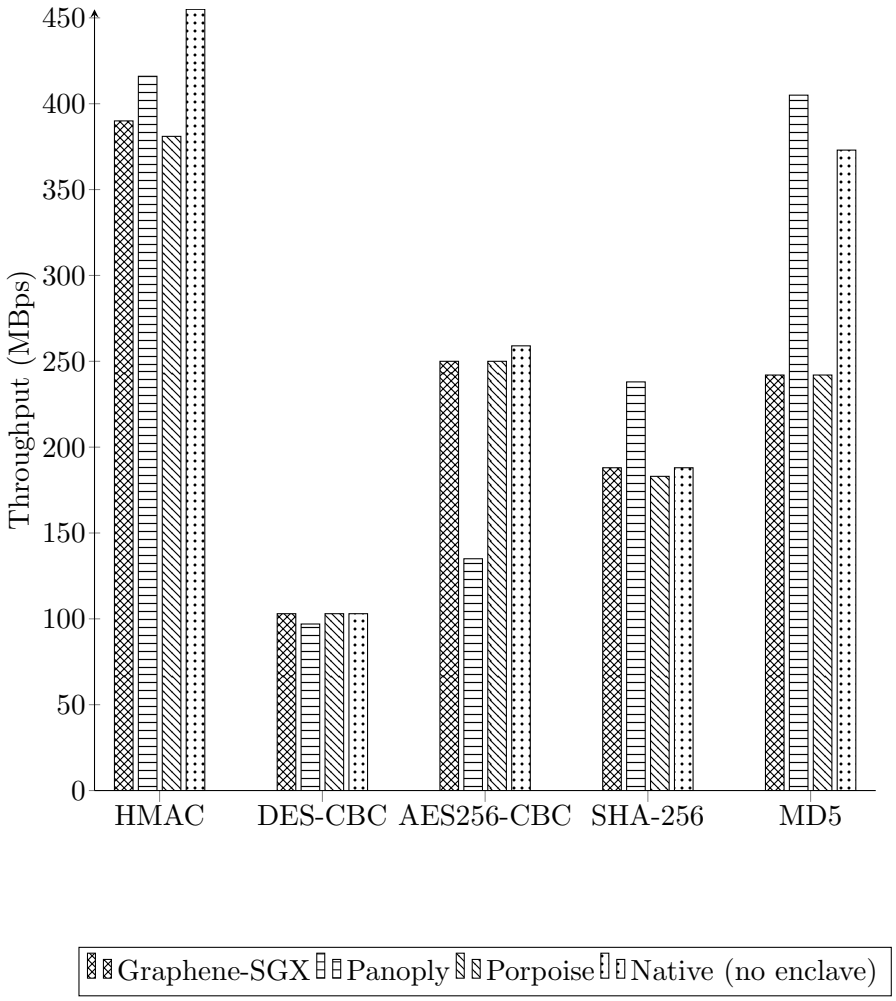
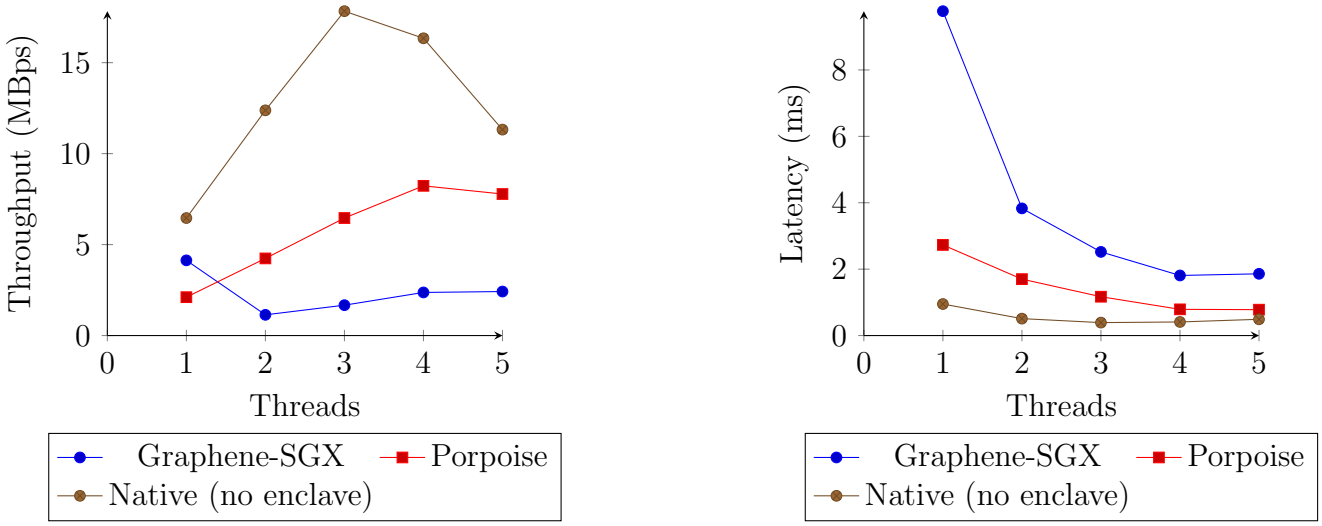


Figure 3.5: Throughput of popular cryptographic functions with different enclave porting frameworks.

In contrast, the number of domain crossings does not grow proportionally with the file size in the case of Panoply, thus explaining the performance difference.

For `OpenSSL`, we found that Panoply outperforms both Graphene-SGX and Porpoise, Figure 3.5. This is because the version of `OpenSSL` ported to Panoply makes extensive changes to optimize the performance. For example, as explained earlier this version replaces the random number generation code included in the `OpenSSL` release with calls to `sgx_read_rand`, a function provided by Intel SGX SDK, which uses the `rdrand` hardware instruction to source randomness.

Both `Memcached` and `H2O` run I/O intensive workloads, making them both network-bound. For both these benchmarks, we find that Graphene-SGX is significantly slower than Porpoise in terms of both throughput and latency. In case of `H2O`, the version running atop Graphene-SGX



(a) Throughput

(b) Latency

Figure 3.6: Performance of Memcached on Memtier workload running 4 threads, 50 connections, and 10,000 requests per client.

saturates at around 10,000 requests per second, with all requests exceeding that threshold being dropped by the Web server. The latency is also rather large, at 132ms per request. Both the versions running atop Panoply and the Porpoise are able to sustain a larger number of requests per second, however, the Panoply version saturates at around 33,000 requests per second, offering a latency of more than 1 second per request. The version running on Porpoise saturates above 40,000 requests per second. With `Memcached`, the version on Porpoise outperforms the version on Graphene-SGX even as we vary the number of server threads, as shown in Figure 3.6.

For the `Cpython` benchmark, both the Graphene-SGX and Porpoise have similar performance on the `pyperformance` benchmark as shown in Figure 3.7.

Evaluation with Microbenchmarks. To understand the raw overheads of enclave crossings imposed by each of the frameworks, we also evaluated them with microbenchmarks. Our microbenchmarks consist of enclave code that cause an enclave crossing by requesting the execution of a system call a million times. We considered a set of system calls shown in Table 3.10.

We find that Graphene-SGX offers the best performance for sequential `read` operations, comparing favourably even with native performance. This is because Graphene-SGX, being a library OS, implements several file caching techniques that avoid costly domain crossings. However, the benefit of these optimizations does not apply when the `read` operations seek to random locations in the file, as shown in Table 3.10. In this case, both Porpoise and

App	Workload	Graphene-SGX	Panoply	Porpoise	Native (no enclaves)
Bzip2	File size (MB)	Time (s)	Time (s)	Time (s)	Time (s)
	16	4.134	9.410	5.585	1.979
	32	7.212	12.738	9.592	3.971
	64	11.732	17.688	17.850	7.957
	128	21.157	27.690	36.438	17.373
	256	44.347	49.921	67.302	32.578
Memcached	Memtier Params	Throughput (MBps), Latency (ms)	Throughput (MBps), Latency (ms)	Throughput (MBps), Latency (ms)	Throughput (MBps), Latency (ms)
	memtier is configured to run with 4 threads, 50 connections/thread, 10,000 req/client	Threads / Throughput / Latency: 1 / 0.4 / 9.77; 2 / 1.15 / 3.83; 3 / 1.67 / 2.52; 4 / 2.37 / 1.81; 5 / 2.42 / 1.86	N/A	Threads / Throughput / Latency: 1 / 2.11 / 2.73; 2 / 4.24 / 1.70; 3 / 6.46 / 1.17; 4 / 8.23 / 0.79; 5 / 7.78 / 0.78	Threads / Throughput / Latency: 1 / 6.46 / 0.95; 2 / 12.38 / 0.51; 3 / 17.83 / 0.39; 4 / 16.34 / 0.41; 5 / 11.32 / 0.49
OpenSSL	Workload	Throughput (KBps)	Throughput (KBps)	Throughput (KBps)	Throughput (KBps)
	HMAC (md5)	390,626	416,091	381,300	455,747
	DES-CBC	103,768	97,886	103,723	103,883
	AES256-CBC	250,578	135,932	250,406	259,107
	SHA-256	188,996	238,508	183,482	186,413
	MD5	242,736	405,965	242,270	373,865
	RSA-2048-sign	303 ops/sec	1284 ops/sec	305 ops/sec	327 ops/sec
	RSA-2048-verify	14,540 ops/sec	37,678 ops/sec	14,562 ops/sec	14,893 ops/sec
H2O	Requests/sec	Latency (ms)	Latency (ms)	Latency (ms)	Latency (ms)
	10,000	132	1.61	1.28	1.29
	20,000	*	1.61	2.30	1.32
	30,000	*	2.65	1.93	1.36
	40,000	*	> 1 sec	18.21	1.77
	50,000	*	> 1 sec	> 1 sec	> 2.13
Cpython	pyperformance	Time (ms)	Time (ms)	Time (ms)	Time (ms)
	html5lib	260		269	106
	pyres	1940		1980	890
	json_load	0.348	N/A	1.030	0.171
	float	432		396	122
	fannkuch	1805		1806	566

Table 3.9: Comparing the performance of various application benchmarks running atop Graphene-SGX, Panoply and Porpoise (RQ4). `Memcached` and `Cpython` are not available atop Panoply (see Table 3.5).

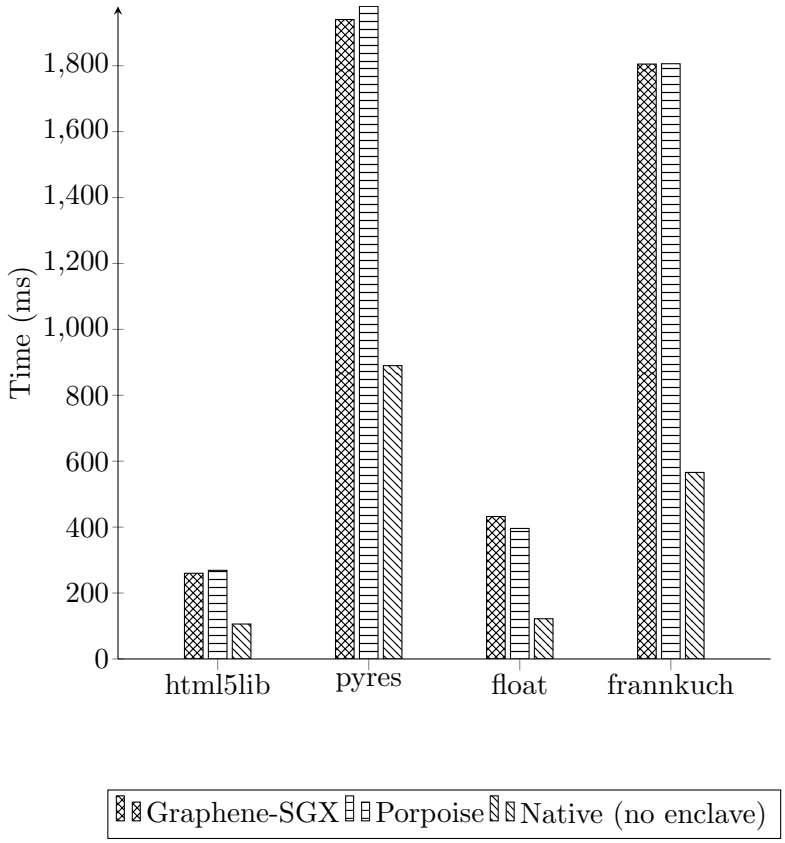


Figure 3.7: Execution time for `pyperformance` benchmarks.

syscall	Graphene-SGX	Panoply	Porpoise	Native
read (sequential)	0.405	3.133	4.295	0.209
write	13.268	3.471	4.742	0.609
open+close	24.946	6.973	9.192	1.103
lseek+read	3489.270	6.186	8.637	0.716
gettimeofday	4.134	2.479	3.668	0.019
getpid	0.0707	2.549	3.9308	0.0022

Table 3.10: Measuring the performance of the frameworks using microbenchmarks. The time reported (in seconds) is for 1 million executions of the system calls (RQ4).

Panoply outperform Graphene-SGX. Graphene-SGX also offers poorer performance for `write` and `open+close`, likely due to the additional operations within its shield. The performance of Panoply and Porpoise is roughly comparable.

Summary (RQ4): No one model appears as the clear favourite with respect to runtime performance. Using a library OS can provide the benefits of file caching for some applications. Both the library-wrapping and instruction-wrapping models perform better for network-bound applications than the library OS model.

3.6 Discussion on Other TEEs

Some of the insights from this work can apply to other TEEs as well. In the case of VM-based TEEs, e.g., Intel TDX, AMD SEV, and Arm CCA, they will face challenges similar to library OS models. Developers will be able to quickly port their workloads due to native binary execution in a VM-based environment. However, they won't be easily re-engineer the application as the entire workload executes within the VM. The overall system will have a larger attack surface due to the inclusion of the guest kernel in the TCB of the application. Recent work [102] performs an empirical evaluation of Intel TDX and AMD SEV-SNP, and reports the performance of confidential virtual machines against native virtual machines. The performance of the confidential virtual machine is impacted due to the cryptographic operations.

Porting applications to Arm TrustZone is challenging as it runs a separate minimal operating system. As Arm TrustZone also has fixed protected memory similar to Intel SGX, splitting the application would be challenging, as there is a narrow interface between the secure world and the normal world. However, due to the minimal trusted operating systems, the TCB of the trusted application would be smaller than the standard confidential virtual machine. However, the main memory is not encrypted, which makes trusted applications prone to physical attacks [142, 78, 53, 52, 167, 95]. Recent work [60] reports a performance comparison between Arm TrustZone and Arm CCA. Both of them have similar performance on memory and CPU-intensive workloads, but Arm CCA outperforms Arm TrustZone on file system workloads.

3.7 Conclusions

No clear consensus has emerged thus far in the community on the right abstractions for enclave programming, especially as concerns porting legacy code to enclaves. We considered the three models that have been proposed in the research literature, namely the library OS, library-wrapping and instruction-wrapping models. Based on our experience porting a number of application benchmarks to Graphene-SGX, Panoply, and Porpoise, we conclude that the choice of the enclave programming model to be used depends on the factors that application developers wish to optimize for:

- *Rapid prototyping.* Developers may wish to quickly prototype an in-enclave version of their application. This can serve as a stop-gap solution that provides the benefits of enclaves as a team develops a version of the application customized for the enclave. The library OS and instruction wrapper models are ideally suited for this setting.
- *Source code availability.* With a legacy application, source code may often be unavailable or recompilation may not be feasible, *e.g.,* due to library compatibility issues. In such settings,

only the library OS model allows developers to create enclave versions of the application. Both the instruction wrapper model and the library wrapper model either require access to source code, or require the legacy binary to be linked with suitable wrappers.

• *Flexibility to re-engineer.* With a quick first-cut of their application executing in the enclave, application developers may wish to optimize the execution of the enclave, *e.g.,* by reducing the number of domain crossings, or reducing the amount of code executing in the enclave. It goes without saying that the application's source code is required for such re-engineering. Both the instruction wrapper and library wrapper models are best suited for this setting.

• *Concerns about TCB size.* Application developers may wish to reduce the amount of code executing within the enclave in an effort to reduce the size of the attack surface of their security-critical code. Only the Panoply implementation of the library wrapper model optimizes for this criterion. However, it also entails executing much of the standard C library outside the enclave (Figure 3.1), and the in-enclave application must take suitable precautions when it makes library calls (*e.g.,* by checking return values, in a manner similar to shields for IAGO attacks).

• *Performance.* In our evaluation, no one model emerged as a clear winner with respect to runtime performance, and the developer must choose the enclave programming model that works best for the application at hand. Library OSes can provide good performance for some applications, *e.g.,* by offering caching and avoiding domain crossings, as saw with `Bzip2` and `Cpython`. However, because enclave execution by itself imposes overheads, and library OSes execute entirely within the enclave, they may also offer poor performance in some cases, as we saw with `Memcached` and `H2O`. The instruction wrapper and library wrapper models do offer the potential for better performance if software developers have the flexibility to profile and re-engineer their applications by reducing domain crossings.

• *Support for evolution.* Finally, with respect to code evolution, the library OS and instruction wrapper models are better suited with respect to application code evolution than the library wrapper model. As discussed in this work, the system call interface evolves much slower than the library call interface, thereby allowing the library OS and instruction wrapper models to provide better support than the instruction wrapper model as the enclave application code evolves to newer versions.

Chapter 4

MazeNet: Protecting DNN Models on Untrusted Cloud Platforms with Trusted Hardware

Machine Learning-as-a-Service (MLaaS) enables DL model owners to outsource inference tasks to a public cloud platform. The model owner trains a deep learning model in-house and uploads the trained model to an MLaaS. For the uploaded model, the MLaaS platform exposes an API endpoint to query the uploaded model with inputs and obtain predictions. However, uploading the trained model to public cloud platforms exposes the model owner to security and privacy risks, as the model is available in plaintext to the cloud provider during inference.

In this work, we present a set of techniques to secure deep learning models with trusted execution environments and propose a secure outsourcing scheme to offload portions of the DL model computations during inference to faster untrusted processors. We implement the presented techniques in MazeNet, a framework to transform pre-trained models into MazeNet models and deploy them on a public cloud platform to provide inference services.

We evaluate MazeNet on popular convolutional neural networks, and the results demonstrate that MazeNet improves the performance of DNN models as compared to a secure baseline model, where the model runs within a trusted environment. MazeNet increases the throughput of the inference task up to 30x and decreases the latency up to 5x for the benchmark models in our experimental evaluation.

4.1 Introduction

Machine Learning-as-a-Service (MLaaS) enables deep learning model owners to deploy trained models on public cloud platforms to provide inference services. The model owner pre-trains a deep learning model in-house and uploads the trained model to a public cloud platform. In turn, MLaaS exposes an API endpoint to the uploaded model to query the model with inputs and obtain predictions.

However, moving the deep learning workload to a public cloud platform exposes the model owner and the uploaded model to security and privacy risks. First, a compromised or malicious MLaaS provider can easily steal the deployed models as the model is available in plain text during inference because the cloud vendor controls the entire hardware and software stack at its data centres. A stolen model leads to financial losses and legal troubles for the model owner. Most of the time, organisations have invested significant financial resources to train state-of-the-art models [67] that solve critical business problems.

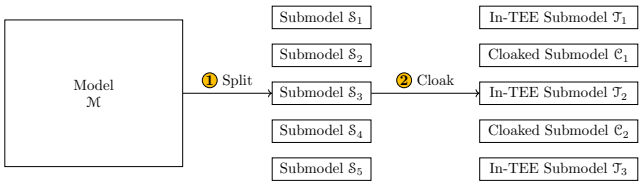
Second, having access to model parameters and intermediate states, the DL model can leak sensitive information about their private training dataset through membership inference attacks [137]. Often, these models are trained on private datasets to improve the accuracy of the learning task. Legal laws in many countries require that private data, such as financial transactions [1], electronic health records [2], insurance, etc., be protected. Otherwise, the defaulting organisation will be penalised heavily [1, 2].

Third, the MLaaS provider can tamper with the uploaded model to influence the results. Therefore, it becomes crucial to protect the integrity and confidentiality of the uploaded models on public cloud platforms.

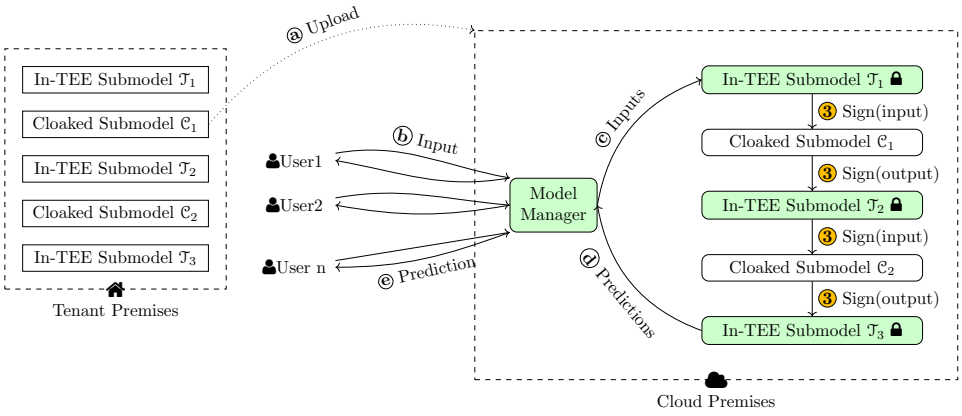
A naive solution to protect the model on an MLaaS platform would be to run the model within a Trusted Execution Environment (TEE) such as Intel Software Guard Extensions (SGX). The TEE will be responsible for ensuring the confidentiality and integrity of the DL model during the inference process. However, directly running the DL model within the TEE is suboptimal.

