Security Services on an Optimized Thin Hypervisor for Embedded Systems

VIKTOR DO

Master's Thesis at SICS
Supervisor: Christian Gehrmann
Examiner: Martin Hell

July 2011

LUNDS UNIVERSITET
Lunds Tekniska Högskola

Abstract

Virtualization has been used in computer servers for a long time as a means to improve utilization, isolation and management. In recent years, embedded devices have become more powerful, increasingly connected and able to run applications on open source commodity operating systems. It only seems natural to apply these virtualization techniques on embedded systems, but with another objective. In computer servers, the main goal was to share the powerful computers with multiple guests to maximize utilization. In embedded systems the needs are different. Instead of utilization, virtualization can be used to support and increase security by providing isolation and multiple secure execution environments for its guests.

This thesis presents the design and implementation of a security application, and demonstrates how a thin software virtualization layer developed by SICS can be used to increase the security for a single FreeRTOS guest on an ARM platform. In addition to this, the thin hypervisor was also analyzed for improvements in respect to footprint and overall performance. The selected improvements were then applied and verified with profiling tools and benchmark tests. Our results show that a thin hypervisor can be a very flexible and efficient software solution to provide a secure and isolated execution environment for security critical applications. The applied optimizations reduced the footprint of the hypervisor by over 52%, while keeping the performance overhead at a manageable level.

Referat

Säkerhetstjänster på en Optimerad Tunn Hypervisor för Inbyggda System

Virtualisering har använts i dataservrar under en lång tid som ett sätt att förbättra utnyttjandet, isolering och drift av datorn. Under senare år har inbyggda enheter dock blivit mer kraftfull, alltmer uppkopplad och kör applikationer på operativsystem med öppen källkod. Det är bara naturligt att tillämpa dessa virtualiseringstekniker på inbyggda system, men med ett annat mål. I dataservrar var det främsta målet att dela den kraftfulla datorn med flera gäster för att maximera utnyttjandet av datorn. För inbyggda system är behoven annorlunda. Istället för att öka nyttjandet av datorn så vill man använda virtualisering för att istället stödja och öka säkerheten genom att erbjuda flera säkra exekverings miljöer för sina virtuella maskiner.

Denna uppsats presenterar design och implementeringen av ett säkerhetsprogram, och visar hur ett tunt virtualiseringslager som är utvecklad av SICS, kan användas för att öka säkerheten för en enda FreeRTOS gäst på en ARM plattform. Utöver detta, så analyserades den tunna hypervisorn för förbättringar med tanke på främst utrymme och prestanda. De utvalda förbättringarna applicerades sedan och bekräftades av profileringsverktyg och prestandatester. Våra resultat visar att tunna hypervisorer är en mycket flexibel och effektiv mjukvaralösning för att tillhandahålla en säker exekveringsmiljö för säkerhetskritiska applikationer. De tillämpade optimeringarna minskade storleken med över 52% samtidigt som prestandan hölls i en hanterbar nivå.

Acknowledgements

I would like to thank my supervisor, Christian Gehrmann, for his support and feedback and for the opportunity to undertake such an interesting thesis. Oliver Schwarz for helping me out with all the technical questions and details. Finally, I want to also thank my family for all the love and support.

Contents

List of Figures

List of Tables

Acronyms

AP access permission

AES advanced encryption standard

API application programming interface

CISC complex instruction set computer

CPSR current program status register

CPU central processing unit

DMA direct memory access

DMAC DMA controller

FIQ fast interrupt request

I/O input/output

IRQ interrupt request

IPC inter process communication

IOMMU I/O memory management unit

MMU memory management unit

OVP Open virtual platforms

OS operating system

RISC reduced instruction set computer

RSA Rivest, Shamir and Adleman

RPC remote procedure call

SHA secure hash algorithm

SICS Swedish institute of computer science

SPSR saved program status register

SWI software interrupt

TLB translation lookaside buffer

VM virtual machine

VMM virtual machine monitor

Chapter 1

Introduction

With the increasing use of computers to handle sensitive and secret information, the issue of trusting a computer to perform a security critical task is becoming increasingly important. Even if the trusted application has been designed with a very high level of security, if the underlying operating system is compromised, any application level protection will become useless. Regrettably, this is often a commodity operating system in where the writer of the trusted application has no control over. Not only does the operating system have full control over the applications, they are also immensely large and complex making it highly vulnerable to attacks. How can one trust one's computer under those circumstances? Previously, security was mainly an issue in personal computers, but because of the rapid increase in both growth and performance in embedded systems, it has become equally important there.

Embedded systems and consumer electronics such as smart phones are now running open and complex operating systems with connections to the outside world. One can also see a huge increase in the embedded software domain with respect to the number of applications and open software, and as expected, there is a clear indication of threats increasing, targeting mobile and sensitive infrastructure devices [8].

There is clearly a demand for a protected environment in which security critical code and data can run isolated. The most direct way to protect a trusted application is to create a completely independent execution environment in hardware with its own memory and processing unit, that should only be accessible to the user through well-defined interfaces. However, this solution tends to be quite ineffective as building an entire secure execution environment in hardware is rather expensive. Secondly, with this setup, it could only keep one application secure and one would usually want to allow multiple stakeholders to run their secure services independently from each other.

This can all be solved with a software solution called *virtualization*. In data servers, virtualization has been used since the 1970s because of its ability to provide multiple isolated execution environments. It is only natural to apply the virtualization techniques to embedded systems, however, with a different approach as the requirement and support for virtualization in embedded systems are quite different.

Data servers strives to increase the utilization of the hardware, while in embedded systems, the focus is put on security through isolation and a smaller trusted code base.

1.1 Goals

This thesis aims to design and implement a security application on an existing SICS developed hypervisor that runs on an ARM platform with a single OS guest. The goal is to demonstrate that the hypervisor can protect the security critical application from malicious software through the isolation properties of the hypervisor. In addition to this, the hypervisor will also be enhanced by improving its footprint and overall performance to better support memory constrained embedded systems. In order to achieve this, these individual goals need to be accomplished:

- Familiarize with the ARM architecture, OVP simulation platform[1] and the SICS developed hypervisor.
- Define a suitable security application that demonstrates the potential power of the hypervisor.
- Implement the selected security application as a secure service upon the hypervisor.
- Analyze the current hypervisor implementation and search for improvements with respect to the hypervisor's footprint and overall performance.
- Implement the identified enhancements as well as verifying them through suitable test suites.

1.2 Thesis Overview

The thesis is organized as follows. Chapter 2 provides background information relevant to the thesis, such as an overview of virtualization techniques, the basics of ARM architecture, and how the SICS developed hypervisor works. Chapter 3 describes the design and implementation of the security services on the hypervisor, including a demonstration of its security. Chapter 4 describes the optimization of the hypervisor and various benchmark tests to confirm the improvements. Chapter 5 presents conclusions for the thesis and future work.

[1] Used to simulate the ARM platform

Chapter 2

Background

2.1 Virtualization

In computer science, the term virtualization can refer to many things. Software can be virtual, as can memory, storage, data and networks. In this thesis, virtualization refers to *system* virtualization in which a piece of software, the *hypervisor* also known as a *virtual machine monitor (VMM)*, runs on top of the physical hardware to share the hardware's full set of resources between its guests called *virtual machines* (VM). Virtualization is not a new concept and has been used for a very long time. It was invented by IBM in the 1960s [12], and at that time, server mainframes were very expensive. To increase utilization, virtualization was applied to make the mainframes sharable between several users and applications. Suddenly, it was now possible to run multiple virtual machines which were exact copies of the underlying host machine. This was revolutionary as the mainframes were now capable to host multiple independent operating systems along with their applications within a single physical machine. But as hardware became less expensive and the x86 architecture server and desktop computers became industry standard, virtualization was almost abandoned during the 1980s and 1990s. However, the growth in x86 servers and desktops soon led to new IT infrastructure and operational challenges such as low utilization, increasing physical infrastructure and IT management costs, and insufficient security and disaster protection. The situation was drastically changed when VMware managed to virtualize the x86 systems in 1999 [27] and the popularity of virtualization has once again been renewed.

2.1.1 Hardware Support for Virtualization

CPU architectures provides multiple operational modes, each with a different level of privilege. For example the x86 architecture provides four protection rings, from Ring 0, the highest privilege mode to ring 3 the lowest. The ARM architecture only has two modes, User and Supervisor mode. These different modes enforce security of the system's resources and execution of privileged instructions.

Generally operating systems are designed to run on native hardware and expect

to run on the most privileged mode in order to take total control over the whole computer system. However in a virtualized environment, the hypervisor will be running in the most privileged mode while the operating system runs in a lower privileged level inside a VM. This complicates matters as the operating system will not be able to execute the privileged instructions necessary to configure and drive the hardware directly. Instead, the privileged insructions are handled by the hypervisor in order to be able to provide the hardware safely to the VM's. Figure 2.1 describes the hypervisor architecture. We have the hypervisor running in the most privileged mode right above the hardware. The guest VM's in turn are running on top of the hypervisor in a less privileged mode. The hypervisor thus manages and provides the hardware resources to the guest VM's.

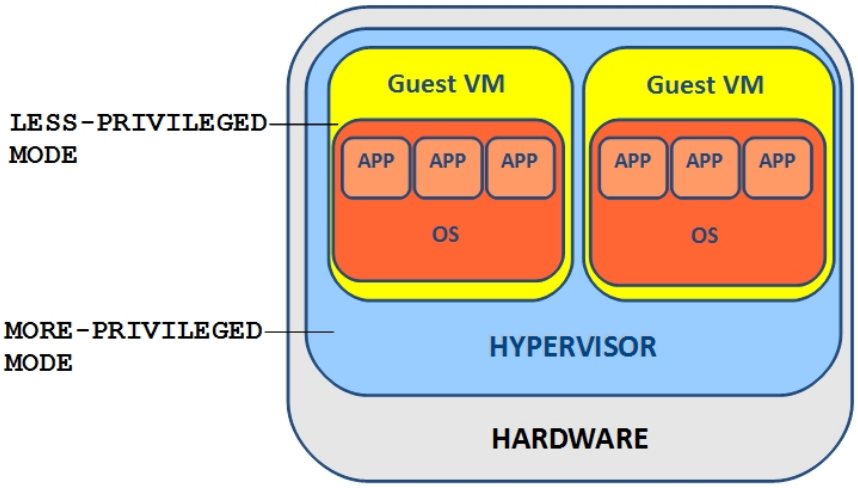

Figure 2.1: Architecture of a hypervisor system

For the guest VM that is running its software, it gets the illusion as if it had full access to the physical machine, while in reality it could be sharing the machine with other software systems. The hypervisor thus maintains the resource allocation between the guests, while it also has the power to intercept on important instructions and events and handle them before they are executed on the real hardware. Another important function of the hypervisor is that it provides isolation of the resources for the virtual machines running on the same physical hardware.

If the security of one virtual machine is compromised, the other virtual machines can continue and run unaffected.

The following is a list of advantages that is achievable with virtualization [29]:

- Isolation
- Minimized size of trusted code base[1]
- Architectural independence
- Simplified development and management
- Resource sharing / Improved utilization
- Load balancing and power saving
- Simplified system migration
- Improved security

In the next section, we will describe how virtualization is achieved.

2.1.2 Classical Virtualization

Popek and Goldberg stated in their paper [23] formal requirements for a computer architecture to be virtualizable. The classifications of sensitive and privileged instructions were introduced in their paper:

- *Sensitive instructions*, instructions that attempt to interrogate or modify the configuration of resources in the system.
- *Privileged instructions*, instructions that trap if executed in an unprivileged mode, but execute without trapping when run in a privileged mode.

To be able to fully virtualize an architecture, Popek and Goldberg stated that the sensitive processor instructions had to be equal to the set of privileged instructions or a subset of it. This criterion has now been termed *classically virtualizable*.

In the following section we present different types of virtualization techniques as each has its own advantages and disadvantages.

2.1.3 General system

In order to understand virtualization, we need to know how a general computer system works when operating without a hypervisor. In Figure 2.2 we show the overview for a general OS running in privileged mode above the hardware. The OS thus has the privilege to execute all machine instructions, including the privileged instructions that control the hardware resources.

[1] Amount of code in the privileged mode

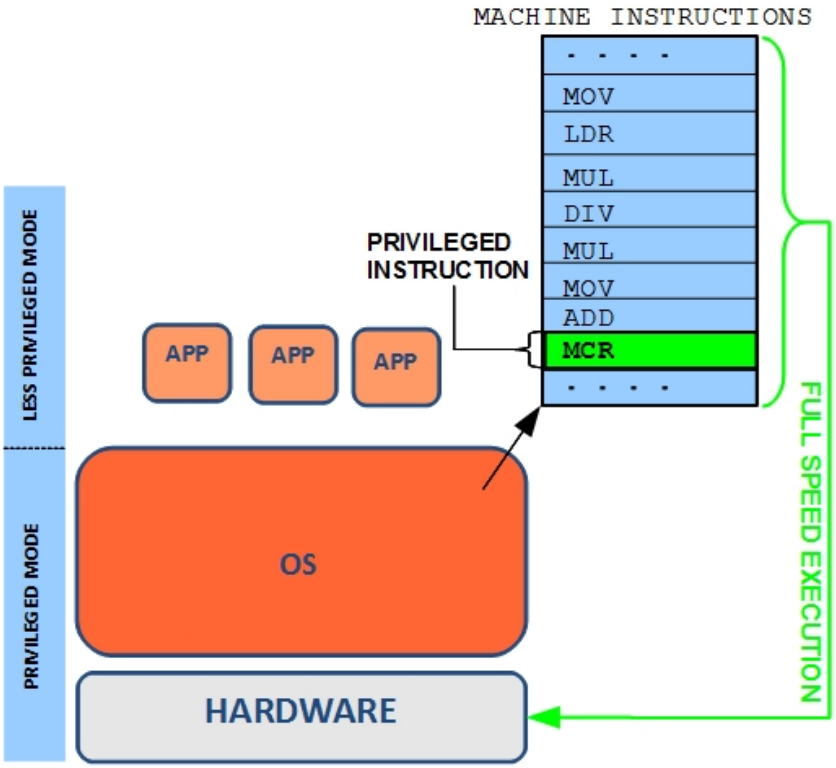

Figure 2.2: General system

2.1.4 Full Virtualization

As discussed earlier, because the hypervisor resides in the most privileged ring, the guest OS which is residing in the less privileged mode, can not execute its privileged instructions. Instead the execution of these privileged instructions has to be delegated to the hypervisor. One way to do this is through applying full virtualization. The idea behind it is, whenever a software is trying to execute privileged instructions in an unprivileged mode, it will generate a so called ”trap“ into the privileged mode. Because the hypervisor resides in the most privileged ring, one could write a trap handler that emulates the privileged instruction that the guest OS is trying to execute. This way, through trap-and-emulate, all the privileged instructions that the guest OS is trying to execute will be handled by the hypervisor, while all other non-privileged instructions can be run directly on the processor, as shown in figure 2.3. The advantage with full virtualization is that the virtualized interfaces provided to the guest operating system has identical interfaces compared to the real machine. This means that the system can execute binary code without any changes, neither the operating systems nor their applications need any adaptation to the virtual machine environment and all code that had originally been written to the physical machine can be reused.

However to apply full virtualization it requires that all sensitive instructions are a subset of the privileged instructions, in order for it to trap to the hypervisor. This is why Popek and Goldberg's criteria classically virtualizable have to be fulfilled in order to apply full virtualization. In the 1970s, this particular hypervisor implementation style, trap-and-emulate was so widespread that, it was thought to be the only practical method for virtualization. A downside with full virtualization is, since a trap is generated for every privileged instruction, it adds significant overhead as each privileged instruction is emulated with many more instructions. In turn we get excellent compatibility and portability.

2.1.5 Binary Translation

In the 90s, the x86 architecture was prevalent in desktop and server computers but still, full virtualization could not be applied to the architecture. Because the x86 architecture contains sensitive instructions that is not a subset of the privileged instructions [27], it fails to fulfill Popek and Goldberg's criteria ”classically virtualizable“. These sensitive instructions would not trap to the hypervisor and it was not possible to execute these sensitive instructions in the unprivileged mode, making full virtualization not possible. VMware has however shown that, with binary translation one could also achieve the same benefits as full virtualization on the x86 architecture. Binary translation solves this problem by scanning the guest code at load or runtime for all sensitive instructions that do not trap before they are executed, and replaces them with appropriate calls to the hypervisor, see Figure 2.4. The technique used is quite complex and increases the code size running in the highest privileged mode, increasing the chance of bugs. Through a security point of

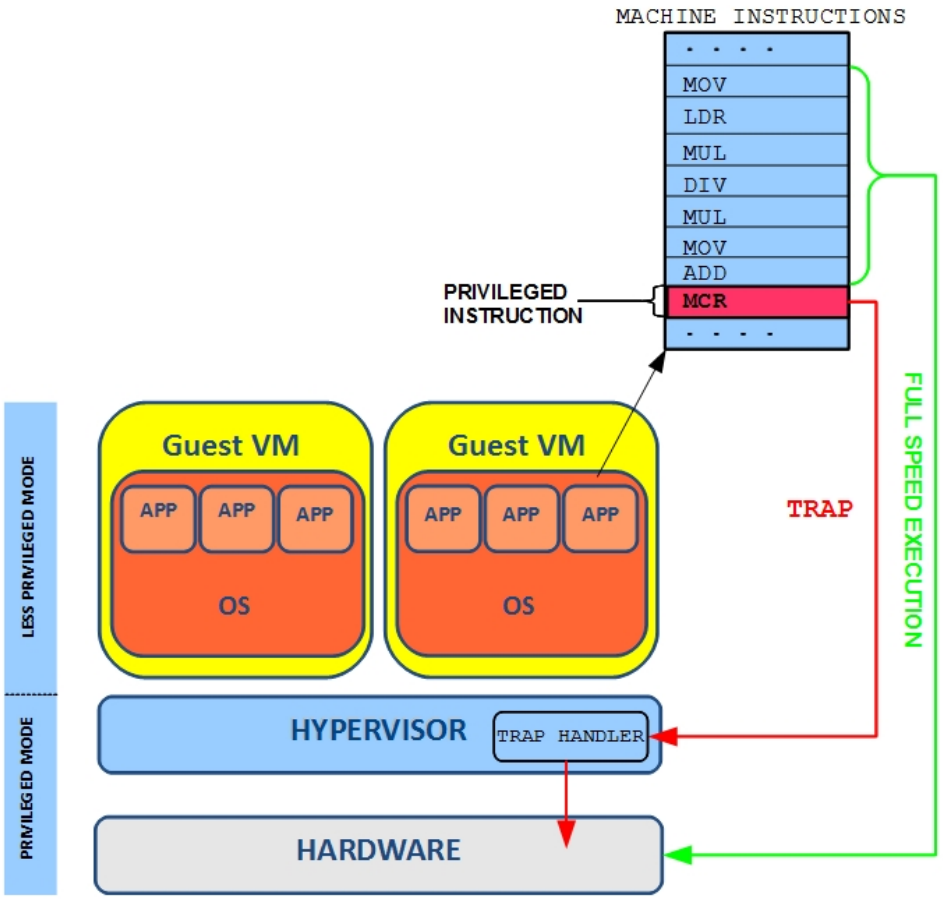

Figure 2.3: Full virtualization

view, one would want the amount of code in the privileged mode to be as small as possible in order to minimize the area of the attack surface. This could affect the security and isolation properties of the entire system.

