ABSTRACT

SHAH, JITESH HARSHAD. ARMithril: A Secure OS Leveraging ARM’s TrustZone Technology. (Under the direction of Dr. Peng Ning.)

The ARM architecture has gained significant popularity in the embedded systems market because of its power-efficient design. ARM cores are dominant in mobile phones, smart phones, tablets, electric meters, routers and other embedded devices. Unfortunately, the popularity of the ARM architecture has made it an attractive target for malicious attacks. In view of the rising security threats, ARM came up with a set of hardware security extensions called "TrustZone" to provide hardware support for writing secure software. This work aims to build a minimalistic secure kernel on top of the TrustZone technology to provide an isolated trusted execution environment for security-sensitive tasks. The contribution of this work is a lightweight secure kernel with support for the notion of tasks, support for synchronous and asynchronous processing of tasks, secure memory, I/O device partitioning and configurable periodic/event-based execution of the secure kernel. Due to the small size of the kernel, it should be easy to analyze its security properties (this work doesn’t aim to do it). Also, provided is an API, in the form of a link-able Linux library, for the normal world to interact with the secure world, conforming to the TrustZone API 3.0 specification released by ARM.
ARMithril: A Secure OS Leveraging ARM's TrustZone Technology

by

Jitesh Harshad Shah

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the degree of
Master of Science

Computer Science

Raleigh, North Carolina

2012

APPROVED BY:

______________  ______________
Dr. Xuxian Jiang    Dr. Ting Yu

______________
Dr. Peng Ning
Chair of Advisory Committee
DEDICATION

To my parents and my brother.
Jitesh Shah hails from Pune, Maharashtra, India. He received the Bachelor of Engineering degree (specialization: Computer Engineering) from Pune Institute of Computer Technology, Pune, India in 2008. He joined Marvell Semiconductors India Pvt. Ltd. as a software engineer in the Mobile R&D BU soon after. He enrolled for the Masters in Computer Science program at NCSU in Fall 2010. He worked in the Cyber Defense Laboratory under the guidance of Dr. Peng Ning from Spring 2011 onwards. He intends to join Samsung Telecommunications America, San Jose after graduation.
Acknowledgements

First and foremost, I would like to thank my advisor, Dr. Peng Ning for showing continued faith in me. This project wouldn’t have been possible without his vast knowledge and experience, insightful advice, timely criticism and support.

I would also like to thank Dr. Xuxian Jiang and Dr. Ting Yu for agreeing to serve on my thesis committee and using their expertise to give me valuable feedback on my work.

I would also like to thank Dr. Ahmed Azab for helping me out of tough spots at critical times and for his valuable suggestions at various points during the project. I would like to thank my colleagues at Cyber Defense Laboratory for their inputs as well.

I would like to thank the Freescale engineers for answering my slew of questions about how-stuff-works.

I would like to thank my very good friends: Prathamesh Tade, Mayur Golatgaonkar and Salil Kanitkar for helping me out through tough times and keeping me motivated.

Lastly I would like to thank my parents, Harshad & Archana Shah, and my brother, Saurabh Shah, for their continued unconditional love and support. Without them I wouldn’t be here.
# TABLE OF CONTENTS

LIST OF TABLES .......................................................... vii
LIST OF FIGURES ........................................................... viii

Chapter 1: Introduction ..................................................... 1
1.1 Related Work ......................................................... 3
1.2 Introduction to ARM and TrustZone .............................. 5
1.3 Under the Hood: TrustZone ......................................... 8
1.3.1 Protecting the Memory Subsystem ......................... 8
1.3.2 Protecting Peripherals ......................................... 11
1.3.3 Interrupts .......................................................... 11
1.3.4 Debugging Interfaces ........................................... 12

Chapter 2: Design .......................................................... 14
2.1 Objective ............................................................... 14
2.2 Boot-up ................................................................. 14
2.3 Switching Between the Secure and the Non-secure World .... 18
2.4 Secure Kernel: Tasks ................................................ 20
2.5 Inter-world Communication ...................................... 21
2.6 Interrupt Latency ..................................................... 22
2.7 Putting It All Together .............................................. 23

Chapter 3: Implementation ................................................. 25
3.1 Monitor-mode: Switching Between Worlds ...................... 25
3.1.1 SRTC Configuration ........................................... 25
3.1.2 Monitor Mode Code ............................................. 26
3.2 Secure Kernel ......................................................... 29
3.2.1 Writing New Services for the Secure World ............ 30
3.2.2 Tasks ............................................................... 30
3.2.3 Notification Mechanism ....................................... 31
3.2.4 Non-secure World Library .................................. 32

Chapter 4: Security Analysis .............................................. 34
Chapter 5: Experimental Evaluation ............................................. 36
5.1 Experiment Setup ............................................................... 36
5.2 Evaluation Methodology ......................................................... 37
5.3 Results ................................................................................... 39
  5.3.1 Application Overhead ..................................................... 39
  5.3.2 Memory Protection Overhead ............................................ 41
  5.3.3 Switching Time ............................................................... 41

Chapter 6: Conclusion ............................................................... 44

References .................................................................................. 45

Appendices .................................................................................. 50
Appendix A. World Switch Code .................................................. 51
Appendix B. Skeleton of TZ Application Code ................................ 54
LIST OF TABLES

Table 5.1: Application Overhead ................................................. 40
Table 5.2: Memory Protection Overhead ......................................... 42
Table 5.3: World Switch Time ..................................................... 42
LIST OF FIGURES

Figure 1.1: ARMv7 register banking ........................................ 6
Figure 1.2: ARMv7 CPU modes with worlds ................................. 7
Figure 1.3: ARMv7 MMU with Security Extensions ....................... 10
Figure 2.1: Secure Kernel Image File Format ............................. 16
Figure 3.1: Architecture ..................................................... 33
Chapter 1

Introduction

According to ARM’s company profile page [1], ARM has shipped more than 20 billion cores till date (July, 2012). The same page puts the ARM’s market share in the smartphone market at 90% and the digital camera market at 80%. Clearly, ARM is a dominant player in the low-power embedded processor market.

Because of the rising popularity of smartphones, attacks of smartphones have been widely studied in academia. Thus, it would be worthwhile to take a look at the security exploits in a popular smartphone OS like Android running on ARM and motivate this work from there. However it must be noted that the attacks are not limited to just smartphones. As an example, a list of attacks against stock routers are listed in [17].

A comprehensive study of Android Malware can be found in [32]. The paper brings forward interesting observations. First, it considers a total of 1260 Android Malware divided in 49 families. One-third of these use root-kits to gain root permissions whereas around half of the studied malware harvest user information. [14, 23] further illustrate the prevalence of malicious attacks on popular embedded devices.

There have been various approaches to securing such devices, such as the software-only approach of TaintDroid [21]. However, ARM is approaching the problem from the other end: security from ground up. ARM developed a set of hardware security extensions called ”TrustZone”. Effectively, TrustZone enables an ARM core to provide an abstraction of two virtual cores (VCPUs). A secure VCPU and a non-secure VCPU.
It adds various hardware mechanisms to ensure that the non-secure VCPU cannot read or write secure VCPU’s resources. The reverse however isn’t true i.e. the secure VCPU has full access to the non-secure VCPU resources.

