TZmediator: Secure Cooperative Execution of Cloud Applications Using POSIX APIs on Arm TrustZone

Taiyo Sato
Kyushu Institute of Technology
taiyo@ksl.ci.kyutech.ac.jp

Kenichi Kourai
Kyushu Institute of Technology
kourai@csn.kyutech.ac.jp

***Abstract*—Edge computing enables clouds to execute their applications closer to end-users. Even on untrusted edge devices, Arm TrustZone can securely run cloud applications in the secure world as trusted applications (TAs). Since the secure world has too high privileges, vulnerabilities in cloud applications could lead to the compromise of the entire system. To minimize the attack surface, it is desirable to run security-insensitive tasks in the normal world as unprivileged client applications (CAs). However, cooperation between CAs and TAs requires the use of the APIs specialized for TrustZone, which prevents flexible interactions. This paper proposes TZmediator to enable cooperative execution using standard POSIX APIs for a cloud application partitioned into two worlds. For inter-world seamless communications, TZmediator creates a shadow process in the normal world for each TA and delegates the invocation of POSIX APIs by a TA to the shadow process. Currently, it supports the message passing APIs, the shared memory API, and the signal API. If users require stronger isolation, TZmediator can execute security-sensitive tasks in TAs using WebAssembly. We have implemented TZmediator on OP-TEE and examined communication performance between CAs and TAs.**

1. Introduction

Recently, edge computing has become increasingly prevalent as a means to improve service quality by executing cloud applications closer to end-users [1]. On edge devices, even the operating system (OS) may not be trustworthy [2]. To run cloud applications on untrusted edge devices securely, trusted execution environments (TEEs) provided by recent processors can be used. A TEE isolates its memory from the rest of the system including the OS and prevents attackers from accessing the code and data used in it. If the memory of a TEE is encrypted, attackers cannot eavesdrop on it even with physical means. Major TEE architectures include Intel SGX [3], Arm TrustZone [4], and RISC-V Keystone [5] for applications and AMD SEV [6], Intel TDX [7], and Arm CCA [8] for virtual machines.

Arm TrustZone is a TEE architecture designed for untrusted edge devices. It provides two distinct worlds: the secure and normal worlds. The secure world is used to run trusted applications (TAs), which can securely execute even programs that handle sensitive information. In contrast, the normal world is used to run client applications (CAs), which execute programs that do not handle sensitive information. Although it is possible to execute the entire cloud application in the secure world, this increases the attack surface. Since the secure world has too high privileges, vulnerabilities in cloud applications could lead to the compromise of the entire system [9]. Therefore, it is desirable to minimize the code executed as TAs in the secure world and execute most pairs of the cloud application as CAs. However, the cooperative execution of CAs and TAs requires the use of the GlobalPlatform APIs ([10], [11]) defined for TEEs, which makes flexible interaction between them difficult.

This paper proposes TZmediator to enable cooperative execution using the standard Portable Operating System Interface (POSIX) APIs [12] for a cloud application partitioned into two worlds. TZmediator executes only the security-sensitive tasks as TAs in the secure world, while it runs the remaining tasks as CAs in the normal world. To enable inter-world seamless communication via POSIX APIs, TZmediator creates a shadow process corresponding to each TA in the normal world. When a TA invokes POSIX APIs, it delegates the invocation to the shadow process. Then, the shadow process invokes the standard library on behalf of the TA. If users require stronger isolation in the secure world, TZmediator can run WebAssembly applications in TAs [13] and enables them to communicate with CAs via the extended WebAssembly System Interface (WASI) [14].

We have implemented TZmediator on OP-TEE [15] and developed TZm libraries for TAs and CAs. The TZm-TA library provides POSIX APIs to TAs to enable communication with CAs. It delegates invoked POSIX APIs to a shadow process using a lightweight remote procedure call (RPC) or a polling-based method. Currently, it supports the message passing APIs such as pipes and sockets, the shared memory API, and the signal API. The TZm-CA library provides the `posix_spawn` function to CAs to seamlessly spawn TAs and create shadow processes. We conducted experiments to examine communication performance between CAs and TAs using TZmediator. Compared with communication inside the normal world, TZmediator could achieve comparable performance in most of the POSIX APIs. In addition, we confirmed that a real-world cloud application could run with negligible overhead using POSIX APIs.

The organization of this paper is as follows. Section 2 describes issues in securely executing cloud applications on untrusted edge devices using Arm TrustZone. Section 3 proposes TZmediator for the cooperative execution of a cloud application across two worlds using POSIX APIs, and Section 4 explains its implementation. Section 5 presents the security analysis. Section 6 shows experiments to examine communication performance using TZmediator. Section 7 describes related work, and Section 8 concludes the paper.

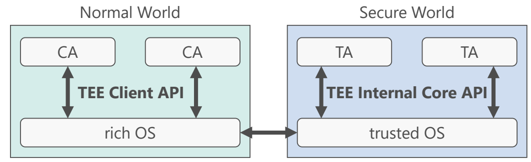


Figure 1. The system architecture of TrustZone.

2. Cloud Applications on Edge Devices

Edge computing executes cloud applications closer to end-users, e.g., edge devices such as Internet of Things (IoT) devices [1]. Thus, it enables lower latency, higher throughput, and real-time processing, compared with centralized cloud computing. It also reduces the amount of communication with clouds and thereby decreases network load. However, edge devices may not be trustworthy because they are outside the control of clouds [2]. If they are not well-maintained, they are vulnerable to attacks such as leakage of sensitive information, tampering with applications, and reverse engineering. Due to limited resources, it is difficult to implement strong security mechanisms on typical edge devices.

To securely execute cloud applications on untrusted edge devices, one promising approach is to use trusted execution environments (TEEs) provided by recent processors. A TEE enables program execution inside a logically isolated memory region. The system outside the TEE cannot eavesdrop on or tamper with the code and data used in the TEE. If the isolated memory of a TEE is encrypted by processors, attackers cannot perform physical memory snooping. Examples of major TEE architectures are Intel SGX [3], Arm TrustZone [4], and RISC-V Keystone [5] for applications and AMD SEV [6], Intel TDX [7], and Arm CCA [8] for virtual machines. By leveraging TEEs, cloud applications can be protected in edge devices even if the OS or the hypervisor is compromised.

