This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2022.3207745

LEAP: TrustZone Based Developer-Friendly TEE for Intelligent Mobile Apps

Lizhi Sun, Shuocheng Wang, Hao Wu, Yuhang Gong, Fengyuan Xu, *Member, IEEE,* Yunxin Liu, *Senior Member, IEEE,* Hao Han, *Member, IEEE,* and Sheng Zhong

Abstract—ARM TrustZone is widely deployed on commercial-off-the-shelf mobile devices for secure execution. However, many Apps cannot enjoy this feature because it brings many constraints to App developers. Previous works have been proposed to build a secure execution environment for developers on top of TrustZone. Unfortunately, these works are still not a fully-fledged solution for mobile Apps, especially for the emerging intelligent Apps. To this end, we propose LEAP, which is a lightweight developer-friendly TEE solution for mobile Apps. LEAP enables isolated codes to execute in parallel and access peripheral (e.g., mobile GPUs) with ease, flexibly manages system resources upon different workloads, and offers the auto DevOps tool to help developers prepare the codes running on it. We implement the LEAP prototype on the off-the-shelf ARM platform and conduct extensive experiments on it. The experimental results show that Apps can be adapted to run with LEAP easily and efficiently. Compared to the state-of-the-art work along this research line, LEAP can achieve an average 3.57× speedup in supporting intelligent Apps using mobile GPU acceleration.

Index Terms—Security and Privacy Protection, ARM trustzone, trusted execution environments.

1 Introduction

Secure execution is always a high-priority objective for mobile Apps processing sensitive data. The TrustZone [1] technology, as the de-facto Trusted Execution Environment (TEE) design for mobile devices, has been introduced to fulfill this demand for years since 2004. Although it provides some basic security services (e.g., secure storage) for mobile Apps, we observe that most Apps cannot utilize it for secure execution since it brings many constraints to third-party App developers, which we detail as follows.

First, TrustZone is designed for vendors rather than third-party App developers. App developers must seek cooperation with vendors if they want to put their sensitive code into TrustZone. Besides, adopting TrustZone requires substantial development efforts and TEE knowledge for developers. Vulnerabilities could be otherwise created and lead to the TEE compromising [2]. Moreover, computing resources in TrustZone are extremely limited [3]. As an example, OP-TEE [4], a popular open-source trusted OS used in TrustZone, only supports applications to run with a single thread, and the total memory available for all trusted applications in TrustZone is 16MB. Such restriction significantly impedes the adoption of TrustZone in the App security. Additionally, rapid App development has dramatically reshaped the mobile computing landscape since 2004. Emerging App security demands (e.g., secure mobile GPU accessing) are not recognized or supported in TrustZone.

Research works have been recently carried out to build security solutions for developers on top of TrustZone (shown in Figure 1). These TEE-based solutions are carefully designed to isolate the execution of protected codes in the Normal World (NW) of ARM architecture rather than in the Secure World (SW) to allow developers to deploy their protected codes. TrustICE [5] first attempts to move the APP ENV [1] out of SW. It only allows one App ENV to run and meanwhile freezes the whole Rich OS (ROS), sacrificing the efficiency. PrivateZone [6] lifts the restriction of frozen ROS by introducing another layer of isolation in NW. OSP [7] further enables the parallel running of multiple APP ENVs with a hypervisor. However, introducing a hypervisor would bring system overheads and security risks. Most recently, SANCTUARY [8] leverages the new TrustZone feature to get rid of the hypervisor. However, it can only support limited parallel APP ENVs. More details on related works are provided in Section 8.

Motivation. However, neither the vanilla TrustZone nor the existing NW-side TEE solutions are a full-fledged developer-friendly TEE design for mobile Apps, especially for the emerging intelligent Apps (or deep learning Apps). Nowadays, most developers deploy their deep learning (DL) models, which are often intellectual properties, with their DL Apps on devices to provide real-time intelligent services. According to recent studies [9], [10], it is feasible to steal such valuable on-device models from these DL Apps. As countermeasures, DarkneTZ [3] tries to protect DL models in TrustZone. However, due to the limited memory, it can only protect the last few layers of the DL model in TrustZone to defend against membership inference attacks [11], leav-

- *L. Sun, S. Wang, H. Wu, Y. Gong, F. Xu and S. Zhong are with the National Key Lab for Novel Software Technology, Nanjing University, Nanjing, China, 210023.*
  E-mail: {lzsun,shuocheng.wang,hako.wu,gyh}@smail.nju.edu.cn, {fengyuan.xu,zhongsheng}@nju.edu.cn
- *Y. Liu is with the Institute for AI Industry Research, Tsinghua University, Beijing, China, 100083.*
  E-mail: liuyunxin@air.tsinghua.edu.cn
- *H. Han is with the Nanjing University of Aeronauticsand Astronautics, Nangjing, China, 211106.*
  E-mail: hhan@nuaa.edu.cn
- *Fengyuan Xu is the corresponding author.*

1. We define the APP ENV as the securely isolated execution environment for protected codes in this paper.

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Nanjing University. Downloaded on March 09,2023 at 01:50:32 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2022.3207745

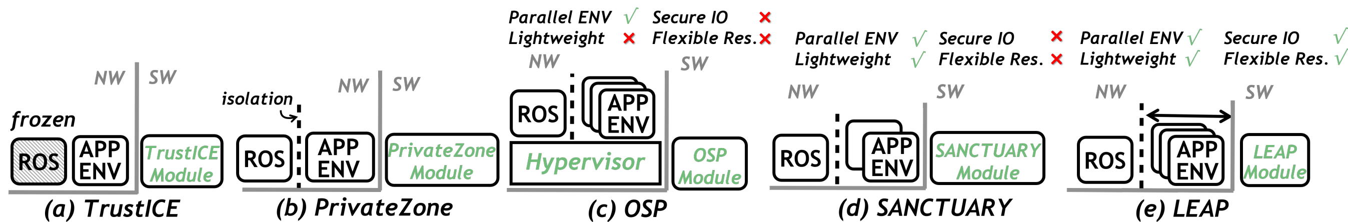


Fig. 1. High-level design comparison between our work LEAP and its related works. ROS is the Rich OS (e.g., Android) in the Normal World of ARM. APP ENV is the securely isolated execution environment for protected codes. Boxed labeled with green texts are key framework components in each TEE-based solution. Key feature differences are annotated in each sub-graph.

ing other layers unprotected. Based on SANCTUARY [8], the latest NW-side TEE solution, OMG [12] can protect the whole model in TEE. However, developers' essential requirements (e.g., GPU acceleration and easy adaptation) are still unconsidered.

The NW-side TEE solutions above, although balancing the security and usability for TrustZone, are not fully developer-friendly for the following reasons. **First**, their secure environments (APP ENVs) lack comprehensive support for the App code execution, specifically the lightweight and parallel isolated environments, secure peripheral access, and flexible resource management. Lightweight is essential for high performance, and parallelism is an important strategy for optimizing performance on the multi-core system, which is widely used by smartphones; more and more codes that require protection contain operations of accessing peripherals, e.g., receiving a cloud-pushed patch inside the APP ENV or securely accessing the mobile GPU for deep learning; adapting resources upon online demands is necessary for parallel isolated environments to reduce resource wasting and meanwhile survive in burst workloads. **Second**, the difficulty of solution adoption is not considered for App developers, especially developers of existing Apps. Usually, it is required to manually modify App codes according to the target TEE-based solution and calculate the resource assignment beforehand. This inconvenience greatly de-motivates App developers to take any action on the solution adoption.

Therefore, we propose **LEAP**, a developer-friendly TEE solution securing critical operations of current and emerging DL Apps. LEAP is lightweight in design and addresses the deficiencies in existing NW-side TEE solutions on the ARM architecture. LEAP can balance the security strength and App usability for six developer-friendly goals below:

(S1) Secure Isolation. The App sandbox (i.e., APP ENV in LEAP) must be isolated with a hardware guarantee.

(S2) Secure Peripherals. The codes inside App sandbox can access peripherals easily and securely, such as the mobile GPU and WiFi, without worrying about sniffing from ROS or codes in other App sandboxes.

(S3) Secure Boot. Each App sandbox can be properly measured for integrity and verified for genuineness before booting.

(U1) Parallel Environment. Multiple lightweight App sandboxes can be isolated and run simultaneously to serve for parallel-running tasks.

(U2) Flexible Resource. The computing resource occupied by App sandboxes can be adjusted on demand in order to prevent resources from being wasted or underutilized.

(U3) Easy Adoption. The auto DevOps [2] tool can be provided for App developers to conveniently adopt LEAP to protect critical executions in their Apps.

Design. LEAP introduces four developer-friendly designs. (1) A lightweight App sandbox isolated by hardware is used to run the sensitive codes, and multiple isolated sandboxes can run in parallel with performance almost as same as the bare-metal case. LEAP proposes a novel policy that is *utilizing virtualization to enforce isolation without virtualizing any resource*. Under this policy, LEAP enables the sandbox to run on bare hardware resources without introducing a hypervisor and enforces the isolation through only managing stage-2 page tables, which avoids the TCB bloating and performance degradation. (2) To enable Apps to securely access peripherals, LEAP introduces a novel exclusive peripheral design that ensures a peripheral already assigned to a sandbox cannot be accessed by any others. LEAP achieves this through the key observation that *ARM adopts Memory-Mapped IO (MMIO)*, which enables us to control IO access through managing stage-2 page tables. Currently, our exclusive peripheral design cannot support all peripherals, and we plan to make it more general in the future. (3) LEAP's resource management allows sandboxes to adjust their computing resources, i.e., CPU cores and memory, according to different workloads. LEAP enables the computing resources to be flexibly adjusted with low overhead. (4) For an existing DL App, a DevOps tool, App Adapter, is introduced to automatically convert it into a LEAP-adapted App through static program analysis. The core function of the App Adapter is to extract the DL modules (containing DL models and inference codes) from the DL Apps and repackage them for execution in the isolated sandbox.

We implement a prototype of LEAP on the off-the-shelf hardware platform, Hikey960, and we show its efficiency and flexibility through extensive experiments. According to our experimental results, DL Apps can be easily adapted to LEAP, and their sensitive codes can be executed efficiently. Compared to the state-of-the-art work [8], LEAP shows excellent benefits in supporting both resource management and peripheral access, e.g., it improves memory utilization by 21.74% and outperforms 3.57× better through secure GPU accessing when severing DL inference tasks.

In summary, our contributions are as follows:

1) We propose a lightweight NW-side TEE, LEAP, which can balance both security and usability

2. The DevOps refers to the splitting, packaging, and deployment of applications.

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Nanjing University. Downloaded on March 09,2023 at 01:50:32 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2022.3207745

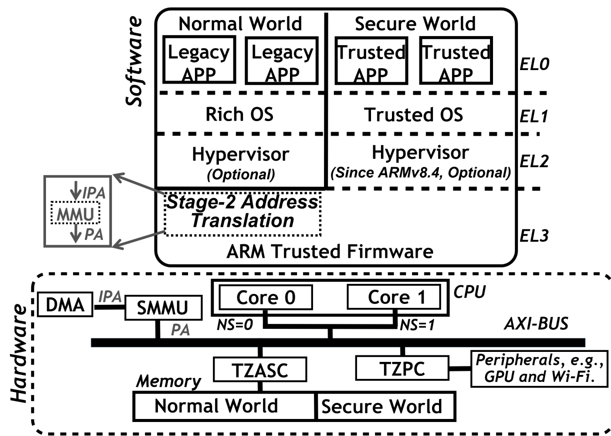


Fig. 2. ARM TrustZone & Stage-2 Address Translation.

specifically for mobile Apps. Compared to existing solutions, LEAP can support lightweight parallel isolated App execution environments featuring flexible resource management.

2) We propose an exclusive mechanism to ensure the secured peripheral access for sensitive application codes. Especially, we enable secure GPU access, which is a key requirement for accelerating secure DL tasks.
3) We implement the LEAP prototype on the off-the-shelf ARM platform without any hardware change. We perform comprehensive analyses and experiments to demonstrate that LEAP is efficient in design, comprehensive in support, and convenient in adoption.

2 Background

2.1 ARM TrustZone

ARM TrustZone [1] is a security extension of ARM processors. As shown in Figure 2, it divides the System-on-Chip (SoC) into two worlds, namely Normal World (NW) and Secure World (SW), to securely manage CPU, memory, and peripherals. A CPU can run in either NW or SW under the control of the *NS-bit* on AXI-Bus. Secure boot [13] is used to ensure the image integrity of the system during the boot procedure. TrustZone Address Space Controller (TZASC), e.g., TZC-400 [14], can isolate the memory by reserving the memory region that can only be accessed in SW. By configuring TrustZone Peripheral Controller (TZPC) [15], peripherals can be isolated, that is, preventing the devices from being accessed from NW. Virtualization in the normal world (EL2) has been introduced since ARMv7, and since ARMv8.4, the TrustZone architecture has evolved with the introduction of virtualization in the secure world (SEL2).