This opens doors to implementation of different security mechanisms. As an example, consider the DRM (Digital Rights Management) system and how it can adopt a TrustZone based approach. In a normal system, a media server sends along an encrypted blob of the media to a device that asks for the said service. The DRM software on the device decrypts the blob and writes the media output to either the screen and/or the audio output device. There are a few points where an attack can be mounted. Firstly, there will be a window where the decrypted blob is in memory before it is sent to the screen. Secondly, an attacker with devices like JTAG debugger can inspect any memory location, reverse engineer the DRM software and extract the key used for decryption or modify memory dynamically to serve malicious purposes like ask the media server to use null encryption.

If the DRM software were implemented as a trusted service in the secure OS, better security can be provided. The non-secure OS fetches the encrypted blob from the media server and passes it on to the trusted DRM service inside the secure OS. The secure OS owns the output devices while it is in operation. Also, secure memory cannot be analyzed or dumped even using tools like JTAG. Thus, it would be very hard for an attacker having even root privileges in the non-secure world to steal secrets belonging to the secure world.

Apart from DRM, other applications like a virus scanner or a remote attestation tool can easily run inside the secure OS with tighter security guarantees. Of course, the applications aren’t limited to the ones listed above. They just help to exemplify the
usefulness of the approach.

Using the hardware guarantees provided by TrustZone, this paper delves into the implementation of a secure OS which runs on the secure VCPU. The following subsection delves into some approaches that try to provide security on embedded devices. Subsection 1.2 provides the background on how TrustZone works. Section 2 sketches out the design and attempts to justify the design decisions made while designing the secure kernel. Section 3 goes into the implementation aspects of ARMithril. Sections 4 and 5 discuss the threat model and performance overhead respectively. Section 6 concludes the dissertation.

1.1 Related Work

Following ARM’s stellar growth, there have been several attempts at providing security features in embedded devices (with hardware support). This work can be broadly classified in three categories: Providing secure boot, providing a trusted execution environment and virtualization approaches. Note that these approaches are not entirely exclusive of one another. One can easily conceive a system employing, for instance, both secure boot and virtualization.

Following on the footsteps on TPMv1.2, the TCG (Trusted Computing Group) came up with the MTM (Mobile Trusted Module) [20] to provide secure boot on embedded devices. A notable improvement over TPM was that MTM was designed so that it can be instantiated multiple times. Thus, several modules can use MTM at the same time, instantiating their own MTM. Significant amount of research has been done on MTM [20, 31, 18, 15].
A TEE (Trusted Execution Environment) provides a sandbox with guarantees for isolated execution and secure storage. There are currently two popular ways of providing a TEE: Secure Element and ARM TrustZone. A Secure Element is a separate hardware entity, either built on-board or attached as a removable device. Since, the chip is physically separate, tampering with it is difficult. Google Wallet [5] uses Secure Element to provide a TEE. ARM TrustZone is another way of providing a TEE. TrustZone vouches for built-in security i.e. it provides a way to run a trusted OS in parallel and in isolation with another untrusted OS. Although ARM lays down the TrustZone architecture, it largely depends on the vendors to provide the interfaces to secure memory and peripherals. Thus, most of the technology behind TrustZone is proprietary. There has been some work using TrustZone in academia [29, 27, 30, 19]. ARM has partnered with GlobalPlatform to design a systematic way of developing secure systems, called the TEE client API [8]. The TrustZone API is a subset of the TEE API in that, the TrustZone API merely defines how to invoke services of the secure world. It doesn’t define how services may be added to the secure world or how these service invoke other security features (like secure storage or MTM). One proprietary TrustZone software is developed by Trusted Logic in partnership with ARM. It is called the Trusted Foundations [9]. There are also at least two mature opensource projects providing a TrustZone-based secure kernel: The OpenVirtualization Project [6] and safeg [7].

Lastly, virtualization is another way of providing isolated execution environments. KVM on ARM [16] and Xen on ARM [12] aren’t still being used in any commodity products we know of. An interesting application of virtualization is the CELLS [13] architecture where the kernel interface is virtualized to run multiple (more than 2) android instances at the same time without much overhead.
A detailed survey of available trustworthy execution methods for embedded systems is provided in [28]. Interested readers should read this paper for an excellent in-depth survey and categorization of security models currently available for ARM processors.

1.2 Introduction to ARM and TrustZone

This section provides a brief introduction to the ARM architecture and a more detailed description of the TrustZone technology.

ARM (Advanced RISC Machine), as the name suggests, is a RISC architecture. ARMv7 has 16 general purpose registers named r0-r15. r13 acts as the stack pointer. r14 is the link register (return address is stored here when a procedure call is made) and r15 is the program counter. Analogous to the status register in x86, ARM has CPSR (Current Program Status Register) which stores the condition flags and execution properties of the currently executing code (CPU mode, Thumb, Jazelle, etc). SPSR (Saved Program Status Register) is the CPSR of the previous mode. Analogous to the concept of security rings in the x86 architecture, ARM implements privilege levels. PL0 is the least privilege level (user) and PL1 is where the kernel runs. The hypervisor, if supported, runs with the highest privilege, PL2.

When the security extensions (TrustZone) aren’t implemented, the CPU can be in one of the following seven modes: User, FIQ (Fast Interrupts), IRQ, Supervisor, Abort, Undefined or System. Except User, all modes are privileged (PL1 mode). Some registers are banked between modes. Figure 1.1 from ARMv7 Architecture Reference Manual [24] shows which registers are banked in which modes. (Note: Monitor mode is introduced by TrustZone)
With the introduction of Security Extensions (TrustZone), from the system programmer’s view, there are two major changes: Introduction of the NS bit (Non-secure) in the SCR (System Control Register) and introduction of a new CPU mode called "Monitor mode".

NS-bit in the SCR controls whether the secure OS or the non-secure OS is executing. The monitor mode is used to switch between secure and non-secure worlds. There are two ways to enter the monitor mode. First is a call to the SMC instruction. SMC can only be called from PL1 or higher. It raises an exception that traps into the monitor...
mode. Secondly, FIQ or IRQ when configured to trap into the monitor mode. Figure 1.2 shows that the value of the NS-bit does not matter in the monitor mode. Thus, it can be used to switch between the secure and non-secure OS.

Figure 1.2: The various CPU modes proposed by ARMv7 architecture and the worlds they exist in.
1.3 Under the Hood: TrustZone

This section goes into the fundamentals of how TrustZone works under the hood. This will create the necessary basis to justify why the proposed design is secure. Most facts from this section have been collected from the TrustZone Whitepaper [25] and the Architecture Reference Manual [24].

The TrustZone Whitepaper differentiates between three kinds of attacks: Hack attacks, shack attacks and lab attacks. Hack attacks are purely software. Shack attacks use cheap commodity hardware like JTAG. Lab attacks use sophisticated instruments like electron microscope. The aim of TrustZone is to avoid hack and shack attacks.

The heart of the TrustZone implementation is an addition of a line on the ARM system bus (AMBA3 AXI). This line - AWPROT/ARPROT - is set high during a non-secure transaction and pulled low during a secure transaction. All non-secure bus masters pull this line high, thus marking the transaction as non-secure. Slaves must inspect this line before an access is allowed. Thus, the ARM core, one of the bus masters, pulls the line high when NS bit is set to 1 and pulls it low when NS bit is 0. Thus, a non-secure master can never access a secure slave unless configured to. With this simple addition, following subsections delve into how protection is insured for the various subsystems like memory, peripherals and interrupts.

