SecDeep: Secure and Performant On-device Deep Learning Inference Framework for Mobile and IoT Devices

Renju Liu
University of California, Los Angeles

Luis Garcia
University of Southern California
Information Sciences Institute

Zaoxing Liu
Boston University

Botong Ou
University of California, Los Angeles

Mani Srivastava
University of California, Los Angeles

Abstract

There is an increasing emphasis on securing deep learning (DL) inference pipelines for mobile and IoT applications with privacy-sensitive data. Prior works have shown that privacy-sensitive data can be secured throughout deep learning inferences on cloud-offloaded models through trusted execution environments such as Intel SGX. However, prior solutions do not address the fundamental challenges of securing the resource-intensive inference tasks on low-power, low-memory devices (e.g., mobile and IoT devices), while achieving high performance. To tackle these challenges, we propose SecDeep, a low-power DL inference framework demonstrating that both *security* and *performance* of deep learning inference on edge devices are well within our reach. Leveraging TEEs with limited resources, SecDeep guarantees full confidentiality for input and intermediate data, as well as the integrity of the deep learning model and framework. By enabling and securing neural accelerators, SecDeep is the first of its kind to provide trusted and performant DL model inferencing on IoT and mobile devices. We implement and validate SecDeep by interfacing the ARM NN DL framework with ARM TrustZone. Our evaluation shows that we can securely run inference tasks with 16× to 172× faster performance than no acceleration approaches by leveraging edge-available accelerators.

CCS Concepts: • **Security and privacy** → **Software security engineering**; **Mobile platform security**; **Trusted computing**.

ACM Reference Format:
Renju Liu, Luis Garcia, Zaoxing Liu, Botong Ou, and Mani Srivastava. 2021. SecDeep: Secure and Performant On-device Deep Learning Inference Framework for Mobile and IoT Devices. In *International Conference on Internet-of-Things Design and Implementation (IoTDI '21), May 18–21, 2021, Charlottesvle, VA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3450268.3453524

1 Introduction

Deep learning (DL) has enabled applications that require complex reasoning about the raw sensor data stemming from the proliferous Internet of Things (IoT). Today, with the goal to enhance application latency and user privacy, deep learning tasks are moving towards mobile and IoT devices [27, 28, 31, 43, 44]. However, the increase in performance swells the burden of security and privacy on resource-constrained devices.

To date, several efforts have been made towards designing hardware primitives to achieve better latency and data privacy on mobile and IoT devices. To improve data security and privacy, recent advances in trusted execution environments (TEEs) (e.g., ARM TrustZone [5]), as a secure area in the processor, have provided an opportunity to revisit the security mechanisms protecting on-device computation and private user information. For optimized processing latency, one can leverage on-device accelerators (e.g., ARM Mali GPU) to provide significantly better inference performance than on-device processors (e.g., ARM Cortex CPU). Our benchmark in §2.2 shows that the accelerators can improve the inference latency by more than two orders of magnitude.

Realizing the promises of better latency and data privacy using the above techniques is easier said than done. The low-power/low-memory IoT setting creates unique challenges that existing solutions do not address: (1) When securing the computation on mobile and IoT devices with TEEs, we should minimize the trusted computing base (TCB) to reduce the attack surface [37]. (2) When performing inference computation, we should enable on-device accelerators for better performance as mobile CPUs are not as performant even with the model compression techniques [17, 19]. Unfortunately, existing efforts have failed to achieve one or more of these dimensions. Specifically, in mobile and IoT devices, we should not adopt the design from the cloud setting with x86 CPUs [15, 32, 39] to run hundreds of thousands of lines of code entirely in their TEEs (e.g., Intel SGX with 128MB secure memory). Furthermore, DL inference tasks using on-device accelerators have yet to be interfaced with TEEs to optimize security and performance simultaneously. Although some prior efforts [41, 45] design a secure path to use GPU on desktops or cloud servers, their designs fundamentally fail to provide a solution for embedded GPUs on mobile or IoT devices due to the the GPUs' architectural design difference. For example, desktop or cloud GPUs have their own memories, but embedded GPUs share the memories with CPU.

In this paper, we address these fundamental limitations by presenting SecDeep. SecDeep aims to achieve secure and performant DL inference on mobile and IoT devices by combining the best of both worlds—securing the GPU-accelerated inference with ARM TrustZone. Specifically, SecDeep protects data confidentiality during the entire on-device DL inference process, from the digitization of the raw sensor data until obtaining the inference results from accelerators. With ARM TrustZone, we extend the guarantees to the integrity of the inference model and the underlying computation

IoTDI '21, May 18–21, 2021, Charlottesvle, VA, USA

ACM ISBN 978-1-4503-8354-7/21/05…$15.00
https://doi.org/10.1145/3450268.3453524

in the accelerator. SecDeep demonstrates that we can adequately secure GPU-accelerated DL inference frameworks on edge devices[1].

In designing SecDeep, there are several technical challenges to address. (1) First, we must minimize the TCB size and the secure memory usage of (potentially) large DL inference models while not significantly degrading the performance or utility. This first challenge implies that we need to identify a portion of the DL model inference frameworks that can reside outside of the TEE while being able to leverage available accelerators. (2) Second, if we are to run a portion of the DL model inference framework outside of the TEE, we must ensure the integrity of the associated code. (3) Finally, we must further ensure the confidentiality of any data that needs to be passed this code residing outside of the TEE. We tackle these challenges as follows:

- *DL model computation split:* to reduce the TCB size of the DL inference model, we split the model computation base into a *confidential computing base* and a *nonconfidential computing base.* The confidential computing base is comprised of any code that interacts with the plaintext input data, e.g., matrix multiplication in a convolutional layer of a deep learning model. The nonconfidential computing base is comprised of any code that does not need to interact with the plaintext tensor data, e.g., GPU configuration code.
- *Runtime integrity checker inside TEE:* to verify any parts of the model and related code running in the untrusted environment, we utilize code signing. At compilation time, our enhanced compiler will sign the nonconfidential computing base with cryptographic hashing. When the model is loaded, the integrity checker sanitizes any access request to the nonconfidential computing base to preserve integrity.
- *Secure runtime data management:* although the nonconfidential code and data are exposed to the untrusted OS, the data are adequately encrypted when they go outside of the TEE. SecDeep adopts similar techniques from [7, 46] in which the kernel page table is treated as a user-space process page table to protect the code's integrity. SecDeep uses Format-Preserving Encryption (FPE) to hide the values of the data.

We implement SecDeep prototype using an IoT development board (HiKey 960) equipped with ARM TrustZone TEE and interface it with an embedded Mali G71 GPU through the ARM NN SDK. We minimize the TCB of the ARM NN framework from 232K sLoC to 1K sLoC. Our evaluation shows that SecDeep achieves $16\times$ to $172\times$ better inference latency than non-accelerator-based solutions for a representative set of on-device deep learning models (SqueezeNet [21], MobileNet V1 [20], MobileNet V2 [35], GoogleNet [36], YoloTiny [34], ResNet50 [18], and Inception [22]). Compared to unsecure GPU-based acceleration, SecDeep introduces $3\times$ to $5\times$ latency overhead as the cost for security.

Contributions and roadmap. We make the following contributions in this paper.

- We present SecDeep, the first system to our knowledge that provides secure and private deep learning model inference for mobile and IoT devices. (§3)
- We develop a technique to minimize the TCB of ARM TrustZone via proper DL computation split and ensure the integrity of the code and the associated split through secure bootup. (§4)
- We show how SecDeep can maintain the performance of DL inference on the edge by securely interfacing on-device accelerators with TEEs. (§5)
- We implement SecDeep for ARM-enabled mobile/IoT devices by interfacing the ARM TrustZone TEE with the ARM NN deep learning computation framework. We minimize the TCB of the ARM NN from 232K sLoC to ~1K sLoC. (§6)
- We evaluate SecDeep on a representative set of deep learning inference models and demonstrate that deep learning on the edge can be secure *and* performant. (§7)

[1]In this paper, we use edge devices and mobile/IoT devices interchangeably.

2 Background and Motivation

We begin by discussing the components of deep learning inference frameworks on mobile and IoT devices. We then describe the state-of-the-art hardware primitives to secure the computation on these devices and the limitations of a strawman solution that directly use these primitives to secure DL inference tasks.