Arm TrustZone is a TEE that is often used for edge devices. It provides two isolated execution environments called worlds, as depicted in Fig. 1. The secure world serves as a TEE and enables the secure execution of applications known as trusted applications (TAs). A trusted OS such as OP-TEE [15], Open-TEE [16], and Trusty [17] also runs inside the secure world to support the execution of TAs. In contrast, the normal world serves as a traditional execution environment, which is also referred to as a rich execution environment (REE). It executes not only traditional applications but also client applications (CAs), which are used to interact with TAs. A rich OS such as Linux runs in the normal world to support these applications. The monitor running below these OSes manages switching between the two worlds. Since it separates system resources strictly, the normal world cannot access resources reserved for the secure world.

However, executing cloud applications in the secure world raises two issues. One is that the secure world has much higher privileges than the normal world. Since cloud applications running on edge devices are often more complex than traditional TAs, they are more likely to contain vulnerabilities. If attackers who have intruded into the normal world exploit such vulnerabilities, they could potentially escalate their privileges and take control of the secure world. As a result, attackers could eavesdrop on and tamper with the entire memory of both worlds and access all the devices. In general, cloud applications do not need such high privileges but rather require strong isolation from the normal world. The other issue is that multiple TAs run in the secure world, i.e., a single execution environment. Compromised cloud applications could potentially attack the other TAs and extract confidential information held by them.

To enhance the security of cloud applications, it is desirable to minimize the code executed as TAs in the secure world like DarkneTZ [18]. In this design, TAs contain only sensitive information and execute only the code handling it. CAs execute the other security-insensitive tasks and communicate with TAs when necessary. In TrustZone, however, cooperative execution between CAs and TAs requires the use of the special APIs defined by GlobalPlatform, specifically, the TEE Client API [10] provided to CAs and the TEE Internal Core API [11] provided to TAs. Although these APIs are standardized, they significantly differ from POSIX APIs such as pipes and sockets, which are commonly used in rich OSes. For example, a TA is executed only in response to a command invocation by a CA. The CA must wait until the execution of the TA completes. As a result, flexible cooperation between CAs and TAs is difficult. This makes the development of cloud applications partitioned into two worlds more challenging.

The threat model in this paper is as follows. First, we assume that the hardware components providing TrustZone functionality, as well as the secure monitor, are free of vulnerabilities and that the protection mechanisms offered by TrustZone are not compromised. We trust software running in the secure world but consider not only TAs without vulnerabilities but also those with vulnerabilities as parts of cloud applications. In contrast, we do not trust software running in the normal world, including the rich OS. Therefore, we assume that CAs as parts of cloud applications could be compromised. We focus on attackers who reside in the normal world and attempt to eavesdrop on or tamper with sensitive information handled by cloud applications. Other types of attacks, such as denial-of-service (DoS) attacks and physical attacks, are out of scope.

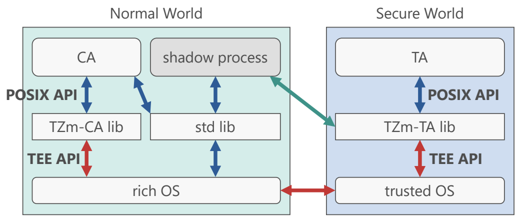


Figure 2. The system architecture of TZmediator.

3. TZmediator

This paper proposes TZmediator to enable cooperative execution using POSIX APIs for a cloud application partitioned into the two worlds of TrustZone. TZmediator executes only security-sensitive tasks as TAs in the secure world and the remaining tasks as CAs in the normal world. To allow inter-world seamless communication between CAs and TAs using POSIX APIs, TZmediator creates a shadow process in the normal world for each TA. The system architecture of TZmediator is illustrated in Fig. 2. The shadow process uses the standard library available in the normal world and issues system calls to the rich OS. Since the semantics of the library and the system calls are equivalent to those used by CAs, high compatibility with POSIX is preserved in communication between CAs and TAs. Examples of POSIX APIs for inter-process communication are the message passing APIs using pipes and sockets, the shared memory API, the signal API, and so on.

To emulate POSIX APIs, TZmediator provides the TZm-TA library to TAs. This library provides the functions for communication in POSIX APIs, e.g., `pipe` and `socket`. When a TA invokes the provided function, the library communicates with the corresponding shadow process in the normal world. The shadow process executes the specified POSIX API on behalf of the TA using the standard library and returns the result to the TA. On the other hand, CAs use POSIX APIs via the standard library provided by the rich OS. Instead of direct communication with TAs running in a different world, they first communicate with the shadow process corresponding to the target TA in the same world. Then, the shadow process communicates with the TA across the worlds.

As a concrete example, let us consider the execution flow when a TA communicates with a CA using a pipe. When the TA invokes the `write` function and writes data to the pipe, the TZm-TA library sends a write request to the shadow process. Then, the shadow process writes the data to the pipe. When the CA invokes the `read` function to read data from the pipe, it obtains data that the TA has written to the pipe via the shadow process. Conversely, when the TA invokes the `read` function to read data from the pipe, the TZm-TA library sends a read request to the shadow process. Then, the shadow process reads data from the pipe and forwards it to the TA. If there is no data in the pipe, the shadow process waits for the CA to write data to the pipe by invoking the `write` function.

Furthermore, TZmediator treats CAs and TAs as normal processes that do not use the GlobalPlatform APIs. This allows users to develop a cloud application as a set of normal processes. For this purpose, TZmediator provides the TZm-CA library to CAs. A CA spawns a new TA using the `posix_spawn` function provided by this library. This function hides the complex procedure of handling a TA with the TEE Client API from the CA. It executes the command of the TA in parallel to the CA by creating a subthread. Similarly, the TZm-TA library provided to TAs hides complex TA handling with the TEE Internal Core API from the TA. It receives the command from a CA and executes the function defined in a cloud application.

TZmediator can enhance security by executing code in TAs using WebAssembly (Wasm) [13] if necessary. Even if cloud applications minimize the size of TAs, the TAs could still be compromised. Wasm enables applications developed in various programming languages to run securely inside a sandbox. This sandbox makes it difficult for compromised cloud applications to attack the trusted OS or the other TAs in the secure world. It should be noted that the sandbox introduces execution overhead in exchange for security. To communicate with the other applications, Wasm applications can use the WebAssembly System Interface (WASI) [14], which provides a POSIX-like API. TZmediator extends WASI to increase compatibility with POSIX. When a Wasm application in a TA invokes a function provided by the WASI library, the Wasm runtime in the same TA invokes the corresponding shadow process via the TZm-TA library.