1.3.1 Protecting the Memory Subsystem

Memory protection is best looked at in three different parts. First, protecting the DRAM chip itself. Even though DRAM is protected, a clever attacker might use cache/TLB to his advantage to bypass the protection. Hence, the second part is protecting the MMU
subsystem. Memory may still be accessed directly via DMA. So, the third part is protecting the DMA controller.

**Protecting the physical memory**

The TrustZone implementation mandates the presence of an address space controller to partition dynamic memory (in most cases, DRAM) into secure and non-secure physical areas. In the ARM’s Versatile Series of boards [2], this job is done by the TZASC (TrustZone Address Space Controller). FreeScale’s i.MX53 [3] series uses EXTMC (External Memory Controller) for the purpose. This controller can be configured to store memory regions as secure. Each memory access must go through this controller. The controller verifies that no non-secure bus transaction tries to access a secure memory area. A security violation is raised otherwise. Thus, even if a non-secure OS maliciously maps a secure page into its address space, the TZASC will deny access to the page. ARM’s Versatile Series has another controller called TZMA (TrustZone Memory Adapter) used to partition static memory (SRAM and ROM) into secure and non-secure spaces. TCM (Tightly Coupled memories), if available can also be partitioned. i.MX53 doesn’t have TCMs. Both TASC and TZMA can only be configured from the secure world.

**Protecting the MMU**

Second part is the protection of the MMU subsystem. The TTBR (Translation Table Base Register), which holds the base address of the page tables, is banked between the secure and the non-secure worlds. The secure world sets NS bit to 1 in its page tables to access non-secure memory. The non-secure world cannot access the secure memory, hence, the NS bit is ignored. Caches and TLBs are tagged with the NS bit too. Thus,
it is equivalent to having a 33-bit address space: a normal 32-bit address space plus the NS-bit. Thus, both secure and non-secure world words can coexist in the cache.

Figure 1.3: MMU translations in the presence of TrustZone security extensions. NS stands for non-secure bit. TID stands for Thread identifier. Figure from TrustZone Whitepaper [25]

Figure 1.3 shows the translation process diagrammatically. Figure is from the TrustZone Whitepaper [25].

Protecting DMA

Some I/O devices can directly access the memory via DMA (Direct Memory Access). A TrustZone aware DMA controller in tandem with the TrustZone aware memory controller can restrict such accesses to the proper world. FreeScale has SDMA (Smart DMA Controller) for the purpose.
1.3.2 Protecting Peripherals

Peripherals are connected to the system via the APB (Advanced Peripheral Bus). The APB is connected to the system bus (AMBA3 AXI) via the AXI-to-APB bridge. Adding NS-bit to the peripherals is trickier since it will require all external peripherals to support and honour the NS-bit. Instead, ARM puts the onus of enforcing peripheral security on the AXI-to-APB bridge. Like the memory controller, the bridge is configured with the peripheral security status. The bridge checks the NS bit of the bus transaction against the configured-in security and allows access to the peripheral only if the NS-bit is properly set. Needless to say, AXI-to-APB bridge can only be configured from the secure world.

1.3.3 Interrupts

ARM provides two kind of interrupts: IRQs and FIQs. FIQ mode has registers r8 to r15 banked so as to reduce the amount of save-restore required and improve the interrupt response time. Interrupt management requires an interrupt controller that supports TrustZone. FreeScale’s i.MX53 comes with a TZIC (TrustZone Interrupt Controller). Since, interrupt behaviour changes with each controller and hence, each vendor, this description is assumed to be valid just for the i.MX53. The generic pieces, valid for all interrupt controllers, will be pointed out specifically to be ARM mandated.

i.MX53’s TZIC supports 128 interrupt lines. Each line can be separately marked as secure or non-secure. Once marked secure, all configuration related to that line can be done only from the secure world (priority, enable-disable, masking, etc). Secure interrupts have a higher priority over non-secure interrupts by default. However, the secure world can configure a non-secure interrupt with a priority in the secure world priority
Interrupts marked as non-secure (by the secure world) can be independently configured by the non-secure world. In case of TZIC, non-secure interrupts are IRQs and secure interrupts as FIQs. When an interrupt is received, the TZIC interrupts the currently running world with the IRQ/FIQ. This is secure since the non-secure world cannot acknowledge the secure interrupts whereas the secure world can choose to handle or forward the interrupt appropriately.

ARM has mandated individual flags in the SCR (System Control Register) via which IRQ or FIQs can be trapped to the monitor mode. When, say, the flag for FIQ is set, on receipt of an FIQ, the core traps to the monitor mode. The monitor mode can then decide where to forward the interrupt or whether to handle it in the monitor mode itself. Although this adds a small latency equal to a world switch to the interrupt response, the method provides a way to directly trap into the secure world to handle interrupts that need immediate attention.

In summary, the TrustZone aware interrupt controller honours the NS-bit and allows only the secure world to configure secure interrupts, thus, keeping both worlds separate.

### 1.3.4 Debugging Interfaces

Although the above mechanisms completely separate the secure and the non-secure world, they avoid only hack attacks. An attacker with JTAG can still read from the secure world memory and steal its secrets. To avoid such shack attacks, ARM has provided two signals to the core that can only be programmed when forging a chip - SPIDEN (Secure Privileged Invasive Debug Enable) and SPNIDEN (Secure Privileged Non-Invasive Debug Enable). When both these signals are pulled low, even debuggers like JTAG cannot access the secure transactions (memory as well as bus) i.e. the debugging circuits on the
board will honour the NS-bit. To allow ease of development, these signals can be pulled high in the development spins of the hardware, thus, allowing JTAG to see the secure transactions too.

In summary, in the above subsections we saw how TrustZone guarantees isolation between the secure and the non-secure world OS. In better words, the secure world can access the non-secure world resources, but the reverse is not true. This lighter form of isolation can be used to implement various security schemes in the secure world (discussed in section 1).
Chapter 2

Design

The aim of this section is to give a birds-eye view of how two Operating Systems co-exist on a single ARM core and a peek into the design decisions involved.

2.1 Objective

The non-secure world operating system is the full-size operating system that will handle the typical user interactions. The secure world operating system is designed to be as minimalistic as possible so that its security properties can be easily verified. The current implementation of the secure OS has around 550 Source Lines of Code.

Another objective behind the design of the secure OS is to make it easy to extend the secure OS by plugging-in custom services. Also, since these services might take long to execute (for example, virus scanning), another objective is to enable asynchronous invocation of the services.

With these high-level objective in mind, let us start diving into the design of the secure OS.

2.2 Boot-up

Whenever a TrustZone-enabled system is powered up, it always starts in the secure mode. That is to say when the ROM transfers control to the bootloader, the bootloader
is executing in the secure world. In order to provide any guarantees of security, any TrustZone-enabled system MUST implement secure boot, at the minimum (Secure boot prevents any unauthorized code from executing. Trusted boot is a stronger requirement where an attestation that only trusted components were loaded needs to be provided). Technologies like TPM (Trusted Platform Module) [10] and MTM (Mobile Trusted Module) [20] are standardized ways of implementing secure boot.

In secure boot schemes, each device comes with its own private key embedded in a safe location in the SoC. This key cannot be read out of the device. Each software component then uses a tree hierarchy rooted at the above private key to verify and load the next component. For example, the ROM verifies the bootloader binary, the bootloader verified the kernel binary and so on. The property of secure boot is that at any stage in the boot process, one can guarantee that all software components loaded since boot till that stage are trusted. Implementing secure boot, thus, guarantees that the software that initially runs in the secure world hasn’t been maliciously modified offline. The secure world kernel can then verify the non-secure world kernel and transfer control over to it.

