SICS Technical Report T2013:02
ISSN 1100-3154

VETE: Virtualizing the Trusted Execution Environment

Arash Vahidi, SICS
Patrik Ekdahl, Ericsson Research

2013-03-05

Table of Contents

Introduction

This document is the final report for the SICS project Virtualizing the Trusted Execution Environment (VETE). This project was carried out in close collaboration with Ericsson Research and with support from ST-Ericsson.

Motivation

Mobile computing devices such as smartphones are an important part of our lives. We are using phones to make payments, to communicate with friends and family, and to carry out an ever increasing parts of our jobs. In short, we are becoming more and more dependent on mobile computing devices and expect such systems to work correctly and securely.

At the same time, phones (and embedded systems in general) are reaching a level of complexity where existence of critical errors in their software is almost inevitable. At the same time, even minor software glitches can have effect on our lives. For example, recently a software error in the alarm application of a popular mobile OS resulted in a large number of people arriving late to their works, which in some places also resulted in further problems with public transportation. These types of problems will only accelerate when cyber-criminals start actively exploiting software glitches in phones and embedded systems for financial gain.

One method to improve security of computing devices is to create isolated layers of software such that a problem in one layer does not affect other layers. For example, in general purpose operative systems, memory protection is used to separate applications from each other and from the operative system. Hence, an error in one application cannot directly affect any other applications. Hypervisor technologies extend this idea by introducing another layer of isolation, which separates multiple operative systems from each other and/or from the bare metal hardware. Hence, hypervisors can be used as *security enablers.*

Project background

The SICS Thin Hypervisor (STH) is a small and portable hypervisor for embedded systems. The small size of STH makes it suitable for many embedded systems where resources are scarce. The STH is currently being updated to support more powerful systems and the ability to host more complex guests such as the Linux kernel. At the same time, the implementation is being formally verified to ensure correctness.

Another focus of this project is the Trusted Execution Environment (TEE) specification of GlobalPlatform. The TEE specification defines an environment where Trusted Applications (TA) can be executed in a secure manner, and normal (possibly untrusted) applications can utilise the functionality of the TA. The specification also defines a method for secure communication between applications.

NovaThor from ST-Ericsson is a family of advanced System-on-Chip (SoC) platform used primarily in smartphones. The NovaThor utilizes a TrustZone enabled ARMv7 CPU, which adds a secondary isolation layer. This isolation layer separates the *Normal world* from the new *Secure world*, inside which trusted application can run with full isolation from the rest of the system including whatever general purpose operative system or applications the system is running.

Some devices in the NovaThor family (such as L9540 and L8540) fully support the TEE specification. For the purpose of this work however, we have chosen the NovaThor U8500 platform which does not fully implement a GlobalPlatform compliant TEE. The U8500 contains a light-weight Trusted Execution Environment that separates Trusted Applications from untrusted applications but does not provide separation between the trusted applications themselves. In theory, a buggy or malicious Trusted Application could compromise security of the device.

Project goals

The goal of this project is to design and implement a hypervisor for the U8500 NovaThor platform that operates inside the Secure world of the main CPUs. The hypervisor will virtualize the underlying hardware in such way that the Trusted Execution Environment either directly or with the help of the hypervisor can provide secure isolation between the Trusted Applications, and between Trusted Applications and normal applications.

Project restrictions

Due to time constraints, in this project we will not consider the following subjects

- Multicore
- Scheduling
- Virtualization of other SoC components such as GPU and DSP

System architecture

The target system for this project is the NovaThor family from ST-Ericsson [1]. The architecture and the associated software is explained in this chapter.

The NovaThor U8500 platform

For this projects, we will use the NovaThor U8500. The U8500 combines a modern dual-core ARM Cortex-A9 CPU with a HSPA+ modem on a single die.

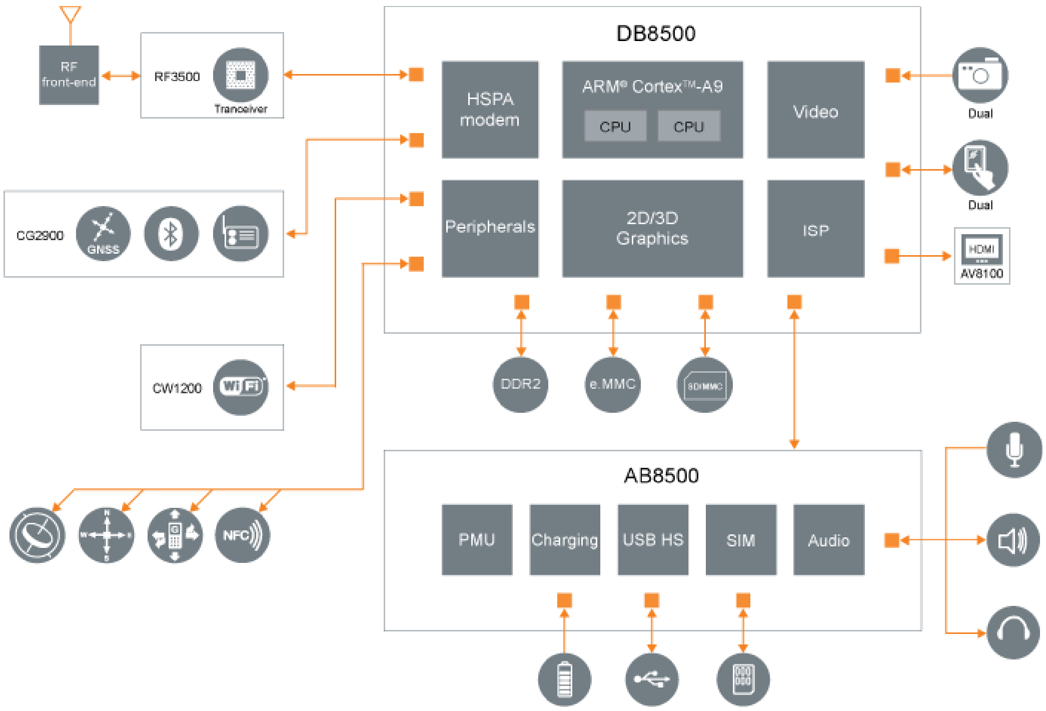


Illustration 1: The NovaThor U8500 platform (source: stericsson.com)

The U8500 incorporates a large number of components such as an ARM Mali 400 GPU and multiple signal processing units. In this project however, we will limit us to the Cortex-A9 CPUs plus some minor peripherals such as the USART for debugging.

ARM Cortex-A9 architecture

The ARM Cortex-A9 is a multicore capable 32-bit ARM processor that implements the ARMv7a architecture [2]. It is currently the most popular architecture in mobile computing devices.

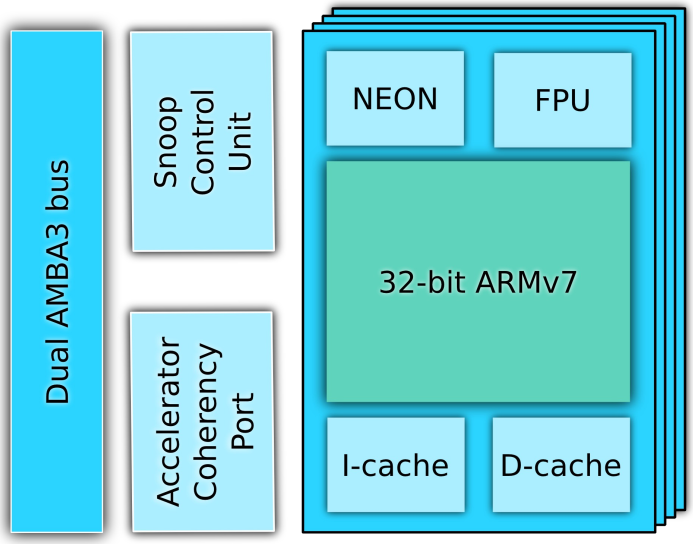


Illustration 2: The ARM Cortex-A9 CPU.

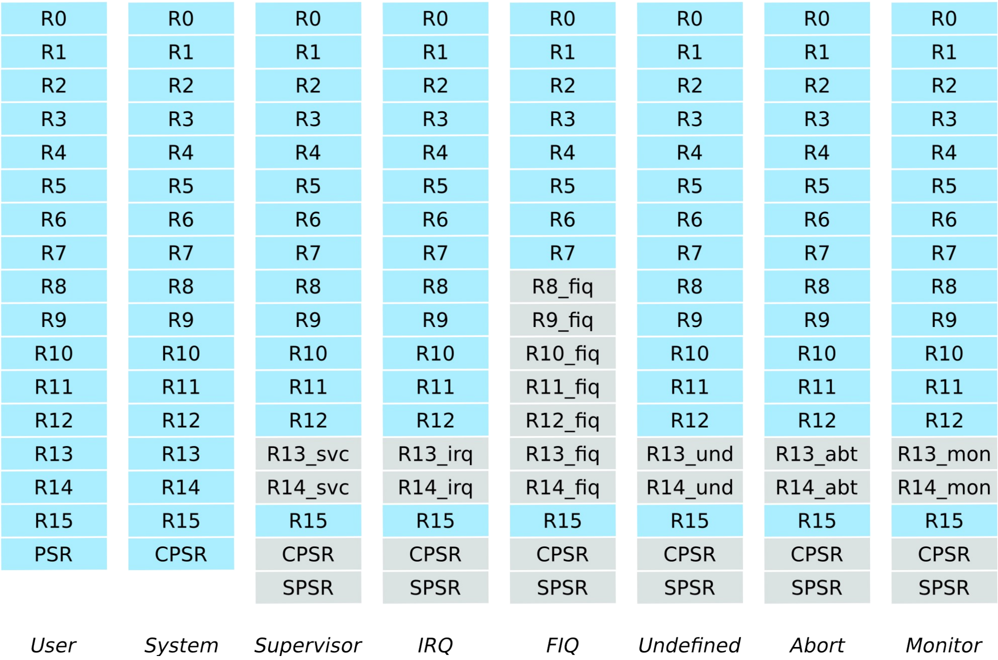
The Cortex-A9 incorporates a number of interesting technologies such as NEON, Thumb-2 and a subset of the ARMv7a security extensions including TrustZone. It does however not include the virtualization extensions that are present in newer versions such as Cortex-A15.

The Cortex-A9 CPU implements the classic 32-bit ARM architecture. It uses a flat 32-bit memory addressing mode and contains 16 32-bit general purpose registers (R0 to R15) in addition to a status register (PSR). The CPU can operate in a number of different state, some of which have higher privileges than others.

