UIEE: Secure and Efficient User-space Isolated Execution Environment for Embedded TEE Systems

Huaiyu Yan[†], Zhen Ling[†*], Xuandong Chen[†], Xinhui Shao[†§], Yier Jin[‡], Haobo Li[†]
Ming Yang[†], Ping Jiang[†], Junzhou Luo[†¶]
[†]Southeast University, Email: {huaiyu_yan, zhenling, xdc, xinhuishao, haobo_li, yangming2002, jiangping, jluo}@seu.edu.cn
[‡]University of Science and Technology of China, Email: jinyier@ustc.edu.cn
[§]City University of Hong Kong [¶]Fuyao University of Science and Technology

***Abstract*—Trusted execution environments (TEE) have been widely explored to enhance security for embedded systems. Existing embedded TEE systems run with a small memory footprint and only provide security critical functionalities in order to maintain a minimal trusted computing base (TCB). Unfortunately, such design choice results in the dilemma that these TEE systems are short in software resources, making it difficult t o e xecute c omplex a pplications w ith l arge c ode bases inside of embedded TEEs. In this paper, we propose a user-space isolated execution environment (UIEE) so as to augment TEE capabilities by directly running un-modified d ata p rocessing applications inside of TEEs without increasing the TCB size. UIEE constructs a sandboxed environment by dynamically allocating a sufficient m emory r egion f or a pplications a nd i solates i t from both the rich execution environment (REE) and TEE, defending UIEE from REE attacks while protecting TEE from a potentially compromised UIEE application. Additionally, we propose a library OS (*i.e.*, Linux kernel library, LKL) based UIEE runtime environment that can provide standard C runtime APIs to UIEE applications. In order to solve the LKL concurrency issues, we propose an LKL thread synchronization mechanism to run the multi-threaded LKL inside of the UIEE which features a singled thread execution model. Furthermore, we design a novel on-demand thread migration mechanism to realize LKL context switching inside of UIEE. We implement and deploy a UIEE prototype on an NXP IMX6Q SABRE-SD evaluation board, and successful run 8 real-world *libc*-based applications inside of UIEE without modification. T he e xperimental r esults s how that UIEE incurs negligible performance overhead. We are the first to propose a TrustZone-oriented LibOS and evaluate its feasibility as well as security features.**

I. Introduction

Trusted execution environments (TEEs) have been widely used to protect the data confidentiality a nd c ode i ntegrity of embedded systems. Modern TEEs like ARM TrustZone [1] create a hardware-enforced isolated region so that programs running inside such region can be protected from attacks issued from the external environment, which are commonly referred to as the rich execution environment (REE). A typical embedded TEE system consists of several trusted applications (TAs) that provide security-critical services to REE programs and a trusted operating system (Trusted OS) to manage the life cycle of each TA. Additionally, in order to maintain a minimal trusted computing base (TCB), existing TEEs only implement necessary operations such as en-/decryption, digital signatures and attestation, and feature a single-threaded execution model. Such design choice would preserve a relatively small attack surface and reduce the number of potential TEE vulnerabilities, enhancing the security level for embedded systems.

The compact design of existing embedded TEEs makes it difficult to realize comprehensive data processing logic within these environments. In contrast to REE systems (*e.g.*, Linux-based systems), existing trusted OSes only provide a small set of system call services with rather limited capabilities. Moreover, TA libraries cannot support common C runtime APIs like file operations which are widely used by various REE applications. In order to fully utilize embedded TEEs for secure data processing, existing research projects take two directions: 1) **application slicing & adaption**: This approach retrofits an existing REE application by identifying its security-critical data processing logic (*i.e.*, cryptographic operations [2], privacy-related I/O operations [3][4] or machine learning model layers [5][6][7]) and adapting such logic into TEEs. The extracted security-critical parts usually have small code bases, thus maintaining a minimal TCB increase with their adaption into TEEs. However, such approach has poor scalability since it works in a case-by-case manner and it takes huge engineering efforts to retrofit existing complex applications [2]. Moreover, the interfaces between the TEE-side security critical part and the remaining program inside REE vary significantly across different applications, making it an error-prone and exhausting task to ensure the secure interaction among them. 2) **isolated execution environment (IEE)**: This approach creates a separate IEE, isolated from both REE and TEE, where security-critical programs run. The IEE approach can protect applications from potential REE attacks while allowing supervised access from IEE applications to TEE resources, thus maintaining a minimal TCB. However, such approach features the very same issue haunting the existing embedded TEEs that existing research projects [8][9][10][11][12][13]

*Corresponding author: Prof. Zhen Ling of Southeast University, China.

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.230208
www.ndss-symposium.org

propose their own unique software runtime environments and consequently developers are still required to adapt existing applications into such execution environments, involving a lot of engineering efforts. Instead, TrustShadow [14] and Shelter [15] choose to run unmodified applications inside IEEs by statically linking applications with their dependency libraries including *libc* and forwarding most system calls to REE, which results in notable runtime performance overhead.

In this paper, we propose a user-space isolated execution environment (UIEE) for embedded TEE systems, that can directly run existing Linux applications inside TEEs without increasing the TCB size. We focus our design on TrustZone-based TEE systems while the general design rationale can be applied to TEEs empowered by other architectures (*i.e.*, Intel SGX/TDX [16], AMD SEV [17], ARM CCA [18]). To begin with, UIEE creates an isolated execution environment separated from both REE and TEE. Specifically, a UIEE runtime memory is dynamically allocated and configured as secure memory using the TrustZone address space controller (TZASC) [19]. Additionally, a separate page table is constructed inside the trusted OS to map the UIEE memory region into a constrained virtual address region so as to isolate UIEE programs from the rest of the TEE components and in turn maintain a minimal TCB size. Inside of the UIEE memory region, we provide a standard C runtime for applications based on a customized C library (*libc*) as well as a library OS (LibOS) [20][21], namely the Linux kernel library (LKL) [22]. To be specific, *libc* provides common C library functions for various applications together with their dependency libraries and issues system calls to the LibOS in form of direct function calls. Additionally, most dependency functions required by the LibOS are provided by reusing the existing TA libraries and trusted OS services while the few remaining ones are redirected to REE.

UIEE design faces several challenges. Since the existing TEEs cannot provide *pthread* multi-threading APIs required by LKL, we investigate a two-stage UIEE execution model where the LibOS is first initialized inside REE using a standard C library to spawn necessary LibOS threads for the first initialization stage. For the second stage, the whole UIEE memory region is isolated from REE and then the control flow transfers to UIEE through the trusted OS. Furthermore, to properly handle the concurrency issues among LKL kernel threads, we propose an LKL thread synchronization mechanism so that all LKL kernel threads finish initialization and go into sleeping mode with their contexts within the REE Linux kernel before entering UIEE. Once the UIEE application starts execution, we propose a novel on-demand thread migration mechanism in order to wake up a sleeping LKL kernel thread and transfer its control flow from the REE Linux kernel to UIEE upon the LKL context switching.

We develop a UIEE prototype and deploy it on an IMX6Q SABRESD development board in order to evaluate its security and performance. The current UIEE prototype introduces a reasonably small TCB increase with a 0.46% increase upon the trusted OS code base and a 3.37% increase upon the trusted OS boot image. Additionally, we successfully run 8 *libc*-based applications, including database, multi-media processing, machine learning, etc., inside UIEE and no modification is made to any application. The experimental results show that UIEE introduces negligible performance overhead to various real-world application workloads during the runtime phase.

In summary, this paper offers the following contributions

- We are the first to offer a TrustZone-oriented isolated execution environment (IEE) with standard C runtime support that enables secure and efficient execution of unmodified REE applications.
- In order to run a multi-threaded LibOS inside a single-threaded TEE, we propose a two-stage LKL bootstrapping scheme as well as the thread synchronization mechanism to solve the LKL thread creation issues. Additionally, we propose an on-demand thread migration mechanism to realize LKL context switching within UIEE.
- We implement a UIEE prototype and assess its security and performance feasibility. The experimental results show that UIEE introduces little TCB increase as well as little performance overhead to real-world application workloads.

II. Motivation & Observation

In this section, we first present our motivations by making a brief summary of current TrustZone-based IEEs in terms of their runtime environments. Then, we present the basic idea of our UIEE approach. Finally, we elaborate on challenges to the UIEE design together with several key observations that help solve these challenges.

TABLE I: Summary on Existing TrustZone-based IEEs

	Hardware Reqs	Software Resource			Flexibility	
		Cust App/Lib	*libc* App/Lib	Linux Syscall	Dynamic Linking	Scalability
Komodo [8]	TZ	○	○	○	○	○
SecTEE [9]	TZ	○	○	○	○	○
OSP [10]	TZ + Virt	●	○	○	○	○
PrivateZone [11]	TZ + Virt	●	○	○	○	○
TrustICE [12]	TZ + Core	●	○	○	○	○
SANCTUARY [13]	TZ	●	○	○	○	○
TLR [23]	TZ	◐	○	○	◐	●
WaTZ [24]	TZ	◐	○	○	◐	●
TrustShadow [14]	TZ	●	●	◐	○	◐
Shelter [15]	CCA	●	●	◐	○	◐
UIEE	TZ	●	●	●	●	●

●: fully supported, ◐: partially supported, ○: not supported.

A. Motivations

Current TrustZone-based embedded TEEs use as little hardware and software resources as possible in order to maintain a minimal trusted computing base (TCB) and in turn fail to support complex and sophisticated trusted applications. Usually, a small amount of physical memory is statically reserved for the TEE during the booting phase. Additionally, existing TEEs only provide a limited set of programming APIs, mostly related to cryptographic operations. Therefore, it is challenging to deploy data processing applications that not only consume large runtime memory but also require various API dependencies (*i.e.* database, multi-media processing, machine learning etc.) inside the TEE.

Although various research projects have been proposed to extend TrustZone capabilities, they still have several limitations in terms of runtime software resources. Instead of retrofitting existing REE applications in a case-by-case manner, we focus on the IEE approaches due to their potential flexibility, as shown in Table I. 1) **Self-contained functions**: Both Komodo [8] and SecTEE [9] realize Intel-SGX enclave primitives [16] based on ARM TrustZone, and yet they can only run self-contained functions inside TEEs like an SGX enclave does. 2) **Customized applications & Libraries**: SANCTUARY[13], OSP[10], PrivateZone[11] and TrustICE[12] propose TrustZone-based sandbox environments which are isolated from both TEE and REE, and applications can be protected from potential REE attacks without increasing the TCB size. However, developers are required to develop separate applications for these systems according to their customized APIs and runtime libraries in a case-by-case manner. Furthermore, OSP [10] and PrivateZone [11] leverage ARM virtualization extensions which are not available in most ARMv7 platforms. 3) **Language runtime**: TLR [23] and WaTZ [24] propose a language runtime for Webassembly and Microsoft .NET respectively inside TEEs while existing embedded TEE systems still favor C programs for its efficiency. 4) ***libc* applications**: TrustShadow [14] directly runs *libc*-based applications by forwarding Linux kernel system calls issued by applications to the REE Linux kernel. Shelter [15] takes a similar design by running an application inside an ARM confidential compute architecture (CCA) confidential machine [18] and provides OS services through forwarded Linux kernel system calls, resulting in relatively high runtime overhead. Additionally, both projects only provide a subset of the Linux kernel system call interfaces and it takes a huge amount of engineering efforts to implement the forwarding routines for all system calls. Besides, the ARM CCA architecture introduced in ARMv9.2 [18] is not available in most existing embedded platforms. In consequence, we are motivated to ask the following inspiring question:

Question: *How to securely and efficiently run unmodified REE applications inside TEE without expanding the system TCB size?*

Due to the limitations of the existing TrustZone-based TEEs, we are motivated to propose an approach that shall achieve the following goals (**G**): **G1: Hardware Compatibility.** Such approach shall only rely on minimal ARM architecture primitives which are pervasively available among various embedded ARM platforms[1], including IoT devices, mobile devices and edge devices, etc. **G2: Security.** In order to maintain a minimal TCB, the modifications to the existing TEE components shall be as little as possible. Moreover, since an application as well as its dependency libraries has a large code base, they shall be separated from the existing TEE components without being considered as part of the TCB. **G3: *libc*-based Runtime.** We are intended to provide a *libc*-based runtime so that developers can develop applications based on *libc* APIs. Furthermore, existing applications or libraries built against *libc* can directly run inside the TEE. Therefore, such approach could provide a comprehensive set of runtime APIs and also save a huge amount of engineering efforts in application adapting. **G4: Performance.** The proposed runtime shall incur little performance overhead to applications during runtime execution.