First, TEEs often have fixed and usually smaller protected memory than the main memory. For example, SGXv1 offers only 128 MB of protected memory. Therefore, a DL model larger than protected memory incurs a performance penalty. Second, TEEs cannot securely utilise untrusted but powerful resources such as co-located processors, main memory, and storage available on the system.

In this work, we present MazeNet, a framework to transform pre-trained models into MazeNet models and securely run them on heterogeneous and distributed systems consisting



(a) On-premises, a Model Builder splits ① the pre-trained model and cloaks ② a subset of submodels to produce a MazeNet model.



(b) On the cloud, a Model Manager deploys the MazeNet model, and exposes an API for users to query the model. During the inference, the MLaaS provider signs inputs and outputs of cloaked submodels ③ that are outsourced to untrusted environments.

Figure 4.1: End-to-end workflow of MazeNet system.

of trusted execution environments and untrusted runtimes. MazeNet uses three key techniques to overcome the limitations of TEEs and provides a secure and fast inference solution.

First, to overcome the fixed protected memory of SGX enclaves, MazeNet splits the DNN model into smaller models, referred to as submodels, such that the maximum memory usage of each submodel during inference is less than the size of the protected memory offered by the SGX enclaves. During inference, MazeNet distributes the submodels to different SGX enclaves. Splitting avoids the performance penalty due to swapping when submodels execute within SGX enclaves. Figure 4.1a shows an illustration of splitting, where a model $\mathcal{M}$ is split ① into five submodels $\mathcal{S}_1 \ldots \mathcal{S}_5$.

Second, running submodels only within TEEs leaves other untrusted resources underutilised. Therefore, to maximise the system utilisation and improve the performance, a subset of submodels is outsourced to untrusted runtime environments. However, outsourcing to an untrusted environment raises privacy risks for the outsourced submodels. Therefore, MazeNet employs its second technique, cloaking, a secure outsourcing scheme to offload submodel evaluation to untrusted hardware. In cloaking, synthetic layers and neurons are added to the submodel, which

	Data transfer (MB)		
Model	TEE→GPU	GPU→TEE	Total
VGG16	34.77	51.71	86.48
ResNet50	38.70	40.39	79.09
DenseNet201	91.43	29.96	121.39

Table 4.1: Data transfers between the TEE and GPU when only linear layers are outsourced to GPUs, while non-linear layer executes within the TEE.

hides the original weights within the synthetic weights to protect the privacy of the outsourced submodel. In the Figure 4.1a, after splitting, submodel $\mathcal{S}_2, \mathcal{S}_4$ are cloaked ② to produce final MazeNet Model.

Third, during inference, the adversary can tamper with any data or computation outside the TEEs, which includes cloaked submodel weights and operations. MazeNet employs digital signatures to detect tampering in offloaded computations. It requires the cloud vendor to sign ③ the inputs for the cloaked submodels and the outputs produced by the cloaked submodels, as shown in Figure 4.1b, to commit to submodel evaluation results. The signatures are later used during an audit phase, where MazeNet independently verifies some of the outsourced computations by re-executing some of them.

Figure 4.1 shows the end-to-end workflow for the MazeNet framework. First, a cloud tenant transforms the pre-trained model into a MazeNet model with Model Builder and uploads the generated model to a public cloud platform, where a Model Manager securely deploys the uploaded model and exposes an API endpoint to query the model.

Prior works [143, 55, 152] have proposed secure outsourcing schemes to outsource linear layers of DL models to hardware accelerators. However, outsourcing only linear layers is suboptimal, as linear layers are frequently followed by non-linear layers. As a result, the intermediate state needs to be constantly moved between the TEE and the GPUs. Table 4.1 shows the data transferred between TEE and non-TEE for prior works. MazeNet overcomes this limitation by outsourcing both the linear and the non-linear layers to GPUs, thereby reducing communication costs. MazeNet can reduce up to 90% of the data transfer cost.

To evaluate the benefits and costs of the proposed techniques, we have built a prototype of the MazeNet framework on top of TensorFlow [48]. We transform popular convolutional neural networks, VGG [138], ResNet [56], and DenseNet [59], into MazeNet models and compare them against a secure baseline model, where the whole unmodified model runs within a TEE. Our evaluation shows that MazeNet can improve the throughput up to 30x and reduce the latency up to 5x for the convolutional neural networks in our experimental evaluation.

To summarise, the following are the main contributions of this work:

- This work presents MazeNet, a framework to secure trained deep learning models on public cloud with trusted execution environments.
- It proposes an outsourcing scheme to offload both linear and non-linear layers to untrusted environments to accelerate the model inference.
- Our evaluation shows that MazeNet can significantly improve the throughput and the latency of DL models as compared to the secure baseline models.

4.2 Background

4.2.1 Deep Neural Networks

Deep Neural Networks (DNNs) consist of layers that are chained together to form a graph. Each node in the graph represents a layer, and each edge represents a tensor, a multi-dimensional array. A layer can take one or more inputs and produce one or more outputs. When the output of layers is input to the following layers, such networks are referred to as feed-forward neural networks. Each layer in the network has a set of parameters or weights that can be fixed or learned during the training phase.

DNNs for image classification tasks contain convolutional layers and are commonly referred to as convolutional neural networks (CNNs). Convolutional layers contain a set of filters – also referred to as kernels – that are applied to the input image or data during a convolution operation to produce feature maps. Filters detect distinct features present in the input data, and enable the sharing of parameters or weights across multiple neurons to reduce the overall number of parameters in the network. The output of the convolution operation may pass through a non-linear activation function such as ReLU [138], which enables the network to learn arbitrarily continuous functions [121]. Convolutional layers can be followed by pooling layers, which downsample the feature maps to limit overfitting and reduce the number of parameters and operations in the following layers [43]. The last layers in CNNs consist of a few fully connected layers, which take the feature maps and produce confidence scores for each class of the classification tasks [59, 138, 56].

4.2.2 Privacy Risks in Trained DNN Models

Deep Neural Networks are expensive to train due to the costs associated with the different stages of training. First, training highly accurate models for complex tasks requires a large and diverse

training dataset to generalise better for the learning task. For instance, the ImageNet[126] dataset used for training state-of-the-art convolutional networks for image classification tasks consists of a curated set of 14 million photos divided into 1000 classes. It takes time to collect, clean and annotate a comprehensive training dataset. Further, the model needs to be fine-tuned with different architectures and hyperparameters to maximise the accuracy of the target task, which further escalates the training cost. To put this in perspective, the XLNet [166] architecture incurred around $60K during training. Thus, it is important for any organisation to guard the privacy of the trained models, as it has invested considerable resources in the training.

Further, private models are often trained on sensitive training datasets that may contain patient health records, medical scans and reports, and financial transactions, to name a few. Prior works have shown that trained models can leak sensitive information about the training dataset through membership inference attacks [29, 137]. Many countries have strict laws on the privacy of medical [2] and financial data [1], and leaking the private data can lead the data owner to financial and legal troubles.

4.2.3 Attacks on DNN Models

Prior work has shown that DNN models are vulnerable to exposing trade secrets [15], leaking training dataset [23], model inversion attacks [40, 41], model stealing attacks [113, 153], adversarial attacks [47, 112, 111, 144], and membership inference attacks [29, 80, 128, 127, 137].

These attacks can be broadly classified into two categories: white-box attacks [19, 47, 112, 144] and black-box attacks [23, 29, 111, 113]. In white-box attacks, the attacker has full access to model parameters and intermediate states to carry out the attacks. The attacker can query the model with inputs, observe the intermediate states during inference, and retrieve prediction scores. Under the black-box setting, the adversary does not have access to the model internals; it can only query the model through an API and obtain prediction scores or labels.

Researchers have demonstrated methods to steal deep learning models even if the adversary has black-box access to the model. However, these attacks focus on stealing the functionality of the model instead of trained parameters or weights. In these attacks, the attacker builds a shadow training dataset by querying the model and recording the input-output pair. The attacker can use public datasets such as CIFAR or ImageNet[126] to query the model, or it can use learning algorithms to sample input data points [109]. With this training dataset, the attacker trains a shadow model that tries to approximate the functionality of the target model that the attacker wants to steal. However, the weights and architecture of the shadow model in these model extraction attacks are entirely different from the target model.

In addition to model stealing attacks, having white-box access to the model enables the attacker to carry out adversarial attacks [47] on the model, where an attacker crafts malicious input samples that look benign to the human eye, but the model misclassifies the input into a class of the attacker's choice. Another line of work has focused on the membership inference attacks [137] where the attacker wishes to determine whether the input data point was part of the training dataset on which the model was trained.

4.2.4 Defences Against Model Stealing Attacks

The research community has proposed several solutions to defend against model-stealing attacks. These solutions can be broadly categorised as follows: cryptography-based approaches [27, 44, 69, 76, 87] and solutions based on hardware-based trusted execution environments [17, 54, 55, 77, 79]. Cryptography-based approaches provide stronger security guarantees but incur performance penalties due to high communication costs and heavy cryptographic operations. On the other hand, hardware-based trusted execution environments provide higher performance with limited security guarantees. TEE-based solutions look promising on the untrusted cloud platforms to provide real-time inference services with high throughput and low latency to meet required service level agreements. Table 4.2 lists the security guarantees and performance scaling of prior works.

4.3 Building MazeNet Models

In this section, we describe the process of transforming a given model into MazeNet models. In the next section, we will look at how to securely deploy the generated model on a public cloud platform to offer inference services.

4.3.1 Threat Model

A cloud tenant owns a model trained on private training data and wants to outsource the model inference service to a public cloud provider without compromising the privacy of the trained model. Our aim is to prevent the cloud vendor from learning the trained parameter weights from the deployed model on its platform. We assume that the cloud vendor does not have access to the training data and is not aware of the architecture of the pre-trained model.

MazeNet admits a strong adversary based on the standard threat model of Intel SGX. It only trusts the Intel SGX platform and the code and data located within SGX enclaves. The adversary controls the privileged system software, including the operating system, firmware, and BIOS. It can read, analyse, and modify arbitrary code and data located outside the Intel SGX enclave. However, it cannot observe or modify any data within SGX enclaves.

Prior Work	**Model Privacy** (server)	**Input Privacy** (server)	**Outsource to Untrusted Hardware**	**Scalable to Large Models**	**Platform / Technique**
Securenets[27]	✓	✓	✓		SMM
Cryptonets[44]					FHE
Chameleon[123]		✓			GC
DeepSecure[125]	✓	✓			GC
Crypflow[76]	✓			✓	MPC
SecureML[105]		✓			MPC
MiniONN[87]		✓			FHE and GC
Gazelle[69]	✓				FHE and MPC
Shadownet [143]	✓		✓		ARM TrustZone
DarknetTZ[103]				✓	ARM TrustZone
OMG[17]	✓				ARM TrustZone
Occlumency[79]		✓			SGX
TensorScone [77]	✓				SGX
DarkNight [55]	✓		✓		SGX
MLCapsule[54]	✓	✓			SGX
Slalom [152]			✓		SGX
MazeNet	✓		✓	✓	SGX

Table 4.2: Feature comparsion of prior works with respect to the privacy of models and user inputs. SMM: Secure Matrix Multiplication, SGX: Intel Software Guard eXention. FHE: Fully Homomorphic Encryption. MPC: Multi-Party Compuation. GC: Garbled Circuits

Adversary Capabilities. The adversary has full access to the outsourced computations that run on untrusted devices, e.g. GPUs and untrusted processors. The adversary can inspect and alter the outsourced DL model computations along with the input and output of those computations. The adversary can obstruct access to resources, e.g., memory and CPU time, such that the enclave application does not make any progress.

Adversary Information. The adversary is not aware of the model architecture and trained weights that the model owner has trained on its premises. However, during the inference process, the adversary can learn from the data and computations delegated outside the enclave. The adversary is aware of all the defence strategies used to protect the privacy of the model, but it does not know about the keys or data that may be generated at runtime within the enclave, unless that data leaves the enclave without any encryption.

The main goal of this work is to protect the privacy of trained parameters of a deep learning model. This work does not aim to protect the privacy of user inputs, which are fed to the model during the inference process, as it is orthogonal to the goal of the model's privacy.

Prior research has demonstrated several side-channel attacks against SGX [161, 26], and Intel is actively working to fix those side-channel attacks [63]. Therefore, side-channel attacks are outside the scope of this work.

4.3.2 Overview of MazeNet

MazeNet enables a cloud tenant to securely deploy a private pre-trained model on a public cloud platform to provide inference services to users, while protecting the privacy of the model from the host platform. It relies on hardware-based trusted execution environments for the privacy of the model, and uses untrusted but powerful co-processors, e.g., low-cost commercial GPUs, to speed up the inference task.

A cloud tenant pre-trains a DNN model on-premises and uses the MazeNet framework to transform it into a MazeNet model. Then, the tenant uploads the generated model to a public cloud platform to provide inference services.

MazeNet framework consists of two parts: MazeNet Model Builder and Model Manager. Given a pre-trained model, Model Builder transforms the given model to a MazeNet model, while the Model Manager carefully deploys the generated model on trusted and untrusted environments to provide inference services.

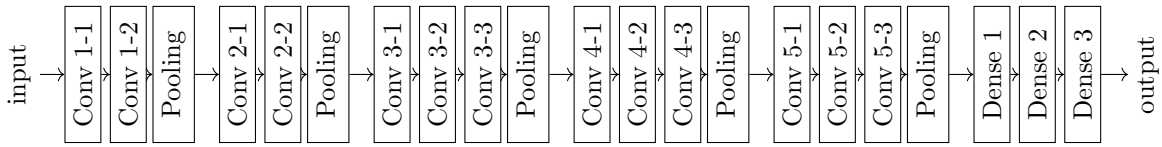
In the next section, we describe the process of transforming pre-training models into MazeNet models, and Section 4.4 describes the process of running the generated model on public cloud platforms.

4.3.3 Splitting DNN Models

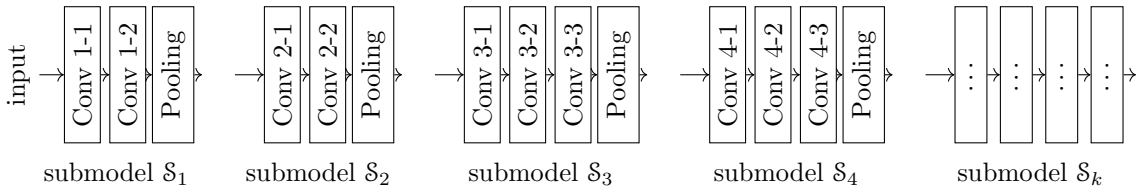
The size of DNN models is increasing as new state-of-the-art architectures are introduced, and larger models are trained to achieve high accuracy on the learning task. Recent report [10] states that computing resources needed for DNNs double every 3.4 months, with GPT-3 model reaching 175 billion parameters [21].

Due to their large size, many DNN models do not fit within the fixed protected memory offered by TEEs. In case of SGX enclaves, the hardware can only cryptographically protect a small portion of main memory [55, 146]. Therefore, models larger than the size of main memory incur a performance penalty due to swapping.

Intel SGX reserves a portion of main memory called Processor Reserved Memory (PRM) during the boot process to store the encrypted pages of the enclaves in the main memory. This portion of memory is referred to as Enclave Page Cache (EPC), whose confidentiality and integrity are backed by the SGX hardware. As this is a fixed portion of memory, enclave applications and DNN models that need more memory than the protected memory incur EPC swapping, where the SGX driver in the kernel seals a few pages of EPC, which were not recently used, and stores them on the unprotected memory. Thus freeing a few pages in the EPC for enclave applications. When the enclave applications need to access any of the swapped pages,



(a) VGG16 model architecture



(b) VGG16 model after splitting into multiple submodels

Figure 4.2: Given a DNN, MazeNet splits it into multiple smaller models called submodels based on the split specifications provided by the user.

the SGX driver restores the swapped-out page after checking for integrity violations when the pages were residing in the unprotected memory.

The EPC swapping process is computationally expensive due to the cryptographic operations required to seal encrypted memory pages. Further, if any page is swapped out and it is required by the enclave application, the application stalls until those pages are restored. Thus, the performance of the enclave application is severely degraded. In the case of deep learning workloads, the throughput and latency of the inference task are affected. We conducted an experiment to determine the impact of EPC swapping on inference workflows. We observed that the throughput in the VGG16 model decreased from 3 images/s to 0.3 images/s, a 10x decrease in throughput, due to EPC swapping. Thus, only a small portion of the DNN models should be placed within SGX enclaves to avoid swapping.

To overcome this limitation of fixed protected memory, MazeNet splits large DNN models into smaller models referred to as submodels, such that each submodel fits within the protected memory of the SGX enclave. After splitting, each submodel can be deployed on different TEEs to avoid swapping.

Splitting DNN models is challenging due to diverse model architectures and complex interactions between layers consisting of residual or skip connections and densely connected layers. The model owner should consider the following parameters when splitting a DNN model.

First, because the model is split across TEEs, the output from one submodel must be transferred to the next submodel residing in a different TEE. Depending on where the model is split, the size of the outputs produced by the submodel can vary significantly. As a result,

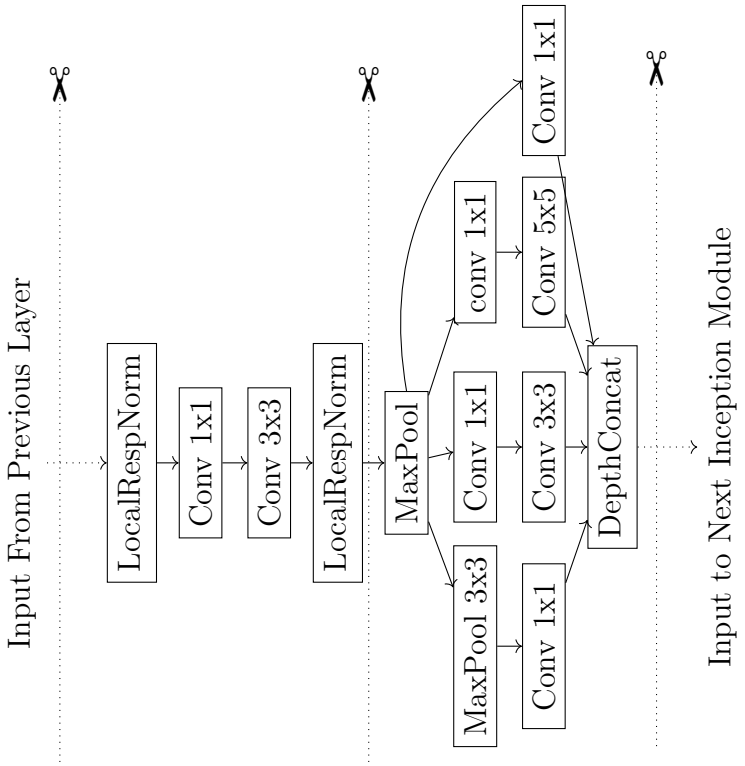


Figure 4.3: An instance of splitting the GoogleNet model [145] consisting of inception modules containing parallel layers, where the model is split at the block or module level.

it is important to choose a split that minimises the size of the inputs and outputs exchanged between submodels to reduce communication overhead across enclaves.

Second, the number of parameters varies across layers, leading to differing computational workloads – particularly in terms of additions and multiplications. As a result, each layer requires a different amount of time to compute results. To ensure efficient execution, the model should be split in a way that balances the computational load across submodels. An imbalanced split may cause stalling, where one submodel waits for data from the previous submodel, which is still computing the results. Such stalls degrade throughput and increase the overall latency of the inference task.

Third, the peak memory usage of each submodel during inference must remain within the bounds of the protected memory. The model should be split in a way that ensures no submodel exceeds this limit; otherwise, EPC swapping may occur, which significantly increases the execution time.

To find a suitable split, a model owner can use either analytical or empirical methods. In the analytical method, the model owner can estimate the execution time by computing the

number of operations, additions and multiplications present in layers. These estimates can be used to balance operations across submodels. Similarly, for the data transfers, the model owner computes the input and output size of each layer and tries to find a split that minimizes the input-output size of the resulting submodels. In the empirical case, the model owner executes the unmodified model and measures the execution time and input-output size of individual layers to determine the split. Once the owner has determined the split, they can proceed with the model splitting.

Figure 4.2 shows one such splitting of the VGG16 model, where it is split into smaller submodels. The VGG16 model consists of 16 trainable layers whose weights are learned during the training, e.g., convolutional and dense layers, and a few non-trainable layers, e.g., flatten and pooling layers. In the given split, the VGG16 model is split after each pooling layer, as it reduces the output size. Therefore, we have six submodels. Five submodels are convolutional blocks containing convolutional and pooling layers, and the last block contains dense layers. The first submodel consists of the first three layers of the VGG16 model. The second submodel consists of the following three layers, and so on. In case of models with more complex architecture, for example, models with residual or skip connections, e.g., ResNet [56], DenseNet [59], models can be split at block levels, where a group of layers highly interconnected than other layers. Figure 4.3 shows one such split of a GoogleNet model.

Formally, a feed-forward DNN model $\mathcal{M}$ can be represented as a directed acyclic graph. In this graph, the layers of the models are represented as nodes of the graph, and the input-output relationship between layers is represented as edges of the graph. A single split of the graph cuts the graph into two disjoint subsets of vertices, and the size of the cut is the number of edges whose vertices are in different subsets. The splitting of a model will consist of multiple such cuts of its graph.