2.2 Stage-2 Address Translation

In ARMv8 architecture, the CPU can execute in four different exception levels (EL0-EL3). Both worlds have the user space (EL0), the kernel space (EL1), and the virtualization extension (EL2). EL3 (monitor mode) is used to respond to world switching. Please note that there is typically no hypervisor running in EL2 on mobile devices due to performance overhead. Therefore, EL2 is usually disabled during the booting procedure.

There are two address translation stages when the virtualization extension is enabled. In the first stage, the virtual machine (VM) translates the virtual address (VA) to an intermediate physical address (IPA) based on its page table. The second stage is called *stage-2 translation*, in which the IPA will be translated to the physical address (PA). The base address of stage-2 page tables is stored in the VTTBR_EL2 register, which can only be accessed in EL2 or a higher exception level. Typically, the hypervisor controls VMs accessing PA through managing stage-2 page tables. Moreover, the second stage cannot be bypassed even if the MMU is turned off by the VM. ARM offers SMMU [16] to translate IPA to PA for the devices which have the Direct Memory Access (DMA) capability. The hypervisor can manage the page tables for SMMU and control the memory access space to prevent the DMA attack.

3 Overview of LEAP

In this section, we first introduce all system components of LEAP, including their roles and functions. We then illustrate how these components interact with each other, a.k.a. the LEAP workflow, throughout the life-cycle of a LEAP sandbox. In the end, we briefly highlight the key designs, which are elaborated with more details in the next section.

3.1 Security Model

Before diving into LEAP design, we first explain our security model. We consider the scenario of protecting the execution of sensitive App codes on the ARM platform with hardware security enforcement. Sensitive codes (i.e., security-critical codes) may want to access peripherals and contain valuable App assets like closed-source DL models.

We assume the Rich OS (ROS) in NW could be malicious or compromised by the adversary. The goal of the adversary is to compromise the execution integrity or access the App assets under protection. We assume the drivers used in LEAP sandbox for peripheral access are benign and bug-free. We also assume some sensitive codes requiring our protection are curious about the execution of other sensitive codes. For example, they may try finding out what sensing data others collect.

We only trust the low-level features of the ARM architecture, including the secure boot, TrustZone, and stage-2 translation. Similar to previous work [17], we do not consider physical attacks like the cold boot [18] and the bus monitoring attacks [19], [20], Deny-of-Service (DoS) attack, and cache side-channel attacks [21], [22], [23], [24].

3.2 System Components

Figure 3 illustrates the high-level design of LEAP. LEAP consists of four components, i.e., LEAP_{ROS}, LEAP_{SOS}, LEAP_{SW}, and LEAP_{ATF}. They are software-based and leverage existing ARM hardware features so that LEAP can be easily deployed on existing mobile devices. **ROS** is the legacy OS running in the NW, e.g., the Android. An App adapting LEAP is called **pAPP**, and its sensitive codes under

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Nanjing University. Downloaded on March 09,2023 at 01:50:32 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2022.3207745

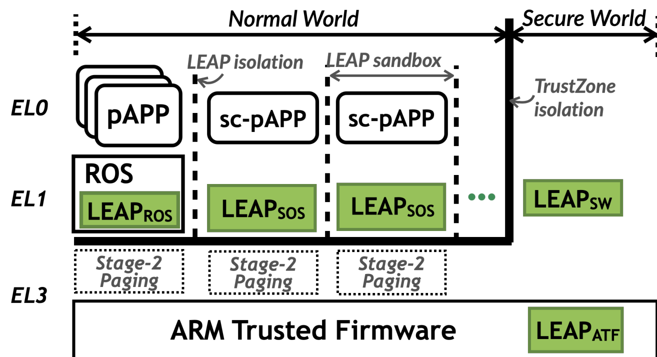


Fig. 3. LEAP System Overview. The green components are LEAP parts.

LEAP protection is called **sc-pAPP**. The **LEAP sandbox** is a sensitive-code execution environment protecting the sc-pAPP and LEAP_{SOS} running inside it. The sc-pAPP is allowed to exclusively access peripherals when needed. Multiple LEAP sandboxes can run in parallel beside ROS with minimal performance influence.

LEAP_{ROS} is a ROS kernel module. It loads images, i.e., sc-pAPP and LEAP_{SOS}, maintains metadata, and pre-allocates resources for LEAP sandbox. A pAPP can create and interact with its sc-pAPP via LEAP_{ROS}. Before switching peripherals or adjusting resources, LEAP_{ROS} prepares the resources and the hardware configuration information to be verified by LEAP_{ATF}.

LEAP_{SOS} is a tiny kernel we tailored and modified from Linux. It is used to provide a minimal runtime inside the LEAP sandbox for a sc-pAPP, named sandbox OS (SOS). LEAP_{SW} interacts with LEAP_{ROS} on behalf of sc-pAPP for resource management. LEAP_{SW} also leverages the rich Linux driver ecosystem to serve various peripheral access needs from sc-pAPP.

LEAP_{SW} is a kernel module in TOS installed by the device vendor. Note it is a part of our Trust Computing Base (TCB). LEAP_{SW} is responsible for key storage and checking the integrity of the LEAP sandbox image before launching it.

LEAP_{ATF} is a patch to the vanilla ARM Trusted Firmware. It also belongs to our TCB. LEAP_{ATF} enforces LEAP sandbox isolation and exclusive peripheral access, manages resources pre-allocated by LEAP_{ROS}, and launches LEAP sandbox.

Except for system components, LEAP also provides an automatic DevOps tool for App developers. This tool, which is called **App Adapter**, can make the DL App adaption of LEAP transparent to its developer, which requires no source code access and extra development efforts. More details are in Section 4.1.

3.3 System Workflow

This part introduces the workflow of LEAP throughout the life-cycle of a sandbox. We describe how to create, initialize, and terminate a LEAP sandbox LEAP_{SOS} and how the LEAP_{SOS} accesses peripherals exclusively and adjusts resources.

Creation. A LEAP-adapted App can be created directly from scratch by a developer or converted from an existing App with the assistance of our DevOps tool. In the converting case, our tool first transforms the App into two parts, the NW part pAPP, and the security-critical part sc-pAPP, with a clean and neat interface between them. Next, it packs the sc-pAPP and LEAP_{SOS} together as an encrypted image and signs it on behalf of the developer. When installing the LEAP-adapted App, this signature is securely stored by LEAP_{SW} for verification purposes in the initialization stage.

Initialization. The LEAP_{SOS} initialization is triggered when the pAPP calls its sc-pAPP counterpart. Once LEAP_{ROS} takes upon the pAPP's request, it pre-allocates resources, i.e., CPU core and memory, for this LEAP_{SOS}. Next, LEAP_{ROS} loads the encrypted packed image, which is prepared in the creation stage, into the allocated memory and notifies LEAP_{ATF} to lock the resources. Then, LEAP_{ATF} asks LEAP_{SW} to verify the integrity. If the verification is passed, LEAP_{SW} will decrypt it as well. LEAP_{ATF} then securely launches it. sc-pAPP will respond to pAPP 's request after booting. Attestation can also be performed during runtime in a similar way like previous works [6], [25].

Peripheral Access. ROS holds all peripheral resources by default. When a sc-pAPP is willing to access one peripheral, LEAP_{SOS} makes a request to LEAP_{ROS}. LEAP_{ROS} checks whether the peripheral is being used, and if it is free, LEAP_{ROS} unloads the device driver (if needed) and informs LEAP_{ATF} to unmap it from ROS and map it to the corresponding LEAP_{SOS} via managing the stage-2 page table. Next, LEAP_{SOS} loads the device driver from ROS, verifies its integrity, and installs it. Then the sc-pAPP in it can use the peripheral. Note that this peripheral cannot be accessed by other LEAP_{SOS} and ROS until it is released from currently-engaged LEAP_{SOS}. To release the peripheral, LEAP_{SOS} unloads the device driver, notifies LEAP_{ATF} to give it back to ROS, and LEAP_{ROS} can bring the peripheral back to ROS.

Resource Adjustment. LEAP_{SOS} is able to request and release resources, i.e., CPU cores and memory, on demand for the sake of efficiency and elasticity. When one LEAP_{SOS} requests more resources, LEAP_{ROS} will prepare the resources and notify LEAP_{ATF} to check whether these resources are secure to be used. Once the check passes, LEAP_{ATF} will assign these resources to the corresponding LEAP_{SOS} and enforce the resource isolation. When releasing resources, LEAP_{SOS} will remove the resources from itself and notify LEAP_{ATF} to return the resources to ROS securely.

Termination. When pAPP no longer needs the sc-pAPP, pAPP sends the shutdown request to the LEAP_{SOS} through LEAP_{ROS} to inform LEAP_{SOS} that it can shut itself down. Then, LEAP_{SOS} informs LEAP_{ROS} of its termination and asks LEAP_{ATF} to shutdown it. LEAP_{ROS} then asks LEAP_{ATF} to release all resources of the terminated LEAP_{SOS}. Released resources are, in the end, returned to ROS.

3.4 Developer-Friendly Designs

In this part, we present several key designs applied in LEAP and the principles behind them at a high level. These designs are driven by the App developer's needs. Additionally, they practice the minimalism design principle and could be an alternative to the current TrustZone hardware evolution.

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Nanjing University. Downloaded on March 09,2023 at 01:50:32 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/TMC.2022.3207745

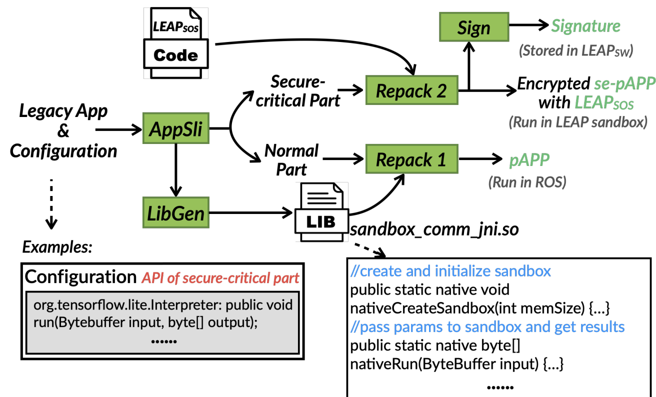


Fig. 4. The processing pipeline of the App adapter.

Automatic App Adapter. The tedious DevOps experience is one of the key reasons why TrustZone and TEE-based solutions are not popular among App developers. Furthermore, many developers may not be familiar with the system programming. Thus, we introduce an auto DevOps tool to transform an App, even without source codes, into a LEAP-ready App.

Isolated Parallel Execution. There may be multiple Apps running in parallel that require protection. At the same time, we want to keep the codes in SW, which is part of our TCB, minimal and fixed without change. Therefore, the attack surface can be reduced. Additionally, we rely on hardware security features to fight against high-privileged threats.

Exclusive Peripheral Management. We design a lightweight mechanism to guarantee that a sc-pAPP can access peripherals exclusively. Moreover, App developers do not have to worry about the availability of peripheral drivers. Currently, our design can only support some devices whose drivers are loadable kernel modules.

Flexible Resources Adjustment. Different computing resources may be required when facing different workloads, so it is hard for a fixed resource assignment to balance task efficiency and resource utilization. The computing resources inside a LEAP sandbox can be flexibly adjusted upon requests from the corresponding pAPP. It is challenging to be achieved given that we get rid of the resource virtualization to gain efficiency inside the App sandbox.

4 Design

The App developer-friendly design realized by LEAP primarily has four techniques, the automatic App adapter used offline for the App preparation, the isolated parallel execution used for the execution of sc-pAPP, the exclusive peripheral management used for secure peripheral access, and flexible resources adjustment used for resource allocation during runtime. Our isolated parallel execution is achieved by only leveraging a small set of existing ARM hardware features - the stage-2 translation, ARM monitor mode, and SEL1 (EL1 in SW) - so that this design can be easily applicable to existing ARM devices.

4.1 Automatic App Adapter

This App adapter is designed to minimize the development efforts when applying for the LEAP protection on an existing App. The automation offers to eliminate the adaption cost concern of non-expert developers. It is currently designed for emerging DL Apps and is intended to demonstrate why the DevOps should be considered, so it does not cover all DevOps demands. The current working scenario of our App Adapter is to extract DL modules (containing DL models and inference codes) from DL Apps. We plan to make it more complete in the future.

Figure 4 illustrates the processing pipeline of the App adapter. Our tool works on the App binary, which has more challenges. To convert a DL App, its developer only has to prepare a configuration file pointing out the entry points of the sensitive codes. To protect the valuable deep learning model with corresponding inference code, developers just list the APIs triggering the inference task in the configuration file for our App adapter. In such file, entry points are listed line by line in the format of <*the class of the function definition: the function prototype*>. Then our App adapter primarily performs two tasks. The first task is to extract the indicated sensitive codes (i.e., sc-pAPP) and the model files out from the targeted normal App, while the second task is to repack sc-pAPP for running in the LEAP sandbox.