Because of the complex scanning techniques of binary translation, the performance overhead is larger than full virtualization. However, binary translation has provided the benefits of full virtualization on an architecture that was previously not fully virtualizable. This has brought a renewed interest in virtualization as the benefits for the x86 data servers were enormous.

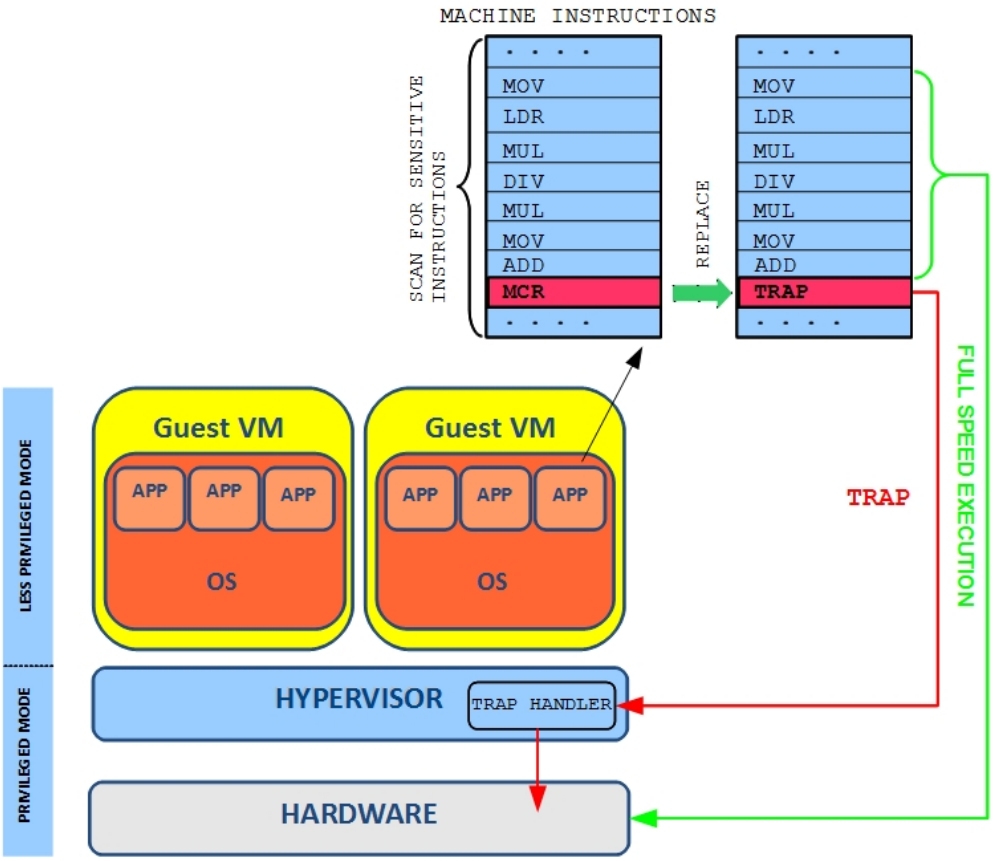

Figure 2.4: Binary translation

2.1.6 Para-virtualization

Para-virtualization was designed to keep the protection and isolation found in the full virtualization but without the performance overheads and implementation complexity in the hypervisor. However to achieve this, you have to sacrifice the convenience to run an operative system unmodified on the hypervisor.

In a para-virtualized system, all the privileged instructions in the operating system kernel have to be modified to issue the appropriate system call that communicates directly with the hypervisor, also called *hypercalls*. This makes para-virtualization able to achieve better performance compared to full virtualization due to the direct use of appropriate hypercalls instead of multiple traps and instruction decoding. Examples on hypercall interfaces provided by the hypervisor are critical kernel operations such as memory management, interrupt handling, kernel ticks and context switching. As each hypercall offer a higher level of abstraction compared to emulation at the machine instruction level, the amount of work that a hypercall can do is a lot more efficient compared to emulating each sensitive machine instruction. Figure 2.5 shows the para-virtualization approach.

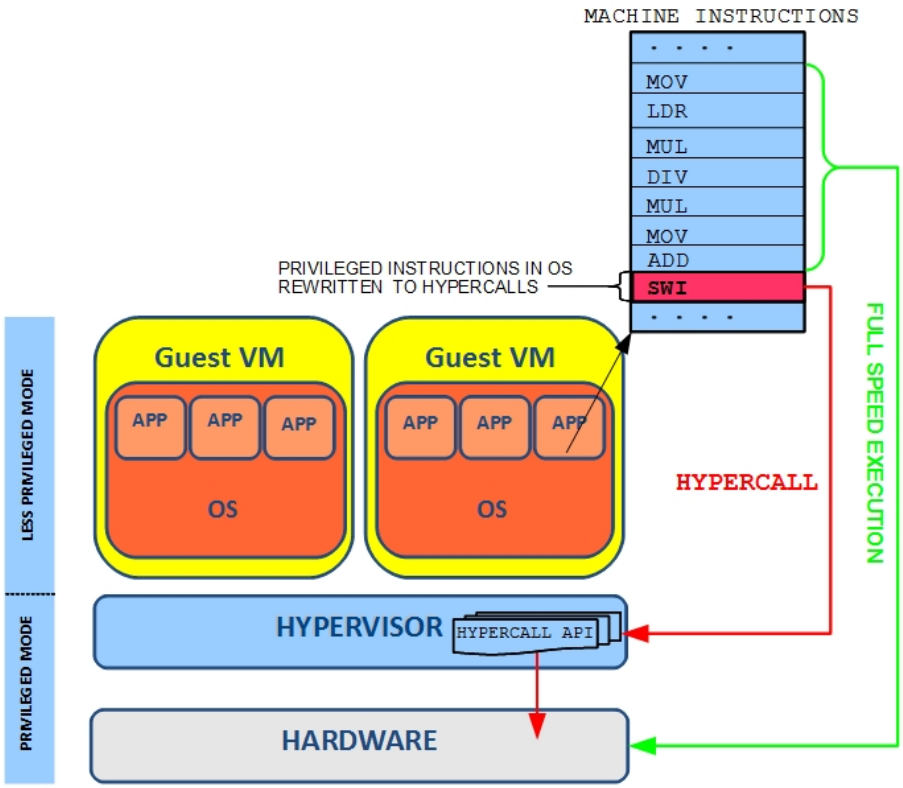

Figure 2.5: Para-virtualization

A hypervisor that uses th para-virtualization approach is Xen on ARM [14] which is able to run multiple isolated high level operating systems. The ARM architecture is a very common CPU in embedded systems, however it is not "classically virtualizable"[2]. This means that virtualization on the ARM architecture can either be achieved through binary translation or para-virtualization. Because embedded systems generally are resource constrained, the performance overhead that binary translation generates is too high, making para-virtualization the best approach for the ARM architecture.

However, the drawback with para-virtualization is that each operating system has to be adapted to the new interface of the hypervisor. This can be quite a large task, and closed-source operating systems like Windows cannot be modified by anyone other than the original vendor. Still, in embedded systems it is common for the developers to have full access to the operating system's source code. The disadvantage to run a modified operating system is not always a big issue; the operating system needs to be ported to the custom hardware either way and at the same time, it performs better.

[2]Except for new ARMv7 architectures supporting TrustZone

2.1.7 Microkernel

Hypervisors are not the only way to achieve virtualization. It has been demonstrated that using a microkernel such as L4 [19] can be used as a hypervisor to support para-virtualized operating systems. However, the approach behind microkernels is different when compared to hypervisors. While the hypervisor was mainly designed to allow multiple VM's to run concurrently on the host computer, microkernels aim to reduce the amount of privileged code to a minimum but still provide the basic mechanism to run an operating system on it. Following these principles, the microkernel's main function is to provide inter process communication, address space management and thread management.

This way policies can be implemented by user level software, utilizing the microkernel provided mechanism as necessary. Operating system services like I/O, device drivers and file-systems can be moved out to the non-privileged mode, decreasing its trusted code size and thus increasing security. This typically requires more adaption from the guest system as it does not try to emulate traditional interfaces like a hypervisor does. Given the different directions and purposes of microkernels and hypervisors, they both share many similarities.

2.1.8 Hardware Virtualization Extensions

Intel and AMD

As the popularity of virtualization keeps rising, hardware vendors started to develop new features to simplify virtualization. In 2006, Intel and AMD released their first generation hardware virtualization extensions for the x86 architecture. Both the Intel-VT [15] and AMD-V [2] processor allows the hypervisor to run in a new root mode below ring 0, which was previously the highest privileged ring. All privileged and sensitive calls have also been set up to automatically trap to the hypervisor, removing the need for either binary translation or para-virtualization. This makes the Intel VT-x and AMD-V classically virtualizable using a trap-and-emulate model in hardware, as opposed to software.

The x86 hardware virtualization extensions was designed to improve virtualization performance in the system, but in [1] the authors stated that, due to high transition overhead between the hypervisor and guests, and a rigid programming model, the first generation hardware virtualization extensions performs poorly. The benchmarks in [1] show; for workload that performs I/O, process creation, and fast context switches, the software outperforms the hardware. It should however be stated that the authors of the paper work for VMware, which makes the research paper biased, as it is in VMware's interest to sell their virtualization software. The authors however acknowledged that the virtualization extension remove the need for binary translation and simplifies the hypervisor design. Both AMD and Intel have announced the development of their second generation hardware virtualization extensions technologies that will have a greater impact on virtualization performance.

ARM

The ARM architecture offers a security extension called TrustZone [26] in ARMv6 or later architectures. It offers support for switching between two separate states, called worlds. One world is secure which is intended to run trusted software, while the other world is normal, where the untrusted software runs. A single core is able to execute code from both worlds, and at the same time ensuring that the secure world software are protected from the normal world. Thus, the secure world controls all partitioning of devices, interrupts and coprocessor access. To control the switch between the secure and normal world, a new processor mode has been introduced called Monitor mode, preventing data from leaking from the secure world to the normal world.

In the latest ARMv7 architecture, the Cortex-A15 processor further introduced hardware virtualization extensions that allow the architecture to be classically virtualized by bringing a new mode called *hyp* as the highest privileged mode, hardware support for handling virtualized interrupts, and extra functionality to support and simplify virtualization. These extra extensions add features to make full virtualization possible and improve the speed of virtualization [7].

2.1.9 Virtualization in embedded systems

As the thesis focuses on virtualization on embedded systems, we will look into the functionality that is inherited from their previous use in servers and workstations. The properties between the two systems are however completely different. For server and desktop computers, power, space or weight are of no concern, while for embedded systems the contrary often holds true. So a re-evaluation in the light of embedded systems is necessary. [16] is an excellent book describing the overview of virtualization for embedded systems.

Architectural coverage

Because the server and desktop markets are largely dominated by the x86 architecture, virtualization approaches have been specifically tailored for this architecture. However the embedded market presents a more divided environment. There is no single dominating processor architecture where there are at least four major architectures in use: ARM, PowerPC, MIPS and Intel. Also for server and desktops, usually the number one requirement is to be able to run all commodity operating systems without modifications. This was the advantage that full virtualization had over para-virtualization, but in embedded systems, it is common for the developer to have access to the full source code of the operating system. Usually the developers have to port the operating system to the specialized embedded hardware, thus using para-virtualization is not such big disadvantage anymore.

Isolation

In servers and desktops, all virtualization approaches feature strong isolation between the VM's and is usually all that is needed to provide a secure and robust environment. A VM that is affected by malicious software will be confined to that VM, as the isolation prevents it from spreading to other VM's. For server and desktop use, this is usually all that is needed because there is no need for VM's to interact with each other in any other ways from how real computers communicate, that is through the network. However in embedded systems, multiple systems generally contribute to the overall function of the device. Thus the hypervisor needs to provide a secure communication interface between the isolated VM's, much like a microkernel IPC, while still preserving the isolation of the system [16].

Code Size

In embedded systems, the size of the memory has a big effect on the cost of the device. They are generally designed to provide their functionality with minimal resources, thus cost and power sensitive devices benefit from a small code size.

In other devices where the highest levels of safety or security is required, every code line represents an additional potential threat and cost. This is called the trusted code base and includes all software that is run in privileged mode, which in general cases includes the kernel and any software modules that the kernel relies on. In security critical applications, all trusted code may have to go through extensive testing. In some cases where security needs to be guaranteed, the security of the system has to be proven mathematically correct and undergo a formal verification. This makes it crucial that the size of the trusted code base is as small as possible as it will make formal verification easier.

In virtualization, the trusted code base will include the hypervisor as it now runs in the most privileged mode. For data server hypervisors like Xen [14], its code base is about 100,000 lines of code which is quite large, but the biggest problem is that it also relies on a full Linux system in the privileged mode. This makes the trusted code base several millions lines of code which makes a formal verification impossible. The reason the Xen and similar hypervisors is so large, is because it is mainly designed for server stations. Most policies are implemented inside the privileged code which embedded systems have very little, or no use of.

In a microkernel all the policies are provided by the servers, while the microkernel only provide the mechanism to execute these policies. This results in a small trusted code base and from a security perspective, for example the L4 microkernel has an big advantage as the size is only about 10,000 lines of code and has also undergone formal verification [17].

Performance

Most often performance is much more crucial and expensive for embedded systems. To be able to get the most out of the hardware, a hypervisor for embedded systems

must perform with a very low overhead as well as being able to provide good security and isolation. The performance overhead that the hypervisor generates depends on many factors such as the guest operating system, hypervisor design and hardware support for virtualization. For embedded systems, it is almost always advantageous to apply para-virtualization as a hypervisor design approach, for the reasons stated in section 2.1.6.

2.1.10 Summary

Until recently, embedded virtualization has received very little interest compared to virtualization of servers and desktops. However, awareness that embedded systems also can benefit from virtualization as a mean to improve security, efficiency and reliability have increased the popularity of embedded virtualization. As the performance of embedded systems continues to grow, one single embedded system is now powerful enough to handle workloads which previously had to be handled by several dedicated embedded systems.

Taking advantage of virtualization, there is a potential to reduce the total number of embedded control units, reducing cost and at the same time increasing performance. Another important aspect is the advances in the mobile embedded devices as today's smart phones provide desktop level software environments. Services like internet banking and surfing the web are available, while the user also has the freedom to install various applications on their mobile device. With these changes, security issues and malicious software has become a threat even in mobile environments. This makes virtualization very attractive as it can provide isolation between different execution environments, separating your security critical applications from the rest. For this reason, many research projects in embedded virtualization are in progress, examples are Xen on ARM [14], OKL4 from Open Kernel Labs [18] and Mobile virtualization platform from VMware [28].

2.2 ARM Architecture

In order to understand virtualization of the ARM architecture, we provide an overview over important components on the ARMv5 platform, especially the ARM-926EJ-S as the SICS developed hypervisor is implemented on this CPU. More information can be found in [6] and [25].

2.2.1 ARM introduction

The ARM core is a *reduced instruction set computer (RISC)* architecture. RISC philosophy concentrates on reducing the complexity of instructions performed by the hardware, while putting a greater demand on the compiler. This way, each instruction is of fixed length of 32-bits and can be completed in a single clock cycle, while also allowing the pipeline to fetch future instruction before decoding the current instruction.

In contrast to RISC, *complex instruction set computer (CISC)* architectures relies more on hardware for instruction functionality, which consequently makes the instructions more complex. The instructions are often variable in size and take many cycles to execute.

As a pure RISC processor is designed for high performance, the ARM architecture uses a modified RISC design philosophy that also targets code density and low power consumption. As a result, the processor has become dominant in mobile embedded systems. It was reported that in 2005, about 98% of more than a billion mobile phones sold each year used at least one ARM processor and as of 2009, ARM processors accounted for approximately 90% of all embedded 32-bit RISC processors [21].

2.2.2 Thumb instruction set

To be able to achieve higher code density, the ARM architecture includes support for an alternative instruction set called *Thumb*. This allows all instructions to be stored in a 16-bit format and be expanded into a 32-bit ARM instruction when they are executed. Although this will result in a lower code performance because of the increase in the number of instructions, it will achieve a higher code density. This can save a lot of space, especially in memory constrained systems. On average, a Thumb implementation of the same code takes up around 30% less memory than the corresponding ARM implementation [25].

However, it offers less flexibility and less functionality due to the small instruction size. For example, only the lower registers r0-r7 are fully accessible, while the higher registers r8-r12 are only accessible to a few instructions. The current program status register (CPSR) and saved program status register (SPSR) registers are also inaccessible when the CPU is in Thumb state.

In order to compensate for the shortcomings of Thumb, ARM released a new version called Thumb-2. It was introduced to achieve similar code density as Thumb, but with the performance and flexibility of ARM instructions. It adds some 32-bit

instructions which allow it to support more functionality such as conditional execution, bit-field manipulation and table branches. However, the ARMv5 architecture that the SICS hypervisor is using does not have support for Thumb-2 as only the newer ARMv7 architectures support it [4].

2.2.3 Current program status register

Beside the 16 general purpose registers from r0 to r15 in the ARM architecture, we have the *CPSR* which the ARM processor uses to monitor and control internal operations. The CPSR is used to configure the following:

- Processor mode: Can be in seven different processor modes, discussed in the next section.
- Processor state: The processor state determines if either ARM, Thumb or the Jazelle instruction set is being used.[3]
- Interrupt masks: Interrupt masks are used to enable or disable the FIQ, IRQ interrupts.
- Condition flags: For the condition flags, it contains the results of ALU operations which update the CPSR condition flags. These are instructions that specify the S[4] instruction suffix and are used for conditional execution to speed up performance.

2.2.4 Processor mode

The ARMv5 contains seven processor modes, which are either privileged or unprivileged. It contains one unprivileged mode *User* and the following modes are all privileged:

- *Supervisor*
- *Fast interrupt request(FIQ)*
- *Interrupt request(IRQ)*
- *Abort*
- *Undefined*
- *System*

When power is applied to the processor, it starts in Supervisor mode, which is generally also the mode that the operating system operates in. FIQ and IRQ correspond to the two interrupt levels available on the ARM architecture. When

[3] ARM - 32 bit, Thumb - 16-bit, Jazelle - 8 bit (Java byte code support)
[4] Certain instructions have the possibility to add an optional S suffix to the instruction

there is a failed attempt to access memory, the processor switches to Abort mode. System mode is used for other privileged OS kernel operations. Undefined mode is used when the processor encounters an instruction that is undefined or unsupported by the implementation. Lastly, the unprivileged User mode is generally used for programs and applications running on the operating system. In order to have full read/write access to the CPSR, the processor has to be in privileged mode.

2.2.5 Interrupts and Exceptions

Whenever an exception or interrupt occurs, the processor suspends the ongoing execution and jumps into the corresponding exception handler in the vector table. The vector table is located at a specific memory address and each four byte entry in the table contains an address which points to the start of a specific routine:

- Reset: Location of the first instruction executed by the processor at power up. The reset vector branches to the initialization code.
- Undefined: When the processor cannot decode an instruction, it branches to the undefined vector. Also occurs when a privileged instruction is executed from the unprivileged user mode.
- Software interrupt: Occurs when the software interrupt (SWI) instruction is used. The instruction is unprivileged and is frequently used by applications when invoking an operating system routine. When used, the processor will switch from user mode to supervisor mode.
- Prefetch abort: Occurs when the processor trying to fetch an instruction from an address without the correct access permissions.
- Data abort: Occurs when the processor attempts to access data memory without correct access permissions.
- Interrupt request: Used by external hardware to interrupt the normal execution flow of the processor.