2.1 DL Inference Framework on Mobile and IoT Devices

Due to the high demand for edge computing needs resulting from data privacy, network latency, and network bandwidth concerns, the most popular deep learning frameworks provide support to run deep learning model inferences at the edge directly from the raw input sensor data. For example, Caffe2, PyTorch, and TensorFlow Lite provide developers with an efficient way to perform deep learning inference at the edge before sending them to the cloud. More generic platforms, such as ARM NN, have emerged from hardware vendors, allowing the aforementioned deep learning frameworks to target common platforms with the same underlying computation base. ARM NN currently supports both Caffe and Tensorflow for a more generic optimized performance on ARM devices. These frameworks expose a common design: a framework composed of a neural network parser along with computation libraries that are optimized for specific operating systems such as Android [2] or iOS [3].

Neural network parser. Most on-device deep learning frameworks consist of a neural network parser [6]. Given a model that is generated using a supported framework such as Caffe or Tensorflow, the parser compiles the model into a graph representation that interfaces with the underlying computation libraries. This graph is constructed in a way that can be optimized for the backend execution.

Computation optimization. On-device deep learning frameworks also typically have backend execution frameworks to optimally execute the associated neural network graph representations depending on the available computation resources. For example, in ARM NN, if multiple backends are available simultaneously, the graph will be established such that multi-computing can be achieved efficiently. The resource optimizer validates the correctness of the input model and optimizes the resources needed for the model. It can remove redundant operations, reshape the data if necessary, reorder the graph constructed by the parser, and determine which acceleration methods to use.

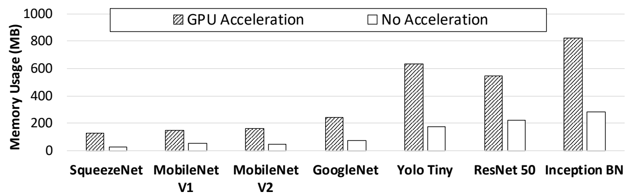

Figure 1. Maximum memory consumption of different Caffe models on an ARM device using ARM NN.

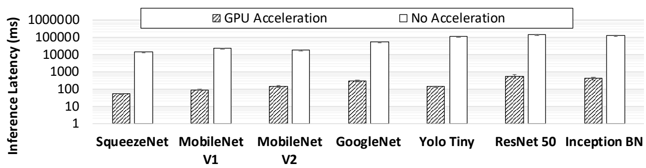

Figure 2. The average inference latency (log scale) of different Caffe Models on an ARM device using ARM NN.

Given these computation frameworks for deep learning inference on the edge, we now discuss how to secure the computation on these devices.

2.2 Secure Computation with TEEs

Although software-based cryptographic mechanisms allow for the protection and sanitization of this digitized data on edge devices, the data can still be leaked either prior to being encrypted or at the time of computation when the data are decrypted. To protect the computation, most mainstream CPU manufacturers provide hardware-assisted TEEs. For example, Intel provides secure guard extensions (Intel SGX) to establish per-application TEE, and AMD also provides secure execution environments (Secure Encrypted Virtualization) to protect the application's data confidentiality. However, the most popular trusted execution environment that provides access protection to peripherals on mobile/IoT devices is ARM TrustZone.

Trusted execution environments. Trusted execution environments (TEEs) are hardware protection mechanisms that isolate the memory into secure memory and unsecure memory. The secure memory can only be accessed by privileged code running inside the TEE while any code can implicitly access the unsecure memory. In ARM TrustZone, the secure memory code resides in secure memory—referred to as the *Secure World* (SW), whose high privilege is designated by setting a special ARM instruction *SMC*. The unsecure code resides in unsecure memory—referred to as the *Normal World* (NW). The context switch between SW and NW is done through a Secure Monitor (SM).

Although one may trivially assume that computation for a large model such as a deep learning model could be placed within the secure world of a TEE, we discuss several reasons why this is a strawman solution.

2.3 Strawman Solution on Secure Inference

A straightforward approach to provide data confidentiality and code integrity for DL inference tasks is to put the entire deep learning framework inside the trusted execution environment. While this sounds a feasible solution, there are two fundamental design flaws associated with it:

Table 1. Lines of code for different deep learning frameworks on mobile/IoT as well as a breakdown for the ARM NN deep learning inference framework. The table highlights the small percentage of code dedicated to the privacy-sensitive tensor computation.

Framework Name	sLoC
TensorFlow Lite [38]	404K
Caffe2 [11]	368K
PyTorch [33]	191K
Deeplearning4j [13]	690K
ARM NN^2 [6]	232K

(a) Lines of code for different deep learning frameworks on edge.

ARM NN No Accel.	sLoC	Pct.
Tensor Preparation	109.9K	90.2%
Tensor Computation	11.95K	9.8%

ARM NN GPU Accel.	sLoC	Pct.
Tensor Preparation	232K	99.95%
Tensor Computation	114	0.05%

(b) Lines of code breakdown for ARM NN deep learning inference framework.

- **Excessive secure memory usage on accelerator-enabled DL inference:** Mobile and IoT devices are typically memory constrained on the order of up to a few Gigabytes. Secure memory is generally limited to tens of Megabytes per application as the initial allocation is deducted from the normal operating system's unsecure memory allocation. For example, in ARM TrustZone, the memory configuration is done at boot time, so the more secure memory, the less unsecure memory for an already resource-constrained device. In this paper, we follow the lead of prior works [1] and allocate only tens of Megabytes of secure memory to provide the same level of protection—limiting the feasibility of running large deep learning inference models within secure memory. For instance, Figure 1 shows the maximum memory consumption of running different Caffe models on an ARM device using ARM NN. Without any GPU acceleration, the smallest model (SqueezeNet) consumes 28 MB of memory. With accelerators enabled, the memory consumption for a model inference could shoot up to 821MB. Note that, without accelerators, the performance of on-device deep learning inference will be degraded by several orders of magnitude—as depicted in Figure 2.
- **Large trusted computing base that increases attack surface:** Table 1a shows that deep learning frameworks could bring hundreds of thousand lines of code. If the whole framework is placed inside TEE, the total trusted computing base (TCB) size will be tremendously large and introduce unnecessary attack surfaces. Upon analysis of the ARM NN deep learning inference framework as shown in Table 1b, we found that, without acceleration, typically 90.2% of the framework code is for tensor computation preparation or for performance optimization, and only about 9.8% is dedicated to mathematical tensor computation that changes the values of the input tensor data and yields the values of the output tensors. With acceleration enabled, the computation and preparation code makes up about 99% of the code. Furthermore, the output of the computation preparation and the performance optimization code only depends on the size of the input rather than the values of the input. Thus, our design aims to leverage the hardware-assisted execution environment to reduce the TCB size while still sanitizing the access to accelerators with high security level by only putting the tensor value computation code inside the TEE while leaving the tensor preparation code, the resource configuration code, and the optimization code outside of TEE.

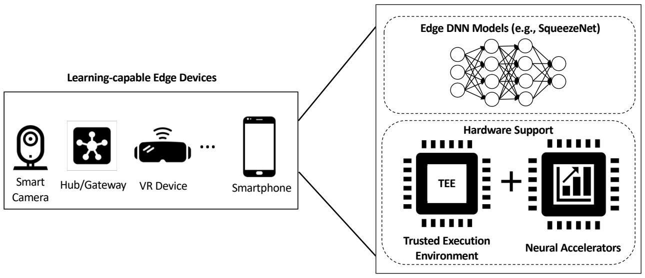

Figure 3. The system model of SecDeep. We aim to secure DL model inferencing on IoT edge devices that are enabled with a TEE and possibly on-device accelerators.

3 Overview

In this section, we describe the scope and workflow of SecDeep before discussing the main technical challenges and insights.

3.1 Problem Scope

System Model. SecDeep is a general secure framework residing on mobile and IoT devices (e.g., IoT gateway) to protect deep learning inference tasks, as shown in Figure 3. We assume the devices are enabled with hardware support for TEEs as well as on-device neural accelerators. We further assume that the DL model provider puts no effort on ensuring the confidentiality of the DL model and its underlying computation framework, e.g., the model may be a publicly available DL model such as SqueezeNet with an open-source computation framework such as ARM NN. To verify the integrity of the DL models in SecDeep, we expect the provider to supply the hashes of the authentic models using cryptographic hash functions (e.g., SHA-3). During inference tasks, SecDeep is designed to achieve (1) user data integrity and confidentiality, (2) the integrity of deployed DL models, and (3) the integrity of supporting codebase, e.g., TensorFlow Lite, ARM NN.