More concretely, the *AppSli* module performs call graph analysis and data flow analysis on the App and extracts the security-critical part, i.e., all codes called by entry points and their related model files. *LibGen* generates a dynamic linking library responsible for the communication between the normal part and the security-critical part according to entry points in the configuration file and the sliced codes. Next, the generated library and the App's normal part are repacked as a pAPP to run on ROS. Therefore, all runtime communications between the normal and the security-critical parts will be forwarded through the generated dynamic linking library. As to the security-critical part, the App adapter compiles it into an executable java program, packs the java program with a LEAP_{SOS}, and encrypts it to produce a LEAP sandbox image. The encrypted image will be signed for integrity verification during secure boot. The signature and the decryption key of the encrypted image will be stored in LEAP_{SW} as the whole App is installed on the user device. The encrypted image will be stored on the disk. We provide more technical details about *AppSli* and *LibGen* as follows.

4.1.1 AppSli Module

The *AppSli* module is built upon the java optimization framework, Soot [26]. Soot is suited for performing various static analyses and instruments on Android Apps. We first decompile the App and locate all targeted entry points. We then build call graphs of the App and traverse all reachable codes from these entry points. We also perform the backward data-flow analysis to maintain the dependency of traversed codes. For example, if a developer-defined object type is used in the traversed code, we need to maintain a copy of the class definition in the traversed code. By iteratively performing backward data-flow analyses, all security-critical code can be found and ready for repackaging. Additionally, since all data dependencies are taken into consideration, the paths where model files are located can be analyzed. And all these model files will also be extracted and ready for repackaging.

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Nanjing University. Downloaded on March 09,2023 at 01:50:32 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2022.3207745

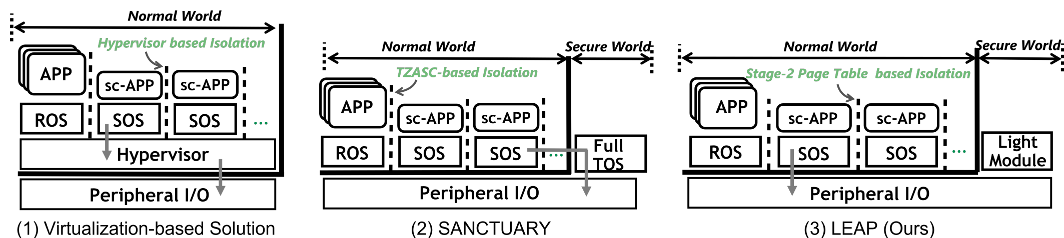


Fig. 5. Current solutions to support parallel isolated executions. The virtualization-based design supports parallel sandboxes through the hypervisor, e.g., OSP [7]. SANCTUARY [8] is a TZASC-based solution, and it can support at most 3 sandboxes in parallel. The last one is the high-level design of our methods. (A detailed design of LEAP can be found in Figure 3.) We detail the difference of LEAP from these works in Section 8.

4.1.2 LibGen Module

The *LibGen* module is used to produce a dynamic linking library, i.e., a communication proxy, which is integrated with the normal part of an App and connects with the corresponding security-critical part. In this library, one component is the code to create and initialize the LEAP sandbox. The sandbox creation functions first notify the LEAP_{ROS} to prepare one CPU core and default 128M memory to launch the sandbox. Then LEAP_{SW} verifies the integrity of the prepared image containing the sc-pAPP before booting. One component is responsible for library and model file dependency. The dependent Android native library in the sensitive part, e.g., OpenCL for GPU access, will be replaced with a corresponding library in Linux. All model files are copied into the sandbox, and the paths to read the model file will be replaced with the paths in the sandbox. The other component is to generate all new entry points for passing parameters between the pAPP and sc-pAPP, with the rely on LEAP_{ROS}. We present an example in Figure 4. For the entry point `<org.tensorflow.lite.Interpreter: public void run(Bytebuffer input, byte[] output);>` provided by the developer, LibGen generates a function `<public static native byte[] nativeRun(ByteBuffer input)>`. This generated function can pass the input data to sc-pAPP through the APIs provided by LEAP_{ROS}. When packing the pAPP, all calls to entry points of the original APP will be replaced with calls to generated ones.

Second, for the DL framework used by the APP, developers need to prepare a corresponding DL framework on Linux since we run Linux in our sandbox.

4.2 Isolated Parallel Execution

It is not intuitive to design an isolated parallel environment, especially given efficiency and security. Figure 5 illustrates some current works that can support parallel isolation, but they have deficiencies in terms of efficiency and security. A virtualization-based design in NW requires a hypervisor, which would bring system overhead to mobile devices when sc-APP is running [7]. SANCTUARY [8] is a TZASC-based (i.e., TZC-400) solution, however, it can only support *at most 3* parallel sandboxes. Because TZC-400 can support at most 8 protected memory regions and each sandbox needs to occupy two protected regions. [3] In contrast, our design is not limited by this and can easily support more parallel sandboxes. Besides, to prevent cache direct attack [8], SANCTUARY proposes to disable the L2 cache, which greatly impacts system performance. We propose a cache protection mechanism to prevent cache direct attack without degrading performance. More details are presented in Section 4.2.3.

3. Secure World also needs to occupy one protected memory region.

4.2.1 Resource Isolation

Figure 5 shows the LEAP's design of parallel isolated execution. LEAP guarantees that the computing resources, i.e., CPU core and memory, used by each sandbox are isolated from ROS and other sandboxes. However, such isolation is not based on virtualizing computing resources. Specifically, LEAP proposes a novel policy that is *utilizing virtualization to enforce isolation without virtualizing any resource*. Under this policy, LEAP lets each sandbox run on its own physical CPU core and memory, and it only enforces the resource isolation through managing stage-2 page tables, which shares a similar idea to NoHype [27].

CPU Isolation. LEAP_{ATF} dynamically removes one physical CPU core from ROS for one LEAP sandbox through Linux CPU hotplug [28] technology. Once the CPU core is removed, ROS will no longer be able to use that core until the core is returned back by the LEAP sandbox. At the same time, each LEAP sandbox can only run on the CPU core assigned to it. In other words, it cannot use other cores that do not belong to it.

Memory Isolation. Since ROS and the LEAP sandbox run on different physical cores, we achieve this goal by managing different stage-2 page tables for them. Specifically, LEAP prepares different sets of stage-2 page tables for the CPU cores that belong to different runtimes. One CPU core and a block of memory will be prepared by LEAP_{ROS} for one LEAP sandbox, and LEAP_{ATF} creates another set of stage-2 page tables and performs an identity mapping, i.e., the virtual address always equals the physical address, for the core. At the same time, LEAP_{ATF} performs an unmapping operation for the stage-2 page tables of ROS to prevent ROS from accessing the memory space.

Parallel Support. LEAP assigns different sandboxes to run on different physical cores and allocates separate memory for them. As a result, every LEAP sandbox can only access its own CPU and memory resources. Since current mobile devices usually equip many cores (e.g., 8 cores) and more than 4GB of memory, it makes LEAP able to support parallel sandboxes easily. For example, for a mobile device with 8 CPU cores, LEAP can support at most 7 sandboxes to run in parallel. However, there is still a challenge we

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Nanjing University. Downloaded on March 09,2023 at 01:50:32 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2022.3207745

need to solve that there are conflicts between multiple OSs since there is no hypervisor. LEAP overcome this challenge through checking the initialization operation for the resource at kernel booting and carefully modifying the kernel codes to change its behavior to avoid conflicts. We provide implementation details in Section 6.1.

Communication Support. Since ROS and every sandbox run on their own CPU and memory resources, we enable ROS and a sandbox to communicate with each other in two ways. First, the request can be sent between them through inter-processor-interrupt (IPI). LEAP_{ROS} and LEAP_{SOS} would know the request type according to the IPI number. Second, the data can be transferred between them through shared memory. LEAP reserves a block of shared memory for every sandbox to communicate with ROS. The shared memory can be accessed by both the ROS and LEAP sandbox. LEAP_{SOS} also can read external files from ROS through shared memory. The shared memory is mapped in the stage-2 page tables of both the ROS and one sandbox, and more details can be found in Section 4.4.1.

4.2.2 Secure Boot

We design an integrity verification mechanism to ensure the secure boot of LEAP sandboxes. Before launching one sandbox, LEAP will verify the integrity of the encrypted runtime image (containing LEAP_{SOS} and sc-pAPP). The signature of the runtime image is produced in the *creation* stage and securely stored by LEAP_{SW}.

Before LEAP_{SW} performs verification, the prepared image to be verified will be first isolated from ROS, which is accomplished by LEAP_{ATF} through managing stage-2 page tables in EL3 directly. LEAP_{SW} is responsible for performing the integrity verification for the image, and it only needs to provide some basic secure services, i.e., key storage, encryption/decryption, and hashing, which keeps a minimal TCB in TrustZone. When verification passes, the LEAP_{ATF} will boot the LEAP sandbox.

To boot one LEAP sandbox, LEAP_{ATF} first starts the core in EL3 and creates another set of stage-2 page tables for it. After all the CPU context is correctly initialized in EL3, LEAP_{ATF} lets the core go back to EL1 instead of EL2 when it returns from EL3 since there is no hypervisor in EL2. Then, LEAP_{SOS} will start to boot in EL1 and run sc-pAPP.

4.2.3 Enhanced Security

As we mentioned before, NW-based memory isolation solutions are vulnerable to the cache direct attack [8]. Hence, an attacker may directly read the memory content from the shared L2 cache. To defend against this attack, SANCTUARY [8] proposes two solutions, i.e., hardware change or disabling the L2 cache, that both have limitations. A hardware change is not available to current hardware and simply disabling the L2 cache would greatly decrease system performance (We show this in Section 6.2).

LEAP defends the direct cache attack by proposing a cache sanitization mechanism. We observe that *the L2 cache is usually physically indexed on ARMv8* [29]. Hence, LEAP can prevent the attacker from successfully translating the virtual address to the physical address through unmapping stage-2 page table entries. However, the stage-2 translation entries may be cached in the translation lookaside buffer (TLB). Therefore, before booting one LEAP sandbox or adjusting a memory region to it, LEAP_{ATF} clears these TLB entries that map to the newly prepared memory space, which has almost no impact on system performance in practical use.

4.3 Exclusive Peripheral Management

4.3.1 Design Challenges

It is non-trivial to design a peripheral management mechanism when considering IO security and usability (e.g., develop effort and efficiency). We show design challenges by proposing two straw-man solutions (Figure 6) and explain why they fail to meet the peripheral management requirements.

Straw-man Solution 1. The first possible design is to redirect all peripheral IO to the Secure World and leverage the hardware-assisted Secure IO. As shown in Figure 6a, all devices are mapped into the Secure World, and all device management modules, e.g., device drivers, are installed into the Trusted OS.

This seemingly simple solution has two serious design flaws. *Usability.* All peripheral drivers needed by App developers should be installed into Secure World in this design. It is at least a hard task, if not an impossible task. Porting or implementing a driver for special Trusted OSs like OP-TEE [4] is difficult and time-consuming even if the corresponding driver for ROSs like Android is open-source. In reality, peripheral drivers are often very complicated and closed-source, rendering the Secure-World driver porting or developing impossible. Additionally, the system programming effort for arbitrary IO redirecting is heavy as well, given that there are so many types of peripheral driver implementations. Therefore, this possible design puts too much burden on the shoulder of application developers. *Security.* Another important reason is that adding so many drivers to Secure World will lead to the TCB explosion.

Straw-man Solution 2. The second possible design is to introduce a driver monitor module, shown in Figure 6b, to ensure there is only one Normal-World driver enabled for a device at a time. When a driver wants to use a device, it should make a request to the driver monitor. After being allowed, it will be enabled and access the requested devices. The driver monitor keeps scanning the normal world to detect if any driver works illicitly.

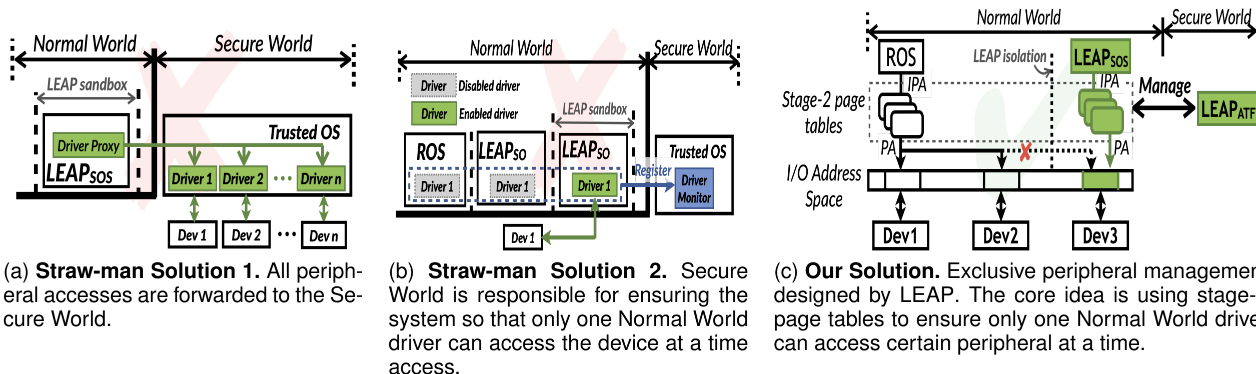
This design also has two problems. *Usability.* It is very costly to scan the kernel memory to detect if any driver works. As reported in DeepMem [30], recognizing a kernel object takes about 13 seconds even in a PC environment, whose computation ability is more powerful than the mobile devices. The overhead of such a design is not acceptable. *Security.* The Normal World OS, e.g., ROS and LEAP_{SOS}, might access the peripheral through directly reading or writing a specific IO address without using a driver. That is, any device access without drivers will bypass the driver monitor and fail this method.