4. Implementation

We have implemented TZmediator on OP-TEE 3.15.0 [15]. OP-TEE provides components necessary to execute TAs, including the OP-TEE OS and the tee-supplicant daemon. The OP-TEE OS is a trusted OS running in the secure world. The tee-supplicant daemon runs in the normal world and handles requests from TAs. OP-TEE also provides libraries conforming to the GlobalPlatform APIs. For a rich OS running in the normal world, we used Linux including the OP-TEE driver. To run Wasm applications in TAs, we have extended WebAssembly Micro Runtime (WAMR) [19] and WASI Preview 1 [14].

4.1. Spawning TAs

TZmediator enables a CA to treat TAs like its child processes by providing the emulated `posix_spawn` function in the TZm-CA library. Fig. 3 illustrates the flow of spawning a TA. To spawn a TA, a CA invokes this function, which is included in POSIX APIs and used to spawn a child process. The original `posix_spawn` takes the pathname of an executable file as an argument, but a CA specifies the string of the universally unique identifier (UUID) assigned to a TA instead. If the normal pathname is specified, the TZm-CA library invokes the original `posix_spawn`, which is provided in the C standard library. Note that TZmediator

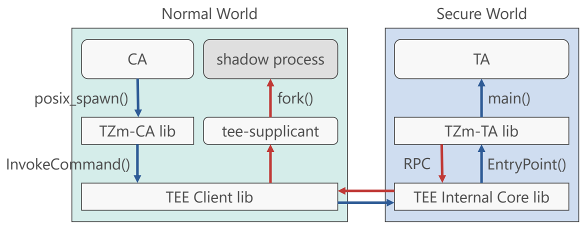


Figure 3. Spawning a TA with a shadow process.

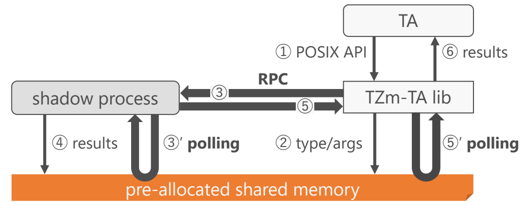


Figure 4. Delegating a POSIX API from a TA to its shadow process.

does not support `fork` and `execve` because it is difficult to emulate them for spawning TAs.

The `posix_spawn` function in the TZm-CA library invokes several functions defined in the TEE Client API to run a TA, so that each cloud application does not need to implement their invocation. First, it invokes several functions to initialize a TA and open a session with the TA. After initialization, it invokes `TEEC_InvokeCommand` to execute the `start` command implemented in the TZm-TA library linked to the TA. After this command invocation, the CA remains blocked until the TA completes the execution of the command. To run the CA in parallel to the TA, the TZm-CA library creates a subthread before the command invocation so that the CA can continue running the cloud application in the main thread.

The TZm-TA library provided to a TA implements the mandatory entry points [20] defined in the TEE Internal Core API, so that each cloud application does not need to implement them. When a CA creates a TA instance and connects to it, several functions are invoked. Upon the invocation of the `start` command by the CA, `TA_InvokeCommandEntryPoint` is triggered. This function invokes the `main` function implemented in a cloud application. If the control returns from the `main` function, the command execution completes.

When spawning a Wasm application in a TA, a CA invokes `posix_spawn` in the TZm-CA library by specifying the pathname of the Wasm application, instead of the UUID of a TA. In TZmediator, a cloud application is first compiled into Wasm bytecode. Then, the bytecode is compiled into native code using the ahead-of-time (AOT) compiler. To spawn a Wasm application, the `posix_spawn` function implicitly runs the Wasm runtime as a TA by specifying its UUID. Then, it stores the specified Wasm application in the buffer shared with the TA and executes the `start` command using `TEEC_InvokeCommand`. In the secure world, the TZm-TA library executes the Wasm runtime, which loads the Wasm application in the shared buffer and executes its `main` function.

4.2. Shadow Process

The TZm-TA library creates a shadow process corresponding to a TA in the normal world when a new session is established. First, it sends an RPC request to the tee-supplicant daemon in the normal world. Then, the daemon creates a child process used for a shadow process and returns its process ID to the TA. This ID is used for specifying the TA. The daemon handles the RPC request using the handler provided by the plugin for the shadow process. The plugin of tee-supplicant is implemented as a shared library using the TEE Client API and dynamically loaded when tee-supplicant is started. A unique UUID is assigned to the plugin and used by TAs to specify the used RPC handler. Upon receiving the RPC request, tee-supplicant dispatches the request to the appropriate plugin based on the specified UUID.

Since a shadow process is a child process of tee-supplicant, TZmediator emulates a shadow process as a direct child process of a CA. For example, a shadow process sets the same process group ID as the CA that has spawned the corresponding TA by default. If the CA specifies a different process group ID to `posix_spawn`, a shadow process sets the specified process group ID. A process group is used to control the distribution of a signal.

4.3. Delegation of POSIX APIs

To delegate POSIX APIs from a TA running in the secure world to a shadow process residing in the normal world, TZmediator can use the RPC mechanism used between a TA and tee-supplicant. However, that mechanism is heavyweight because it needs to switch worlds six times. First, a TA performs an RPC to allocate shared memory for passing the arguments of the POSIX API. Then, it performs an RPC to execute the specified POSIX API in the shadow process. Finally, it performs an RPC to deallocate the shared memory.

To reduce this overhead, TZmediator provides a *lightweight RPC* mechanism, which needs only two world switches. As illustrated in Fig. 4, it pre-allocates shared memory between a TA and its shadow process and reuses it for all the invocations of POSIX APIs. When a CA spawns a TA, it creates a shared memory object by specifying a unique name and maps the shared memory. Using the address of the shared memory, it registers the shared memory using the TEE Client API. The name of the created shared memory object is passed from the CA to the shadow process via the TA. Finally, the shadow process opens the shared memory object using the received name and maps the shared memory.

When the TA executes a function in POSIX APIs, the TZm-TA library stores the function type and its arguments in the buffer created in the shared memory. Then, it switches the context to the normal world and wakes up the corresponding shadow process. This thread waits for the shadow process to complete the execution of the POSIX API. When the shadow process stores the result of the execution in the shared buffer and wakes up the waiting thread, the thread switches the context to the secure world. Finally, the TA receives the result from the shared buffer.