What the specific routine will do is generally controlled by the operative system. However, when applying virtualization to the system, all the routines will be implemented inside the hypervisor.

2.2.6 Coprocessor

The ARM architecture makes it possible to extend the instruction set by adding up to 16 coprocessors to the processor core. This makes it possible to add more support for the processor, such as floating-point operations.

Coprocessor 15 is however reserved for control functions such as the cache, *memory management unit (MMU)* and the *translation lookaside buffer (TLB)*. In order

to understand how the hypervisor can provide improved security by isolating different resources, it is important to understand the mechanics behind the memory management of the ARM architecture.

2.2.7 Memory management unit

Through coprocessor 15 on the ARM architecture, the MMU can be enabled. Without an MMU, when the CPU accesses memory, the actual memory addresses never change and map one-to-one to the same physical address. However with an MMU, programs and data run in virtual memory, an additional memory space that is independent of the physical memory. This means that the virtual memory addresses have to go through a translation step prior to each memory access. It would be quite inefficient to individually map the virtual to physical translation for every single byte in memory, so instead the MMU divides the memory into contiguous sections called pages. The mappings of the virtual addresses to physical addresses is then stored in the *page table*. In addition to this, access permission on the page table is also configurable.

To make the translation more efficient, a dedicated hardware, the TLB handles the translation between virtual and physical addresses and contains a cache of recently accessed mappings. When a translation is needed, the TLB is searched first and if it is not found, a page walk occurs, which means it continues to search through the page table. When found, it will be inserted into the TLB, possibly evicting an old entry if the cache is full.

The virtualization of memory efficiently supports multitasking environments as the translation process allows the same virtual address to be held in different locations in the physical memory. By activating different page tables during a context switch, it is possible to run multiple tasks that have overlapping virtual addresses. This approach allows all tasks to remain in physical memory and still be available immediately when a context switch occurs.

2.2.8 Page tables

There are two levels of page tables in the ARM MMU hardware. The first level is known as the master page table and contains 4096 page table entries, each describing 1MB of virtual memory, enabling up to 4GB of virtual memory. The level one master page table can either be a section descriptor, coarse page table descriptor or a fine page table descriptor. A section descriptor provides the base address of a 1MB block of memory, while a coarse page table descriptor contains a pointer to a level two coarse page table and the fine page table descriptor contains a pointer to a level two fine page table.

A coarse page table has 256 entries while a fine page table has 1024 entries, splitting the 1MB that the table describes into 4KB and 1KB blocks respectively. The second level descriptor also defines a tiny, small or a large page descriptor. Large page defines a 64KB page frame, small page defines a 4KB page frame and

tiny page defines a 1KB page frame. Figure 2.6 shows the overview of the first and second level page tables.

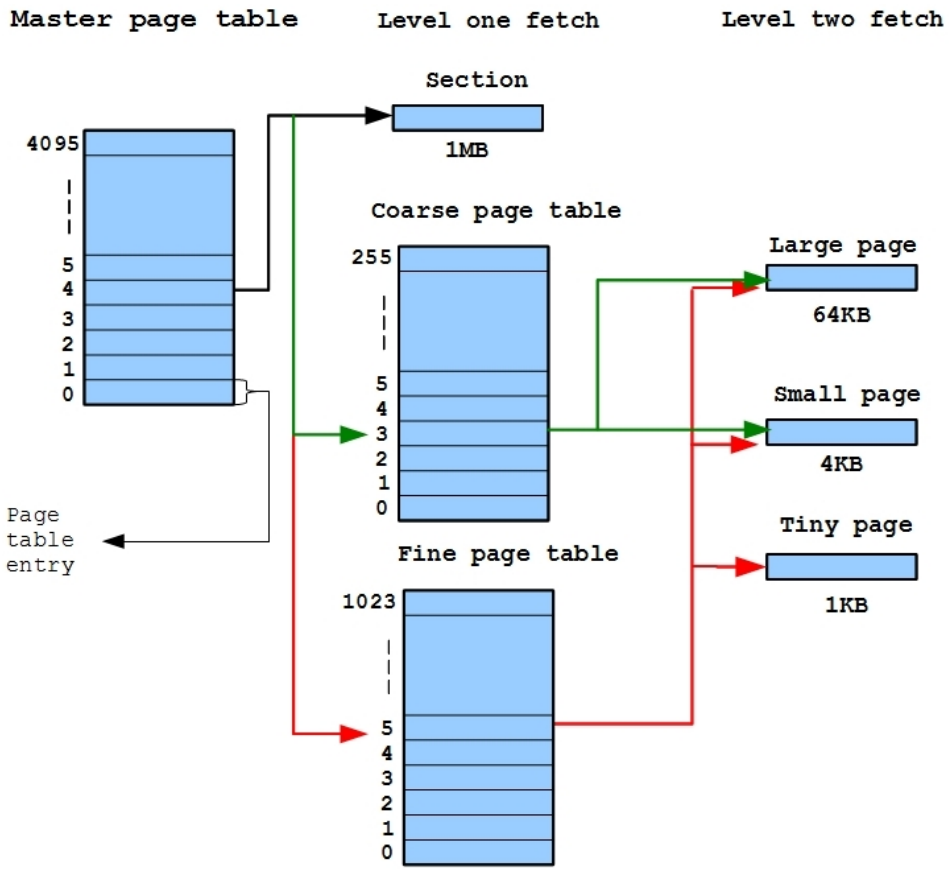

Figure 2.6: TLB fetch

The translation process always begins in the same way at system startup; the TLB does not contain a translation for the requested virtual address so it initiates a level one fetch. If the address is a section-mapped access it returns the physical address and the translation is finished. But if it is a page-mapped access (coarse or fine page table), it requires an additional level two fetch into either a large, small or tiny page in where the TLB can extract the physical address.

Common for all levels of page tables is that it contains configuration for cache, write buffer and access permission. The domain configuration[5] is however only configurable for the first level descriptors, associating the page table with one of the 16 MMU domains. This means that it can only be applied at 1MB granularity; individual pages cannot be assigned to specific domains.

[5]Domains are addressed in the next section

2.2.9 Domain and Memory access permissions

Memory accesses are primarily controlled through the use of domains, and a secondary control is the access permission set in the page tables. As mentioned before, the level one page descriptors could be assigned to one of the 16 domains. When a domain has been assigned to a particular address section, any access to that section must obey its domain access rights. Domain access permissions can be configured through the CP15:c3[6] register and each of the 16 available domains can have the following bit configurations.

- Manager (11): Access to this domain is always allowed
- Client (01): Access controlled by permission values set in the page table entry.
- No Access (00): Access to this domain is always denied

If the configuration is set to Client, it will look at the access permission of the corresponding page table. Table 2.1 shows how the MMU interprets the two bits in the AP bit field of the page table.

AP bit	**User mode**	**Privileged mode**
00	No access	No access
01	No access	Read and write
10	Read only	Read and write
11	Read and write	Read and write

Table 2.1: Page table AP Configuration

In addition to the access permission bits in the page table, there is the S (system) and R (rom) bits in the CP15:c1 register that can modify access permission globally.

Setting the S bit changes all pages with ”No access“ permission to allow ”read access“ for only privileged mode tasks while setting the R bit sets the permission to ”read access“ for both privileged and user mode tasks. These two bits give the possibility to speed access to large blocks of memory without the cost of going through every page table entry and changing the AP for them. The S and R bit only affects the configuration if the AP is set to ”00 - No Access“ and is ignored in other cases. Access control decisions based on the S and the R bit are shown in Table 2.2.

AP bit	**S bit**	**R bit**	**User mode**	**Privileged mode**
00	0	0	No access	No access
00	0	1	Read only	Read only
00	1	0	No access	Read only
00	1	1	Unpredictable	Unpredictable

Table 2.2: Page table S & R Configuration

[6] coprocessor 15, register c3

With the help of the domain access control and page-level protection, we can isolate different memory regions in the system to achieve the wanted security configuration. Detailed examples on the domain and page table configurations are discussed in Chapter 3.

2.3 SICS Hypervisor

The hypervisor software was developed by Heradon Douglas [9] as his Master Thesis in 2010 and has since then been under continuous development. To understand the implemented security services of my thesis, we are going to give a quick overview of the SICS developed hypervisor.

The hypervisor was designed to run on the ARM architecture, specifically the ARM926EJ-S CPU, and supports the FreeRTOS kernel as a single guest. All hardware and peripherals are simulated using the Open Virtual Platforms (OVP) [22] simulation environment. The main goal is to improve the security for an embedded system, mainly with the help of the isolation properties that a hypervisor can provide. Figure 2.7 shows the basic structure of the system.

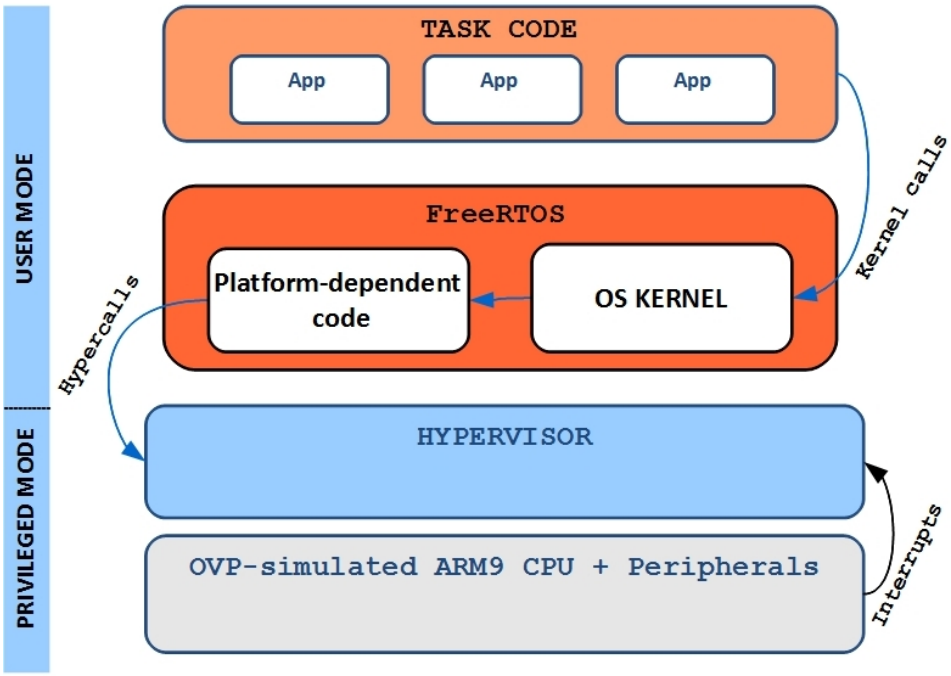

Figure 2.7: Structure of the hypervisor system

The system has three central components:

- The core FreeRTOS kernel
- Platform dependent code that is used by the core kernel

- The hypervisor

FreeRTOS kernel

The core FreeRTOS kernel has remained almost completely unchanged except from some minor modifications on how the task applications are allocated. Previously, the kernel allocated memory for all tasks from the same heap. Heradon added the extra functionality to allocate task memory from a pool of separated heaps which also gives you the possibility to create isolation between the different application tasks. Except from this change, the core kernel was used as it was.

Platform dependent code

The platform dependent code[7] is responsible for carrying out critical, low level actions which requires privileged instructions. Because the kernel now runs in the unprivileged mode, it meant that the platform dependent portion of the FreeRTOS code was para-virtualized, replacing all the privileged instructions with hypercalls.

The hypervisor

The hypervisor was designed specifically for an ARM platform and contains boot code, exception handlers, hardware setup code, and the hypercall interface to allow safe implementation of critical, platform-dependent functionality. It also supports multiple execution environments by having several virtual guest modes. As each guest mode has its own memory access configuration, it uses the MMU to create and enforce the memory isolation between the operating system, its applications and most importantly the security critical applications. As the hypervisor is the only one who can execute privileged code, it is also the only one who can modify and configure the MMU.

2.3.1 Guest Modes

The hypervisor supports an arbitrary number of ”virtual“ guest modes. As each guest mode has their own memory configuration and execution context, the hypervisor always controls which current guest mode is executing. There are currently four guest modes defined in the hypervisor:

- *Kernel* mode: For executing kernel code
- *Task* mode: For executing application code
- *Trusted* mode: For executing trusted code
- *Interrupt* mode: For executing interrupt code

[7] In this case the ARM platform

These virtual guest modes are necessary in the ARM architecture, because we only have two security rings, privileged and unprivileged. The hypervisor has to reside in the privileged ring while all other software such as, the operating system, task applications and security critical application have to reside in the unprivileged ring. Therefore to keep the separation between the software located in the unprivileged ring, we need these virtual guest modes. With the new ARMv7 virtualization extensions, most of these virtual guest modes would not be needed because it introduces a new Hypervisor execution mode that is of higher priority than the Supervisor mode. This enables the hypervisor to execute at a higher privilege than the Guest OS, while the Guest OS can execute with its traditional operating system privileges, removing the need to apply para-virtualization. We could thus simplify the design drastically, and only need trusted and interrupt mode for monitoring the security critical applications and interrupts.

The memory configuration of the system is set up so that it is easy to separate the address space of the different guest modes[8]. Depending on which the current guest mode is, the memory access to the different domains can be set up differently to suit the security needs of the system. The hypervisor then make sure that the correct corresponding virtual guest mode is running, depending on whether kernel, task or trusted code is executing. Whenever an interrupt is generated, the hypervisor will change the guest mode to interrupt mode[9]. In the next section we will go through how memory isolation is achieved.

2.3.2 Memory Protection

With the help of the linker script file, we can control where the hypervisor, kernel, task and trusted code are placed in the memory. Through the domain AP and the page table AP we protect different parts of the system by separating the memory addresses into several domains according to Figure 2.8.

Hypervisor protection

The hypervisor and the critical devices such as the timer and interrupt controller are located in the hypervisor domain. This domain is only accessible in privileged mode, which the system boots up in. At system boot, the hypervisor sets up the hardware and configures the MMU according to our security configurations[10]. The hypervisor then switches the processor to user mode and the current virtual guest mode to kernel mode, and continues the execution to the FreeRTOS kernel application. Transition back to privileged mode only occurs on hypercalls or hardware exceptions, ensuring that no one except the hypervisor can tamper with the memory configurations of the MMU.

[8] Detailed memory configuration of the system is shown in Chapter 3

[9] For DMA interrupts it will change to the guest mode that issued the DMA request

[10] The security configurations of the MMU domain and page tables are described in detail in Chapter 3

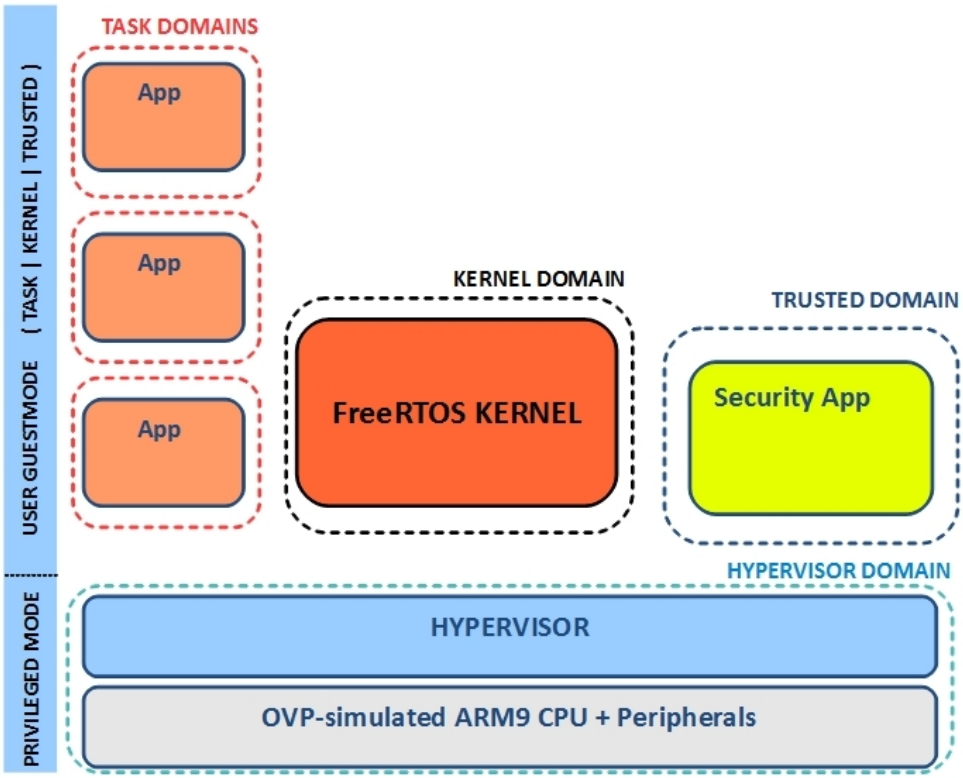

Figure 2.8: MMU Domains

Kernel protection

The kernel code and data are located in the kernel domain. In normal cases, the FreeRTOS kernel API is available for task code, but it is now hidden behind a collection of wrapper functions. Because we need to protect our kernel from the task applications, the kernel functions are wrapped around the enter transition and exit transition hypercall. The enter transition hypercall changes the current virtual guest mode in the hypervisor to kernel mode, this in order to get access to the kernel memory space. This provides a secure interface to use the kernel API without compromising the security of the kernel. When the kernel API call is finished, an end transition hypercall is issued to change the current guest mode back to task mode, disabling the kernel domain and yielding back to the calling task. Figure 2.9 shows the memory domain access configuration for the kernel mode.

Task protection

All individual tasks are given their own domain in order to provide isolation between each task. However this approach does limit the amount of applications because the MMU only supports 16 domains. If feasible, tasks that are known to be trustworthy

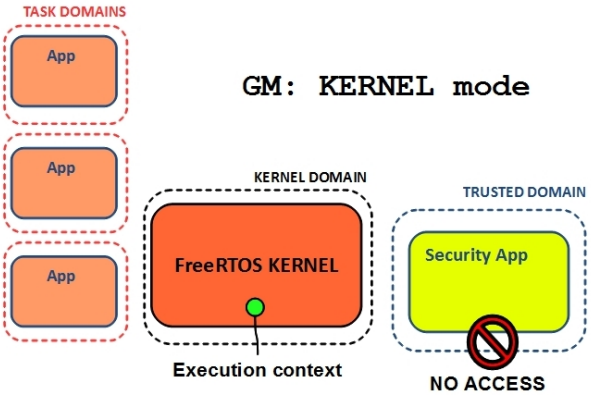

Figure 2.9: Kernel mode domain access

and mutually trusting can be located in the same domain together.

Figure 2.10 shows the memory domain access configuration for the task mode.

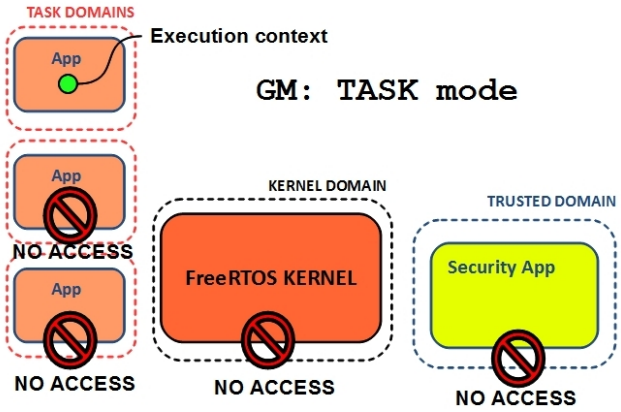

Figure 2.10: Task mode domain access

Security critical application protection

Lastly, a domain is reserved for our security critical applications. This domain will be completely isolated from all other domains in order to protect the data. To use these secure services, a secure well defined interface is provided that can be called trough a remote procedure call (RPC). This will be described in the next section. A typical scenario is a commodity operating system and an isolated service domain offering secure services to the untrusted applications.