Consider the model $\mathcal{M} = \{L_1, L_2, \ldots, L_n\}$ with n layers, where the layers L_i are arranged in a topological ordering of the vertices of the corresponding graph. A split or cut partitions the vertices into subsets. The first subset is referred to as the submodel $\mathcal{S}_1 = \{L_1, L_2, \ldots, L_{k1}\}$ and it consists of first $k1$ layers. The remaining subgraph is cut again to obtain second submodel $\mathcal{S}_2 = \{L_{k1+1}, L_{k1+2}, \ldots, L_{k1+k2}\}$ with $k2$ layers. The process is continued for the remaining subgraph. The number of layers, $k1$, $k2$, $\ldots$, km, in each submodel is provided by the model developer after the analytical or empirical evaluation.

4.3.4 Submodel Cloaking

The submodels obtained from splitting can be hosted on TEEs to provide secure inference services. However, deploying submodels only on TEEs leaves other powerful but untrusted system

resources underutilised. These untrusted resources can range from unprotected main memory, faster processors such as GPUs, Tensor Processing Units (TPUs), and Neural Processing Units (NPUs). Therefore, a subset of the submodels is deployed in the untrusted runtime environment to boost the performance of the inference service. The submodels deployed on TEEs are referred to as in-TEE submodels, while the submodels deployed outside the enclaves are referred to as non-TEE submodels.

However, outsourcing submodels to untrusted environments poses the following security and privacy risks, which were earlier mitigated by TEEs. First, a passive adversary can observe the delegated submodels in plain text, and therefore, they can learn the connection between different layers along with their parameters and weights. Second, an active adversary can tamper with outsourced computation. It can replace the weights of outsourced submodels, change the results, or perform replay attacks on outsourced computations.

To protect the privacy of outsourced submodels, we employ our second technique of cloaking, where the submodels are cloaked before deploying them to untrusted environments. In cloaking, synthetic layers and neurons are added to the submodel to produce a cloaked submodel, where the original submodel is embedded within the cloaked submodel.

Key idea. The key idea behind cloaking is that parameters or weights of a layer are a set of matrices, and it is difficult to distinguish whether a given matrix is part of a trained model, if the matrices were drawn from a trained model and a sample distribution containing trained parameters. For example, filters in convolutional layers detect different features. Given two filters, each detecting different features, it is difficult to determine which one of the filters is part of the trained model.

Prior work on model explanations that tries to reason about the working of DL models suggests that individual filters and units need a global view of the entire model to reason about the usefulness or role of individual units or filters in the predictions [144].

Often, there is no unique solution to the learning task on which a DL model is trained. A DL model tries to learn an internal representation of the training dataset. Training a DL model with different initial weights yields entirely different model weights. This problem is further exacerbated by over-parameterisation, where the model may contain a higher number of parameters than required. Thus, there exists a large set of trained weights that can be part of trained models.

Therefore, when synthetic weights are added in submodels, an adversary cannot distinguish between the embedded weights of the original models and the newly added synthetic weights. It can only observe the partial computation, which contains a mix of actual and synthetic computation. The keys to recover results from cloaked submodels are securely stored within

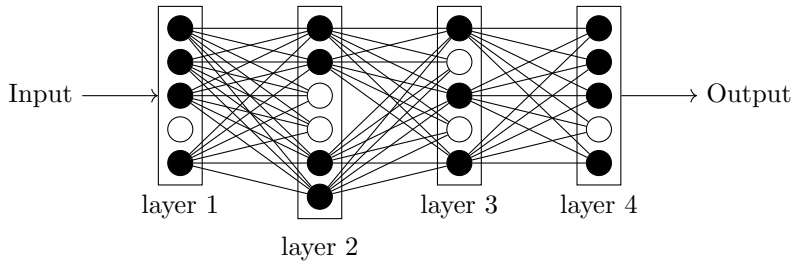


Figure 4.4: Cloaking Phase 1: Adding synthetic neurons in a four layer neural network. Synthetic neurons (○) are added between existing neurons (●) during the first phase of cloaking. Currently, the output of synthetic neurons (○) are not connected to the input of following layers as it will affect the correctness of embedded neurons (●). The output of synthetic neurons will be used later as input to synthetic layers which will be added in the second phase of cloaking.

the TEEs.

Selection of submodels for cloaking. The submodels obtained after splitting are categorised into in-TEE and non-TEE submodels. The submodel containing the first layer always runs within a TEE as it operates on plain text user input and can leak privacy of input sample [38]. Therefore, the first layer is always categorised as an in-TEE submodel. Similarly, the last layer of the DNN model computes the confidence score or prediction label. Thus, the submodel containing the last layer is also categorised as an in-TEE submodel. The remaining submodels are selected alternatively to be categorised as in-TEE and non-TEE submodels. The in-TEE submodels are deployed on TEEs, whereas non-TEE submodels are cloaked and deployed on untrusted environments.

From the previous example of splitting VGG16 model in Figure 4.2b, We obtained five submodels $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_5$. Out of these five submodels, three submodels $\mathcal{S}_1, \mathcal{S}_3, \mathcal{S}_5$ are categorised as in-TEE submodels; recall that the first and last layer should execute within TEE. The remaining two submodels $\mathcal{S}_2, \mathcal{S}_4$ are labelled non-TEE, thus each of them is cloaked.

4.3.5 Cloaking Process

A non-TEE submodel is cloaked in two phases. In the first phase, synthetic neurons are inserted into the layers of a non-TEE submodel, which hides the embedded neurons within a layer. Then, in the second phase, synthetic layers are added to the submodel obtained from the first phase to hide embedded layers. To steal the embedded submodel from the cloaked submodel, an adversary has to first guess the layers and then the neurons to recover the embedded submodel.

4.3.5.1 Phase 1: Adding Synthetic Neurons.

In the first phase of cloaking, synthetic neurons are added to the existing layers. A neuron computes a weighted sum of the input and applies a non-linear activation function, e.g., ReLU, to compute the output. The weights for the synthetic neurons are sampled from the same probability distribution as that of the existing neurons in the layer to which the synthetic neurons are added.

Sample weight distributions. The synthetic weights for neurons should be drawn from the probability distribution of trained parameters for the corresponding layer. However, the probability distribution for each layer is unknown beforehand, and during the training, layers learn one of the instances of these weights from the probability distribution. The probability distribution differs across layers as each layer is responsible for learning a specific level of features. A sample population of weights can be created for each layer of the model during the training and hyperparameter tuning phase by recording the weights of each layer. Moreover, a sample population can also be generated by training the model multiple times with different initial weights.

Synthetic filters. Convolutional layers contain filters with dimension (h, w, c_{in}, c_{out}), where h and w is height and width of filters, c_{in} is the number of input channels, c_{out} is the number of feature maps produced by the convolutional layers. To add s synthetic filters, s matrices with the same height, width and input channels are drawn from the sample distribution for the given convolutional layer, and stacked with the existing filters, which changes the filters dimension of the convolutional layer to $(h, w, c_{in}, c_{out} + s)$. As filters are independent of each other, they are shuffled to hide the newly added filter from the existing ones.

Synthetic neurons. To add synthetic neurons in dense layers, column vectors are sampled from a sample weights distribution and inserted into the weight matrix. If the size of the vector on which the neuron performs the weighted sum is n and the dense layer has m neurons, then the size of the existing weight matrix is (n, m). To add s synthetic neurons, s column vectors are sampled from the distribution and added to the weight matrix. Then, the vectors within the weight matrix are shuffled to hide new column vectors. The final weight matrix will have $(n, m + s)$ size.

Adding synthetic neurons and filters increases the size of the output produced by layers, which affects the following layers due to a mismatch in the expected input dimension and actual input. For example, a dense layer initially has 16 neurons and produces a vector of size 16. If 16 neurons are added, then it produces a vector of size 32 instead of 16, which is not compatible with the following layer, which was expecting the input size to be 16. Therefore, the output of

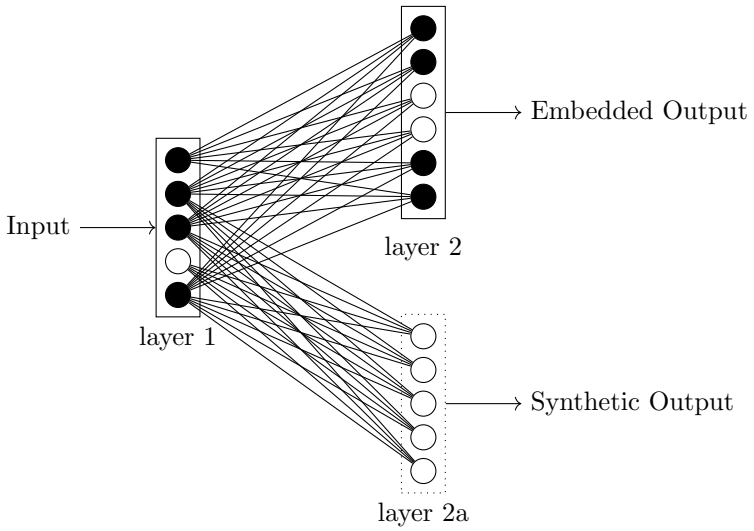


Figure 4.5: Cloaking Phase 2: Adding a synthetic layer, `layer 2a`, to a two layer, `layer 1` and `layer 2`, fully connected neural network where `layer 1` contains one synthetic neuron (○) and `layer 2` contains 2 synthetic neurons (○) that were added during the first phase of cloaking. Only the correct output of `layer 1` produced by embedded neurons (●) are input to `layer 2`, whereas a compatible subset is selected as input to `layer 2b`.

the cloaked layer is filtered before feeding it to the following layers. Figure 4.5 shows that the output of synthetic neurons is filtered before feeding them to the following layer. The output of the synthetic neurons is used later in the second phase of cloaking when synthetic layers are added.

Further, shuffling the filter and weight matrices of a layer to hide the synthetic neurons changes the ordering of elements in the output tensor, which causes the following dense and convolution operations to compute incorrect results during the dot product operation between the shuffled output from the previous layer and the layer's weights. Therefore, the weights of the following layers are also reordered such that dot products compute the same results before the synthetic neurons were added.

After synthetic neurons are added, embedded weights are hidden from the adversary. However, the adversary still knows that a subset of the weights are from the embedded model for any given layer. To further hide the embedded layer as well, we add synthetic layers.

4.3.5.2 Phase 2: Adding Synthetic Layers

In the second phase of cloaking, synthetic layers are added in non-TEE submodels obtained from phase 1 to build the final cloaked submodel. Synthetic layers hide the embedded layers of the submodel in cloaked submodels.

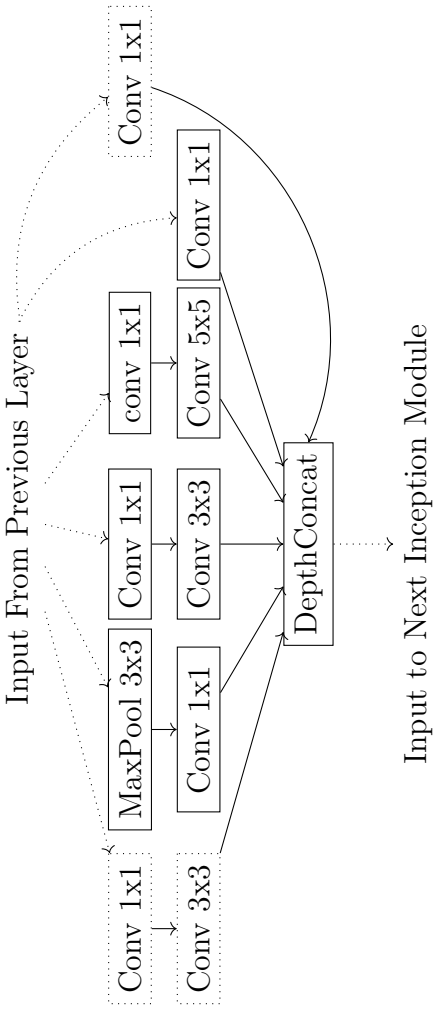


Figure 4.6: Cloaking Phase 2: Adding synthetic layers, Layer, to one of the inception modules of the GoogleNet model [145].

A submodel can be hidden in another model as new architectures have been introduced, which move away from the traditional sequential architecture where the output of one layer is input to the immediately following layer. State-of-the-art models differ widely in their architecture from the introduction of skip connections in ResNet [56], densely connected layers in DenseNet [59], parallel layers in the Inception module of GoogleNet [145], and ensemble models, [56, 42], which combine multiple independent models to form a bigger model. Thus, a submodel may appear to be part of multiple architectures. Further, adding synthetic layers increases the difficulty of identifying embedded layers based on the input-output relationship between the layers in the cloaked submodels.

A synthetic layer can be attached to a submodel by selecting intermediate outputs of existing

layers as inputs. In contrast to phase 1 of cloaking, adding synthetic layers does not alter computations of other layers present in the submodel, as the output of the synthetic layer is not fed to any embedded layer. A synthetic layer will be followed by a set of synthetic layers, and the final or last synthetic layers in the cloaked submodel will produce synthetic outputs that are forwarded to enclaves for further processing. During the inference phase, the enclaves have access to keys that can filter synthetic output from embedded outputs.

Input of synthetic layers. To add a synthetic layer to the submodel, the intermediate output of one of the intermediate layers is selected as an input to the new layer. However, the output will contain both synthetic and embedded output as synthetic neurons were added in phase one of cloaking. Therefore, a subset of values compatible with the synthetic layer is selected to create input for the synthetic layer. The number of selected values will depend on the type of the synthetic layer.

Convolutional synthetic layer. To add convolutional layers, the filter weights are drawn from the sample distributions. The sample distributions differ depending on the depth of the synthetic layer in the overall model. For example, initial layers detect simple features such as edges, while deeper layers detect complex features such as facial expression. The filter weights of the given size are drawn from the sample distribution corresponding to the depth of the synthetic layer.

Dense synthetic layer. Similarly, for dense synthetic layers, weight matrices of compatible size are drawn from the sample distribution and added to the synthetic layer.

The synthetic outputs produced by the synthetic layers are later eliminated by the TEEs. MazeNet uses taints to track the status of each value produced by the layers in the cloaked submodel. The taint produced during cloaking is transferred securely to SGX enclaves, which filter synthetic and embedded outputs.

Tracking of synthetic and embedded outputs: For each input, intermediate, or output tensor, a corresponding taint tensor is maintained. Each value in the taint tensor indicates whether the corresponding value in the actual tensor is `real` or `synthetic`. For the initial user input, the taint tensor is initialised with `real` label. As the input passes through the synthetic layers and synthetic neurons, the corresponding value in the taint tensor is set to `synthetic`. Similarly, `real` label is set for values produced by embedded layers or neurons. Figure 4.7 shows the rules for taint propagation when the input passes through different layers.

The taint tensors of the last layers in a submodel are stored and used later in enclaves to filter real output from synthetic output. As the input flows through the same order of synthetic layers – the same set of computations is performed – the taint tensor is static and computed during the cloaking process. During inference, the enclave loads taint tensors to filter synthetic

<Model> ::= **Model**(< layers >, < inputs >, < outputs>)	input.taint = new_real_taint(shape=input.shape)	(4.1)
<inputs> ::= <tensors>		(4.2)
<outputs> ::= <tensors>		(4.3)
< input > ::= **Tensor**		(4.4)
< output > ::= **Tensor**		(4.5)
<tensors> ::= **Tensor**|**Tensor** <tensors>		(4.6)
<layers> ::= <layer>|<layer><layers>		(4.7)
<layer> ::= <dense>|<conv>|<flatten>|<merge>		(4.8)
<dense> ::= <output> = **Dense**(units)<input>	{output.taint = dense_taint(neurons_info)}	(4.9)
<conv> ::= <output> = **Conv**(filters)<input>	{output.taint = convolution_taint(neurons_info)}	(4.10)
<flatten> ::= <output> = **Flatten**<input>	{output.taint = flatten(input.taint)}	(4.11)
<merge> ::= <output> = **Merge**<inputs>	{output.taint = merge(inputs.taint)}	(4.12)

Figure 4.7: Taint propagation rules for different types of layers in a feed-forward neural network where terminal symbols in bold, **Terminal**, and non-terminal symbols are in brackets, <non-terminal>. Rule 4.1 initializes the taint for the model's input. The function `new_real_taint()` in this rule sets the taint for the input tensor, and all values in the taint tensor are labeled as real. For dense layers, the taint is determined based on the position of embedded and synthetic neurons, as described by `neurons_info`. The function `dense_taint()` in rule 4.9 assigns a `real` label for outputs corresponding to embedded neurons, and a `synthetic` label for outputs from synthetic neurons. Similarly, for convolutional layers, the function `convolution_taint()` calculates the taint, assigning real to values produced by embedded filters and synthetic to those produced by synthetic filters. In flatten and merge layers, as defined in rules 4.11 and 4.12, the transformations applied to the input taint tensor are the same as those applied to the input tensor itself.

and real values. The synthetic values are discarded, and the real values are fed to the in-TEE submodel.

Figure 4.8 shows the grammar for generating synthetic layers and corresponding taint during cloaking. Initially, the taint tensor corresponding to the input is initialised with `real` label. Function `gen_dense()` and `gen_conv()` samples weights for the given layer and generates the taint tensor based on the position of synthetic and embedded neurons. Other layers, such as Flatten or pooling, apply the same transformation on the taint tensor as they apply to the input tensor.

Figure 4.9 shows an end-to-end MazeNet model for one of the instances of the VGG16 MazeNet model obtained from splitting and cloaking. After splitting VGG16 model, the submodels $S_1, S_2, \ldots, S_n$ were alternatively divided into in-TEE $\{S_1, S_3, \ldots, S_n\}$ and non-TEE $\{S_2, S_4, \ldots, S_{n-1}\}$ submodels. Then, non-TEE submodels were cloaked to produce cloaked submodels. In this instance, the first submodel for cloaking, S_2, consists of three layers: `Conv 2-1`, Conv 2-2, and `Pooling Layers`. During cloaking, one additional layer of `Conv 2-1` type

$$Model \rightarrow \textbf{Model}(input, output, Layers) \qquad \{cloakedModel = model(input, output, Layers)\} \tag{4.13}$$

$$input \rightarrow in = \textbf{Input}(Shape) \qquad \{in.taint = real(Shape);\} \tag{4.14}$$

$$Layers \rightarrow out = \textbf{Dense}(units)(in); Layers \quad \{out.taint, Dense.weights = gen\_dense(in.taint, Dense.weights, \text{synthetic})\} \tag{4.15}$$

$$Layers \rightarrow out = \textbf{Conv}(filters)(in); Layers \quad \{out.taint, Conv.weights = gen\_conv(in.taint, Conv.weights, \text{synthetic})\} \tag{4.16}$$

$$Layers \rightarrow out = \textbf{Flatten}(in); Layers \quad \{out.taint = flatten(in.taint)\} \tag{4.17}$$

$$Layers \rightarrow \epsilon \tag{4.18}$$

$$out \rightarrow \textbf{Identifier} \tag{4.19}$$

$$in \rightarrow \textbf{Identifier} \tag{4.20}$$

$$units \rightarrow \textbf{Int} \tag{4.21}$$

$$shape \rightarrow \textbf{Tuple} \tag{4.22}$$

$$filters \rightarrow \textbf{Int} \tag{4.23}$$

$$weights \rightarrow \textbf{Tensor} \tag{4.24}$$

Figure 4.8: Grammar defining the generation of a cloaked submodel, including input processing, layer transformations, and weight modifications. Terminal symbols are in bold, **Terminal**, and non-terminal in normal text. Rule 4.14 initialises the taint tensor with `real` label through `real()` function which creates a tensor of given shape filled with real label. Function `gen_dense()` and `gen_conv()` samples synthetic parameters and shuffles the sampled parameters with embedded parameters, and returns the new weights along with taint.

is added, followed by three additional layers of `Conv 2-2` type. Finally, a pooling layer is added after `Conv 2-2` type layer. Similarly, the remaining non-TEE submodels were cloaked.

4.4 Running MazeNet Models

A MazeNet model obtained after splitting and cloaking is deployed on a public cloud platform to provide interference services. A Model Manager on the cloud manages the life cycle of the uploaded MazeNet models, from deploying the models to orchestrating the inference workflow.