4.3.2 Our Design

Our design follows three principles. (1) Developers should be able to easily access all off-the-shelf peripherals in LEAP sandbox, just as in the Normal World. (2) The peripheral access should be lightweight and efficient. The overhead of

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Nanjing University. Downloaded on March 09,2023 at 01:50:32 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2022.3207745



(a) **Straw-man Solution 1.** All peripheral accesses are forwarded to the Secure World.

(b) **Straw-man Solution 2.** Secure World is responsible for ensuring the system so that only one Normal World driver can access the device at a time access.

(c) **Our Solution.** Exclusive peripheral management designed by LEAP. The core idea is using stage-2 page tables to ensure only one Normal World driver can access certain peripheral at a time.

Fig. 6. Straw-man solutions of peripheral management and our solution. Note only our solution can meet the security and usability requirements in our scenario.

peripheral access from the sandbox should not be greater than that from ROS. (3) Only one Normal-World execution, i.e., ROS or a sandbox, can access a peripheral at a time. Please note this exclusive access design is a trade-off between system security and usability. Designing a scheme that allows multiple sandboxes to access peripherals in parallel would increase system complexity and make it hard to ensure system security.

Figure 6c illustrates our peripheral management mechanism, and our novel design abides all the above three principles. The first principle is accomplished by using a tailored Linux kernel as LEAP_{SOS} so that all the device drivers in the Linux ecosystem can be directly reused. When compiling LEAP_{SOS}, these drivers will be compiled into loadable kernel modules (LKMs), and LEAP_{SOS} can also verify their integrity when installing them. The last two principles are achieved through manipulating the stage-2 tables. The stage-2 page tables are normally used to enforce memory isolation. However, the key observation of our novel design is that ARM adopts Memory-Mapped IO (MMIO), which provides us with the opportunity to control IO access through managing stage-2 page tables.

Recall there may be multiple sandboxes and ROS parallelly run in LEAP on different cores, LEAP_{ATF} sets different stage-2 page tables for each of them. When in use, the LEAP_{SOS} can request LEAP_{ROS} for the device. If the device is free, i.e., no process is using it, LEAP_{ROS} performs device switching procedure as described in 3.3. The LEAP_{ATF} assigns the device to the requester by modifying its stage-2 page tables on the fly. If the requested device has been occupied by execution, the requester has to try later or wait until the device is available before it can gain access permission to it. When a sandbox uses a device, all other sandboxes' and ROS's page table entries of this device will be marked as invalid to ensure exclusive access. However, since every sandbox can directly operate the peripheral, there is a challenge that there are conflicts when devices are switched. We need to carefully modify the kernel codes to avoid conflicts.

The stage-2 page tables that control the peripheral access are stored in a block of physical memory reserved by LEAP_{ATF}. This memory region is never mapped to ROS or LEAP_{SOS} to prevent them from accessing it. The stage-2 page table takes 2MB and 4KB mapping for memory space and IO space, respectively. The page tables of each execution only use less than a 2MB memory region to address and use peripherals. In our prototype system, there are 8 CPU cores. So the reserved memory region is only 16M.

Device Requirements. Currently, our design cannot support all the peripherals on mobile devices, and it requires the peripheral to satisfy the following two requirements. First, the device needs to be relatively independent. Specifically, its device driver can be compiled as a LKM, and the driver is not shared by other devices. [4] Second, ROS does not always need to occupy the device. In other words, the device can have free time (e.g., a few seconds or longer) when it is not used by the ROS so that other sandboxes can have opportunities to use it. Although some devices cannot be supported (e.g., the USB device), many common peripherals (e.g., Bluetooth and WiFi) on mobile devices can meet these requirements.

The two requirements serve the purpose that we need to unload the driver from ROS during device switching. Unloading the driver from ROS has two advantages: First, it can avoid the conflicts that may be caused when the same device is initialized by two drivers. Second, it prevents ROS from trying to access the unmapped device through the device driver. Here, we assume that when the device driver is unloaded, ROS will not try to access the unmapped device. [5] Since unmapping the device is just an enforced access control policy, it cannot guarantee that ROS will not attempt to access the unmapped device.

GPU Access. Unfortunately, the GPU device cannot meet the second aforementioned requirement since ROS may always need to use the GPU to perform GUI rendering. For example, Android uses GPU to perform GUI rendering about every 16ms (60fps). As a result, the GPU device is always busy that the GPU device driver cannot be unloaded. If we do not unload the GPU driver in ROS, the ROS will still try to access the unmapped GPU through the driver, which will cause a stage-2 page fault. As there is no code in EL2 to handle it, the page fault will lead to a GUI crash and system reboot.

To this end, we design a scheme that allows LEAP_{SOS} to access GPU securely without unloading the GPU driver in

4. Some device drivers cannot be compiled as a LKM and some devices may share the same device driver, e.g., a USB device may rely on the USB bus driver, which is shared by many devices. LEAP cannot support these devices yet.

5. A malicious ROS may still try to access the device without the device driver if it will, however, this will lead to a stage-2 page fault.

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Nanjing University. Downloaded on March 09,2023 at 01:50:32 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2022.3207745

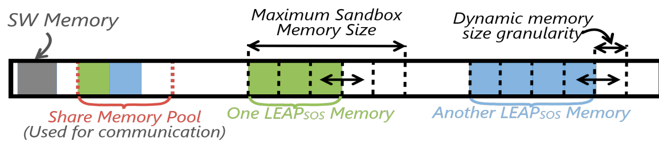


Fig. 7. Sandbox memory layout and dynamic memory scheme design.

ROS. The core idea is to prevent ROS from accessing GPU anymore when GPU is switched out. Specifically, we utilize the feature that GPU can be suspended to stop the ROS operating on GPU temporarily. We let LEAP_{ROS} issue GPU suspending through the GPU driver, and all rendering tasks of ROS will be briefly suspended until the GPU is resumed. When the GPU is suspended, ROS will not call the GPU driver to access GPU anymore. As a result, LEAP_{SOS} can use GPU driver to safely use GPU for computing in its own memory space because mobile GPU uses main memory as computing memory.

In general, for the GPU device, we replace the driver unloading operation with the suspending GPU operation, and other operations remain unchanged for device switching. Suspending the GPU is aimed at preventing ROS from trying to access the unmapped GPU, which will cause a stage-2 page fault. Whether ROS chooses to suspend GPU or not, it cannot access the unmapped GPU since unmapping the GPU is enforced by LEAP_{ATF}. One more thing we need to mention is that suspending the GPU will lead to a brief frozen GUI for the ROS since GPU is temporarily unavailable. The GUI rendering will get resumed when the GPU is returned. We will discuss this problem in detail in Section 7.

4.4 Flexible Resources Adjustment

Dynamic memory adjustment and CPU cores can effectively balance the system workload and improve the system resources' utilization, especially for emerging DL Apps. We detail the resources management in two parts, i.e., the dynamic memory adjustment and dynamic CPU adjustment. There are two challenges that need to be solved. First, the resources need to be adjusted with a low overhead for a low latency requirement. Second, we need to avoid memory fragmentation during memory adjustment.

4.4.1 *Dynamic Memory Adjustment*

LEAP proposes two mechanisms, i.e., *memory pool sharing* and *continuous allocation policy*, to allocate LEAP memory. The memory pool sharing is used to manage the shared memory between ROS and LEAP sandbox. Although ROS and LEAP_{SOS} can send requests to each other through IPI, they need to use shared memory to transfer the data between them. The continuous allocation policy is responsible for the preparation and adjustment of LEAP_{SOS} memory on the fly. Figure 7 illustrates these memory management schemes.

The memory pool sharing maintains all data communication channels, i.e., shared memory, in the same continuous memory region. LEAP_{ROS} continuously allocates a new shared memory region from this pool when booting a new LEAP sandbox. Each sandbox has a fixed and exclusive communication channel. The start address and the size of the shared memory are fixed once the sandbox is started. In order to prevent the LEAP sandbox from accessing others' communication channels, LEAP_{ATF} will not map others' communication channels to this sandbox with the access control guaranteed by the stage-2 page table.

When booting a new sandbox, LEAP_{ROS} will first pre-allocate a memory region with the default size, e.g., 128MB, for it. When one LEAP_{SOS} needs to increase its memory size, it notifies LEAP_{ROS} how much extra memory it needs, and LEAP_{ROS} will try to prepare enough memory for it. LEAP can define a maximum memory size that can be used by every sandbox. The continuous allocation policy ensures that LEAP_{ROS} always allocates continuous physical memory for each LEAP_{SOS} so that the whole memory space of the LEAP_{SOS} is always continuous, no matter how many adjustments are performed. Ensuring the physical continuity of the memory region can reduce system maintenance costs and the complexity of TCB. A trivial method to ensure continuous memory is to reserve a large block of memory for each sandbox. However, the reserved memory cannot be used by ROS, which wastes system resources when there is no sandbox running. Therefore, We apply the Linux Contiguous Memory Allocator [31] (CMA) technology to weakly reserve several continuous memory blocks for LEAP sandboxes.

When dynamically adjusting memory size, LEAP_{ATF} always checks the legality of the dynamic changed memory region, including memory address and memory size, to ensure that it is physically continuous with the memory space of current LEAP_{SOS}, it does not exceed its memory limitation, and it does not overlap with other memory regions.

To determine when to perform memory adjustment, LEAP_{SOS} always monitors its memory usage. When it finds that there is not enough memory, it requests ROS for more memory. And it gives the dynamic memory back to ROS when that memory is freed. In our prototype implementation, we hook the function in the kernel, i.e., security_vm_enough_memory_mm, to detect whether there is insufficient memory.

4.4.2 *Dynamic CPU Adjustment*

A LEAP_{SOS} is assigned with one CPU core by default at startup, and LEAP_{ROS} will set the core to a maximum frequency for LEAP_{SOS} to improve performance. However, pAPP can assign more CPU quota to LEAP_{SOS} so that it can request more cores from ROS on demand. This dynamic CPU adjustment design can achieve a good system workload balance. When adjusting the CPU cores, the LEAP_{SOS} can also request for a big core or little core according to its need to optimize the overall execution and energy consumption.

Basic Design. The basic CPU adjustment design is also based on Linux CPU hotplug [28] technology, and it works as follows. When LEAP_{SOS} wants to adjust its CPU cores, it issues a request to LEAP_{ROS}. LEAP_{ROS} checks whether LEAP_{SOS} is allowed to use more cores and if there is any available core. If any core is available, LEAP_{ROS} notifies LEAP_{ATF} to remove the core from ROS. LEAP_{ATF} clears the core's cache to prevent data leakage and securely shutdowns the core. In the end, LEAP_{SOS} requests LEAP_{ATF}

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Nanjing University. Downloaded on March 09,2023 at 01:50:32 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2022.3207745

for the core through CPU hotplug interface, and LEAP_{ATF} initializes the core with the correct context and boots the core for that LEAP_{SOS}. The LEAP_{SOS} will give the surplus cores to ROS through a similar procedure if it finds that the CPU is not busy anymore. LEAP_{SOS} always holds at least one core, i.e., the booting core, since it will never be adjusted.

Optimization. The CPU adjustment design described above requires one physical shutdown and one booting process every time the core is adjusted, which may cause unnecessary system overhead. Therefore, we design an optimization method in LEAP_{ATF}, which is more lightweight. Every time before adjusting one core, LEAP_{ROS} first informs LEAP_{ATF} that it will perform an adjustment, then it uses the CPU hotplug interface to ask LEAP_{ATF} to shut down the core as usual. LEAP_{ATF} will perform the cache cleaning operation for that core. However, it will not physically shut down the core but let the core enter a busy waiting state to wait for LEAP_{SOS} requesting for it. When LEAP_{SOS} requests for the core, LEAP_{ATF} can quickly initialize the context for the core and adjust it to LEAP_{SOS}. We show the benefits of this optimization in Section 6.3.