Mode	*Privileged?*	*Entered by*	*Remarks*
		User modes	
USER	No	-	User mode, the normal execution mode
		Classic privileged modes	
FIQ	Yes	Fast interrupt	
IRQ	Yes	Interrupt	
SVC	Yes	SVC	Supervisor mode
ABT	Yes	Access error	Abort: data or pre-fetch abort
SYS	Yes	-	
UND	Yes	Invalid instructions	
		Hypervisor extensions	
HYP	Yes	HVC	Hypervisor mode (not used in this work)
		Security extensions	
MON	Yes	SMC	Monitor call, TruztZone entry. Also entered by secure IRQ/FIQ/aborts

Table 1: The ARM CPU states. Note that the hypervisor mode is not available in out target platform.

Some of the CPU register are *banked* in some CPU states. For example, when the CPU is running in the *user mode*, the stack pointer register (R13) is actually the program counter for that specific mode (R13_user). When an interrupt arrives, the CPU will switch to IRQ mode and start using the R13_irq register instead. This is in contrast to some other architecture such as x86, where the stack is used to store main registers during mode-changes.



User	System	Supervisor	IRQ	FIQ	Undefined	Abort	Monitor
R0	R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8_fiq	R8	R8	R8
R9	R9	R9	R9	R9_fiq	R9	R9	R9
R10	R10	R10	R10	R10_fiq	R10	R10	R10
R11	R11	R11	R11	R11_fiq	R11	R11	R11
R12	R12	R12	R12	R12_fiq	R12	R12	R12
R13	R13	R13_svc	R13_irq	R13_fiq	R13_und	R13_abt	R13_mon
R14	R14	R14_svc	R14_irq	R14_fiq	R14_und	R14_abt	R14_mon
R15	R15	R15	R15	R15	R15	R15	R15
PSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR	SPSR	SPSR	SPSR	SPSR	SPSR

Illustration 3: Banked registers in ARM.

The ARM architecture defines a unified method to access up to 16 co-processors (CP0-CP15). The ARM specification reserves two of these for debugging (CP14) and system control (CP15). The latter is of most important when designing operative systems or hypervisors as it among other thing controls caches and memory management hardware.

Memory management in ARM

Protection of system memory is a central part of hypervisor design. It is achieved by configuring the Memory Management Unit (MMU), which is logically placed between the unmanaged CPU core and the physical memory (including memory mapped I/O registers) and translates physical memory into a virtual memory representation based on a mapping table.

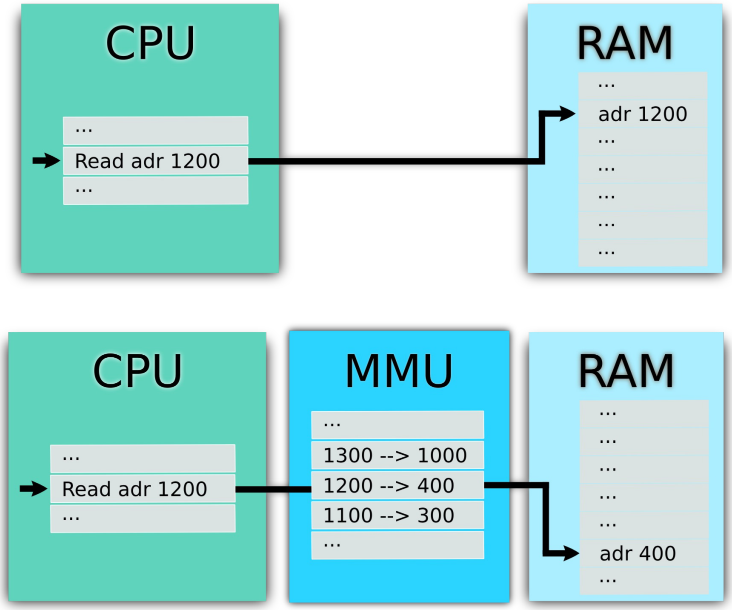


Illustration 4: Memory access with (top) and without memory translation using MMU (bottom). The MMU with help of a translation table, changes accessed address from 1200 to 400.

MMU in ARM architectures that are of interest to us (ARMv5, v6 and v7a) can be configured in different setups. The one of interest to us is the classic two-level page tables with 4KB pages. In short, the mapping between the physical and virtual memory is defined by a level-1 page table containing 4096 entries each pointing to zero or one level-2 page table containing 256 entries. This creates a mapping with a granularity of $2^{32}/(4096 * 256) = 4096$ bytes in the memory, a 4KB *page*[1].

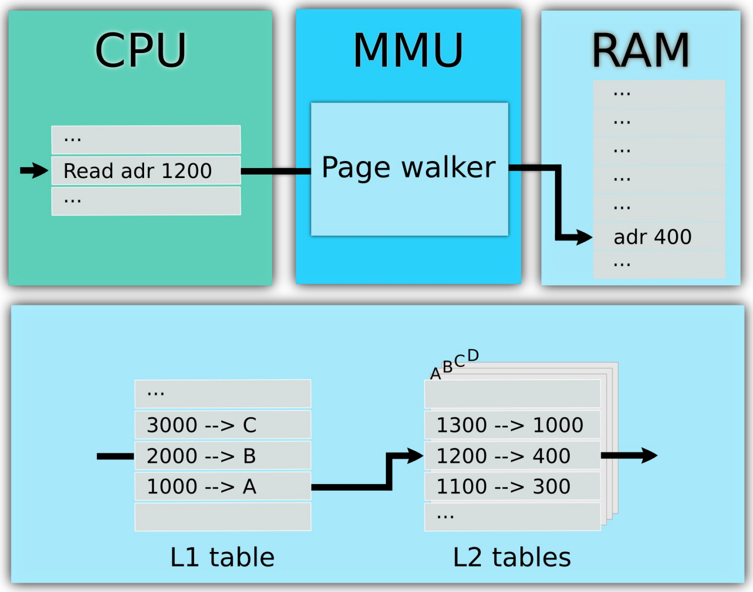


Illustration 5: MMU memory access with two-level page tables. The "walker" inspects L1/L2 tables to find a suitable mapping.

[1] But for better readability, illustrations in this document mostly use L1 and L2 tables of granularity of 1000 and 100.

When the CPU tries to access a memory location, the virtual address is converted to a physical address by performing a page walk, i.e. by looking at the page table structures. Since this process is costly (each CPU memory access requires up to two additional memory reads), a simple cache called Translation Look-aside Buffer (TLB) is used to store some previous page walks. In ARM this is all performed entirely in hardware, while in some other architecture such as MIPS the TLB is filled and maintained by software.

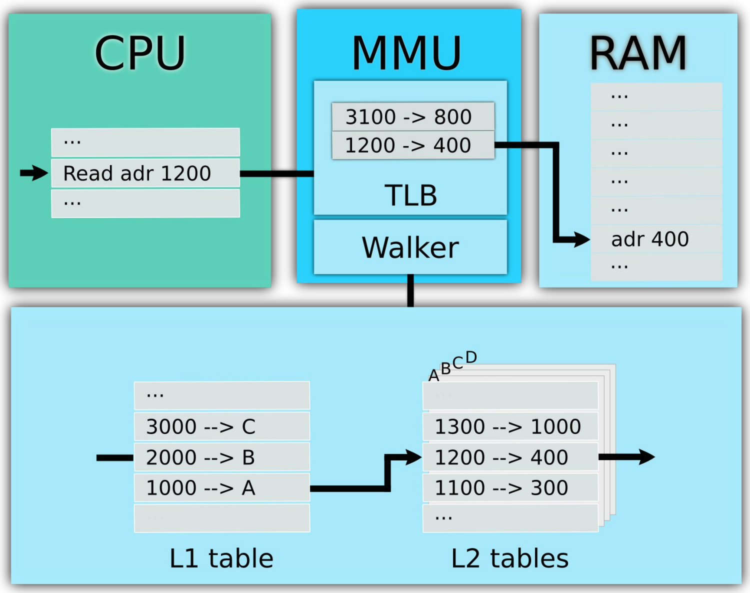


Illustration 6: Memory management unit with TLB, which acts as a translation cache.

Mappings for some memory locations may be invalid or inaccessible for various reasons (such as a privilege level mismatch). In this case as the CPU reads from or writes to these locations an *access abort* is generated. This redirects the CPU to an access abort handler[2] while changing the CPU state to *abort mode*. The handler then decides how to continue, for example by terminating the offending process or by creating the required mapping.

One important detail in memory management is that modifying page tables does not immediately affect the active memory mapping. The TLB takes precedence over the page tables so to be 100% sure that page tables changes are taken into account one must invalidate the TLB data. Also, as seen in Illustration 7, in some systems the cache operates on virtual addresses[3] and must be invalidated when the address mappings change.

[2] In ARMv5/6/7a, two such handlers exist, one for invalid data access (data abort) and one for execution at invalid memory address (pre-fetch abort).

[3] This arrangement is called Virtual Indexed Virtually Tagged (VIVT).

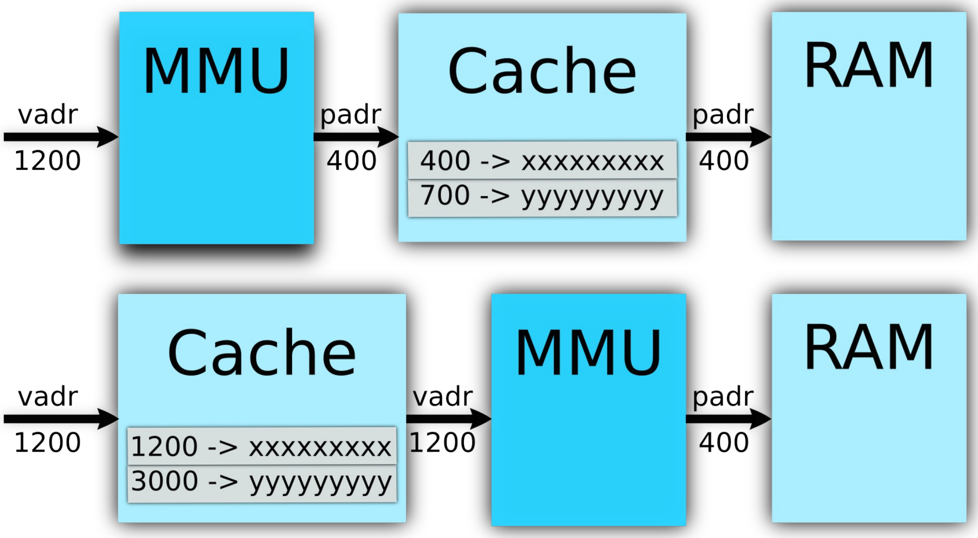


Illustration 7: Different cache placement strategies: ARMv6 and ARMv7 (top) compared to ARMv5 (bottom).

Due to these cache and TLB issues, modifying the memory mapping is a very costly operation and must be avoided as much as possible. In fact, performance of operative systems and hypervisors are often closely related to how good they are at avoiding invalidating the cache/TLB.

TrustZone technology

The TrustZone security extension provides a simple method for adding yet another isolation layer to the processor [3]. It divides the CPU state into *Secure* and a *Normal* “worlds”. Mode changes between Normal and Secure worlds are performed by a minimal software component called the Monitor.