The Model Manager runs within a TEE enclave and provides API endpoints to manage the deep learning life cycle. First, the model owner attests the Model Manager instance to verify that the cloud vendor is running a genuine instance of Model Manager on a TEE platform. Then, the model owner transfers the MazeNet model to Model Manager through a secure channel such as an SSL connection terminating within the TEE. Then, the Model Manager

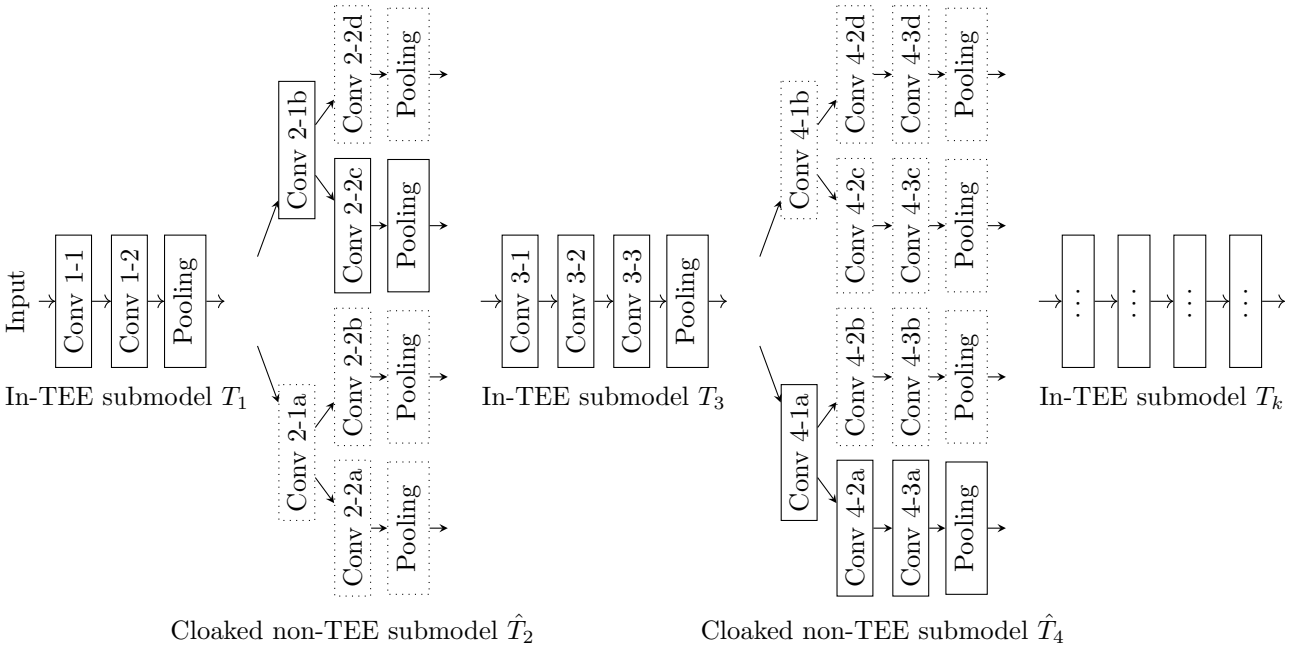


Figure 4.9: One of the instance of VGG16 MazeNet model obtained after splitting and cloaking. Layers with solid rectangles, Layer, in the cloaked submodels are embedded layers from non-TEE submodels, whereas layers with the dotted rectangles, Layer, are synthetic layers.

deploys in-TEE submodels within TEEs and deploys non-TEE cloaked submodels on untrusted runtime environments, and exposes an API to query the deployed models.

Integrity of Cloaked Submodel Evaluation. During the inference process, an adversary on the cloud can observe the cloaked submodels and their evaluation in plain text. It can examine the inputs to the cloaked submodels and its weights, intermediate results, and flow of intermediate results through different layers of the cloaked submodel. Further, an active adversary can alter any data in untrusted environment during any stage of submodel evaluation. It can alter the inputs to the cloaked submodel, change the weights, or replace results computed by different layers to influence the results of the inference process. Moreover, it can perform replay attacks where it replaces the intermediate results with prior results that it observed in earlier queries. For instance, during an image classification task, the adversary queries the model with a cat image and records the intermediate results. Later, during any other query, the adversary can replace the intermediate results with previous state recorded during earlier query. As a result, the last in-TEE submodel will produce confidence scores or labels chosen by adversary even if the adversary does not control in-TEE submodels.

To detect integrity violation during cloaked submodel evaluation, MazeNet relies on digital signatures. The cloud vendor computes a signature $Sign_{sk}(h_i)$, where sk is cloud vendor private

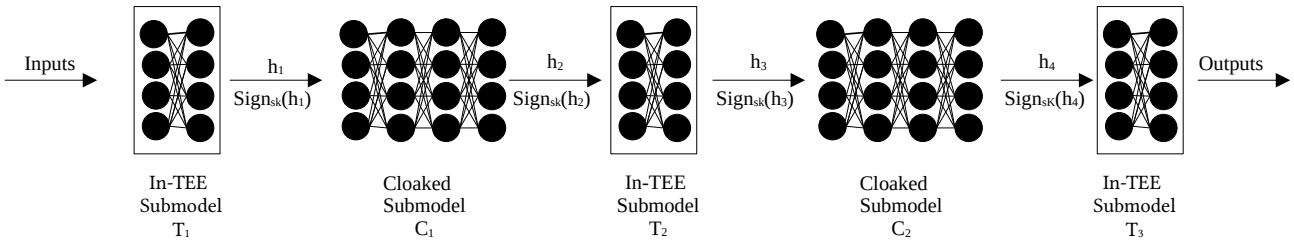


Figure 4.10: During inference, the cloud vendor signs the inputs and outputs of the cloaked submodels with their private key sk. MazeNet can later verify the correctness of outsourced computation by re-executing the computation and verifying the signatures. In addition to the outputs, the cloud vendor submits the input and output signatures to the following enclave, which verifies the validity of the signatures against the outputs.

key, for each input h_i and output h_{i+1} corresponding to each cloaked submodel. The enclave which receives the intermediate output h_{i+1} as input for the in-TEE submodel matches the intermediate outputs and corresponding signatures. On successful verification, the enclave filters embedded outputs from synthetic outputs and feeds the filtered output to in-TEE submodel to proceed with the inference process. The enclave sends both the signatures to the Model Manager for auditing.

The Model Manager can randomly verify some of the outsourced computation by re-executing those computations within a TEE, or outsource them to a non-colluding party, such as other cloud vendors or in-premise execution. For the verification, the Model Manager saves the output of the first in-TEE submodel in secure storage to asynchronously authenticate the outsourced computations. It stores the output of the first in-TEE submodel instead of raw input to reduce privacy leaks as fine details are removed once the input passes through initial layers of convolutional networks [38].

Figure 4.10 shows signature computation in a MazeNet model consisting of three in-TEE submodels and two cloaked submodels. The TEE hosting the first in-TEE submodel saves the submodel output and passes it to the cloaked submodel. The cloud vendor computes the signature with its private key sk and proceeds with submodel evaluation on the inputs. Once the results are available, the cloud vendor again signs the output and sends both the signatures and outputs to the following TEE, which validates the signature with the results. Upon successful validation, the input is filtered and fed to the in-TEE submodel. This process is repeated for each cloaked submodel.

Filtering of embedded and synthetic outputs. After verifying the digital signatures, the enclave filters the embedded outputs from synthetic outputs within the results received from the cloaked submodel. The output of a cloaked submodel contains both synthetic and embedded

outputs due to the presence of synthetic layers and neurons, which were added during the cloaking phase. The taint tensors obtained during the cloaking phase, which contain labels, are securely pre-loaded into the enclave, based on which the enclave filters the embedded outputs from synthetic outputs. After filtering, the embedded outputs are fed into the in-TEE submodel while the synthetic outputs are discarded.

Inference API. To query the model, users send their inputs to the API endpoints exposed by the Model Manager. The model manager scales and crops the user input to adhere to the requirements of the queried model. Then, it routes the user input through an ordered series of in-TEE submodels and cloaked submodels to evaluate the inference task. At the end, the results obtained from the last in-TEE submodel are returned to the user.

4.5 Security Analysis of MazeNet Models

All the in-TEE submodels run within TEEs, therefore in-TEE submodel weights are secured by the hardware from adversaries. In the case of non-TEE submodels, each non-TEE submodel $\hat{T} \in \hat{\mathfrak{T}}$ is cloaked to get corresponding cloaked submodel $C \in \mathcal{C}$. To steal the weights of the embedded non-TEE submodel $\hat{T}$ from cloaked submodel C, the adversary needs to correctly identify embedded layers and neurons from synthetic ones. Equivalently, the adversary can steal the submodel if it can correctly guess the synthetic outputs from the embedded outputs, which are filtered by the enclaves.

Each subset of the output from the cloaked model corresponds to a unique model. However, only one of them corresponds to the embedded model. Thus, the problem of submodel stealing for the adversary reduces to correctly identifying a subset from all possible subsets of a given set. For a set of size $|S|$, there are $2^{|S|} - 1$ possible subsets, excluding the empty subset. Therefore, if a cloaked submodel produces N tensors as outputs, then the probability that an adversary can correctly guess the embedded non-TEE submodel $\hat{T}$ is

$$P(\hat{T}) = \frac{1}{2^N - 1}$$

For the entire model, the number of expected weights that would be presented in a randomly extracted submodels would be the sum of expected weights in the individual cloaked submodel. Therefore, the number of weights present randomly extracted model by the adversary would be:

$$E[\text{Embedded Weights}] = \sum_{C \in \mathcal{C}} \frac{|W_C|}{2^{||C||} - 1}$$

Here, $|W_C|$ is the number of embedded weights or parameters present in cloaked submodel C, and $||C||$ is the number of output tensor produced by the cloaked submodel C.

4.6 Implementation

We have built the MazeNet framework on TensorFlow [48]. It consists of two components: MazeNet Model Builder and Model Manager.

A model owner runs MazeNet Model Builder on its premises to build a MazeNet model for a given pre-trained model. As enclaves have memory constraints, MazeNet uses TensorFlow Lite framework for running in-TEE submodels. However, the untrusted environment does not have such constraints, so cloaked models are deployed with the standard TensorFlow framework. Model Builder accepts pre-trained models in TensorFlow `SavedModel` format, and produces MazeNet models consisting of in-TEE and cloaked submodels. The in-TEE submodels are exported in `TFLite` format, and the cloaked submodels in `SavedModel` format. In addition to the MazeNet model, the Model Builder produces taint tensors that are required during the inference process. The in-TEE submodels and taint tensors are confidential; therefore, they are delivered to the enclave over a secure channel.

On the cloud, Model Manager manages the life cycle of MazeNet models. It is responsible for securely deploying in-TEE submodels within the TEEs and cloaked submodels in untrusted environments. It exposes an API endpoint to query the model and orchestrates the inference process.

Our implementation relies on Intel SGX as a TEE. As applications do not run out-of-box on Intel SGX, the research community and industry have developed multiple frameworks to run applications on SGX. These include Graphene-SGX [155], Porpoise [132], Panoply [136], SGX-LKL [118]. Among these, we have selected Graphene-SGX as it supports the Python programming environment and the TensorFlow deep learning framework.

To implement the MazeNet framework, we have added cloaking support for popular layers in the TensorFlow framework which were present in our benchmark models – VGG16, ResNet50, and DenseNet201. Table 4.3 lists the various types of layers present in each model. In total, TensorFlow has around 150 types of layers, and we implemented cloaking support for 20 TensorFLow layers for the benchmark models.

In the implementation, adding cloaking support for TensorFlow layers required 715 lines of Python code in Model Builder, and around 510 lines for Model Manager as reported by `pygount` [5]. Further, MazeNet can be easily extended to other models by implementing cloaking for unsupported layers present in the model.

Model	Types of TensorFlow Layers Present in Model
VGG16	Conv2D, Dense, MaxPooling2D, Flatten
ResNet50	Conv2D, Dense, BatchNormalization, Add, GlobalAveragePooling2D, MaxPooling2D, ZeroPadding2D, Activation
DenseNet201	Conv2D, Dense, BatchNormalization, Concatenate, GlobalAveragePooling2D, MaxPooling2D, ZeroPadding2D, AveragePooling2D

Table 4.3: Different types of TensorFlow layers present in the benchmark models.

4.7 Evaluation

To find the benefits and costs of the presented techniques, we transform popular convolutional neural networks into MazeNet models and evaluate the performance of the MazeNet models. Our evaluation focuses on two key aspects of deep learning inference services: throughput and latency. Furthermore, we quantify the overheads arising from different sources in the MazeNet inference system.

Benchmark models. For a comprehensive evaluation, we identified networks with diverse architectures to cover a broad range of models. For the sequential models, where the output of one layer is input to the immediately following layer, we have selected the VGG16 model [74]. In the case of functional models, where the output can be input to more than one following layer, we have taken the ResNet50 model [56]. Further, for the highly connected architectures, we have chosen the DenseNet201 model [59], where a layer takes outputs of all preceding layers within a block as inputs.

4.7.1 Experimental Setup

MazeNet models consist of two types of submodels: in-TEE submodels and cloaked submodels. During inference, the in-TEE submodels are deployed on TEE-equipped systems, while the cloaked submodels are deployed on non-TEE systems.

1. TEE systems: The in-TEE submodels obtained from splitting are deployed on TEE systems. Based on the parameters in Table 4.4 to split models, the Model Builder produces three in-TEE submodels and two cloaked submodels. The in-TEE submodels are deployed on three TEE systems that are equipped with an Intel i7-7700 desktop-class CPU, which supports SGXv1 with 128 MB of cryptographically protected memory, and 32 GB of main memory.

2. Non-TEE systems: The remaining two cloaked submodels are deployed on non-TEE systems without SGX support. The cloaked submodels are provisioned on a server-class

Xeon Gold 6150 CPU, having 36 cores and 72 threads, along with 256 GB of main memory. As the server-class machine has a high core count and ample main memory, we deploy all the cloaked submodels on the same machine.

Secure baseline. To evaluate the performance of MazeNet models, we compare them against a secure baseline. The secure baseline system is the same TEE (i7-7700) system described above for MazeNet. The secure baseline system executes unmodified pre-trained DL models within an Intel SGX enclave. It can run models that are larger than the size of the protected memory with EPC swapping at the cost of a performance penalty. We do not have a secure baseline for the Non-TEE system (Xeon 6150) described above for MazeNet, as it does not support any trusted execution environment.

Non-secure baseline. Similarly, we have two non-secure baseline systems where the entire unmodified model runs within a single system. The two non-secure baseline systems consist of the TEE (i7-7700) and non-TEE (Xeon 6150) systems described above for MazeNet.

Generating MazeNet models. As pre-trained model weights are required to generate MazeNet models, we have used the trained models provided by the Keras library [28]. Keras is a library built on top of TensorFlow [48] to enable developers to quickly build and ship DL models. It contains a few popular models along with pre-trained weights. In Keras, the model architecture is stored as a TensorFlow computation graph, and weights are stored in a custom `SavedModel` format. MazeNet Model Builder takes a model stored in `SavedModel` format as input, along with splitting and cloaking parameters, to build a corresponding MazeNet model.

In MazeNet, the model developer provides the set of synthetic layers to be added during the cloaking phase. For evaluation, existing layers of the submodels were duplicated and added to the submodels to produce cloaked submodels.

There are three key configuration parameters (model split, cloak factor, and submodel width) for building a MazeNet model, which influence the performance of the generated model. Table 4.4 lists the configuration parameters used in our evaluation for building MazeNet models.

1. **Split** states how the given model should be split into smaller submodels. For instance, the VGG16 model consists of 16 trainable layers (convolutional layers and dense layers) and 6 non-trainable layers (pooling layers and flatten layers). According to the Table 4.4, layers $1, 11$ and 22 of the VGG16 model are designated as in-TEE layers. Based on this, the Model Builder produces three in-TEE submodels: $S_1 = \{L_1\}, S_2 = \{L_{11}\}, S_3 = \{L_{22}\}$, while the remaining layers $\{L_2, \ldots, L_{10}\}, \{L_{12}, \ldots, L_{21}\}$ are be part of two cloaked submodels.

Model	Cloak factor	Cloaked Submodel width	In-TEE layers
VGG16	10%	10	1, 11, 22
ResNet50	10%	10	1-7, 92-102, 174-177
DenseNet201	10%	10	1-7, 49-137, 477-709

Table 4.4: Configuration parameters used to generate MazeNet models for the benchmark models considered in the evaluation.

2. **Cloak factor** represents the percentage of synthetic parameters or weights (neurons in dense layers and filters in convolutional layers) to be added to build a cloaked layer. Consider the layer 2 of the VGG16 model, which is a convolutional layer with 64 filters, and a cloak factor of 10%, from the Table 4.4, will add 10% synthetic filters. Therefore, the cloaked convolutional will have 72 filters instead of 64.

3. **Cloaked submodel width** limits the maximum width during cloaking. When a synthetic layer is added at depth d, it increases the width of the submodel, where the width is the number of layers present at a given depth. An increase in the width of the cloaked submodel corresponds to extra computations and higher data transfer costs during inference. Thus, this parameter limits the maximum width of cloaked submodels.

Model deployment. Once a MazeNet model is built, an instance of Model Manager loads the in-TEE submodels on the TEE systems and the cloaked submodels on the non-TEE systems. The Model Manager exposes an API endpoint for the user to query the deployed model with inputs and obtain prediction results. When a user queries the MazeNet model with inputs, the Model Manager routes the inputs through a series of in-TEE and cloaked submodels, ultimately computing the inference results and returning the prediction scores or labels to the user.

The first query to the models after model deployment takes an inordinate amount of time to return the inference results. In the first query, internal states are set up in the TensorFlow framework to store the intermediate states. Therefore, the results from the first query are excluded from the measurements.

Inference task. The inference service hosting models performs image classification on images drawn from the ImageNet [126] dataset. The ImageNet dataset consists of curated high-resolution images of different sizes that were collected from the internet. Furthermore, the images are labelled into 1,000 categories. As the benchmark models accept fixed-size inputs, the images are cropped and scaled before feeding them to the model under evaluation. The models compute confidence scores on inputs that assign the probabilities of the input belonging to 1,000 categories.

Clients. In real-world deployments, a single model would handle requests from multiple users. To evaluate how MazeNet models perform under such conditions, we measured their throughput and latency when queried by concurrent clients running as processes. Each client instance simulates a unique user and sends one query at a time, waits for the response before issuing the next.

Batch size. In standard deep learning workloads, models are queried in batches instead of a single input instance. In batching, multiple individual inputs are grouped to form a batch, and the model performs the evaluation on the whole batch in one forward pass. Batching benefits from the parallelism available on modern hardware, where multiple data and instructions are executed in parallel.

4.7.2 Throughput Results

First, we evaluate the throughput of MazeNet models and compare them against the secure baseline models. The MazeNet models are generated from the configuration parameters given in Table 4.4 and deployed as per the experimental setup in Section 4.7.1.

To evaluate throughput, we query the models with 128 input samples that are split across eight clients, which query the models in parallel. We repeat this experiment with different batch sizes and report the maximum throughput, across batch sizes, for each model in Figure 4.11. The results demonstrate that MazeNet models achieve higher throughput when compared to the secure baseline models. MazeNet models benefit from the faster untrusted processors, whereas the secure baseline models are limited by the weak computational power of the trusted CPU.

However, the speedup observed across models differs significantly from 30x in VGG16 to 2x in DenseNet201. The speedup is more significant in VGG16 as it is computationally expensive, requiring 30.96 GFLOPs as compared to ResNet50 and DenseNet201 models, which require 7.73 GFLOPs and 8.58 GFLOPs, respectively. Thus, the VGG16 MazeNet model benefits more from powerful untrusted processors. Further, the speedup will vary with the computational power of the untrusted hardware.

During the experiments, we observed that each MazeNet model achieved maximum throughput at a different batch size. Therefore, we next measure the role of batch size in the throughput of MazeNet models. In standard deep learning workflows, increasing the batch size increases the throughput of the system. However, the throughput saturates at some batch size.

To measure the impact of batch size, eight clients query the model in batches. The clients draw multiple input samples from the ImageNet dataset, form a batch of a given size, and query the model with the batch. The experiment is repeated for different batch sizes, and the results are presented in Figure 4.12. Indeed, increasing the batch size improves the throughput

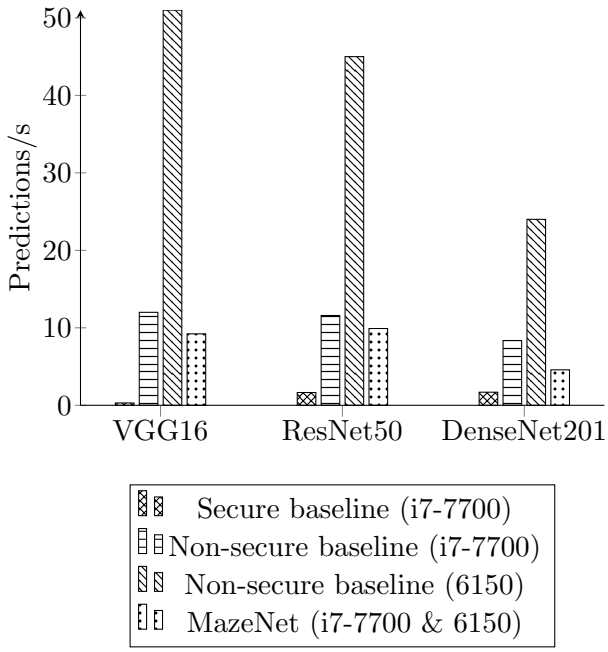


Figure 4.11: Maximum throughput observed in secure baseline models and MazeNet models, where MazeNet models were generated from the parameters given in Table 4.4.

of MazeNet models. The VGG16 and DenseNet201 models achieve the highest throughput at a batch size of eight, whereas ResNet50 achieves maximum throughput at a batch size of four.

However, increasing the batch size beyond the optimal point leads to a significant decline in throughput, whereas throughput saturates in standard deep learning workloads. The primary reason for this degradation of throughput is EPC swapping in enclaves due to the large intermediate state produced by in-TEE submodels. Doubling the batch size doubles the size of the intermediate state produced by layers and submodels. This EPC swapping of intermediate states is different from the earlier EPC swapping of model weights, where the weights of large DL models do not fit within the protected memory and incur swapping. Therefore, selecting the right batch size becomes crucial for maximising the throughput of MazeNet models.

Finally, we evaluate how the throughput scales with increasing inference workload. Initially, a single client queries the model with a fixed batch size of one. Then, the number of clients is progressively increased over a period of time, while keeping the batch size fixed, to simulate the increasing workload.