Note that although there are standards like MTM in place to provide secure boot, as of today, each ARM core manufacturer implements their own versions of secure boot. Freescale provides something called HAB (High Assurance Boot) in its i.MX53 series. A brief description of HAB can be found in the i.MX53 processor reference manual [22] - Chapter 7 (System Boot). Since, each vendor provide their own way of implementing secure boot, we won’t discuss secure boot in this work. It is assumed that anyone who uses TrustZone MUST implement secure boot for the security guarantees to be applicable.
Secure Kernel Image File Format

An ARM bootloader passes boot-time parameters to the kernel via an ATAGS blob placed at a predetermined offset in memory. This ATAGS blob has configuration parameters like board ID, revision ID, physical memory chips and addresses and the kernel commandline. Since, a TZ system has two kernels (secure and non-secure), there needs to be two ATAGS blobs. One for each kernel. The bootloader (in our case uboot), provides ATAGS for the secure kernel. Thus, the secure kernel image should have the following three parts at the minimum: The secure kernel image, the non-secure ATAGS blob and the non-secure kernel image. The file format designed to hold this is straight-forward and shown in figure 2.1.

![Figure 2.1: The file format used for a secure kernel image. Note that this image format is designed to work only with the uboot bootloader.](image)

The uboot header is 40 bytes in size and contains the size of the secure kernel image. Each blob is then preceded by its size. The locations to drop the blobs in physical memory are hardwired in the secure kernel. Uboot loads this big image and transfers control to the secure kernel. The secure kernel image knows the offsets for each blob and copies
the blobs to appropriate locations.

Overview of Secure Kernel Boot
As mentioned earlier, when the secure kernel gets control, all resources in the system are marked as secure and hence, unusable by the non-secure world. The job of the secure world is to then selectively allocate resources the non-secure world needs. The early boot of the secure world is best illustrated by a high-level algorithm.

1. Turn off interrupts, turn off caches, invalidate TLBs, enter SVC mode. Initialize stack for the SVC mode.

2. Read secure world ATAGS from a pre-determined location and read in the size and location of the physical memory.

3. Copy the non-secure world ATAGS blob and the non-secure world kernel at appropriate locations in the physical memory.

4. Mark the secure world memory as protected in the Address Space Controller (TZASC for ARM boards and EXTMC for FreeScale series).

5. Initialize the I/O devices the secure world is going to use. Mark the secure world I/O devices are protected by configuring the AXI-to-APB bridge.

6. Configure the TrustZone enabled Interrupt Controller to mark the interrupts needed by the non-secure world as non-secure. Configure priorities of interrupts. Register the interrupt vector table for the secure world. Initialize stacks for the IRQ and FIQ mode. Interrupts can now be enabled.
7. Initialize the monitor mode code (used to switch between secure and non-secure world). Initialize the monitor mode vector table. Initialize stack for the monitor mode.

8. Complete secure world initialization. Start tasks on the secure world. Optionally, enable secure timer to periodically switch between the secure and the non-secure world (more details in a later section).

9. Call SMC instruction to transfer control to the non-secure world. Non-secure world boots from here.

2.3 Switching Between the Secure and the Non-secure World

The objective of this section is to discuss the various strategies to switch between the worlds. The discussion of the actual implementation can be found in section 3 (Implementation).

After following the algorithm outlined above, we have both the secure world and non-secure world running in parallel. But, there is an important piece of the puzzle missing: what triggers the switch between the worlds? There are two types of triggers possible - periodic switch based on a timer and event-based switch. Both have their own advantages, disadvantages and applications.

In the periodic switch model, there is assumed to be a timer that can only be programmed by the secure world. If that is not true, the non-secure world can merely disable the timer and prevent the secure world from running. Such an assumption is not over-
burdening since most vendors do provide a secure timer as part of a TrustZone enabled system. FreeScale provides SRTC (Secure Real Time Clock) in its i.MX53 series. The power source of this clock also needs to be secured. Once such a timer exists, the interrupt line corresponding to it can be marked as secure and the interrupt can be directly made to trap into the monitor mode. Periodically switched systems are effective when the secure world runs long-running asynchronous tasks. Also, since the timer is secure, guarantees can be made about progress of tasks in the secure world. Typical examples include virus scanners and remote attestation utilities. Disadvantage of such an approach is that for systems that need to conserve battery, periodic wake-ups due to secure timer interrupts will cause loss of battery.

For systems that cannot bear the extra burden on the battery, event-based switching is an option. In this model, the non-secure world invokes the services of the secure world when certain conditions are met. Such calls are usually synchronous. A typical example would be a DRM application. The secure world will be invoked synchronously only when media needs to be consumed. The primary disadvantage of the event-based approach is that it is possible to implement a DoS attack against the secure service. But, for some applications like the DRM application listed above, such attacks are a low priority threat.

One thing to note is that it is not necessary to choose between both these models. They can coexist in the same system. The discussion henceforth assumes that the system is hybrid i.e. both periodic as well as event based switching models are implemented at the same time.
2.4 Secure Kernel: Tasks

Now that we have figured out how to switch between the worlds, let’s delve a little deeper into what the secure kernel does after boot-up.

In a purely event-based model, the implementation of the secure kernel is pretty uninteresting. One can imagine a big switch case where the variable is the service requested by the non-secure world. Once the service execution is complete, the execution returns back to the non-secure world.

The more interesting case is when a periodic model is implemented. The concept of time slices goes hand-in-hand with any periodic switching model. Depending on the requirement of each world, time slices can be allocated to each world. In the current implementation, a 1:3 (secure:non-secure) time slice is the default. It can be easily changed from a configuration file. Since a service might not run to execution in a single secure world timeslice, the secure world needs to have a notion of a task which can be suspended and resumed. Since, the calls are asynchronous, the non-secure world might queue more tasks before the previous tasks have completed to execution. Thus, the secure world kernel needs to implement, at the very minimum, a queue of tasks and a scheduling strategy. There needs to be a way to asynchronously notify the non-secure world when a task completes (and store the result till the non-secure world collects it). If there is no task in the queue waiting to be executed, the secure world should prematurely end its time-slice and yield to the non-secure world. Our implementation supports all these operations. With the interest of demonstrating the utility of the approach, a simple scheduling strategy namely, first come first serve was chosen. A more sophisticated approach might be appropriate for a production system. The notification system hijacks
an unused interrupt line from the TZIC to cause an interrupt into the non-secure world.

2.5 Inter-world Communication

If the non-secure world can invoke tasks, there might be services where a blob of data needs to be passed between both the worlds. The most efficient way of communicating data between both the worlds is shared memory. Since the non-secure world cannot access the secure world memory, the non-secure world should provide the buffers.

The problem with shared memory is that, with the introduction of the NS-bit the address space effectively is NS-bit plus the normal 32-bit address space. In other words, if the same page is mapped in both the secure and non-secure worlds, memory aliasing will occur. To avoid memory aliasing, each world must agree on a synchronization primitive. The following protocol is an example:

1. Before en-queuing a task, the non-secure world allocates a physically contiguous (merely for simplicity) region of memory and writes the data there. The d-cache for that region is flushed and invalidated. No more read or write accesses are done on this region (to avoid the data being re-cached). The physical location of this buffer is passed to the secure world. A task request is enqueued.