[1]In this paper, we specifically focus on ARM Cortex-A platforms with TrustZone extensions.

B. Basic Design

In order to support a comprehensive TEE runtime environment with a *libc* runtime, we plan to build a TrustZone-based user-space isolated execution environment (UIEE) where an internal library OS (LibOS) [20][21] together with a *libc* resides. We first dynamically allocate the UIEE memory region from the REE memory which is later configured as secure memory using TrustZone address space controller (TZASC) [19] (**G1**). Moreover, we isolate UIEE from TEE by maintaining a separate page table for UIEE memory regions. Consequently, UIEE can protect applications from a malicious REE and at meantime defend TEE programs (*i.e.*, the trusted OS and TAs) against potential attacks issued from UIEE programs, thus maintaining a minimal TCB (**G2**). Afterwards, applications as well as its dependency libraries, a LibOS *libc*, a LibOS as well as a LibOS dependency library are loaded into the UIEE memory region and mapped to a predefined virtual address region. The LibOS *libc* can provide a *libc* runtime for applications while the LibOS can provide all system call interfaces required by the LibOS *libc* (**G3**). Additionally, the LibOS still requires external operations provided by the LibOS dependency library to realize privileged functionalities such thread management or clock time retrieval, which can be implemented by reusing existing TEE services or being redirected to REE. Finally, LibOS provides system call services directly within UIEE in form of function invocation, resulting in better performance compared with the forwarded system call approaches discussed in §II-A (**G4**).

C. Observation & Challenges

As the very first thing, we need to determine which LibOS is best suitable for TrustZone-based TEEs. Since most existing LibOSes [25][26][27][28][29][30][31] mainly focus on x86-based cloud platforms and some specifically target the SGX-based TEEs, an idea LibOS shall satisfy the following criteria (**Crit**): 1) it can provide comprehensive REE kernel services (*e.g.*, Linux kernel system calls); 2) instead of implementing a new LibOS, we expect that the target LibOS can be adapted to ARM-based platform without major modification and its required external services can be realized using existing TEE services in order to save engineering efforts. In consequence, we choose the Linux kernel library (LKL) [22] as the target LibOS since it is a *libc*-based architectural port of the Linux kernel. The Linux kernel implements most REE system call services (**Crit1**). Additionally, LKL has inherent support for the ARM architecture and requires a well-defined set of external services in form of function pointers, which are

architecture-independent and can be realized using existing TEE services (**Crit2**).

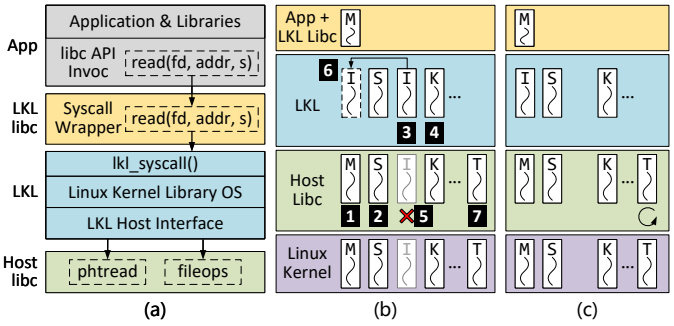

Fig. 1: **The LKL Architecture and Threading Model.** Fig. (a) shows the overall LKL architecture. Fig. (b) shows the thread states during the LKL initialization phase where **M**: main thread, **S**: swapper thread, **T**: timer thread, **I**: init thread, **K**: LKL kernel thread. The read cross ($\times$) in Fig. (b) indicates that a *libc* thread exits and its corresponding Linux kernel thread context is cleared. Fig. (c) shows the final thread states after LKL initialization.

As shown in Figure 1-(a), a typical LKL architecture consists of four components, namely an application, an LKL *libc*, LKL and a host *libc* which all run in the user space. Taking file reading as an example, the application invokes the standard file reading function, *i.e.* `read()`, defined in the LKL *libc* which further issues a `read` system call to LKL through a direct function call to the corresponding LKL service routine, *i.e.* `sys_read()`. Then, LKL handles this system call in a similar way that a normal Linux kernel does and finally invokes the host *libc* to conduct real block I/O, reading the file contents from the disk. Finally, the whole function invocation path is returned and the application retrieves the file contents for further processing.

LKL requires that the host platform implement a set of external services in form of function pointers which we refer to as the LKL host interface. As an architecture port of the Linux kernel, LKL emulates hardware capabilities expected by the Linux kernel based on the host interface, including operations related to semaphore, mutex, thread management, memory allocation, timing, I/O memory mapping and jump buffer, etc. Specifically, LKL leverages the *pthread* APIs to spawn multiple kernel threads during the initialization phase and these kernel threads conduct crucial Linux kernel tasks during the runtime phase such as block I/O. Such requirement leads to the first challenge.

Challenge 1: Thread Creation Issues. Existing embedded TEE systems [32] take a single-threaded programming model and the current embedded TEE OSes cannot provide *pthread*-compatible multi-threading APIs which are required by LKL.

Instead of porting *pthread*-compatible libraries into TEEs, which increases the TCB size, we investigate the LKL threading model and try to solve such dilemma accordingly. As shown in Figure 1-(b), a typical LKL thread context is maintained by 4 entities, namely the LKL *libc*, LKL, the host *libc* and the host Linux kernel. During the LKL initialization phase, the host *libc* runs in the main thread context, loading and linking applications, their dependency libraries and LKL (Fig. 1-(b), ❶). Then, the host *libc* creates a new thread to execute the LKL startup routine which we refer to as the swapper thread [33] with LKL process ID 0 (Fig. 1-(b), ❷). The swapper thread is responsible for initializing the whole LKL Linux kernel using the host interface. After the LKL Linux kernel has been fully initialized, it creates the init thread with LKL process ID 1 and attaches it to a host *libc* as well as a host kernel thread context (Fig. 1-(b), ❸). Meanwhile, multiple Linux kernel threads are created and they execute concurrently, interleaving each other's time slice (Fig. 1-(b), ❹). After the init process finishes initialization, the contexts of its corresponding host *libc* thread and host Linux kernel thread are cleared (Fig. 1-(b), ❺). Then, LKL attaches the init's LKL thread to the main thread's host *libc* thread, changing its common name from "init" to "host0" and all following system calls issued from the application are handled in this LKL thread context (Fig. 1-(b), ❻). Moreover, LKL creates a *host-libc*-based timer to emulate a hardware timer and, upon a timer expiration, a separate timer thread is created by the host *libc* which invokes the LKL timer expiration interrupt handler (Fig. 1-(b), ❼). The final state of all LKL threads shown in Figure 1-(c) results in our first observation.

Observation 1: Most LKL kernel threads only run during the initialization phase and never get scheduled afterwards. Therefore, we do NOT necessarily need all LKL kernel threads inside UIEE.

Based on such key observation, we are inspired to investigate a two-stage LKL bootstrapping method. For the first stage, LKL is fully initialized inside REE using the *host-libc*-based host interface until all its kernel threads complete initialization. Then, after the control flow transfers back to the main thread with all LKL kernel threads in sleeping mode, we change the LKL host interface to one that is implemented based on TEE services by conducting transparent runtime dynamic linking. For the second stage, the main thread invokes REE-side TEE service APIs in order to transfer the control flow to TEE where the trusted OS gets the chance to isolate the whole UIEE memory region including applications and LKL from REE and provides all LKL external services. With such approach, all LKL threads are properly created and initialized inside REE. Additionally, such design choice can solve **Challenge 1** and maintain a minimal TCB without supporting *pthread* multi-threading APIs inside TEE (§IV-A).

Nevertheless, when applications run within UIEE after the two-stage LKL bootstrapping process, some kernel threads are scheduled for crucial kernel tasks such as block I/O and interrupt handling, leading to the second challenge.

Challenge 2: Context Switching Issues. Once UIEE memory isolation is applied, the LKL thread context is split into two parts where the thread context maintained by the REE Linux kernel remains in REE while other contexts are within TEE. In turn, LKL cannot conduct context switching inside UIEE with an in-complete thread context.

To solve such challenge, we investigate the LKL context switching mechanism. In order to handle concurrency, LKL introduces a per-thread scheduling semaphore [34] so that there is only one thread could be running at a time. Upon context switching, LKL wakes up the thread scheduled to run by releasing its corresponding semaphore through the semaphore APIs provided by the host interface. The host interface then invokes the Linux kernel fast mutex (`futex`) subsystem that switches the thread state from sleeping mode to running mode. Such mechanism results in our second observation.

Observation 2: We can reuse the REE Linux kernel `futex` subsystem to rebuild the LKL thread context inside the trusted OS, enabling context switching inside UIEE based on trusted OS scheduling services.

Based on this observation, when an LKL kernel thread is scheduled to run by LKL for the first time, we redirect the semaphore releasing operation from UIEE to the REE Linux kernel where we also monitor the process states of all LKL kernel threads. Once an LKL thread is waked up and scheduled to run by the REE Linux kernel, we transfer its control flow directly from the REE Linux kernel to the trusted OS where a corresponding TEE-side thread context is created. Finally, the trusted OS resumes execution of the LKL thread inside UIEE and all following context switches are conducted entirely inside TEE, thus solving **Challenge 2**.

III. Assumption & Threat Model

We assume that all TEE programs including the trusted OS and TAs are trustworthy and secure. Meanwhile, the REE programs are assumed secure during the system booting phase, which can be ensured through secure boot [35]. Therefore, the REE Linux kernel as well as applications can be considered secure and initialize the UIEE environment into a trusted state. Once the UIEE environment is properly set up after the UIEE initialization phase, REE programs including the Linux kernel may be compromised, thus becoming in-secure during the UIEE runtime phase.

We assume that the underlying hardware's implementation adheres to its corresponding specifications and functions in the same manner. Also, we assume that the deployment platform is equipped with TrustZone-aware peripherals, specially TZASC, in order to accomplish dynamic memory isolation with minimal hardware requirements.

We assume that a potential attacker may compromise the REE Linux kernel and try to retrieve security critical data from the UIEE applications by launching privileged attacks from REE. Moreover, the attacker may also try to steal data from the trusted OS or TAs by exploiting vulnerabilities of UIEE programs. However, since the attacker has the capability of shutting down the whole system, we do NOT defend against Denial-of-Service (DOS) attacks. Additionally, advanced hardware-oriented attacks are considered out of scope, including side channel attacks [36], [37], [38], bus snooping attacks [39], DMA attacks [40] and cold boot attacks [41], etc.

IV. UIEE Design

In this section, we elaborate on the detailed UIEE design. First, we present the general UIEE architecture, including the software components and the basic workflow. Then, we elaborate on several key UIEE mechanisms namely the memory isolation scheme to isolate UIEE from both REE and TEE, and the thread management mechanisms to enable context switching within UIEE.

