|  |
| --- |
| Iowa State University |
| Android Security Final Design Report |
| Senior Design MAY 12-08 |
|  |
| **Alexander M.. Frisvold, Alex Meyer, Nazmus Sakib, Eric Van Buren** |
| **4/26/2012** |



Table of Contents

[1. Project Overview 2](#_Toc323182616)

[1.1. Executive Summary 2](#_Toc323182617)

[1.2. Acknowledgement 2](#_Toc323182618)

[1.3. List of Terms 2](#_Toc323182619)

[1.4. Problem Statement 3](#_Toc323182620)

[1.5. Operating Environment 3](#_Toc323182621)

[1.6. Intended Users and Uses 3](#_Toc323182622)

[1.7. Expected End Product and Other Deliverables 4](#_Toc323182623)

[2. Design Requirements 4](#_Toc323182624)

[2.1. Functional Requirements 4](#_Toc323182626)

[2.2. Non-Functional Requirements 5](#_Toc323182627)

[2.3. Concept Sketch 5](#_Toc323182628)

[2.4. High-Level Design 8](#_Toc323182629)

[2.5. Low-Level Design 9](#_Toc323182630)

[2.5.1. Software Stack 9](#_Toc323182643)

[2.5.2. TPM Services 9](#_Toc323182644)

[2.5.3. Android Security Application 10](#_Toc323182645)

[2.7. Alternative Designs 11](#_Toc323182652)

[3. System Analysis and Design tradeoffs 14](#_Toc323182653)

[3.1. Android OS vs. Other Mobile Operating Systems 14](#_Toc323182655)

[3.2. Developing From Scratch vs. Using Pre-Existing Components 14](#_Toc323182656)

[3.3. gsl\_rng\_ranlxs2 vs. other generators in the GNU Scientific Library 14](#_Toc323182657)

[3.4. Difficulties Encountered 15](#_Toc323182658)

[4. Standards 16](#_Toc323182659)

[4.1. Process Improvement Plan 16](#_Toc323182661)

[4.2. Product Acceptance Plan 16](#_Toc323182662)

[4.3. Coding Standards 16](#_Toc323182663)

[4.4. GNU Open Source Standards 16](#_Toc323182664)

[5. Operation Manual - Build Document 17](#_Toc323182665)

# Project Overview

## Executive Summary

Our project is to develop a working emulator for an Android device so that it is able to accurately represent the security extension of the device’s CPU, known as ARM TrustZone. Using TrustZone, we will provide a Trusted Platform Module (TPM) service.

ARM TrustZone works by providing two virtual processors backed by hardware based access control. This enables the application core to switch between two states, referred to a worlds, in order to prevent information from leaking from the more trusted world (secure world) to the less secure world (non-secure world).

The way we approached this problem was by constructing a software stack consisting of a hardware emulator, a microkernel, a run-time environment, and the Android OS. We made necessary modifications to different layers in the stack to allow TrustZone support.

## Acknowledgement

* Dr. George Amariucai – Faculty Advisor – Provided advice on the project management through weekly meetings.
* Victor Lukasik – Client Advisor – Discussed the project and provided advice during bi-weekly meetings.
* Johannes Winter – Institute for Applied Information Processing and Communications. Graz, University of Technology – Created experimental version of QEMU with basic support for ARM TrustZone.

## List of Terms

**Trusted Platform Module (TPM)** – a piece of hardware that provides a means of a secure storage area that is encrypted using a protected private key, making it so only that chip can decrypt the data.

**TrustZone** – security extensions made to the ARM chipset family. They provide the means to implement a trusted execution environment by being able to create two virtual processors on one core and enforcing a strict separation between the two.

**Trusted Execution Environment** **(TEE)** – is a separate execution environment that runs alongside the Rich OS but remains isolated from it. This isolation allows for the safe execution of authorized security software and helps to enforce protection, confidentiality, integrity and the access rights of the data and resources that belong to the software running in the TEE.

**QEMU –** a powerful processor emulator that uses dynamic binary translation to allow it to be ported to many different CPU architectures.

**API -** Application Programming Interface; is a particular set of rules and specifications that a software program can follow to access and make use of the services and resources provided by another particular software program.

**Fiasco Microkernel –** software needed to implement the runtime environment that will run on top of it, this is the only piece of software that will run in the secure world of the processor.