2. If the secure world implements an MMU, it maps these physical pages in its own address space. If it doesn’t, these physical addresses will be directly usable. While the secure world is accessing this memory, the non-secure world doesn’t touch these locations. When the task is complete, the secure world flushes and invalidates its d-cache for the accessed range. The non-secure world is then notified that the task is complete.
3. The non-secure world accesses the buffers normally. Since, these aren’t cached, the updated values written by the secure world are accessed.

Note that the onus of allocating physically contiguous pages and flushing the caches is on the API library in the non-secure world. The applications do not have to worry about it.

2.6 Interrupt Latency

Now that we have the secure and the non-secure world time multiplexed over the single ARM core, an important question comes into picture: What if we get an interrupt for the non-secure world but we are executing in the secure world (or vice-versa) i.e. what if we get an interrupt and we are in the wrong world? An important thing to note here is that, unless configured to trap into the monitor mode, an interrupt traps into the currently executing world (irrespective of its security setting). However, since the interrupt controllers are vendor specific, it can be argued that the interrupt behaviour is different for different vendors.

A straightforward answer might be to switch to the correct world as soon as the interrupt is received. However, this methodology can then be used to mount a DoS attack on the secure world (if a non-secure interrupt is asserted when the secure world is entered, it will immediately switch back without the secure world getting an opportunity to execute). With this straightforward approach, the interrupt latency will be increased by a maximum time of two world switches (If the interrupt is asserted just as the execution traps into the monitor mode from the non-secure world, then the interrupt will be seen only when the monitor code exits into the secure world. Another world switch will then
be needed to enter the non-secure world). Thus, keeping the world switch time low can help reduce the maximum interrupt latency.

To reduce the interrupt latency of secure world interrupts, they can be configured to trap into the monitor mode itself. With this configuration, there will be no extra latency to respond to secure world interrupts. Since secure world interrupt also have a higher priority than the non-secure world interrupts, it would make sense to implement handlers that need quick response as secure interrupts.

A strategy to deal with the DoS attack can be to gather time-slice credits each time the secure world yields to a non-secure world interrupt. If the credit value reaches above a threshold, some guaranteed progress has to be made (i.e. secure world executes with interrupts turned off) before the yielding to the non-secure world interrupt. This is a trade-off between guaranteeing progress of the secure world and reducing interrupt latency time.

## 2.7 Putting It All Together

In summary, secure boot should be implemented by any system using TrustZone. The minimalistic secure kernel initializes and starts the non-secure kernel. The switching between them is based on a hybrid model - periodic as well as event based. A secure timer is used for the periodic switch. The secure kernel supports the notion of tasks. Tasks can be kicked off by either world (non-secure world in the normal cases though). The worlds communicate using shared memory buffers provided by the non-secure world. Interrupt latency can increase since the execution might be in the wrong world when an interrupt for the other world is received. There is no extra overhead to respond to secure
interrupts configured to trap in the monitor mode.

The next section dives into the nitty-gritties of the implementation.
Chapter 3

Implementation

The implementation of the secure kernel is done on FreeScale’s i.MX53 Quick Start Board [4]. It features a TrustZone-enabled Cortex-A8 core clocked at 1GHz and 768MB of RAM. The implementation section is divided into three subsections - each for a major subsystem: The monitor mode switching code, the secure kernel and the non-secure world API to invoke services of the secure world.

3.1 Monitor-mode: Switching Between Worlds

Since, the SRTC (Secure Real Time Clock) configuration is essential to periodically switching worlds, lets start with looking at how that is done.

3.1.1 SRTC Configuration

The SRTC, by default, can be accessible only by the secure world. The bit that controls its access is NA (Non-secure Access) in the SRTC Control Register (SRTC_LPCR). At boot-up this bit is set to 0 which means the SRTC cannot be reprogrammed by the non-secure world.

The SRTC is driven by two clocks. The counters are driven by a low power timer clock (32Khz) which is in an always-on power domain. The status and control registers are driven by a high-speed peripheral clock. The security of the SRTC lies in the fact that
the low-power clock cannot be disabled except by shutting down the device. Combined with secure-only access to the SRTC, there is no way for the non-secure world to avoid executing the secure world. The only issue then is the high-power clock that runs the status and control registers. The non-secure world does have unsecurable controls to turn off the high-power clock. However, this only affects reading out values from the SRTC. The monitor mode code re-enables the clock each time it is executed.

The last steps in securing the SRTC interrupt are to mark the SRTC interrupt (line 24) as secure in the TZIC. Remember from an earlier section that, in FreeScale’s implementation, marking an interrupt as secure automatically makes it an FIQ. Thus, to trap to the monitor mode everytime SRTC interrupt is asserted, FIQs are configured to trap to the monitor mode. This can be done by setting bit number 2 in the SCR (System Control Register) [24]. SCR is co-processor number (c1, c1, 0, 0). Following snippet of ARM assembly shows how this can be achieved.

```c
#define SCR_FIQ_BIT (1 << 2)
mrc p15, 0, r1, c1, c1, 0
orr r1, r1, #SCR_FIQ_BIT
mcr p15, 0, r1, c1, c1, 0
```

"mrc" instruction is used to read from a co-processor register (SCR) into general purpose register r1. "mcr" is used to write back to SCR with FIQ bit set i.e. FIQs will trap to the monitor mode.

### 3.1.2 Monitor Mode Code

SRTC is now setup and secured. Lets assume it is configured to go off at 32Hz (can be easily changed from a configuration file). This sections goes into the details of the ac-
tual world switch code. Recall that there are two entry-points to the world-switch code: Periodic-timer interrupt and explicit call to the smc instruction. Both will eventually converge to a single point.

**Periodic Switch**

When a secure timer interrupt is triggered, the execution traps into the monitor mode. The source world can thus, be the secure or the non-secure world. Registers r0-r12 of the source world should be restored. "lr" register contains return address to the saved world. Looking at figure 1.1, we can see that the CPU modes banked versions of the registers are shared between secure and the non-secure world. Thus, the monitor mode should save and restore all the banked versions of the registers too (Some modes can be excused. For example, FIQ mode is exclusive to the secure world whereas IRQ mode is exclusive to the non-secure world. These don’t need to be saved/restored). The NS-bit has the value of the source world i.e. if the execution was trapped from the non-secure world, NS-bit is set. Otherwise, it is reset. This enables detection of the source and target worlds. With this entry conditions, lets see the algorithm for the periodic switch case:

1. Push general purpose registers r0-r12 on the stack. Read SCR (it has the NS-bit) and push it on the stack too.

2. Enter Secure mode i.e. set NS-bit to 0.

3. Recall the SRTC quirk discussed in the SRTC section above. The high-power peripheral clock needs to be turned-on, if turned off by the non-secure world. This enables reading of the SRTC status registers.
4. Read the SRTC ISR (Interrupt Status Register). Read status of the 32Hz interrupt and acknowledge the interrupt.

5. Time slice accounting. If time-slice hasn’t expired, mark the target world same as the source world and return immediately. If time-slice of the source world has expired, continue with the switch code. Do the time-slice credit adjustments.

6. switch_worlds: Pop saved SCR (and hence, NS-bit). Determine the source world and the target world.

7. Pop saved registers r0-r12 into memory that saves state of source world. Save other registers like lr, svc mode sp, svc mode lr, etc.