To avoid world switches, TZmediator supports a *polling-based method* with the shared memory as well. The shadow process polls the buffer in the shared memory to detect when a TA sends a new request. Similarly, the TZm-TA library polls the shared buffer to detect the completion of the execution of the POSIX API. To reduce the CPU overhead due to polling, the TA and the shadow process can repeat the check of the shared buffer and a short sleep. Since the OP-TEE OS supports only a millisecond-order sleep, TZmediator provides a microsecond-order sleep by modifying the TEE Internal Core API, the OP-TEE OS, and the Linux kernel. Note that the TA needs to switch the context to the normal world to sleep.

4.4. Message Passing API

TZmediator supports various POSIX APIs for message passing between CAs and TAs, e.g., two types of pipes and several types of sockets.

4.4.1. Pipes. When a TA creates a *named pipe*, also known as a FIFO, it invokes the `mkfifo` function provided by the TZm-TA library. This function stores the specified pathname and access mode in the buffer shared with the shadow process and then invokes the shadow process using the lightweight RPC or the polling-based method. Then, the shadow process invokes the `mkfifo` function in the C standard library using the received pathname and access mode. Therefore, the TA needs to specify a valid pathname in the normal world. The function invokes the rich OS by issuing the `mknodat` system call and creates a named pipe in the normal world. The shadow process stores the result in the shared buffer and returns the result to the TA via the TZm-TA library. To delete a named pipe, a TA executes the `unlink` function in the normal world via the shadow process.

To open a named pipe, a TA invokes the `open` function with its pathname. It can open both the named pipe created by itself and the one created by a CA or another process in the normal world. Like `mkfifo`, this function is executed by the shadow process on behalf of the TA. Then, it returns the resulting file descriptor to the TA. The TA uses this file descriptor for the following pipe operations. When it sends a byte stream to the pipe using the `write` function, the TZm-TA library copies the specified data to the shared buffer and invokes the shadow process. The shadow process writes the data in the shared buffer to the pipe specified by the file descriptor. When a TA receives a byte stream from the pipe using the `read` function, the TZm-TA library invokes the shadow process. The shadow process reads data from the pipe and stores it in the shared buffer. Then, the library copies the data in the shared buffer to the buffer specified by the TA. To close a named pipe, the TA invokes the `close` function with the file descriptor.

When a CA communicates with a TA using *unnamed pipes*, it first invokes the `pipe` function in the C standard library to create a pair of pipes for reading and writing. Then, it specifies a pair of file descriptors returned by this function as an argument of `posix_spawn` to share them with a newly created TA. When the `posix_spawn` function in the TZm-CA library spawns a TA, it passes the file descriptors to the shadow process created by the TA using a mechanism called *file descriptor passing*. This mechanism allows a process to transfer its file descriptor to another process using a UNIX domain socket and share it between the two processes. According to the argument of `posix_spawn`, the shadow process usually redirects the passed file descriptors to the standard input and output. As a result, the TA can implicitly use the inherited file descriptors to perform pipe operations through the standard input and output of the shadow process.

4.4.2. Sockets. When a TA creates a socket, it invokes the `socket` function in the TZm-TA library. This function takes an address family such as `AF_UNIX` for UNIX domain and `AF_INET` for TCP and UDP. Then, the shadow process invokes the standard `socket` function to create a socket in the normal world. Then, it returns the resulting file descriptor to the TA.

When a TA acts as a client and connects to a CA, it invokes the `connect` function in the TZm-TA library. For a UNIX domain socket, it specifies the pathname used by the CA as its argument. For a TCP socket, it specifies a *loopback address* and a unique port because the TA and the CA run on the same host. When a TA acts as a server and waits for incoming connections from a CA, it invokes the `bind`, `listen`, and `accept` functions provided by the TZm-TA library. For a UNIX domain socket, it specifies a valid pathname in the normal world as an argument of `bind`. Once a connection is established, the TA sends and receives data using the `send` and `recv` functions or the `write` and `read` functions in the TZm-TA library. For a UDP socket, the TA uses the `sendto` and `recvfrom` functions by specifying a loopback address without establishing a connection.

4.5. Shared Memory API

To support shared memory between a CA and a TA, TZmediator emulates the POSIX API for shared memory, as shown in Fig. 5. When a CA spawns a TA, the TZm-CA library pre-allocates shared memory with sufficient size and registers it because it is difficult to allocate shared memory after a TA is spawned. This is similar to the registration of the shared memory used for communication between a TA and its shadow process. Unlike that, the library creates the data structure for managing POSIX shared memory objects in this pre-allocated shared memory. The address and size of the created shared memory are passed to a TA when the

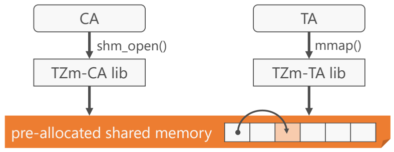


Figure 5. The emulation of POSIX shared memory.

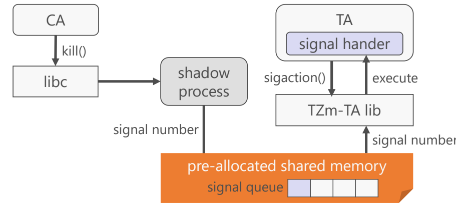


Figure 6. The emulation of signal delivery.

TZm-CA library executes the `start` command defined in the TZm-TA library.

Both the TZm-CA and TZm-TA libraries provide a set of emulated functions for manipulating POSIX shared memory. When a CA or a TA invokes the `shm_open` function with a name, the function registers a new POSIX shared memory object with that name in the management data structure and returns a new file descriptor. The `ftruncate` function allocates a new memory region with the specified size in the pre-allocated shared memory for the specified shared memory object. The `mmap` function returns the address of the allocated memory region, while the `munmap` function does nothing. The `shm_unlink` function removes the specified shared memory object from the management data structure. Note that the shadow process is not involved in the manipulation of POSIX shared memory.

4.6. Signal API

TZmediator emulates signal delivery in a shadow process and the TZm-TA library, as depicted in Fig. 6. When a shadow process is created, it registers a signal handler for all catchable signals. A CA sends a signal to a TA by specifying the ID of the TA to the standard `kill` function. Since that ID is the same as the ID of the shadow process, the signal is delivered to the shadow process by the rich OS. The invoked signal handler stores the number of the received signal in the signal queue created in the memory shared with the TA. Note that TZmediator does not support the KILL and STOP signals because those signals cannot be caught.