Note that the two TrustZone worlds exists in parallel to the traditional ARM CPU modes. The Secure world maintains its own set of state registers and its own virtual memory configuration.

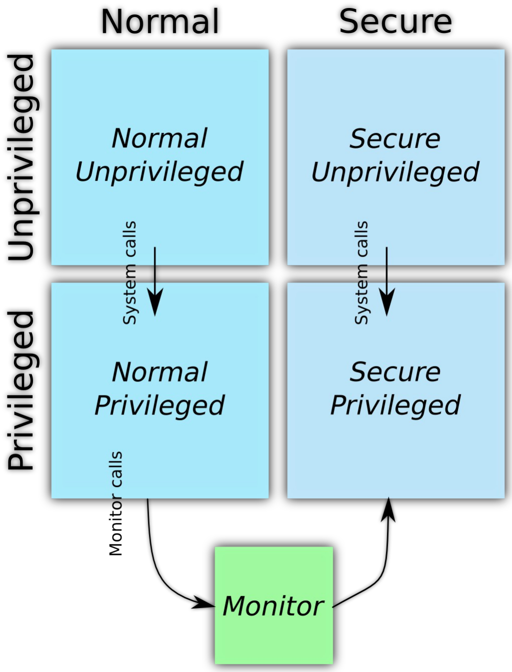


Illustration 8: ARM TrustZone.

GlobalPlatform

GlobalPlatform is a non-profit association, which among other things publishes various specifications for security in embedded system. The specification that is of interest to us in this project is the specification for Trusted Execution Environment in devices. In particular, we are considering the following parts of the specification:

- TEE System Architecture v1.0 [4]
- TEE Client API v1.0 [5]
- TEE Internal API v1.0 [6]

Which together present a framework for secure execution of Trusted Application within the Trusted Execution Environment and define a standardized way to communicate with them. The GlobalPlatform software architecture is pictured in Illustration 9.

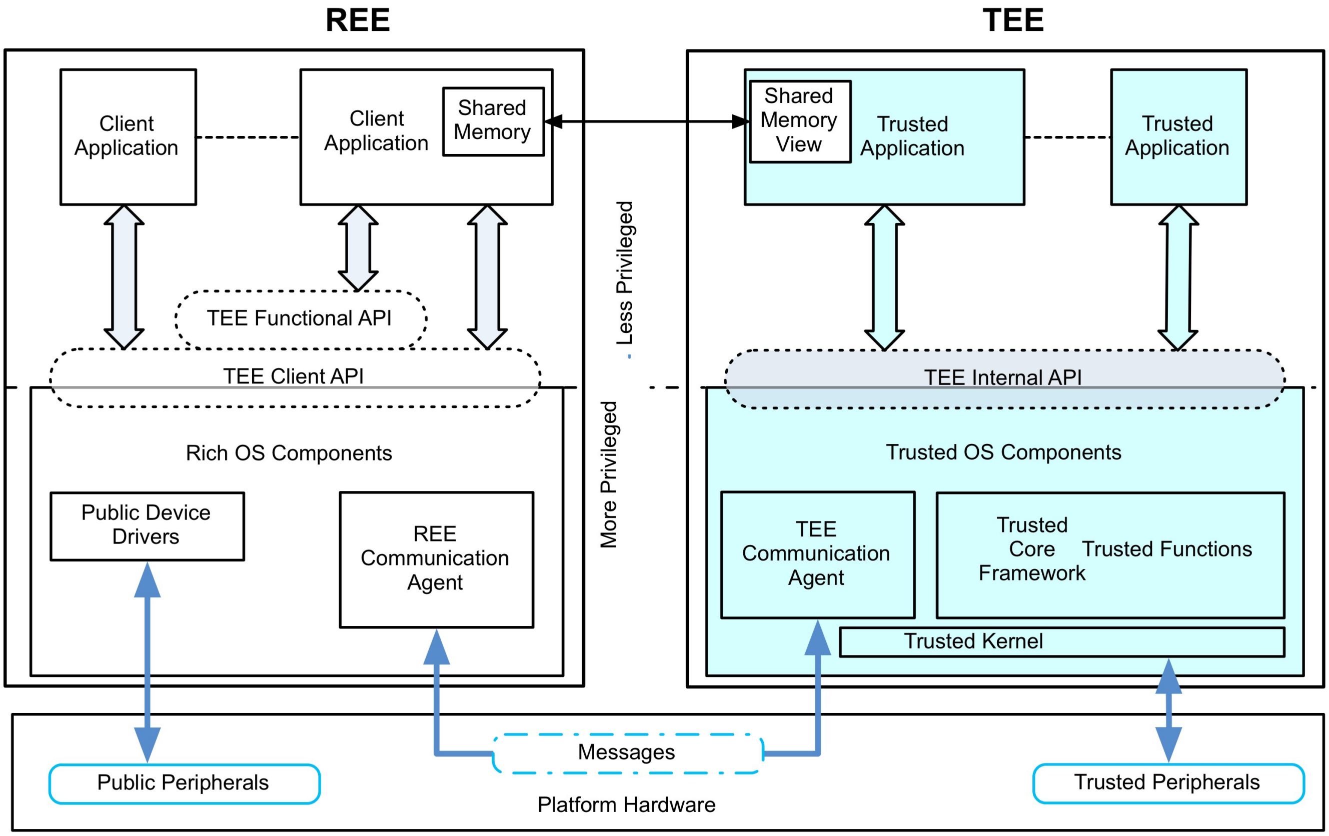


Illustration 9: GlobalPlatform TEE Architecture (source: [4])

The two basic environments are the Rich Execution Environment (REE) and the Trusted Execution Environment (TEE). The REE is were the normal operating system e.g. Linux/Android runs. The TEE acts as a lightweight trusted operating system, running only verified and trusted applications. A Client Application (CA) running in the REE would connect to a TA using the TEE Client API, which is a low level functional interface to allow a CA to access and exchange data with a TA running in the TEE. The communication between the REE and the TEE is handled by the communication agents over a channel which is strictly controlled by the TEE.

The Client API

When a CA would like to utilize the services of a TA, it will connect to that TA and create a session. A session is a way to logically link a chain of command to a TA. The session has its own state and context which the TA can use to bind together different commands from the same CA.
The GlobalPlatform specification does not specify the available commands that the TA should respond to but leave that up to the TA developer to define. The Client API provides the means to execute those commands from the CA in the REE, passing parameters and results back and forth.

The Trusted Execution Environment

Within the TEE, the GlobalPlatform architecture identifies three major components [4]:

1. The Trusted Applications, which make use of the TEE Internal API.
2. The TEE Internal API library implementation.
3. The Trusted OS components, which are shared amongst all TAs, and whose role is to handle the system level functionality required by the TEE.

The TEE Internal API consists of five main services:

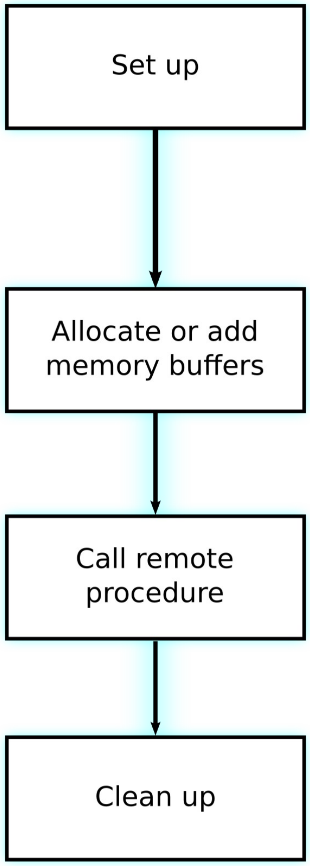
1. The Core API which handles the memory management, session-setup and command invocation from the CA to the TA.
2. The Trusted Storage API, which provides a confidentiality and integrity protected storage on a per TA basis.
3. A Cryptographic Operations API, which supports both symmetrical and asymmetrical cryptography as well as hash functions.
4. A Time and Date API.
5. An Arithmetical API, which allows the TA developer to implement proprietary algorithms.

The TEE Internal API is the only interface for the TA to the rest of the system. It cannot access any hardware resources on its own, and there is no notion of dynamic libraries in the TEE. It is possible for a TA to become a client to another TA. In this case, the calling TA acts much in the same way as a normal CA from the REE, and have to use the same TEE Client Protocol. The caller cannot directly call functions in the callee, but must use the Remote Procedure Call scheme defined by the specification.

Remote Procedure Calls

In GlobalPlatform, a Remote Procedure Call (RPC) is a well defined method for performing calls into other applications, possibly in a different security domain. As secure RPC is the basis of all security services, the specification takes special care of the defined RPC mechanism. In fact, the Client API is essentially a very detailed definition for secure RPC.

Somewhat simplified, an RPC between Client and Trusted Applications has the following structure:



1. **InitializeContext**: a context is created
2. **OpenSession**: A session in attached to this context. The user may also specify the receiving TA and user credentials here
3. The user now declares the shared memory to be used during RPC calls
   1. **AllocateSharedMemory**: A new buffer can be allocated specifically for RPC calls
   2. **RegisterSharedMemory**: The user can also declare any of its own memory buffers to be used for RPC
4. **InvokeCommand:** The user now invokes any number of synchronous RPC calls with the intended parameters, such as the shared memories previously registered
5. **ReleaseSharedMemory/CloseSession/FinalizeContext**: Before exiting, the user should clean up by releasing the allocated buffers, session and the context.

Each InvokeCommand call carries up to 4 parameters of one of the following types:

- **VALUE**: A 32-bit value
- **MEMREF_WHOLE/PARTIAL**: A memory buffer allocated for RPC.
- **MEMREF_TEMP**: A user memory buffer

The direction of these parameters are also declared by the caller. For example *TEEC_VALUE_OUTPUT* is a 32-bit value that is written by the callee while *TEEC_MEMREF_TEMP_INOUT* is a memory location that is written to/read from by both the caller and the callee.

This additional information allows the TEE, as the RPC provider, to examine and process RPC and its parameters without any prior knowledge about the functions that are being called. This will be used in later chapters in this document to provide similar functionality using a hypervisor.

Hypervisors

Virtualization of hardware is a method to provide “virtual” hardware resources (such as CPU or RAM) to software. This virtualized hardware may be called a *virtual machine* and the software operating this hardware is called a *hypervisor* or a *Virtual Machine Monitor [7]*. The software running on the virtual machine is called the *guest* software.

Uses of hypervisors

Virtualization have many benefits. The most obvious benefit is probably the ability to simultaneously run multiple systems on a single CPU. Another benefit is the ability to migrate software between different identical machines, which is often used in data centers to minimize number of active machines and consequently electricity costs.