Figure 4.13 shows the throughput of MazeNet models at varying workloads. Initially, the overall throughput of the MazeNet inference system increases with an increase in workload. However, the throughput saturates when the number of clients crosses six. In MazeNet, the

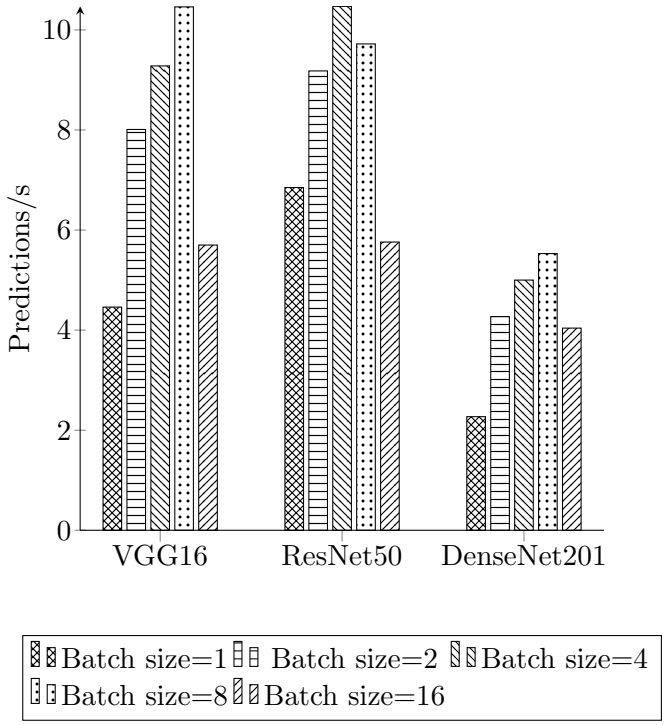


Figure 4.12: Throughput of MazeNet models at different batch sizes. Increasing the batch size improves the throughput of MazeNet models. However, increasing the batch size beyond a certain point reduces the throughput due to EPC swapping as the enclaves cannot store the entire intermediate state in the protected main memory.

submodels are deployed on different systems. When the Model Manager receives an input and routes it through a sequence of submodels, only one submodel is actively processing the input, while the other submodels are idle as they have either already processed the input or waiting for the intermediate results to begin processing. As the number of clients increases, the Model Manager schedules the pending queries in pipelined mode, where the next query is scheduled for execution on the first submodel, while the second submodel is still processing the earlier query. As the workload keeps increasing, the throughput saturates when all the submodels are actively processing different queries, or one of the submodels becomes the bottleneck in the pipelined execution.

From the previous two experiments, the throughput of models increases with both the increase in batch size and the increase in the number of clients. The system achieves maximum throughput when the batch size is around four to eight, while the number of clients is around six to eight. This set of parameters results in maximum utilisation of system resources to obtain the highest throughput.

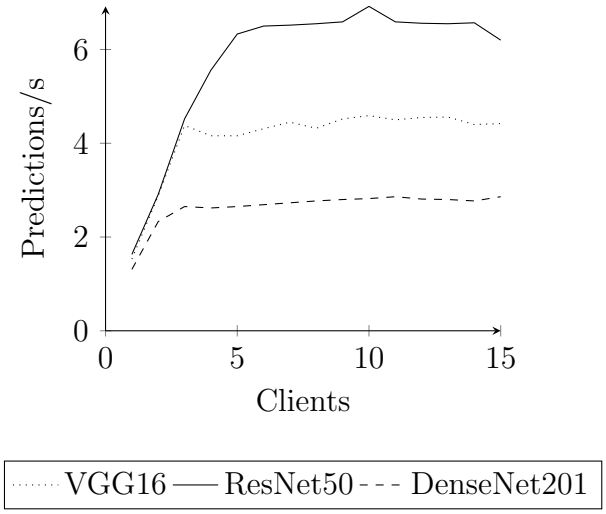


Figure 4.13: Throughput of MazeNet models at different workloads. The x-axis represents the number of clients simultaneously querying the model with a fixed batch size of one. As the number of clients increases, the workload applied to the inference services increases.

4.7.3 Latency Results

Next, we shift our focus to query latency of models, which indicates the user experience. A lower latency results in a responsive session for the user.

In our experiments, query latency is the time duration a client has to wait, after sending the inputs to the model, until it receives inference results. To measure the latency of models, we query the model with a single client that sends a single input (batch size=1) to the model and waits for results before sending the next query. The client measures the time between sending the inputs and obtaining results. The experiment is repeated 100 times with different inputs, and the average latency is reported in Figure 4.14. The results show a significant improvement in latency for the VGG16 MazeNet model as compared to the secure baseline model, where the latency drops from 3 seconds to 0.6 seconds, a 5x reduction in latency.

However, there is no latency improvement for the ResNet50 and DenseNet201 MazeNet models. There are two main sources that contribute to the latency of MazeNet models, in addition to computational operations. First is the time spent on computing the digital signatures by cloaked submodels and verification of signatures by enclaves. Second is the time spent on data transfers between cloaked and in-TEE submodels. As compared to the VGG16 MazeNet Model, the ResNet50 and DenseNet201 MazeNet models spend more time on computing digital signatures and data transfers than performing deep learning operations, which can be accelerated by untrusted hardware.

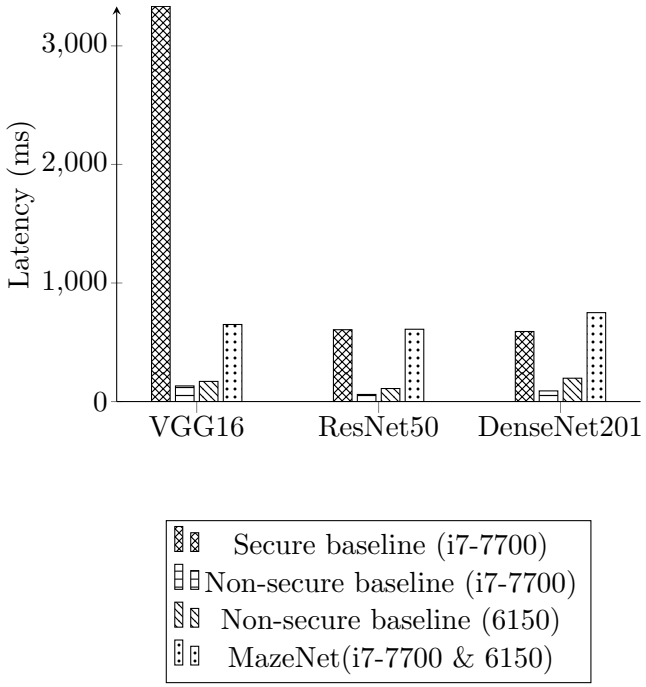


Figure 4.14: Latency of baseline and MazeNet models when a single client queries the model with a single input image. The Y-axis denotes the time (milliseconds) the client has to wait for the results.

Latency with multiple Clients. Next, we evaluate how the latency is impacted when multiple clients simultaneously query the models to simulate the scenario where a single model handles multiple users. Initially, we began with a single client which queries the model with a single input. Then, progressively, we increase the number of clients over time.

Figure 4.15 presents the latency trends of MazeNet models under varying numbers of clients. Initially, the latency increases slightly when the number of clients is increased from one to five. As the number of clients increases, the number of active models in the inference pipeline increases. However, when the pipeline is full or saturated, further increasing the number of clients proportionally increases the latency as the new input queries are put in the wait queue by the Model Manager.

Latency at different batch sizes. Next, we investigate the impact of batch size on the latency of MazeNet models. We measure the latency of MazeNet models at different batch sizes while keeping the number of clients constant (one) to avoid queueing by the Model Manager. Initially, the client sends the query containing a single input image and records the time taken by the model to return the results. Then, it gradually increases the number of input samples in the batch.

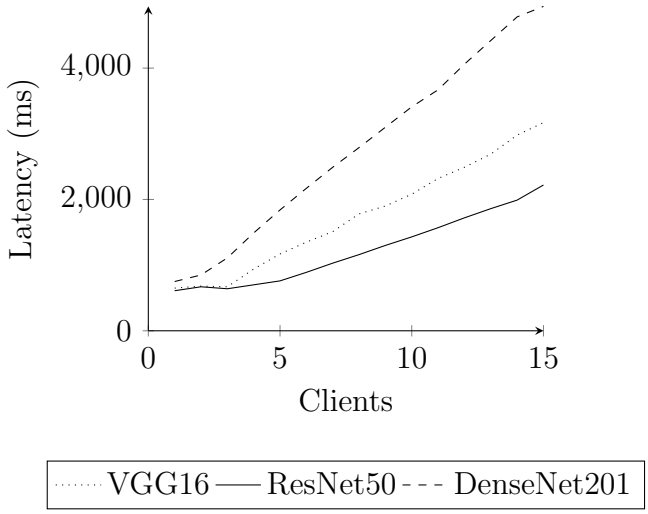


Figure 4.15: Latency of MazeNet models when they are queried by multiple clients in parallel. The x-axis denotes the number of clients simultaneously querying the model, while the y-axis shows the average latency observed by each client.

Figure 4.16 reports the latency for the entire batch. For all the models, the latencies at different batch sizes follow the same pattern. The batch latency increases with an increase in batch size. However, the average time spent per input sample within a batch reduces from 0.60 seconds to 0.30 seconds when the batch size is increased from one to fifteen. Similarly, it decreases from 0.56 seconds to 0.40 seconds for ResNet50, and from 0.71 seconds to 0.41 seconds for DenseNet201. The results show that batching can reduce the average time spent per single input instance during inference, while the overall latency increases due to the higher time spent in evaluating multiple inputs in the batch.

4.7.4 Overheads

During the MazeNet inference, there are three key sources of overhead when compared to standard inference without any splitting and cloaking. First is the overhead of executing in-TEE submodels within SGX enclaves, as applications run slower within enclaves due to the overheads intrinsic to SGX enclaves, e.g., domain crossing, copying data to and from enclave memory. Second is the overhead of network communication costs arising from data transfers between the in-TEE and cloaked submodels that are located on different hosts. Third is the additional computations due to the presence of synthetic layers and synthetic neurons that were added in the cloaked submodels during the cloaking process. In this section, we will quantify individual overheads of these three sources and discuss a few optimisations that can help to reduce these overheads.

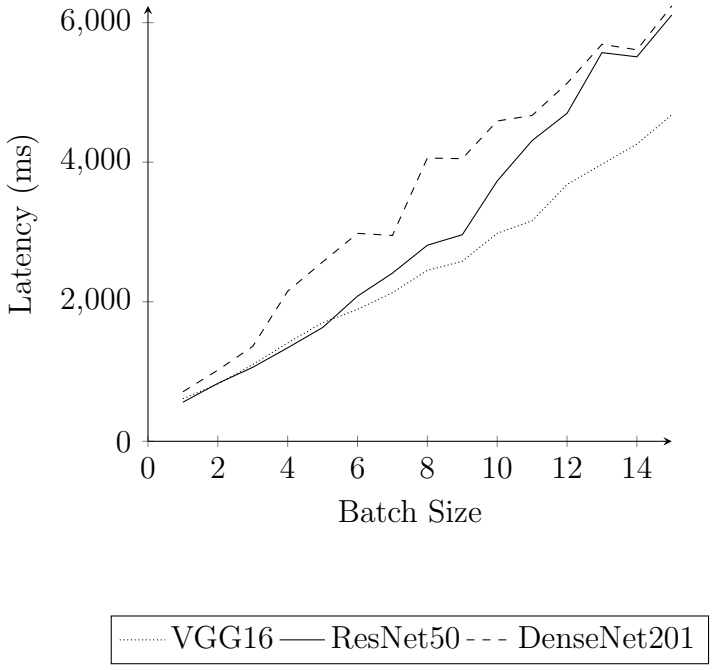


Figure 4.16: Latency of MazeNet models at different batch sizes, where they are queried from a single client. The y-axis reports the latency for the entire batch.

Overhead of enclave execution. Applications run slowly within enclaves due to the overheads intrinsic to enclave execution, such as a memory copy of data between the enclave and the user space memory, entry and exit of control flow to the enclaves, and other overheads. In MazeNet, the intermediate data is frequently transferred across submodels, which necessitates memory copy and context switches for enclave processes.

To quantify the overhead of enclave execution, we measure the throughput of MazeNet models in two configurations: in-TEE submodels within TEEs and in-TEE submodels outside TEEs. In the first configuration, we run MazeNet models in the standard configuration as per the experimental setup described in Section 4.7.1. In the second configuration, in-TEE submodels are deployed in an untrusted environment to avoid the overheads of enclave execution, while the remaining setup remains the same. We measure the throughput of the VGG16 MazeNet model in both configurations at multiple batch sizes from eight clients.

Figure 4.17 plots the throughput of the MazeNet model in both configurations, in-TEE submodels within the TEEs and outside the TEEs, for the VGG16 MazeNet model. The throughput of standard configuration, in-TEE submodels with TEEs, is within 20% of the other configuration for batch sizes up to eight, as the throughput is bottlenecked by cloaked submodel execution time instead of the enclave execution. However, at higher batch sizes, EPC

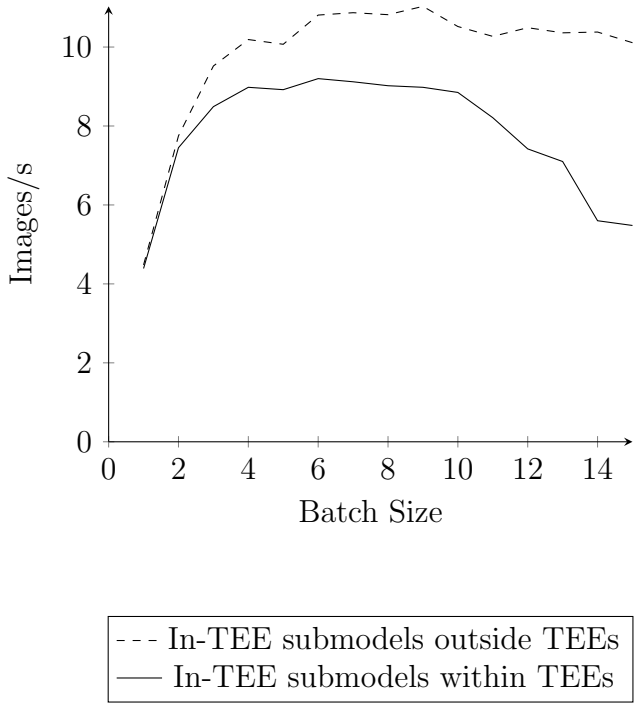


Figure 4.17: **Overhead of enclave execution.** Throughput of the VGG16 MazeNet model when the in-TEE submodels are deployed within TEEs v/s outside the TEEs. Deploying the in-TEE submodels outside TEEs avoids enclave execution costs. The overhead of enclave execution is low at small batch sizes, but it increases with the batch size.

swapping occurs due to the larger intermediate state produced by the in-TEE submodels, which shifts the bottleneck from the cloaked submodel to EPC swapping in enclaves. Consequently, the throughput of in-TEE submodels within the TEEs configuration starts to decrease with higher batch sizes. Thus, the overhead of enclave execution is more prominent when there is EPC swapping, while the overhead is minimal without EPC swapping.

Network Overhead. Intermediate results produced by the in-TEE and cloaked submodels are transferred over the network, which introduces network overhead during the inference process. We evaluate the overhead due to the 1 Gigabit Ethernet network used in our experimental evaluation.

To isolate network overhead, we run both the in-TEE and cloaked submodels on the same non-TEE system, the server machine described in the experimental setup, Section 4.7.1. This setup eliminates the need for network transfers during the inference process. As the in-TEE submodels run outside the enclave, similar to the previous experiment, we can offset the performance gains arising from running in-TEE submodels outside the enclave, up to 20% for smaller batch sizes.

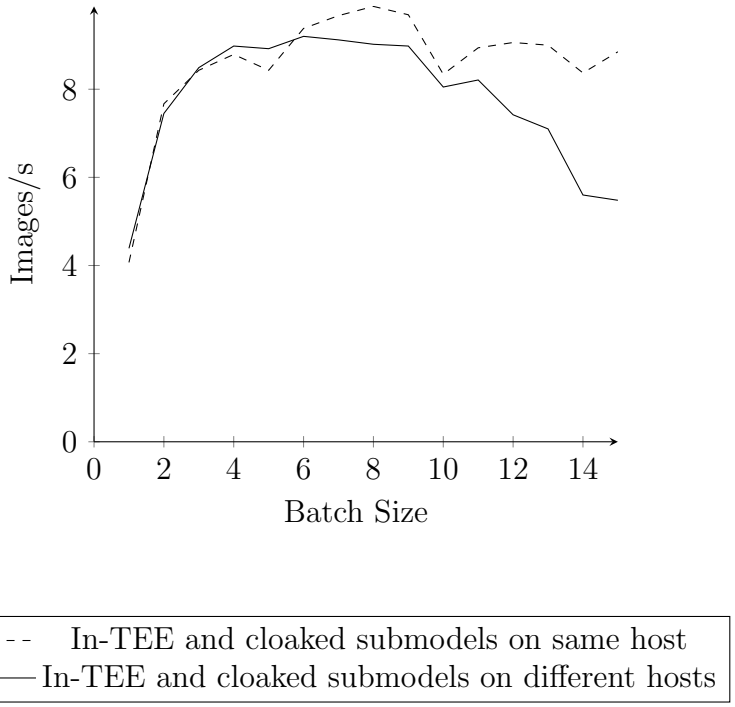


Figure 4.18:
Network overhead. Throughput of the VGG16 MazeNet model when the in-TEE and cloaked submodels are deployed on the same host to avoid network transfers.

We compare the throughput of the VGG16 MazeNet model in the above new configuration, when the in-TEE and cloaked submodels are co-located on the same host, with the standard configuration, where the in-TEE submodels are located on TEEs on different hosts. The throughput in both cases is similar, as the inference process is compute-bound. During inference, submodels of the VGG16 MazeNet model transfer around 10 MB of data per single input inference. The MazeNet model in standard configuration achieves around eight inferences per second; therefore, the network bandwidth of one Gigabit does not introduce any significant overhead.

However, with more powerful untrusted processors, the network bandwidth may become the bottleneck. One of the optimisation strategies to reduce the amount of data transfers is to reduce the precision of the intermediate results when they are transferred across submodels, while the submodel performs the remaining computation on standard precision. But reducing the floating point precision may reduce the accuracy. To evaluate the drop in accuracy, we measured the accuracy when the floating point precision was reduced from 32 bits to 16 bits. For the VGG16 MazeNet model, the top-1 accuracy remains the same at 68%, although the

Model	Standard	MazeNet Model			Increase
		in-TEE Submodels	Cloaked Submodels	Total	
	(GFLOPs)				
VGG16	30.96	1.85	151.76	153.60	5x
ResNet50	7.73	0.684	60.85	61.54	8x
DenseNet201	8.58	3.17	51.18	54.36	6x

Table 4.5: Number of Floating-Points Operations (FLOPs) in standard benchmark models and Mazenet models when the models were cloaked with the configuration parameters in Table 4.4.

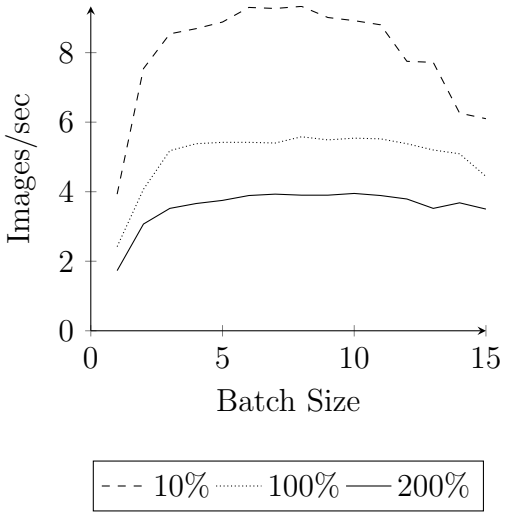
raw confidence scores differ slightly due to the drop in precision.

Overhead of additional computations.: The synthetic layers and neurons added during the cloaking phase to produce the cloaked submodels add additional computations that were not present in the original model. To quantify the overhead due to the additional computations, we compare the number of floating-point operations required to compute inference results in unmodified models and MazeNet models.

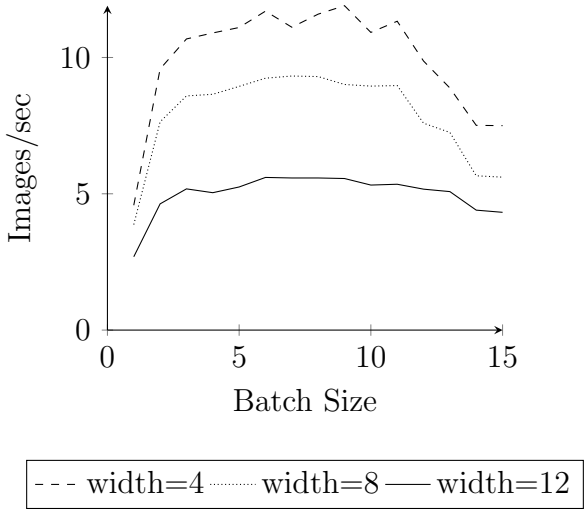
Table 4.5 lists the number of floating point operations (FLOPs) in unmodified and MazeNet models, which were generated from the parameters given in Table 4.4. The number of floating-point operations increases by 5x to 8x in MazeNet models as compared to unmodified models. However, the additional operations in cloaked submodels execute on faster and powerful untrusted hardware, which is an order of magnitude faster than the TEEs, 4 cores on the TEE system v/s 36 cores on non-TEE systems in our experimental setup. Other hardware accelerators, such as GPUs and TPUs, can even provide up to 10x to 100x more computational power than CPUs.