A. UIEE Overview

We design the user-space isolated execution environment (UIEE), a comprehensive TrustZone-based IEE environment with *libc* support. There are 7 UIEE software components in total:

- **Application & Libraries** are the application code built with *libc* APIs.
- **LKL *libc*** provides a C runtime for applications.
- **LKL** is the LibOS providing Linux system call services to the LKL *libc*.
- **Trusted Application Library (*libtee*)** is a customized library that implements LKL host interfaces inside UIEE based on existing TEE services.
- **UIEE Driver** is an REE Linux kernel module responsible for allocating an initial memory region for UIEE which will later be isolated from REE.
- **UIEE Loader** is a REE dynamic ELF loader to load the corresponding ELF files of UIEE software components into the initial UIEE memory region. Additionally, the UIEE loader also provides the external services required by LKL during the initialization phase.
- **UIEE Session Manager** is a trusted OS module that enforces UIEE memory isolation and manages TEE thread contexts for UIEE.

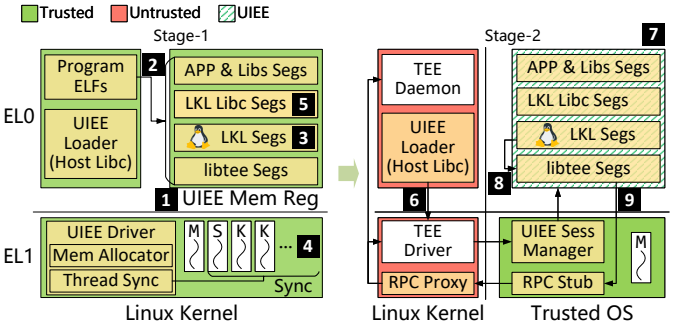

Fig. 2: **General Architecture of UIEE.** Step numbers 1-5 belong to the first stage while step numbers 6-9 belong to the second stage.

We investigate a two-stage UIEE bootstrapping scheme where LKL and LKL *libc* are initialized inside REE for the first stage while the UIEE runtime environment is established for the second stage to address the LKL kernel thread creation issues discussed in **Challenge 1**. As shown in Figure 2, for the first bootstrapping stage, right after the secure boot process, the customized UIEE loader implemented based on an existing *libc* invokes the UIEE driver to allocate a continuous physical memory region as the UIEE runtime memory and map all memory pages into a predefined virtual address space (Fig. 2, ❶). Then, the UIEE loader maps the corresponding ELF files of the applications, the libraries, the LKL *libc*, LKL and *libtee* into its virtual address space, resolves their ELF headers to locate loadable segments and then relocates each loadable segments into the allocated UIEE memory region through memory copy. Additionally, the UIEE loader conducts dynamical linking among all segments and specifically links LKL against the loader itself (*i.e.*, the host *libc*) as well as initializing the UIEE runtime stack and heap (Fig. 2, ❷). Once all programs have been properly linked, the UIEE loader invokes the LKL initialization routine, which in turn creates various kernel objects and spawns multiple kernel threads using the POSIX APIs defined by the UIEE loader (Fig. 2, ❸). During the LKL initialization process, the UIEE loader waits until all LKL kernel threads complete initialization. To this end, we design a thread synchronizer inside the UIEE driver that checks on the LKL thread states on behalf of the UIEE loader, ensuring that all LKL kernel threads are initialized and go into sleeping mode. After all LKL threads are fully initialized, the UIEE loader pauses the LKL timer and switches the LKL host interface from the UIEE-loader-defined one to the *libtee*-defined one by conducting transparent dynamic linking process. Note that such runtime switching is thread-safe since all LKL threads are sleeping without any timer expiration interrupts (Fig. 2, ❹). As the last step of the first stage, the UIEE loader invokes the initialization routine of the LKL *libc* (Fig. 2, ❺).

As for the second bootstrapping stage, the UIEE loader transfers the control flow to the TEE by invoking the REE-side TEE service APIs, passing the application entry point and the stack top address as arguments (Fig. 2, ❻). After retrieving the arguments, the UIEE session manager configures the UIEE physical memory region as secure memory using TZASC and reconstructs the UIEE page table inside the trusted OS in order to achieve memory isolation (§IV-B). Then, it creates a new TA session for the UIEE main thread as well as essential user-space thread contexts. Then, it transfers the control flow to the application by jumping into the retrieved entry point (Fig. 2, ❼). During the application runtime phase, the LKL *libc* together with LKL provides a *libc*-based runtime for the application and its dependency libraries. Most of the LKL host interface invocations can be directly served by *libtee* and the trusted OS (Fig. 2, ❽) while few remaining ones are redirected to REE through the remote procedure call stubs and handled by either the REE-side TEE driver or the REE-side TEE daemon (Fig. 2, ❾). As for LKL context switching, we propose a novel on-demand LKL thread migration mechanism to wake up a sleeping LKL kernel thread and transfer its control flow to UIEE, solving the LKL thread context switching issues discussed in **Challenge 2** (§IV-C).

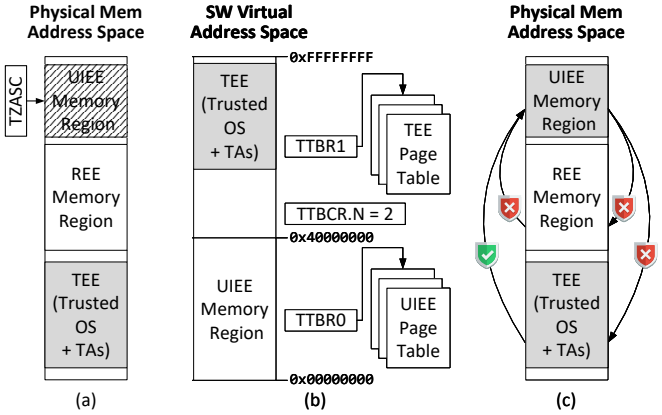

Fig. 3: **UIEE Memory Isolation Mechanism.** Fig. (a) describes the UIEE physical memory layout. Fig. (b) demonstrates the UIEE page table setup. Fig. (c) shows the final memory access state after memory isolation.

B. Memory Isolation

We leverage TZASC and dedicated page tables to realize UIEE memory isolation. Once the main thread enters TEE after invoking the REE-side TEE service APIs in order to initialize UIEE, the UIEE session manager first configures the UIEE memory pages as secure memory using TZASC, preventing further REE access, as shown in Figure 3-(a). We reconstruct UIEE page tables inside the trusted OS to isolate UIEE from TEE components and maintain a minimal TCB while preserving the memory mapping semantics established by the UIEE driver. To be specific, ARM specifies two ARM translation table base registers (TTBR) [42], namely `TTBR0` and `TTBR1`. Additionally, the translation table base control register (TTBCR) [42] determines the translation boundary address VA_b, where virtual addresses below VA_b are translated using the page table specified in `TTBR0` while those above using `TTBR1` which is used by the trusted OS by default. For ARMv7 platforms, we preserve the first 1GB of the TEE virtual address space for UIEE memory by setting up `TTBCR.N=2`, as shown in Figure 3-(b). Afterwards, we reconstruct a UIEE page table inside the trusted OS based on the original page table built by the REE UIEE driver and assign its physical address to `TTBR0`. During the new page table construction, the whole virtual address space layout is preserved in the newly-created page table so that the UIEE programs can directly run without further linking.

The final memory access permission is shown in Figure 3-(c). TZASC is used to configure the dynamically allocated UIEE memory pages as secure so that REE programs cannot access UIEE code or data. Additionally, we maintain separate page tables for UIEE memory regions and other TEE components respectively in order to protect the trusted OS as well as

TAs from illegal access issued from the UIEE components, maintaining a minimal TCB size. Finally, UIEE programs cannot access any REE memory including the Linux kernel and other REE processes once the isolation is enforced.

C. Thread Management

We investigate a UIEE threading model based on the existing LKL threading model discussed in §II-C. To begin with, we specify how a UIEE thread context is defined. Additionally, instead of re-implementing the thread management schemes inside the trusted OS, we choose to reuse the existing Linux kernel thread scheduling and synchronization subsystems by proposing two novel mechanisms namely the thread synchronization mechanism that enables the two-stage bootstrapping scheme (§IV-A) solving **Challenge 1** (§II-C) and the on-demand thread migration mechanism solving **Challenge 2** (§II-C).

1) UIEE-specific Thread Context: We design the UIEE-specific thread context based on the LKL threading model discussed in §II-C. To be specific, we augment the LKL thread context with UIEE specific information, as shown in Figure 4-(a). To begin with, we add a flag (`in_uiee`) inside the thread local storage (TLS) [43] to indicate whether the corresponding user-space threads' control flows are transferred into UIEE so that LKL can conduct thread migration for a thread whose control flow is still within REE. Additionally, we add several flags to the Linux kernel `task_struct` to represent whether a thread is a UIEE thread (`ts.is_uiee`), whether it finishes initialization (`is_synced`) (§IV-C2) and whether it is migrated to UIEE or not (`is_miged`), respectively. Finally, each UIEE thread is bound with a TEE TA session so as to access the trusted OS services, thus an additional flag (`ta.is_uiee`) is added for the trusted OS to distinguish it from normal TAs.

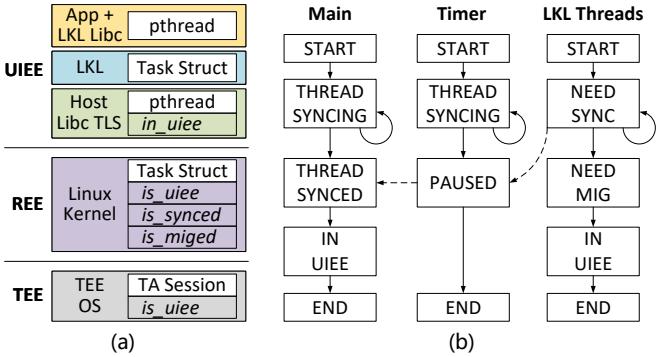

Fig. 4: **UIEE Threading Model.** Fig. (a) describes the thread context definition for a UIEE thread. Fig. (b) presents the thread state machine for the main, the timer and the LKL threads, respectively.

2) Thread Synchronization: The thread synchronization mechanism ensures that the timer is paused and all LKL threads finish initialization during the LKL initialization within the first-stage bootstrapping phase. We demonstrate the thread synchronization procedure using the thread state machine shown in Figure 4-(b). Following the LKL initialization process discussed in §II, the timer thread and multiple LKL threads are created after the main thread starts execution and they run concurrently, interleaving with each other. Meanwhile, the main thread blocks, waiting for all LKL threads finish initialization. The UIEE driver sets the corresponding `ts.is_uiee` flags and initializes each `is_synced` flag as *True* after all LKL threads are created during the LKL initialization phase. When an LKL thread is scheduled to run, its `is_synced` flag is set as *False*, indicating that this thread does not finish its initialization job. Additionally, the main thread conducts a periodic thread state examination, where it determines that the LKL threads have NOT finished execution if there is any *False* `is_synced` flag and in turn resets all `is_synced` flags back to *True*. Such procedure continues until all `is_synced` flags are *True* for multiple rounds to avoid concurrency issues. This mechanism enables the main thread to verify that all LKL kernel threads have completed initialization and entered sleeping mode waiting for their scheduling semaphores to be released, thereby establishing a synchronization point before the main thread can proceed with further execution. Finally, the UIEE driver's thread synchronizer sets all LKL kernel threads' their `is_miged` flags as *False* indicating that their control flows are still in REE and the main thread enters UIEE execution by invoking the REE-side TEE service API.