We envision IoT device vendors and IoT cloud operators being early adopters of such a framework, given the supporting evidence that inference tasks are being pushed closer to the edge. As such, we consider two system scenarios to deploy SecDeep. (1) In the first scenario, an IoT cloud backend needs the inference information from a mobile or IoT device on the edge. The backend sends the request to the edge devices, and the edge device only returns the final inference output from SecDeep instead of the raw, potentially large sensor information to preserve user privacy and reduce network latency. (2) The second system scenario is a mobile or IoT edge device that needs to perform end-to-end local inferencing without needing to access or share any information with the IoT cloud backend. With this system model, we now consider the threat model for SecDeep.

Threat Model: SecDeep considers a strong adversary that aims to compromise the operating systems in order to intrude, forge, and modify the inference tasks, as well as to steal user data from the non-protected processes. Thus, we cannot trust any part of the software stack—including the OS—that resides outside of a TEE. As other concurrent efforts that offload computations to the TEEs [7, 26, 46], we assume the trustworthiness of a TEE since our scope is on how to efficiently leverage TEEs to secure accelerator-enabled DL inference computation. Thus, the mitigation and prevention of side-channel attacks, denial-of-service attacks, and cyber-physical attacks are outside the scope of this paper. Given the system and threat models of SecDeep, we then summarize the goals of its design.

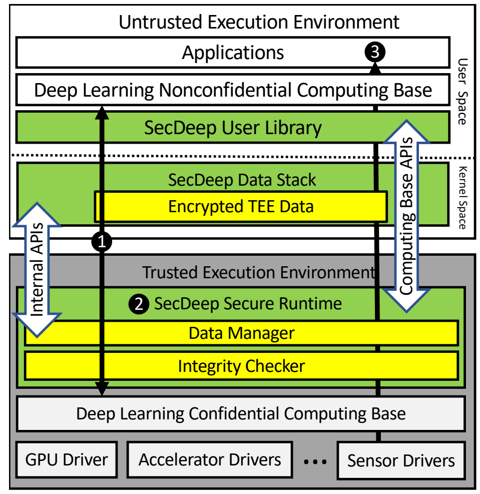

Figure 4. The architecture overview of SecDeep. The green and yellow shaded areas are the components of SecDeep framework and the dark grey shaded areas are the TEE. SecDeep's workflow has three steps: ❶ transforming the deep learning inference computation base; ❷ secure, confidential, and performant execution; and ❸ securing the inference results.

Goals: SecDeep aims to protect *data confidentiality* during inference, starting from the digitization of the raw sensor data until obtaining the inference results. This protection implies that the confidentiality of any intermediate, generated metadata will also be protected. Further, SecDeep aims to ensure the *integrity* of the inference code and the associated model. Finally, SecDeep aims to utilize a *minimal trusted computing base* size with *reasonable inference latency and energy consumption* while incurring *no inference accuracy loss.* We illustrate how these design goals are achieved by walking through the SecDeep workflow.

3.2 SecDeep Workflow

Figure 4 shows the architecture of SecDeep. At a high-level, SecDeep can be broken down into three steps: ❶ transforming the deep learning inference computation base for trusted execution, ❷ secure, confidential, and performant execution of the deep learning inference model, and ❸ securing the inference result.

1. DL model computation transformation for trusted execution: To identify which components of the DL inference code and data should reside within the TEE, we first split the deep learning libraries into two parts: a *confidential computing base* that executes in a TEE, and a *nonconfidential computing base* that executes in the untrusted execution environment. Generally, any code that only requires access to tensor metadata, e.g., tensor shapes, rather than the plaintext tensor data, will be designated to the nonconfidential computing base. Otherwise, the code will be designated as confidential. The confidential and nonconfidential computation base are annotated at a functional level with preprocessor directives to enforce this computation split at compile-time (as is done in Figure 5a). Given a split computation base, the SecDeep system is initialized through a secure boot that ensures the integrity of

[2]including GPU acceleration backends in ARM Compute Library

the entire SecDeep code base [4] as well the confidentiality of the designated code. With a secure boot in place, we can now describe how SecDeep handles the aforementioned split computation base at runtime.

2. Secure execution: After SecDeep properly loads the framework code, SecDeep starts to load user data and perform inference tasks in the following steps.

- First, the user data (e.g., sensor data) are securely loaded into the TEE via protected drivers. Thus, the confidentiality and integrity of the data are guaranteed.
- Second, once the data are inside the TEE, SecDeep's *data manager* decides if any nonconfidential code or data needs to be exported to the nonconfidential computing base due to memory footprint limitation. When some data are set to be exported, SecDeep encrypts them inside the TEE to ensure confidentiality.
- Third, to perform the inference, the nonconfidential computing base uses encrypted data with the model parameters for the current neural network layer to configure the tensor information and send it back to the confidential computing base. Inside the TEE, SecDeep then decrypts the data and begins executing the current layer collaboratively using a protected accelerator (e.g., an embedded GPU). After the results have been computed for the current layer, SecDeep will repeat the same procedure until the inference process is complete.

3. Secure output: Because the inference process up until the output is secure and confidential, securing the output is trivial, e.g., the results can be signed or encrypted before being sent back to the requester. Therefore, we focus on the challenges and design of the first two components of the SecDeep framework.

3.3 Challenges and Key Insights

Given this workflow, we highlight the key design challenges and our associated approach for each.

Challenge 1: Managing the TCB size for the TEE. Performance with limited secure memory is constrained. SecDeep needs to use limited secure memory to provide protections for the input data along with any data generated throughout the inference process while providing a performant deep learning inference framework. SecDeep utilizes Format-Preserving Encryption (FPE) along with an on-demand table to fulfill this requirement, as described in Section 5.2. Although the on-demand table requires more memory, our experiments show that it significantly reduces the necessity for encryption and decryption of often-used values and, thus, significantly reduces overhead.

Challenge 2: Ensuring code integrity outside of the TEE. Any code cannot be modified by the compromised OS after it is loaded into memory outside of the TEE. SecDeep treats the kernel page table as a user-space process page table. This allows SecDeep to forward every modification from the kernel page table after the system boots up to the TEE to ensure the integrity of the inference code running outside of the TEE as described in Section 5.1.

Challenge 3: Ensuring data confidentiality outside of the TEE. Because some of the code, i.e., the nonconfidential computing base, will reside outside of the TEE, there is an inherent risk when computing with confidential data. Because this code only requires information about the *properties* of the confidential data, e.g., data size, the TEE generates a placeholder value for the confidential tensor data. In particular, SecDeep uses format-preserving encryption (FPE) to generate encrypted data with the same size and length as the plaintext tensor data.

```
/* Sample Confidential Computing Base Code */
float[] CPUActivation[CONFIDENTIAL] (inputTensor,
                                     tensorInfo,
                                     activationFunc,
                                     float weight, float bias)
{
    int tensorSize = tensorInfo.getTensorSize();
    float[tensorSize] results;
    for (int i = 0 ; i < tensorSize ; i++) {
        float tensor_value = inputTensor.getData(i);
        switch(activationFunc) {
            case Linear:
                results[i] = weight * tensor_value + bias;
                break;
            case SoftReLu:
                results[i] = log(1.0f + exp(tensor_value));
                break;
            // Other Activation Function Computations
            ...
        }
    }
    return results;
}
```

(a) Confidential computing base code.

```
/* Sample Nonconfidential Computing Base Code */
void configureGPUActivationLayer(inputTensor, *layerInfo)
{
    // Build config options. e.g. DataType
    int dimension = inputTensor.data().dimension();
    DataType type = inputTensor.type();
    ...

    String configured_option = build_options(dimension, type, ...);
    String layer_id = "activation_layer_1";
    String gpu_kernel = "activation_layer";
    // Set the GPU configuration
    layerInfo.setGPUConfigure(configured_options, layer_id, gpu_kernel);
}
```

(b) Nonconfidential computing base code.

Figure 5. Example snippets of confidential and nonconfidential deep learning inference computation code. Confidential code requires access to the plaintext tensor data, while nonconfidential code only requires information about the tensor metadata, e.g., the dimensions.

4 Transforming Inference Computation for Secure Execution

We now describe the design of the first major component of the SecDeep workflow. We describe how the deep learning computation base is split into confidential and nonconfidential codes. We then explain how SecDeep ensures the integrity of the code and the associated split at bootup.

4.1 Splitting Deep Learning Computing Base

As discussed in Section 2, placing the entire DL inference computation framework within a TEE is infeasible and only increases the attack surface of the TEE. Thus, given a DL inference computation framework, we need to identify a minimal set of code that needs to be protected inside TEE. In this case, we aim to protect only code that is designated as *confidential.* This code will be annotated at development time so that it can be separated from the nonconfidential code at compile-time and loaded into the TEE.