The TZm-TA library checks the signal queue in the shared memory periodically, e.g., when the TA invokes a function provided by the library. If a signal number is stored in the queue, the library dequeues it and invokes the corresponding signal handler. If the TA has registered a custom handler for that signal, the library invokes that handler defined by the cloud application. The registration of signal handlers is performed using the `signal` or `sigaction` function. These functions do not invoke the shadow process but register signal handlers inside the library. If a custom handler is not registered, the library performs the default action for that signal. If the default action is termination, the library terminates the TA. In the polling mechanism of the signal queue, a signal is not delivered until a TA invokes the TZm-TA library. However, we assume that a TA frequently uses POSIX APIs for cooperative execution with a CA.

To send a signal, a TA invokes the `kill` function provided by the TZm-TA library. This function takes the process ID of a CA and a signal number as its arguments. The process ID of a CA can be obtained using the `getppid` function, which is used to return the ID of the parent process. This function obtains the process ID of a CA from the memory shared with the CA. The TZm-TA library invokes the shadow process to send the specified signals to the CA via the rich OS.

4.7. WASI Extension

TZmediator extends WASI so that Wasm applications can use communication-related POSIX APIs. WASI Preview 1 we used supports only a part of the POSIX APIs. When a Wasm application running in a TA invokes the function provided by the WASI library, the function invokes the Wasm runtime, which runs in the same TA. Then, the Wasm runtime invokes the TZm-TA library, which finally invokes the shadow process. Thus, a Wasm application in a TA can communicate with a CA in a similar way to a native application. For example, it can use various functions for message passing, e.g., `read` and `write`.

TZmediator allows Wasm applications to access POSIX shared memory only via POSIX APIs. Since Wasm provides memory as a single, contiguous byte array called linear memory, Wasm applications cannot directly access the shared memory outside it due to the boundary checking of the Wasm runtime. Therefore, TZmediator extends POSIX APIs so that Wasm applications can perform the manipulations of not only the linear memory but also the shared memory. For example, the extended `strcpy` function can take two addresses of the linear memory or the shared memory. If the address is outside the range of the linear address, the Wasm runtime checks that the address is within the shared memory. It also translates the address of the linear memory into the host address because the address of the linear memory is valid only for Wasm applications. Then, it copies the string between the linear memory and the shared memory.

TZmediator supports signal delivery to Wasm applications by using *indirect calls*. When a Wasm application invokes the `sigaction` function, the Wasm runtime registers the specified signal handler. Since a function is identified by a function index in Wasm, the Wasm runtime cannot directly execute the registered function when a signal is delivered to a TA. Therefore, the Wasm runtime indirectly invokes the corresponding function using the registered index. To enable this indirect call, Wasm applications export the functions used as signal handlers at link time.

5. Security Analysis

Since TZmediator runs shadow processes in the normal world for inter-world seamless communication using POSIX APIs, attackers in the normal world could compromise them. They could eavesdrop on, tamper with, and prevent communication between CAs and TAs. However, they could mount these attacks more easily by compromising CAs. In addition, it is more difficult to compromise shadow processes because they run in higher privileges than CAs. Therefore, using shadow processes does not lower the security of TrustZone.

The introduction of POSIX APIs between CAs and TAs may expand the attack surface. For example, insufficient input validation could lead to traditional types of vulnerabilities, such as buffer overflows, which could be exploited by attackers in the normal world. Using Wasm as TAs could address some of the issues. Furthermore, risks may also arise from the parallel execution of CAs and TAs. If the shared memory regions used for communication are not properly synchronized, race conditions or cache coherence issues may occur. These potentially undermine the integrity and consistency of exchanged data. Careful memory management is necessary to prevent unintended information leakage or data corruption.

6. Experiments

We conducted several experiments to measure communication performance between a CA and a TA in a cloud application using TZmediator. First, we examined the impact of the delegation method of POSIX APIs. Then, we compared TZmediator with traditional communications using the TEE APIs and between two CAs inside the normal world. We also inspected scalability in terms of the number of cloud applications. In addition, we measured the performance of a real-world cloud application. We used an Armadillo-X2 device [21], which is equipped with an ARM Cortex-A53 processor and 2 GB of memory. We ran OP-TEE 3.15.0 in the secure world and Linux 5.10 in the normal world.

6.1. Communication Performance

We examined communication performance using various delegation methods of POSIX APIs from a TA to its shadow process, as described in Section 4.3. We used the original RPC mechanism, the lightweight RPC mechanism, and the polling-based method. For polling, we configured the wait time to 0 or 1 μs. We performed communication between a CA and a TA using various POSIX APIs. As the message passing APIs, we used named and unnamed pipes, and UNIX domain, TCP, and UDP sockets. When we used the shared memory API, the API was not delegated from the TA to its shadow process to access POSIX shared memory. Instead, both the CA and the TA polled the shared memory with no wait time or a wait time of 1 μs by themselves. When we used the signal API, the TA used the polling-based method to check the signal queue in the pre-allocated shared memory. Since the shared memory used in the original RPC is temporary, the TA checked the signal queue in the shadow process using the RPC. In this experiment, the TA ran a native application.

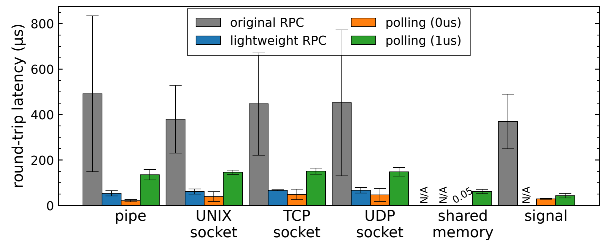


Figure 7. The round-trip latency of CA-TA communication.

6.1.1. Round-trip Latency. We measured the round-trip latency between a CA and a TA using POSIX APIs. For the message passing APIs, the CA sent 4 KB of data to the TA using a pipe or a socket. When the TA received the data via the shadow process, it sent the data back to the CA via the shadow process using the same pipe or socket. For the shared memory API, the CA wrote 4 KB of data to POSIX shared memory. After the TA read the data and then wrote the data to the shared memory, the CA read that data. For the signal API, the CA sent a signal to the TA waiting for signal delivery using the `pause` function. When the TA received the signal via the shadow process, it sent the same signal to the CA in the signal handler.

The latency was similar for all POSIX APIs, as shown in Fig. 7. We omit the result of unnamed pipes due to space limitations. The result was similar to that of named pipes. Compared with the original RPC, the lightweight RPC reduced the latency by 319-438 μs. This is because the number of world switches was reduced. Polling with no wait time achieved the lowest latency, specifically 0.05-48 μs, thanks to no world switch. However, polling with a wait time of only 1 μs increased the latency largely because a sleep in the TA needed a world switch. This latency was much higher than that in the lightweight RPC but still lower than that in the original RPC.