3) On-demand Thread Migration: We propose the UIEE thread migration mechanism to transfer the control flow of LKL threads from the REE to UIEE in an on-demand manner upon LKL context switching during the UIEE runtime phase. Recall that, after the two-stage bootstrapping produce, only the main thread switches to UIEE execution while all LKL threads are in sleeping mode waiting for their own scheduling semaphores to be released by another thread. We design a set of *libtee*-based semaphore APIs to redirect the semaphore releasing operations issued by LKL to the REE Linux kernel `futex` subsystem upon LKL context switching. Once the `futex` subsystem wakes up and schedules the LKL thread to run, we migrate the thread's context into TEE directly from the REE Linux kernel and resume its execution inside UIEE.

Consider the following scenario where the application within UIEE invokes `read()` from the main thread context to retrieve file contents and then the LKL Linux kernel assigns a kernel worker thread to conduct the block I/O operations. The detailed workflow is shown in Figure 5. From the very beginning, the worker thread is created during the LKL initialization phase. After completing initialization, it goes into sleeping mode, waiting for further service requests. Additionally, we create a per-thread jump buffer $JBuf_w$ inside the thread TLS in order to preserve the worker thread's REE context (Fig. 5, ❶). Then, after LKL initialization, the application starts execution within UIEE. Upon the file reading request, the main thread releases the worker thread's semaphore by invoking the semaphore releasing operation defined by *libtee*. Then, *libtee* retrieves the thread context from the TLS heap and checks

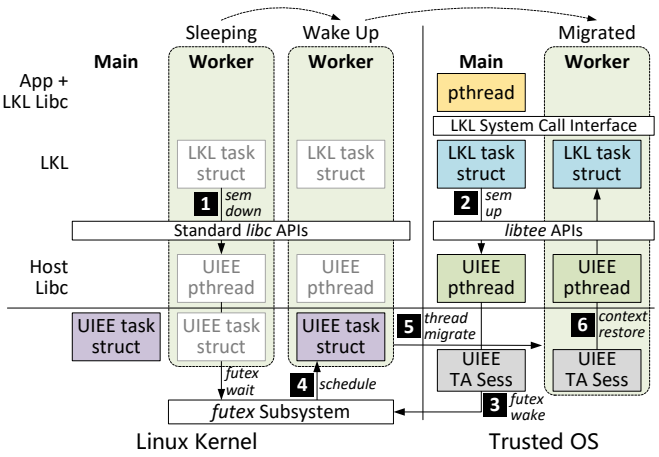

Fig. 5: **UIEE On-demand Thread Migration Mechanism.** Note that the UIEE memory isolation boundaries are removed for a clear workflow presentation.

whether its corresponding `in_uiee` flags is *True*. If not, the worker thread context is still in the REE Linux kernel and requires migration (Fig. 5, ❷). Afterwards, a `futex` wake RPC is issued by *libtee* to the Linux kernel to switch the worker thread's REE Linux kernel context to running mode (Fig. 5, ❸) and the Linux kernel scheduler would select it for execution in a subsequent time slice (Fig. 5, ❹). However, once the thread is scheduled to run, the control flow is meant to return to the user-space host *libc*, *i.e.* the UIEE loader, from the REE Linux kernel, which is in-accessible to REE after UIEE memory isolation. In order to transfer its control flow to UIEE, a hook function within the REE Linux kernel (§V) is invoked to inform the UIEE session manager of the thread ID. Additionally, the hook function sets its `is_miged` flag as *True*, indicating that the thread's context is migrated to UIEE. (Fig. 5, ❺). Finally, the UIEE session manager creates a new TA session with its `ta.is_uiee` flag as *True* for the worker thread and restores its execution context using $JBuf_w$ saved before (Fig. 5, ❻). The LKL thread context switching among the migrated threads is conducted entirely within the TEE using the trust OS services and all LKL kernel threads are migrated in such same manner upon the first invocation after the two-stage bootstrapping procedure.

V. Implementation

We implement a UIEE prototype based on the software components listed in Table II. In this section, we present implementation details on several UIEE components.

Trusted OS. We build the trusted OS based on the OPTEE OS 4.3.0 [44] and implement the UIEE session manager as an OPTEE pseudo-TA (PTA). Based on the existing OPTEE OS TA management framework, the UIEE session manager creates a new user-TA session for each UIEE thread once the thread transfers into TEE execution.

REE OS. We build the REE OS based on the Linux kernel 6.6.0 [45] by adding the UIEE thread migration hook inside `ret_to_user()` which will be triggered right before a UIEE thread returns to user-mode execution from the Linux kernel. Additionally, we implement a set of kernel-level TEE service APIs to invoke the UIEE session manager from the Linux kernel.

libtee. *libtee* implements the LKL host interface based on the existing user-TA libraries. Out of 40 LKL host interface operations[2], 15 are realized based on native TEE services, 2 are redirected to REE and the remaining operations are left as non-defined since they are only used in the first-stage bootstrapping scheme discussed in §IV-A and never called during the UIEE application runtime phase. As for the 2 redirected ones, 1 operation related to clock time retrieval while the other calls the REE Linux kernel `futex` subsystem triggering the thread migration process discussed in §IV-C3.

As for block I/O, UIEE features an in-memory block device implementation where an *ext4*-formatted disk image is loaded into the UIEE memory region and mounted to LKL during the initialization phase. Such design choice ensures that all block I/O operations are conducted within UIEE in order to provide secure block I/O for UIEE applications, and also avoids the potential performance cost of full disk encryption [27].

TABLE II: LoC Statistics of UIEE Components

Component	Version	Modifed	Added	Removed
REE OS	Linux 6.6.0	1	961	0
Trusted OS	OPTEE OS 4.3.0	26	1997	15
UIEE Driver	Customized LKM	0	532	0
UIEE Loader	musl libc 1.2.4	40	2633	53
LKL	Linux 4.14	89	679	95
LKL *libc*	musl libc 1.2.4	65	43	84
libtee	Customized TA Lib	0	1232	0
TEE Deamon	OPTEE *tee-supplicant*	7	340	85
Total LoC		**228**	**8417**	**332**

VI. Performance Evaluation

In this section, we first introduce the evaluation platform setup. Then, we conduct system benchmark experiments, micro benchmark experiments and application case studies to demonstrate the feasibility and performance of UIEE.

A. Platform Setup

We evaluate the current UIEE prototype on an NXP IMX6Q SABRE-SD evaluation board [46], a powerful evaluation platform equipped with 4 ARM Cortex-A9 cores and 1 GB DRAM. To ensure a consistent evaluation environment for all experiments, we only enable one CPU core and meanwhile turn off the CPU frequency scaling features so that the CPU can work stably in a pre-defined frequency of 792 MHz.

B. System Benchmark

We conduct two sets of system benchmarking experiments to evaluate the performance overhead that UIEE introduces to the vanilla REE and TEE environment. To be specific, we leverage the vanilla Linux kernel together with the original

[2]The UIEE LKL host interface operations are specified in Appendix C.

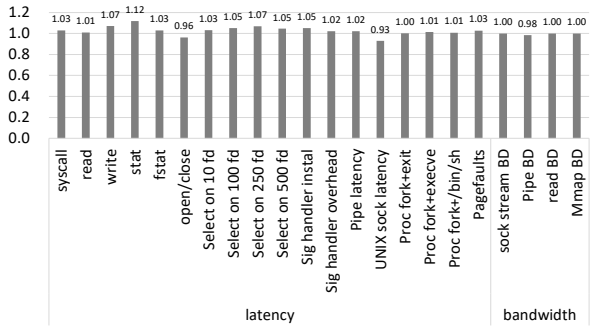

Fig. 6: Normalized LMbench Results.

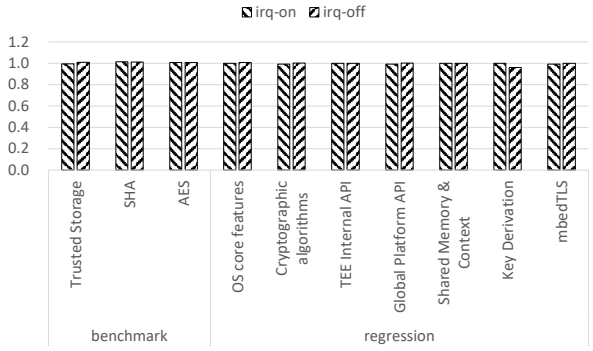

Fig. 7: Normalized OPTEE *xtest* Results.

OPTEE OS[3] as the baseline setting. Additionally, for all experiments, we build the corresponding OPTEE OS using the ARM instruction set [42] and disable non-aligned memory access features.

1) LMBench: For the REE system benchmark, we run the LMBench 3.0 [47] on both the baseline setting and UIEE. We conduct LMBench measurements for 10 rounds and take the average value as the final result. As shown in Figure 6, the results show that UIEE introduces an average 2.65% overhead in terms of the system call latency compared to the baseline setting. This is due to the `is_miged` flag checking conducted by the UIEE thread migration hook during each task context switching. Additionally, UIEE introduces negligible performance overhead for the bandwidth test cases.

2) OPTEE xtest: For the TEE system benchmark, we leverage the OPTEE official test suite namely *xtest* [48] to evaluate the performance (benchmark tests) and the functionality correctness (regression tests) of the UIEE trusted OS. We run *xtest* under both the baseline setting and the UIEE setting for 5 rounds, and used the average elapsed time as the final results. Both UIEE and the vanilla OPTEE OS pass the same test case sets, indicating that the UIEE trusted OS preserves all functionalities of the original OPTEE OS as intact. Additionally, as shown in Figure 7, UIEE introduces negligible performance overhead to normal TEE services with non-secure interrupts enabled or not.

[3]We conduct minor modification to port the original OPTEE OS to our evaluation platform.

C. Micro Benchmark

We conduct two sets of micro benchmarks to investigate the UIEE bootstrapping time as well as the system call latencies.

1) Bootstrapping Time: We measure the elapsed time of each UIEE booting step and compare them with the case with a vanilla LKL. As discussed in §IV-A, we divide the UIEE first-stage bootstrapping process into 6 steps, namely the UIEE memory region allocation & mapping step, the UIEE stack preparation step, the LKL initialization step, the LKL thread synchronization step, the LKL *libc* initialization step and finally the UIEE LKL host interface switching step. We measure the elapsed time of each step for 5 times and take the average value as the final results, as shown in Table III. The UIEE memory allocation & mapping step takes the most time of 1483.374 *ms* since the current UIEE prototype allocates and maps a whole physically continuous memory region of 88 MB for one time. For the following bootstrapping steps, the UIEE loader conducts dynamic memory allocation using such prepared memory region and thus no further host kernel service (such as `brk()` or `mmap()`) is invoked to map new heap regions. Such design leads to the result that the UIEE LKL initialization step takes 106.966 *ms* which is about 54 *ms* less than the time taken by the vanilla LKL and the same goes for the LKL *libc* initialization step. Additionally, the LKL thread synchronization process takes about 41 *ms* on average, incurring reasonably small extra bootstrapping time. The remaining two steps for UIEE stack preparation and LKL host interface switching take 0.108 *ms* in total, which is negligible compared to other steps.

TABLE III: Elapsed Time for Different UIEE Booting Steps

Bootstrapping Step	vanilla LKL	UIEE
UIEE Region Allocation & Mapping (ms)	-	1483.374
UIEE Stack Preparation (ms)	-	0.015
LKL Initialization (ms)	160.461	106.966
LKL Thread Synchronization (ms)	-	41.147
LKL *libc* Initialization (ms)	0.305	0.025
UIEE Host Interface Switching (ms)	-	0.093

2) UIEE System Call Latency: We measure various LKL system call (syscall) latencies inside UIEE and compare them with those of the original host Linux kernel and the vanilla LKL. Additionally, for UIEE, we conduct two sets of experiments with non-secure interrupts[4] enabled and disabled, respectively. For each case, we measure the elapsed time for 20,000 continuous syscalls for 3 rounds and take the average value as the final result. As shown in Figure 8, for the NULL syscall (*i.e.* `getpid()`), both LKL and UIEE take less time since LibOSes serve syscalls through direct function invocation which takes less time than the common OS syscall invocation mechanism using CPU exception. Meanwhile, for the `seek()` & `read()` case, both LKL and UIEE demonstrate smaller latencies because LKL (Linux kernel) stores file contents inside the kernel buffer cache after the first-time file reading and serves the following reading operations on the

[4]Mostly timer interrupts which are handled by the REE Linux kernel.