To determine when to adjust the CPU core, the LEAP_{SOS} always monitors its CPU usage. If LEAP_{SOS} finds that its CPU is busy for a while and its core numbers are within its CPU quota, it requests ROS for one more core. On the contrary, when LEAP_{SOS} finds that the surplus core is free for a while, it releases the core back to ROS. In our implementation, there is a kernel thread in LEAP_{SOS} that continuously monitors the CPU usage of a sandbox. If it finds that the average usage of the CPU is above 99% for 2 seconds, it performs a CPU adjustment operation. When LEAP_{SOS} finds that the average usage of the surplus core is below 40% for 5 seconds, it releases the surplus core to ROS.

5 SECURITY ANALYSIS

In this section, we discuss how LEAP defends against possible attacks under our security model (See Section 3.1). Since LEAP provides hardware-assisted isolation among ROS and different sandboxes, the malicious codes, whether in the ROS or a LEAP sandbox, cannot access data or compromise executions in another LEAP sandbox.

Malicious LEAP_{ROS} Manipulation. A compromised ROS can manipulate the LEAP_{ROS} installed by LEAP. The LEAP_{ROS} is responsible for preparing the sandbox image and pre-allocating the resources. So malicious manipulations lie in the sandbox creation and resources management. When creating a new sandbox, the compromised LEAP_{ROS} may prepare malicious LEAP_{SOS} and sc-pAPP images to compromise secure services. LEAP copes with this attack with a Secure Boot mechanism (See Section 4), which can ensure the LEAP sandbox images' integrity before launching the image. The malicious ROS can also misconfigure resources during resource adjustment. To be specific, when a sandbox increases its memory, ROS can maliciously prepare a memory region for the requester that has already been used by another sandbox. LEAP solves this kind of attack by checking the configurations' legitimacy through LEAP_{ATF} (See Section 4.4). Similarly, LEAP_{ATF} also ensures that a compromised ROS cannot allocate a CPU core that has already been occupied by a sandbox to another one through verification when creating sandboxes or adjusting CPU cores.

Peripheral IO Eavesdropping. The compromised ROS cannot successfully access the IO addresses of a peripheral occupied by a LEAP sandbox. It is because these addresses are blocked in the stage-2 address translation, which is controlled by the LEAP_{ATF}. At the same time, the IO address translation for this device is also blocked for other LEAP sandboxes. Therefore, one LEAP sandbox cannot successfully perform IO Eavesdropping to other sandboxes, either. For some devices capable of DMA, LEAP, except using the same method to block peripheral DMA, replies on the ARM's SMMU [16] to prevent bypassing the main memory access control. Thus, a compromised ROS or a malicious LEAP sandbox cannot eavesdrop on the data in a peripheral occupied by a LEAP sandbox.

Cache Direct Attack. As discussed in SANCTUARY [8], a compromised ROS may access the memory region to be allocated to the sandbox to cache it in the L2 cache. After the memory adjustment, the compromised ROS tries to access the sandbox's memory space through the L2 cache. LEAP proposes a cache sanitization technique (See Section 4.2) to defend against this kind of attack by clearing the CPU cores' TLB entries related to the newly-allocated memory. For different LEAP sandboxes, the memory space that belongs to one sandbox is never mapped to other sandboxes. So one sandbox cannot directly access the address space of another sandbox, nor can it read the memory space of another sandbox through the cache because it can never successfully translate the address space that belongs to others to a valid physical address which is required by L2 cache indexing.

6 EVALUATION

In this section, we describe the experimental setup, followed by a comprehensive evaluation of LEAP by answering the following three questions:

1) How does our isolation design perform when compared with other isolation methods?
2) How does the design of the flexible resource help the sandbox balance the workload?
3) How does our exclusive peripheral design perform when accessing peripherals?

Last, the case study of a real-world GPU-accelerated machine learning application demonstrates how easily and efficiently an application can run with LEAP.

6.1 Experimental Setup

Hardware. We implemented a prototype of LEAP on Hikey960, a widely-used development board with the same SoC as many COTS smartphones (e.g., Huawei P10). The board equips with eight cores (4 Cortex-A53 + 4 Cortex-A73) with big.LITTLE architecture, a 4GB physical memory of which 3.5 to 4GB address space is used for peripheral I/O address space. For peripherals, a Mali-G71 GPU, a WiFi module, and a Bluetooth module are available.

Software. Android 9.0.0_r31 (kernel version 4.14) and a popular open-source Trusted OS, OP-TEE (v3.4.0) [4], were

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Nanjing University. Downloaded on March 09,2023 at 01:50:32 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2022.3207745

TABLE 1
Booting time, memory consumption, and shutdown time comparison.

Measurement	LEAP	SANCTUARY	KVM/ARM
Booting (ms)	532	503	760
Mem. consumption (MB)	128	128	135
Shutdown (ms)	629	625	680

TABLE 2
Data copy time of different data sizes through shared memory.

Data Size	Time (ms)	Data Size	Time (ms)
64KB	16.58	4MB	39.46
256KB	17.69	16MB	110.65
1024KB	22.46	64MB	323.42

chosen as the LEAP's ROS and TOS, respectively. We used the standard ARM Trusted Firmware patched with $LEAP_{ATF}$ in EL3. The whole LEAP system has 4,689 lines of code (LOC), including $LEAP_{ATF}$ (539 LOC), $LEAP_{SW}$ (651 LOC), $LEAP_{ROS}$ (1,327 LOC), $LEAP_{SOS}$ (972 LOC), and DevOps (1,200). $LEAP_{SOS}$, the LEAP sandbox's OS, utilized a customized Linux kernel (v3.13) whose size is only about 9MB. We implemented a dynamic memory adjustment mechanism for it according to the Linux memory hotplug [32]. *Conflicts Elimination.* In order to avoid conflicts between ROS and LEAP sandbox's OS, we put extra engineering effort. First, the initialization code of GIC was removed since there is no need for a sandbox to initialize GIC, which has already been done by the ROS. Second, the code for setting system clocks was modified to prevent the sandbox from resetting the system clocks when switching devices. Last, the in-memory file system ramfs was used in $LEAP_{SOS}$, which can also reduce the booting time. The sandbox can read an external file from ROS through $LEAP_{SOS}$. Please note that these modifications only need to be done once and they are not OS version specific. Therefore, our modification efforts can be reused.

Methodology. We compared our LEAP to a virtualization-based solution, KVM/ARM, and a SANCTUARY [8] prototype. KVM/ARM is a virtualization method that is available on our ROS, [6] and SANCTUARY [8] is the current state-of-the-art work of TEE in Normal World. The software environment, i.e., ROS, TOS, and ATF, of the three prototypes are exactly the same. We cross-compiled qemu-kvm for ROS to boot virtual machine (VM). Since SANCTUARY is not open-sourced, we reproduced it following the paper [8] with one modification for a fair comparison. SANCTUARY uses a micro-kernel OS in its sandbox, while the reproduced SANCTUARY prototype uses the same sandbox OS as our LEAP. Note that this modification will not hurt its design idea. Unless specified, we enabled the L2 cache for SANCTUARY. [7] All three prototypes used the same hardware settings. Since LEAP sets a fixed CPU frequency for sc-pAPP, we set the CPU frequency to a fixed maximum frequency for all prototypes, and we let the CPU cool down between every experiment.

6.2 Sandbox Execution Performance

We evaluated the execution performance of the sandbox at both kernel and application levels. As a result, we conclude that our design has a small initialization overhead and a high communication efficiency with ROS. It also proves that introducing stage-2 translation only brings a negligible overhead (maximum 2%), and our lightweight sandbox enables a high application performance when compared with other solutions.

6.2.1 Kernel Performance

We first evaluated the booting time, memory consumption, shutdown time as well as communication performance with ROS. Moreover, we run LMbench [33] in the sandbox to measure the system call performance.

The results of booting time, memory consumption, and shutdown time of three prototypes are shown in Table 1. It shows that all prototypes can be booted within one second. This is because the kernel is tailored, and the in-memory file system, ramfs, is used. We notice that LEAP is slightly slower than SANCTUARY since LEAP needs to set up stage-2 page tables during the booting procedure. However, LEAP still performs better than KVM/ARM, which indicates that LEAP is lighter than virtualization. For memory consumption, since we need to boot a Linux kernel, all three prototypes are configured to start with 128MB of memory. From Table 1, we can see that LEAP and SANCTUARY consumed 128MB memory which is not surprising since they allocated all memory before booting. Interestingly, KVM/ARM consumed 135MB of memory which is even larger than the required memory, and we think this is because KVM also needs some extra memory to manage the VM.

We also measured the performance of the two types of communication, i.e., IPI and shared memory, between the sandbox and the ROS. We only do this measurement for LEAP since our SANCTUARY prototype has the same communication design with LEAP because how its shared memory is implemented is not explained in its paper [8], and our KVM/ARM prototype does not support shared memory between host and guest. First, we measured the time cost, which starts at the ROS (the sandbox) making a request and ends at the sandbox (the ROS) receiving the request through IPI. As a result, it takes 23.89us for the ROS to communicate with the sandbox and 53.12us for the sandbox to communicate with the ROS, respectively. For shared memory performance, we report the data copy cost between the ROS and the sandbox with different data sizes. The results are shown in Table 2. It shows that data can be quickly transferred between the ROS and a sandbox which enables a high data communication efficiency.

Last, the system call performance in the sandbox is shown in Figure 8, and the results are normalized. In this experiment, we also measured the performance of SANCTUARY with its L2 cache disabled ("SANC W/O L2" in Figure 8). As is shown, compared to LEAP, the performance overhead on system call of the KVM/ARM is about 28% on average. The overhead is even up to 133% for the *exec* system call. It indicates that LEAP has a better performance than virtualization since it does not virtualize

6. We cannot compare with OSP [7] because it is not open-sourced and its design is based on ARMv7 architecture. Since it is also related to KVM/ARM, the results can also reflect its performance to some extent.

7. Please note that enabled the L2 cache for SANCTUARY would improve its performance, however, it may suffer from cache direct attack. More details are in Section 4.2.3.

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Nanjing University. Downloaded on March 09,2023 at 01:50:32 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/TMC.2022.3207745

TABLE 3
DL models used in our experiments to benchmark runtime performance and their specifications.

Model Name	GFLOPs	Params (M)	Model Size (MB)
MobileNetv2 [34]	0.32	3.5	14
GoogleNet [35]	1.51	13	27
AlexNet [36]	0.72	61.1	233
ResNet18 [37]	1.82	11.69	45
ResNet50 [37]	4.12	25.56	98
ResNet101 [37]	7.85	44.55	170
ResNet152 [37]	11.58	60.19	230
Inceptionv4 [38]	12.31	42.68	163

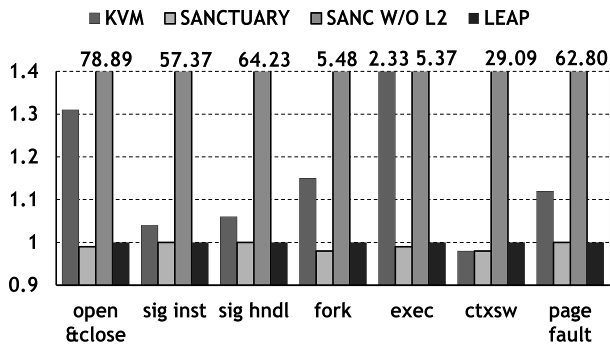


Fig. 8. LMBench benchmark results.

any resource. Compared with SANCTUARY, LEAP has a similar performance to SANCTUARY when its L2 cache is enabled. However, SANCTUARY is much slower (up to 78.89×) than LEAP when the L2 cache is disabled. *Stage-2 translation overhead.* Compare LEAP with SANCTUARY when its L2 cache is enabled, we can also see the overhead brought by the stage-2 translation since SANCTUARY does not introduce stage-2 translation. Figure 8 indicates that the overhead introduced by the stage-2 translation is negligible. The maximum overhead brought by stage-2 translation is about 2%.

6.2.2 *Application Performance*

For the application performance, we measured the performance of two types of tasks, i.e., encryption and DL model inference. For the encryption, we measured the encryption performance of different data sizes. For model inference, we measured the inference time of 8 popular convolution neural network (CNN) models (See Table 3) to perform a classification task. [8]

The encryption spec was set to AES-256-CBC. We use MNN [39] as our DL framework for model inference, and these models are available from MNN or converted from Caffe Model-Zoo [40]. The benchmarks on the three platforms were all performed on the same physical big core. Since qemu-kvm provides virtual cores to a VM, we set the qemu-kvm to provide one virtual core to a VM and bind it to a physical big core for fairness. We also measured the performance of SANCTUARY with its L2 cache disabled.