Figure 2.11 shows the memory domain access configuration for the trusted mode.

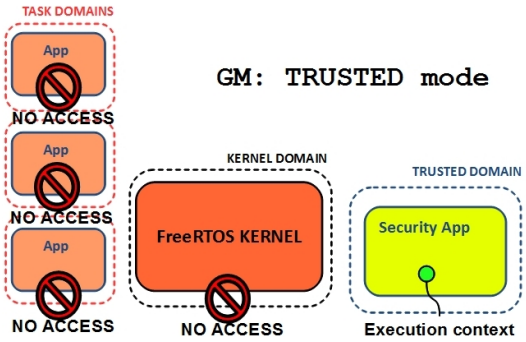

Figure 2.11: Trusted mode domain access

2.3.3 Hypercall interface

To provide a safe access to privileged functionality, the hypervisor offers 11 hypercalls. These are used by the FreeRTOS platform-dependent code, tasks and the MMU wrappers. A hypercall is triggered by the SWI instruction. Each hypercall can be found in Table 2.3

ID	Description	Origin restriction
EIN	Enable user mode interrupts	kernel
DIN	Disable user mode interrupts	kernel
SCO	Set mode context	kernel
GCO	Get mode context	kernel
BTR	Begin transition	wrappers
ETR	End transition	wrappers
ENC	Enter user mode critical section	no restriction
EXC	Exit user mode critical section	no restriction
RPC	Remote procedure call	no restriction
ENR	End remote procedure call	no restriction
END	End DMA	no restriction

Table 2.3: Hypercall interface

The "Origin restriction" column in Table 2.3 refers to where the hypervisor restricts the origin of the hypercall. The first four calls must originate from the FreeRTOS kernel while the BTR and ETR must originate from the MMU wrappers. For the ENC, EXC, RPC, ENR and END hypercalls, no origin restriction is needed as it can be issued directly by tasks.

Enable and Disable user mode interrupts

EIN and DIN hypercalls are used to enable and disable the IRQ and FIQ interrupts for user mode.

Set and Get mode context

The SCO and GCO hypercalls are used by the kernel to save and restore execution context. Used every time the kernel switches task context.

Begin and End transition

The BTR hypercall is used in the kernel API wrapper functions to change the current guest mode to kernel mode. Most kernel functions are wrapped around these two hypercalls in order to make sure that the virtual guest mode is in kernel mode. Kernel mode is needed because it is the only mode that has access to the kernel address space. The ETR hypercall is used to exit the kernel mode and switch back to task mode in order to give back the execution context to task code.

Enter and Exit user mode critical section

A critical section is a piece of code that accesses a shared resource that must not be concurrently accessed by more than one thread of execution. The ENC and EXC hypercall can be called by any task to ensure that it will be the only task with exclusive rights to the shared resource in the critical section. The hypercalls will simply disable interrupts on entry of the critical section and enable it again on exit. This prevents any other task, including an interrupt from getting the CPU until the original task leaves its critical section.

Begin and End Remote procedure call

The RPC hypercall is used to communicate between different guest modes. This requires that the mode offers an RPC interface and the parameters are shared via general registers and special parameter structures.

Possible calls that can be made with RPC is starting the kernel scheduler and Yielding tasks. The parameters that are sent with the RPC hypercall states what kind of operation is to be performed. As the scheduler and Yield operation are both kernel operations, the hypervisor changes guest mode to kernel mode first and then returns execution to the kernel RPC handler where the functions can be performed with kernel access. When the function call is finished, it should call the End RPC hypercall to change back the guest mode to task mode and yield back execution to the calling task. This is similar to the kernel wrapper functions.

For this thesis, we will add RPC functionality that can communicate with the security critical applications by switching the execution context to trusted mode.

End Direct memory access

After a DMA transfer is finished, a DMA interrupt is generated to call the designated guest handler and tell that the DMA transfer is complete. The handler will then use the End DMA hypercall to yield back to the hypervisor, which in turn will check if

there are any other DMA transfers in the queue and eventually give back execution to the interrupted guest mode. DMA is explained in section 2.3.5.

2.3.4 Interrupts

Through the MMU mechanisms, the hypervisor protects critical hardware, such as the interrupt controller and timer. No task can therefore manipulate the timer interrupt.

Whenever a timer interrupt occurs, the hypervisor interrupt handler saves the execution context of the interrupted task, which includes the CPU registers, state and guest mode. The hypervisor then disables user mode interrupts, changes the current guest mode to interrupt mode and returns to the designated kernel handler function. The kernel handler can then perform other activities, such as scheduling another task for execution by restoring its context and re-enabling the user mode interrupts.

2.3.5 DMA Virtualization

Direct memory access (DMA) is a technique in where a specialized hardware is used to copy data much faster, and at the same time free up the CPU to do other tasks in the meantime. The DMA controller (DMAC) is the device used to control the functions of DMA.

Because the DMA device is an independent hardware, it does not follow the memory configurations of the processors MMU. This could easily be used to compromise the security of the system by getting access to protected memory, such as the hypervisor. The common solution to this problem is using a special hardware device, the Input Output Memory Management Unit (IOMMU), which main purpose is to prevent illegal accesses on the bus. The IOMMU is much similar to an ordinary MMU except that it also addresses peripheral devices. However, an IOMMU does in general not even exist on the ARMv5 platform. Fortunately in the ARM926EJ-S processor, it is possible to control the DMA accesses without the IOMMU.

The SICS colleague Oliver Schwarz has in [24], implemented a DMA protection mechanism purely based on software and MMU functionality. The approach is to emulate the DMA controller, meaning the guests do no interact directly with the physical controller. Instead each access attempt will result in a trap into the hypervisor. This way, the hypervisor can control and check the access permission according to a defined access policy and manage the tasks before forwarding it to the physical DMAC. When the DMA transfer is finished, the hypervisor forwards the interrupt it received from the DMAC to the respective guest.

2.3.6 Summary

This section described the overview of the SICS developed hypervisor on the ARM926 EJ-S CPU. With the help of different virtual guest modes and its memory isolation

properties, it provides the tool to protect security critical applications from malicious code. In the next chapter, we will demonstrate the usability and power of the SICS hypervisor solution by implementing a security service upon the hypervisor.

Chapter 3

Implementation of a Security Service

In order to demonstrate the potential power of the hypervisor, an application that offers secure services was implemented upon the hypervisor.

As the hypervisor gives us the possibility to switch between different execution environments with their own memory configurations, we want to define a setup that can provide us with memory isolation between our secure applications and our regular applications. The next section shows how this was achieved.

3.1 Hypervisor Configuration

Before we start implementing the security services onto our hypervisor, we need to make sure that the hypervisor is configured to enforce an access policy that is both safe for the hypervisor, OS kernel and our security critical applications. Through our linker script, we define where the different software regions are located in the memory and it looks according to Figure 3.1. We have the following regions:

- **Hypervisor**: At the bottom address 0x0000, we have the privileged hypervisor region where it stores the hypervisor code and data, the vector table and the stacks for handling exceptions. Dedicated memory addresses are also provided for the page tables and all hardware peripheral devices.
- **Task**: The task region stores the MMU wrapper codes and the main function that starts up kernel tasks and the scheduler.
- **Kernel**: Stores the OS kernel code and data.
- **Trusted**: The security critical code resides in this memory region.
- **Shared**: Stores library code and shared system resources.
- **Taskpool**: Contains five regions that are used by the kernel for its tasks.
- **Shared RPC**: Stores the RPC parameters.
- **Flash**: Stores flash data.

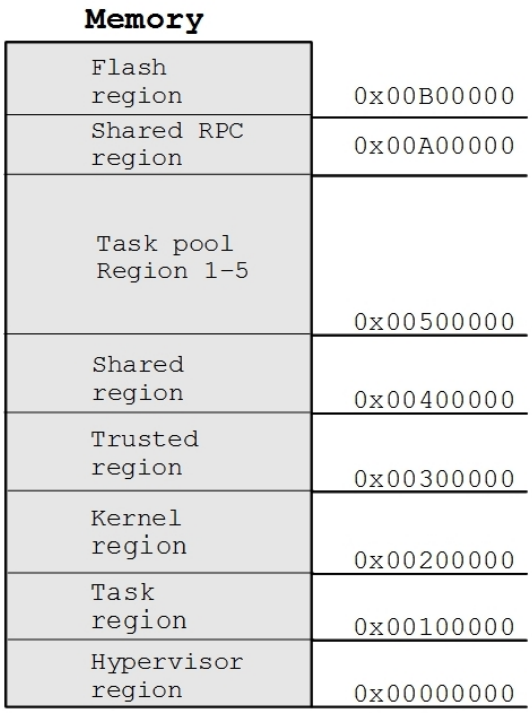

Figure 3.1: Physical memory regions of the system

Now, in order to allow full resource control to the hypervisor, the boot file sets up the vector table and boots into the hypervisor in the processors privileged mode[1]. All other software such as the guest OS, the applications and trusted services runs in the processors unprivileged mode[2], in order to prevent access to privileged instructions. Then through the use of the MMU domain AP and the secondary control in the page table AP, we can control memory access for our system.

3.1.1 Assigning domain and AP to the page tables

Through the configuration in the page tables, each memory region in Figure 3.1 is assigned to one of the 16 domains available in the MMU. In addition, through the CP15:c3 register each domain is configurable to *manager*, *client* or *no access*. In our system, only client and no access is used. If the domain is set to client access, it means that it will check the AP in the page table. Compared to manager access, which gives full access to all memory regions located in that domain, client access provides us with a more fine grained control over our system. The assigned domain and access permissions for the page tables in the different memory regions can be seen in Table 3.1.

[1]Supervisor mode
[2]User mode

Region	Domain	AP (User mode)	AP (Privileged mode)
Hypervisor	0	No Access	Read/Write
Device	0	No Access	Read/Write
Shared	0	Read/Write	Read/Write
Task	1	Read/Write	Read/Write
Kernel	2	Read/Write	Read/Write
Trusted	3	Read/Write	Read/Write
TaskPool 0	4	Read/Write	Read/Write
TaskPool 1	5	Read/Write	Read/Write
TaskPool 2	6	Read/Write	Read/Write
TaskPool 3	7	Read/Write	Read/Write
TaskPool 4	8	Read/Write	Read/Write
SharedRPC	9	Read/Write	Read/Write
Flash	10	Read/Write	Read/Write

Table 3.1: Page table AP Configuration

3.1.2 Domain access in Guest mode

We have defined three virtual guest modes that the hypervisor can switch between and these are the following: *kernel*, *task* and the *trusted* mode. There is also a fourth guest mode *interrupt*, however it is only used by the hypervisor to handle interrupts and will not be shown here. By having different virtual guest modes, we can have different domain access configurations for each mode that suits our security needs. Regular applications are configured to run in the virtual guest mode *task*, while the OS is configured to run in the virtual guest mode *kernel*. Most important, the trusted secure applications are configured to run in the virtual guest mode *trusted*. In our configurations, we have assigned a single domain that our trusted applications reside in (domain 3). It is however possible, to expand this with another trusted domain for other security critical applications to provide isolation between them. The hypervisor will then be responsible for switching address spaces and maintaining the virtual privilege level of the current mode. Table 3.2 shows how each virtual guest mode's memory configuration is set up.

Domain	**10**	**9**	**8-4**	**3**	**2**	**1**	**0**
MemRegions	**Flash**	**Shared RPC**	**TaskPool 0-4**	**Trusted**	**Kernel**	**Task**	**Hypervisor SharedLib Devices**
GuestMode							
GM_TRUSTED	01	01	00	01	00	00	01
GM_KERNEL	01	01	01	00	01	01	01
GM_TASK	01	00	01	00	00	01	01

Table 3.2: Domain access configuration for the hypervisor guests modes.
00 - No access,
01 - Client (Access checked against AP bit in the page table, shown in table 3.1)

If we look at the domain access permission for the virtual guest mode *task*, the kernel memory area (domain 2) are set to no access. This effectively isolates the kernel from the applications. At the virtual guest mode *kernel*, the domain access permission to hypervisor (domain 0), task (domain 1), kernel (domain 2) and taskpool (domain 4-8) are all set to client. This means that for these domains, accesses are checked against the access permission bit in the page table settings. Looking at the access permissions in table 3.1 for unprivileged mode, the access permissions for these domains are all set to read/write except for the hypervisor and device region. This protects the hypervisor software and the devices from illegal accesses when the processor is in the unprivileged mode.

As we can see on the configuration, the trusted domain (domain 3) is not accessible from the *task* or the *kernel* mode. Even if the task/kernel domain has been infected by a malicious application it still cannot access the trusted domain. The only virtual guest mode that can access the trusted domain is *trusted* mode which only the hypervisor can switch to. This way, a secure configuration is achieved by having our untrusted applications located in the task domain while our trusted application reside in the trusted domain.

One thing worth mentioning again, not only do the different guest modes have their own memory areas, they also have their own execution contexts. Whenever the hypervisor is instructed to switch the virtual guest mode, it configures the domain access permission on the MMU according to the configuration in table 3.1 and saves and restores the context[3] of the corresponding guest modes.

To summarize, each time a memory access is performed, the MMU looks at which domain the page table belongs to. The next step is to check the domain AP for the domain (Table 3.2). If it is set to no access, permission is denied. For client access, it continues to check the AP in the page table (Table 3.1). With the help of the MMU, page tables and the different virtual guest modes, we have defined a secure access policy to our system. The hypervisor configuration code is included in Appendix-A1.

3.1.3 Secure services in trusted mode

Because the trusted domain is isolated and inaccessible from the other domains, the secure services running on the trusted domain are made available to the applications through dedicated hypercalls implemented in the hypervisor. This is called remote procedure call (RPC) and the arguments that are sent with the RPC tells which guest mode to switch to and what kind of services that we want to perform. The RPC will generate a software interrupt (SWI) which is a privileged operation causing it to trap to the hypervisor. The hypervisor then analyze the parameters of the RPC and checks the configurations if the accesses are correct and allowed. After the hypervisor has switched to *trusted* mode, the secure services are then made sure to only rely on encrypted and integrity protected data from external memories. This

[3]Saving and restoring the registers in the processor(r0-r15)

provides us with a basic level of security against physical attacks on the system. When the trusted service is finished with its operations it issues a hypercall "end RPC" which yields back to the calling guest mode.

The hypervisor thus provides isolation between different execution environments and communication between the untrusted domains and the trusted domain are only allowed through secure interfaces on the trusted application.

3.2 Implementation Approach

Developing software in an ARM platform was at the beginning of the project a completely new area for us. As we also were given an early deadline to present the application at the SICS open house event, rather than invest too much time into designing for future unknown challenges, we decided to create fast prototypes that would handle the challenges as they were presented. Each prototype provided us with results and experience, which we used in the subsequent phase. Much similar to "code and fix" as time was pressed, we immediately started to produce code and debugged it as it was being written.

The initial phase also provided us with a backbone on how to further optimize the hypervisor on the next phase of the thesis, described in Chapter 4.

3.3 Scenario

The use case that we defined is the following; an application asks a trusted service to see a bank contract and then digitally signs it. The signature is then verified for its authenticity. The sample security application will be designed and implemented by us and contains security services that you would expect from a real application. In order to realize this use case, we need some cryptographic services.

3.3.1 Cryptographic Services

First of all, the bank contract contains confidential information. In order to protect this document we need a service to encrypt and decrypt our bank contract. For this we use the advanced encryption standard (AES-128). It is suitable for encrypting larger messages as it's a symmetric-key algorithm which is faster than the corresponding asymmetric-key[4] algorithm [13]. A common security practice is to encrypt the AES key with an asymmetric-key encryption, and for this we use the RSA[5] algorithm. As RSA uses larger keys (typically 1024-2048 bits) compared to AES (128-256 bit), it makes it slower and more suitable for encrypting small messages, like the AES key. So in summary, our bank contract is encrypted with the AES key, while our AES key is encrypted with the RSA key. In order to decrypt the bank document, one needs to first decrypt the AES key with the RSA key. Only

[4]Also called public key cryptography

[5]Stands for Rivest, Shamir and Adleman who first publicly described it

then can we use the decrypted AES key to decrypt our bank contract.

For the digital signature, one also utilizes the functions of asymmetric cryptography. In this case we can reuse the RSA algorithm. However, as we described earlier RSA is a very slow algorithm so we first create a small unique hash value from the bank contract. We will choose secure hash algorithm (SHA-256) for generating our hash value as it is a commonly used standard.

With the help of the hypervisor we can utilize its memory isolation mechanism, to store all the cryptographic services and keys in a trusted memory domain that is isolated from all other domains. The hypervisor can then manage the communication between the different guest modes, for example when an application wants to use a secure service.

3.4 Implementation

As most cryptographic libraries was not designed for the ARM platform, we tried to find compatible cryptographic libraries. This was much harder than we expected, and because of our initial inexperience of the hypervisor and the ARM platform, we did not succeed to port the libraries to the ARM platform. The problem was that the instructions that the ARM926EJ-S processor can perform is limited. For example, it could only handle up to 32-bit operations, and no floating point operations were supported [5] which led to compile errors when trying to incorporate the library to our platform. The security application is also to be implemented in a constrained embedded platform which makes a big cryptographic library undesirable with the footprint in mind.

So as a fast solution, well known and portable C implementations of the various cryptographic services were found on the Internet and we tailored it to work with the ARM-EJ926 platform. Source code for the trusted service and the application using the trusted service are included in Appendix-A2 and A3, respectively.

3.4.1 Material

For the AES-128 bit algorithm, Brian Gladman's implementation was used [11]. It uses cipher-block chaining mode with cipher-text stealing and uses a random generated initialization vector with the help of George Marsaglia's pseudo random number generator (PRNG). It is a multiply with carry (MWC) PRNG that concatenates two 16 bit MWC generators [20]. The SHA-256 implementation was also used from Brian Gladman's work [11].

For the RSA-1024 bit algorithm, Michael J Fromberger's implementation was used [10]. It uses an arbitrary precision integer arithmetic package to represent big numbers as RSA uses 1024 bits to represent a number which the built in types of C cannot handle. The possibility to sign and verify SHA-256 hash values were added to the RSA algorithm. All the source code were ported to work with the hypervisor and the ARM platform.

3.4.2 Security application

The use case is repeated for the readers sake; an application asks a trusted service to see a bank contract and then digitally signs it. The signature is then verified for its authenticity. We will here describe the step through step processs of how the hypervisor works together with the security application.

Depicted in Figure 3.2, we have the task, kernel and trusted domain in which their respective applications reside in (other domains are not shown for clarity). The external memory peripheral holds the AES encrypted bank contracts and the RSA encrypted AES keys. The bank contract is encrypted with the AES key and the AES key is encrypted with the public key of RSA (step 1). The private key that is able to decrypt this AES key is located in the secure isolated trusted domain.