6.1.2. Throughput. We measured the throughput between a CA and a TA using POSIX APIs. For the message passing APIs, the TA continuously sent 4 KB of data to the CA. For the shared memory API, the TA repeatedly wrote 4 KB of data to the ring buffer created in shared memory. We calculated the throughput from the time needed until the CA completed receiving all the data. We did not measure the throughput of signals because the same signal sent is discarded until it is delivered to the CA.

The trend of the throughput was similar for the message passing APIs. As shown in Fig. 8, the throughput in the lightweight RPC was 5.5-7.3x higher than that in the original RPC. Furthermore, polling with no wait time achieved 2.6-5.8x higher throughput than the lightweight RPC. However, polling with a wait time of 1 μs decreased the throughput largely. Compared with the latency, only the throughput

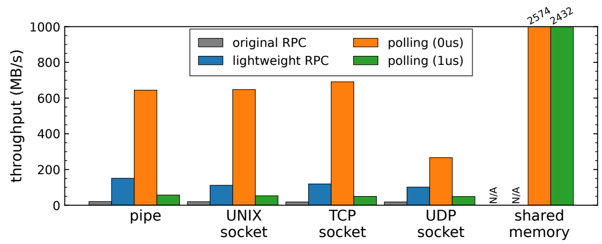


Figure 8. The throughput of CA-TA communication.

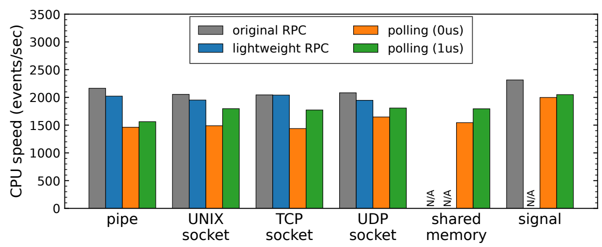
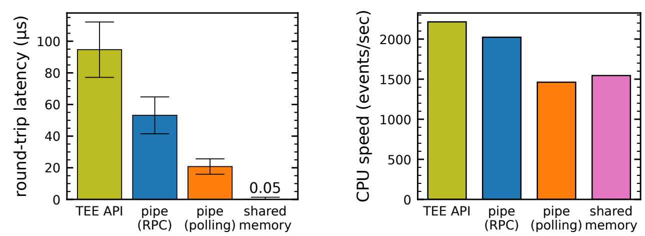


Figure 9. The system performance during CA-TA communication.



(a) Latency

(b) System performance

Figure 10. The performance of the communication using TEE APIs.

in polling with no wait time was much higher. This is probably because the world switches affected the throughput significantly. For the shared memory API, in contrast, the throughput was 3.7-10x higher than that in the message passing APIs. It was almost not degraded even by polling with a wait time of 1 μs.

6.1.3. Overhead. To examine the impact of TZmediator on the entire system, we measured the system performance in the normal world. We executed sysbench 1.0.20 [22] and measured the score of the CPU benchmark with four threads. During the execution, we continuously performed communication between a CA and a TA like the experiment of measuring the latency.

As shown in Fig. 9, the overhead of TZmediator was almost the same for all the POSIX APIs. Unlike the latency and the throughput, the benchmark achieved the highest score in the original RPC. The reason is that the frequency of communication was the lowest. Since the frequency was higher in the lightweight RPC, the system performance became 0.2-6.6% lower. In contrast, polling with no wait time degraded the system performance by 14-32%, compared with the original RPC. This is because the CPU overhead was large due to a busy wait in addition to more frequent communication. A slight wait time in polling increased the system performance.

6.1.4. Summary. Polling with no wait time achieved the best performance in latency and throughput, but the system performance was the worst due to the CPU overhead. This delegation method is the most performance-oriented. In contrast, the lightweight RPC was the second best in not only latency and throughput but also overhead. This is the most balanced delegation method and outperforms polling with a time wait of 1-μs. Therefore, we used these two delegation methods in the following experiments.

6.2. Comparison with TEE APIs

We measured the latency when using the TEE APIs between a CA and a TA and compared it with that of POSIX APIs. The CA wrote 4 KB of data to the memory shared with the TA and executed the `benchmark` command of the TA using the TEE Client API. When the TA was invoked by the TEE Internal Core API, it read the data and then wrote 4-KB data to the shared memory. After the execution of the command, the CA read the data from the shared memory. We also measured the system performance during the communication. We did not measure the throughput because the CA cannot continuously send data without waiting for the completion of the command execution in the TA.

Fig. 10(a) shows the latency in the TEE APIs as well as those in named pipes and POSIX shared memory. The latency of pipes was 41 μs lower than that of the TEE APIs even when we used the lightweight RPC. When we used shared memory, the latency was decreased by 95 μs. In contrast, the system performance was the highest in the TEE APIs, as shown in Fig. 10(b), but the difference from that in pipes with the lightweight RPC was only 8.7%.

6.3. Comparison with CA-CA Communication

We measured communication performance between two CAs in the normal world and compared it with that between a CA and a TA. The CAs sent and received data using not the TEE Client APIs but POSIX APIs as normal processes. Fig. 11 shows the latency of CA-CA communication as well as CA-TA communication using TZmediator. Compared with CA-CA communication, the latency in TZmediator increased by up to only 4.4 μs in the polling-based method. When we used the lightweight RPC, the latency became 23-35 μs higher. As shown in Fig. 12, the throughput in socket communication was only 8.8-18% lower when we used polling, but pipes degraded the throughput largely. Since a larger buffer was used for pipes in the Linux kernel, the two CAs could run more asynchronously. The system performance degraded only by 8.6-13% when we used the lightweight RPC, as shown in Fig. 13.

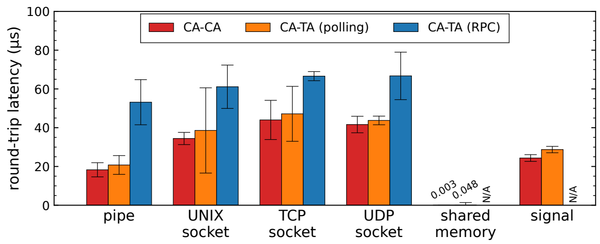


Figure 11. The round-trip latency of CA-CA communication.