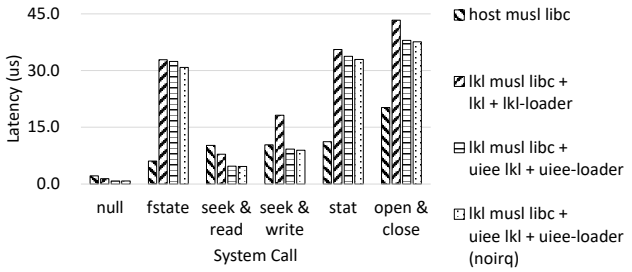

Fig. 8: System Call Latency Results.

same file directly using the buffered contents without further device I/O. Instead, for other syscalls involving file system access, LKL takes 74% - 440% more time than the original host Linux kernel since all such operations are realized by the underlying LKL host interface and in turn served by the host Linux kernel. Moreover, compared to the vanilla LKL, UIEE shows less syscall latencies ranging from -40% to -2% due to the neat implementation of *libtee*. Finally, turning off non-secure interrupts during the UIEE execution phase would result in a slightly less syscall latency for most cases.

D. Application Case Studies

To demonstrate the feasibility of UIEE, we run 8 real-world applications[5] inside UIEE and evaluate the corresponding performance overhead. All applications are directly retrieved from their corresponding online repositories and built using a GNU cross-compile toolchain with GCC 11.3.1 inside a Ubuntu 22.04 ARM64 container against a standard musl *libc* following the same development routine as any Linux application would go through. Note that except for minor logging code to record execution time, NO modification is made to any of these applications since the whole UIEE framework is completely transparent to the user-space applications. For each case study, we run the corresponding application under the 4 experiment settings 1) **baseline**: applications use the original musl *libc* they are built against; 2) **LKL**: applications use the LKL *libc* together with the LKL; 3) **UIEE**: applications run inside UIEE with non-secure interrupts enabled; 4) **UIEE-noirq**: applications run inside UIEE with non-secure interrupts disabled. Additionally, we run each experiment for 3 rounds and take the average value as the final result.

1) SQLite: We leverage the *speedtest1* benchmark shipped with SQLite [49] for the SQLite case study. *speedtest1* conducts totally 32 test cases involving various SQL-based database operations which can be categorized into 6 groups according to the corresponding test case codes and we measure the time needed to finish each test case group. As shown in Figure 9, the results show that LKL incurs an extra overhead of 4.23% compared with the baseline on average. Instead, UIEE achieves the same performance as the baseline setting, 4.08% performance boost compared to LKL. Such performance gain may result from the fact that UIEE programs run inside a physically continuous memory region (§IV-A) exhibiting more spatial locality, which favors memory-intensive applications like databases. Finally, turning off non-secure interrupts during the UIEE execution phase brings about an extra 2.07% performance gain on average. This is because UIEE applications are never interrupted by the timer expiration interrupts during the execution phase once non-secure interrupts are disable and in turn exhibits better performance.

2) FFmpeg: We leverage FFmpeg [50] to evaluate UIEE for multi-media data processing tasks by conducting video encoding with different video codec formats and resolutions. To be specific, three commonly-used video codec formats, namely `yuv420p`, `rbg24` and `gray`, are selected and two video clips of both 480p and 720p resolution are generated for each codec format. Once a video clip generation completes, the video encoding speed is recorded as the experimental results. As shown in Figure 10, for `yuv420p` and `gray` formats, both LKL and UIEE incurs little overhead compared to the baseline. Additionally, for the `rgb24` format, LKL outperforms the baseline by achieving 16.76% and 17.17% speed increase for 480p and 720p resolution respectively while UIEE achieves 11.07% speed increase on average compared to the baseline.

3) Polybench: We leverage Polybench [51] to evaluate UIEE for scientistic computing tasks. In total, six Polybench test cases are selected including data mining, linear algebra and stencils. For each test round, we run each test case for 10 times and record the average elapsed time as the final result for each round. As shown in Figure 11, LKL increases the execution time by 1.89% on average compared to the baseline. Instead, for data mining tasks, namely correlation and covariance, UIEE achieves better performance by reducing the execution time by 11.44% and another 1.15% execution time can be saved if non-secure interrupts are disabled. Additionally, for the other 4 tasks, UIEE introduces an average overhead of 1.50%, whereas UIEE-noirq demonstrates performance comparable to the baseline with negligible overhead.

4) OpenSSL: We utilize the OpenSSL [52] *speed* test suite to evaluate the performance of three cryptographic algorithms—MD5, SHA256, and AES. The processing throughput for various data block sizes is recorded as the benchmark result. For the MD5 test case, the LKL and UIEE settings introduce the highest runtime overhead compared to the other test cases, with average overheads of 7.29%, 6.50%, and 5.35% for LKL, UIEE, and UIEE-noirq, respectively. However, for larger data block sizes (8192 bytes), UIEE incurs only a minimal hashing speed penalty of 1.50% compared to the baseline, while UIEE-noirq achieves a performance improvement of 0.56% over the baseline. In the SHA256 and AES test cases, both LKL and UIEE exhibit negligible overhead, with average values of 1.73% and 1.43%, respectively. When non-secure interrupts are disabled, UIEE-noirq achieves a slight performance advantage of 0.49% compared to the baseline.

5) Machine Learning Applications: We run 4 machine learning models inside UIEE, namely SqueezeNet [53], LeNet-5 [54], DarkNet-Tiny [55] and DarkNet [55]. We measure the elapsed time for model deduction and use it as the experiment results. Since DarkNet models require a large memory region,

[5]Applications are specified in Appendix B.

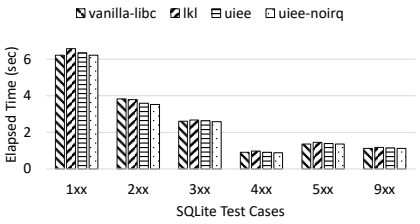

Fig. 9: SQLite *speedtest1* Results

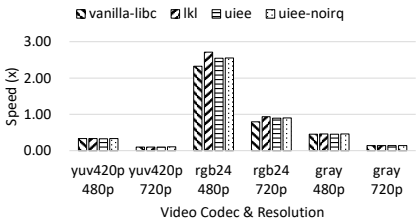

Fig. 10: FFmpeg Video Encoding Results

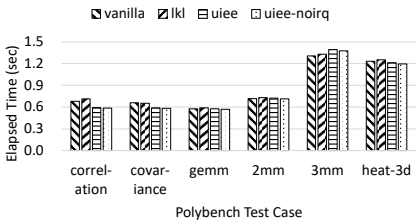

Fig. 11: Polybench Results

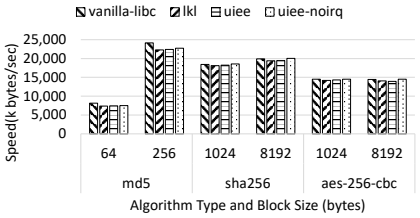

Fig. 12: OpenSSL Results

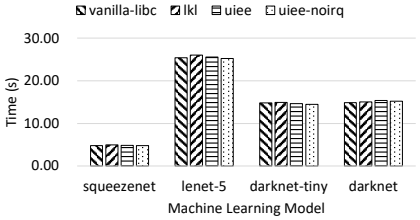

Fig. 13: Machine Learning Application Results

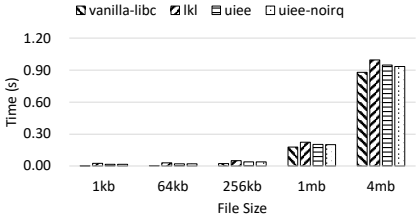

Fig. 14: *bzip2* File Compression Results

we reserve 180MB runtime memory for LKL when running the DarkNet models. As shown in Figure 13, for all four machine learning models, UIEE only incurs a negligible overhead of 0.98% on average compared with the baseline. Additionally, with non-secure interrupts disabled, UIEE can achieve a slight performance improvement of 0.42%.

6) bzip2: We leverage bzip2 [56] to evaluate UIEE for data compression tasks involving intense disk I/O. Specifically, we set up five experiment sets to compress files of sizes 1KB, 64KB, 256KB, 1MB, and 4MB. In each experiment, the file is read from the LKL disk, compressed, and the resulting compressed file is written back to the LKL disk, with the elapsed time serving as the experiment result. As shown in Figure 14, for files with small sizes (less that 1MB), UIEE exhibits high overhead (79% - 608%) compared with the baseline setting. Such overhead mainly results from the fact that each file operation has to go through two Linux kernel disk I/O layers, one from LKL and another from the host Linux kernel. For 4MB files, UIEE incurs a much smaller overhead of 7.72%, or 6.25% with non-secure interrupts disabled. Since UIEE leverages an in-memory disk setup to improve performance while also addressing disk-related security issues, UIEE achieves an average 20.17% performance improvement compared with LKL.

VII. Security Evaluation

In this section, we evaluate the UIEE security features by analyzing the TCB size increase and discussing how UIEE defends against various attacks.

A. TCB Size Evaluation

We investigate the TCB size increase caused by UIEE in terms of LoC and the trusted OS image size. The TEE system TCB consists of all TAs and the trusted OS which contains an embedded secure monitor. All components (§IV-A) inside the UIEE memory region, *i.e.* applications & libraries, LKL *libc*, LKL and *libtee*, are not considered as part of the TCB since they are isolated from the trusted OS and TAs by the UIEE memory isolation mechanism (§IV-B). The original OPTEE OS and TAs have a code base of 442,326 LoC which results in an trusted OS boot image of 547,260 bytes after compilation. As shown in Table II, the total modification to the trusted OS is of 2,038 LoC which is only 0.46% of the TCB LoC size. Additionally, the final UIEE boot image is 565,676 bytes, which is 3.37% larger than the original trusted OS image. Therefore, UIEE introduces a reasonable small TCB increase.

B. Security Analysis

The major security goal of UIEE is to protect the code integrity and data confidentiality of UIEE applications from REE attacks, regardless of privileged or unprivileged ones. Additionally, since the UIEE components have large code bases with potential vulnerabilities, we are intended to protect the trusted OS and TAs from possible attacks initiated from UIEE applications and LKL.

1) Protecting UIEE from REE Attacks: UIEE protects applications from REE attacks in both the initialization and the runtime execution phase. As investigated in §IV-A, all UIEE components are initialized inside REE during the system boot phase and each UIEE component can be initialized to a legal state since we leverage the secure boot mechanism in order to ensure a trusted REE state at boot time. After all UIEE components have been properly initialized, all REE components including the privileged Linux kernel are considered untrusted because they may be compromised by an attacker during the runtime execution phase. However, since the UIEE memory region is isolated before the REE components start execution (§IV-B), REE programs cannot access UIEE memory during the runtime phase. We conduct two experiments where an attacker tries to read/write the UIEE memory region from a REE user-space application and the

Linux kernel. Consequently, each attempted access triggers a system bus error and in turn both attacks fail.