There are other interesting uses of hypervisors that are probably more apparent in embedded systems. For example, hypervisors can be used to abstract hardware. One may run identical software on different hardware platforms and use the hypervisor as a *hardware abstraction layer*. This has important economical benefits: when upgrading hardware, vendors can shorten development and testing time and avoid re-certification. Manufacturers may also employ virtualization to easily replace end-of-life components with more modern components. Another advantage of hypervisors in embedded system is that designers can use hypervisors to “patch” hardware bugs. For chip designers, this may translate to significant economical wins by reducing NRE costs and improving time to market.

The focus of this this project however is on security properties of hypervisors. By providing *isolation* between different software components, we use a hypervisor as a very powerful *security enabler*.

Different properties of hypervisors

As seen in Illustration 10, Hypervisors exists in two flavors: type 1 hypervisors which run directly on the bare metal hardware and type 2 hypervisors which are themselves hosted in an operative system. For example, XEN is a type 1 hypervisors while Oracle VirtualBox is a type 2 hypervisor. The operative system running below a type 2 hypervisors is called the *host* OS.

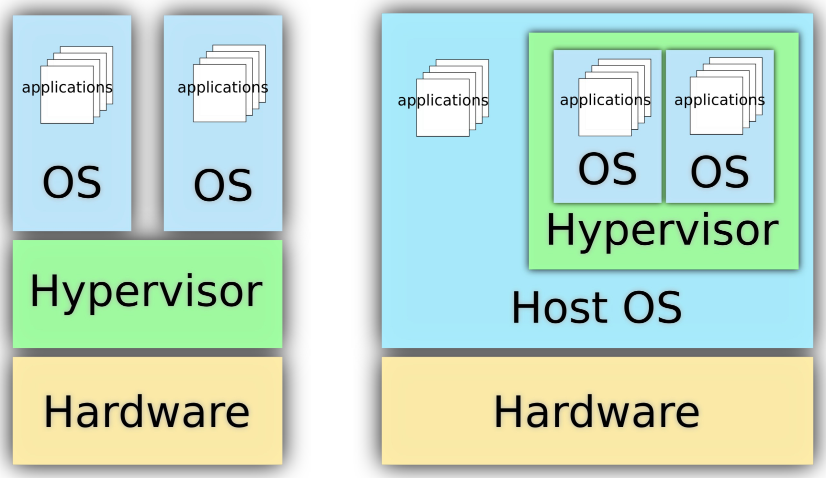


Illustration 10: Type 1 hypervisors (left) run directly on hardware while type 2 hypervisors execute within a host OS

Type 1 hypervisors are normally utilized in resource constrained embedded systems and in data centers where a host OS is not available or needed. Type 2 hypervisors are mostly used on personal computers, for example to run legacy software.

Hypervisor privilege levels

Popek and Goldberg define three interesting properties of a hypervisor [8]:

1. Fidelity: the virtualized software should behave essentially identical to that when running without a hypervisors.
2. Safety: the hypervisor must have full control over the virtualized resources (i.e. the virtualized software cannot directly change the hypervisors or the underlying hardware).
3. Performance: as many instructions as possible should be executed without the intervention of the hypervisor.

The safety requirement of Popek and Goldberg have some important impacts on the implementation of hypervisors. First and fore most, a hypervisor must run in a higher *privilege level* than the guest software. Assuming that the guest software consists of a guest operative system and its applications, ideally three privilege levels are required. This is sometimes not available, thus one may need to integrate guest OS and applications into the same privilege level, as shown in Illustration 11.

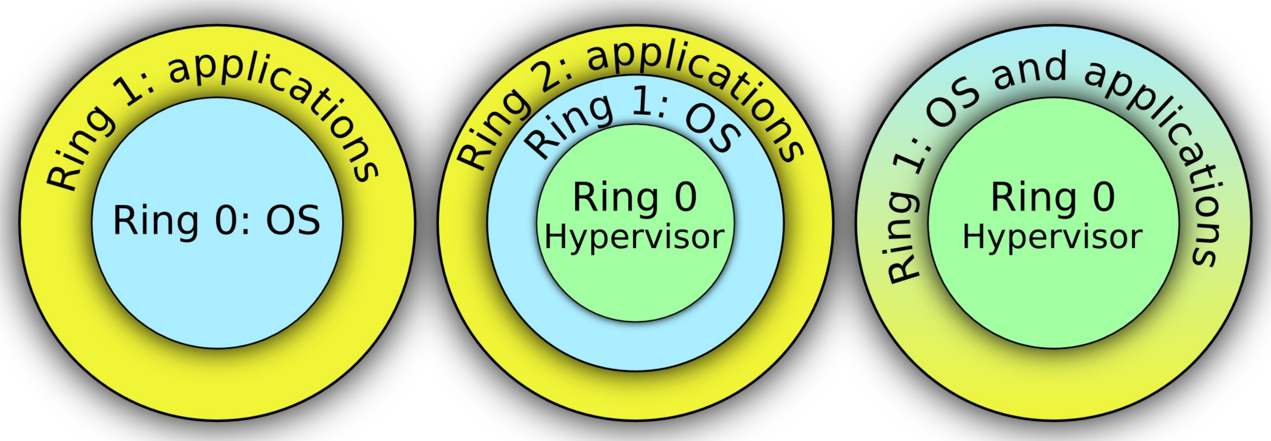


Illustration 11: Rings of power and privilege separations. Normal execution with two privilege levels (left), virtualized execution with three privilege levels (middle) and a virtualized solution with only two privilege levels (right).

Sensitive ("dangerous") CPU operations should be exclusive to the privilege level of the hypervisor. Some examples of such operations are

- operations modifying the memory protection
- operations modifying the privilege levels
- operations modifying the entry and exit to other privilege levels
- operations modifying how interrupts, aborts and system calls behave

Sensitive operations can generally be divided into privileged memory access (access to memory containing sensitive data or I/O-registers) and privileged instructions (instruction only available to certain privilege modes). The former is easy to handle in modern CPUs by means of memory management units. Privileged instructions however can be a bit harder to virtualize.

Traditionally, sensitive operations have been limited to the operative system with the applications requesting such operations through a system call. With virtualization, it is the responsibility of the hypervisor to perform sensitive operations on behalf of the OS. There are multiple approaches to this:

1. binary translation: the guest software is at runtime translated to another software where sensitive operations are replaced with other operations or with calls to the hypervisor
2. para-virtualization: the guest software is modified at compile time in such way that sensitive operations are replaced with calls to the hypervisors
3. full virtualization: guest software executes on the hardware at a lower privilege level. Attempts to perform sensitive operations are detected and forwarded to the hypervisors which then takes appropriate action.

The last approach is closest to the Popek and Goldberg definition of a hypervisor. In practice however, it may be the least efficient (e.g. due to exception handling cost) and secure approach (e.g. due to not all unprivileged operations being "safe").

Hyprevisor operations: context storage, hypercalls and RPC

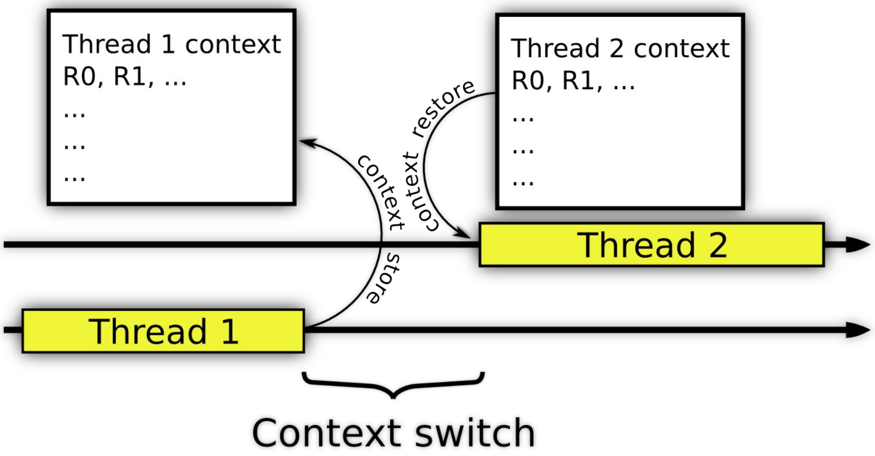


Illustration 12: Context switching between two threads.

State of any software running on a CPU is identified by its memory in addition to the internal state of the CPU. This generally translates to the accessible contents of its address space in addition to the CPU registers[4]. In a time-sharing system, one switches between different software by replacing the state of a software by that of another one. This operation is generally referred to as *context switching*.

Normally, the operative system handles context switching between applications. In a virtualized environment, this task is at least partially performed by the hypervisors. Furthermore, the hypervisor handles context switching between guests themselves if more than one is present. Hypervisor performs context switching during scheduling but may also at other occasions such as during interrupts, exceptions and *hypercalls*.

	The guest communicates with the hypervisor by means of *hypercalls*. This is normally identical to the mechanisms of performing system calls if the system was running without a hypervisors. On the classic ARM architecture hypercalls translate to a software interrupt using the assembler instruction "SVC" (Supervisor Call)[5]. This

[4] For the sake of simplicity, we will ignore state of the co-processors, caches and so on for now.

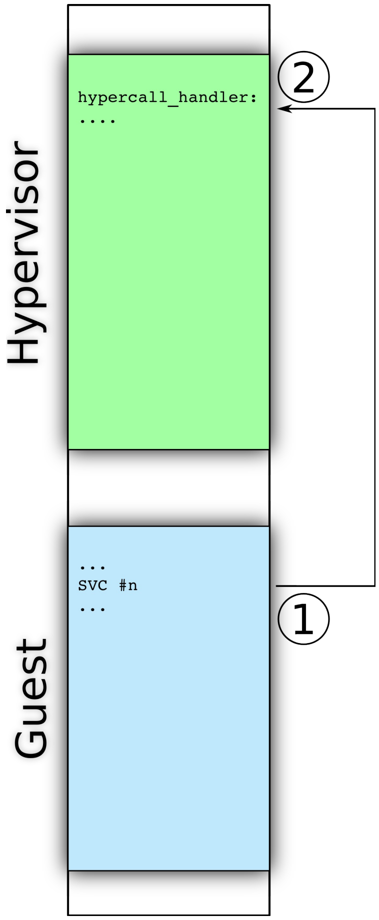


Illustration 13: A hypercall from the guest into the hypervisor.

instruction can be given a 24-bit parameter, for example to identify the type of the hypercall. Additional parameters can be passed along in the general purpose registers.

Illustration 13 Outlines a hypercall sequence. Initially, when at (1) the CPU is in an unprivileged state. After the user executes the SVC instruction, the CPU state is changed to privileged mode (supervisor mode, SVC, to be exact) and the execution flow is redirected to the hypercall handler defined by the hypervisors (2). Here, the hypervisor may store the context of the calling guest for later use.