8. Restore state of the target world (All registers listed above).

9. Set the NS-bit to reflect the new target world. Perform a return.

Find the source code listing for the above algorithm in Appendix 6. Since the actual code is lengthy, the listing provides only the code for actual save-restore of registers i.e. the core world switch code (Everything from switch_worlds label above). For the whole code, refer to the source tarball (or contact the authors).

**Event-based Switch**

This branch is entered with a call to the "smc" instruction. The entry condition in this case is that registers r0-r3 are scratch and may or may not be saved/restored. Registers r4-r12 must be saved and restored. Also, as discussed above, banked copies of registers and "lr" must be saved and restored. Lets look at the algorithm:
1. Push general purpose registers r0-r12 on the stack. Read SCR and save it on the stack.

2. Based on the value of register r0 and the source world, decide the task to perform (might be NULL if just world switch is requested). Call the appropriate service. Return value is in r0.

3. Adjust time-slice credits. If no world switch is necessary, perform an immediate return. If a world switch is required, Jump to "switch_worlds" label in above algorithm.

### 3.2 Secure Kernel

The discussion of any TrustZone-based secure kernel should always start with how memory and I/O protection is implemented. However, we are bound by Non-disclosure agreements with FreeScale and hence, cannot reveal how the protections were put in place. Suffice to say, each vendor is mandated to have an address space controller that can mark physical memory regions are secure and the AXI-to-APB bridge controller that can mark peripherals as secure. Thus, this section will assume that all security protections are in place already and the boot of secure world is complete (discussed earlier). This section will go into details about how to write services for the secure kernel, how tasks are implemented and how the notification system works.
3.2.1 Writing New Services for the Secure World

One of the goals of this work is to enable easy addition of new services to the secure world. Thus, the core of the secure kernel provides only a mechanism to register services and to invoke them as tasks.

Each service has a unique ID (let’s call it UID). If one is writing an app in strict conformance with the TrustZone API Specification 3.0 [26], then one might obtain a UUID (Universally Unique Identifier) for their service. This UUID can be used as the UID in this secure kernel. With this UID, the service can be registered using the following API:

```c
int cdl_tz_register_service(unsigned long uid,
                           int (*proc)(void *));
```

Note that all services have to be statically compiled-in the secure kernel. Dynamic addition/removal of service is not supported yet. Thus, after adding a new C file containing the service definition, an entry into the Makefile is needed to enable the service.

3.2.2 Tasks

Each service registered above can now be invoked as a task from either world. From the secure world, it is merely a matter of calling `enqueue_task()` with the service UID. From the non-secure world, one needs to make an SMC call with `enqueue_tasks’ identifier in
register r0 to enqueue a task. The specification of enqueue_task is as follows:

```c
unsigned long enqueue_task(unsigned long uid, void *data_ptr);
```

Note that the enqueue_task function will not wait for the task to complete. The caller of the task can do something else while this task is being processed and collect results later i.e. an asynchronous model is implemented.

Once a task enters the queue and is in the ready state, the scheduler can schedule it at will. Since implementation of a complex scheduling strategy doesn’t add a functional value to the work, we have chosen to keep the scheduling strategy simple and merely for demonstration purposes. The scheduling uses the first come first serve strategy to execute tasks.

When a task finishes execution, the return value is collected and stored until the caller collects the result. A notification is sent to the caller about the completion of the task.

### 3.2.3 Notification Mechanism

Asynchronous notification can be achieved by raising an interrupt in the non-secure world. TZIC provides a way to raise software interrupts by writing to the TZIC_SWINT (TZIC Software Interrupt) register. Since TZIC doesn’t provide any user-programmable interrupt line, we’ve hijacked interrupt line 24 (meant for PATA. In our experience, most
embedded devices do not use PATA, but if one does, the interrupt line can very easily be changed to another unused one). Line 24 is marked as non-secure in TZIC.

Once the TZIC_SWINT is written to, the secure world fetches the next task. When the execution switches to the non-secure world, the interrupt handler is called. The interrupt handler then fetches the result (return value) from the secure world and passes it on to the caller.

### 3.2.4 Non-secure World Library

The non-secure world implements a dynamically loadable Linux library which provides the TrustZone API specification 3.0 [26]. The library is backed by a TrustZone driver (kernel module). This module is responsible for interacting with the secure world.

Appendix 6 shows a skeleton of code an application can use to write a TZ application and link with this library.

The TZ driver (kernel module) keeps a state machine of all tasks currently enqueued for processing. It keeps track of which process forked the app. It is this drivers’ responsibility to respond to task complete notifications and wake-up the corresponding process (if sleeping). In summary, the TZ driver interacts with the secure world to create tasks and collect results.

The overall architecture of the implementation is depicted if figure 3.1.
Figure 3.1: The high-level architecture of ARMithril.
Chapter 4

Security Analysis

Since this secure kernel builds on the ARM’s TrustZone technology, it inherits all properties already provided by TrustZone. Following paragraphs explain the threat model and how our solution addresses these attacks.

The TrustZone Whitepaper [25] differentiates between three kinds of attacks: A hack attack, a shack attack and a lab attack. A hack attack is a software only attack. A shack attack can use cheap commodity hardware like JTAG debuggers whereas a lab attack is highly sophisticated attack using devices like electron microscopes. TrustZone provides protections against the hack and shack attack.

Protection against hack attack implies that if the non-secure world is compromised, there should be no software only way to read/write sensitive data from/to the secure world or execute unauthorized code in the secure world. We claim that since hardware resources are divided among both the worlds and the division is enforced using TrustZone (TZASC, TZMA, AXI-to-APB bridge, etc) such an attack is not possible. The only way the non-secure world can influence the secure world is when they are communicating via shared memory during execution of a task. This greatly reduces the surface area for a hack attack. The onus of properly verifying the data shared by the non-secure world then falls on the services in the secure world. This code is usually small and should be thoroughly audited before putting in production.

Even if hack attacks could be avoided, today’s embedded devices come with debugging
interfaces like the JTAG interface. JTAG can halt the processor mid-execution, analyze the contents of memory or even modify the memory at runtime. These devices can then be used to dump the secure world memory and steal sensitive information or to influence the secure world execution maliciously. These class of attacks are known as shack attacks. TrustZone and hence, this secure kernel provides protection against shack attacks too. Deasserting SPIDEN and SPNIDEN will close the debug interfaces of the secure world. Implementing secure boot guarantees that malicious software is not executed in the secure world.

Protecting against hack and shack attacks gives sufficient protection against remote attackers, hobbyist device owners and software developers. By sufficient, we mean to imply a trade-off between the cost required to break the security mechanism and the economic gain from that. However, there is a class of users - who will use sophisticated instruments to mount a lab attack. This implementation doesn’t address this class of attacks.

In summary, this implementation of the secure kernel guarantees protection from hack attacks where the non-secure world is compromised and shack attacks. No sensitive information can be leaked out of the secure world or malicious code executed in the secure world if the attack falls in the above two categories.
Chapter 5

Experimental Evaluation

The next step in evaluating a security solution is to measure its performance overhead. A foolproof security solution which hinders real work is not practical. Since this secure OS runs time multiplexed with the normal OS on the same core, it is important to understand how much overhead it incurs. Following paragraphs introduce the experiment setup and the metrics used to measure this overhead.