The encryption performance with different data size and the inference time of 8 DL models on three prototypes are

8. We use CNN models to perform benchmarks due to their popularity on mobile devices.

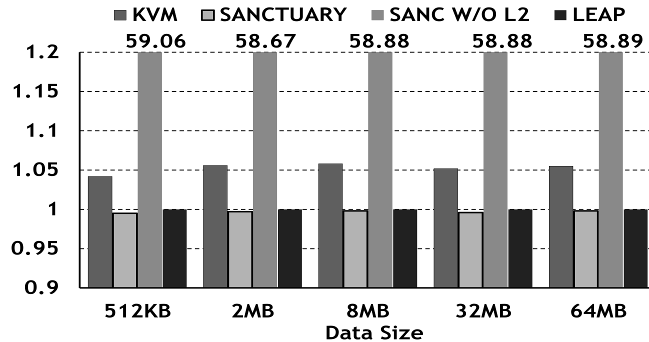


Fig. 9. Encryption performance comparison with different data sizes.

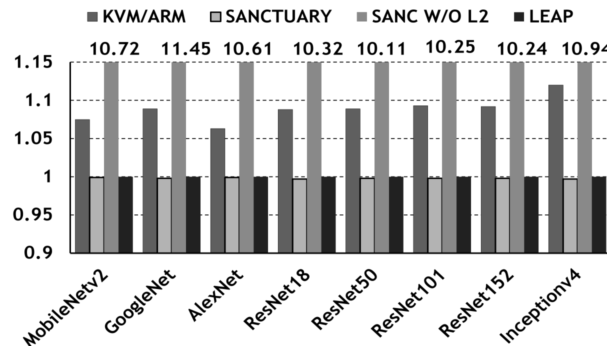


Fig. 10. Inference time comparison with different DL models.

shown in Figure 9 and Figure 10, respectively. LEAP performs about 5% and 10% on average better than KVM/ARM in encryption and DL inference task. LEAP has a similar performance to SANCTUARY when its L2 cache is enabled. It also shows that SANCTUARY performs worse than LEAP when its L2 cache is disabled, e.g., 10.58× slower on average in DL inference, which indicates that SANCTUARY failed to have a good balance between security and efficiency.

Again, it proves that introducing stage-2 translation brings negligible overhead. The average overhead is only about 1%. We think that such a small overhead benefits from two aspects. First, LEAP does not virtualize the CPU core, which avoids the overhead introduced by virtual core context switching. Second, LEAP adopts big page mapping (i.e., 2MB mapping) for the stage-2 page table. Compared with 4KB mapping, big page mapping could bring better performance because it will bring fewer TLB entry conflicts.

6.3 Flexible Resources Performance

To demonstrate the benefits of flexible resources, we profiled the resource adjustment cost and evaluated its performance with different workloads. We only compare LEAP with SANCTUARY below since both our KVM/ARM and SANCTUARY prototype do not support flexible resources. The results conclude that our flexible resource design can be performed in an efficient manner, and it enables a high application performance as well as high resource utilization.

6.3.1 *Resource Adjustment Cost*

We profiled the resource adjustment cost. For CPU adjustment, we measured the adjustment time for both big

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Nanjing University. Downloaded on March 09,2023 at 01:50:32 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2022.3207745

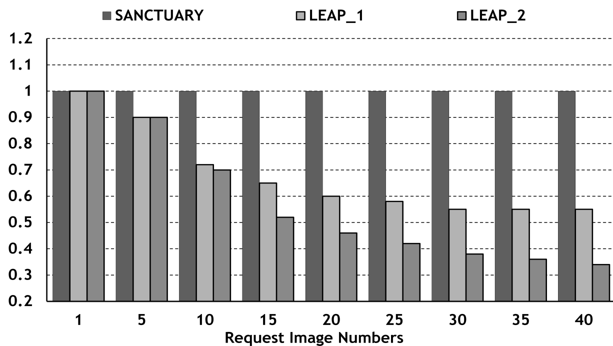


Fig. 11. Task completion time comparison with different image numbers and CPU quotas.

and little cores. For memory adjustment, we measured the adjustment time for a block size of 16MB memory.

Table 4 shows the experimental results. The "w/ opt" and "w/o opt" represents enabling CPU adjustment optimization or not. It shows that all adjustment operations can be performed within 80ms. Moreover, it proves that our CPU adjustment optimization can reduce the system overhead. It performs about 1.48× to 2.51× better with our optimization because it avoids physically turning off the CPU core during adjustment. The ability to adjust resources at such a small cost demonstrates the flexibility and efficiency of LEAP in terms of flexible resources adjustment.

6.3.2 Flexible Resource Benefits

To show the benefits of flexible resource adjustment under different workloads, as an example, we built two test applications and measured their performance under different workloads.

First, a DL application used ResNet50 to perform the classification task. We enabled dynamic CPU adjustment for LEAP, and we set LEAP_{SOS} to increase its core when it detected that the CPU was busy for more than 2 seconds. We changed the total number of images for classification and recorded the total inference time with different CPU quotas.

Figure 11 shows how dynamic CPU adjustment can help applications balance different workloads. LEAP _1 or LEAP _2 represents that LEAP_{SOS} is allowed to increase 1 or 2 cores dynamically, and the inference time for different image numbers is normalized to SANCTUARY. When there is only one image needed for classification, dynamic CPU adjustment will not be triggered, so LEAP and SANCTUARY have the same performance. However, as the number of images increases to 5, LEAP starts to dynamically increase one CPU core to speedup inference, resulting in 1.1× to 1.8× speed up as the number of images increases. Furthermore, when 2 CPU quotas are allowed, LEAP starts to request for the second dynamic core when there are 10 images to be classified, and it provides up to 2.9× acceleration compared to SANCTUARY when there are 40 images.

Second, we tested a ciphertext query App that accepts the key provided by a user as a query keyword, performs the query in the encrypted file with key-value data, and returns the results to the user. To speed up the query procedure, the query App caches the decrypted data in the memory. The encryption method we chose is the same as

TABLE 4
Flexible resource adjustment cost profiling results.

Operation	Resource Type		Time (ms)
Increase	little core	w/o opt	137
		w/ opt	55
	big core	w/o opt	199
		w/ opt	79
	memory		54
Decrease	little core	w/o opt	72
		w/ opt	42
	big core	w/o opt	92
		w/ opt	62
	memory		56

TABLE 5
The execution time and resource utilization rate with different memory allocation strategies.

Memory Size (MB)	Time(s)	Resource utilization
30	35.50	98.98%
50	34.43	96.37%
60	27.76	93.43%
80	22.58	85.52%
100	17.63	70.39%

the secure storage encryption method provided by OP-TEE, which uses AES-128-CBC to encrypt files, and the size of each encrypted block is 256 bytes.

We generated 10 encrypted files containing different key-value pairs of different sizes, ranging from 10MB to 100MB, and we also randomly generated 10 query sequences for each file. We measured the time to complete 10 queries for each file and recorded the total time cost to complete all queries for 10 files. In both SANCTUARY and LEAP, we set the cached memory size to 10MB. However, the query App on LEAP can dynamically adjust its memory size, which is set at a 16MB granularity to handle files with different sizes.

It took about 19.24 seconds to finish all queries for LEAP and the time for SANCTUARY was 61.64 seconds. LEAP performs about 3.20 × faster than SANCTUARY. Although it is possible to make SANCTUARY allocate a large memory size in advance to improve efficiency, however, this will greatly waste resources because most of these memories are not used most of the time. We measured the SANCTUARY performance with different memory allocation sizes and compared its efficiency and resource utilization with LEAP. The result is shown in Table 5.

As Table 5 shows, when SANCTUARY increases the pre-allocated memory size, it indeed improves efficiency. However, resource utilization also decreases. The resource utilization rate of LEAP is 92.13%. Compared with SANCTUARY, LEAP is faster than SANCTUARY by 1.44 × in the case of a similar resource utilization rate (93.43%). When SANCTUARY and LEAP have similar performance, SANCTUARY's resource utilization rate is lower than LEAP by 21.74%. More importantly, when security-critical Apps need to handle a variety of workloads, it is difficult to choose an appropriate resource allocation strategy in advance to balance resource usage and application performance well.

6.4 Peripheral Access Performance

We only evaluated the WiFi, Bluetooth, and GPU devices on our prototype since their device drivers are loadable.

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Nanjing University. Downloaded on March 09,2023 at 01:50:32 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2022.3207745

TABLE 6
Peripheral mapping/unmapping time profiling results.

Operation	Device	ROS Time (ms)	Sandbox Time (ms)
Mapping	GPU	55	121
	WiFi	193	188
	Bluetooth	117	125
Unmapping	GPU	35	23
	WiFi	43	37
	Bluetooth	33	29

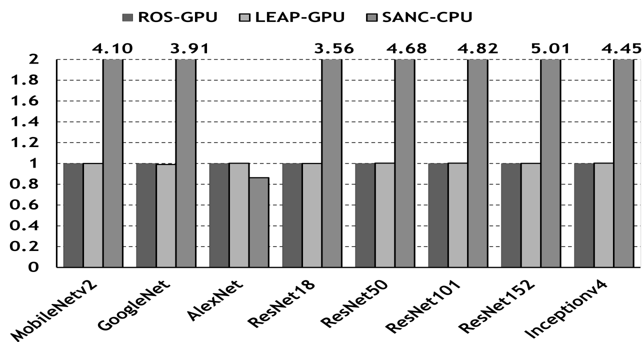


Fig. 12. GPU (CPU) performance comparison with different DL models.

We used these devices to evaluate the LEAP's performance in accessing peripherals. These devices cannot be securely accessed in our KVM/ARM and SANCTUARY prototype since our qemu-kvm does not support virtualizing these devices and SANCTUARY relies on TrustZone to perform secure IO. [9] Hence, we first evaluate our peripheral mapping/unmapping overhead, then we compare the peripheral access performance of LEAP with native ROS, and the results show that our peripheral access introduces negligible overhead.

6.4.1 Peipheral Mapping/Unmapping Overhead

To know the peripheral switching overhead between the ROS and a sandbox, we measured the mapping and unmapping overhead of different peripherals, i.e., GPU, WiFi, and Bluetooth module, in the ROS and a sandbox, respectively. The unmapping procedure starts at the kernel unloading the driver (or suspending for GPU), and it ends at the LEAP_{ATF} finishing the unmapping device address space in the stage-2 page table. The mapping procedure starts at the LEAP_{ATF} performing the mapping device address space, and it ends at the kernel loading the driver (or resuming for GPU).

The experimental results are shown in table 6. The results show that the mapping or unmapping operation in the ROS and a sandbox can be performed within 200ms, which indicates that these devices are able to be switched between the ROS and a sandbox with little overhead.

6.4.2 Peipheral Performance

GPU Performance. We measured the inference time of 8 different DL models on GPU for both LEAP and ROS to evaluate the GPU performance. The benchmark results are presented in Figure 12. It also includes the inference time of SANCTUARY running these models on big cores

9. Although they are possible to be configured to TrustZone through TZPC, however, OP-TEE lacks these device drivers.

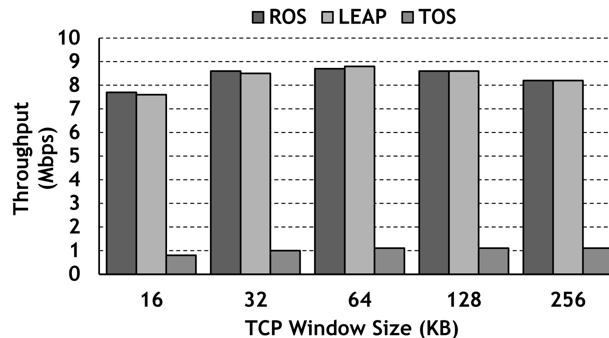


Fig. 13. iPerf networking benchmark results.

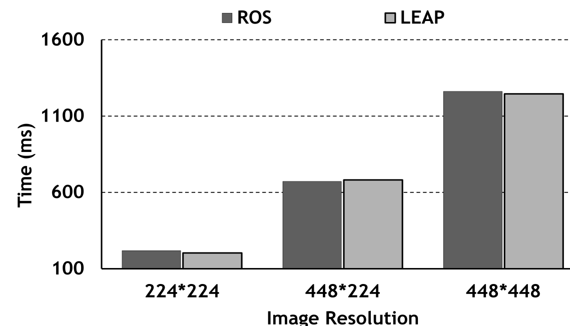


Fig. 14. Bluetooth performance comparison with different resolution.

for comparison. It can be found that the performance of accessing the GPU from LEAP is comparable to that of accessing the GPU from ROS. That is, LEAP does not incur performance overhead to the GPU access. More importantly, this experiment shows the significant advantages brought by peripheral access. Compared with SANCTUARY, LEAP performs about 3.91× to 5.01× better than SANCTUARY through accessing GPU securely.