Deep learning confidential computing base. To minimize the code running inside TEE, we design the confidential computing base to be composed of the deep learning inference computation that requires access to the unencrypted, plaintext values of the tensor data. For example, Figure 5a shows a snippet of code for different activation functions for a neural network. The functions require access to the tensor values (Line 10) to calculate the activation output for the next layer of the neural network. The variable `inputTensor` cannot be replaced by any placeholder value without losing the fidelity of the original computation. However, as per Section 2, we find that this confidential base typically makes up a very small percentage of the overall computation base. Line 2 also shows an example of how a developer may annotate a function as confidential with a preprocessor directive (`[[CONFIDENTIAL]]`[3]).

Deep learning nonconfidential computing base. Any code that does not require access to the plaintext, unencrypted tensor data is designated as *nonconfidential* and will reside outside of the TEE in the untrusted software stack. For example, Figure 5b shows a snippet of code that observes the input tensor shape and configures the GPU accordingly. The only interaction with the confidential variable `inputTensor` involves extracting the variable dimensions and data type on Lines 5 and 6. The GPU configuration call does not require access to the tensor values. Therefore, this code can easily be refactored such that the tensor data is replaced with a placeholder variable that has arbitrary values with the same shape, size, and data type. This maintains the fidelity of the original function and would not compromise the integrity of the overall computation.

Despite this code being designated as nonconfidential, its integrity is still imperative to the overall computation base. We describe how we maintain its integrity in Section 5.1. Before doing so, an underlying assumption is that SecDeep's base confidentiality enforcement mechanism, along with the peripherals, has been secured upon booting the system. We describe how a secure path can be established from sensor peripherals to accelerators in the following subsection.

4.2 Securing the Path from Sensor Peripherals to Accelerators

SecDeep needs to create a secure path such that the raw sensor data along with any generated, intermediate metadata are protected when interfacing with accelerators. To achieve such protections, SecDeep utilizes the properties of TEE to disable access to the protected sensors and accelerators from the untrusted OS. SecDeep configures the memory-mapped IO addresses of the sensors and accelerators into the secure memory of TEE such that, upon booting up, those TEE-protected memory-mapped IO addresses can only be accessed by the privileged code inside TEE, but not the untrusted OS. For example, in ARM TrustZone, if the memory-mapped IO addresses for the sensors are configured as secure memory addresses before the system boots, any access to those addresses from the untrusted will be trapped to a higher level execution (i.e., Boot Loader Stage 3 (BL3)) through exceptions, and the secure world in the ARM TrustZone is able to decide whether such access requests should be granted or not.

[3]This syntax is similar to the syntax used by a previous work that annotated code that can parrot-transformed (approximated) by a neural network [14].

5 Secure and Performant Inference Execution with Accelerators

In this section, we describe how the SecDeep secure runtime provides runtime protection for the entire deep learning inference framework. SecDeep collaboratively works with the data stack to serve as secure storage outside of TEE when any datum is exchanged between the TEE and the untrusted execution environment. The SecDeep secure runtime is comprised of two major components: a runtime integrity checker and a data manager. To enable both components, we later detail the secure API exposed by SecDeep that facilitates the confidential data exchange between trusted and untrusted computing bases. The corresponding data sanitization of the secure runtime enables SecDeep to securely leverage available accelerators without leaking private data.

5.1 Runtime Integrity Checker

To confirm the integrity of the code running in the deep learning nonconfidential computing base as well as the deep learning model, we design a checker located inside of the TEE for verification. The key intuition of the integrity checker's design is the verification of the hash value of both the model and the code in a trusted mode. The integrity checker works together with an enhanced compiler that signs the nonconfidential computing base code at compilation time to make sure the integrity is preserved when loading the code and the model. After the code and the model have been loaded, the integrity checker sanitizes any access request to the memory of the nonconfidential computing base to make sure the untrusted OS cannot modify the code in the nonconfidential computing base after the secure boot. This sanitization is enforced by trapping the modification of the kernel page table into a higher-level model such as BL3 in ARM. We next describe the design details of the integrity checking mechanisms for both the nonconfidential computing base code and the associated DL model.

Nonconfidential computing base code integrity. To detect the code integrity before any code is loaded into the memory of the nonconfidential computing base, we first modify the associated compiler to hash the nonconfidential computing base code running outside of the TEE. To hash the code at compilation time, the compiler identifies what code belongs to the nonconfidential deep learning computing base by excluding any code that has been annotated as confidential (e.g., Figure 5a). The extracted nonconfidential computing base is then hashed accordingly at compile time. At runtime, SecDeep's integrity checking service temporarily stores the hash value into the secure memory using any key exchange algorithm (e.g., as Diffie–Hellman algorithm). The integrity checker then allocates memory regions for the nonconfidential computing base. After the code has been loaded, the integrity checker computes the hash of the loaded code and compares it with the hash value supplied at compilation time to verify the code's integrity.

To ensure that the integrity of the loaded code is protected from the modifications by the untrusted OS during execution time, we hide the code pages from the OS kernel—as depicted in Figure 6. We first configure all kernel page tables to be *read-only* during the aforementioned secure boot. At runtime, if the OS needs to modify a kernel page table, e.g., modifying the page table base registers (PTBR) in ARM, such a request will be trapped into the TEE. Within the TEE, SecDeep's integrity checker will ensure that the page table modification will not result in mapping a kernel page table into a nonconfidential computing base memory regions via a table walking attack [10, 23].

In this scenario, another potential attack is to swap the page table with a compromised one such that the nonconfidential computing base code will be accessed through the new kernel page table. To protect against such attacks, SecDeep disables the base registers that modify the kernel page table by removing the page swapping instructions and trapping the write instructions into TEE. SecDeep then checks the access to ensure that the new page table will be mapped into sensitive memory regions when a new kernel module has been loaded. Although similar approaches have been used in prior works [7, 46], these approaches need additional mechanisms to perform data confidentiality verification at the same time. Because SecDeep has pre-processed the data confidentiality issue, we simplify their approach to obtain better performance with the same level of security.

Finally, SecDeep needs to make sure the exception handler outside of the TEE is not able to make modifications to the kernel page table when an exception has been trapped by a higher privileged code. Similar to its code integrity protection techniques, SecDeep modifies the exception handler such that any exceptions will be trapped and forwarded to the TEE. The TEE will examine the code to ensure that the code does not contain any modifications in the memory regions from either the nonconfidential computing base or the secure buffer. If the exception does not violate the code's integrity check, the exception will be returned back to the untrusted TEE. The saved registers will be restored for further execution. However, if an exception contains any modifications to the sensitive regions, the exceptions will never be returned and the user will be notified of the malicious behavior.

Deep learning inference model integrity. Although the deep learning inference model will reside in the same untrusted memory as the nonconfidential computing base, the design of the integrity checking mechanism will require a slightly different approach. As per our system model, we assume that the provided model will be stored in the storage media or may be downloaded from the internet. In either case, we assume that we will also be provided a hash of the authentic model using cryptographic hash functions. Thus, SecDeep's integrity checker focuses only on detecting the model's integrity and not the protection of the model before it is loaded in memory. However, if one wants to protect the model from being modified in a future implementation, secure communication can be established with the cloud through a TEE [12].

Further, since our system model does not require the DL model to be confidential—as well as the aforementioned constraints of secure memory, SecDeep loads the model in the nonconfidential computing base outside of the TEE. Instead of verifying the integrity of the DL model against its hashed value within the TEE, SecDeep utilizes a mechanism provided by the TEE to set up a read-only (nonconfidential) buffer for the non-TEE code—while the TEE code has read and write privileges to the buffer. To ensure a secure buffer design, we take a similar approach as the nonconfidential computing base and hide the memory region from the untrusted OS kernel. In particular, we change the mapping from the kernel page table to the buffer region.

Given the secure buffer design, SecDeep's integrity checking service computes the hash of the model and passes the signature to the nonconfidential computing base through the read-only buffer.

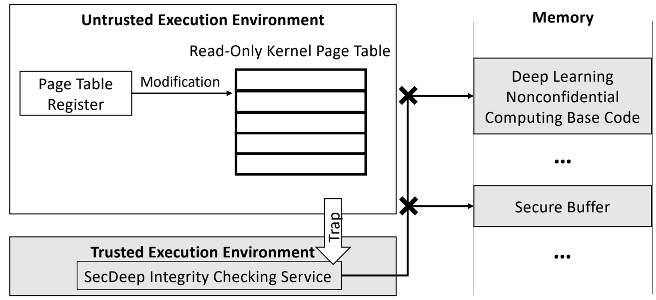

Figure 6. When the untrusted OS tries to modify the kernel page table, the request will be trapped to the TEE to make sure the mapping does not map to sensitive memory regions. The shaded areas are the sensitive memory regions.