UIEE also defends against attacks launched through UIEE operations that are redirected to REE. To be specific, there are totally 2 such operations including the thread migration process (§IV-C3) and clock time retrieval (§V). To begin with, the thread migration process wakes up a sleeping LKL thread and transfers its control flow to UIEE execution, during which all thread data reside inside the UIEE memory region which is in-accessible to REE. As for the clock time retrieval function, no confidential data is passed to the REE. Moreover, since the return values of these 2 operations do NOT involve any virtual address values and all memory management operations are performed entirely within UIEE (Appendix C), it is difficult for any potential REE attackers to launch Iago attacks [57]. Finally, any corrupted return value of these redirected operation would probably result in a UIEE panic, leading to availability issues which is considered out-of-scope while no confidential data is leaked.

2) Protecting TEE from Potential UIEE Attacks: An attacker can never access critical TEE data by compromising a UIEE program. Firstly, UIEE programs can only access the dedicated UIEE memory region allocated during the UIEE initialization phase and cannot directly access TEE memory region since we maintain a separate page table for UIEE programs (§IV-B). Additionally, any data exchange between UIEE programs and the trusted OS goes through sanity checking[6] so that the data addresses fall within the first 1GB virtual address space reserved for UIEE.

An attacker cannot break the UIEE memory isolation boundaries. Since UIEE is a user-space runtime environment and the UIEE page table is exclusively managed by the UIEE session manager inside the trusted OS, UIEE programs cannot map non-UIEE memory regions by manipulating the existing UIEE page table or switching to a malicious one.

Another concern is that an attacker may directly issue system calls from UIEE applications to the underlying trusted OS, bypassing the LKL host interface as well as *libtee*. Note that such threat is a common issue faced by all systems, not just UIEE. The mitigation to such attack is still an open research topic and various approaches have been proposed such as syscall filtering [58], binary rewriting [59], control flow integrity [60][61], syscall integrity [62], etc. We consider these works as complementary to UIEE, and plan to incorporate some mechanisms into UIEE in the future.

VIII. Discussion

We compare UIEE with two state-of-the-art research projects, namely TrustShadow & Shelter, in terms of performance, hardware compatibility and software flexibility.

A. Performance

Both TrustShadow and Shelter provide all application-required OS services by forwarding the syscalls issued from applications to the REE Linux kernel, which results in relatively high performance overhead due to frequent world switching. Nevertheless, UIEE provides OS services directly using a customized LibOS, avoiding heavy world-crossing events, and thus achieves nearly native or even better data processing efficiency compared with the vanilla REE system.

We first analyze the syscall handling routines of all three systems in order to theoretically assess the performance benefits of UIEE over TrustShadow and Shelter. Without losing generality, we take `mmap` as an example for a neat illustration. TrustShadow maintains two separate page tables in both the REE Linux kernel and the trusted OS respectively and reuses the Linux kernel memory management subsystem to handle memory mapping. As shown in Figure 15, for TrustShadow, when the application issues `mmap`, a forwarder within the trusted OS forwards such syscall to the REE Linux kernel that handles such request. The REE Linux kernel modifies the REE page table according to the memory mapping request, and such modification is checked by the trusted OS on `mmap` return. If the modification is legal, it will be updated to the shadow page table maintained by the trusted OS. Such routine involves 2 EL0-EL1 switches and 4 EL1-EL3 switches. Similarly, a `mmap` syscall issued from a Shelter application is first forwarded to the secure monitor by the Shelter driver and then to the REE Linux kernel. On `mmap` return, the secure monitor checks the page table modified by the Linux kernel and returns the final virtual address to the Shelter application through the Shelter driver. Such routine also involves 2 EL0-EL1 switches and 4 EL1-EL3 switches. As for UIEE, a `mmap` syscall can be issued in form of function call without any exception level switches, resulting in less performance overhead.

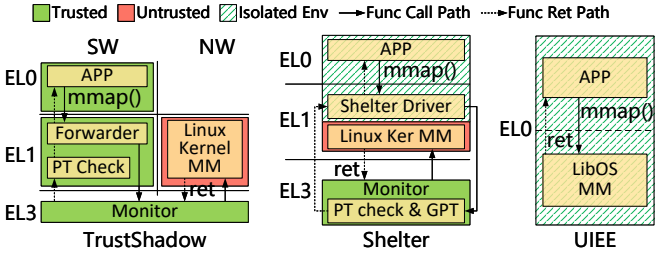

Fig. 15: TrustShadow, Shelter & UIEE Syscall Handling Routine

Since TrustShadow is close-sourced and Shelter leverages a totally different hardware setup, we conduct approximate simulation experiments so as to assess the performance overhead introduced by the syscall forwarding approach. Without losing generality, we run applications in REE and hook the `m(un)map` syscall handler within the REE Linux kernel by redirecting it to the trusted OS for page table checking in order to simulate the syscall forwarding approach investigated in TrustShadow and Shelter. Such simulated syscall handling routine also involves 2 EL0-EL1 switches and 4 EL1-EL3 switches. Note that all experiments only forward two syscalls (*i.e.*, `m(un)map`), which results in an optimized performance results for the syscall forwarding approach. More performance overhead would be foreseen if all syscalls are forwarded. As

[6]Detailed parameter checking between *libtee* and the trusted OS are elaborated in Appendix C.

shown in Table IV, the forwarding routine itself introduces 655% overhead to the `mmap` syscall latency for a single memory page and an extra 10% overhead is introduced for page table checking. With a larger memory size, the overhead introduced by forwarding routine decreases while more time is needed to check the page table since a larger memory region involves more page table entries. We use SQLite to demonstrate the performance impact of the syscall forwarding approach on real-world applications. As shown in Table V, this approach introduces the highest overhead of 21.24% in test case 2xx, which has the highest frequency of `m(un)map` syscall invocations (460.7 *syscalls/sec*). Additionally, the page table checking routine contributes little overhead of 1.04%.

TABLE IV: *mmap* Syscall Latency with Syscall Forwarding & Page Table Checking

Mem	vanilla-libc	forward		forward & PT check		uiee	
Size	Lat (μs)	Lat (μs)	overhead	Lat (μs)	overhead	Lat (μs)	overhead
4K	96.7	729.7	**654.83%**	739	**664.48%**	20.7	-78.62%
32K	229.0	909.7	**297.23%**	976	**326.20%**	113.3	-50.51%
128K	835.3	1503.0	**79.93%**	1766	**111.41%**	445.3	-46.69%
512K	3071.3	3735.0	21.61%	4768	**55.24%**	1751.7	-42.97%
2M	12197.3	12877.3	5.57%	17066	39.92%	6899.0	-43.44%

TABLE V: SQLite *speedtest1* Results with Syscall Forwarding & Page Table Checking

Test	syscall statistics		uiee	forward		forward & PT check	
Cases	mmap #	munmap #	time (s)	time (s)	overhead	time (s)	overhead
1xx	1062	985	6.356	6.956	**9.43%**	7.041	**10.77%**
2xx	828	827	3.592	4.355	**21.24%**	4.448	**23.84%**
3xx	100	100	2.633	2.676	1.63%	2.689	2.11%
4xx	44	41	0.906	0.961	**5.99%**	0.968	**6.80%**
5xx	145	140	1.388	1.448	4.27%	1.460	**5.14%**
9xx	6	4	1.146	1.124	-1.92%	1.126	-1.77%

B. Hardware compatibility

Shelter is built upon the brand new ARM CCA features which are currently not pervasively available for most embedded platforms while UIEE only leverages basic TrustZone memory isolation features namely TZASC and page tables to realize the isolated domain, achieving more hardware compatibility. Note that the UIEE designs are compatible with ARM CCA features in that the UIEE memory isolation schemes can be realized based on the ARM CCA GPC features.

C. Software flexibility

Considering that both TrustShadow and Shelter require that applications be statically linked with all their dependency libraries as well as *libc*, UIEE leverages the commonly-used dynamic linking approach and thus achieves more flexibility. Additionally, both TrustShadow and Shelter require implementing the forwarding routines for the redirected syscalls. Therefore, the set of realized syscall forwarding routines limits the scope of applications that can run inside TrustShadow and Shelter. In contrast, UIEE provides full Linux kernel syscall services and can in turn support most existing Linux applications without further engineering efforts.

IX. Related Work

A. TEE Library OSes

Various TEE-oriented library OSes are designed to provide a versatile application runtime inside TEEs. Haven [25] creates a runtime environment based on the Drawbridge LibOS inside SGX enclave to protect unmodified Windows applications. FlexOS [26] proposes a LibOS component isolation and hardening framework to efficiently compare security and performance trade-off among various configurations. Graphene-SGX [31] port the Graphene LibOS into Intel SGX with functionality improvements including integrity enforcement for libraries and secure multi-threading support. SGX-LKL [27] retrofits LKL into Intel SGX, featuring minimal, protected and oblivious host interfaces. Occlum [29] proposes a memory-safe, multi-process LibOS for Intel SGX that can directly run legacy applications with minimal source code modification. CubicleOS [28] realizes intra-LibOS component isolation using MPK and similar approaches [30] have been proposed to realize intra-unikernel user/kernel isolation and safe/unsafe kernel isolation using Intel MPK [63]. However, these LibOSes focus on x86 platforms and some specifically target SGX-based TEEs, making it difficult to port these LibOSes into TrustZone-based TEEs while UIEE proposes the first LibOS architecture for TrustZone-based embedded TEEs.

B. TrustZone-based TEEs

Multiple approaches have been proposed to enhance the capabilities of existing TrustZone-based TEEs. To mitigate board-level hardware attacks such as bus snooping [39], Ginseng [64] proposes a TEE running completely inside the ARM core registers. Additionally, Case [65] and CacheIEE [66] take similar design choices by running TEEs completely inside the ARM core caches. Komodo [8] and SecTEE [9] realize Intel SGX primitives based on TrustZone. SANCTUARY[13], OSP[10], PrivateZone[11] and TrustICE[12] propose TrustZone-based sandbox environments to protect customized applications from REE attacks while maintaining a minimal TCB. These projects propose their own unique software runtime and it takes a huge amount of engineering efforts to adapt existing applications to these systems. TLR [23] and WaTZ [24] propose TrustZone-based language runtime for .NET and webassembly respectively while the current embedded TEEs still favor C-like programming languages. TrustShadow [14] and Shelter [15] propose to run unmodified applications inside TEEs and forward most system calls to REE, which results in notable runtime performance overhead. 3rdParTEE [67] constructs an isolated runtime environment based on TZASC and TTBR to run third-party kernel modules in the trusted OS. However, 3rdParTEE requrires that the kernel modules are self-contained and do NOT use Linux kernel APIs, which simplifies its TEE runtime. Driverlets [68], StongBox [69], MyTEE [70] and LDR [71] investigate approaches to provide peripheral drivers support for trusted OSes.

X. Conclusion

We present the user-space isolated execution environment (UIEE), a comprehensive TrustZone-based IEE environment with *libc* support for data processing applications. UIEE creates an isolated memory region separated from both REE and TEE so that applications running inside UIEE can be protected from a malicious REE and we defend TEE against potential attacks issued from UIEE programs, maintaining a minimal TCB. Additionally, we are the first to propose a TrustZone-oriented LibOS architecture in order to provide a standard C runtime inside UIEE. We evaluate the feasibility of UIEE on an NXP IMX6QSABRESD board using various real-world applications and the experimental results show that UIEE introduces negligible overheads to real-world workloads. UIEE is available from: https://github.com/SparkYHY/UIEE.

Acknowledgment

We thank the anonymous reviewers for their constructive comments and suggestions. This research was supported in part by National Key R&D Program of China No. 2023YFC3605800, National Natural Science Foundation of China Grant Nos. 62232004 and 92467205, CCF-Huawei Populus Grove Fund, Jiangsu Provincial Key Laboratory of Network and Information Security Grant No. BM2003201, Key Laboratory of Computer Network and Information Integration of Ministry of Education of China Grant Nos. 93K-9, and Collaborative Innovation Center of Novel Software Technology and Industrialization. Any opinions, findings, conclusions, and recommendations in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] A. Ltd. (2024) Arm trustzone for cortex-a. [Online]. Available: https://www.arm.com/technologies/trustzone-for-cortex-a

[2] K. Ying, A. Ahlawat, B. Alsharifi, Y. Jiang, P. Thavai, and W. Du, "Truz-droid: Integrating trustzone with mobile operating system," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys*, 2018.