The application that wants to see the bank contract performs an RPC which "ask's" the hypervisor to switch the current guest mode *task* to *trusted* mode in order to get access to the trusted services. Because the hypervisor configuration does not allow switching directly from task mode to trusted mode, the RPC must be forwarded to *kernel* mode first, then back into the hypervisor and finally into *trusted* mode where the security service can be called (step 2). When the hypervisor has switched to *trusted* mode it starts a direct memory access (DMA) transfer of the chosen encrypted bank contract and respective encrypted key into the trusted domain (step 3). One important point is that the DMA controller is also virtualized, meaning that guests do not interact with it directly. Instead each access traps into the hypervisor where it controls that the application has proper access before forwarding it to the real DMA controller. When the transfer is finished a DMA interrupt is generated to inform the issuer that the transfer operation is complete.

The trusted service then uses the private RSA key to decrypt the AES key, which it then uses to decrypt the encrypted bank contract (step 4). All these operations are performed inside the trusted domain in order to not leak out any sensitive information. The contract is then securely displayed to the application and the application decides to sign the contract. The trusted application generates a SHA-256 hash value of the bank contract and signs this with a different private RSA key[6] (step 5). This signature is then passed with the RPC parameters back to the application (step 6). The application now has the signature and can verify it with a third party, for example a bank. In this demo application, we simply send the signature back to trusted domain with the RPC arguments and verify the signature there. The verification is done by extracting the SHA-256 hash value from the signature with the corresponding public RSA key. It then compares this hash value with the generated SHA-256 hash value from the plaintext bank contract. If both matches, then the authenticity of the signature is valid.

To demonstrate the security of the hypervisor, a malicious application was also implemented to try and hack into the trusted domain. The malicious application was given the address of the unencrypted plaintext bank contract residing in the trusted domain. Through regular memory accesses and DMA transfers, it tries to

[6]Security practice to not use the same RSA key to sign and encrypt data

steal the contract. These accesses to the trusted domain will be interfered by the hypervisor (step 7).

3.4.3 Conclusion

With the help of the hypervisor we have successfully managed to implement a secure isolation between the different memory domains and as well provide a secure service in the trusted domain. This way the cryptographic keys stored in the trusted domain are not accessible by other domains, they can only ask trusted mode to use these keys to perform the cryptographic services that it provides.

Keep in mind that for demonstration purposes we have defined one trusted domain that offers security critical services, but we are by no means limited to just one. One could allow multiple stakeholders (banks, mobile operators, media services) to run their own secure services independently from each other and still maintain complete isolation and security from each other.

The application implemented was the first part of the thesis and its main purpose was to demonstrate the effectiveness of the hypervisor at the SICS Open House event in Stockholm.

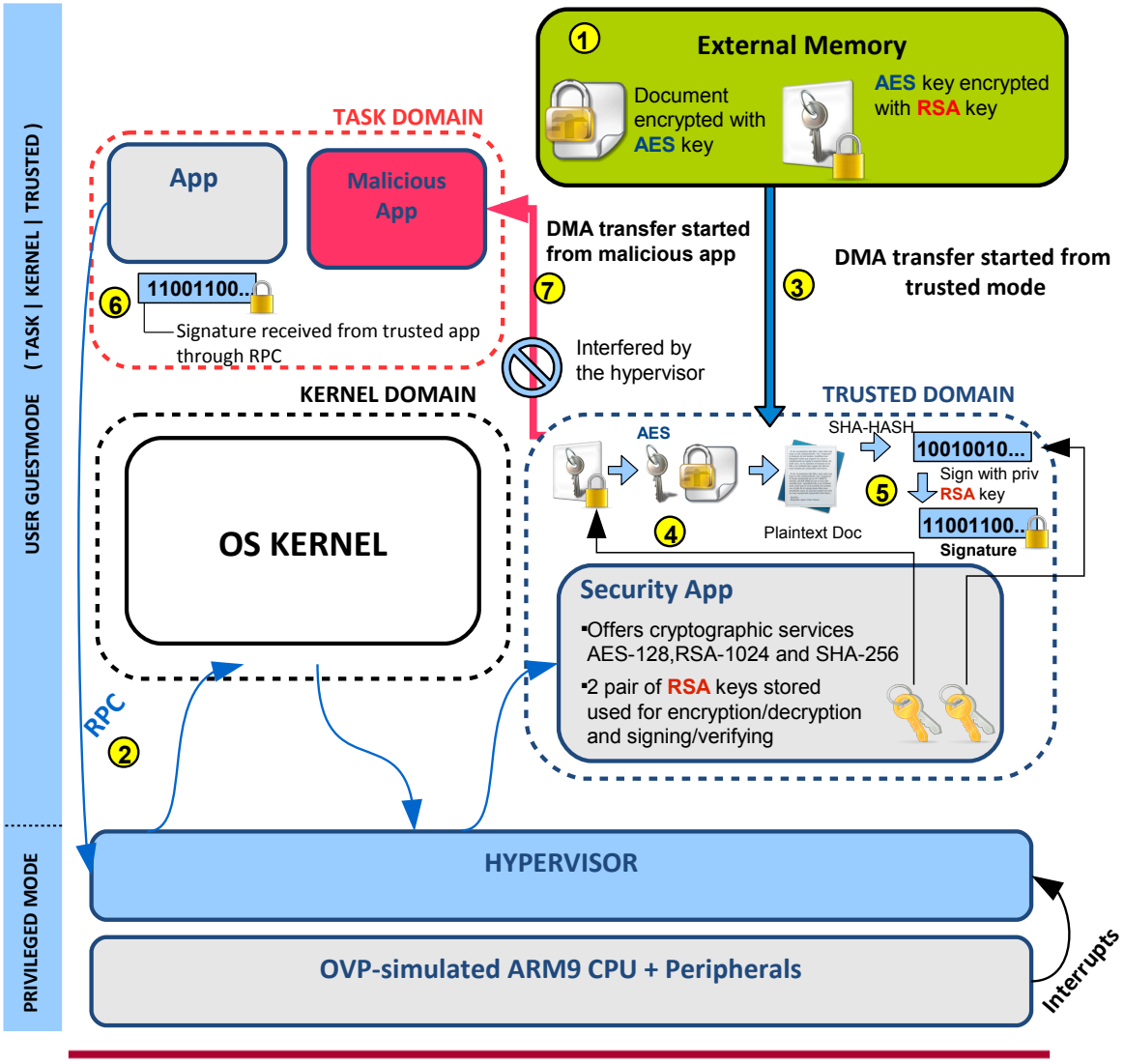

Figure 3.2: Hypervisor demo

Chapter 4

Optimization

In order to provide a hypervisor designed for hardware constrained embedded systems, it is important that the software is as small as possible. Another important aspect is that the small footprint will ease the formal verification of the hypervisor. We have showed that the hypervisor provides secure isolation with our security application that we demonstrated on the SICS open house. The next step is to optimize the hypervisor and we are going to focus on the footprint. At the same time, we will carefully profile the hypervisor as we do not want the optimization of the footprint to have too big impact on performance.

4.1 Memory Footprint

When we are discussing the memory footprint of the hypervisor, we are referring to how much memory space the hypervisor would use or reference while running. This includes all memory regions such as text, static data sections, heap and stacks. The text section is the place in computer memory where the compiled code of the program itself resides. All variables that are declared and initialized *before* runtime are stored in the stack (static allocation). Similar, all variables that are created or initialized *at* runtime will reside in the heap (dynamic allocation).

Now, by optimizing the hypervisor code, we can reduce the size of each memory section. The greatest reduction can be achieved by minimizing the size of the compiled code; however we also have to address dynamic memory allocation in the hypervisor. As dynamic memory can be allocated and freed during the execution of the program, it is hard to guess how much dynamic memory is actually used. A good approach is to analyze the utilization of the heap memory while the hypervisor executes, in order to determine the peak memory usage. However, the hypervisor rarely use dynamic memory allocation as there is only one use of malloc in the hypervisor. Therefore, our optimization will concentrate on the footprint of the compiled hypervisor and not the dynamic memory usage.

4.2 Current structure of Hypervisor

The size of the hypervisor is currently **44 766** bytes with no optimization flags on the compiler and with the debug flag turned off. Table 4.1[1]shows the size of each symbol. The tables were generated with the GNU Binutils tools [30] arm-elf-nm and arm-elf-size.

Name	Type	Class	Size	Section
hyper_flpt	d	OBJECT	16384	fl_pt
hyper_slpt	d	OBJECT	12288	sl_pt
dmaAbortHandler	t	FUNC	1224	hyper_functions
dmaInterruptHandlerImpl	t	FUNC	872	hyper_functions
dataAbortHandler	T	FUNC	808	hyper_functions
hypercallRPC	t	FUNC	708	hyper_functions
hypercallImpl	t	FUNC	620	hyper_functions
dmaAccessCheck	t	FUNC	600	hyper_functions
hyperSetUpMMU	t	FUNC	528	hyper_functions
hypercallBeginTransition	t	FUNC	520	hyper_functions
hypercallEndTransition	t	FUNC	484	hyper_functions
try_to_submit	t	FUNC	476	hyper_functions
hypercallEndRPC	t	FUNC	448	hyper_functions
modeStates	d	OBJECT	400	hyper_data
hyperInitData	t	FUNC	372	hyper_functions
faultStatusSource	t	FUNC	340	hyper_functions
hyperChangeGuestMode	T	FUNC	308	hyper_functions
enqueue	t	FUNC	284	hyper_functions
hyperSetUpTimer	t	FUNC	224	hyper_functions
hypercallGetModeContext	t	FUNC	204	hyper_functions
hypercallSetModeContext	t	FUNC	200	hyper_functions
dmaBurst	t	FUNC	196	hyper_functions
hypercallEndInterrupt	t	FUNC	172	hyper_functions
setupDMA	t	FUNC	160	hyper_functions
hyperSetUpHW	t	FUNC	152	hyper_functions
popContextStack	T	FUNC	148	hyper_functions
dequeue	T	FUNC	148	hyper_functions
pushContextStack	T	FUNC	144	hyper_functions
hypercallHandler	T	FUNC	144	hyper_functions

Continued on next page

[1]The *Type* column shows the symbol type. If lowercase, the symbol is local; if uppercase, the symbol is global (external). These are the following types used:
D - The symbol is in the initialized data section
R - The symbol is in a read only data section
T - The symbol is in the text (code) section

Table 4.1 – continued from previous page

Name	Type	Class	Size	Section
progAbortHandler	T	FUNC	140	hyper_functions
hypercallEndDMA	T	FUNC	140	hyper_functions
dmaInterruptHandler	T	FUNC	124	hyper_functions
hyperTickHandlerImpl	T	FUNC	120	hyper_functions
hyperTickHandler	t	FUNC	104	hyper_functions
hyperMain	T	FUNC	100	hyper_functions
DEFAULT_MODE_STATE	r	OBJECT	100	.rodata
DMAQ	d	OBJECT	88	hyper_data
hyperInit	t	FUNC	72	hyper_functions
hypercallOriginError	t	FUNC	64	hyper_functions
hypercallNumError	t	FUNC	52	hyper_functions
hyperPanic	T	FUNC	48	.text
writeReg32	t	FUNC	44	.text
readReg32	t	FUNC	40	.text
setUpMode	t	FUNC	28	hyper_functions
interruptedMode	d	OBJECT	4	hyper_data
guestTickHandler	d	OBJECT	4	hyper_data
dmaInterruptedMode	d	OBJECT	4	hyper_data
dmaHandlingMode	d	OBJECT	4	hyper_data
currentModeState	d	OBJECT	4	hyper_data
currentModeContext	d	OBJECT	4	hyper_data
currentGuestMode	d	OBJECT	4	hyper_data

Table 4.1: Symbol size in hypervisor

Table 4.2 shows the total sum of the hypervisors sections.

Section	Size
.text	448
.data	0
.bss	0
.rodata	4000
.hyper_data	516
.hyper_functions	11112
fl_pt	16384
sl_pt	12288
Total	**44748**

Table 4.2: Total hypervisor size

With the help of these tables, we are able to locate those parts of the hypervisor that are suitable for optimization.

4.3 Profiling

For the profiling of the software, we are using Imperas Verification, Analysis and Profiling (VAP) tools. As we are using a simulated hardware platform, the recorded performance is in OVP-simulated instruction counts. In order to maintain a benchmark baseline that we can compare our result with, we used the same benchmark tests from Heradon's thesis.

4.3.1 Benchmark

The tests covers the utilization of the MMU kernel wrappers, hypercalls, interrupts and yielding which are the key performance burdens imposed by the hypervisor. All the tests use five tasks to perform a parallelizable math workload which consists of 10000 work units. The task takes work units in a loop inside a critical section in where interrupts are disabled. It then leaves the critical section which enables the interrupts and carries out the work unit. Each time a critical section is entered/exited, hypercalls are utilized. The tests are both run in preemptive and non-preemptive mode and the hypervisor is compiled with no optimization flags in the benchmark.

In the first test called "MathTest", the tasks will continue to take work units until it is preempted by the FreeRTOS scheduler. One effect of running the test in non-preemptive mode is that one task will carry the whole workload by itself until it is finished and will never yield. In the second test "YieldingMathTest", the tasks yield after completing five work units which will result in a significant execution of the yield mechanism. Whenever a task yields, it issues a hypercall to the hypervisor which saves context and returns to the kernel handler function which starts another task. In the third test "WrapperMathTest", the tasks call an arbitrary kernel function after completing five work units which results in a significant execution of the wrapper mechanism. When a kernel API function is called, hypercalls are issued to enter and exit guest kernel mode which modifies MMU settings.

4.3.2 Problems

As the development of the hypervisor continued after Heradon's thesis work, the benchmark results from the original thesis are not longer relevant. The hypervisor has gone through big design changes since the original thesis and more functionality has been added such as, support for multiple execution environments and DMA support. Also the new version of the hypervisor contains less assembler code, as it has been rewritten to C code instead.

However, the further development of the hypervisor should not have a big impact on the benchmarks. To get a new baseline benchmark that we can compare with our optimized hypervisor, we reran the benchmark tests with the new version of

the hypervisor[2]. To our surprise the performance overhead of the benchmark tests was now 220% compared to the results from Heradon's thesis. Trying to run the benchmark in preemptive mode would also cause a data abort with a page domain fault status, telling us that we dont have access to the kernel memory area. The problem is that the hypervisor fails to change guest mode when the kernel tries to pre-emptive a task and do a context switch. The results are shown in Table 4.3.

Benchmark	**Current hypervisor**	**Heradon's Thesis hypervisor**	**Baremetal kernel**
MathTest, Preemp	N/A	10,696,343	10,472,960
MathTest, Non-premp	23,436,605	10,696,268	10,469,648
YieldingMathTest, Preemp	N/A	11,885,640	11,109,395
YieldingMathTest, Non-preemp	25,964,590	11,880,823	11,105,748
WrapperMathTest, Preempt	N/A	11,128,362	10,612,814
WrapperMathTest, Non-preemp	25,004,582	11,124,540	10,609,665

Table 4.3: Hypervisor benchmark

This indicates that the further development of the hypervisor has introduced some errors as the added functionalities and design changes should not theoretically have such a big impact on performance. Clearly this needed to be corrected before we could continue with our optimizations as correct behavior from the hypervisor is of utmost importance.

We used the VAP tool *linecoverage* to profile the version of the hypervisor that was used on Heradon's thesis benchmarks. The tool *linecoverage* outputs an html file which shows how many times each line in the hypervisor source code is executed. This was repeated with the current version of the hypervisor, and anomalies could be detected by comparing both results. After investigating the problem with careful comparisons and extensive debugging, we found out that the main problem was that the enter and exit critical section hypercalls have been removed from the hypervisor interface. Instead, the OS kernel had been changed to use the MMU kernel wrappers to do the exact same function. As discussed earlier, the kernel wrapper functions "wraps" the kernel system call with begin and end transition hypercalls[3], leading to an exaggerated exercise of virtual guest mode changes. The core functionality of the hypervisor was not affected, except it would run more than twice as slow.

The error was corrected and to improve performance making it match the former benchmarks, the most executed methods were also rewritten to assembler. The new benchmark results can be seen in Table 4.4. Unfortunately, as this error was found in the very late phase of the thesis, we only managed to find time to tune up the performance for MathTest. These results will now be our new baseline benchmark.

[2]The hypervisor version we got at the start of our thesis

[3]Hypercall interface was described in Chapter 2.3.3

Benchmark	**Corrected hypervisor**	**Heradon's Thesis hypervisor**	**Baremetal kernel**
MathTest, Preemp	**10,639,182**	10,696,343	10,472,960
MathTest, Non-premp	**10,631,831**	10,696,268	10,469,648
YieldingMathTest, Preemp	**13,235,897**	11,885,640	11,109,395
YieldingMathTest, Non-preemp	**13,221,812**	11,880,823	11,105,748
WrapperMathTest, Preemp	**12,066,224**	11,128,362	10,612,814
WrapperMathTest, Non-preemp	**12,054,816**	11,124,540	10,609,665

Table 4.4: Hypervisor benchmark

4.3.3 Profiling function count

Table 4.5 shows how many times each functions in the hypervisor is called when running the benchmark MathYieldTest.

Function name	**Count**
hypercallHandler	28068
hcEnterCritical	10019
hcExitCritical	10019
hypercallImpl	8030
hyperChangeGuestMode	4045
hypercallSetModeContext	2001
hypercallGetModeContext	2000
hypercallRPC	2001
hypercallEndRPC	2001
hypercallEndInterrupt	16
hyperTickHandler	16
hyperTickHandlerImpl	16
setUpMode	6
hypercallEndTransition	5
hypercallBeginTransition	5
writeReg32	1
hyperMain	1
hyperSetUpMMU	1
hyperSetUpHW	1
hyperInitData	1
hyperInit	1
hyperSetUpTimer	1
setupDMA	1

Table 4.5: Hypervisor function count

This gives us valuable information on which parts of the hypervisor significant

execution time is spent. These results will only be a guideline in where to put effort, as these results depends on the software we are executing. Functions that are never executed are not shown in the table.

4.4 Implementation of the optimization

With the help of our profiling results, we have a clear baseline that we can compare with and confirm our progress, and in which parts to apply our optimizations. The main objective of the optimization is to minimize the footprint of the hypervisor, while maintaining the performance. The next sections describe the implemented optimizations on the hypervisor.

4.4.1 Removing static variables

One obvious start is to look at the parts that occupy most space. Looking at the top of Table 4.1, we can see that the symbols *hyper_flpt* and *hyper_slpt* are the largest symbols in the hypervisor. They both take up 28 672 bytes in total, which is about 64% of the hypervisor. These are the first and second level page tables used by the MMU to translate virtual addresses and control access permission on the different domains. Looking closer into the source code, the page tables are defined as static 32-bit arrays with a length of 4096 that are located at a fixed specific address (0x8000).

Now, the linker script is already configured to reserve space for the page tables in memory, in order so that the hypervisor always has access to it. This makes it unnecessary to create a static array structure in the hypervisor to hold the values of all the page tables.

The static array was therefore removed from the hypervisor, and the page tables were instead configured and accessed with direct pointers to the page table's physical address. As the goal with the thesis was to minimize the footprint of the hypervisor, we also feel that the page tables are not something that should be part of the hypervisors footprint. The amount of memory that the page tables occupy is directly depending on how the page tables are configured in each system. Therefore, we decide to omit the size of the page tables from our hypervisor optimization results and start our baseline comparisons from this point. Table 4.6 shows the size of our new baseline hypervisor.