Exactly what a hypercall means is defined by the hypervisor API. Normally, a number of hypercalls are dedicated for executing privileged operations that the guest OS no longer is allowed to perform by itself (e.g. enabling/disabling interrupts or modifying virtual memory). Furthermore, one or more hypercalls are used for performing Remote Procedure Calls (RPC) between different applications and guests.

Often, calls from the hypervisors into the guests (see this as a reverse hypercall) are performed as an RPC from the hypervisor to the guest. For example, interrupts and access aborts can be forwarded to the guest OS as an RPC. This would mean that a large number of hypervisors operations revolve around RPC, hence the RPC implementation is significant to the security and efficiency of the hypervisor.

[5] On platforms supporting virtualization extensions, there is also a dedicated hypercall instruction.

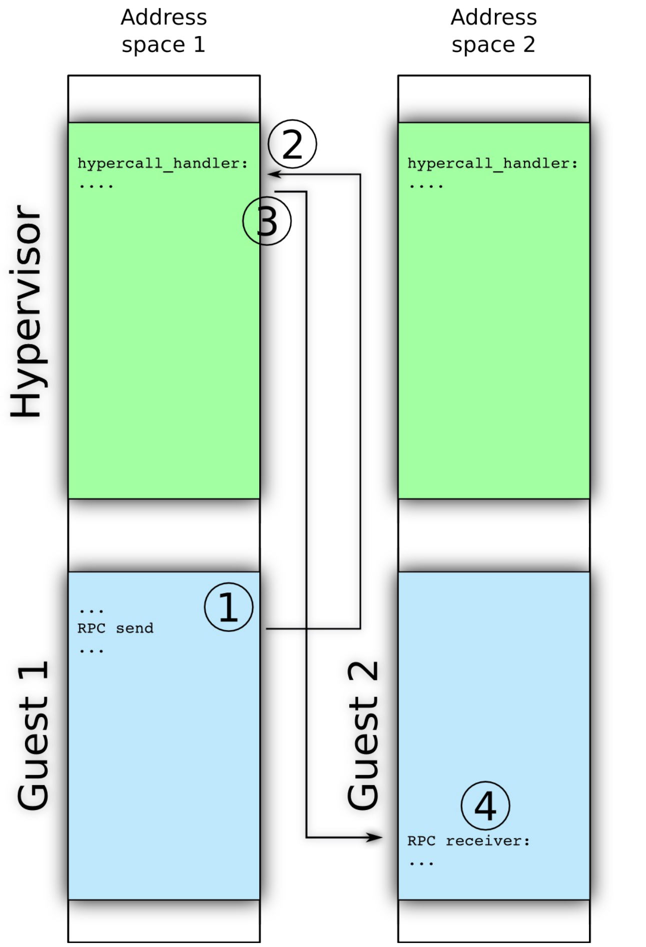


Illustration 14: Communication between guests by using hypercalls. Guest one performs an RPC send (1), which is intercepted by the hypervisor (2). The RPC is forwarded to guest 2 and its RPC handler (3 & 4).

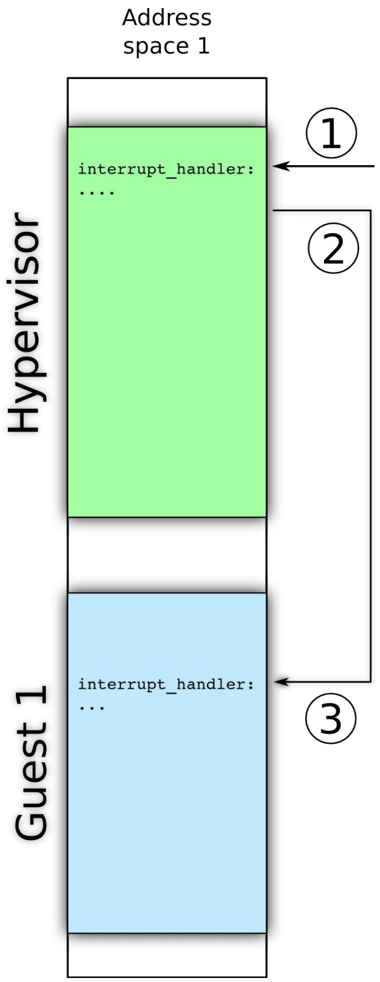


Illustration 15: Interrupt forwarding. A hardware interrupt is received by the hypervisor (1), which forwards it to the guest (2 & 3).

Like any function call, an RPC may be accompanied with a number of parameters. Some such parameters can be stored in CPU registers. However, if the data to be transferred between caller and callee exceeds is too large, the hypervisors must either manually copy the required data between the two parties or utilize shared memory regions. The latter can be performed in a number of different ways

- pre-allocated share buffers: a number of memory sections are reserved for RPC data
- session-allocated share buffers: a number of memory buffers are allocated in advance for a series of RPC calls (an RPC session)
- automatic memory sharing: during the RPC, address spaces of the caller and callee are modified in such way that the required memory buffers from one is available to the other.

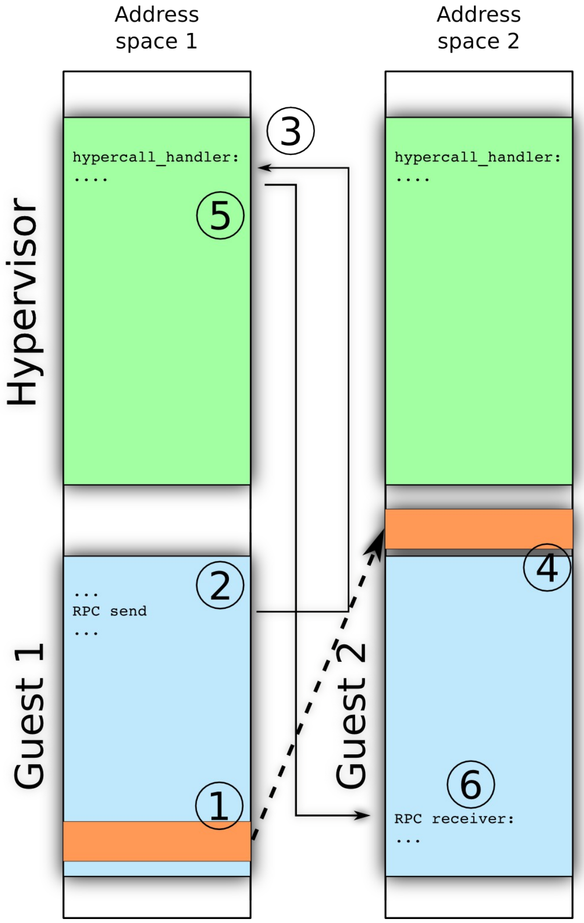


Illustration 16: Mapping of shared memory during RPC calls.

Memory sharing, as seen in Illustration 16: Guest 1 prepares some memory area to be used with the RPC (1). When guest 1 initiates the RPC (2), the hypervisor intercepts it (3) and maps the required memory into the memory space of the receiving guest (4). The hypervisor then redirects the execution flow to the receiving guest (5 & 6).

The STH-TEE Implementation

One of the main objective with this project was to demonstrate a new architecture where Trusted Applications can be isolated from each other using a hypervisor-based solution. The benefit would be that if the hypervisor is small enough to be formally verified, we would have a formally verified separation of the TAs. This means that we will not specifically be running different operating systems and applications on top as virtual machines but the TAs themselves would run directly on a virtual machine. Since the hypervisor in itself does not provide much application support, there is a need for an application framework to support the trusted applications. As mentioned earlier, good choice for such a framework is the GlobalPlatform TEE framework [5].

The SICS Thin Hypervisor

The SICS Thin Hypervisor (STH) is a minimal type 1 hypervisor for embedded systems on the ARM architecture. The STH differs from other hypervisors in a number of key areas

- The code is very small (between 8 and 32 KB depending on the configuration). The small code base makes development and verification very easy. In fact, STH is currently in the process of being formally verified.
- Performance is relatively high, the overhead introduced by virtualization is usually negligible.
- It has been designed to function on any CPU that implements an MMU hence it can function on old hardware that do not naively support hypervisors and virtualization extensions.
- High portability. The hypervisor has a modular design that allows easy migration to new platforms.

As mentioned before, hypervisors can be used as a security-enablers in embedded systems. By adding the hypervisor isolation layer to a system, one can limit the power of the guest OS and/or applications. For example, we have in the past demonstrated use of STH to transparently add memory protection to an unprotected embedded system [9]. Furthermore, given that embedded systems are often cost sensitive, one may use the hypervisors to combine multiple hardware component into a single virtualized component running different guests with possibly different security objectives in parallel.

In this particular project, we will use the isolation layer provided by the hypervisor to ensure that highly sensitive software can securely run on the same CPU as less sensitive software.

Combined GlobalPlatform and STH architecture

To achieve the hypervisor-based separation of TAs they should run as virtual machines on top of the hypervisor. This provides us with a problem regarding how the TEE Internal API and Trusted OS components should be implemented. We could link the complete TEE Internal API library to each TA running in its virtual machine (VM) and have the Trusted OS parts implemented directly in the hypervisor.

This solution, however, has several downsides. Firstly, the memory available inside TrustZone is generally very limited and to have a copy of the TEE Internal API library in each VM would consume a lot of memory. Secondly, if the TEE Trusted OS parts are incorporated into the hypervisor, the hypervisor code base will grow and that makes any formal verification of the separation much harder.

We have chosen to implement the TEE Trusted OS and Internal API in a separate VM, and have the TAs communicate to the TEE Internal API using the generic RPC provided by the hypervisor.

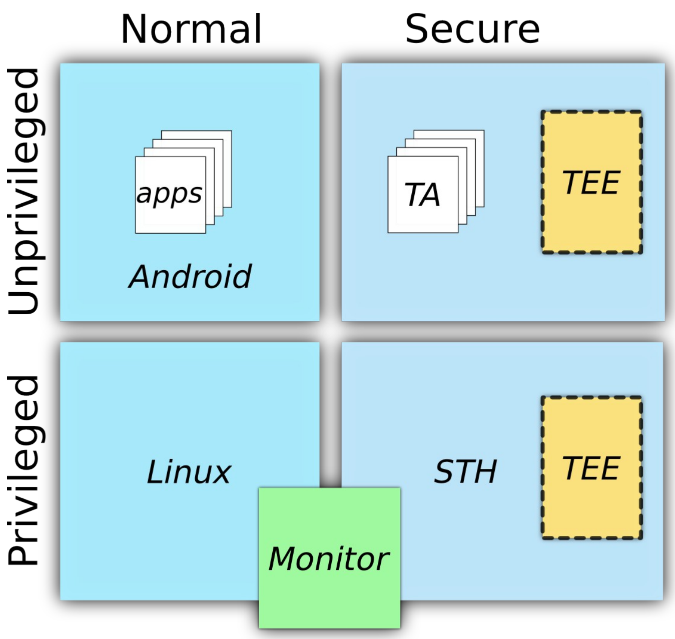


Illustration 17: The combined STH and GlobalPlatform TEE solution. The picture highlights that TEE can be implemented inside and outside of the hypervisor.