When the model is loaded, the nonconfidential computing base checks whether it has been verified by the integrity checker.

5.2 Data Management

The final component of SecDeep's secure runtime is the data manager, which primarily manages the confidential data communication between the deep learning nonconfidential computing base and the secure memory inside the TEE. In particular, SecDeep's data manager is responsible for providing the associated data sanitization by replacing the raw data with encrypted data that has the same dimensions—as shown in the sample snippet in Figure 5b. Further, if the secure memory is running low, the data manager is also responsible for encrypting any data that needs to be stored outside of the TEE in the untrusted data stack. Hence, the data manager design has two requirements. First, the original data's confidentiality cannot be leaked. This means that the attackers cannot reverse engineer any secrets from the supplied encrypted data. The second requirement is that the dimensions of the original data must be the same as the supplied data. To satisfy these needs, SecDeep uses a format-preserving encryption (FPE) [9] function to sanitize the data. FPE encrypts the plaintext value of each basic element of the tensor data while ensuring the dimensions of the data are retained. For example, if the tensor data represents an array of integers, the FPE encrypts every integer of the array to create an array of encrypted integers. This array has the same length and data size as the plaintext tensor data array.

However, if we simply encrypt and decrypt the data using FPE whenever there is a context exchange between the trusted and untrusted execution environments, this implies that the total computation for every confidential value will be doubled, i.e., the data will need to be encrypted when exiting the TEE and decrypted upon entry. A prior study [26] confirmed that there is indeed a large overhead incurred from such frequent swapping. Hence, to provide efficient swapping, SecDeep utilizes a table inside the TEE to maintain the mapping of encrypted data to the raw data. This method will ensure the "decryption" time to be constant, i.e., it will have a complexity of O(1) by simply referring to the table if the encrypted datum has been created. If an encrypted datum is designated to enter to the TEE from the untrusted execution environment, SecDeep's data manager first refers to the table to retrieve the original plaintext datum. If the datum cannot be found in the table, the data manager performs a decryption method to obtain the original raw datum. This table-mapping approach is summarized in

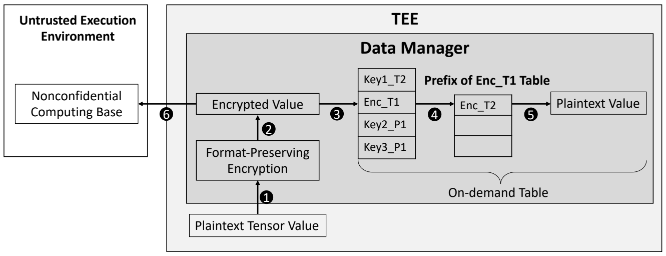

Figure 7. The sanitization procedures of the confidential raw data before leaving the TEE. The shaded areas are trusted. The plain datum is first encrypted using FPE (❶-❷), then the encrypted datum is stored as the key in the on-demand table with the plaintext datum as the value (❸-❺). The sanitized datum is then delivered to the nonconfidential computing base (❻).

Figure 7: when any confidential data needs to exit the TEE, it is first encrypted using FPE (❶-❷), and then stored in the table using the encrypted data as the key and the plaintext data as the value (❸-❺). Equivalent plaintext values will have the same encrypted values.

On-demand table maintenance. As discussed, the table will inevitably require extra usage of secure memory. There are a couple of optimizations SecDeep adopts to reduce the extra secure memory consumption. First, SecDeep's data manager is to only maintain an entry as long as it is needed. An entry is created when the raw datum needs to leave the TEE. When the raw datum is encrypted and exfiltrated, a count of the datum entry increase by one (starting from zero). When the datum is returned into the TEE to be encrypted, the counter is decreased by one. If the count becomes zero, the entry is removed from the table. Second, if the secure memory is full, SecDeep's data manager unit uses a cache evicting algorithm (e.g., least frequently used) to release more memory and move the encrypted data outside of the TEE. When one layer's computation is finished, all of the intermediate data will be destroyed unless they are needed for the next layer, which will be indicated by the nonconfidential computing base's results.

The traditional implementation of the table data structure usually reserves enough space at initiation time and increases the size by copying the existing table into a larger memory chunk. Due to the constraints of the secure memory, we design an *on-demand* table mechanism to save the mapping of the decrypted data. Inspired by traditional kernel OS design, we design SecDeep's on-demand table to be segmented into small chunks by having a multi-level table (❸-❺). The table entry will only be created when data needs to be stored but does not need to reserve a large space at the initiation time like the traditional table, which is able to ultimately save secure memory usage, and this design also does not require a large consecutive memory if the key-pairs are large.

5.3 Confidential Data Exchange through Secure API

As depicted in Section 3.2, SecDeep's secure API enables confidential data exchange between various components of SecDeep, including those residing both inside and outside of the TEE. Table 2 summarizes the five secure API functions exposed by SecDeep. The API is split into two categories: 1) computing base API that

```
1.  /* Nonconfidential Computing Base */
2.  void prepartion(model)
3.  {
4.      loaded_model = c_model_init(model);
5.      for (auto layer: loaded_model)
6.      {
7.          tensor_info = c_tensor_request(layer);
8.          // Computation such as configurations
9.          ...
10.         c_output_result(layer);
11.     }
12. }
```

(a) The nonconfidential computing base sample code that configures the input tensor for each layer.

```
1.  /* Confidential Computing Base */
2.  void model_verification(model)
3.  {
4.      loaded_model = i_model_load(model);
5.      hash_verify(loaded_model);
6.  }
7.
8.  void data_store(data)
9.  {
10.     encrypted_data = encrypt(data);
11.     i_data_store(encrypted_data);
12. }
```

(b) The confidential computing base sample code to verify the model integrity, and store the necessary data outside of TEE.

Figure 8. Sample code of how confidential computing base and nonconfidential computing base interact with other components using secure API.

enables the communication between the confidential and nonconfidential computing bases, and 2) the internal API that enables the communication between SecDeep's data stack and secure runtime.

Computing base API. The computing base API calls are used to send tensor information between the confidential computing base and the nonconfidential computing base. For instance, Figure 8a provides a sample code snippet for a secure API request from the nonconfidential computing base to the confidential computing base. Once the nonconfidential computing base has configured the input tensor and the resource requests to use a GPU, the code in the nonconfidential computing base will call `c_output_result(layer)` (Line 10) to pass the results to the confidential computing base via SecDeep's secure runtime data manager using a secure buffer.

Internal APIs. The internal API is used to exchange the data stored on the data stack in the untrusted execution environment with the data in SecDeep's secure runtime. For example, Figure 8b shows a sample code snippet of the integrity checker verifying and signing the deep learning model. The secure function `i_model_load (model)` is called to load the model from the data stack (Line 4). The API is designed to use secure monitor code (SMC) to establish a secure buffer such that a malicious OS cannot modify the contents as described in Section 5.1 by properly hiding the memory region from the kernel page table.

6 Implementation

In this section, we discuss how we prototype the design of SecDeep.

Table 2. Summary of secure API in SecDeep.

Category	Name	Description
Computing Base API	`c_tensor_request(layer)`	Request encrypted tensor data from the deep learning confidential computing.
	`c_output_result(layer)`	Send results of the specific layer from the nonconfidential to confidential computing base.
	`c_model_init(model)`	Requests SecDeep secure runtime to load the model.
Internal API	`i_data_store(data)`	Store encrypted data from the deep learning confidential computing base.
	`i _model_load(model)`	Send the model to SecDeep secure runtime for integrity verification.

Table 3. Lines of code implemented for SecDeep.

Model Name	Repo	sLoC
User Library	ARM NN	694
	ComputeLibrary	126
Integrity Checker	ATF	897
	OP-TEE	69
	Linux	457
	LLVM Compiler	70
Data Manager	OP-TEE	1635
Total		3948

6.1 System Setup

We prototype SecDeep using ARM NN with a Caffe deep learning model on a HiKey960 Android development board. The Hikey960 board was enabled with an embedded GPU–a Mali G71 MP8 graphics processor, and ARM TrustZone support. For the TEE, we use ARM Trusted Firmware (ATF) with Open Portable Trusted Execution Environment (OP-TEE) within ARM TrustZone. Because the driver of Mali GPU is not fully open-source, we have to simulate the secure access in our implementation. Although we prototype SecDeep on an ARM A-series dev board, SecDeep is portable to lower-end ARM processors such as Cortex-M23 and Cortex-M33 processors with ARM TrustZone.