[3] K. Ying, P. Thavai, and W. Du, "Truz-view: Developing trustzone user interface for mobile OS using delegation integration model," in *Proceedings of the 9th ACM Conference on Data and Application Security and Privacy, CODASPY*, 2019.

[4] A. Ahlawat and W. Du, "Truzcall: secure voip calling on android using arm trustzone," in *2020 6th International Conference on Mobile And Secure Services, MobiSecServ*, 2020.

[5] 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 *Proceedings of the 18th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys*, 2020.

[6] M. S. Islam, M. Zamani, C. H. Kim, L. Khan, and K. W. Hamlen, "Confidential execution of deep learning inference at the untrusted edge with ARM trustzone," in *Proceedings of the 13th ACM Conference on Data and Application Security and Privacy, CODASPY*, 2023.

[7] A. A. Messaoud, S. B. Mokhtar, V. Nitu, and V. Schiavoni, "Shielding federated learning systems against inference attacks with ARM trustzone," in *Proceedings of the 23rd International Middleware Conference, Middleware*, 2022.

[8] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno, "Komodo: Using verification to disentangle secure-enclave hardware from software," in *Proceedings of the 26th Symposium on Operating Systems Principles, SOSP*, 2017.

[9] S. Zhao, Q. Zhang, Y. Qin, W. Feng, and D. Feng, "Sectee: A software-based approach to secure enclave architecture using TEE," in *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2019.

[10] 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 *Proceedings of the 2016 USENIX Annual Technical Conference, USENIX ATC*, 2016.

[11] J. S. 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. Dependable Secur. Comput., TDSC*, vol. 15, no. 5, pp. 797–810, 2018.

[12] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang, "Trustice: Hardware-assisted isolated computing environments on mobile devices," in *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN*, 2015.

[13] F. Brasser, D. Gens, P. Jauernig, A. Sadeghi, and E. Stapf, "SANCTUARY: arming trustzone with user-space enclaves," in *Proceedings of the 26th Annual Network and Distributed System Security Symposium, NDSS*, 2019.

[14] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger, "Trustshadow: Secure execution of unmodified applications with ARM trustzone," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys*, 2017.

[15] Y. Zhang, Y. Hu, Z. Ning, F. Zhang, X. Luo, H. Huang, S. Yan, and Z. He, "SHELTER: extending arm CCA with isolation in user space," in *Proceedingsof the 32nd USENIX Security Symposium, Security*, 2023.

[16] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptol. ePrint Arch.*, p. 86, 2016.

[17] I. Advanced Micro Devices. (2023) Amd secure encrypted virtualization (sev). [Online]. Available: https://www.amd.com/en/developer/sev.html#:~:text=Usesonekeypervirtual, guestoperatingsystemandhypervisor.

[18] A. Ltd. (2024) Arm confidential compute architecture. [Online]. Available: https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture

[19] ——. (2010) Corelink trustzone address space controller tzc-380 technical reference manual. [Online]. Available: https://developer.arm.com/documentation/ddi0431/c/introduction/about-the-tzasc

[20] A. Madhavapeddy, R. Mortier, C. Rotsos, D. J. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: library operating systems for the cloud," in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, 2013, pp. 461–472.

[21] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt, "Rethinking the library OS from the top down," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, 2011, pp. 291–304.

[22] O. Purdila, L. A. Grijincu, and N. Tapus, "Lkl: The linux kernel library," in *Proceedings of the 9th RoEduNet IEEE International Conference*, 2010.

[23] N. Santos, H. Raj, S. Saroiu, and A. Wolman, "Using ARM trustzone to build a trusted language runtime for mobile applications," in *Proceedings of the 19th Architectural Support for Programming Languages and Operating Systems, ASPLOS*, 2014.

[24] J. Ménétrey, M. Pasin, P. Felber, and V. Schiavoni, "Watz: A trusted wasm runtime environment with remote attestation for trustzone," in *Proceedings of the 42nd IEEE International Conference on Distributed Computing Systems, ICDCS*, 2022.

[25] A. Baumann, M. Peinado, and G. C. Hunt, "Shielding applications from an untrusted cloud with haven," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, 2014.

[26] H. Lefeuvre, V. Badoiu, A. Jung, S. L. Teodorescu, S. Rauch, F. Huici, C. Raiciu, and P. Olivier, "Flexos: towards flexible OS isolation," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, 2022.

[27] C. Priebe, D. Muthukumaran, J. Lind, H. Zhu, S. Cui, V. A. Sartakov, and P. R. Pietzuch, "SGX-LKL: securing the host OS interface for trusted execution," *CoRR*, vol. abs/1908.11143, 2019.

[28] V. A. Sartakov, L. Vilanova, and P. R. Pietzuch, "Cubicleos: a library OS with software componentisation for practical isolation," in *Proceedings*

of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS, 2021.
[29] Y. Shen, H. Tian, Y. Chen, K. Chen, R. Wang, Y. Xu, Y. Xia, and S. Yan, "Occlum: Secure and efficient multitasking inside a single enclave of intel SGX," in *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, 2020.
[30] M. Sung, P. Olivier, S. Lankes, and B. Ravindran, "Intra-unikernel isolation with intel memory protection keys," in *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE*, 2020.
[31] C. Tsai, D. E. Porter, and M. Vij, "Graphene-sgx: A practical library OS for unmodified applications on SGX," in *Proceedings of the 2017 USENIX Annual Technical Conference, USENIX ATC*, 2017.
[32] Linaro. (2024) Open portable trusted execution environment, optee. [Online]. Available: https://www.op-tee.org/
[33] S. Boutnaru. (2022) The linux process journey — pid 0 (swapper). [Online]. Available: https://medium.com/@boutnaru/the-linux-process-journey-pid-0-swapper-7868d1131316
[34] LKL. (2024) Lkl thread info definition. [Online]. Available: https://github.com/lkl/linux/blob/master/arch/lkl/include/asm/thread_info.h
[35] Z. Ling, H. Yan, X. Shao, J. Luo, Y. Xu, B. Pearson, and X. Fu, "Secure boot, trusted boot and remote attestation for ARM trustzone-based iot nodes," *J. Syst. Archit.*, vol. 119, p. 102240, 2021.
[36] N. Zhang, K. Sun, D. Shands, W. Lou, and Y. T. Hou, "Trusense: Information leakage from trustzone," in *Proceedings of the 37th IEEE Conference on Computer Communications, INFOCOM*, 2018.
[37] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *Proceedings of the 27th USENIX Security Symposium, USENIX Security*, 2018.
[38] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *Proceedings of the 40th IEEE Symposium on Security and Privacy, S&P*, 2019.
[39] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. B. Kang, "Vigilare: toward snoop-based kernel integrity monitor," in *Proceedings of the 19th ACM Conference on Computer and Communications Security, CCS*, 2012.
[40] A. Markuze, A. Morrison, and D. Tsafrir, "True IOMMU protection from DMA attacks: When copy is faster than zero copy," in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, 2016.
[41] 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.
[42] A. Ltd. (2011) Arm architecture reference manual armv7-a and armv7-r edition. [Online]. Available: https://developer.arm.com/documentation/ddi0406/c/
[43] GNU. (2024) Thread local storage. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Thread-Local.html
[44] Linaro. (2024) Optee source code repository. [Online]. Available: https://github.com/OP-TEE/optee_os
[45] L. Torvalds. (2024) Linux kernel source code repository. [Online]. Available: https://github.com/torvalds/linux
[46] NXP. (2023) Nxp imx6q sabre-sd development board. [Online]. Available: https://www.nxp.com/design/development-boards/i-mx-evaluation-and-development-boards/sabre-board-for-smart-devices-based-on-the-\\i-mx-6quad-applications-processors:RD-IMX6Q-SABRE
[47] intel. (2018) lmbench 3.0. [Online]. Available: https://github.com/intel/lmbench
[48] Linaro. (2019) Imx optee test. [Online]. Available: https://github.com/nxp-imx/imx-optee-test/tree/imx_4.14.98_2.3.0
[49] Sqlite source code repository. [Online]. Available: https://github.com/sqlite/sqlite.git
[50] Ffmpeg source code repository. [Online]. Available: https://github.com/FFmpeg/FFmpeg.git
[51] Polybench source code repository. [Online]. Available: https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1
[52] Openssl source code repository. [Online]. Available: https://github.com/openssl/openssl.git
[53] Squeezenet source code repository. [Online]. Available: https://github.com/royliuyu/squeezenet
[54] Lenet-5 source code repository. [Online]. Available: https://github.com/fan-wenjie/LeNet-5
[55] Darknet source code repository. [Online]. Available: https://github.com/pjreddie/darknet
[56] bzip2 source code repository. [Online]. Available: https://github.com/libarchive/bzip2
[57] S. Checkoway and H. Shacham, "Iago attacks: why the system call API is a bad untrusted RPC interface," in *Proceedinsg of the 2013 Architectural Support for Programming Languages and Operating Systems, ASPLOS*, 2013.
[58] Security-enhanced linux. [Online]. Available: https://en.wikipedia.org/wiki/Security-Enhanced_Linux#:~:text=Security%2DEnhanced%20Linux%20(SELinux),mandatory%20access%20controls%20(MAC)
[59] M. Wenzl, G. Merzdovnik, J. Ullrich, and E. R. Weippl, "From hack to elaborate technique - A survey on binary rewriting," *ACM Comput. Surv.*, vol. 52, no. 3, pp. 49:1–49:37, 2019.
[60] J. Li, X. Tong, F. Zhang, and J. Ma, "Fine-cfi: Fine-grained control-flow integrity for operating system kernels," *IEEE Trans. Inf. Forensics Secur., TIFS*, vol. 13, no. 6, pp. 1535–1550, 2018.
[61] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," *ACM Comput. Surv.*, vol. 50, no. 1, pp. 16:1–16:33, 2017.
[62] C. Jelesnianski, M. Ismail, Y. Jang, D. Williams, and C. Min, "Protect the system call, protect (most of) the world with BASTION," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, 2023.
[63] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, "libmpk: Software abstraction for intel memory protection keys (intel MPK)," in *Proceedings of the 2019 USENIX Annual Technical Conference, USENIX ATC*, 2019.
[64] M. H. Yun and L. Zhong, "Ginseng: Keeping secrets in registers when you distrust the operating system," in *Proceedings of the 26th Annual Network and Distributed System Security Symposium, NDSS*, 2019.
[65] N. Zhang, K. Sun, W. Lou, and Y. T. Hou, "Case: Cache-assisted secure execution on ARM processors," in *Proceedings of the 36th IEEE Symposium on Security and Privacy, SP*, 2016.
[66] J. Wang, K. Sun, L. Lei, Y. Wang, J. Jing, S. Wan, and Q. Li, "Cacheiee: Cache-assisted isolated execution environment on ARM multi-core platforms," *IEEE Trans. Dependable Secur. Comput., TDSC*, vol. 21, no. 1, pp. 254–269, 2024.
[67] J. Jang and B. B. Kang, "3rdpartee: Securing third-party iot services using the trusted execution environment," *IEEE Internet Things J.*, vol. 9, no. 17, pp. 15 814–15 826, 2022.
[68] L. Guo and F. X. Lin, "Minimum viable device drivers for arm trustzone," *Proceedings of the 17th European Conference on Computer Systems, EuroSys*, 2022.
[69] Y. Deng, C. Wang, S. Yu, S. Liu, Z. Ning, K. Leach, J. Li, S. Yan, Z. He, J. Cao, and F. Zhang, "Strongbox: A GPU TEE on arm endpoints," in *Proceedings of the 29th ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2022.
[70] S. Han and J. Jang, "Mytee: Own the trusted execution environment on embedded devices," in *Proceedings of the 30th Annual Network and Distributed System Security Symposium, NDSS*, 2023.
[71] H. Yan, Z. Ling, H. Li, L. Luo, X. Shao, K. Dong, P. Jiang, M. Yang, J. Luo, and X. Fu, "Ldr: Secure and efficient linux driver runtime for embedded tee systems," in *Proceedings of the 31st Annual Network and Distributed System Security Symposium, NDSS*, 2024.
[72] Toradex. (2024) Contiguous memory allocator - cma (linux). [Online]. Available: https://developer.toradex.com/software/linux-resources/linux-features/contiguous-memory-allocator-cma-linux/
[73] GNU. (2024) Auxiliary vector. [Online]. Available: https://www.gnu.org/software/libc/manual/html_node/Auxiliary-Vector.html

APPENDIX A
UIEE MEMORY REGION LAYOUT & INITIALIZATION

The UIEE driver inside the Linux kernel is responsible for allocating and mapping a user-space memory region for UIEE. Upon system startup, the UIEE driver allocates a set of continuous memory pages using the Linux kernel continuous memory allocator (CMA) [72]. Then, it maps these physical

memory pages into three virtual address space regions, as shown in Figure 16.