The architecture consists of three main data flows as pictured in Illustration 18:

1. from the Client Application (CA) to the TEE Core
2. from the TEE Core to the TAs
3. from the TEE Core to the Hypervisor

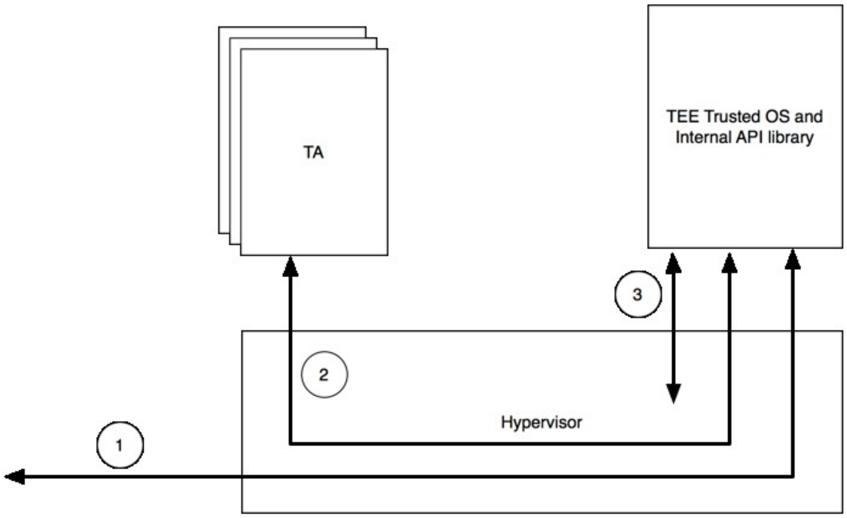


Illustration 18: Architectural data flows.

The channel from the CA to the TEE Core (1) implements the GlobalPlatform Client API. This protocol is used to talk to the TA from the CA. So far we have only implemented the five basic commands: *TEEC_InitializeContext*, *TEEC_FinalizeContext*, *TEEC_OpenSession*, *TEEC_CloseSession* and

TEEC_InvokeCommand. For simplicity, we have not yet implemented any shared memory functionality in the emulator.

The TEE Core handles the bookkeeping of TAs, and the ongoing sessions from the CAs as well as the TEE Internal Core API, as defined by GlobalPlatform. The channel from the TEE Core to the TAs (2) is used to route requests from the CA to the correct TA, based on the session. And also to forward TEE Internal API calls in the TA to the library implementation.

Most of the TEE Core functionality is self-contained in the TEE Core VM but some actions requires the aid of the hypervisor. For instance the loading of a TA. The hypervisor will get a system call (a hypercall) utilizing channel (3) from the picture above. For example, the TEE can ask the hypervisor to load a specific TA and the hypervisor will answer back with an RPC address for the newly loaded TA. This address is then used by the TEE in the communication over channel (2). Conversely, when the hypervisor needs to inform the TEE that some event has occurred, for example an unexpected termination of a TA, it will perform a reversed hypercall.

The TEE emulator

To be able to speed up the development, we started to implement a hypervisor emulator running on a unix platform. The emulator provides the two basic needs; to be able to load other programs (virtual machines) and to be able to relay RPC messages between those programs. It does not handle any scheduling as that is left to the underlying OS. In this way we could more easily develop the TEE Core and the test TAs and CAs.

The emulator uses TCP sockets to provide both the RPC and the hypercalls. The TAs and the TEE are single threaded to allow for a more easy deployment on the target hardware. Thus, requests coming from a CA or Internal API calls from a TA into the TEE are queued and served on a FIFO basis. The main TEE workflow is pictured in Illustration 19.

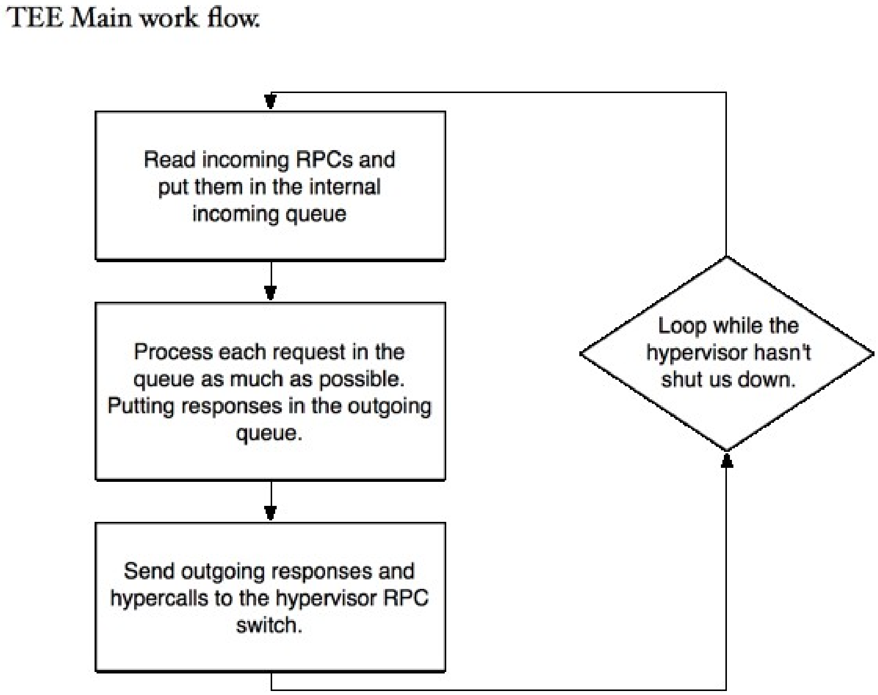

TEE Main work flow.
Read incoming RPCs and put them in the internal incoming queue
Process each request in the queue as much as possible. Putting responses in the outgoing queue.
Send outgoing responses and hypercalls to the hypervisor RPC switch.
Loop while the hypervisor hasn't shut us down.

Illustration 19: Main TEE Work flow.

Some requests cannot be fully served directly. For example an Open Session request from a CA might trigger a series of events in the TEE where it has do perform a hypercall to ask the hypervisor to load the TA for which the session is intended. Hence, we process each request as much as we can in each iteration of the run loop, and when an additional RPC communication is needed we put that request on hold, picking it up again as soon as we have received a response.

Due to the serial nature of the communication channels, each function call and its parameters and return value must be serialized before sending it over the RPC channel. This of course slows down the implementation but we did not have time to implement a robust shared memory architecture.

Memory organization in STH

The STH runs in privileged mode and executes the guest OS and applications in unprivileged mode. The STH is always present in virtual memory but only accessible in privileged mode.

The STH makes heavy use of Address Space Identifiers (ASID) to minimize address space switching cost. In the ARM architecture this translates to ARM memory domains, which are 16 in number. Memory domains can be connected to different sections of the virtual memory and be enabled/disabled individually. This allows the hypervisors to hold multiple guests or applications in the same address space as the hypervisor and switch between these without any need to perform expensive TLB and (in some cases) cache flushing operations.

An example is seen in Illustration 18: At each point in time, the MMU domain of exactly one guest is active. The hypervisor is always active in a privileged domain.

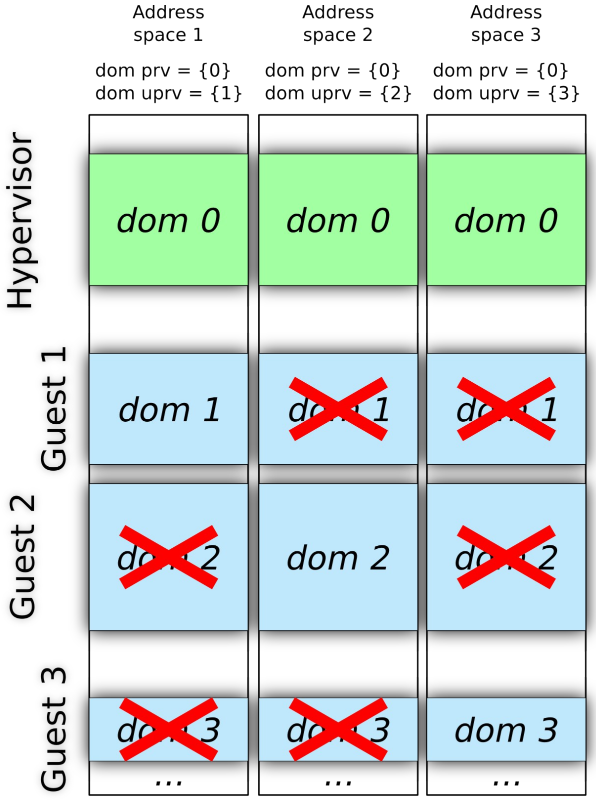


Illustration 20: Multiple guests sharing a single memory map by using MMU domains.

The target device for this project contains a small amount embedded memory which can be used for security sensitive applications (since the data never leaves the chip) or low-power or multimedia applications. We decided to place the hypervisor within 256KB of this memory, which should leave enough space for other application and still provide us with enough memory to experiment. This memory is mapped at the top of the virtual address space. The current hypervisor implementation has the memory configuration shown in Illustration 21.

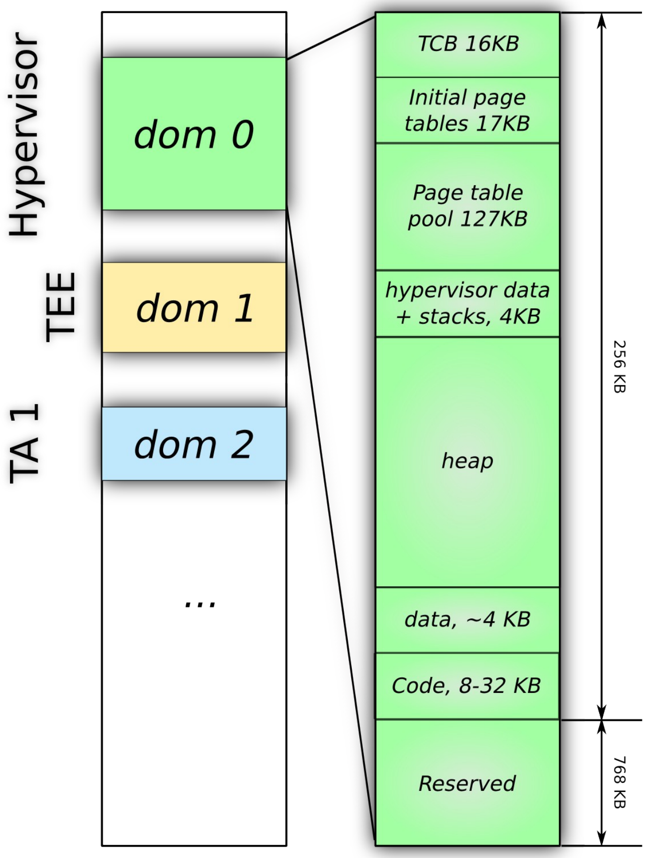


Illustration 21: Hypervisor internal memory configuration.