Network Performance. We run iPerf [41] to benchmark the network throughput for LEAP and ROS when they utilize the WiFi module. At the same time, we additionally run iPerfTZ [42] to measure the network throughput for OP-TEE, and it can be referenced as the network performance of SANCTUARY since SANCTUARY relies on TOS to perform IO. iPerfTZ [42] is an open-source tool that measures the OP-TEE network throughput by forwarding network data to a client process running in NW. Note that it is not a secure way, but we have to measure OP-TEE in this way because it lacks a WiFi driver. The benchmarks were run in the same settings. We set the socket buffer size to 128KB and tested the network throughput with different TCP windows sizes. Results are presented in Figure 13. The performance of accessing the network in LEAP is comparable to that of accessing the network in ROS. However, for OP-TEE, the network throughput of this naive solution is only about 12.5% that of LEAP. The poor network throughput is due to the frequency context switch between ROS and TrustZone to transfer the network data.

Bluetooth Performance. It is a common scenario that IoT devices offload their computing tasks to mobile devices through Bluetooth. To measure the Bluetooth performance, we set the ROS/sandbox to use Bluetooth and established

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Nanjing University. Downloaded on March 09,2023 at 01:50:32 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2022.3207745

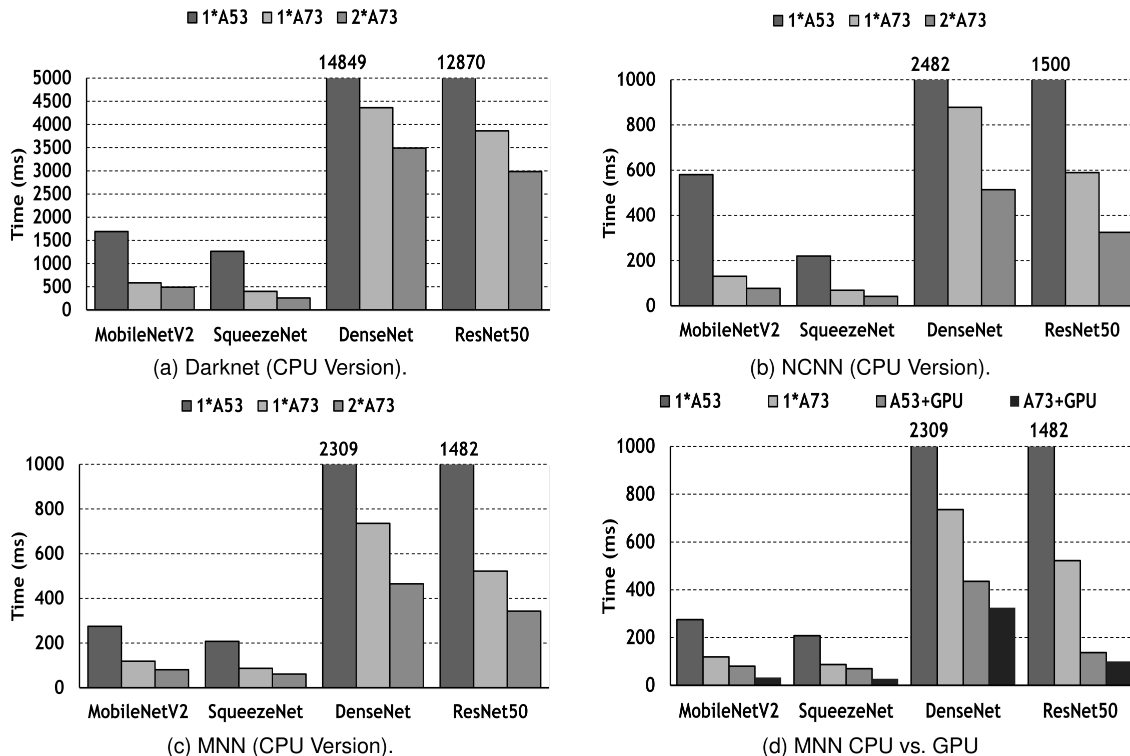


Fig. 15. Inferring time with different settings.

a connection with it using another Hikey960 board. The two boards established a Bluetooth connection with each other through the L2CAP protocol. We transferred images of different resolutions between two boards and measured the time required for the transfer. The experimental results are shown in Figure 14. It also shows that LEAP has a comparable Bluetooth performance with ROS.

6.5 Case Study

We perform case studies on how a representative App, a DL inference using the mobile GPU acceleration, adopts the LEAP for secure model execution. By applying LEAP, the model of the demo App can easily avoid being stolen and defend against other security attacks. We have selected three examples. The first one is an MNN-based [39] intelligent App that is deployed on LEAP platform through our LEAP adapter automatically; the other two intelligent Apps are developed from scratch with NCNN [43] and DarkNet [44] framework. Below we will first study the results of automatic adaptation and then evaluate the system performance on these three examples.

Deploy with LEAP Adaptor. Please recall that our LEAP Adaptor works on the existing DL Apps, and all operations are done on the intermediate code. This demo App (210,000 lines of intermediate code) is a DL inference of image classification with the Mali mobile GPU acceleration, representing a popular emerging App category. The sensitive part to protect contains the DL model, and its inference framework is MNN. We adopt this intelligent App to our LEAP through the LEAP Adaptor, described in Section 4.1. Our LEAP Adaptor takes 11s to complete the adaptation, occupies 1G of memory, and uses two CPU cores. The Adaptor adds only 80 lines of code to the original App. The generated sc-pAPP has a total of 856 lines of code.

Develop from scratch. We also adapt two example Apps manually to show how to develop a LEAP-enabled App from scratch. The split is completed in the following steps. First, we add an integrated LEAP_{ROS} API lib into the App project. Second, we add the function of booting the LEAP sandbox in JNI code, and the code will be called when the App starts. Third, we modify part of the JNI code that switches the local DL framework, i.e., NCNN and Darknet, call to the ”remote” DL framework call of the sandbox. Therefore, when there is an inference request, it will be forwarded to LEAP sandbox, the inference procedure will be performed in LEAP sandbox, and the inference result will be sent back. The application with the modified JNI code is called pAPP. Finally, we package the sensitive codes as sc-pAPP into the ramfs of a pre-distributed LEAP sandbox image.

We evaluate the LEAP's performance with these end-to-end demo Apps. We develop several applications with different models and DL frameworks and run the applications with both the CPU and GPU of the prototype. In addition to conducting the measurement on LEAP, We train four models, i.e., SqueezeNet [45], MobileNetV2 [34], DenseNet201 [46], and ResNet50 [37], for each framework.

Figure 15 shows the performance of running the demo applications in both LEAP. The CPU version means running the demo application with the big or little cores on Hikey960. And more than one cores represent the situation that it dynamically requests CPU cores from ROS for inference. When the demo applications deployed in LEAP uses the CPU to perform the inference, the inference speed for the big core is 2.9× to 3.4× faster than the little core for DarkNet, and 2.5× to 4.4× for NCNN, and 2.3× to 3.1× for MNN, respectively. Moreover, LEAP's flexible resource adjustment enables the inference speed on the big core to

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Nanjing University. Downloaded on March 09,2023 at 01:50:32 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/TMC.2022.3207745

improve 1.2× to 1.8× for DarkNet, 1.6× to 1.8× for NCNN, and 1.4× to 1.6× for MNN.

Without loss of generality, we compare the inference speed of CPU and GPU based on MNN. As SANCTUARY can only run with a single CPU core without GPU access, it will show how our secure GPU access can accelerate the DL Apps' performance. As shown in Figure 15d, when the demo Apps run with a little core to perform inference with GPU, it is 2.9× to 10.8× faster than the little core and 2.2× to 3.8× faster than the big core. When the demo Apps run on a big core with GPU, it is 7.7× to 14.8× faster than the little core and 2.2× to 5.22× (3.57× on average) faster than the big core.

7 LIMITATION

General DevOps. At present, our automatic DevOps tool can only apply to DL Apps to protect their valuable models. It only supports Java language since mobile Apps are mostly developed in Java. Although mobile Apps may also contain some native C/C++ libraries, supporting C/C++ language is not the scope of this work. Besides, our automatic DevOps tool is not fully automated since there is also some manual work that needs to be done by developers for it. However, the manual work is easy for developers since they only need to point out the entry points of the sensitive codes. Designing a general automatic DevOps tool for other types of Apps will have new challenges to be solved. For example, sc-App may rely on various Android shared libraries or system services, which should be resolved correctly in automatic DevOps. We plan to make the DevOps tool more general in our future work.

Peripheral Access. LEAP does not allow multiple sandboxes to access one peripheral in parallel, which may bring some constraints. First, it would cause a frozen GUI for ROS when one sandbox uses the GPU for the secured DL tasks. However, the frozen GUI lasts only a short time (usually hundreds of milliseconds) because the DL models used on mobile devices are usually lightweight. If such a short frozen period is unacceptable or the DL tasks require a quite long time for inference, the App can also choose to use the CPU for the secured inference. Performing the secured inference on the CPU would not cause a frozen GUI, and our flexible CPU resource adjustment can also enable an efficient secured DL inference. Second, peripherals cannot be shared simultaneously, which may reduce the potential benefits of parallelism. As we aimed to provide high security, we sacrificed some system usability. The other limitation is that LEAP cannot support all peripherals on mobile devices, which prevents it from being a general solution. We plan to make it to be a general design for more devices and enable peripherals securely shared by parallel sandboxes at a finer granularity in our future work.

Malicious Driver. In this work, we assume that the driver used in LEAP_{SOS} is benign and bug-free. Although a malicious or buggy driver can not affect other sandboxes, it may compromise the sandbox it resides in. To prevent this, we can refer to some driver isolation works [47] to prevent malicious drivers from compromising the sandbox.

Sandbox Density. Our sandbox isolation is based on an exclusive CPU design. Therefore, the maximum number of sandboxes is limited by the number of CPU cores on the device. We plan to increase the sandbox density in future work to support more parallel environments.

8 RELATED WORK

8.1 TEE designs based on TrustZone

NW-Side TEE Solutions. The first kind of work devotes itself to creating TrustZone-assisted isolation in NW to improve the TrustZone's usability. Figure 1 illustrates some representative works of this type, i.e., TrustICE [5], PrivateZone [6], OSP [7], and SANCTUARY [8]. We will compare these works with our LEAP one by one. TrustICE designs an isolated computing environment in NW without using a hypervisor. However, when the isolation environment is running, ROS and other isolation environments will be frozen. In addition, the TrustICE sandbox cannot adjust its resources on-demand flexibly. PrivateZone proposes an isolation environment in NW and enables security-critical code to run in the isolated environment instead of running in SW. PrivateZone can only maintain one isolation environment, so codes from different developers run in one sandbox. The lack of isolation among different developers' codes can cause security concerns. In addition, PrivateZone can neither flexibly adjust resources to balance the workload nor can it guarantee the peripheral access's security. OSP enables virtualization in NW to provide SGX-like enclaves. This work uses a hypervisor to support the enclave's isolation, and the hypervisor will bring overhead when the sensitive code is running [7]. In addition, OSP cannot support flexible resource adjustment and secure peripheral access. SANCTUARY aims to provide a NW isolation environment through TZASC [48], a hardware mechanism of TrustZone used to control memory access permission. Compared with LEAP, SANCTUARY can only support limited parallel execution. Moreover, it does not support secure peripheral access and flexible resource. There is another work, vTZ [17], which provides virtual TrustZone on cloud servers. Different from LEAP, vTZ is designed to provide each VM with its own virtual TrustZone rather than making it easier for applications in these VMs to enjoy TrustZone. Moreover, its virtualization-based method is oriented to cloud computing scenarios rather than mobile scenarios, which have more limited computing resources.

SW-Side TEE Solutions. The second kind of work tries to improve the SW's usability and security. Work [49] slices the security-critical part of an App by annotating the sensitive data in the source code and porting the sliced part into SW. TrustShadow [50] explores how to run legacy Apps in SW. It introduces a runtime to help legacy Apps run in SW without any modification. secTEE [25] proposes an Enclave-like design in SW to isolate the security-critical services from other SW software. TEEv [51] and PrOS [52] introduce the virtualization technology to the SW through software-based isolation. However, these works import the third-party executable code into SW and enlarge the TCB. A larger TCB is inherently more vulnerable to compromise, and the code imported by a third-party developer may exacerbate the security issues. Our LEAP has a tamper-resisted TCB. After development, no executable code will be added to the SW.

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Nanjing University. Downloaded on March 09,2023 at 01:50:32 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2022.3207745

8.2 DL Model Protection on Mobile Device

DarkneTZ [3] is the first work that attempts to utilize TrustZone to protect the DL models. Unfortunately, due to the limited resources, it can only put the last few layers of the model in TrustZone to defend against membership inference attacks [11]. Different from it, LEAP can enable to protect entire model in TEE without resource restrictions, which has a more powerful protection capability. Recently, OMG [12] managed to protect the whole DL model based on SANCTUARY [8]. However, it cannot satisfy the essential needs of DL Apps, such as easy adaptation, flexible resources, and GPU acceleration. These are all not supported by SANCTUARY while achieved by LEAP.