Section	Size
.text	448
.data	0
.bss	0
.rodata	4000
.hyper_data	516
.hyper_functions	11144
Total	**16108**

Table 4.6: Total hypervisor size after removing static variables

4.4.2 Debug mode

The current hypervisor contains alot of debug information that demonstrates that the hypervisor behaves correctly. This is implemented with various text output on the screen, and for the SICS open house demonstration, we[4] have also developed a graphical user interface that collects these printouts and displays them in a user friendly way. However, this has increased the footprint of the hypervisor and is not needed in the real release.

By introducing a debug mode, we can define a regular printf macro when debug is defined, and an empty macro otherwise. This way, we can save some footprint when the hypervisor is not in debug mode by omitting all the printf statements.

Following table shows the updated size after we have implemented debug mode.

Section	Size
.text	404
.data	0
.bss	0
.rodata	2572
.hyper_data	516
.hyper_functions	10064
Total	**13556**

Table 4.7: Total hypervisor size with debug mode implemented with the hypervisor in release mode

[4]GUI implemented by SICS colleague Oliver Schwarz

4.4.3 Thumb Mode

In order to minimize the footprint of the hypervisor, it is vital to compile the code as a mixture of ARM and Thumb[5] instructions. This process is called interworking and is supported by the ARM GCC cross-compiler.

The processor is able to switch between ARM and Thumb mode dynamically under run-time, but unfortunately the ARM GCC cross-compiler does not support mixing ARM and Thumb code in the same object file. This led to that we had to divide our hypervisor into two object files, one that only contains ARM code, and one that only contains Thumb code. We had to break the design of the hypervisor and rearrange all data variables and functions to each respective file. This leads to poor overall design, but there was no other way to achieve this optimization with our chosen compiler. For example if we had access to ARM's own compiler [3], we could add "#pragma thumb" before a function, and it would be compiled into thumb instructions. In that case, there would be no need to change the design of the hypervisor.

It is clear that we need an approach to decide how to mix ARM and Thumb code that simultaneously provides compact code size and good performance. This is where we can use the information that we extracted from the VAP profiling tools in order to obtain small footprint without causing loss in performance. As a starting point we will move as much code as possible into thumb code and analyze the benchmark and size. As a second step, we will look at the hypervisor functions in where most execution time is spent. If moving these functions to ARM code generate increased speed with an acceptable small increase in code size, we will choose to move it to ARM code instead. Where it is possible, the remainder of the program will be compiled into Thumb code.

Table 4.8 on the left side shows the size of the hypervisor that has been compiled into ARM instructions while Table 4.9 on the right side shows the size of the hypervisor that has been compiled into Thumb instructions.

Section	**Size**
.text	456
.data	0
.bss	0
.rodata	284
.hyper_data	536
.hyper_functions	2840
Total	**4116**

Table 4.8: Size of hyper.o

Section	**Size**
.text	4732
.data	0
.bss	0
.rodata	2456
.hyper_data	0
.hyper_functions	220
Total	**7408**

Table 4.9: Size of hyperThumb.o

Adding the size of these two files together, the hypervisor is **11 524** bytes. Introducing THUMB mode has thus saved us 2032 bytes, a 15,0% decrease in footprint counting from the last optimization. This has increased the cycle count to

[5]Thumb was discussed in section 2.2.2

13,798,549, an increase with 4,4% compared to our baseline value **13,221,812** which is a acceptable overhead.

Our next step was to investigate if moving the most executed functions to ARM code would achieve any significant increase in performance. Looking at Table 4.5 we chose to move hyperCallImpl, hyperChangeGuestMode, hypercallRPC and hypercallEndRPC to ARM code and analyze the data. The other functions with high counts were already located in ARM code. The results are in Table 4.10.

	Size (bytes)			Performance (instructions)		
Function	**Thumb**	**ARM**	**%**	**Thumb**	**ARM**	**%**
hypercallImpl	11 524	11 548	0,2%	13,798,548	13,638,739	1,2%
hyperChangeGuestMode	11 548	11 652	0,9%	13,638,739	13,362,074	2,1%
hypercallRPC & hypercallEndRPC	11 652	12 024	3%	13,362,074	13,302,044	0,4%

Table 4.10: Benchmark

The results for moving hypercallImpl and hyperChangeGuestMode to ARM code had improved the performance with a very small size increase and was kept. However, hypercallRPC and hypercallEndRPC were moved back to Thumb code as a 3% increase in code size while only getting a 0,4% increase in performance was not worth it. The end result for the Thumb optimization is **11 548** bytes in total size for the hypervisor and **13,362,074** instruction cycles. A total of **14,8%** in reduction of code size and **1.0%** increase in instruction counts.

4.4.4 GCC Optimization flags

There are four different optimization flags O1, O2, O3 and Os where the last one optimizes the code with the size in focus. Table 4.11 and 4.12 shows the hypervisor compiled with the optimization flag -Os.

Section	Size
.text	280
.data	0
.bss	0
.rodata	972
.hyper_data	536
.hyper_functions	2924
Total	**4712**

Table 4.11: Size of hyper.o

Section	Size
.text	1968
.data	0
.bss	0
.rodata	1560
.hyper_data	0
.hyper_functions	156
Total	**3684**

Table 4.12: Size of hyperThumb.o

The total size of the hypervisor is **8396** bytes and the performance is 12,558,289 instruction counts. Next we tested with the optimization flag -O3 on the ARM compiled part of the hypervisor as we know that this part contains most of the performance critical code. The results are shown in Table 4.13 and 4.14.

Section	Size
.text	292
.data	0
.bss	0
.rodata	0
.hyper_data	536
.hyper_functions	3860
Total	**4688**

Table 4.13: Size of hyper.o

Section	Size
.text	1968
.data	0
.bss	0
.rodata	1560
.hyper_data	0
.hyper_functions	156
Total	**3684**

Table 4.14: Size of hyperThumb.o

The total size of the hypervisor is now **8372** bytes with a instruction count of 12,335,662. It was expected that the -O3 flag would increase the performance of the hypervisor, it was however unexpected that it would also decrease the size.

4.4.5 Summary

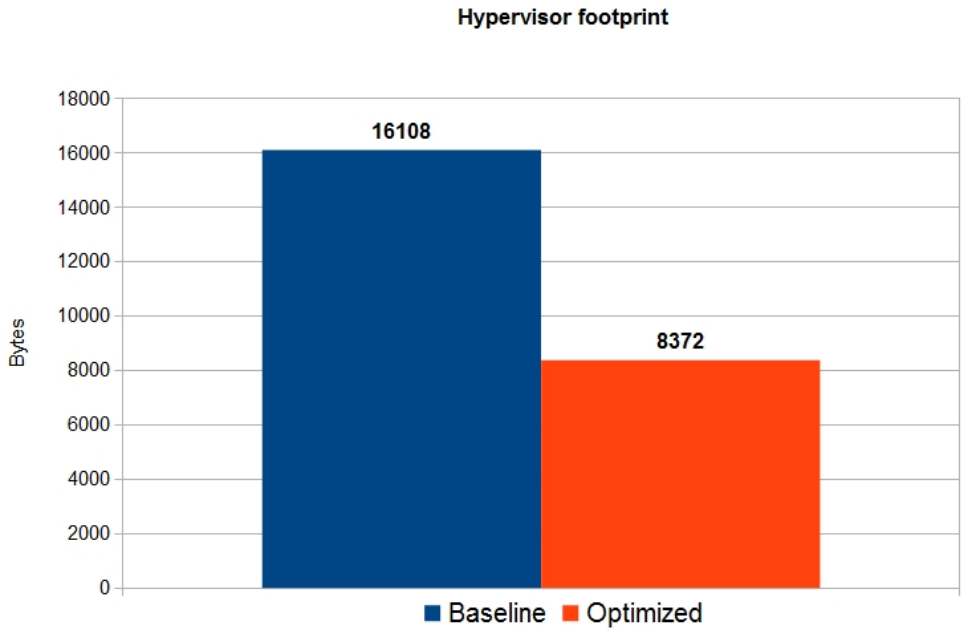

Figure 4.1: Footprint of the hypervisor

After applying all the mentioned optimizations, the size of the hypervisor are shown in Figure 4.1 and the benchmark are shown in Table 4.2. We have managed to reduce the size of the hypervisor by 52% and at the same time managed to keep the performance overhead to an acceptable level.

Looking at the comparison in performance for the baremetal kernel (no hypervisor), the performance overhead was at 0.4% for the benchmark MathTest that we managed to performance optimize. However as time was limited and focused on optimization on the footprint, we did not have time to optimize the context switching and guest mode transition on the hypervisor. The consequence was that YieldingMathTest had an overhead as high as 11.1% while WrapperMathTest had an overhead of 7.9%. It should however be remembered that these benchmarks demonstrates the exaggerated use of hypercalls, context switching and kernel to task transitions.

Also if time allowed, one could continue and reduce the footprint of the hypervisor. One example is to have the possibility to configure what kind of functionality that you want for the hypervisor. By removing unwanted functionality, the footprint can be reduced even further so that the hypervisor can be used in extremely memory limited systems. For example, the DMA functionality adds up to 1888

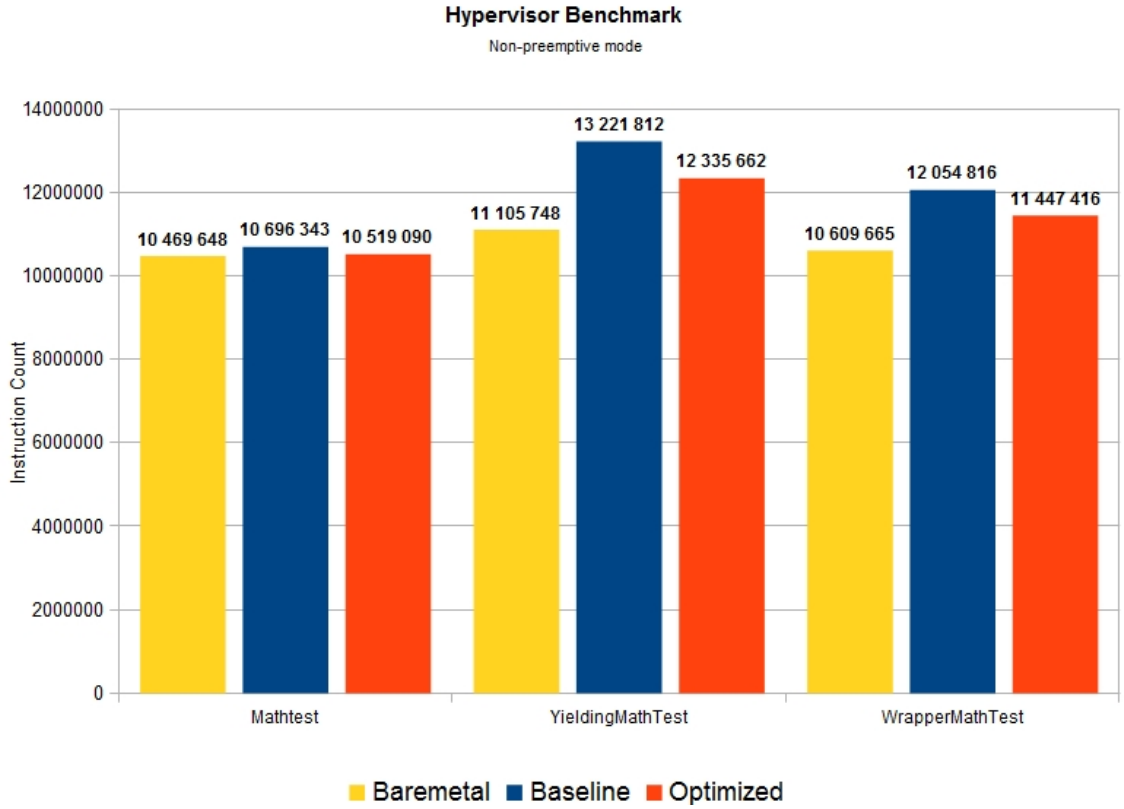

Figure 4.2: Benchmark performance of hypervisor

bytes in total[6], reducing the size of the hypervisor to 6484 bytes, if it is decided that DMA functionality is not needed. A simple hypervisor configuration script could easily be written to remove these parts if it is ever needed.

As the goal with the thesis was to optimize the size of the hypervisor, we feel that it was achieved as the footprint of the hypervisor has decreased by 52%, to 8372 bytes.

[6]Used arm-elf-nm tool on the optimized hypervisor to control the size of DMA functions

Chapter 5

Conclusion

The thesis work was not easy as a major part of the hypervisor is written in ARM assembly code. This was mainly because it is the only way to communicate with hardware, peripherals and the processor registers. The second reason is because; in some situations the compiler does not have as much information as the developer regarding the system, in these cases hand written assembly results in more efficient code.

When things went wrong, it was very hard and burdensome to debug the environment. Debugging assembler code such as hypercalls required dumping the stack and processor registers on the debug console after each instruction step, which requires great patience. The learning curve in the beginning of the thesis was very high, as I had no prior experience of working with hypervisors, the ARM architecture and the hardware emulation platform OVP. It was thus very suitable to begin the thesis with the implementation of the security service in order to get familiar with the developing environment.

As a result of this thesis, we have demonstrated that with the help of the SICS developed hypervisor, we managed to increase the security of our embedded system. With the hypervisor offering multiple virtual guest modes, each with its own execution environment, we can enforce an access policy that is secure for our security critical applications. This was demonstrated by implementing a security service that offered cryptographic services such as AES-128, RSA-1024 and SHA-256. By storing all the cryptographic keys in the isolated trusted domain, we could make sure that it is kept safe from all other software. This successfully demonstrates the usability and power of the hypervisor.

As for the second part of the thesis, the goal was to optimize the hypervisor with respect to footprint and overall performance. With the help of the OVP profiling tools, we could extract useful information on how to approach the optimization of the hypervisor. The goal was to reduce the footprint of the hypervisor while keeping the performance at an acceptable level. The results shows that this was also achieved. The hypervisor's footprint decreased by 52% while the performance increased in all benchmarks if we compare it to the baseline hypervisor. However, comparing the performance against the baremetal kernel gives a performance over-

head of 11% at most[1] which certainly can be improved.

Future Work

While we successfully managed to demonstrate the power of the hypervisor in this thesis, the hypervisor was built to support the very simple operating system FreeRTOS. This is not a rich OS such as Linux, and for future work it is desired that a suitable Linux derivative is ported to work with the SICS hypervisor.

Formal verification of the hypervisor is also desired to guarantee security with the assurance level of a mathematical proof. In fact, Oliver Schwarz which is a PhD student at KTH and employed by SICS, is currently working on formal verification of the hypervisor.

Currently the hypervisor supports the ARMv5 architecture, specifically the ARM926EJ-S CPU. It would be advantageous if the hypervisor could support additional hardware platforms such as the new ARMv7 architecture as it has additional hardware support for virtualization.

[1] YieldingMathTest benchmark

Appendix A

Source Code

A.1

Listing A.1: Hypervisor config file

```
#include "hyperconfigbase.h"
#include "hyperconfig.h"
#include "hyper.h"

extern void kernelRPCHandler(unsigned callNum, void* params);
extern void trustedRPCHandler(unsigned callNum, void* params);
extern void appMain(void);
extern void kernelTickHandler(void);
extern void dmaAppHandler(uint32_t channel);
extern void dmaTrustedAppHandler(uint32_t channel);

/* Linker symbols. */
extern const int __fiq_sp__, __irq_sp__, __svc_sp__, __abt_sp__,
    __und_sp__, __sys_sp__,
  __kernel_sp__, __trusted_sp__, __interrupt_sp__, __page_size__,
  __hyper_mb_start__, __hyper_mb_count__, __hyper_mapped_pages__,
  __trusted_mb_start__, __trusted_mb_count__, __trusted_mapped_pages__
      ,
  __kernel_mb_start__, __kernel_mb_count__, __kernel_mapped_pages__,
  __task_mb_start__, __task_mb_count__, __task_mapped_pages__,
  __shared_libs_mb_start__, __shared_libs_mb_count__,
      __shared_libs_mapped_pages__,
  __guest_entry_sp__,
  __kernel_transition_start__, __kernel_transition_end__,
  __dom_pool_task_mb_start__,
  __shared_rpc_mb_start__,
  __sl_pt_size__;

/* Standard definitions of Mode bits and Interrupt (I & F) flags in
    PSRs */
#define MODE_USR 0x10            /* User Mode */
#define MODE_FIQ 0x11            /* FIQ Mode */
#define MODE_IRQ 0x12            /* IRQ Mode */
```

```
#define MODE_SVC 0x13            /* Supervisor Mode */
#define MODE_ABT 0x17            /* Abort Mode */
#define MODE_UND 0x1B            /* Undefined Mode */
#define MODE_SYS 0x1F            /* System Mode */
#define I_BIT 0x80               /* when I bit is set, IRQ is disabled
    */
#define F_BIT 0x40               /* when F bit is set, FIQ is disabled
    */

#define HC_DOM_DEFAULT 0
#define HC_DOM_TASK 1
#define HC_DOM_KERNEL 2
#define HC_DOM_TRUSTED 3

#define HC_DOM_SHARED_RPC 9
#define HC_DOM_FLASH 10

#define HC_DOM_POOL_TASK_BITMAP \
  ((1 << HC_DOMPOOL_TASK_BASE) | \
   (1 << (HC_DOMPOOL_TASK_BASE + 1)) | \
   (1 << (HC_DOMPOOL_TASK_BASE + 2)) | \
   (1 << (HC_DOMPOOL_TASK_BASE + 3)) | \
   (1 << (HC_DOMPOOL_TASK_BASE + 4)))

#define HC_DOMAC_POOL_TASK_ALL \
  ((1 << (2 * HC_DOMPOOL_TASK_BASE)) | \
  (1 << (2 * (HC_DOMPOOL_TASK_BASE + 1))) | \
  (1 << (2 * (HC_DOMPOOL_TASK_BASE + 2))) | \
  (1 << (2 * (HC_DOMPOOL_TASK_BASE + 3))) | \
  (1 << (2 * (HC_DOMPOOL_TASK_BASE + 4))))


HYPER_DATA static const HC_MemDomain
  DomDefault = {.name = "default"},
  DomTask = {.name = "task"},
  DomKernel = {.name = "kernel"},
  DomTrusted = {.name = "trusted"},
  DomTaskPool0 = {.name = "task_pool_0"},
  DomTaskPool1 = {.name = "task_pool_1"},
  DomTaskPool2 = {.name = "task_pool_2"},
  DomTaskPool3 = {.name = "task_pool_3"},
  DomTaskPool4 = {.name = "task_pool_4"},
  DomSharedRPC = {.name = "shared_rpc"},
  DomFlash = {.name = "flash"};

#define HC_DOMAC_ALL \
  ((1 << (2 * HC_DOM_DEFAULT)) | \
  (1 << (2 * HC_DOM_TASK)) | \
  (1 << (2 * HC_DOM_KERNEL)) | \
  (1 << (2 * HC_DOM_TRUSTED)) | \
  (1 << (2 * HC_DOM_SHARED_RPC)) | \
  (1 << (2 * HC_DOM_FLASH)) | \
  HC_DOMAC_POOL_TASK_ALL)
```