We implement the user library inside ARM NN and the associated ARM Compute Library to provide the secure API. We implement the integrity checker within the OP-TEE and modify the Linux kernel. We also modify the LLVM compiler to perform the confidential code extraction and provide the signatures for the codebase. We implement the data manager inside OP-TEE. Table 3 summarizes the 3.9K lines of code for the implementation.

To evaluate our framework, we built a deep learning inference application using ARM NN to perform image classification using different Caffe models supplied for the Hikey960 reference board. We now detail how each component of the SecDeep is implemented.

6.2 Deep Learning Computing Base Split Annotation

We leverage ARM NN and ARM Compute Library with the support of Caffe inference framework to build the confidential computing base and nonconfidential computing base. Upon analysis, we found that most of the ARM NN code is for resource preparation such as building the computation nodes of a special graph or the model parser that loads the model into memory. The ARM NN documentation revealed that the tensor data can only be accessed through the function `Map()` inside the structure `ITensorHandle`. Thus, we were able to script the identification of all the code that uses these functions and analyze whether they are using the sensitive tensor data. We found that the only functions of the ARM NN code that needed the aforementioned confidential designation were the functions associated with the tensor input layers that process all the input data and other tensor metadata such as padding.

6.3 Secure Runtime

We build the secure runtime inside ARM TrustZone using both OP-TEE OS and ARM Trusted Firmware (ATF). The OP-TEE OS is responsible for processing the model integrity checking. The ATF traps all of the kernel page table modifications and computes the code hashing. The ATF is also responsible for checking whether the kernel modification will map to a memory region that holds nonconfidential computing base code and data and the secure buffer.

Integrity checker. We implement the runtime integrity checker for both the deep learning model and the code inside the nonconfidential computing base. We implement an MD5 hash mechanism inside OP-TEE to compute whether the model has been tampered with while loading it onto the inference framework.

For the nonconfidential computing base code integrity checker, we first modify the Linux kernel page table entry functions such as clear_pte_bit() and set_pte() so that, every time these functions are called, they will be trapped to Boot Loader Stage 3 (BL3) through SMC. When the BL3 handler functions in ATF receive such requests, the BL3 handler functions determine which request they need to handle. If the OS tries to load code into the nonconfidential computing base, the BL3 handler functions use SHA1 to compute the hash of the code and compare it with the compiler-supplied hash. If the kernel page table modification request should not load the code into nonconfidential computing base, the BL3 handler functions walk through all of the page tables to ensure that the modification does not map to the nonconfidential computing base nor the secure buffer.

Offline signature generation for nonconfidential computing base code through the LLVM compiler. To generate the signature for the nonconfidential computing-based code, we extract any code that was not designated as confidential using the aforementioned annotations. We modified the LLVM compiler such that during the code emission stage, when the LLVM compiler detects the confidential designation, it computes the hashing for the code block for that function. We use a SHA-1 hashing algorithm to do the hashing computing and verification for the instructions within the designated code blocks. We then sign the hash values and store the signature into the data segment of the program. The program is later loaded into secure memory for verification.

6.3.1 Data Manager.

We implement the runtime data manager inside OP-TEE. We use Advanced Encryption Standard (AES) with Counter (CTR) mode as the format-preserving encryption (FEP) method because AES-CTR provides the same length of the output as the input. We also implement a two-level table for our on-demand hash table, where the key is the encrypted data and the value is the plaintext data. We maintain the table using the least frequently used (LFU) mechanism. We also evaluate different maximum table size values allowed before adding a new entry in the next section.

Table 4. Benchmark models used for evaluation.

	Model Name	Model Size
Light	SqueezeNet	5 MB
	MobileNet V1	17.1 MB
	MobileNet V2	14.4 MB
Med.	GoogleNet	28.3 MB
	Yolo Tiny	65.6 MB
Heavy	ResNet-50	102.5 MB
	Inception BN	137.8 MB

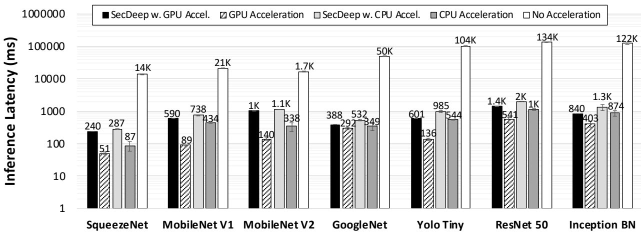

Figure 9. The overall latency introduced by SecDeep and the comparison with CPU acceleration and GPU acceleration.

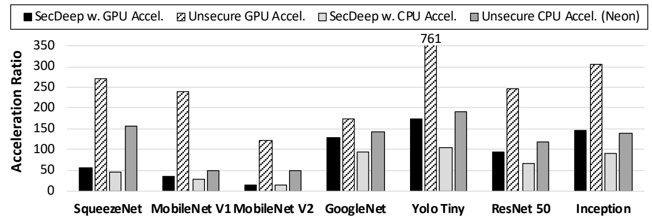

Figure 10. The acceleration ratio of each acceleration method and SecDeep in respect of execution the model without acceleration

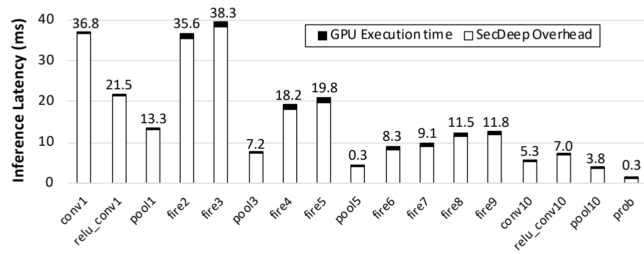

Figure 11. The SecDeep overhead breakdown for each layer of SqueezeNet.

7 Evaluation

In this section, we will discuss how we evaluate SecDeep with various parameters to show that (1) SecDeep achieves secure DL model inferencing with superb latency (e.g., 172× better than CPU) via secure GPU acceleration, and (2) incurs only a minimal TCB size and computation/energy overhead.

Benchmark Models. Our implementation of SecDeep supports Caffe models. Thus, we chose to evaluate popular Caffe models with varying size as listed in Table 4. We first chose 3 popular, lightweight (less than 20 MB) models: SqueezeNet [21], MobileNet V1 [20], and MobileNet V2 [35]. We then evaluated 2 medium weight (less than 100 MB) models: GoogleNet [36] and Yolo Tiny [34]. Finally, we used 2 heavy weight (greater than 100 MB) models: ResNet50 [18] and Inception BN [22]. These are a representative set of on-device models with varying capabilities.

Trusted Computing Base Size. Based on our implementation, the current TCB size is 1015 sLoC, where 901 sLoC comes from OP-TEE and 114 sLoC comes from ARM NN. Compared to the total computing base size 232K[4] of ARM NN, SecDeep has provided a tiny TCB size. All of the GPU kernel code resides outside of the TEE because the kernel code is only configured by the CPU.

7.1 Inference Latency

We run inferencing experiments for all of our benchmark models at least 10 times to measure the average latency using 16MB of secure memory. This is a sufficient amount of memory to support the minimal TCB within the TEE (OP-TEE's kernel is typically a few MB) as well as an on-demand table with 1M entries.

Overall latency. We run the inference experiments for each model using SecDeep with GPU acceleration, unsecure GPU acceleration, SecDeep with CPU acceleration, unsecure CPU acceleration, and no acceleration. We compute the average inference latency for each model and summarize the results in Figure 9. Our experiment shows that, although SecDeep with GPU acceleration is slower than unsecure GPU acceleration, it is still comparable to unsecure CPU acceleration and SecDeep with CPU acceleration—it is even faster than unsecure CPU acceleration for the Inception BN model. Most importantly, SecDeep is significantly faster than the case where no acceleration is enabled. This result also shows that the inference latency is not fully proportional to the size of the model. For example, MobileNet V2 is only about one-fifth of Yolo Tiny's model size, but MobileNet V2's inference latency is slower than Yolo Tiny. This is because MobileNet V2 generates more intermediate results and the table hit ratio is lower, resulting in more decryption computations.

In terms of acceleration ratio, Figure 10 shows, for the best case (ResNet-50), SecDeep is able to accelerate up to 172 times more than the case with no acceleration. Even in the worst case (MobileNet V2), SecDeep is able to accelerate 16 times faster than when no acceleration is enabled. Our results have shown that using SecDeep can achieve both acceptable latency and enable secure protection.