The additional operations can be further broken down into operations from the synthetic neurons and synthetic layers. Thus, we evaluate how different amounts of synthetic neurons and synthetic layers affect the throughput of MazeNet models by adding different amounts of synthetic layers and neurons in the MazeNet model.

Overheads due to synthetic neurons. First, we focus on how additional synthetic neurons affect the throughput of MazeNet models. Specifically, we assess the change in throughput when varying the proportion of synthetic neurons (10%, 100%, and 200%) during the cloaking phase, while keeping all other parameters constant with those listed in Table 4.4 to generate the MazeNet models. Figure 4.19a illustrates the throughput of VGG16-based MazeNet models across different batch sizes for each configuration. The throughput halves when the amount of synthetic neurons is increased from 10% to 100%. The throughput further drops by an



(a) Throughput of the VGG16 MazeNet models when different amounts of synthetic neurons are added in the cloaked submodels.



(b) Throughput of VGG16 MazeNet models when different amounts of synthetic layers are present in cloaked submodels.

Figure 4.19: Throughput of the VGG16 MazeNet model when different amounts of synthetic computations were added during the cloaking phase by adding synthetic neurons and layers.

additional 25% when the amount of synthetic neurons is increased again from 100% to 200%. These findings demonstrate the rate at which the throughput of MazeNet models declines as the proportion of synthetic neurons increases.

Next, we evaluate the impact of synthetic layers on the throughput of MazeNet models. During the cloaking phase, different numbers of synthetic layers are added to obtain MazeNet models. Figure 4.19b shows the throughput of three models, each having different amounts of synthetic layers. Both the synthetic neurons and the synthetic layers increase the number of computations in cloaked submodels and increase the amount of data transfer from cloaked submodels to in-TEE submodels. Therefore, the throughput of MazeNet models is affected based on the cloaking parameters, which control the number of synthetic neurons and synthetic layers to be added during the cloaking phase.

Overhead of digital signatures. Apart from synthetic computations, the input and output of the cloaked submodels are signed by the cloud vendor's private key, and the signatures are verified within the enclave. We measured the overhead due to signature computations. The throughput remains largely unaffected as the signature computations are overlapped with submodel evaluation. However, the latency in the VGG16 model drops from 0.60 seconds to 0.30 seconds without digital signatures.

4.8 Limitations

There are a few limitations of this work. First, MazeNet only protects the privacy of the models and does not protect the privacy of the inputs. When the input image is processed by the first few convolution layers of the first in-TEE submodel, the intermediate output sent to the cloaked submodel located outside the enclave may reveal high-level details, e.g., the user input to the model is an image of an animal. Second, the developer needs to provide the optimal split and the synthetic layers' architecture. A random split of the model can adversely affect the performance due to higher data transfer across submodels or an imbalance in the number of operations across submodels, which causes one of the models to become a bottleneck in the pipelined execution, leaving other submodels idle or underutilized.

4.9 Protecting DNNs With Other TEEs

In the place of Intel SGX, other TEEs can be used for running deep learning inference workloads. VM-based TEEs, Intel TDX, AMD SEV-SNP, and Arm CCA, do not suffer from fixed protected memory as was the case with SGX. Therefore, splitting of the model is not required. However, they still cannot securely use general-purpose hardware accelerators. Therefore, in those cases, they can use MazeNet to offload a portion of computations to untrusted computing devices to meet the performance requirements. The VM-based TEE can host the set of in-TEE submodels, while the cloaked submodels are outsourced to untrusted devices to speed up the inference process.

MazeNet, however, is not suitable for TEEs on mobile devices, Arm TrustZone and Arm CCA, as deep learning operations are computationally expensive and rapidly drain the device's battery. Further, transferring data across the mobile network will hamper the inference performance. Instead, the model should be optimized with techniques like quantization and model distillation [50] to reduce the cost and the number of DL operations.

In the case of recent high-end server-grade Nvidia GPUs that are available with confidential compute capability, DNNs can be directly deployed on these GPUs within the GPU-TEE, while the VM-based CPU TEE can orchestrate the DL model provisioning and expose the APIs required to query the model. In the case of older GPUs, where the TEE is not available, or in the case of GPUs from other manufacturers, MazeNet can be used to speed up the inference task.

4.10 Conclusion

In this work, we presented MazeNet to protect the privacy models on public cloud platforms with TEEs, and introduced methods to outsource portions of computation during inference to untrusted hardware. Our outsourcing scheme outsources both the linear and non-linear layers. We implemented the presented techniques in a prototype framework, MazeNet, to build MazeNet models from given pre-trained models and deploy the MazeNet models on a public cloud platform to provide inference services. Our evaluation of popular convolutional networks demonstrates that MazeNet models can improve the throughput by up to 30x and the latency by up to 5x as compared to the secure baseline models considered in our evaluation.

Chapter 5

Related Works

5.1 Extensions of TEEs

The research community has proposed a plethora of solutions to extend the scope of trusted execution environments beyond traditional CPU-based TEEs to encompass GPUs and other discrete and integrated peripheral devices. Some of the work further isolates various components within trusted compute base, thus reducing the attack surface and enhancing the security.

Sanctuary [20] presents a new security architecture that leverages ARM TrustZone to run applications in an isolated environment in the normal world with assistance from TrustZone Address Space Controllers [12]. It protects the applications, referred to as Sanctuary Apps, in this isolated environment from the malicious operating system and as well as protects the operating system from malicious Sanctuary Apps. Moving applications from the secure world to the normal world into an isolated environment reduces the trusted computing base of the secure world and reduces the attack surface. Another work vTZ [58] proposes to virtualise ARM TrustZone to enable multiple virtual machines to have separate TEEs.

StrongBox [35] proposes an isolated execution environment for Arm mobile integrated GPUs by leveraging the hardware-based memory protection offered by ARM TrustZone. It allows a GPU to be configured in secure mode such that the host hypervisor cannot access GPU data. It uses the Arm TrustZone Address Space Controller to protect GPU memory from untrusted DMA devices and extends stage-2 address translation to protect GPU memory from the hypervisor and kernel.

ACAI [141] presents an Arm CCA-based design to extend the trusted execution environment of realm VMs to PCIe devices such as discrete GPUs and FGPAs. It uses existing ARM CAA-aware system buses and memory protections to protect communication between realm VM and PCIe devices. A device can be allocated to a realm VM to work in realm mode during realm

creation, where the host hypervisor and other realm VMs cannot access real-mode devices. It uses existing memory encryption to protect confidentiality during data transfers over PCIe.

Targeting ARM mobile SoCs, Portal [129] introduces a secure device I/O mechanism that avoids the overhead of memory encryption. Instead, it relies on the memory isolation features of ARM CCA to establish a protected plaintext memory region that is accessible only to the Realm VM and designated devices. This design enables efficient and secure I/O operations without compromising confidentiality.

Aster [75] explores mutual isolation between the secure world and the normal world. While TrustZone prevents normal-world access to secure-world resources, Aster complements this by employing ARM CCA's Granular Protection Checks (GPC) to prevent secure-world access to the normal world.

5.2 Frameworks to Ease Enclave Development

The key pieces of work related to our own are of course the frameworks used to port legacy applications to enclaves: Haven [16], Graphene-SGX [155], Panoply [135], LKL-SGX [118], SCONE [13] and lxcsgx [149]. Since we have discussed these projects at length, we will focus our discussion in this section on other related work.

Porting applications to enclaves. Prior work has ported applications in several specific domains to enclaves. These include frameworks for MapReduce tasks [131], language environments for JavaScipt [46] and Rust [36], BitCoin and Blockchain applications [154, 168], in-memory databases [117], object stores [73], and middleboxes [151]. SCONE [13] and lxcsgx [149] use the instruction-wrapping model to port containers (Docker in case of SCONE and lxc containers in case of lxcsgx) into enclaves.

Each of these projects focused on providing an enclave version of a *specific* application or class of applications. Our focus, in contrast, is on generic frameworks that can be used to port any application to enclaves. Naturally, because the projects cited above are tailored to individual applications, we expect the resulting enclaves to perform better than applications ported using the generic frameworks. Since they are tailored from scratch for specific application domains, they also are better engineered to just run the sensitive portions of the applications within the enclaves, rather than the entire application, as we did in this paper. Thus, we view the frameworks discussed in this paper as stop-gaps, that can be used to get enclave applications up and running quickly, as developers work on rewriting the applications to tailor them to enclaves.

Finally, there is also work on creating secure in-enclave file systems [100, 6]. The main goal here is to ensure that file accesses that cross the enclave boundary do not reveal any information

about the data being accessed inside the enclave. These file systems use techniques inspired from the oblivious-RAM literature to provide such guarantees. The frameworks discussed in this paper do not offer such guarantees, and may leak information (*e.g.,* about the files accessed, the number of bytes read) even if the data is stored encrypted in the files. However, they can be integrated with such ORAM-inspired file systems to provide stronger guarantees.

Tool support to write enclave code. Glamdring [85] and Civet [156] offer tool support to automatically partition a legacy application into an enclave and non-enclave part. An application developer provides a specification of the portions of the code/data that are sensitive. These tools perform static taint analysis of the application to identify data and control dependencies, and identify the enclave boundary. Following this, they automatically partition the code and create the enclave; they rely on one of the models described in this paper as the enclave execution framework.

Researchers have also developed tools to help developers build secure enclaves from scratch. The key consideration for these tools is to ensure that enclaves do not accidentally leak sensitive data to untrusted code. Sinha *et al.* [140] developed a programming framework that ensures information-release confinement, *i.e.,* that cleartext data never leaves the enclave. To do this, they provide a verified, in-enclave trusted library and a simple API consisting of just a few simple calls (`send`, `recv`) to interact with non-enclave code. Provided that an application developer adheres to this simple interface, they can guarantee that the enclave will not accidentally leak data to untrusted code. Moat [139] is a similar analysis tool that analyzes the machine code of enclaves and determines whether there are any unintended information leaks to untrusted code.

5.3 Trusted Execution of Deep Neural Networks

To protect deep learning models, previous research has leveraged both cryptography-based methods and hardware-assisted trusted execution environments (TEEs) to secure various stages of deep learning workflows, including protecting the privacy of training data, safeguarding user inputs during inference, and enabling the training of confidential models across data distributed among multiple parties.

Trusted hardware-based solutions. A number of existing solutions have leveraged hardware-based Trusted Execution Environments (TEEs) to secure deep learning workloads, particularly focusing on inference on edge devices and in cloud environments.

Offline Model Guard (OMG) [17] protects the privacy of models and user inputs by extending ARM TrustZone with Sanctuary [20], a user-space enclave built on ARM TrustZone Address Space Controller. It loads the model within the Sanctuary enclave which protects the

confidentiality and integrity of the model from the untrusted normal world during inference. For training, Chiron [62] protects the privacy of training datasets from the cloud provider. It loads the training dataset in a Royan sandbox built on Intel SGX. The sandbox provides confidentiality to the dataset during training. TensorScone [77] extends Scone [14] to support the TensorFlow framework on Intel SGX enclaves for private training and inference of deep learning models. PPFL [104] presents a solution for federated learning learning, where multiple parties train a joint model while protecting the privacy of their training dataset. It relies on TEEs present on both the edge devices and a central server to privately compute the gradient updates inside the TEEs during the training process to prevent privacy violations.

These approaches based on trusted hardware provide strong security grantees, but the performance of the systems is limited as they rely only on TEEs, and leave other powerful resources available on the system underutilized.

TEE and GPU-based approaches. To address the computational limitations of the CPU-based trusted execution environments, several approaches have presented techniques to offload portions of deep learning workloads to untrusted yet high-performance hardware accelerators like GPUs.

Slalom [152] introduces a secure outsourcing framework that enables the delegation of linear layers in deep learning models to a nearby, untrusted but high-performance co-processor. The model is partitioned such that linear layers are executed on the untrusted hardware, while non-linear layers are processed securely within the TEE. To preserve the confidentiality of user inputs during inference, Slalom applies a masking technique to the inputs inside the trusted environment – adding a random vector that serves as a one-time pad – before sending the inputs outside the TEE. Later, the output from the outsourced computation on masked data is then recovered inside the TEE. To ensure the integrity of the results, Slalom employs Freivalds' algorithm within the TEE to verify the correctness of the outsourced computation. While Slalom protects the privacy of the inputs and integrity of the outsourced computation, it does not safeguard the confidentiality of the model's weights.

ShadowNet [143] presents another method based on linear transformations to protect the privacy of model weights in addition to the privacy of the user input. It outsources the linear layers of the model to mobile GPUs and runs non-linear layers within the Arm TrustZone. The linear transformations ensure the privacy of the outsourced model weights and the input when the computation is outsourced to GPUs located outside the secure world.

To speed up the training on private datasets with TEE and GPUs, Darknight [55] presents an input encoding scheme based on matrix masking. The training dataset is encoded within the TEE to protect privacy. Then, it outsources the linear computations to a set of non-colluding

GPUs to speed up the training without compromising the privacy of the training dataset. The results from multiple GPUs are collected within the TEE, and the final result is computed by combining the results from a subset of the GPUs.

Instead of protecting the privacy of the model after training, TEESlice [171] proposes to split the model before training. In contrast to previous approaches which only run non-linear layers within TEE, TEESlice run small privacy-sensitive linear layers within TEE in addition to non-linear layers. TEESlice consists of two types of layers: public layers and privacy-sensitive layers. The public layers are taken from publicly available models which are trained on the public dataset, whereas the privacy-sensitive weights are learned during the training on the private dataset. The public layers are outsourced to untrusted GPUs, whereas the privacy-sensitive layers execute within TEE. Therefore, the adversary does not learn additional information from the outsourced layers.

The main limitation of these techniques based on linear transformation is that they only outsource linear layers to an untrusted environment for fast and efficient execution. However, most of the time linear layers are followed by a non-linear layer. Therefore, the intermediate states or outputs of the linear layers need to be constantly moved between the TEE and the GPUs. MazeNet overcomes this limitation by outsourcing both linear and non-linear layers to untrusted environments, thus reducing the constant need to transfer data between TEE and the GPUs.

Cryptographic Solutions. Cryptography-based solution uses homomorphic encryption schemes [44, 130, 33] and multi-party computation [88, 125, 105, 76, 122, 123] protocols or a combination of both techniques [101, 69] to secure the privacy of training dataset during training, and the privacy of user inputs and trained model during inference.

CryptoNets [44] transforms neural networks into CryptoNets using homomorphic encryption that performs predictions over encrypted data. The user encrypts the private input with a public key and sends the encrypted input to a cloud server hosting CryptoNets. The cloud server applies the CryptoNet on the encrypted input to compute encrypted results. The encrypted results are sent to the user who owns the private key to decrypt the results.

SecureNets [27] proposes a secure matrix transformation scheme to outsource matrix multiplication operations in deep neural networks to untrusted cloud providers. For each layer of a DNN model containing matrix multiplications, the user securely transforms the input and layer weights matrix, and sends the transformed matrices to a cloud server. The cloud server performs the matrix multiplication operation and sends the results back to the user. The user verifies the correctness of the outsourced matrix multiplication using Freivalds' algorithm. On successful verification, the results are recovered, and non-linear transformations are applied to

the results. The process is repeated for the remaining layer to compute the inference results.

In secure multi-party computation approaches [88, 125, 101, 105, 76, 122], multiple parties execute a interactive protocol to jointly evaluate a function on their private data without revealing their private data to other parties. SecureML [105] presents a secure two-party computation scheme where two non-colluding servers jointly train a model on their private training dataset. CryptFlow [76] and CryptFlow2 [122] transform a TensorFlow inference code into a secure multi-party computation protocol. Gazelle [69] uses a combination of homomorphic encryption scheme for linear layers and Yao's garbled circuits for non-linear layers to perform secure inference.

Cryptography-based techniques face two primary challenges when applied in real-world deployments. The first is the significant computational overhead associated with cryptographic operations. The second is the high communication cost incurred by interactive protocols. Together, these limitations make such approaches less practical for applications where low latency and high throughput are critical.

Apart from intellectual property loss from a stolen model, having access to model weights makes the deep learning model vulnerable to different attacks.

Membership Inference Attacks. Having white-box access to deep learning models where an adversary can observe the model weights and intermediate state during inference makes the deep learning model vulnerable to Membership Inference Attacks [137] and Adversarial Attacks [47]. In membership inference attacks, an adversary wants to breach the privacy of the training dataset. The adversary wants to learn whether a given input sample was part of the training dataset. To learn about the membership of the given input sample, the adversary queries the model with the input and observes the intermediate state and results. Models tend to produce higher confidence scores for inputs they were trained on, especially the deeper layers closer to the output are more prone to leaking membership information. To protect against membership inference attacks, DarknetTZ [103] proposes to run sensitive layers, which leak membership information, within a TEE.

Models are vulnerable to membership inference attacks even if the adversary has only black-box access to the model, where only the confidence score or label is revealed to the user [29]. ModelGuard [147] proposes to perturb the output by adding noise such that the output class label remains the same but confidence scores are changed so that the adversary cannot infer the membership.

In MazeNet, the adversary either does not have access to model weights, in-TEE layers, or cannot distinguish between embedded and synthetic layers. Therefore, membership inference attacks are prevented. In case of black-box access, the enclave can securely add the noise, as

proposed by prior works, within the enclave to protect against membership inference attacks in black-box settings.

Adversarial Attacks. Another popular class of attack on deep learning models are adversarial-example attacks [47, 110] where the adversary finds adversarial examples from the input space which appear benign to human eyes but the model misclassifies them to a label of attacker choice. In these attack, the adversary queries the model with inputs and observes the intermediate state to fine-tune the inputs such that the model predicts the target class selected by the adversary. Thus, preventing access to model weights and intermediate states limit the discovery of adversarial examples.

In MazeNet, the adversary can observe the intermediate of cloaked submodel, but the intermediate state can be misleading due to the presence on synthetic weights and synthetic output.

Model Extraction Attacks. The goal of model extraction attacks [68, 108, 109] is different from model stealing attacks [67] where the attackers want to steal the weights of a trained model. In model extraction attacks, the adversary's goal is to steal the functionality of the model, i.e. produce the same output for a given input as of the target model. In model extract attacks, the adversary builds a shadow training dataset by querying the model with samples from public datasets or synthetic samples produced by learning algorithms. The attacker queries the model and records the input-output pair. Once the attacker has collected enough pairs, the attacker trains a shadow model on the shadow training dataset which tries to replicate the functionality of the target model.

To protect deep learning models against model extraction attacks, prior works have presented different techniques that either detect that the model is under model extraction attacks or perturb the output [170, 70, 68] such that the attackers perform poorly when they train the model on the shadow training dataset. The model extraction attacks and defences are orthogonal to model weights stealing.

Chapter 6

Conclusion

In the last decade, the development of deep learning models has exploded in popularity, with deep learning models getting integrated into everyday applications from text messaging to software development. Deep learning models have become the core part of enterprise applications, and well-trained models are essential for the growth of businesses. A stolen model can result in financial losses and forgone revenue for the business owners. Thus, it becomes important to protect deep learning models.

This dissertation presents methods to protect the privacy of trained deep learning models on public cloud platforms through hardware-based trusted execution environments. Traditional cloud services do not provide sufficient security guarantees to protect private and sensitive client workloads from compromised or privileged cloud service providers. Hardware-based trusted execution environments are promising candidates to secure private workloads from privileged adversaries on public cloud platforms. However, hardware-based trusted execution environments impose restrictions on the applications that run within the trusted runtime to provide security guarantees. These restrictions increase the barrier to the adoption of trusted execution environments for running diverse workloads.

In this dissertation, we studied the challenges involved in porting deep learning workloads to trusted execution environments. It uncovers the challenges faced by application developers when they want to run their existing workload on trusted execution environments. In this dissertation, we have focused on Intel SGX as it was recently launched at the beginning of this work, and it offered a strong threat model that was suitable for running sensitive applications on public cloud platforms. Furthermore, it was commercially available on inexpensive consumer desktop processors in addition to server-grade processors. However, the insights and techniques presented in this work can be applied to other trusted execution environments offered by competing silicon vendors.

As the applications do not run out-of-the-box on SGX, the first part of the dissertation focuses on the efforts required to port existing applications to Intel SGX enclaves. The software community had developed various frameworks to port existing applications to SGX enclaves. However, in the beginning, it was not clear which framework should be used to port deep learning workloads to Intel SGX enclaves. Therefore, we performed a detailed study of the frameworks for porting commodity applications to the SGX enclaves. The porting frameworks can be broadly classified into three categories: the library OS model, the library wrapper model, and the instruction wrapper model. Unfortunately, there was no publicly available implementation for the instruction wrapper model to port applications. Therefore, we built a candidate implementation, Porpoise, to port applications to SGX enclaves. Porpoise is a secondary contribution of this dissertation. Once we had a candidate implementation of the instruction wrapper model, we resumed our study, which focuses on four key parameters: porting effort, re-engineering effort, security and runtime performance. We ourselves ported a handful of popular applications with each method to find the benefits and costs of each model.

From the study, we discovered that applications with high memory usage, such as deep learning applications, incur a significant performance penalty within the SGX enclaves. Thus, the second part of the dissertation focuses on overcoming the memory and computational limitations of SGX enclaves and improving the performance of deep learning workloads.