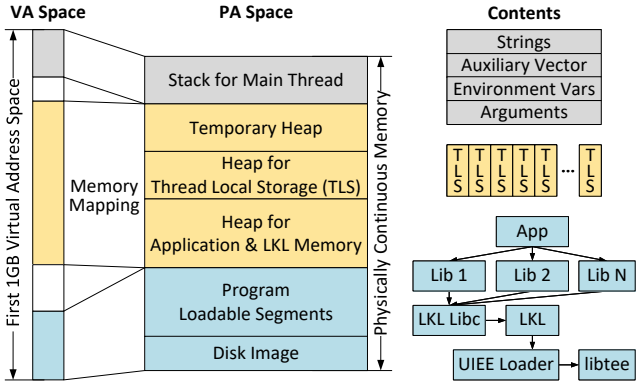

Fig. 16: UIEE Memory Layout.

Stack Region is used as the runtime stack for the application. Additionally, since a standard C program retrieves arguments from the stack using the `argv` pointer of the `main()` function, the UIEE loader resolves the original stack prepared by the Linux kernel during the process startup phase and reconstructs the stack contents on the UIEE stack accordingly, including application command line arguments, environment variables and the auxiliary vector [73]. Moreover, entries of the auxiliary vector also need to be calibrated if they contain pointers with non-UIEE virtual addresses.

Heap Region. We have modified the UIEE loader's memory allocator so that any newly-created objects reside in the UIEE memory region in order to protect UIEE data after isolation. The heap region consists of three sub-regions, namely an initial heap region, a thread local storage (TLS) [43] region and an LKL memory region. The initial heap region is a temporary one used for dynamic memory allocation during the UIEE bootstrapping phases, which can be released after UIEE bootstrapping. Additionally, we reserve a dedicated heap for the TLS of each LKL kernel thread which contain vital thread state information including thread-specific stacks, *pthread* objects, etc. Such design choice makes it convenient for the TEE UIEE session manager to retrieve and maintain the thread states, simplifying the LKL thread management within UIEE. Finally, the LKL memory region is used as the application and LKL runtime memory. The corresponding memory address and size are passed to LKL in form of a kernel boot command during the LKL bootstrapping phase and LKL manages such memory region as a normal Linux kernel does.

Program & Disk Region is used to store the ELF loadable segments of application, LKL *libc*, LKL and *libtee* as well as an optional LKL disk image. Since we load several ELFs that contain multiple global symbols with the same name, especially two *libc*s *i.e.* the UIEE loader and the LKL *libc*, an external symbol may be linked to a wrong definition resulting in a corrupted program state. For example, LKL's call to the `read()` function may be incorrectly linked to the LKL *libc*, creating an endless file reading loop. To solve such dilemma, we maintain a partial ordering relationship of [app+libs, LKL *libc*, LKL, UIEE loader, *libtee*] during ELF segment relocation. As for symbol resolution, the UIEE loader starts from the next ELF on the partial ordering chain and tries to find the very first symbol definition among the ELFs behind.

APPENDIX B
UIEE PERFORMANCE EVALUATION APPLICATION SPECIFICATION

The applications used for UIEE performance evaluation are presented in Table VI.

TABLE VI: UIEE Applications

Application	Description
SQLite 3.48.0 [49]	*SQLite* is a light-weight SQL database extensively used in embedded systems where minimal configuration and resource usage are crucial.
FFmpeg 5.0 [50]	*FFmpeg* is a powerful open-source multimedia framework for handling video, audio, and other multimedia files and streams.
PolyBench 4.2.1 [51]	*PolyBench* is a benchmarking suite with various scientific computation algorithms including linear algebra, data mining, and solvers, etc.
OpenSSL 1.1.0 [52]	*OpenSSL* is a widely-used open-source library for SSL and TLS protocols with support for all popular cryptographic algorithms.
SqueezeNet [53]	*SqueezeNet* is a lightweight deep learning model designed for image classification tasks.
LeNet-5 [54]	*LeNet-5* is a convolutional neural network (CNN) architecture for handwritten digit recognition using the MNIST dataset.
DarkNet [55]	*DarkNet* is an open source neural network framework written in C and CUDA that supports both CPU and GPU computation.
bzip2 1.0.8 [56]	*bzip2* is a high-performance data compression program that uses the Burrows-Wheeler transform (BWT) and Huffman coding to compress files.

APPENDIX C
UIEE LKL HOST INTERFACE SPECIFICATION

Table VII presents the UIEE LKL host interface. The total number of LKL host interface functions is 40, of which 36 are inherently defined by the original LKL and the remaining 4 are newly added for UIEE. With the UIEE two-stage bootstrapping scheme (§IV-A), *libtee* only needs to implement 17 LKL host interface functions instead of all 40 of them saving a huge amount of engineering efforts. The realized LKL host interface functions depend on 6 trusted OS syscalls, among which 3 are originally provided by the trusted OS while the other 3 are newly added for UIEE execution.

libtee is the only legal interface through which LKL can access trusted OS services and UIEE conducts the following sanity checking on the trusted OS syscalls invoked by *libtee*. Note that the sanity checking is meant to protect the trusted OS from potential attacks issued from UIEE applications.

- `log` takes an un-formatted string as well as its length, and the trusted OS ensures that the whole string resides in the UIEE memory region. Additionally, since the passed string is un-formatted, the attacker cannot launch format string attacks.
- `panic` immediately shuts down UIEE execution including all UIEE threads and thus no harm would be done to the trusted OS.

- `futex` initiates the thread migration process discussed in §IV-C3. It redirects the fast mutex call to the REE Linux kernel without (de)referencing the passed arguments.
- `wait_queue` takes a UIEE semaphore/mutex address and uses it as an index to find the corresponding wait queue to conduct synchronization. The syscall ensures that the passed address lays within the UIEE memory region.
- `get_tp` returns the user TA thread ID of the calling thread and it is hard to exploit such syscall considering its simple semantics.
- `get_time` is to retrieve the wall-clock time from the REE Linux kernel and such interface is hard to exploit considering its simple semantics.

TABLE VII: UIEE LKL Host Interface Specification

#	Category	LKL Host Interface Prototype	Origin	Type	Trusted OS Syscall
1	logging	`void print(const char *str, int len);`	LKL	Native	`log`
2		`int vsnprintf(char *, size_t, const char *, va_list);`	UIEE	Native	-
3	error	`void panic(void);`	LKL	Native	`panic`
4	semaphore	`struct lkl_sem* sem_alloc(int count);`	LKL	-	-
5		`void sem_free(struct lkl_sem *);`	LKL	-	-
6		`void sem_up(struct lkl_sem *);`	LKL	RPC	`futex*`
				Native	`wait_queue*`
7		`void sem_down(struct lkl_sem *);`	LKL	Native	`wait_queue*`
8		`void sem_update_tid(struct lkl_sem *);`	UIEE	-	-
9	mutex	`struct lkl_mutex *mutex_alloc(int recursive);`	LKL	-	-
10		`void mutex_free(struct lkl_mutex *);`	LKL	-	-
11		`void mutex_lock(struct lkl_mutex *);`	LKL	Native	`wait_queue*`
12		`void mutex_unlock(struct lkl_mutex *);`	LKL	Native	`wait_queue*`
13	threading	`lkl_thread_t thread_create(void (*f)(void *), void *arg);`	LKL	-	-
14		`void thread_detach(void);`	LKL	-	-
15		`void thread_exit(void);`	LKL	-	-
16		`int thread_join(lkl_thread_t tid);`	LKL	-	-
17		`int thread_equal(lkl_thread_t, lkl_thread_t);`	LKL	Native	-
18		`lkl_thread_t thread_self(void);`	LKL	Native	`get_tp*`
19		`long gettid(void);`	LKL	-	-
20		`struct lkl_tls_key *tls_alloc(void (*destructor)(void *));`	LKL	-	-
21		`void tls_free(struct lkl_tls_key *);`	LKL	-	-
22		`int tls_set(struct lkl_tls_key *, void *data);`	LKL	-	-
23		`void *tls_get(struct lkl_tls_key *);`	LKL	-	-
24		`void mark_in_uiee(void *);`	UIEE	Native	-
25		`void idle_loop_callback(void);`	LKL	-	-
26	memory management	`void* mem_alloc(unsigned long size);`	LKL	-	-
27		`void mem_free(void *);`	LKL	-	-
28		`void* boot_mem_alloc(unsigned long);`	UIEE	-	-
29	timing	`void* timer_alloc(void (*fn)(void *), void *arg);`	LKL	-	-
30		`int timer_set_oneshot(void *timer, unsigned long delta);`	LKL	-	-
31		`void timer_free(void *timer);`	LKL	-	-
32		`unsigned long long time(void);`	LKL	RPC	`get_time`
33	non-local jump	`void jmp_buf_set(struct lkl_jmp_buf *, void (*f)(void));`	LKL	Native	-
34		`void jmp_buf_longjmp(struct lkl_jmp_buf *, int ret);`	LKL	Native	-
35		`int jmp_buf_set_raw(struct lkl_jmp_buf *);`	LKL	Native	-
36	I/O setup	`void* ioremap(long addr, int size);`	LKL	-	-
37		`int iomem_access(void *addr, void *val, int size, int write);`	LKL	-	-
38	block I/O	`int get_capacity(struct lkl_disk, unsigned long long *res);`	LKL	Native	-
39		`int block_read(int fd, size_t offset, size_t len, void *buf);`	LKL	Native	-
40		`int block_write(int fd, size_t offset, size_t len, void *buf);`	LKL	Native	-

Origin indicates whether an LKL host interface is defined by the original LKL. "*LKL*": such interface is inherently defined by LKL. "*UIEE*": such interface is newly added for UIEE. **Type** indicates the implementation status of an LKL host interface inside *libtee*. "*Native*": such interface is implemented based on *libtee* and the trusted OS. "*RPC*": such interface is redirected to the REE and handled by the Linux kernel or other REE user-space threads. Others left blank ("-") are not implemented by *libtee*. A **trusted OS syscall** marked with a suffix "*" indicates that the corresponding syscall is newly added for UIEE while others without "*" are inherent OPTEE OS syscalls.