Overhead breakdown. We further break down the overhead of SecDeep introduced for different layers of a deep learning model. As shown in Figure 11, we accounted for the overhead in each of SqueezeNet's layers. The results show that the overhead of SecDeep mainly comes from the TrustZone execution, i.e., the world context switch time and the encryption. Furthermore, the first few layers have higher overhead than the last several layers. This discrepancy is due to the fact that the last few layers generate less intermediate data and because the cached table hit rate is high. Further, the overhead of SecDeep is not completely proportional to the size of the tensor input of each layer. For example, if the input tensor size of `fire4` and the input tensor size of `fire3` are both $55 \times 55 \times 128$, but the latency of `fire4` is significantly smaller than the latency of `fire3`. These results also demonstrate that future optimizations could focus on how to store the values of the encrypted data in a table such that the table hit rate can be high enough to benefit the overall performance.

[4]This only counts the GPU acceleration code.

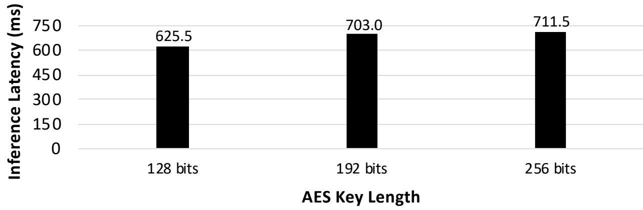

Figure 12. SecDeep latency of different AES key lengths for MobileNet V1.

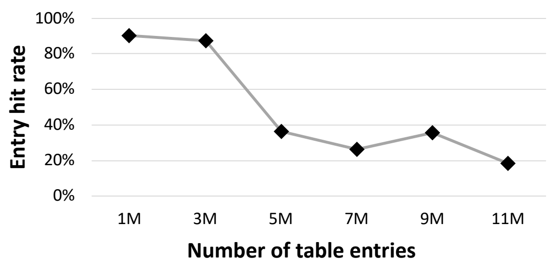

Figure 13. The hit rate of table contents in varied table entries for MobileNet V2.

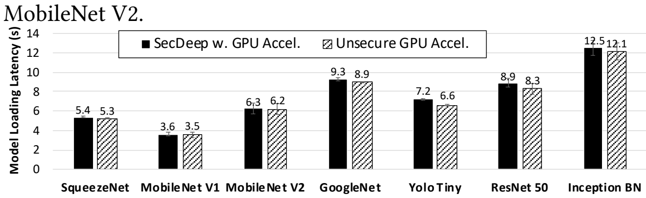

Figure 14. Comparing model loading latencywith and without SecDeep's integrity checker on a GPU-enabled device.

Varying the encryption key length for FPE. Our experiments use AES with CTR mode as the format-preserving encryption method. Although our key length is 128 bits, we run the inference experiment with all three different sizes of AES key lengths, i.e., 128 bits, 192 bits and 256 bits, for MobileNet V1. Our results—summarized in Figure 12—have shown that even if we use the highest AES security standard with a 256-bit key, the overall latency has only increased 13.7% with the lowest AES security standard that uses 128-bit keys. Our SecDeep design has validated that the data confidentiality can be very robust without sacrificing much computation latency.

7.2 Table size options

Although the table is dynamically created, we observed that some of the table contents are hardly hit in the future. Hence, they waste the already-limited secure memory. In this experiment, we test the hit rate for tables with varying upper limits for the table size as shown in Figure 13. The results show that the hit rate significantly reduces after 3 million entries. Hence, for the best performance, 3 million entries should be used to ensure that the memory will not be significantly wasted while providing good performance.

7.3 Model Loading Latency

We evaluate the model loading time using and without using SecDeep. The model loading time refers to loading the model into memory and converting the model to a graph that ARM NN understands for further execution. We implement the MD5 hashing mechanism to

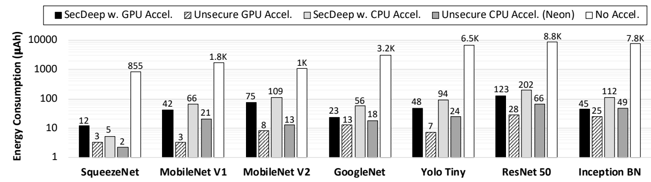

Figure 15. Comparing energy consumption when inferencing with and without SecDeep.

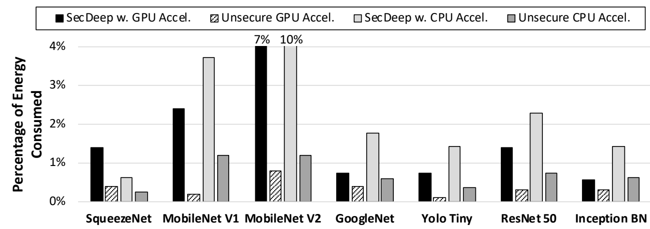

Figure 16. The overhead percentage of the energy consumption with and without SecDeep, CPU acceleration, and GPU acceleration.

check whether the model has been tampered with before loading it to the nonconfidential computing base. Our results in Figure 14 show that the hash checking has introduced marginal overhead when loading the model compared to the unsecure GPU acceleration's model loading time. For the lightweight and mediumweight models, the model loading overhead of SecDeep is less than 4%, while the model loading overhead for the heavyweight models is less than 7.8%.

7.4 Energy Consumption

The last evaluation we characterized is power consumption. Because these are mobile and IoT devices, energy management is a critical concern. To measure the power consumption of the inference process under the aforementioned scenarios, we connect the Monsoon power monitor to our HiKey960 reference board. Figure 15 summarizes the results for each benchmark model. Our experiments show that although GPU is more power-hungry, the overall latency of GPU acceleration is much smaller. Hence, less power consumption is achieved when using GPU acceleration. Although SecDeep consumes slightly more energy than unsecure CPU accelerations in most cases, it still consumes significantly less energy than without using acceleration as illustrated in Figure 16. Our experiments demonstrate that SecDeep can achieve acceptable performance and energy consumption.

8 Discussion and Future Work

In this section, we will discuss the limitations of SecDeep as well as the future research directions.

8.1 Limitations

Since SecDeep does not trust the operating system, any attacks that can maliciously obtain privileged access to the OS (e.g., CVE-2018-8781, CVE-2018-14634,CVE-2019-8635, and CVE-2019-1159) are diminished in terms of the attacking deep learning execution on the edge—where both the data confidentiality and accuracy can be guaranteed from the sensor digitization to the DL results via TEEs. However, one of the assumptions SecDeep makes is that the TEE is always secure and trusted. Possible side-channel vulnerabilities for TEEs may hinder the assumptions of SecDeep. can diminish the protections of SecDeep. Although side-channel attacks are out of this paper's scope, extra protection mechanisms [24, 25] or data validation methods [47] could be implemented to reduce the effects from side-channel attacks.

8.2 Future Work

Training at the Edge. Although SecDeep focuses on deep learning inference at the edge, another direction SecDeep is targeting is secure training on the edge [48]. Given that sensors are increasingly deployed at the edge, models need to be updated frequently to improve accuracy.

Pruning the Model. To increase performance, future works can focus on optimizing deep learning models to adapt to the associated hardware, e.g., such as deep compression [17]. However, although the model size is reduced, such pruning does not solve the biggest constraints of running a secure deep learning inference framework on the edge—limited secure memory.

Autonomous confidential annotation. Ideally, developers would employ annotations for functions as confidential or nonconfidential from the start. However, as was done in this paper, we envision existing frameworks would have to be retroactively annotated. Future work can focus on autonomously or semi-autonomously annotating the code that requires access to plaintext tensor values as an analogous semi-autonomous solution used in Ct-Wasm [42]. Using this approach could significantly help reduce ML application developers' efforts to adapt their design with SecDeep.

9 Related Work

In this section, we will discuss the related works of SecDeep.