In the second part of the dissertation, we present MazeNet, a framework to accelerate deep learning inference workflows on TEEs by outsourcing a portion of computation to untrusted processors, where the TEE ensures the security of the model and inference workflow, and the untrusted process improves the performance. MazeNet relies on a secure outsourcing scheme that offloads both linear and non-linear layers to untrusted runtime environments. MazeNet transforms pre-trained models into MazeNet models and runs them on heterogeneous environments consisting of TEE and non-TEE systems. We implemented a prototype of MazeNet on TensorFlow and transformed popular convolutional neural networks into MazeNet models to evaluate the benefits and costs. Our experimental evaluation demonstrates that MazeNet can improve the throughput by 30x and the latency by 5x.

The methods presented in the dissertation were evaluated on Intel SGX enclaves. However, the methods can be applied to other trusted execution environments as well. Chapter 2 reviewed alternative trusted execution environments offered by different processor vendors that can be used to protect deep learning models on the cloud and other platforms. These TEEs target different workloads from mobile applications to confidential virtual machines and employ various hardware and software-based protection mechanisms to ensure the security of trusted execution environments. However, TEEs cannot securely access the untrusted processors present on the

system and can use techniques presented in Chapter 4 to improve the performance of inference workloads.

Although the techniques presented in this dissertation enable tenants to run private deep learning models on public cloud platforms, there are a few limitations that can be handled in future work.

6.1 Future Work

1. **Training of deep learning models.** The techniques presented in Chapter 4 focus on the inference of deep learning models. Training is another important aspect of the deep learning pipeline. The privacy of deep learning models and training datasets can be protected with trusted execution environments by running the entire training process within TEEs. However, this approach would be very inefficient due to the following reasons. First, the intermediate states produced by each layer during the forward pass need to be stored much longer for the backward pass in backpropagation to compute gradients and update the model weights. Thus, the overall memory requirement of training is much higher than that of inference. Second, the models are trained on large training datasets, which requires high computational power. Therefore, hardware accelerators become essential for training. We need innovative solutions to offload training to hardware accelerators while ensuring the privacy of models and training datasets.

2. **Privacy of user inputs.** MazeNet in Chapter 4 does not provide privacy to the inputs of the deep learning model during the inference process. Privacy of inputs is necessary for certain financial and medical data. One of the possible directions to explore the privacy of inputs can be based on input encoding schemes and data augmentation schemes [169, 61], where the input data is transformed before being fed into a deep learning model, and the model performs predictions on the transformed instance.

3. **Recurrent neural networks.** It will be interesting to extend MazeNet to recurrent neural networks, where a layer may additionally take input from the following layers as feedback. During the splitting phase, care must be taken to ensure that there are no backwards edges between submodels, i.e., a submodel should not depend on the input produced by later submodels. However, if such a split does not exist, then new approaches are needed to securely outsource recurrent neural networks to untrusted hardware accelerators.

4. **Large language models.** With the release of GPT-3 [22] and ChatGPT, large language

models have gained popularity in commercial deployments to offer services ranging from writing text, summaries, and code. Large language models are multiple orders of magnitude bigger than convolutional neural networks in size and computational power required to perform tasks, 138 million parameters in VGG16 [138] v/s 65 billion in LLaMa [150] and 671 billion in DeepSeek-V3 [86]. Thus, a single model often does not fit within a single GPU. Further, the architecture of large language models varies significantly from recurrent neural networks-based encoder and decoder models, transformers [162], and Mixture of Experts [150]. Generally, individual layers of large language models are more computationally expensive than convolutional neural networks. Thus, it is challenging to split large language models and offload them securely and efficiently to hardware accelerators, as even a single layer may struggle to execute within TEEs because of high computational and memory requirements due to the scaled dot-product attention mechanism [163]. Thus, we need techniques that can scale secure outsourcing schemes to very large and computationally expensive models. One solution to fit large memory-intensive models within TEEs is to reduce the size of the model with model quantization [172], model distillation [50], and model pruning techniques [90]. In model quantization, the inference is performed with lower precision for floating-point operations, e.g., `Float16`, `Float8`, rather than the higher precision on which the model was trained, i.e., `Float32`. Recent hardware advances have introduced new formats that reduce the accuracy loss due to quantization, e.g., Nvidia FVFP4 [7]. While quantization reduces the model size, it adversely affects the accuracy of the model by 5% to 10% [89], which is huge as compared to recent advances in model accuracy. In model distillation, smaller models referred to as student models are trained from a larger model, referred to as the teacher model. In model pruning, inactive weights are removed based on certain criteria, such as the neurons that were not active during the validation phase or were active in a small set of inputs. These techniques aim to reduce the size of the model so that it can fit within the limited memory of edge and mobile devices and use less computational power.

The expanding scope and availability of hardware-based trusted execution environments offer attractive solutions to run confidential computations on cloud platforms and edge devices. This dissertation identifies challenges in protecting deep learning workloads with trusted execution environments and presents solutions to overcome some of the limitations of trusted execution environments. However, there are still challenges in software support and performance limitations that need to be handled to enable wider adoption of trusted execution environments, enabling the transition from legacy workloads to confidential workloads.

Bibliography

[1] Right to financial privacy act of 1978. 12 USC §§3401 – 3422. 5, 61, 65

[2] Health insurance portability and accountability act of 1996, August 1996. PUBLIC LAW 104–191—AUG. 21, 1996. 5, 61, 65

[3] Exploring qualcomm's trustzone implementation, 2015. URL http://bits-please.blogspot.com/2015/08/exploring-qualcomms-trustzone.html. Accessed: 2025-04-03. 24

[4] Intel trust domain extensions. https://www.intel.com/content/www/us/en/content-details/690419/intel-trust-domain-extensions.html, August 2021. 4, 13, 15, 16

[5] Thomas Aglassinger. Pygount. https://pypi.org/project/pygount/. 83

[6] Adil Ahmad, Kyungtae Kim, Muhammad Sarfaraz, and Byoungyoung Lee. Obliviate: A data oblivious filesystem for intel sgx. 01 2018. doi: 10.14722/ndss.2018.23296. 44, 101

[7] Eduardo Alvarez, Omri Almog, Eric Chung, Simon Layton, Dusan Stosic, Ronny Krashinsky, and Kyle Aubrey. Introducing nvfp4 for efficient and accurate low-precision inference, June 2025. URL https://developer.nvidia.com/blog/introducing-nvfp4-for-efficient-and-accurate-low-precision-inference/. 110

[8] Tiago Alves. Trustzone: Integrated hardware and software security. ***Information Quarterly***, 3:18–24, 2004. 1, 13, 22, 23

[9] AMD. Amd sev-snp: Strengthening vm isolation with integrity protection and more. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf, 2020. 1, 13, 17, 20, 21

[10] Dario Amodei and Danny Hernandez. Ai and compute. https://openai.com/blog/ai-and-compute/. 68

[11] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata. Innovative technology for CPU based attestation and sealing. In ***Workshop on Hardware and Architectural Support for Security and Privacy***, 2013. 33

[12] Arm Ltd. ***ARM CoreLink TZC-400 TrustZone Address Space Controller Technical Reference Manual***, 2013. URL https://developer.arm.com/documentation/ddi0504/latest/. Accessed: 2025-04-07. 100

[13] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. SCONE: Secure Linux containers with Intel SGX. In ***ACM/USENIX Symposium on Operating System Design and Implementation***, 2016. 7, 32, 41, 42, 44, 51, 101

[14] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, David Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. Scone: Secure linux containers with intel sgx. In ***Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation***, OSDI'16, page 689–703, USA, 2016. USENIX Association. ISBN 9781931971331. 103

[15] Giuseppe Ateniese, Luigi V Mancini, Angelo Spognardi, Antonio Villani, Domenico Vitali, and Giovanni Felici. Hacking smart machines with smarter ones: How to extract meaningful data from machine learning classifiers. ***International Journal of Security and Networks***, 10(3):137–150, 2015. 65

[16] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with Haven. ***ACM Transactions on Computer Systems***, 33(3), September 2015. 6, 31, 35, 101

[17] Sebastian P. Bayerl, Tommaso Frassetto, Patrick Jauernig, Korbinian Riedhammer, Ahmad-Reza Sadeghi, Thomas Schneider, Emmanuel Stapf, and Christian Weinert. Offline model guard: Secure and private ml on mobile devices. In ***Proceedings of the 23rd Conference on Design, Automation and Test in Europe***, DATE '20, page 460–465, San Jose, CA, USA, 2020. EDA Consortium. ISBN 9783981926347. 66, 67, 102

[18] Jeevan S. Bhungal. Intel® trusted execution technology (txt) for client platforms. Intel Developer Article, July 2023. URL https://www.intel.com/content/www/us/en/developer/articles/tool/intel-trusted-execution-technology.html. Updated on July 19, 2023. 15

[19] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. Evasion attacks against machine learning at test time. In ***Machine Learning and Knowledge Discovery in Databases***, pages 387–402, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-40994-3. 65

[20] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Sanctuary: Arming trustzone with user-space enclaves. In ***NDSS***, 2019. 100, 102

[21] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D 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 Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, ***Advances in Neural Information Processing Systems***, volume 33, pages 1877–1901. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf. 68

[22] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. ***Advances in neural information processing systems***, 33:1877–1901, 2020. 109

[23] Nicholas Carlini, Chang Liu, Úlfar Erlingsson, Jernej Kos, and Dawn Song. The secret sharer: Evaluating and testing unintended memorization in neural networks. In ***28th USENIX Security Symposium (USENIX Security 19)***, pages 267–284, Santa Clara, CA, August 2019. USENIX Association. ISBN 978-1-939133-06-9. URL https://www.usenix.org/conference/usenixsecurity19/presentation/carlini. 65

[24] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. Sok: Understanding

the prevailing security vulnerabilities in trustzone-assisted tee systems. In ***2020 IEEE Symposium on Security and Privacy (SP)***, pages 1416–1432. IEEE, 2020. 24

[25] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call API is a bad untrusted RPC interface. In ***Proceedings of the 2013 International Conference on Architectural Support for Programming Languages and Operating Systems***, 2013. 31, 35, 38

[26] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In ***2019 IEEE European Symposium on Security and Privacy (EuroS&P)***, pages 142–157. IEEE, 2019. 13, 67

[27] Xuhui Chen, Jinlong Ji, Lixing Yu, Changqing Luo, and Pan Li. Securenets: Secure inference of deep neural networks on an untrusted cloud. In Jun Zhu and Ichiro Takeuchi, editors, ***Proceedings of The 10th Asian Conference on Machine Learning***, volume 95 of ***Proceedings of Machine Learning Research***, pages 646–661. PMLR, 14–16 Nov 2018. URL https://proceedings.mlr.press/v95/chen18a.html. 66, 67, 104

[28] François Chollet. Keras. https://keras.io/api/applications/. 85

[29] Christopher A. Choquette-Choo, Florian Tramer, Nicholas Carlini, and Nicolas Papernot. Label-only membership inference attacks. In Marina Meila and Tong Zhang, editors, ***Proceedings of the 38th International Conference on Machine Learning***, volume 139 of ***Proceedings of Machine Learning Research***, pages 1964–1974. PMLR, 18–24 Jul 2021. URL https://proceedings.mlr.press/v139/choquette-choo21a.html. 65, 105

[30] Cloud Hypervisor Contributors. Update tdx documentation, 2024. URL https://github.com/cloud-hypervisor/cloud-hypervisor/issues/7181. Accessed: 2025-07-14. 16

[31] OP-TEE Community. Op-tee documentation, 2025. URL https://optee.readthedocs.io/. Accessed: April 2, 2025. 22, 23

[32] Victor Costan and Srinivas Devadas. Intel sgx explained. ***Cryptology ePrint Archive***, 2016. 1

[33] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. Chet: an optimizing compiler for fully-homomorphic neural-network inferencing. In ***Proceedings of the 40th ACM SIGPLAN conference on programming language design and implementation***, pages 142–156, 2019. 104

[34] Tom Woller David Kaplan, Jeremy Powell. Amd memory encryption. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/memory-encryption-white-paper.pdf, 2021. 4, 13, 17, 18

[35] Yunjie Deng, Chenxu Wang, Shunchang Yu, Shiqing Liu, Zhenyu Ning, Kevin Leach, Jin Li, Shoumeng Yan, Zhengyu He, Jiannong Cao, et al. Strongbox: A gpu tee on arm endpoints. In ***Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security***, pages 769–783, 2022. 100

[36] Yu Ding, Ran Duan, Long Li, Yueqiang Cheng, Yulong Zhang, Tanghui Chen, Tao Wei, and Huibo Wang. Poster: Rust sgx sdk: Towards memory safety in intel sgx enclave. In ***Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security***, CCS '17, pages 2491–2493, 2017. ISBN 978-1-4503-4946-8. 101

[37] Samsung Electronics. ***Samsung Knox***, 2021. URL https://image-us.samsung.com/SamsungUS/samsungbusiness/solutions/topics/iot/071421/Knox-Whitepaper-v1.5-20210709.pdf. 22

[38] Tarek Elgamal and Klara Nahrstedt. Serdab: An iot framework for partitioning neural networks computation across multiple enclaves. In ***2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)***, pages 519–528, 2020. doi: 10.1109/CCGrid49817.2020.00-41. 73, 81

[39] Facebook. Pytorch, November 2023. URL https://github.com/https://github.com/pytorch/pytorch. 1

[40] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In ***Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security***, CCS '15, page 1322–1333, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450338325. doi: 10.1145/2810103.2813677. URL https://doi.org/10.1145/2810103.2813677. 5, 65

[41] Matthew Fredrikson, Eric Lantz, Somesh Jha, Simon Lin, David Page, and Thomas Ristenpart. Privacy in pharmacogenetics: An end-to-end case study of personalized warfarin dosing. In ***Proceedings of the 23rd USENIX Conference on Security Symposium***, SEC'14, page 17–32, USA, 2014. USENIX Association. ISBN 9781931971157. 65

[42] Mudasir A Ganaie, Minghui Hu, Ashwani Kumar Malik, Muhammad Tanveer, and Ponnuthurai N Suganthan. Ensemble deep learning: A review. ***Engineering Applications of Artificial Intelligence***, 115:105151, 2022. 76

[43] Hossein Gholamalinezhad and Hossein Khosravi. Pooling methods in deep neural networks, a review. ***arXiv preprint arXiv:2009.07485***, 2020. 64

[44] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In Maria Florina Balcan and Kilian Q. Weinberger, editors, ***Proceedings of The 33rd International Conference on Machine Learning***, volume 48 of ***Proceedings of Machine Learning Research***, pages 201–210, New York, New York, USA, 20–22 Jun 2016. PMLR. URL https://proceedings.mlr.press/v48/gilad-bachrach16.html. 66, 67, 104

[45] Github. Octoverse: The state of open source and rise of ai in 2023, November 2023. URL https://github.blog/2023-11-08-the-state-of-open-source-and-ai/. 1

[46] David Goltzsche, Colin Wulf, Divya Muthukumaran, Konrad Rieck, Peter Pietzuch, and Rüdiger Kapitza. Trustjs: Trusted client-side execution of javascript. In ***Proceedings of the 10th European Workshop on Systems Security***, EuroSec'17, pages 7:1–7:6, 2017. ISBN 978-1-4503-4935-2. 101

[47] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In ***International Conference on Learning Representations (ICLR)***, 2015. URL https://arxiv.org/abs/1412.6572. 5, 65, 66, 105, 106

[48] Google. Tensorflow, November 2023. URL https://github.com/tensorflow/tensorflow. 1, 63, 83, 85

[49] Google. Trusty tee, 2025. URL https://source.android.com/docs/security/trusty. Accessed: April 2, 2025. 22, 23

[50] Jianping Gou, Baosheng Yu, Stephen J Maybank, and Dacheng Tao. Knowledge distillation: A survey. ***International journal of computer vision***, 129(6):1789–1819, 2021. 98, 110

[51] Gramine. Gramine documentation. https://gramine.readthedocs.io/en/v1.6/devel/features.html, 2023. Accessed 16-05-2025. 21

[52] Michael Gruhn and Tilo Müller. On the practicability of cold boot attacks. In ***2013 International Conference on Availability, Reliability and Security***, pages 390–397. IEEE, 2013. 2, 13, 17, 58

[53] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. Lest we remember: cold-boot attacks on encryption keys. ***Communications of the ACM***, 52(5):91–98, 2009. 2, 13, 17, 58

[54] Lucjan Hanzlik, Yang Zhang, Kathrin Grosse, Ahmed Salem, Maximilian Augustin, Michael Backes, and Mario Fritz. Mlcapsule: Guarded offline deployment of machine learning as a service. In ***2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)***, pages 3295–3304, June 2021. doi: 10.1109/CVPRW53098.2021.00368. 66, 67

[55] Hanieh Hashemi, Yongqin Wang, and Murali Annavaram. Darknight: An accelerated framework for privacy and integrity preserving deep learning using trusted hardware. In ***MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture***, MICRO '21, page 212–224, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385572. doi: 10.1145/3466752.3480112. URL https://doi.org/10.1145/3466752.3480112. 63, 66, 67, 68, 103

[56] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In ***2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)***, pages 770–778, 2016. doi: 10.1109/CVPR.2016.90. 11, 63, 64, 71, 76, 84

[57] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. del Cuvillo. Using innovative instructions to create trustworthy software solutions. In ***Workshop on Hardware and Architectural Support for Security and Privacy***, 2013. 30, 33

[58] Zhichao Hua, Jinyu Gu, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. {vTZ}: virtualizing {ARM}{TrustZone}. In ***26th USENIX Security Symposium (USENIX Security 17)***, pages 541–556, 2017. 100

[59] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In ***Proceedings of the IEEE conference on computer vision and pattern recognition***, pages 4700–4708, 2017. 11, 63, 64, 71, 76, 84

[60] Haoyang Huang, Fengwei Zhang, Shoumeng Yan, Tao Wei, and Zhengyu He. Sok: A comparison study of arm trustzone and cca. In ***2024 International Symposium on Secure and Private Execution Environment Design (SEED)***, pages 107–118. IEEE, 2024. 58

[61] Yangsibo Huang, Zhao Song, Kai Li, and Sanjeev Arora. InstaHide: Instance-hiding schemes for private distributed learning. In Hal Daumé III and Aarti Singh, editors, ***Proceedings of the 37th International Conference on Machine Learning***, volume 119 of ***Proceedings of Machine Learning Research***, pages 4507–4518. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/huang20i.html. 109

[62] Tyler Hunt, Congzheng Song, Reza Shokri, Vitaly Shmatikov, and Emmett Witchel. Chiron: Privacy-preserving machine learning as a service, 2018. URL https://arxiv.org/abs/1803.05961. 103

[63] Intel. Q3 2018 speculative execution side channel update. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00161.html. 67

[64] Intel. Software guard extensions programming reference, revision 2, 2014. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf. xiii, 34

[65] Intel. ***Intel® 64 and IA-32 Architectures software Developer's Manual***. Intel, September 2016. Available at https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3c-part-3-manual.pdf. 15

[66] intelsdk. Intel sgx for linux. https://github.com/intel/linux-sgx. 43

[67] Matthew Jagielski, Nicholas Carlini, David Berthelot, Alex Kurakin, and Nicolas Papernot. High accuracy and high fidelity extraction of neural networks. In ***29th USENIX Security Symposium (USENIX Security 20)***, pages 1345–1362. USENIX Association, August 2020. ISBN 978-1-939133-17-5. URL https://www.usenix.org/conference/usenixsecurity20/presentation/jagielski. 61, 106

[68] Mika Juuti, Sebastian Szyller, Samuel Marchal, and N Asokan. Prada: protecting against dnn model stealing attacks. In ***2019 IEEE European Symposium on Security and Privacy (EuroS&P)***, pages 512–527. IEEE, 2019. 106

[69] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. Gazelle: A low latency framework for secure neural network inference. In ***Proceedings of the 27th USENIX Conference on Security Symposium***, SEC'18, page 1651–1668, USA, 2018. USENIX Association. ISBN 9781931971461. 66, 67, 104, 105

[70] Sanjay Kariyappa and Moinuddin K Qureshi. Defending against model stealing attacks with adaptive misinformation. In ***Proceedings of the IEEE/CVF conference on computer vision and pattern recognition***, pages 770–778, 2020. 106

[71] Hormuzd Khosravi. Runtime encryption of memory with intel® total memory encryption - multi-key. Whitepaper, Intel Corporation, October 2022. URL https://www.intel.com/content/dam/www/central-libraries/us/en/documents/2022-10/intel-total-memory-encryption-multi-key-whitepaper.pdf. 15

[72] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. ***ACM SIGARCH Computer Architecture News***, 42(3):361–372, 2014. 2

[73] R. Krahn, B. Trach, A. Vahldiek-Oberwagner, T. Knauth, P. Bhatotia, and C. Fetzer. Pesos: Policy enhanced secure object store. In ***Proceedings of the Thirteenth EuroSys Conference***, EuroSys '18, 2018. ISBN 978-1-4503-5584-1. 101

[74] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In ***Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1***, NIPS'12, page 1097–1105, Red Hook, NY, USA, 2012. Curran Associates Inc. 11, 84

[75] Mark Kuhne, Supraja Sridhara, Andrin Bertschi, Nicolas Dutly, Srdjan Capkun, and Shweta Shinde. Aster: Fixing the android tee ecosystem with arm cca. ***arXiv preprint arXiv:2407.16694***, 2024. 101