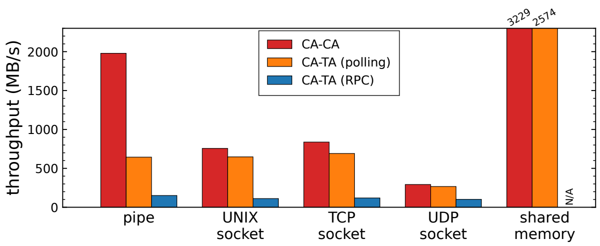


Figure 12. The throughput of CA-CA communication.

6.4. Comparison with Wasm

We measured communication performance between a CA and a Wasm application running in a TA and compared it with that when running a native application in a TA. Figs. 14 and 15 show the latency and the throughput, respectively. When we used the lightweight RPC, the latency in a Wasm application was only 4.8-8.9 μs higher, and the throughput was only 5.0-24% lower. This is thanks to the relatively large overhead of the RPC. When we used polling, in contrast, the latency increased by 15-18 μs, except for signals, and the throughput decreased by 4.5-68%. This is due to the overhead of Wasm and WASI. The system performance was almost the same between native and Wasm applications, as shown in Fig. 16.

6.5. Scalability

To examine the scalability of TZmediator, we measured the latency when increasing the number of running cloud applications, each of which consisted of one CA and one TA, up to four. Each CA and TA communicated with each other using named pipes. We ran either a native or a Wasm application in all the TAs. Fig. 17 shows the average latency. When we used polling, the latency was kept lower while the number of cloud applications was small. However, it became 2.9x and 3.9x higher for native and Wasm applications, respectively, when four cloud applications ran. When we used the lightweight RPC, in contrast, the latency increased gradually as the number of cloud applications increased. When four cloud applications ran, the latency was 28-93 μs lower than when we used polling. This means that the lightweight RPC is better in terms of scalability.

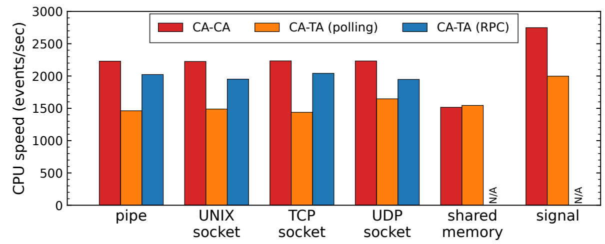


Figure 13. The system performance during CA-CA communication.

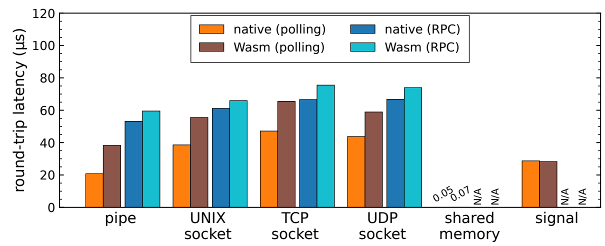


Figure 14. The round-trip latency of CA-Wasm communication.

6.6. Real-world Cloud Application

To examine the performance of a real-world application using TZmediator, we modified DarkneTZ [18] for cooperative execution using POSIX APIs. DarkneTZ is a framework that combines model partitioning with TrustZone to reduce the attack surface of deep neural networks (DNNs). In this experiment, we executed the 9th softmax layer in the secure world and the other nine layers in the normal world. The CA and the TA exchanged input and output data using POSIX APIs. We used a pre-trained model on the MNIST dataset of handwritten digit images. We measured the inference time and compared it with the original DarkneTZ using the TEE APIs.

We confirmed that DarkneTZ with TZmediator could succeed in correct inference. DarkneTZ with POSIX shared memory increased the time only by 2.9%, compared with the original implementation. In contrast, the inference time was 14% longer in DarkneTZ with named pipes even when we used polling.

7. Related Work

Intel SGX is one of the TEEs and enables a process to create protection domains called enclaves inside it and to securely run enclave applications in an untrusted environment. Users develop enclave applications using the SGX SDK, which provides a special API that is largely different from POSIX APIs. Several systems have been proposed to provide POSIX APIs to enclave applications. SCONE [23] provides a custom C library to enclave applications. This library invokes the OS using a mechanism called asynchronous system calls without switching the context between an enclave and the outside of it. When an enclave

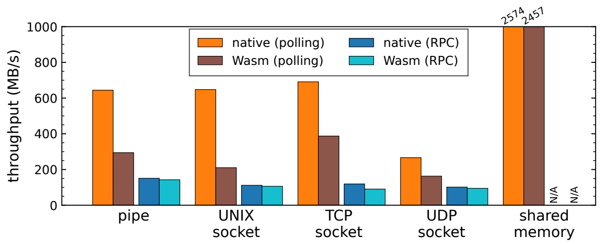


Figure 15. The throughput of CA-Wasm communication.

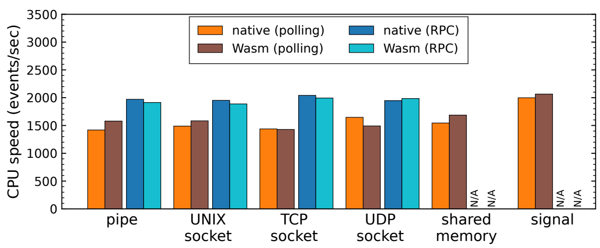


Figure 16. The system performance during CA-Wasm communication.

passes a request to an OS thread via the request queue, the OS thread executes the system call and returns the result via the response queue. Using system calls, enclave applications can communicate with other processes via the OS.

There are systems that partition a complex enclave application into small ones and enable cooperative execution among multiple enclaves using POSIX APIs. Graphene-SGX [24] (currently, Gramine) provides not only the C standard library but also a library OS to enclave applications. Multiple enclave applications can perform direct communication via the library OS. The library OS supports `fork` and `execve`, which create a new enclave and execute a new enclave application, respectively. Occlum [25] also provides a library OS and allows users to execute multiple applications inside an enclave using threads. As a result, it can efficiently achieve inter-process communication only in the library OS.

eMCOS POSIX [26] is a fully POSIX-compliant real-time OS running in the secure world of TrustZone using the VOSySmonitor hypervisor [27]. It enables cooperative execution among multiple processes and threads with POSIX APIs. However, it supports inter-process and inter-thread communication only inside the secure world. It does not provide any interface for communication with processes running in the normal world. Moreover, it is not always suitable for running cloud applications because processes and threads on the real-time OS are scheduled with high priority.