A.1.

```
#define HC_DOMAC_KERNEL  \
  ((1 << (2 * HC_DOM_DEFAULT)) | \
  (1 << (2 * HC_DOM_TASK)) | \
  (1 << (2 * HC_DOM_KERNEL)) | \
  (1 << (2 * HC_DOM_SHARED_RPC)) | \
  (1 << (2 * HC_DOM_FLASH)) | \
  HC_DOMAC_POOL_TASK_ALL)

#define HC_DOMAC_TRUSTED \
  (1 << (2 * HC_DOM_DEFAULT)) | \
  (1 << (2 * HC_DOM_TRUSTED)) | \
  (1 << (2 * HC_DOM_FLASH)) | \
  (1 << (2 * HC_DOM_SHARED_RPC))

#define HC_DOMAC_INTERRUPT HC_DOMAC_ALL

#define HC_DOMAC_TASK \
  ((1 << (2 * HC_DOM_DEFAULT)) | \
  (1 << (2 * HC_DOM_TASK)) | \
  (1 << (2 * HC_DOM_FLASH)) | \
  HC_DOMAC_POOL_TASK_ALL)

#define HC_CAP_TASK \
  (HC_CAP_RPC | \
  HC_CAP_BEGIN_TRANSITION)

#define HC_CAP_INTERRUPT \
  (HC_CAP_GET_MODE_CONTEXT | \
  HC_CAP_SET_MODE_CONTEXT | \
  HC_CAP_RPC | \
  HC_CAP_RESTORE_MODE)

#define HC_CAP_KERNEL \
  (HC_CAP_RPC | \
  HC_CAP_DISABLE_INTERRUPTS | \
  HC_CAP_ENABLE_INTERRUPTS | \
  HC_CAP_SET_MODE_CONTEXT | \
  HC_CAP_GET_MODE_CONTEXT)

#define HC_CAP_TRUSTED \
  (HC_CAP_DISABLE_INTERRUPTS | \
  HC_CAP_ENABLE_INTERRUPTS | \
  HC_CAP_SET_MODE_CONTEXT | \
  HC_CAP_GET_MODE_CONTEXT)

/*
 * Bitmask constants for specifying guest mode
 * contexts that can be get/set.
 */
#define HC_GM_TRUSTED_MASK   (1 << HC_GM_TRUSTED)
#define HC_GM_KERNEL_MASK    (1 << HC_GM_KERNEL)
#define HC_GM_INTERRUPT_MASK (1 << HC_GM_INTERRUPT)
#define HC_GM_TASK_MASK      (1 << HC_GM_TASK)
```

```
#define HC_GM_ALL_MASK \
  (HC_GM_TRUSTED_MASK | \
  HC_GM_KERNEL_MASK | \
  HC_GM_INTERRUPT_MASK | \
  HC_GM_TASK_MASK)

#define HC_PAGE_SIZE LINKER_SYM_UINT(__page_size__)

#define HC_RPC_KERNEL_MASK (1 << HC_RPCHANDLER_KERNEL)
#define HC_RPC_TRUSTED_MASK (1 << HC_RPCHANDLER_TRUSTED)

#define HC_TRANSITION_KERNEL_MASK (1 << HC_TRANSITION_KERNEL)

HYPER_DATA static const HC_GuestMode
  GM_Trusted = { .name = "trusted",
    .domainAC = HC_DOMAC_TRUSTED,
    .domainPool = -1,
    .capabilities = HC_CAP_TRUSTED,
    .rpcHandlers = 0,
    .transitionAreas = 0,
    .setModeContexts = 0,
    .getModeContexts = 0,
    .restoreModes = 0,
    .dmaAllowed = true,
    .shadowDMAC = NULL,
    .dmaHandler = dmaTrustedAppHandler},
  GM_Kernel = { .name = "kernel",
    .domainAC = HC_DOMAC_KERNEL,
    .domainPool = -1,
    .capabilities = HC_CAP_KERNEL,
    .rpcHandlers = HC_RPC_TRUSTED_MASK,
    .transitionAreas = 0,
    .setModeContexts = HC_GM_TASK_MASK,
    .getModeContexts = HC_GM_TASK_MASK,
    .restoreModes = HC_GM_TASK_MASK,
    .dmaAllowed = true,
    .shadowDMAC = NULL,
    .dmaHandler = NULL},
  GM_Interrupt = { .name = "interrupt",
    .domainAC = HC_DOMAC_INTERRUPT,
    .domainPool = -1,
    .capabilities = HC_CAP_INTERRUPT,
    .rpcHandlers = 0,
    .transitionAreas = 0,
    .setModeContexts = HC_GM_TASK_MASK,
    .getModeContexts = HC_GM_TASK_MASK,
    .restoreModes = HC_GM_TASK_MASK,
    .dmaAllowed = false,
    .shadowDMAC = NULL,
    .dmaHandler = NULL},
  GM_Task = { .name = "application",
    .domainAC = HC_DOMAC_TASK,
    .domainPool = HC_DOMAINPOOL_TASK,
    .capabilities = HC_CAP_TASK,
```

A.1.

```
      .rpcHandlers = HC_RPC_KERNEL_MASK,
      .transitionAreas = HC_TRANSITION_KERNEL_MASK,
      .setModeContexts = 0,
      .getModeContexts = 0,
      .restoreModes = 0,
      .dmaAllowed = true,
      .shadowDMAC = NULL,
      .dmaHandler = dmaAppHandler};

  HYPER_DATA static const HC_RPCHandler
    RPCHandler_Kernel = {
      .name = "kernel_rpc_handler",
      .mode = HC_GM_KERNEL,
      .entryPoint = kernelRPCHandler,
      .sp = LINKER_SYM_UINT(__kernel_sp__)
      },
    RPCHandler_Trusted = {
      .name = "trusted_rpc_handler",
      .mode = HC_GM_TRUSTED,
      .entryPoint = trustedRPCHandler,
      .sp = LINKER_SYM_UINT(__trusted_sp__)
      };

  HYPER_DATA static const HC_DomainPool DomPoolTask = {
    .name = "domain_pool_task",
    .domains = HC_DOM_POOL_TASK_BITMAP};


  /* many linker symbols */
  HYPER_DATA static const HC_MemRegion
    MemHyper = {.name = "hypervisor",
      .mbStart = LINKER_SYM_UINT(__hyper_mb_start__),
      .mbCount = LINKER_SYM_UINT(__hyper_mb_count__),
      .mappedPages = LINKER_SYM_UINT(__hyper_mapped_pages__),
      .accessControl = HC_MEM_USERNOACCESS,
      .domain = HC_DOM_DEFAULT},
    MemTask = {.name = "task",
      .mbStart = LINKER_SYM_UINT(__task_mb_start__),
      .mbCount = LINKER_SYM_UINT(__task_mb_count__),
      .mappedPages = LINKER_SYM_UINT(__task_mapped_pages__),
      .accessControl = HC_MEM_USERREADWRITE,
      .domain = HC_DOM_TASK},
    MemKernel = {.name = "kernel",
      .mbStart = LINKER_SYM_UINT(__kernel_mb_start__),
      .mbCount = LINKER_SYM_UINT(__kernel_mb_count__),
      .mappedPages = LINKER_SYM_UINT(__kernel_mapped_pages__),
      .accessControl = HC_MEM_USERREADWRITE,
      .domain = HC_DOM_KERNEL},
    MemTrusted = {.name = "trusted",
      .mbStart = LINKER_SYM_UINT(__trusted_mb_start__),
      .mbCount = LINKER_SYM_UINT(__trusted_mb_count__),
      .mappedPages = LINKER_SYM_UINT(__trusted_mapped_pages__),
      .accessControl = HC_MEM_USERREADWRITE,
      .domain = HC_DOM_TRUSTED},
```

```
MemSharedLibs = {.name = "shared_libs",
  .mbStart = LINKER_SYM_UINT(__shared_libs_mb_start__),
  .mbCount = LINKER_SYM_UINT(__shared_libs_mb_count__),
  .mappedPages = LINKER_SYM_UINT(__shared_libs_mapped_pages__),
  .accessControl = HC_MEM_USERREADWRITE,
  .domain = HC_DOM_DEFAULT},
MemTaskPool0 = {.name = "task_pool_0",
  .mbStart = LINKER_SYM_UINT(__dom_pool_task_mb_start__),
  .mbCount = 1,
  .mappedPages = 256,
  .accessControl = HC_MEM_USERREADWRITE,
  .domain = HC_DOMPOOL_TASK_BASE},
MemTaskPool1 = {.name = "task_pool_1",
  .mbStart = LINKER_SYM_UINT(__dom_pool_task_mb_start__) + 1,
  .mbCount = 1,
  .mappedPages = 256,
  .accessControl = HC_MEM_USERREADWRITE,
  .domain = HC_DOMPOOL_TASK_BASE + 1},
MemTaskPool2 = {.name = "task_pool_2",
  .mbStart = LINKER_SYM_UINT(__dom_pool_task_mb_start__) + 2,
  .mbCount = 1,
  .mappedPages = 256,
  .accessControl = HC_MEM_USERREADWRITE,
  .domain = HC_DOMPOOL_TASK_BASE + 2},
MemTaskPool3 = {.name = "task_pool_3",
  .mbStart = LINKER_SYM_UINT(__dom_pool_task_mb_start__) + 3,
  .mbCount = 1,
  .mappedPages = 256,
  .accessControl = HC_MEM_USERREADWRITE,
  .domain = HC_DOMPOOL_TASK_BASE + 3},
MemTaskPool4 = {.name = "task_pool_4",
  .mbStart = LINKER_SYM_UINT(__dom_pool_task_mb_start__) + 4,
  .mbCount = 1,
  .mappedPages = 256,
  .accessControl = HC_MEM_USERREADWRITE,
  .domain = HC_DOMPOOL_TASK_BASE + 4},
MemSharedRPC = {.name = "shared_rpc",
  .mbStart = LINKER_SYM_UINT(__shared_rpc_mb_start__),
  .mbCount = 1,
  .mappedPages = 256,
  .accessControl = HC_MEM_USERREADWRITE,
  .domain = HC_DOM_SHARED_RPC},
MemFlash = {.name = "flash",
  .mbStart = 0x10,
  .mbCount = 1,
  .mappedPages = 256,
  .accessControl = HC_MEM_USERREADWRITE,
  .domain = HC_DOM_FLASH},
/* Devices occupy last megabyte in memory. */
MemDevices = {.name = "devices",
  .mbStart = 0xFFF,
  .mbCount = 1,
  .mappedPages = 0x100,
  .accessControl = HC_MEM_USERNOACCESS,
```

A.1.

```
      .domain = HC_DOM_DEFAULT};

  HYPER_DATA static const HC_TransitionArea TransitionAreaKernel = {
    .name = "KernelTrans",
    .fromMode = HC_GM_TASK,
    .toMode = HC_GM_KERNEL,
    .startAddress = LINKER_SYM_UINT(__kernel_transition_start__),
    .endAddress = LINKER_SYM_UINT(__kernel_transition_end__)
  };

  HC_Config hyperConfig = {
    .guestEntryPoint = appMain,
    .guestEntrySP = LINKER_SYM_UINT(__guest_entry_sp__),
    .guestEntryMode = HC_GM_TASK,
    .privModes = {
      {.sp = LINKER_SYM_UINT(__fiq_sp__), .initialCPSR = MODE_FIQ |
          I_BIT | F_BIT},
      {.sp = LINKER_SYM_UINT(__irq_sp__), .initialCPSR = MODE_IRQ |
          I_BIT | F_BIT},
      {.sp = LINKER_SYM_UINT(__svc_sp__), .initialCPSR = MODE_SVC |
          I_BIT | F_BIT},
      {.sp = LINKER_SYM_UINT(__abt_sp__), .initialCPSR = MODE_ABT |
          I_BIT | F_BIT},
      {.sp = LINKER_SYM_UINT(__und_sp__), .initialCPSR = MODE_UND |
          I_BIT | F_BIT},
      {.sp = LINKER_SYM_UINT(__sys_sp__), .initialCPSR = MODE_SYS |
          I_BIT | F_BIT}},
    .memDomains =
      {&DomDefault,
      &DomTask,
      &DomKernel,
      &DomTrusted,
      &DomTaskPool0,
      &DomTaskPool1,
      &DomTaskPool2,
      &DomTaskPool3,
      &DomTaskPool4,
      &DomSharedRPC,
      &DomFlash, 0, 0, 0, 0, 0},
    .guestModes = {&GM_Trusted, &GM_Kernel, &GM_Interrupt, &GM_Task},
    .rpcHandlers = {&RPCHandler_Kernel, &RPCHandler_Trusted},
    .transitionAreas = {&TransitionAreaKernel},
    .memRegions = {&MemHyper, &MemTask, &MemKernel, &MemTrusted,
      &MemSharedLibs, &MemTaskPool0, &MemTaskPool1, &MemTaskPool2,
      &MemTaskPool3, &MemTaskPool4, &MemDevices,&MemSharedRPC,&MemFlash
          },
    .domainPools = {&DomPoolTask},
    .interruptConfig = {
      .interruptMode = HC_GM_INTERRUPT,
      .sp = LINKER_SYM_UINT(__interrupt_sp__),
      .tickHandler = kernelTickHandler
    }
  };
```

A.2

Listing A.2: Trusted service

```
/*
 * trustedservice.c
 *
 *  Created on: 30 mar 2011
 *      Author: Viktor Do
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "trusted_service.h"
#include "dmacRegisters.h"
#include "hyper.h"
#include "impTypes.h"
#include "rsa.h"
#include "aescbc.h"
#include "sha2.h"
#include "mprsa.h"
#include "report.h"

//TODO remove buffer and use only decrypted for immediate storage?
    Check string lengths in program.
//TODO Use memset to reset sensitive memory

FLASH_DATA static unsigned char *encryptedStorage[10];  // vector with
     pointer to encrypted data
FLASH_DATA static unsigned char encryptedAESKey[128]; //key is
    encrypted with RSA

TRUSTED_DATA static char decrypted[50000];
TRUSTED_DATA static unsigned char buffer[50000+16];  //buffer to hold
    the transfered data from flash

TRUSTED_DATA static unsigned char decryptedAESKey[128]; //key is
    encrypted with RSA
//TRUSTED_DATA static unsigned int  nbytes;

TRUSTED_DATA static unsigned int nbytesVector[10];

// 1024-bit RSA keys in HEX
//The RSA-keys are generated from http://www.mobilefish.com/services/
    rsa_key_generation/rsa_key_generation.php
TRUSTED_DATA static char *g_modulus =
  "f0f4fa85b4ad3f6d9171995085b31640c4c8ed28e5c9eb5106f62acea46ee83c"
  "0c575f03ab918d9aee62fdd5fc7b5a350d4775618f646583ef6a0c50985123ac"
  "4271ae3cdaba97f5d6527217971f2cc0bdbbfa2886afedfe783e1b170ca5e279"
  "e6fd07e9efffd99c1f4f35c30644f86227cfc32a5253a89ebfb22862f1085b35";

TRUSTED_DATA static char *g_d =
  "e2d4cc0e18834b859af8c4ea6fa2a29d406322177112bfaa9c921ac4433980f8"
  "1e6a15b0ffcf5aedf1e250b12428ff479803a035c26631c69d18491589fe4043"
```

A.2.

```
  "f2cf8d591c93256dda125bb1466a2199fab20081b6806ddda740b73e35e73c63"
  "e794ba34197aa423064286feb2e019b4521a05405b27b2f232619a00bedeac95";

TRUSTED_DATA static char *sign_modulus =
  "dcc03cf173394f9befe0a2d51fcd4b24e952efcd0b657663826d6d3ad3c94e5e"
  "219d8e5353c842356247f9a1e4a2dd6b20b17163127cfd2c0cdcc93788fb63b3"
  "cee562af937530a2f2a1036b2bf4a12de36480e92b32ec8d1217d1579a471de1"
  "5295ee666cf7638af85acc2fcc9efc61bea0c930c776dcea792fd2694fb85fdd";

TRUSTED_DATA static char *sign_d =
  "a8b6a3dd455affe50628814ab1cb8d2ae0c86a4e23ef9fd3ddd3143069bce910"
  "3850da7e050280d79c0db6546d11ac783bbc62147e04d8d9d9dac44e957acc6f"
  "2c94b8d78e1df4c0fb0b40e05cf1cbb35337633afe48cc2c2fbe052ef31f6e08"
  "80fe7aab8a78b8dcd2472d383e923146031d6c4a6b217340beda6b19fa7c3761";

TRUSTED_DATA static char *g_e = "10001";

////////////////////   MACROS   //////////////////

#define LOGT(_FMT, ...)  tellf(REP_TRUSTED_NAME, _FMT,  ## __VA_ARGS__
    )

////////////////////////////////////////////////////

static void useDMA(int dataID)                  TRUSTED_FUNCTION;
void trustedRPCHandler(unsigned callNum, void* params)
    TRUSTED_FUNCTION;
static void finishRPC()                      TRUSTED_FUNCTION;
static void initFlashData(TrustedArgs *args)          TRUSTED_FUNCTION
    ;
static void calculateSHA(char *message, unsigned char *hval)
    TRUSTED_FUNCTION;
static void getSignature(TrustedArgs *args)           TRUSTED_FUNCTION
    ;
static void signContract(TrustedArgs *args)           TRUSTED_FUNCTION
    ;
static void verifySignature(TrustedArgs *args)
    TRUSTED_FUNCTION;

//void srand_mwc(); //pseudo random number generator

static inline void writeReg32(Uns32 address, Uns32 offset, Uns32 value
    )
{
    *(volatile Uns32*) (address + offset) = value;
}

static inline Uns32 readReg32(Uns32 address, Uns32 offset)
{
    return *(volatile Uns32*) (address + offset);
}

static void dmaBurst(Uns32 ch, void *from, void *to, Uns32 bytes)
{
```

```
    Uns32 offset = ch * DMA_CHANNEL_STRIDE;
 // printf("dmaBurst ch:%d  bytes:%d\n", ch, bytes);
    writeReg32(DMA_BASE, DMA_C0_SRC_ADDR + offset, (Uns32)from);
    writeReg32(DMA_BASE, DMA_C0_DST_ADDR + offset, (Uns32)to);
    writeReg32(DMA_BASE, DMA_C0_CONTROL  + offset, bytes /*| 0
        x80000000*/);
    writeReg32(DMA_BASE, DMA_C0_CONFIGURATION + offset, 0x8001);
}

/*Called to initialize the encrypted data and encrypted aes key in
    flash data**/
static void initFlashData(TrustedArgs *args){
  int aesErr = 0; //error handling
  int rsaErr = 0;
  int i;
//  hello(REP_TRUSTED_NAME);
//  tell(REP_TRUSTED_NAME,"In trusted mode initFlashData");
  TRUSTED_DATA static char data[50000] = "Transfer of 20.000 SEK from
      Sven Svensson (Nordea, 1234-56789) to Anna Svensson (SEB,
      987654321). ";
  TRUSTED_DATA static char data1[50000] = "Buying 5 shares of the
      Ericsson stock."; //TODO cant get atmel_ram.ld to put the string
       in trusted rodata, it puts it in kernel space

  TRUSTED_DATA static char *dataStorage[10];

  dataStorage[0] = data;
  dataStorage[1] = data1;
  FLASH_DATA static unsigned char encrypted[50000+16]; //data in flash
       memory is encrypted (16 is size of IV added to message)
  FLASH_DATA static unsigned char encrypted1[50000+16];

  encryptedStorage[0] = encrypted;
  encryptedStorage[1] = encrypted1;

  nbytesVector[0] = strlen(data); //setting the global variable, used
      in aesDecrypt
  nbytesVector[1] = strlen(data1);

//32 bytes HEX key digits used as key to AES-128
//TRUSTED_DATA static char sessionKey[33] = "0123456789
    abcdeffedcba9876543210"; AES key
    char sessionKey[33];  //TODO memset this area to 0 at end of
        function, dont need this is trusted domain?
    generateAESKey(sessionKey); //generates a random 16 byte AES key (
        AES-128)
//tellf(REP_TRUSTED_NAME,"Generated a AES sessionKey: %s",sessionKey);
//tell(REP_TRUSTED_NAME,"Encrypting AES session key with RSA");
  rsaErr = rsaEncrypt(sessionKey,g_e,g_modulus,encryptedAESKey);
  if(rsaErr != 0){
    tell(REP_TRUSTED_NAME,"RSA encryption failed.\n");
    *args->success = 0;
  }
  else{
```