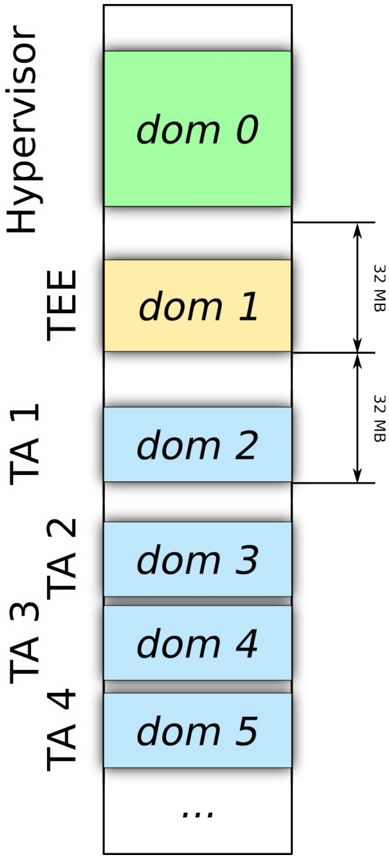


Illustration 22: Suggested GP memory configuration.

For the sake of simplicity, we have assumed that trusted applications require less than 32MB and are not more than 8 in number. This allows us to squeeze the whole GlobalPlatform implementation into a single page table. To switch between the hypervisor, the TEE kernel and the trusted applications, one only needs to enable / disable the corresponding MMU domains with no need for updating the page table or clearing the TLB or flushing the caches.

Memory management within the hypervisor

The STH needs to perform a number of memory operations to function and to serve its guests. In this particular implementation, we have divided the memory operations into three types:

1. Dynamic memory used by the hypervisor itself.
2. Memory that must be physically aligned to a page or a fraction or a multiple of it.
3. Memory databases for mapping physical and virtual memory.

As these operations are crucial to the safety of the system, we decided to keep them as simple as possible. This would allow us to document and verify all memory operations with ease.

Dynamic memory in STH

An efficient implementation of a heap allocator (i.e. malloc() and free()) can be quite large and complex. In fact, the one currently used in GNU libc is larger than the entire STH [10]. Given that dynamic memory is rarely used in the hypervisor and that maximum performance is of less importance to us, we decided to instead create our own very simple and small heap allocator.

The STH heap allocator is based on the simple "reservation list" design outlined in [11]. We use a *first-fit* approach and do not allow the heap to grow or shrink after initialization. The location of STH heap is defined by the static memory map shown in Illustration 21.

While the above limitations result in a somewhat inefficient heap allocator, it is also more than an order of magnitude smaller than traditional heap allocators. This in turn makes verification and security evaluation much easier.

Physically aligned memory in STH

Some memory buffers must have certain physical attributes. For example, level-1 page tables must align to 16 KB boundaries. Such memories can be allocated using the heap allocator, but that can lead to a very bad use of available memory. We decided to implement a separate allocator for this type of memories where some physical pages are allocated during boot and placed into a memory pool. If the pool is dynamic, it will be administrated using a simple bit-set.

The level 1 and 2 page tables, which are 1 and 16 KB in size and aligned with the same amount, are allocated using this approach. Given that the whole pool is linear in physical and virtual memory, the physical and virtual address of the each table can be computed very quickly and without any need for a separate mapping table. This allows a very simple, fast and efficient memory allocation scheme for page tables.

Mapping databases in STH

In order to implement the functionality required for secure memory management, the hypervisor must maintain a number of databases. For example, consider the following hypothetical mapping operations:

```
void *get_more_memory(PROCESS *proc, size_t size) {
      adr_t p = get_free_physical_memory(size);
      adr_t v = get_free_virtual_memory(proc, size);
      add_mapping(proc, p, v, size);
      return (void *)v;
}
```

Listing 1: A hypothetical memory mapping operation.

To manage this, the hypervisor must maintain a number of databases of free, allocated and shared memories in a very complex system of tables and lists with non-trivial dependencies that can cause all sorts of problems. For example, in many modern operative systems the kernel may need to investigate a memory access violation by looking up something in a table that itself is not currently accessible, which results in another memory access violation which starts a chain reaction that sometimes is very hard to understand.

Another common problem is that the memory management and security subsystems are more or less separated. Our hypothetical mapping function above does not receive any information about the purpose of this mapping and can therefore not take any security measures. In the best case, security is handled in a separate step after the call to our hypothetical function...

The STH port for U8500 continues our tradition of prioritizing simplicity and security over performance by using a very simple scheme to maintain the memory mapping databases, while enforcing the *least privilege principle*. To do this, first, we classify physical and virtual memory as follows

```
P_TYPE = { private RAM, private IO, public RAM, public IO, … }
V_TYPE = { Hypervisor, Kernel, Task, … }
V_SUBTYPE = { Code, Stack, Heap, IO, Shared, … }
```

We define some policies for how these types can be associated. For example, private physical memory is exclusive to the hypervisor and Stack/Code/Heap cannot reside inside a physical IO region. We also divide the physical and virtual memory into memory areas:

```
M_AREA = <BASE, SIZE>
V_AREA: P_TYPE → list of M_AREA
V_SUBAREA: (V_TYPE, V_SUBTYPE) → list of M_AREA
```

The physical memory areas are machine dependent and non-overlapping. The virtual memory areas are defined by the system software and also non-overlapping. Finally, the hypervisor maintains a list of mappings for each process:

```
M_TYPE = <V_SUBTYPE, permissions, … >
MAPPING = <M_AREA, V_ADR, M_TYPE>
PROCESS = < V_TYPE, list of MAPPING, AddressSpace, … >
```

This allows us to us to extend our hypothetical mapping function to take into account the *intent* of the operation and ensure the security policies are followed:

```
void *get_more_memory(PROCESS *proc, P_TYPE ptype, V_SUBTYPE vsubtype,
                 size_t size) {
      adr_t p = get_free_physical_memory(proc, ptype, size);
      adr_t v = get_free_virtual_memory(proc, vsubtype, size);
      permissions pm = get_allowed_permissions(ptype, p->V_TYPE, vsubtype);
      add_mapping(proc, pm, p, v, size);
      return (void *)v;
}
```

Listing 2: Extended version of the hypothetical memory mapping operation.

This extension can be used to enforce our security policies. For example, from *get_allowed_permissions()* one may enforce the MMU NS and NX bits (for pages that are in the Normal world and not executable, respectively) and by modifying *get_free_physical_memory() one can* could verify that the requesting process is allowed to access the requested memory area.

While this approach does limit the way memory is used, it will also simplify the implementation greatly. Furthermore, all security policies will be built-in and inherent instead of, as it usually is the case, an afterthought. Another advantage of this approach is that creates a unified method for managing memory of both guests and the hypervisor, which significantly reduces the complexity and the code/memory footprint of the STH.

In fact, such simplification of the memory management components is more or less a requirement if formal verification is to be applied.

RPC implementation in STH

The STH implementation of RPC was chosen to mimic that of secure RPC in GlobalPlatform closely. Recall from previous discussion (page 14) that RPC mechanism in the Client API also specifies format of the parameters. This information is visible to the hypervisor as the RPC provider, hence it can without any prior knowledge about the Trusted Application and the provided functions ensure that parameters are transferred between caller and callee in a correct and secure manner.

Secure transportation of parameters is trivial for value parameters: from the stored contexts, simply copy the involved register in correct direction. For shared memories however, much more work is needed. The STH handles shared memories by temporarily mapping pages into the memory space of callee TA, as previously seen in Illustration 16. Furthermore, the hypervisor can use the page table access flags to ensure that parameter directions are maintained. For example, if a parameter is of the type *TEEC_MEMREF_PARTIAL_INPUT*, then the corresponding temporary page table entries are set to read-only to stop the callee from modifying it.

Unfortunately, user allocated shared memories (i.e. *TEEC_MEMREF_TEMP_xxx*) introduce their own set of security problems. As seen in Illustration 23, such memories are not always aligned to page boundaries and when mapped into the callee address space they may also reveal adjacent memory locations to the callee.

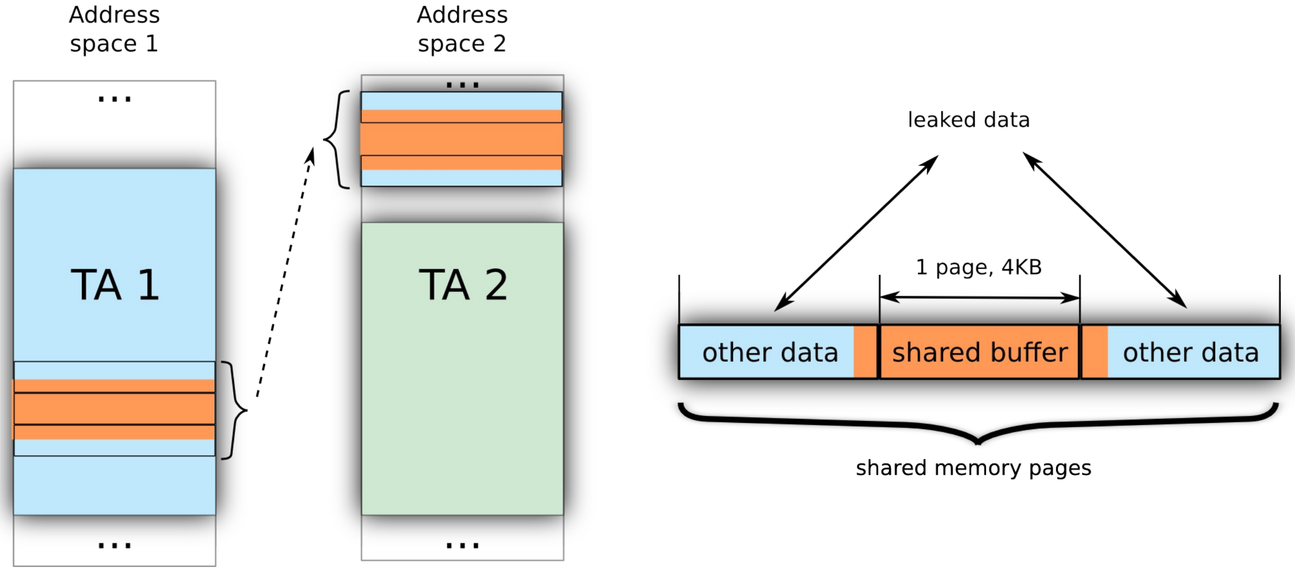
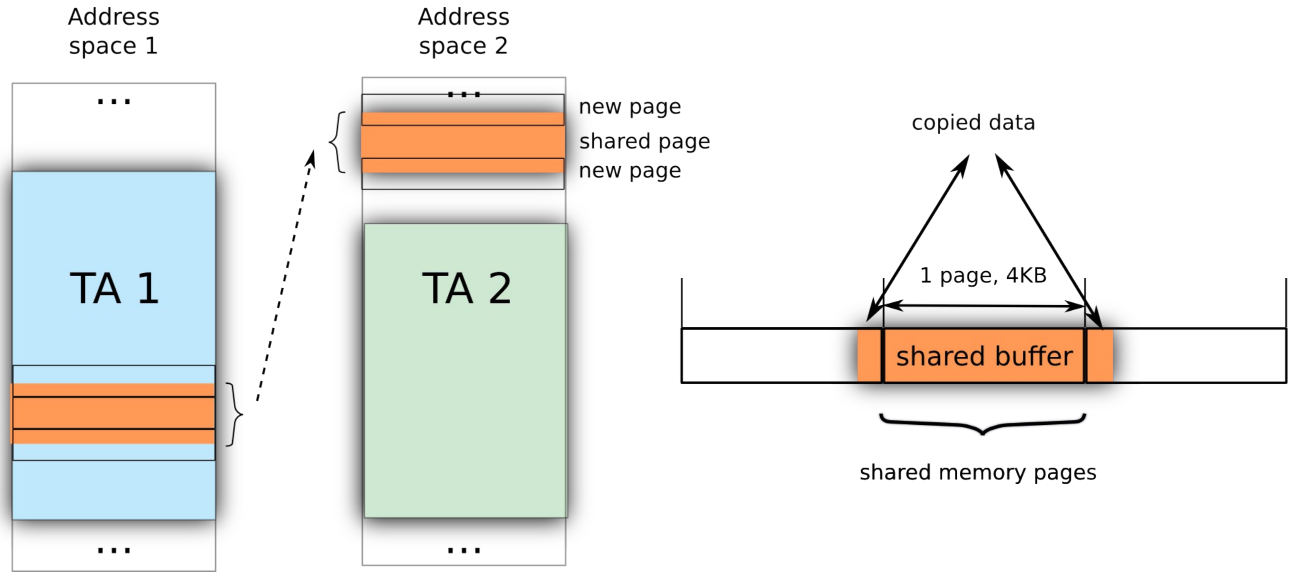