5.1 Experiment Setup

As mentioned in the implementation section, this secure OS is implemented on a FreeScale’s i.MX53 Quick Start Board (1GHZ Cortex-A8 processor with 768MB RAM, independent 32b I-cache and D-cache and 256kb unified L2 cache). The non-secure world runs Linux kernel 2.6.38 provided by FreeScale (unmodified). The userspace runs an unmodified Ubuntu system. Android (Gingerbread) was also tested and works flawlessly. The benchmarking tool we chose however was hassle-free to run on the Ubuntu filesystem provided by FreeScale. In order to simulate a real system, the kernel and the filesystem are booting from the SD card and Ubuntu runs in runlevel 5 (i.e. with the X server and an attached monitor). The periodic timer is set to go off at 32Hz. The time slice ratio is 1:3 (secure:non-secure). The secure world runs with I-cache and D-cache on, but MMU turned off. With this setup in place, lets now look into the actual experiment parameters.
5.2 Evaluation Methodology

From an application programmers’ perspective, it is important to know how much the applications suffer because of the overhead introduced by the secure OS. The overhead might be different for CPU-bound applications and I/O bound applications. Thus, any metric to measure this overhead should measure both these values. Since the physical cache is now shared between both the worlds, there should be tests which stress the cache. Also, testing against real world workloads (like process creating, shell-script execution, web-page rendering, etc) is much more likely to give results relevant to the practical applications. Furthermore, using a standard benchmark is always preferable since they have been widely tested against many machines and more dependable. With these criteria in mind, we chose to use UnixBench [11] to measure the application performance. UnixBench has been around since 1983 and has seen contributions from lots of developers over time. UnixBench includes tests that individually stresses one subsystem of the other. The Dhrystone tests are designed to stress the CPU and cache. The Whetstone test focuses on floating-point operations. The file-copy tests focuses on I/O. Two more tests, Pipe-based context switching (parent and child process communicating via unnamed pipe) and Shell scripts execution test the overall system. The varied nature of these tests will give application programmers great insight into which subsystem suffers from the highest overhead.

UnixBench is a constant time benchmark. As in, it runs tests for a constant amount of time and reports the work done in that time as an index. Thus, a machine that can run more iterations in the same amount of time will have a higher index value. This means, lower index values means slower execution. To calculate an index, baseline values
already hardcoded into UnixBench are used. All individual indices are then combined to make a global index for the system as a whole. Although the global value is good, in our results, we will look through each individual index value to get an insight into where the overhead occurs the worst.

Even though results from UnixBench will answer most of the application programmers’ questions, system programmers will need a more fine-grained evaluation of the overhead. The most important question to answer is the worst case interrupt response latency after the secure kernel has been introduced. Rather than measure actual interrupt latencies over a long period of time and report the maximum value, it is more accurate to break down the problem into a deterministically measurable value. So which is the worst time to get an interrupt for the non-secure world? The moment when the SRTC interrupt goes off and the ARM core is about to trap into the monitor mode from the non-secure world seems to be the worst time, since, the core would have to switch back to the non-secure world to handle the interrupt. Thus, the worst case interrupt response latency will increase by time required to perform two world switches. The world switch time is largely dependent on the quality of code, features in the secure world (MMU, caches) and vendor specific interrupt acknowledgement times. Thus, instead of reporting raw world switch values, it will be helpful to report two separate values: The time required for the ARM core to trap inside the monitor mode and the actual world switch time. The former is the minimum bound on the world switch time. The latter gives an idea of the actual overhead caused to do a world switch. Note that the latter will change significantly with different vendors. Typically both these times should be in low microseconds.

To device an accurate way to measure the switch time, we use the Performance
Monitoring Unit (PMU) of the ARMv7 core. The PMU provides a cycle counter called PMCCNTR (Performance Monitoring Cycle Counter). To actually read from and write to the PMU registers incurs only one instruction overhead. Thus, the cycle counter is accurate. The accurate cycle count for context switch from any world to monitor mode can be measured by placing two cycle count instructions strategically: One before the smc instruction and another as the first instruction of the smc handler. Measuring world switch time is simpler, the cycle counts at the entry of the exception and at the exit are measured. The cycle counts are then converted to time in microseconds for a more intuitive result.

With these measurement principles, the next section reports results of the tests that were performed.

5.3 Results

5.3.1 Application Overhead

The following table shows the result of UnixBench executed with 25 iterations.

Let us first look at the comparison between a non-TZ system and TZ system with no task running. Refer to table 5.1.

As discussed before, Dhrystone and Whetstone are CPU-bound benchmarks. The former has string and integer manipulations whereas the later has double-precision floating-point operations. Note that CPU bound tasks are hardly affected by the presence of the secure kernel. The File-copy benchmark exclusively exercises the I/O overhead. Note that in all three such benchmarks, the overhead due to merely presence of the secure kernel is around 7%. Two reasons for the overhead are sharing of cache space with the
Table 5.1: Runs of UnixBench on system without the system kernel and one with it.

<table>
<thead>
<tr>
<th>Test Name</th>
<th>Without TZ</th>
<th>With TZ</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No task</td>
<td>Task: always on</td>
</tr>
<tr>
<td>Dhrystone 2</td>
<td>296.9</td>
<td>296.4</td>
</tr>
<tr>
<td>Double-precision Whetstone</td>
<td>22.6</td>
<td>22.6</td>
</tr>
<tr>
<td>Exec</td>
<td>81.3</td>
<td>81.7</td>
</tr>
<tr>
<td>File Copy (1024 bufsize, 2000 blocks)</td>
<td>46.8</td>
<td>43.2</td>
</tr>
<tr>
<td>File Copy (256 bufsize, 500 blocks)</td>
<td>44.7</td>
<td>41.7</td>
</tr>
<tr>
<td>Filecopy (4096 bufsize, 8000 blocks)</td>
<td>35.9</td>
<td>31.6</td>
</tr>
<tr>
<td>Pipe Throughput</td>
<td>230.3</td>
<td>161.3</td>
</tr>
<tr>
<td>Pipe-based Context Switch</td>
<td>154.1</td>
<td>126.3</td>
</tr>
<tr>
<td>Process Creation</td>
<td>72.3</td>
<td>35.4</td>
</tr>
<tr>
<td>Shell Script (1 concurrent)</td>
<td>93.9</td>
<td>80.7</td>
</tr>
<tr>
<td>Shell Script (8 concurrent)</td>
<td>100.7</td>
<td>101.0</td>
</tr>
<tr>
<td>System Call Overhead</td>
<td>312.5</td>
<td>280.0</td>
</tr>
<tr>
<td><strong>Overall Index</strong></td>
<td><strong>90.7</strong></td>
<td><strong>78.5</strong></td>
</tr>
</tbody>
</table>

secure world and increased interrupt response latency because of the presence of the secure world. The four benchmarks: Pipe throughput, pipe-based context switch, process creating and system call overhead measure the exact time to carry out a certain operation. An inopportune context switch to the secure world during the operation will introduce big outliers in their reading and that is why we can see a significant drop in their performance index. There is a 10% drop in the Single Shell Script benchmark too. This can be attributed to the I/O portion of the hybrid CPU and I/O workload a shell-script gives rise to. In summary, with just the presence of the secure kernel, CPU bound processes face low overhead whereas I/O bound processes suffer an overhead of almost 7-10%.