A.2.

```
//    tellf(REP_TRUSTED_NAME,"Encrypting message with random generated
     AES-128 one time session key:\n\t\"%s\"\n",sessionKey);

      for(i = 0;i < 2; i++){ //how many messages to encrypt
        aesErr = 0;

      aesErr = aesEncrypt(sessionKey,dataStorage[i],encryptedStorage[i
          ],i);
      if(aesErr != 0){
        tell(REP_TRUSTED_NAME,"AES encryption failed.\n");
        *args->success = 0;
      }
      else{
//        tellf(REP_TRUSTED_NAME,"Encryption successfull,trying to
    print encrypted data! \n %s \n\n", encryptedStorage[i]);
        *args->success = 1;
      }
      }
  }
}

volatile static bool bReceived[2] = {false, false};

/**DMA Contract into trusted domain space, decrypt and send back to
    task*/
static void getSignature(TrustedArgs *args){

  int rsaErr = 0, aesErr = 0;

  hello(REP_TRUSTED_NAME);
  tell(REP_TRUSTED_NAME,"In trusted mode getSignature()");
  tell(REP_TRUSTED_NAME,"Starting DMA transfer of the encrypted
      document and key from external memory into trusted domain.");
  useDMA(args->dataID); //DMA the encrypted data into trusted space

  while(!bReceived[0] || !bReceived[1]);

  rsaErr = rsaDecrypt(encryptedAESKey,g_d,g_modulus,decryptedAESKey);
  if(rsaErr != 0){
    tell(REP_TRUSTED_NAME,"RSA encryption failed.\n");
    *args->success = 0;
  }
  else{
    aesErr = aesDecrypt(decryptedAESKey,encryptedStorage[args->dataID
        ],decrypted,nbytesVector[args->dataID]);
  }

  if(aesErr != 0){
    tell(REP_TRUSTED_NAME,"AES encryption failed.\n");
    *args->success = 0;
  }
  else if(aesErr == 0 && rsaErr == 0){
    *(args)->success = 1;
    signContract(args);
```

```
  }

}

static void calculateSHA(char *message, unsigned char *hval){
  sha256_ctx    sha_ctx[1];

  sha256_begin(sha_ctx);
  sha256_hash(message, strlen(decrypted), sha_ctx);
  sha256_end(hval, sha_ctx);
}

static void signContract(TrustedArgs *args){
  hello(REP_TRUSTED_NAME);
  tell(REP_TRUSTED_NAME,"In trusted mode signContract()");

  sha256_ctx    sha_ctx[1];
  unsigned char hval[SHA256_DIGEST_SIZE];
  sha256_begin(sha_ctx);
  sha256_hash(decrypted, strlen(decrypted), sha_ctx);
  sha256_end(hval, sha_ctx);
  tell(REP_TRUSTED_NAME,"SHA-256 DIGEST of message:");
  pr_hex_block(REP_TRUSTED_NAME,hval,SHA256_DIGEST_SIZE,1);

  //e is default 0x10001;
  mp_int  modulus, d;
  mp_err err;
  int olen;

  mp_init(&modulus);
  mp_init(&d);

  mp_read_radix(&modulus, sign_modulus, 16);
  mp_read_radix(&d, sign_d, 16);

  args->signature = malloc(129);

  err = mp_pkcs1v15_sign(hval,SHA256_DIGEST_SIZE,&d, &modulus,&args->
      signature,&olen);
  if(err != 0){
    tell(REP_TRUSTED_NAME,"Signature failed in Application.\n");
  }
  else{
    tell(REP_TRUSTED_NAME,"Signature successfully generated, passing
        to App!");
  }

  mp_clear(&d);
  mp_clear(&modulus);

}

/*Should be provided by extern source, here for demonstration purpose
    **/
```

A.2.

```
static void verifySignature(TrustedArgs *args){
  hello(REP_TRUSTED_NAME);
  tell(REP_TRUSTED_NAME,"In trusted mode verifySignature()");

  char *ptx;
  int olen;

  unsigned char *g_e = "10001"; //public key is always 0x10001, will
      not effect security

  mp_int  modulus, e;
  mp_err err;
  mp_init(&modulus);
  mp_init(&e);

  mp_read_radix(&modulus, sign_modulus, 16);
  mp_read_radix(&e, g_e, 16);
  err = mp_pkcs1v15_verify(args->signature,128,&e,&modulus,&ptx,&olen)
      ;
  if(err != 0){
    tell(REP_TRUSTED_NAME,"Verify Signature failed\n");
  }
  else{
    tell(REP_TRUSTED_NAME,"This is the hash value we got from senders
        signature");
    pr_hex_block(REP_TRUSTED_NAME,ptx,SHA256_DIGEST_SIZE,1); //print
        hashvalue that we got from signature //crashes here sometimes

//  decrypted[5] = 'g';//sabotage message
    tell(REP_TRUSTED_NAME,"This is our own calculated hash value!");
    unsigned char hval[SHA256_DIGEST_SIZE];
    calculateSHA(decrypted,hval); //input contact, output hvalue

    tell(REP_TRUSTED_NAME,"SHA-256 TRUSTED DIGEST:");
    pr_hex_block(REP_TRUSTED_NAME,hval,SHA256_DIGEST_SIZE,1);

    if(!strncmp(hval,ptx,32)){
      tell(REP_TRUSTED_NAME,"Message signature is valid!");
    }
    else
      tell(REP_TRUSTED_NAME,"Message signature is wrong. Signature
          failed\n");

    mp_clear(&e);
    mp_clear(&modulus);
  }
}

static void useDMA(int dataID){

  // write to DMAC registers to start burst
  int nbytes = nbytesVector[dataID];
  //We use the two channels to send the encrypted message into the
      buffer
```

```
  if(nbytes % 2){ //odd bytes
    dmaBurst(0, encryptedStorage[dataID], buffer, ((nbytes-1) / 2));
        //transfer encrypted data from flash to buffer in trusted
        domain
    dmaBurst(1, encryptedStorage[dataID] + ((nbytes-1)/2), buffer + ((
        nbytes-1)/2), ((nbytes-1) / 2) + 1 + 16);// 16 is the size of
        the IV and 1 is the extra byte because we subtracted to get a
        even division
  }
  else{ //even bytes
    dmaBurst(0, encryptedStorage[dataID], buffer, nbytes / 2); //
        transfer encrypted data from flash to buffer in trusted domain
    dmaBurst(1, encryptedStorage[dataID] + (nbytes / 2), buffer + (
        nbytes / 2), (nbytes / 2) + 16);     // 16 is the size of the
        IV
  }

}

void trustedRPCHandler(unsigned callNum, void* params)
{
  switch(callNum){
  case 0:
    initFlashData(params);
    break;
  case 1:
    getSignature(params);
    break;
  case 2:
    verifySignature(params);
    break;
  default:
    printf("Unknown trusted operation: %d\n", callNum);
  }
  finishRPC();
}

void dmaTrustedAppHandler(uint32_t channel)
{
  bReceived[channel] = true;
  tellf(REP_TRUSTED_NAME, "DMA-data #%d received.", (channel+1));

  // go back to HV
  bye(REP_TRUSTED_NAME);
  ISSUE_HYPERCALL(HYPERCALL_END_DMA);

}

void finishRPC()
{
  ISSUE_HYPERCALL(HYPERCALL_END_RPC);
}
```

A.3

Listing A.3: Task using trusted service

```
//////////////////    INCLUDES    ////////////////////////
#include "FreeRTOS.h"
#include "task.h"
#include <stdio.h>
#include <string.h>
#include "dmacRegisters.h"
#include "impTypes.h"
#include <stdlib.h>

#include "report.h"
#include "trusted_service.h"
#include "sha2.h"
#include "mprsa.h"
#include "mwc.h"
//////////////////////   CONSTANTS   ///////////////////

#define DMAT_STACK_SIZE (configMINIMAL_STACK_SIZE*2)

#define IE_STACK_SIZE (configMINIMAL_STACK_SIZE*2)

//////////////////////   MACROS   ///////////////////

#define LOGT(_FMT, ...)  printf( "TEST DMA - TASK: " _FMT,  ##
    __VA_ARGS__)

//////////////////////   PROTOTYPES   ///////////////////

static portTASK_FUNCTION( vDMATestTask, pvParameters );
static portTASK_FUNCTION( vDMAMaliciousTask, pvParameters );

//////////////////////   HELP FUNCTIONS ///////////////////

void taskDoRPC(unsigned dest, unsigned callNum, void* params);

static inline void writeReg32(Uns32 address, Uns32 offset, Uns32 value
    )
{
    *(volatile Uns32*) (address + offset) = value;
}


static inline Uns32 readReg32(Uns32 address, Uns32 offset)
{
    return *(volatile Uns32*) (address + offset);
}


static void dmaBurst(Uns32 ch, void *from, void *to, Uns32 bytes)
{
    Uns32 offset = ch * DMA_CHANNEL_STRIDE;
```

```
    //LOGT("dmaBurst ch:%d  bytes:%d\n", ch, bytes);
    writeReg32(DMA_BASE, DMA_C0_SRC_ADDR + offset, (Uns32)from);
    writeReg32(DMA_BASE, DMA_C0_DST_ADDR + offset, (Uns32)to);
    writeReg32(DMA_BASE, DMA_C0_CONTROL  + offset, bytes /*| 0
        x80000000*/);
    writeReg32(DMA_BASE, DMA_C0_CONFIGURATION + offset, 0x8001);
}

void pr_hex(const unsigned char *bytes, int nbytes,int compact);
////////////////////   TASKS   ////////////////////

void vStartDMATestTask( unsigned portBASE_TYPE uxPriority, char**
    strings)
{
  xTaskCreate( vDMATestTask, ( signed char * ) "DMAT", DMAT_STACK_SIZE
      , (void *) strings, uxPriority, ( xTaskHandle * ) NULL );
}

void vStartMalicousTask( unsigned portBASE_TYPE uxPriority)
{
  xTaskCreate( vDMAMaliciousTask, ( signed char * ) "MAL",
      DMAT_STACK_SIZE, NULL, uxPriority, ( xTaskHandle * ) NULL );
}


static portTASK_FUNCTION( vDMATestTask, pvParameters )
{
  srand(0); //for random number generator in aescbc
  srand_mwc(); // Automatically generates a seed, must be called once
      before calling rand_mwc()

  TrustedArgs *tru_arg;
  tru_arg = malloc(sizeof(TrustedArgs));
  tru_arg->success = malloc(sizeof(int));
  tru_arg->dataID = 1; //which file to get from flash

  doRPC(RPC_KERNEL,KERNEL_RPC_TRUSTED_INIT,tru_arg); //init the
      crypted contract in flash space domain, dont need argument
//  hello(REP_APP_NAME);
//  tell(REP_APP_NAME,"Back in application after initialisation of
    encrypted data in trusted mode.");
  if(*tru_arg->success != 1)
    printf("Initialization of encrypted data failed in trusted mode\n
        ");
  else{
    hello(REP_APP_NAME);
    tell(REP_APP_NAME,"In application");
    tell(REP_APP_NAME,"Doing an RPC into trusted domain to get
        signature of document.");
    doRPC(RPC_KERNEL,KERNEL_RPC_TRUSTED_GET_SIGNATURE,tru_arg); //get
        the signature
    hello(REP_APP_NAME);
    tell(REP_APP_NAME,"Back in application after get signature in
        trusted mode.");  //should we copy it into task domain space?
```

A.3.

```
    if(*tru_arg->success != 1){
      tell(REP_APP_NAME,"Get Signature failed in trusted mode\n");
    }
    else{
      tell(REP_APP_NAME,"We are in task mode and we got this signature
           from trusted app:");
      pr_hex_block(REP_APP_NAME,tru_arg->signature,128,0);
      tell(REP_APP_NAME,"Lets Verify the signature.\n");
      doRPC(RPC_KERNEL,KERNEL_RPC_TRUSTED_VERIFY_SIGNATURE,tru_arg);
    }
  }
//  printf("Address of signature:%x\n",tru_arg->signature);
  portYIELD ();
  free(tru_arg->signature);
  free(tru_arg->success);
  free(tru_arg);
  while(1); //wait for other task to finish
  exit(0);
}


static portTASK_FUNCTION( vDMAMaliciousTask, pvParameters )
{
  portYIELD();
  int apa = 42;
  printf("\nn\tMAL Variable located in address space: %x",&apa);
  hello(REP_MAPP_NAME);
  tell(REP_MAPP_NAME,"\n\nIn Malicious task application");
  //This task will try access the decrypted contract from trusted
      domain space with DMA
  char stealContract[16];
  char putStuffHere[11];
  int a;
    for(a=0; a<15; a++)
      stealContract[a] = a*3+30;
    stealContract[15] = '\0';

//  unsigned char *stealSignature;
//  stealSignature = (char*)0x42fad0; //this changes sometimes, point
    to address of signature
//  tell(REP_MAPP_NAME,"We have stolen the signature from other task
    !");
//  pr_hex(stealSignature,128,0);

  /*We assume that the malicious user got hold of the address to the
      decrypted contract
   *,he try to use DMA to transfer this information from trusted
       domain into the local pointer stealContract**/
  tell(REP_MAPP_NAME,"Try to use DMA to steal decrypted contract from
      trusted domain");
  dmaBurst(0,(void*)0x326a2c,putStuffHere , 10);
  //This generated a Illegal access try from the DMA abort handler
```

```
  //now he tries to use a pointer to point to the address of the
      decrypted contract
  //This will generate a Page domain fault when he is trying to access
       the trusted domain
  tell(REP_MAPP_NAME,"Try to access decrypted contract in trusted
      domain without DMA.");
  stealContract[0] = *((char*) (0x326a2c));

  tellf(REP_MAPP_NAME,"Buffer content: %s",stealContract);
  tell(REP_MAPP_NAME,"Attack not successful. Giving up.");
  bye_for_good(REP_MAPP_NAME);
  exit(0);
}

////////////////// DMA Interrupt ////////////////////////////
void dmaAppHandler(uint32_t channel)
{
    // report
    hello(REP_TRUSTED_NAME);
    LOGT("\nDMA handler of task mode called.\n");
    LOGT("(but nothing to be done)\n");
    bye(REP_TRUSTED_NAME);

    // back to hypervisor
  ISSUE_HYPERCALL(HYPERCALL_END_DMA);
}
```

Bibliography

[1] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. *SIGPLAN Not.*, 41:2–13, October 2006.

[2] AMD. Amd virtualization technology. http://sites.amd.com/us/business/it-solutions/virtualization/Pages/amd-v.aspx.

[3] ARM The architecture for the Digital World. Arm compiler toolchain. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0491c/BCFGCHBD.html.

[4] ARM The architecture for the Digital World. Arm processor instruction set architecture. http://www.arm.com/products/processors/technologies/instruction-set-architectures.php.

[5] ARM The architecture for the Digital World. Arm926 processor specifications. http://www.arm.com/products/processors/classic/arm9/arm926.php.

[6] ARM. Arm architecture reference manual. http://www.scss.tcd.ie/~waldroj/3d1/arm_arm.pdf.

[7] ARM. Virtualization is coming to a platform near you. http://www.arm.com/files/pdf/System-MMU-Whitepaper-v8.0.pdf, 2010-2011.

[8] CSO. Mobile malware: What happens next. http://www.cso.com.au/article/267157/mobile_malware_what_happens_next?pp=1, 2008.

[9] Heradon Douglas. Thin hypervisor-based security architectures for embedded platforms. Master's thesis, The Royal Institute of Technology KTH, Stockholm, Sweden, 2010.

[10] Michael J Fromberger. Cryptography technology c implementations. http://spinning-yarns.org/michael/mpi/.

[11] Brian Gladman. Cryptography technology c implementations. http://gladman.plushost.co.uk/oldsite/cryptography_technology/index.php.

[12] Robert P. Goldberg. Survey of Virtual Machine Research. *Computer*, pages 34–45, 1974.

[13] D. Gollman. *Computer Security.* John Wiley and Sons LTD, second edition, 2005. Chapter 11.

[14] Joo-Young Hwang, Sang-Bum Suh, Sung-Kwan Heo, Chan-Ju Park, Jae-Min Ryu, Seong-Yeol Park, and Chul-Ryun Kim. Xen on arm: System virtualization using xen hypervisor for arm-based secure mobile phones. *Communications Society*, pages 257–261, 2008.

[15] Intel. Intel virtualization technology. `http://www.intel.com/technology/virtualization/technology.htm?iid=tech_vt+tech`.

[16] Robert Kaiser. Applicability of virtualization to embedded systems. In Massimo Conti, Simone Orcioni, Natividad Martinez Martinez Madrid, and Ralf E.D. E. D. Seepold, editors, *Solutions on Embedded Systems*, volume 81 of *Lecture Notes in Electrical Engineering*, pages 215–226. Springer Netherlands, 2011.

[17] Rafal Kolanski and Gerwin Klein. Formalising the l4 microkernel api. In *Proceedings of the 12th Computing: The Australasian Theroy Symposium - Volume 51*, CATS '06, pages 53–68, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[18] Open Kernel Labs. Okl4 microvisor. `http://www.ok-labs.com/products/okl4-microvisor`.

[19] Jochen Liedtke. L4 reference manual - 486, pentium, pentium pro, 1996.

[20] George Marsaglia. Random number generator for c. `http://www.cse.yorku.ca/~oz/marsaglia-rng.html`.

[21] Communications of the ACM. Interview with steve furber. `http://cacm.acm.org/magazines/2011/5/107684-an-interview-with-steve-furber/fulltext`, 2009.

[22] Open Virtual Platforms. `http://www.ovpworld.org`.

[23] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17:412–421, July 1974.

[24] Oliver Schwarz. Dma virtualization.

[25] A.N. Sloss, D. Symes, and C. Wright. *ARM system developer's guide: designing and optimizing system software.* The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier/ Morgan Kaufman, 2004.

[26] ARM Security Technology. Building a secure system using trustzone technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2005-2009.

[27] VMware. Understanding full virtualization, paravirtualization and hardware assist. http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf.

[28] VMware. Vmware mvp mobile virtualization platform. http://www.vmware.com/products/mobile/overview.html.

[29] VMWare. The benefits of virtualization. http://www.nasba.com/server/assets/VMW_09Q3_WP_SMB_IT_EN_P6_R1.pdf, 2009.

[30] Wookey, Chris Rutter, Jeff Sutherland, and Paul Webb. The gnu toolchain for arm targets howto. http://www.aleph1.co.uk/oldsite/armlinux/docs/toolchain/toolchHOWTO.pdf.