[76] Nishant Kumar, Mayank Rathee, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. Cryptflow: Secure tensorflow inference. Cryptology ePrint Archive, Paper 2019/1049, 2019. URL https://eprint.iacr.org/2019/1049. https://eprint.iacr.org/2019/1049. 66, 67, 104, 105

[77] Roland Kunkel, Do Le Quoc, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. Tensorscone: A secure tensorflow framework using intel sgx, 2019. URL https://arxiv.org/abs/1902.04413. 66, 67, 103

[78] Dayeol Lee, Dongha Jung, Ian T Fang, Chia-Che Tsai, and Raluca Ada Popa. An {Off-Chip} attack on hardware enclaves via the memory bus. In ***29th USENIX Security Symposium (USENIX Security 20)***, 2020. 2, 13, 17, 58

[79] Taegyeong Lee, Zhiqi Lin, Saumay Pushp, Caihua Li, Yunxin Liu, Youngki Lee, Fengyuan Xu, Chenren Xu, Lintao Zhang, and Junehwa Song. Occlumency: Privacy-preserving remote deep-learning inference using sgx. In ***The 25th Annual International Conference on Mobile Computing and Networking***, MobiCom '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450361699. doi: 10.1145/3300061.3345447. URL https://doi.org/10.1145/3300061.3345447. 66, 67

[80] Zheng Li and Yang Zhang. Membership leakage in label-only exposures. In ***Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security***, CCS '21, page 880–895, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384544. doi: 10.1145/3460120.3484575. URL https://doi.org/10.1145/3460120.3484575. 65

[81] ARM Limited. Building a secure system using trustzone technology. https://documentation-service.arm.com/static/5f212796500e883ab8e74531, December 2008. 22

[82] ARM Limited. Trustzone for armv8-a, January 2020. 22

[83] ARM Limited. Realm management extension guide. https://developer.arm.com/documentation/den0126/latest, 2021. 24, 25

[84] ARM Limited. Arm realm management extension (rme) system architecture. https://developer.arm.com/documentation/den0129/latest, 2021. 26

[85] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter Pietzuch. Glamdring: Automatic application partitioning for intel SGX. In ***2017 USENIX Annual Technical Conference (USENIX ATC 17)***, pages 285–298, Santa Clara, CA, July 2017. USENIX Association. ISBN 978-1-931971-38-6. URL https://www.usenix.org/conference/atc17/technical-sessions/presentation/lind. 31, 49, 102

[86] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. ***arXiv preprint arXiv:2412.19437***, 2024. 110

[87] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious neural network predictions via minionn transformations. In ***Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security***, CCS '17, page 619–631, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349468. doi: 10.1145/3133956.3134056. URL https://doi.org/10.1145/3133956.3134056. 66, 67

[88] Jian Liu, Mika Juuti, Yao Lu, and Nadarajah Asokan. Oblivious neural network predictions via minionn transformations. In ***Proceedings of the 2017 ACM SIGSAC conference on computer and communications security***, pages 619–631, 2017. 104, 105

[89] Zhenhua Liu, Yunhe Wang, Kai Han, Wei Zhang, Siwei Ma, and Wen Gao. Post-training quantization for vision transformer. ***Advances in Neural Information Processing Systems***, 34:28092–28103, 2021. 110

[90] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. Rethinking the value of network pruning. In ***International Conference on Learning Representations***, 2019. 110

[91] ARM Ltd. ***ARM1176JZF-S Technical Reference Manual***, 2004. URL https://developer.arm.com/documentation/ddi0301/h/. 22

[92] ARM Ltd. ***ARM Architecture Reference Manual***, issue i edition, 2005. URL https://developer.arm.com/documentation/ddi0100/i/. 22

[93] Arm Ltd. ***Arm Confidential Compute Architecture***, 2021. URL https://developer.arm.com/documentation/den0125/0300. Accessed: April 2, 2025. 1, 4, 13, 22, 26

[94] Arm Ltd. ***Arm Architecture Reference Manual for A-profile architecture***, 2022. URL https://developer.arm.com/documentation/ddi0487/latest/. Accessed: April 2, 2025. 22

[95] Jubayer Mahmod and Matthew Hicks. Untrustzone: Systematic accelerated aging to expose on-chip secrets. In ***2024 IEEE Symposium on Security and Privacy (SP)***, pages 4107–4124. IEEE, 2024. 13, 17, 58

[96] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, and U. R. Savagaonkar V. Shanbhogue. Innovative instructions and software model for isolated execution. In ***Workshop on Hardware and Architectural Support for Security and Privacy***, 2013. 30, 33

[97] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In ***Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy***, HASP '13, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450321181. doi: 10.1145/2487726.2488368. URL https://doi.org/10.1145/2487726.2488368. 13

[98] memtier. A high-throughput benchmarking tool for redis and memcached. https://github.com/RedisLabs/memtier_benchmark. 52

[99] Microsoft. Microsoft cloud strength fuels first quarter results, October 2023. URL https://www.microsoft.com/en-us/Investor/earnings/FY-2024-Q1/press-release-webcast. 1

[100] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa. Oblix: An efficient oblivious search index. In ***2018 IEEE Symposium on Security and Privacy (SP)***, pages 279–296, May 2018. doi: 10.1109/SP.2018.00045. 44, 101

[101] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. Delphi: A cryptographic inference system for neural networks. In ***Proceedings of the 2020 Workshop on Privacy-Preserving Machine Learning in Practice***, pages 27–30, 2020. 104, 105

[102] Masanori Misono, Dimitrios Stavrakakis, Nuno Santos, and Pramod Bhatotia. Confidential vms explained: An empirical analysis of amd sev-snp and intel tdx. ***Proceedings of the ACM on Measurement and Analysis of Computing Systems***, 8(3):1–42, 2024. 58

[103] Fan Mo, Ali Shahin Shamsabadi, Kleomenis Katevas, Soteris Demetriou, Ilias Leontiadis, Andrea Cavallaro, and Hamed Haddadi. Darknetz: Towards model privacy at the edge using trusted execution environments. In ***Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services***, MobiSys '20, page 161–174, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379540. doi: 10.1145/3386901.3388946. URL https://doi.org/10.1145/3386901.3388946. 67, 105

[104] Fan Mo, Hamed Haddadi, Kleomenis Katevas, Eduard Marin, Diego Perino, and Nicolas Kourtellis. Ppfl: privacy-preserving federated learning with trusted execution environments. In ***Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services***, MobiSys '21, page 94–108, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384438. doi: 10.1145/3458864.3466628. URL https://doi.org/10.1145/3458864.3466628. 103

[105] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In ***2017 IEEE Symposium on Security and Privacy (SP)***, pages 19–38, 2017. doi: 10.1109/SP.2017.12. 67, 104, 105

[106] musl-libc. musl - an implementation of the standard library for linux-based systems. https://git.musl-libc.org/cgit/musl. 42

[107] NVIDIA Corporation. Confidential compute on nvidia hopper h100. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/HCC-Whitepaper-v1.0.pdf, 2023. Accessed: 2025-07-14. 11, 13, 27, 28

[108] Tribhuvanesh Orekondy, Bernt Schiele, and Mario Fritz. Knockoff nets: Stealing functionality of black-box models. In ***Proceedings of the IEEE/CVF conference on computer vision and pattern recognition***, pages 4954–4963, 2019. 106

[109] Soham Pal, Yash Gupta, Aditya Shukla, Aditya Kanade, Shirish Shevade, and Vinod Ganapathy. Activethief: Model extraction using active learning and unannotated public data. In ***Proceedings of the AAAI conference on artificial intelligence***, volume 34, pages 865–872, 2020. 5, 65, 106

[110] Ren Pang, Hua Shen, Xinyang Zhang, Shouling Ji, Yevgeniy Vorobeychik, Xiapu Luo, Alex Liu, and Ting Wang. ***A Tale of Evil Twins: Adversarial Inputs versus Poisoned Models***, page 85–99. Association for Computing Machinery, New York, NY, USA, 2020. ISBN 9781450370899. URL https://doi.org/10.1145/3372297.3417253. 106

[111] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In ***2016 IEEE European Symposium on Security and Privacy (EuroS P)***, pages 372–387, 2016. doi: 10.1109/EuroSP.2016.36. 65

[112] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In ***2016 IEEE European symposium on security and privacy (EuroS&P)***, pages 372–387. IEEE, 2016. 65

[113] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In ***Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security***, ASIA CCS '17, page 506–519, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349444. doi: 10.1145/3052973.3053009. URL https://doi.org/10.1145/3052973.3053009. 65

[114] Tony Peng. The staggering cost of training sota ai models. https://medium.com/syncedreview/the-staggering-cost-of-training-sota-ai-models-e329e80fa82, 2019. 5

[115] Sandro Pinto and Nuno Santos. Demystifying arm trustzone: A comprehensive survey. ***ACM computing surveys (CSUR)***, 51(6):1–36, 2019. 22

[116] D. Porter, S. Boyd-Wickizer, J. Powell, R. Olinsky, and G. Hunt. Rethinking the library OS from the top-down. In ***ASPLOS***, 2011. 36

[117] C. Priebe, K. Vaswani, and M. Costa. EnclaveDB: A secure database using SGX. In ***IEEE Symposium on Security and Privacy***, 2018. 31, 101

[118] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A. Sartakov, and Peter Pietzuch. Sgx-lkl: Securing the host os interface for trusted execution. In ***arXiv:1908.11143***, August 2019. 6, 31, 35, 36, 83, 101

[119] O. Purdila, L. Grinjincu, and N. Tapus. LKL: The linux kernel library. In ***RoEduNet IEEE International Conference***, 2010. 36

[120] QEMU Project. Qemu monitor. https://qemu-project.gitlab.io/qemu/system/monitor.html, 2024. Accessed: 2025-04-07. 18

[121] Andrinandrasana David Rasamoelina, Fouzia Adjailia, and Peter Sinčák. A review of activation function for artificial neural network. In ***2020 IEEE 18th world symposium on applied machine intelligence and informatics (SAMI)***, pages 281–286. IEEE, 2020. 64

[122] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. Cryptflow2: Practical 2-party secure inference. In ***Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security***, CCS '20, page 325–342, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370899. doi: 10.1145/3372297.3417274. URL https://doi.org/10.1145/3372297.3417274. 104, 105

[123] M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. Cryptology ePrint Archive, Paper 2017/1164, 2017. URL https://eprint.iacr.org/2017/1164. https://eprint.iacr.org/2017/1164. 67, 104

[124] Dan Rosenberg. Reflections on trusting trustzone. https://www.blackhat.com/docs/us-14/materials/us-14-Rosenberg-Reflections-on-Trusting-TrustZone.pdf, 2014. 24

[125] Bita Darvish Rouhani, M. Sadegh Riazi, and Farinaz Koushanfar. Deepsecure: Scalable provably-secure deep learning. In ***Proceedings of the 55th Annual Design Automation Conference***, DAC '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357005. doi: 10.1145/3195970.3196023. URL https://doi.org/10.1145/3195970.3196023. 67, 104, 105

[126] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. ***International Journal of Computer Vision (IJCV)***, 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y. 65, 86

[127] Ahmed Salem, Yang Zhang, Mathias Humbert, Pascal Berrang, Mario Fritz, and Michael Backes. Ml-leaks: Model and data independent membership inference attacks and defenses on machine learning models. In ***Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS)***, 2019. 65

[128] Ahmed Salem, Apratim Bhattacharya, Michael Backes, Mario Fritz, and Yang Zhang. Updates-Leak: Data set inference and reconstruction attacks in online learning. In ***29th USENIX Security Symposium (USENIX Security 20)***, pages 1291–1308. USENIX Association, August 2020. ISBN 978-1-939133-17-5. URL https://www.usenix.org/conference/usenixsecurity20/presentation/salem. 65

[129] Fan Sang, Jaehyuk Lee, Xiaokuan Zhang, and Taesoo Kim. Portal: Fast and secure device access with arm cca for modern arm mobile system-on-chips (socs). In ***Proceedings of the 2025 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA***, pages 12–14, 2025. 101

[130] Amartya Sanyal, Matt Kusner, Adria Gascon, and Varun Kanade. Tapas: Tricks to accelerate (encrypted) prediction as a service. In ***International conference on machine learning***, pages 4490–4499. PMLR, 2018. 104

[131] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In ***Proceedings of the 2015 IEEE Symposium on Security and Privacy***, SP '15, pages 38–54, Washington, DC, USA, 2015. IEEE Computer Society. ISBN 978-1-4673-6949-7. doi: 10.1109/SP.2015.10. URL https://doi.org/10.1109/SP.2015.10. 31, 101

[132] Kripa Shanker, Arun Joseph, and Vinod Ganapathy. An evaluation of methods to port legacy code to sgx enclaves. In ***Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering***, ESEC/FSE 2020, page 1077–1088, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370431. doi: 10.1145/3368089.3409726. URL https://doi.org/10.1145/3368089.3409726. 83

[133] Di Shen. Exploiting trustzone on android. ***Black Hat USA***, 2:267–280, 2015. 24

[134] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. Occlum: Secure and efficient multitasking inside a single enclave

of intel sgx. In ***Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems***, ASPLOS '20, page 955–970, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371025. doi: 10.1145/3373376.3378469. URL https://doi.org/10.1145/3373376.3378469. 6

[135] S. Shinde, D. Le Tien, S. Tople, and P. Saxena. Panoply: Low-TCB Linux applications with SGX enclaves. In ***Networked and Distributed Systems Security Symposium***, 2017. 7, 32, 33, 39, 101

[136] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. Panoply: Low-tcb linux applications with sgx enclaves. In ***NDSS***, 2017. 9, 21, 83

[137] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In ***2017 IEEE symposium on security and privacy (SP)***, pages 3–18. IEEE, 2017. 5, 61, 65, 66, 105

[138] K Simonyan and A Zisserman. Very deep convolutional networks for large-scale image recognition. In ***3rd International Conference on Learning Representations (ICLR 2015)***. Computational and Biological Learning Society, 2015. 63, 64, 110

[139] R. Sinha, S. Seshia, S. Rajamani, and K. Vaswani. Moat: Verifying the confidentiality of enclave programs. In ***ACM Conference on Computer and Communications Security***, 2015. 31, 102

[140] R. Sinha, M. Costa, A. Lal, N. Lopes, S. Rajamani, S. Seshia, and K. Vaswani. A design and verification methodology for secure isolated regions. In ***ACM SIGPLAN Conference on Programming Language Design and Implementation***, 2016. 31, 44, 102

[141] Supraja Sridhara, Andrin Bertschi, Benedict Schlüter, Mark Kuhne, Fabio Aliberti, and Shweta Shinde. {ACAI}: Protecting accelerator execution with arm confidential computing architecture. In ***33rd USENIX Security Symposium (USENIX Security 24)***, pages 3423–3440, 2024. 100

[142] Yang Su and Damith C Ranasinghe. Leaving your things unattended is no joke! memory bus snooping and open debug interface exploits. In ***2022 IEEE International Conference on Pervasive Computing and Communications Workshops and***

other Affiliated Events (PerCom Workshops), pages 643–648. IEEE, 2022. 2, 13, 17, 58

[143] Zhichuang Sun, Ruimin Sun, Changming Liu, Amrita Roy Chowdhury, Somesh Jha, and Long Lu. Shadownet: A secure and efficient system for on-device model inference, 2020. URL https://arxiv.org/abs/2011.05905. 63, 67, 103

[144] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In ***International Conference on Learning Representations (ICLR)***, 2014. 65, 72

[145] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In ***Proceedings of the IEEE conference on computer vision and pattern recognition***, pages 1–9, 2015. 70, 76

[146] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. Vault: Reducing paging overheads in sgx with efficient integrity verification structures. ***SIGPLAN Not.***, 53 (2):665–678, mar 2018. ISSN 0362-1340. doi: 10.1145/3296957.3177155. URL https://doi.org/10.1145/3296957.3177155. 68

[147] Minxue Tang, Anna Dai, Louis DiValentin, Aolin Ding, Amin Hass, Neil Zhenqiang Gong, Yiran Chen, and Hai "Helen" Li. ModelGuard: Information-Theoretic defense against model extraction attacks. In ***33rd USENIX Security Symposium (USENIX Security 24)***, pages 5305–5322, Philadelphia, PA, August 2024. USENIX Association. ISBN 978-1-939133-44-1. URL https://www.usenix.org/conference/usenixsecurity24/presentation/tang. 105

[148] The LibVMI Project. Libvmi: Simplified virtual machine introspection. https://github.com/libvmi/libvmi, 2023. Accessed: 2025-04-07. 18

[149] Dave (Jing) Tian, Joseph Choi, Grant Hernandez, Patrick Traynor, and Kevin Butler. A practical intel sgx setting for linux containers in the cloud. In ***ACM Conference on Data and Application Security and Privacy***, 2019. 7, 32, 49, 101

[150] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. ***arXiv preprint arXiv:2302.13971***, 2023. 110

[151] Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. Shieldbox: Secure middleboxes using shielded execution. In ***Proceedings of the Symposium on SDN Research***, SOSR '18, pages 2:1–2:14, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5664-0. doi: 10.1145/3185467.3185469. URL http://doi.acm.org/10.1145/3185467.3185469. 101

[152] Florian Tramer and Dan Boneh. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. In ***International Conference on Learning Representations***, 2019. URL https://openreview.net/forum?id=rJVorjCcKQ. 63, 67, 103

[153] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction APIs. In ***25th USENIX Security Symposium (USENIX Security 16)***, pages 601–618, Austin, TX, August 2016. USENIX Association. ISBN 978-1-931971-32-4. URL https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/tramer. 65

[154] M. Tran, L. Luu, M. Kang, I. Bentov, and P. Saxena. Obscuro: A bitcoin mixer using trusted execution environments. In ***Proceedings of the 34th Annual Computer Security Applications Conference***, ACSAC '18, pages 692–701, 2018. ISBN 978-1-4503-6569-7. 101

[155] C. Tsai, D. E. Porter, and M. Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In ***USENIX Annual Technical Conference***, 2017. 6, 31, 33, 35, 83, 101

[156] C. Tsai, J. Son, B. Jain, J. McAvey, R. Popa, and D. Porter. Civet: An efficient Java partitioning framework for hardware enclaves. In ***USENIX Security Symposium***, 2020. 102

[157] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. Cooperation and security isolation of library oses for multi-process applications. In ***Proceedings of the Ninth European Conference on Computer Systems***, EuroSys '14, pages 9:1–9:14, 2014. ISBN 978-1-4503-2704-6. 36

[158] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E. Porter. A study of modern linux api usage and compatibility: What to support when you're supporting.

In ***Proceedings of the Eleventh European Conference on Computer Systems***, EuroSys '16, pages 16:1–16:16, 2016. ISBN 978-1-4503-4240-7. 45

[159] Chia-Che Tsai, Donald E. Porter, and Mona Vij. Graphene-sgx: A practical library os for unmodified applications on sgx. In ***Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference***, USENIX ATC '17, page 645–658, USA, 2017. USENIX Association. ISBN 9781931971386. 9, 21

[160] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In ***USENIX Security Symposium***, 2018. 35

[161] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In ***Proceedings fo the 27th USENIX Security Symposium***. USENIX Association, 2018. 13, 67

[162] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. ***Advances in neural information processing systems***, 30, 2017. 110

[163] Christopher Wolters, Xiaoxuan Yang, Ulf Schlichtmann, and Toyotaro Suzumura. Memory is all you need: An overview of compute-in-memory architectures for accelerating large language model inference. ***arXiv preprint arXiv:2406.08413***, 2024. 110

[164] wrk. A constant throughput, correct latency recording http benchmarking tool variant of wrk. https://github.com/giltene/wrk2. 52

[165] B. Xing, M. Shanahan, and R. Leslie-Hurd. Intel® software guard extensions software support for dynamic memory allocation inside an enclave. In ***Workshop on Hardware and Architectural Support for Security and Privacy***, 2016. 46

[166] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. Xlnet: Generalized autoregressive pretraining for language understanding, 2019. URL https://arxiv.org/abs/1906.08237. 65

[167] Salessawi Ferede Yitbarek, Misiker Tadesse Aga, Reetuparna Das, and Todd Austin. Cold boot attacks are still hot: Security analysis of memory scramblers in modern processors. In ***2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)***, pages 313–324. IEEE, 2017. 2, 13, 17, 25, 58

[168] Rui Yuan, Yu-Bin Xia, Hai-Bo Chen, Bin-Yu Zang, and Jan Xie. Shadoweth: Private smart contract on public blockchain. ***Journal of Computer Science and Technology***, 33(3):542–556, May 2018. 101

[169] Hongyi Zhang, Moustapha Cisse, Yann N. Dauphin, and David Lopez-Paz. mixup: Beyond empirical risk minimization. In ***International Conference on Learning Representations***, 2018. URL https://openreview.net/forum?id=r1Ddp1-Rb. 109

[170] Jiliang Zhang, Shuang Peng, Yansong Gao, Zhi Zhang, and Qinghui Hong. Apmsa: Adversarial perturbation against model stealing attacks. ***IEEE Transactions on Information Forensics and Security***, 18:1667–1679, 2023. 106

[171] Ziqi Zhang, Chen Gong, Yifeng Cai, Yuanyuan Yuan, Bingyan Liu, Ding Li, Yao Guo, and Xiangqun Chen. No privacy left outside: On the (in-) security of tee-shielded dnn partition for on-device ml. In ***2024 IEEE Symposium on Security and Privacy (SP)***, pages 3327–3345. IEEE, 2024. 104

[172] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. Trained ternary quantization. In ***International Conference on Learning Representations***, 2017. 110