Illustration 23: Data leakage when sharing memory pages.

This can be addressed by creating a copy of the user memory when adjacent memory locations has been erased. This has some performance implication if the shared memory is very large. Fortunately we can utilize some simple optimizations to reduce their effect:

- Copy only pages that contain other data, the rest are mapped as before. This is shown in Illustration 24
- Map and copy lazily: pages are not copied in either direction before they have been read from/ write to by the callee.



Deployment of STH

Normally, the STH is distributed as a single binary that also includes some support functions required to load the first guests into the memory. In most embedded system this binary can be programmed into flash memory and ran immediately at boot.

In this project however, the target platform boots from an internal ROM and continues to load signed binaries from ROM or flash memory. We can not control or modify any aspects of this boot process. The earliest time we can upload and execute the STH is as the bootloader (u-boot) is being activated. At this point of time however, the CPU is in the Normal world.

To be able to run the STH in privileged mode with the Secure world, we needed to perform the following operations:

1. Create a TA that can take over the Secure world TEE and its TA's plus the TrustZone and the structures provided by the ROM code. This must be done in a “live” system.
2. This TA will be able to receive the STH payload from the Normal world and execute in privileged Secure state.
3. Modify the bootloader to load our TA into the original TEE and activate it before Linux is loaded.

Note that all binaries must have been signed with the appropriate keys for this to work.

Summary

This chapter summarizes the project and discusses the problems we faced, lessons we learned and outlines possible future directions.

Problems and obstacles

During this project we encountered a number of problems that need to be mentioned. The first problem was the complexity of the target system, which not only affected our porting efforts but also had some implications on the security of the hypervisor: The U8500 contains a number of components that, if not configured properly, may be used to circumvent the memory isolation enforced by our hypervisor. To solve such issues, one must carefully analyze all SoC components and limit user access to sensitive parts. This was however left out of this project due to time constraints.

Another problem we faced was the high memory usage of the hypervisor. Ideally, one would like to place the hypervisor code and data in embedded SRAM inside SoC to avoid exposing it to the outside world. It was however noticed that only a small part of the tiny embedded SRAM could be allocated to the hypervisors as the remainder was reserved for various multimedia buffers. We have currently limited the hypervisors to 256 KB (this includes a small guest and all page tables), but this is still twice as large as the preferred size for this platform. We are however confident that the hypervisors size can be reduced significantly once a stable development base have been found (the original version of STH required only 32KB of memory).

A much larger problem we faced was incorporating the hypervisor and its guests into a device that already had some security mechanisms in place. Normally, the hypervisor executes firsts of all, and can then configure the system to its liking. In this case however the system was already set up by existing ROM code and subsequent TEE code stored in flash memory. While we eventually found a way to circumvent this, it required a major portion of the allocated project time to get there.

Current status

This project was divided into a number of milestones:

1. Port the hypervisor to the U8500 platform and demonstrate this by running two instances of an RTOS simultaneously in the Normal world
2. Implement an emulator that can be used for development of TA and CA on a desktop PC.
3. Move the hypervisor to the Secure world
4. Create a simple TEE guest and some simple Trusted Applications to run on the hypervisor.

After some small initial problems with the platform (the public documentation we used at this time did not cover some, to the hypervisor, essential information about power management, which took some time to discover), the first milestone was reached without any major issues.

For the second milestone, we have implemented an emulator capable of handling Open and Close Session and Invoke Command requests from the CA to the TA. The TEE supports several sessions from different CAs towards the same TA and several different sessions to different TAs from the same CA.

The Third milestone proved to be harder to reach. Ideally, one would only need to re-program the device new firmware containing the hypervisor. This however turned out to be impossible since, as mentioned earlier, the devices booted of ROM code inside the U8500 that could not be replaced. Furthermore, the ROM code started other software components that in turn set up the TrustZone barriers and shut us out of the Secure world. In order to, in a secure fashion, get the hypervisor past these barriers and into the Secure world we eventually came up with the complex procedure presented at page 30. Nevertheless, the time spent on this milestone was much larger than planned.

At the time of writing this document, we are working towards the fourth milestone. Due to delays in the project we have not been able to test the GlobalPlatform framework within an STH running on real hardware. We hope however to demonstrate a simple TEE guest on top of the hypervisor in the beginning of next year.

Conclusions

After carefully examining the GlobalPlatform architecture, we noted that some parts of the APIs can be built upon a hypervisor. We further noticed that the isolation and RPC mechanisms in GlobalPlatform are of special interest to us and can be implemented on top of an existing hypervisor with relativity small amount of work.

Given that a hypervisor implementation can be quite small and efficient, a hypervisor-based approach to GlobalPlatform may allow high security at a very low cost. To demonstrate this, the SICS Thin Hypervisor was modified to support a simplified Trusted Execution Environment and some simple Trusted Application.

We have not yet performed any in depth analysis of the security of this new system.

Future work

During this project, we learned a great deal about the involved technologies, software and hardware. We believe that at this point, there are a number of interesting directions for continuing this road:

1. An in-depth analysis of the Trusted Execution Environment inside the Secure world provided by the TrustZone and the hypervisor.
2. Formal verification of the STH U8500 implementation to ensure its correctness.
3. Implementation of multiple guests inside the Secure world and upon a single hypervisors. For example, we would like to completely separate the trusted system services (such as SIM services) and secure services provided by third-parties (for example, Digital Rights Management services). If we were to maintain two virtualized guests for fully trusted and “less trusted” services, both confirming to the GlobalPlatform specification, a number of interesting applications would be possible.
4. An implementation that takes advantage of the ARM virtualization extension and a comparison with the current approach.

We also believe that the hypervisor can be used for other purposes than security. For example

1. Implementation of a power saving mechanism that take advantage of virtualization. For example, by transparently moving software components to different types of memory.
2. Use of a hypervisor for testing and diagnostics in complex software stacks. For example, using the hypervisor to override the cache management in Linux and study its effect on various types of software.

Bibliography

[1] ST-Ericsson, "*St-Ericsson NovaThor Platforms - Powering the next generation of mobile devices*". ST-Ericsson fact sheets, 2012

[2] ARM, "*Cortex-A9 Technical Reference Manual*". ARM technical documentation, 2012

[3] ARM, "*ARM Security Technology - Building a Secure System using TrustZone Technology*". ARM technical documentation, 2009

[4] GlobalPlatform, "*TEE System Architecture v1.0*". GlobalPlatform specification, December 2011

[5] GlobalPlatform, "*TEE Client API Specification*". GlobalPlatform specification, July 2010

[6] GlobalPlatform, " *TEE Internal API Specification v1.0*". GlobalPlatform specification, December 2011

[7] J. Smith, R. Nair, "*The architecture of virtual machines*". IEEE Computer 38, 2005

[8] G.Popek, R. Goldberg, "*Formal Requirements for Virtualizable Third Generation Architectures*". Communications of the ACM 17, 1974

[9] H. Douglas, "*Thin Hypervisor-Based Security Architectures for Embedded Platforms*". Masters thesis, Royal Institute of Technology, 2010

[10] D. Lea, "*A Memory Allocator*". Technical Report, State University of New York, 1992

[11] D. Knuth, "*The Art of Computer Programming. Volume 1: Fundamental Algorithm*". Addison-Wesley, 1997

Terminology

Abbreviation	Meaning
ARM	A CPU architecture designed by ARM Holdings
API	Application Programming Interface
GP	GlobalPlatform
ASID	Address Space Identifier. Normally a hardware assisted memory-area tagging mechanism
FIFO	First-in First-out, a queue
SoC	System-on-chip
STH	The SICS Thin Hypervisor
TA	Trusted Application (GlobalPlatform)
CA	Client Application (GlobalPlatform)
TEE	Trusted Execution Environment (GlobalPlatform)
VMM	Virtual Machine Monitor (a hypervisor)
TLB	Translation Look-aside Buffer. Small hardware cache for speeding up memory translation.
RPC	Remote procedure call. Call between processes, guests, etc.

Revision History

Revision	Date	Owner	Notes
1.0	2012-12-01	AV	Document created
1.1	2012-12-7	AV	Created intro chapter
1.2	2012-12-11	AV	Created hypervisor chapter
1.3	2012-12-12	AV	Created system architecture chapter
1.4	2012-12-20	AV+PE	STH/TEE stuff gets its own chapter. So does GP
1.5	2012-12-21	AV+PE	Added STH-TEE picture, added missing references + STH boot
1.6	2013-02-20	AV+PE	Clarified the part about U8500 TEE implementation.
1.7	2013-03-05	AV+PE	Released