Secure machine learning. Previous works have explored securing deep learning frameworks algorithmically when the machine learning models are offloaded to cloud environments. Occlumency [26] leverages Intel SGX to secure deep learning inferencing in cloud environments to preserve data privacy without trusting the cloud service provider. Similarly, Ohrimenko et al. [32] use trusted enclaves to collect sensitive data from distributed clients and run oblivious machine learning training processes. Slalom [40] uses Intel SGX by partially offloading linear layers of DNNs to untrusted CPUs to obtain high performance without sacrificing the data privacy. DeepEnclave [15] uses cloud-assisted SGX for inferencing to overcome the shortage of secure memory on the edge. Privado [39] uses Intel SGX to load different models into an enclave and defend the side-channel attack through access patterns. However, unlike SecDeep, the above works do not provide a secure path to use the available accelerators such as embedded GPUs for the inference. [8] uses privacy-preserving algorithms with assistance of ARM TrustZone to protect the access to the peripherals to achieve ML inferencing data privacy. However, unlike SecDeep, this work does not protect the inferencing data integrity. Although some prior works such as Graviton [41] and Yu et al. [45] design a secure path to GPU. However, their designs fail on mobile and IoT embedded GPU because it shares memories with CPU. Moreover, embedded GPUs are more resource constraint than desktop or cloud GPUs.

TrustZone applications on the edge. ARM TrustZone has been widely adopted in different designs to achieve the security requirement of an app or a system in the research field. Ginseng [46] uses secure registers to hide sensitive variables. However, it's NP computation complexity is problematic for our design that has many sensitive variables. TZ-RKP [7] uses ARM TrustZone to monitor whether the OS is compromised. However, TZ-RKP is unable to protect the integrity of the applications running on the OS. VirtSense [29] split applications into sensitive code and insensitive code. PROTC [30], uses ARM TrustZone to sanitize drone control commands running inside the TEE. These approaches only trust the sensitive code and do not provide protections for the insensitive code. TrustShadow [16] runs an entire secure application inside a TEE, but is not feasible for the large DL models we are considering.

10 Conclusion

We propose the SecDeep DL model computation framework that securely uses available accelerators to provide performant on-device inference on mobile and IoT devices. SecDeep leverages the benefits of TEEs to achieve both high performance and a small TCB size with limited secure memory. We prototype SecDeep on a HiKey 960 development board using ARM TrustZone, and our experiments show that SecDeep can achieve up to 172× model inference acceleration while using only 16MB of secure memory and while minimizing the TCB by 92.4%.

Acknowledgments

The research reported in this paper was sponsored in part by the National Science Foundation (NSF) under award #CNS-1705135, by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, and by the Army Research Laboratory (ARL) under Cooperative Agreement W911NF-17-2-0196. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the ARL, DARPA, NSF, SRC, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

References

[1] J. Amacher and V. Schiavoni. On the performance of arm trustzone. In *Distributed Applications and Interoperable Systems*, 2019.
[2] Android. Android neural networks api. https://developer.android.com/ndk/guides/neuralnetworks.
[3] Apple. Machine learning on ios. https://developer.apple.com/machine-learning/.
[4] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *Proceedings. 1997 IEEE Symposium on Security and Privacy (Cat. No.97CB36097)*, pages 65–71, 1997.
[5] ARM. https://developer.arm.com/ip-products/security-ip/trustzone.
[6] ARM NN (Neural Network). https://github.com/ARM-software/armnn.
[7] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, 2014.
[8] S. P. Bayerl, T. Frassetto, P. Jauernig, K. Riedhammer, A. R. Sadeghi, T. Schneider, E. Stapf, and C. Weinert. Offline model guard: Secure and private ml on mobile devices. In *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 460–465, 2020.
[9] M. Bellare, T. Ristenpart, P. Rogaway, and T. Stegers. Format-preserving encryption. In *Selected Areas in Cryptography*, 2009.
[10] J. V. Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *26th USENIX Security Symposium (USENIX Security 17)*, 2017.
[11] Caffe2 on Smartphone. https://caffe2.ai/docs/mobile-integration.html.
[12] W. Dai, H. Jin, D. Zou, S. Xu, W. Zheng, L. Shi, and L. T. Yang. Tee: A virtual drtm based execution environment for secure cloud-end computing. *Future Generation Computer Systems*, 49:47 – 57, 2015.
[13] DL4J. https://deeplearning4j.org/.

[14] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 449–460. IEEE, 2012.

[15] Z. Gu, H. Huang, J. Zhang, D. Su, A. Lamba, D. Pendarakis, and I. Molloy. Securing input data of deep learning inference systems via partitioned enclave execution. *arXiv preprint arXiv:1807.00969*, 2018.

[16] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger. Trustshadow: Secure execution of unmodified applications with arm trustzone. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. Association for Computing Machinery, 2017.

[17] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[18] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[19] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 784–800, 2018.

[20] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[21] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *CoRR*, abs/1602.07360, 2016.

[22] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[23] D. Jang, H. Lee, M. Kim, D. Kim, D. Kim, and B. B. Kang. Atra: Address translation redirection attack against hardware-based external monitors. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*.

[24] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh. Safespec: Banishing the spectre of a meltdown with leakage-free speculation. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*.

[25] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, Baltimore, MD.

[26] T. Lee, Z. Lin, S. Pushp, C. Li, Y. Liu, Y. Lee, F. Xu, C. Xu, L. Zhang, and J. Song. Occlumency: Privacy-preserving remote deep-learning inference using sgx. In *The 25th Annual International Conference on Mobile Computing and Networking*.

[27] E. Li, Z. Zhou, and X. Chen. Edge intelligence: On-demand deep learning model co-inference with device-edge synergy. In *Proceedings of the 2018 Workshop on Mobile Edge Communications*, pages 31–36, 2018.

[28] H. Li, K. Ota, and M. Dong. Learning iot in edge: Deep learning for the internet of things with edge computing. *IEEE network*, 32(1):96–101, 2018.

[29] R. Liu and M. Srivastava. Virtsense: Virtualize sensing through arm trustzone on internet-of-things. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution*.

[30] R. Liu and M. Srivastava. Protc: Protecting drone's peripherals through arm trustzone. In *Proceedings of the 3rd Workshop on Micro Aerial Vehicle Networks, Systems, and Applications*, DroNet '17, 2017.

[31] S. S. Ogden and T. Guo. $MODI$: Mobile deep inference made efficient by edge computing. In *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*.

[32] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious multi-party machine learning on trusted processors. In *25th USENIX Security Symposium (USENIX Security 16)*.

[33] PyTorch on Android. https://pytorch.org/mobile/android/.

[34] J. Redmon and A. Farhadi. YOLO9000: better, faster, stronger. *CoRR*, abs/1612.08242, 2016.

[35] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov, and L. Chen. Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation. *CoRR*, abs/1801.04381, 2018.

[36] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015.

[37] R. Tartler, A. Kurmus, B. Heinloth, V. Rothberg, A. Ruprecht, D. Dorneanu, R. Kapitza, W. Schröder-Preikschat, and D. Lohmann. Automatic OS kernel TCB reduction by leveraging compile-time configurability. In *Eighth Workshop on Hot Topics in System Dependability (HotDep 12)*, Hollywood, CA, 2012. USENIX Association.

[38] Tensorflow Lite on Android. https://www.tensorflow.org/lite/guide/android.

[39] S. Tople, K. Grover, S. Shinde, R. Bhagwan, and R. Ramjee. Privado: Practical and secure DNN inference. *CoRR*, abs/1810.00602, 2018.

[40] F. Tramer and D. Boneh. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. In *International Conference on Learning Representations*, 2019.

[41] S. Volos, K. Vaswani, and R. Bruno. Graviton: Trusted execution environments on gpus. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 681–696, Carlsbad, CA, Oct. 2018. USENIX Association.

[42] C. Watt, J. Renner, N. Popescu, S. Cauligi, and D. Stefan. Ct-wasm: Type-driven secure cryptography for the web ecosystem. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[43] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia, et al. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–344. IEEE, 2019.

[44] X. Xu, Y. Ding, S. X. Hu, M. Niemier, J. Cong, Y. Hu, and Y. Shi. Scaling for edge inference of deep neural networks. *Nature Electronics*, 1(4):216–222, 2018.

[45] M. Yu, V. D. Gligor, and Z. Zhou. Trusted display on untrusted commodity platforms. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*.

[46] M. H. Yun and L. Zhong. Ginseng: Keeping secrets in registers when you distrust the operating system. In *23rd Network and Distributed Security Symposium (NDSS 2019)*, San Diego, CA, 2019.

[47] Z. Zheng and A. N. Reddy. Towards improving data validity of cyber-physical systems through path redundancy. In *Proceedings of the 3rd ACM Workshop on Cyber-Physical System Security*, CPSS '17, page 91–102, New York, NY, USA, 2017. Association for Computing Machinery.

[48] G. Zhu, D. Liu, Y. Du, C. You, J. Zhang, and K. Huang. Toward an intelligent edge: wireless communication meets machine learning. *IEEE Communications Magazine*, 58(1):19–25, 2020.