9 Conclusion

We present a developer-friendly Normal World TEE, LEAP, for mobile Apps. We comprehensively analyze the design requirements of App developers, and LEAP introduces four techniques to respond to developers' needs. We implement the LEAP prototype on Hikey960 and conduct a comprehensive evaluation of it. The results show that LEAP enables parallel protection sandbox running with full-fledged execution flexibility for the intelligent mobile Apps, which indicates that LEAP balances security and usability well in mobile scenarios.

Acknowledgments

We would like to thank the editors and the anonymous reviewers for their valuable comments helping us to improve this work. This work was supported by the National Key R&D Program of China (#2021YFB3100300, and #2020YFB1005900), NSFC (#61872180, #61872179, and #61872176), Jiangsu "Shuang-Chuang" Program, Jiangsu "Six-Talent-Peaks" Program, The Leading-edge Technology Program of Jiangsu Natural Science Foundation (#BK20202001), Science Foundation for Youths of Jiangsu Province (#BK20220772).

References

[1] T. Alves and D. Felton, "Trustzone: Integrated hardware and software security," accessed: Jun. 5, 2021. [Online]. Available: http://docplayer.net/51242536-Trustzone-integrated-hardware-and-software-security-enabling-trusted-computing-in-embedded-systems.html

[2] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto, "Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems," in *Proc. IEEE Symp. Secur. Privacy*, 2020, pp. 1416–1432.

[3] F. Mo, A. S. Shamsabadi, K. Katevas, S. Demetriou, I. Leontiadis, A. Cavallaro, and H. Haddadi, "Darknetz: towards model privacy at the edge using trusted execution environments," in *Proc. 18th Int. Conf. Mobile Syst. Appl. Serv.*, 2020, pp. 161–174.

[4] Linaro, "Open portable trusted execution environment," accessed: Jun. 5, 2021. [Online]. Available: https://www.op-tee.org/

[5] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang, "Trustice: Hardware-assisted isolated computing environments on mobile devices," in *Proc. 45th Annu. IEEE/IFIP Int. Conf. Depend. Syst. Netw*, 2015, pp. 367–378.

[6] J. Jang, C. Choi, J. Lee, N. Kwak, S. Lee, Y. Choi, and B. B. Kang, "Privatezone: Providing a private execution environment using arm trustzone," *IEEE Trans. Depend. Secure Comput.*, vol. 15, no. 5, pp. 797–810, 2016.

[7] Y. Cho, J. Shin, D. Kwon, M. Ham, Y. Kim, and Y. Paek, "Hardware-assisted on-demand hypervisor activation for efficient security critical code execution on mobile devices," in *2016 USENIX Annu. Tech. Conf.*, 2016, pp. 565–578.

[8] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf, "Sanctuary: Arming trustzone with user-space enclaves," in *Netw. Distrib. Syst. Secur. Symp*, 2019.

[9] M. Xu, J. Liu, Y. Liu, F. X. Lin, Y. Liu, and X. Liu, "A first look at deep learning apps on smartphones," in *Proc. 28th Int. Conf. World Wide Web*, 2019, pp. 2125–2136.

[10] Z. Sun, R. Sun, L. Lu, and A. Mislove, "Mind your weight (s): A large-scale study on insufficient machine learning model protection in mobile apps," in *30th USENIX Secur. Symp.*, 2021.

[11] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, "Membership inference attacks against machine learning models," in *Porc. IEEE Symp. Secur. Privacy*, 2017, pp. 3–18.

[12] 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 Des., Automat. Test Europe Conf. Exhib.*, 2020, pp. 460–465.

[13] ARM, "Arm security technology building a secure system using trustzone technology," accessed: Jun. 5, 2021. [Online]. Available: https://developer.arm.com/documentation/genc009492/c

[14] *ARM CoreLink TZC-400 TrustZone Address Space Controller*, 2013. [Online]. Available: https://developer.arm.com/documentation/ddi0504/c/

[15] *PrimeCell Infrastructure AMBA 3 TrustZone Protection Controller (BP147)*, 2004. [Online]. Available: https://developer.arm.com/documentation/dto0015/a/

[16] *ARM System Memory Management Unit Architecture Specification*, 2016. [Online]. Available: https://developer.arm.com/documentation/ihi0062/dc/

[17] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "vtz: Virtualizing arm trustzone," in *26th USENIX Secur. Symp.*, 2017, pp. 541–556.

[18] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: cold-boot attacks on encryption keys," *Commun. ACM*, vol. 52, no. 5, pp. 91–98, 2009.

[19] A. Huang, "Keeping secrets in hardware: The microsoft xbox tm case study," in *Int. Workshop Cryptographic Hardware Embedded Syst.*, 2002, pp. 213–227.

[20] M. G. Kuhn, "Cipher instruction search attack on the bus-encryption security microcontroller ds5002fp," *IEEE Trans. Comput.*, vol. 47, no. 10, pp. 1153–1157, 1998.

[21] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+flush: a fast and stealthy cache attack," in *Proc. 13th Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*, 2016, pp. 279–299.

[22] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *23rd USENIX Secur. Symp.*, 2014, pp. 719–732.

[23] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Cryptographers' track RSA Conf.*, 2006, pp. 1–20.

[24] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *24th USENIX Secur. Symp.*, 2015, pp. 897–912.

[25] S. Zhao, Q. Zhang, Y. Qin, W. Feng, and D. Feng, "Sectee: A software-based approach to secure enclave architecture using tee," in *Proc. 2019 ACM SIGSAC Conf. Comput. Commun. Secur.*, 2019, pp. 1723–1740.

[26] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan, "Optimizing java bytecode using the soot framework: Is it feasible?" in *Proc. 9th Int. Conf. Compiler Construction*, 2000, pp. 18–34.

[27] E. Keller, J. Szefer, J. Rexford, and R. B. Lee, "Nohype: virtualized cloud infrastructure without the virtualization," in *Proc. 37th Annu. Int. Symp. Comput. architecture*, 2010, pp. 350–361.

[28] S. Bhat, "Interaction of suspend code (s3) with the cpu hotplug infrastructure," accessed: Jun. 5, 2021. [Online]. Available: https://www.kernel.org/doc/Documentation/power/suspend-and-cpuhotplug.txt

[29] *ARM Cortex-A Series – Programmer's Guide for ARMv8-A*, 2015. [Online]. Available: https://static.docs.arm.com/den0024/a/DEN0024A_v8_architecture_PG.pdf

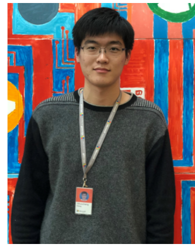
[30] W. Song, H. Yin, C. Liu, and D. Song, "Deepmem: Learning graph neural network models for fast and robust memory forensic

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Nanjing University. Downloaded on March 09,2023 at 01:50:32 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in IEEE Transactions on Mobile Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TMC.2022.3207745

analysis," in *Proc. 2018 ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 606–618.
[31] M. "mina86" Nazarewicz, "A deep dive into cma," accessed: Jun. 5, 2021. [Online]. Available: https://lwn.net/Articles/486301/
[32] T. kernel development community, "Memory hotplug," accessed: Jun. 5, 2021. [Online]. Available: https://www.kernel.org/doc/html/latest/admin-guide/mm/memory-hotplug.html
[33] L. W. McVoy and C. Staelin, "lmbench: Portable tools for performance analysis," in *USENIX Annu. Tech. Conf.*, 1996, pp. 279–294.
[34] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proc. IEEE Conf. Comput. vision pattern Recognit.*, 2018, pp. 4510–4520.
[35] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. vision pattern Recognit.*, 2015, pp. 1–9.
[36] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances neural Inf. Process. Syst.*, vol. 25, pp. 1097–1105, 2012.
[37] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. vision pattern Recognit.*, 2016, pp. 770–778.
[38] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *Proc. AAAI Conf. Artif. Intell.*, 2017.
[39] X. Jiang, H. Wang, Y. Chen, Z. Wu, L. Wang, B. Zou, Y. Yang, Z. Cui, Y. Cai, T. Yu, C. Lv, and Z. Wu, "Mnn: A universal and efficient inference engine," in *3rd Conf. Mach. Learn. Syst.*, 2020.
[40] P. Whelan, "Model zoo," accessed: Jun. 5, 2021. [Online]. Available: https://github.com/BVLC/caffe/wiki/Model-Zoo
[41] L. B. N. Laboratory, "iperf3: A tcp, udp, and sctp network bandwidth measurement tool," accessed: Jun. 5, 2021. [Online]. Available: https://github.com/esnet/iperf
[42] C. Göttel, P. Felber, and V. Schiavoni, "iperftz: Understanding network bottlenecks for trustzone-based trusted applications," in *Int. Symp. Stabilizing, Saf. Secur. Distrib. Syst.*, 2019, pp. 178–193.
[43] Tencent, "ncnn," accessed: Jun. 5, 2021. [Online]. Available: https://github.com/Tencent/ncnn
[44] J. Redmon, "Darknet: Open source neural networks in c," accessed: Jun. 5, 2021. [Online]. Available: http://pjreddie.com/darknet/
[45] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and < 0.5mb model size," in *arXiv:1602.07360*, 2016.
[46] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proc. IEEE Conf. Comput. vision pattern Recognit.*, 2017, pp. 4700–4708.
[47] Y. Huang, V. Narayanan, D. Detweiler, K. Huang, G. Tan, T. Jaeger, and A. Burtsev, "Ksplit: Automating device driver isolation," in *16th USENIX Symp. Operating Syst. Des. Implementation*, 2022, pp. 613–631.
[48] ARM, "Trustzone address space controller," accessed: Jun. 5, 2021. [Online]. Available: https://www.arm.com/products/silicon-ip-security/address-space-controllers
[49] K. Rubinov, L. Rosculete, T. Mitra, and A. Roychoudhury, "Automated partitioning of android applications for trusted execution environments," in *2016 IEEE/ACM 38th Int. Conf. Softw. Eng.*, 2016, pp. 923–934.
[50] 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 *Proc. 15th Annu. Int. Conf. Mobile Syst. Appl. Serv.*, 2017, pp. 488–501.
[51] W. Li, Y. Xia, L. Lu, H. Chen, and B. Zang, "Teev: virtualizing trusted execution environments on mobile platforms," in *Proc. 15th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environments*, 2019, pp. 2–16.
[52] D. Kwon, J. Seo, Y. Cho, B. Lee, and Y. Paek, "Pros: Light-weight privatized se cure oses in arm trustzone," *IEEE Trans. Mobile Comput.*, vol. 19, no. 6, pp. 1434–1447, 2019.

Lizhi Sun received the BS degree from the Department of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, in 2018. He is currently working toward the PhD degree with the Department of Computer Science and Technology of Nanjing University. His research interests include mobile computing, system security and trusted execution environments.



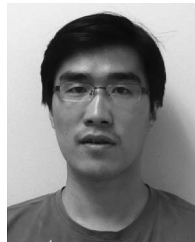
Shuocheng Wang received the BS degree from the School of Computer Science and Engineering, Sun Yat-sen University, in 2018. He is currently working toward the MS degree with the Department of Computer Science and Technology of Nanjing University. His research interests include system security, trusted execution environments.

Hao Wu received the PhD degree, with the Distinguished Dissertation Award, from the Department of Computer Science and Technology, Nanjing University, in 2021. He is currently an assistant researcher with the Department of Computer Science and Technology, Nanjing University. His research interests include intelligent mobile computing and privacy computing.

Yuhang Gong received the BS degree from the Department of Computer Science and Technology, Northwestern Polytechnical University in 2019. He is currently working toward the MS degree with the Department of Computer Science and Technology of Nanjing University. His research interests include Android program analysis.

Fengyuan Xu received the PhD degree, with the Distinguished Dissertation Award, from the College of William and Mary. He is currently a professor with the Computer Science Department, Nanjing University. His research interests include the broad areas of systems and security, with a focus on data driven security analytics, deep learning security & applications, mobile & edge computing, and trusted execution environments. He is a member of the IEEE.

Yunxin Liu received the BS, MS, and PhD degrees from the University of Science and Technology of China, Tsinghua University, China, and Shanghai Jiao Tong University, China, respectively. He is a Guoqiang Professor and Principal Investigator at Institute for AI Industry Research (AIR), Tsinghua University. His current research interests are focused on mobile and edge computing. He received MobiCom 2015 Best Demo Award, PhoneSense 2011 Best Paper Award, and SenSys 2018 Best Paper Runner-up Award. He is a senior member of the IEEE.



Hao Han received the PhD degree in computer science from the College of William and Mary, Williamsburg, VA, USA, in 2013. He is currently a professor with the Computer Science Department, Nanjing University of Aeronautics and Astronautics. His research interests include wireless networks, mobile computing, cloud computing and RFID systems. He is a member of the IEEE.

Sheng Zhong received the BS and MS degrees from Nanjing University, in 1996 and 1999, respectively and the PhD degree from Yale University, in 2004, all in computer science. His research interest include security, privacy, and economic incentives.

© 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Nanjing University. Downloaded on March 09,2023 at 01:50:32 UTC from IEEE Xplore. Restrictions apply.