The above considered the overhead in case of a secure world with no task running i.e. the overhead of merely the world switches. Now consider a case where the secure world
has a long running task. To simulate this, we started a task in the secure world with an infinite loop inside it. Recall that the time slice ratio used is 1:3 (secure:non-secure) i.e. ideally we should see a 25% overhead in all our operations. Now, compare column 1 (no TZ) and column 3. Notice that both CPU bound benchmarks (Dhrystone and Whetstone) suffer from an exact 25% overhead. The I/O bound file-copy benchmarks’ overheads vary wildly from 12% to 35%. Note that with small bufsize, the overhead is less and as the buffer size increases, the overhead increases. A reason for this would be the sharing of cache lines among both the worlds. Higher bufsize needs more cache space and hence, a higher chance of eviction from the cache when secure world executes. Thus, we conclude that to optimize I/O performance, it is necessary to optimize cache usage in the non-secure world.

5.3.2 Memory Protection Overhead

Memory protection by programming the TZASC, TZMA is the most invasive aspect of TrustZone. Each memory access has to go through the TZASC. As such, it is necessary that the memory protection controller be fast. This section executes the UnixBench in two cases: with memory protection turned on and memory protection turned off.

Table 5.2 shows that enabling memory protection hardly affects the application performance (about 2% of performance gain. Some of it might be due to environment variance too).

5.3.3 Switching Time

Lastly, table 5.3 summarizes monitor trap time and world switch time. The time in microseconds is derived by dividing the cycle count with the CPU frequency (1 GHz).
Table 5.2: Runs of UnixBench on a TZ system with and without memory protection enabled.

<table>
<thead>
<tr>
<th>Test Name</th>
<th>With TZ (No task)</th>
<th>Mem Prot. On</th>
<th>Mem Prot. Off</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dhrystone 2</td>
<td>296.4</td>
<td>297.3</td>
<td></td>
</tr>
<tr>
<td>Double-precision Whetstone</td>
<td>22.6</td>
<td>22.6</td>
<td></td>
</tr>
<tr>
<td>Exec</td>
<td>81.7</td>
<td>82.7</td>
<td></td>
</tr>
<tr>
<td>File Copy (1024 bufsize, 2000 blocks)</td>
<td>43.2</td>
<td>49.4</td>
<td></td>
</tr>
<tr>
<td>File Copy (256 bufsize, 500 blocks)</td>
<td>41.7</td>
<td>51.4</td>
<td></td>
</tr>
<tr>
<td>Filecopy (4096 bufsize, 8000 blocks)</td>
<td>31.6</td>
<td>30.9</td>
<td></td>
</tr>
<tr>
<td>Pipe Throughput</td>
<td>161.3</td>
<td>165.3</td>
<td></td>
</tr>
<tr>
<td>Pipe-based Context Switch</td>
<td>126.3</td>
<td>132.3</td>
<td></td>
</tr>
<tr>
<td>Process Creation</td>
<td>35.4</td>
<td>33.5</td>
<td></td>
</tr>
<tr>
<td>Shell Script (1 concurrent)</td>
<td>80.7</td>
<td>79.4</td>
<td></td>
</tr>
<tr>
<td>Shell Script (8 concurrent)</td>
<td>101.0</td>
<td>99.7</td>
<td></td>
</tr>
<tr>
<td>System Call Overhead</td>
<td>280.0</td>
<td>294.2</td>
<td></td>
</tr>
<tr>
<td>Overall Index</td>
<td>78.5</td>
<td>80.0</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.3: Time required to switch between the worlds.

<table>
<thead>
<tr>
<th>Test Name</th>
<th>Time (in microseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monitor trap time</td>
<td>0.45</td>
</tr>
<tr>
<td>World Switch time</td>
<td>14</td>
</tr>
</tbody>
</table>

Note that the context switch time is very small. Of the order of half a microsecond. Also, the world switch time is acceptably low i.e. 14 microseconds. With these values, the maximum added interrupt latency is $2 \times (14 + 0.45) = 28.9$ microseconds. The system designer should take this value into consideration while designing the system (and handle the interrupt from the monitor mode, if this latency is unacceptable).

In summary, from the above values we can conclude that the presence of the secure kernel causes hardly any performance over-head for CPU-bound tasks. The I/O bound tasks do experience about a 7% slowdown. Memory protection incurs another small overhead. The secure:non-secure timeslice should be wisely chosen to avoid slowdowns.
in the non-secure world. Also, the world switch times are low, but the system designer should take into consideration that it will to the interrupt response latency.
Chapter 6

Conclusion

We presented ARMithril in this paper - a lightweight secure kernel that runs in a trusted execution environment in parallel with a commodity Operating System. We showed how ARM’s TrustZone technology can be leveraged to write a small and hence, easily verifiable operating system that can perform security sensitive tasks like DRM backends, virus scanner, etc. We showed how isolation can be guaranteed by physically partitioning resources like CPU, memory, interrupts and I/O. Our implementation conforms to the TrustZone API 3.0 which means porting applications to run on our TZ-based system only means porting the secure service to the secure kernel.

Our future work includes feature additions to the secure kernel. Making MMU optionally available, implementing a better task scheduling algorithm and providing a richer set of utility functions inside the secure kernel that the secure services can directly use. Conformance to GlobalPlatform’s TEE client API is also a long term goal.
REFERENCES


Appendix A

World Switch Code

/* This branch is called from two locations. FIQ and
 * from smc handler. So, any modifications below this point
 * should be carefully reviewed so as to be compatible with
 * both. */

switch_worlds:
    tst r2, #0x01

    /* Save to */
    ldreq r0, =secure_state
    ldrne r0, =non_secure_state

    /* Save r0–r12 */
    pop {r4–r12}
    stmia r0!, {r4–r12}
    pop {r4–r7}
    stmia r0!, {r4–r7}

    mrs r1, spsr
stmia r0!, {r1, lr}  /* spsr, lr */

/* Switch to SVC mode. Get SP and come back * to monitor mode */
cps #0x13
stmia r0!, {sp}
stmia r0!, {lr}
cps #0x16

/* restore from */
ldreq    r0, =non_secure_state
ldrne    r0, =secure_state

mov     r3, r0  /* save current r0 */
add     r0, r0, #16 /* skip r0 to r3 for now. */
          * We need them as scratchpad */

/* Restore old state */
ldmia r0!, {r4–r12}
ldmia r0!, {r1, lr}
msr spsr_cxsf, r1

/* Restore SVC’s sp */
cps #0x13
ldmia r0!, {sp}
ldmia r0!, {lr}
cps #0x16

/* Switch to other state */
eor r2, r2, #0x1 /* Flip the NS-bit */
mcr pl5, 0, r2, c1, c1, 0

ldmia r3, {r0–r3}
movs pc, lr
Appendix B

Skeleton of TZ Application Code

```c

tz_return_t   ret;
tz_device_t   dev;
tz_operation_t op;
tz_uuid_t     uuid = 10;

printf("Initialising\n");

/* Open TZ Device */
ret = TZDeviceOpen("cdl-tz", NULL, &dev);
if (ret != TZ_SUCCESS) {
    printf("Failed to open device\n");
    exit(1);
}

/* Operating init */
ret = TZOperationPrepareOpen(&dev, &uuid,
                            NULL, NULL, NULL, &op);
if (ret != TZ_SUCCESS)
    exit(1);
```
/* Start task */
TZOperationAsyncStart(&op, NULL);

/* Collect result. Can be made to return
 * if task is not complete */
TZOperationAsyncGetResult(&op, NULL);
printf("Task complete\n");

/* Close device */
TZDeviceClose(&dev);