WebAssembly System Interface (WASI) [14] is the system interface for Wasm and is currently under standardization. WASI Preview 1 provides APIs similar to POSIX, but its socket support is limited. In WASI Preview 2, socket support has been significantly enhanced. Beyond WASI, WASIX [28] has been proposed as an extension of WASI

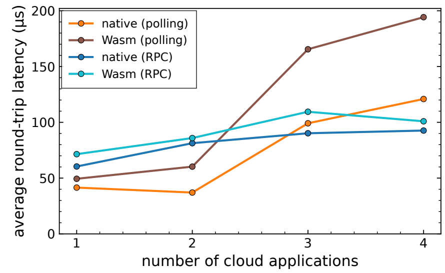


Figure 17. The scalability of TZmediator.

to increase POSIX compatibility. WebAssembly Linux Interface (WALI) [29] provides the interface of the Linux system calls and allows Linux applications to run as Wasm applications.

ReZone [9] partitions the secure world of TrustZone and allows each TA to be executed in isolation. It creates a sandboxed environment called a zone for each TA and provides a separate trusted OS for each zone. Zones can access only their assigned resources, while access to other zones or the normal world is restricted. Even if a cloud application running as a TA is compromised, ReZone can prevent attacks against the outside of that zone. However, the amount of resources available to cloud applications can be limited because the resources assigned to the secure world are partitioned among zones. Using ReZone with TZmediator is useful because TZmediator runs only small security-sensitive tasks in the secure world.

8. Conclusion

In this paper, we proposed TZmediator to enable cooperative execution using POSIX APIs for a cloud application partitioned into the two worlds of TrustZone. TZmediator executes only security-sensitive tasks as TAs in the secure world and also runs them as Wasm applications if stronger isolation is necessary. To enable communication across worlds using POSIX APIs, it creates a shadow process corresponding to each TA in the normal world. A TA delegates the invocation of POSIX APIs to the shadow process. Then, the shadow process invokes the APIs using the standard library on behalf of the TA. We confirmed that communication performance between CAs and TAs was high in most of the POSIX APIs through experiments.

One of our future work is to enable a TA to spawn its child processes as other TAs and to achieve cooperative execution with them using POSIX APIs. Furthermore, we are planning to develop cloud applications in which multiple CAs and TAs run in parallel and cooperate via POSIX APIs. Such cloud applications would achieve more efficient cooperation.

Acknowledgments

This work was supported by JST, CREST Grant Number JPMJCR21M4, Japan.

References

[1] International Data Corporation, "IDC's Worldwide Edge Spending Guide Taxonomy, 2025: Release V1," 2025.

[2] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge Computing: Vision and Challenges," in *IEEE Internet of Things Journal*, vol. 3, no. 5, 2016, pp. 637–646.

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

[4] ARM Ltd., "ARM Security Technology – Building a Secure System Using TrustZone Technology," White Paper, 2009.

[5] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An Open Framework for Architecting Trusted Execution Environments." Proc. European Conf. Computer Systems, 2020.

[6] Advanced Micro Devices, Inc., "Secure Encrypted Virtualization API Version 0.24," 2020.

[7] Intel Corporation, "Intel Trust Domain Extension," White Paper, 2023.

[8] ARM Ltd., "Arm Confidential Compute Architecture," https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture.

[9] D. Cerdeira, J. Martins, N. Santos, and S. Pinto, "ReZone: Disarming TrustZone with TEE Privilege Reduction," in *Proc. Security Symp.*, 2022, pp. 2261–2279.

[10] GlobalPlatform, "GlobalPlatform Device Technology TEE Client API Specification Version 1.0," 2010.

[11] ——, "GlobalPlatform Technology TEE Internal Core API Specification Version 1.3.1," 2021.

[12] IEEE, "IEEE Standard for Information Technology – Portable Operating System Interface (POSIX) Base Specifications, Issue 8," 2024.

[13] J. Menetrey, M. Pasin, P. Felber, and V. Schiavoni, "WaTZ: A Trusted WebAssembly Runtime Environment with Remote Attestation for TrustZone," in *Proc. Int. Conf. Distributed Computing Systems*, 2022, pp. 1177–1189.

[14] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with WebAssembly," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, 2017, p. 185–200.

[15] Linaro Ltd., "OP-TEE," https://www.trustedfirmware.org/projects/op-tee.

[16] Open-TEE, "Open-TEE," https://open-tee.github.io/.

[17] Google Inc., "Trusty tee — android open source project," https://source.android.com/docs/security/features/trusty.

[18] F. Mo, A. S. Shamsabadi, K. Katevas, S. Demetriou, I. Leontiadis, A. Cavallaro, and H. Haddadi, "DarkneTZ: towards model privacy at the edge using trusted execution environments," in *Proc. Int. Conf. Mobile Systems, Applications, and Services*, 2020, p. 161–174.

[19] Bytecode Alliance, "WebAssembly Micro Runtime (WAMR)," https://github.com/bytecodealliance/wasm-micro-runtime.

[20] OP-TEE, "Trusted Applications," https://optee.readthedocs.io/en/latest/building/trusted_applications.html.

[21] Atmark Techno, Inc., "Armadillo-X2," https://armadillo.atmark-techno.com/armadillo-x2.

[22] A. Kopytov, "sysbench," https://github.com/akopytov/sysbench/tree/1.0.20.

[23] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "SCONE: Secure Linux Containers with Intel SGX," in *Proc. Symp. Operating Systems Design and Implementation*, 2016, pp. 689–703.

[24] C. Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX," in *Proc. Annual Technical Conf.*, 2017, pp. 645–658.

[25] 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 *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, 2020.

[26] eSOL Co., Ltd., "eMCOS POSIX - High Performance, POSIX-Compliant Real-Time Operating System Designed for Critical Multicore," https://www.esol.com/embedded/product/emcos-posix_overview.html.

[27] P. Lucas, K. Chappuis, B. Boutin, J. Vetter, and D. Raho, "VOSYSmonitor, a TrustZone-based Hypervisor for ISO 26262 Mixed-critical System," in *Proc. Conf. Open Innovations Association FRUCT*, 2018, pp. 231–238.

[28] Wasmer Inc., "WASIX - The Superset of WASI," https://wasix.org/.

[29] A. Ramesh, T. Huang, B. L. Titzer, and A. Rowe, "Empowering WebAssembly with Thin Kernel Interfaces," in *Proc. European Conf. Computer Systems*, 2023, p. 1–20.