**Fiasco L4Re** – provides a basic set of services and abstractions, which will be used to implement and run applications on top of the microkernel.

**L4 Android/ L4Linux –** an operating system developed to run on top of the Fiasco L4Re run-time environment, provides a layer for us to develop applications on top of. Also has been specifically modified to replicate the Android

**CodeSourcery G++ Cross Compiler –** The cross compiler we used to compile all of the source code, a cross compiler was necessary because all of the code was written for x86 instruction set and we needed it to run on our ARM processor.

**Secure Monitor Call (SMC) –** A new instruction added by ARM to enable switching execution environments from normal world to the safe world and back again.

## Problem Statement

This document will detail the current plant of implementation for our senior design project. Our project is to develop a working emulator for an Android device, be it a cell phone or a tablet, such that it is able to accurately represent an ARM TrustZone.

As of right now there are no commercial Android emulators that can correctly depict the workings of an ARM TrustZone. Therefore, application developers are forced to test their devices on actual hardware to see if their code works. This is both dangerous to the device and time consuming. The logical next step is to try and emulate the inner workings of TrustZone so that developers can then start writing applications to use it.

## Operating Environment

The extended Android QEMU emulator will allow for the full system emulation of the L4Android operating system on a virtual ARM CPU that implements the TrustZone architecture on an x86 Linux host machine, more specifically Ubuntu 10.04 LTS. The existing QEMU source is written in C and any necessary modifications will also be written in that language. The Fiasco.OC microkernel and the L4Re runtime environment are implemented in C and C++ so we will be using those languages to make any changes to those two components. The Android applications we will develop will be written in Java using the Eclipse IDE.

## Intended Users and Uses

This software stack is designed for two different types of users. The first and most likely the largest group of users is going to be Android application developers. This suite allows these users to work on their normal Android applications but also test how any of their TrustZone functionality will work. While there are lots of development suits out there, this stack is the only fully open source, so free, option.

The second, smaller, group would be a people who are simply interested in exploring ARM’s TrustZone. This group would be academics, hobbyists and security professionals who are interested in obtaining a better understanding of TrustZones behavior. This software stack would also be preferable to this group because of its open source roots. The users can view all source code and manipulate it in any way they want, for free.

## Expected End Product and Other Deliverables

The final goal of this project would be to deliver a working software stack that correctly emulates an Android application using the ARM TrustZone. The project would allow developers to emulate the principle of the ARM TrustZone functionality and using this functionality also to also provide a TPM service. Our software stack should correctly emulate the ARM Cortex-A7 chipset as well as provide a random number generator and public/private key generator. The current design was decided based upon the idea that we would like every part to be open source and modifiable. As there is no current emulator like ours on the market, much of our project is based on research and gaining a full understanding of ARM TrustZone and TPM functionality.

# Design Requirements

1.

## Functional Requirements

1. **The modified Fiasco.OC microkernel will run seamlessly over Mr. Winter’s extended version of QEMU.**

The layer between the Fiasco microkernel and Johannes Winter’s augmented QEMU is the lowest layer in the stack. Getting this layer to work correctly is important because the Fiasco microkernel emulates the Secure Monitor and will need the augmented QEMU because it emulates the ARM processor.

1. **The modified L4 runtime environment will run seamlessly over the modified Fiasco.OC microkernel.**

This layer in the stack is the second lowest layer. The L4 runtime environment provides very little to the kernel, mainly some basic interfaces. The L4Re aims at providing more convenient abstractions for application development. It comprises low-level software components that interface directly with the microkernel. The root-task, known as Moe, and the root-page, known as Sigma0, are the most basic components of the runtime environment.

1. **The L4Android operating system will run seamlessly over the modified L4 runtime environment.**

The L4Android OS is needed with the L4 Runtime environment because it allows us to run the Android OS on top of the Fiasco.OC microkernel. We need the Android OS because we will be emulating an Android phone.

1. **Our software stack will use the secure world to provide two TPM services: random number generation and RSA key generation.**

The secure world of the emulated TrustZone should provide a trusted execution environment that can be used to securely provide the two cryptographically important TPM services.

1. **An Android application will be able to use the TPM services provided and will be encrypt and decrypt sensitive data using the secure world of our software stack.**

The Android application will be able to use the implemented TPM services to encrypt and decrypt sensitive data, specifically usernames and passwords. The application will run on the secure world of the software stack to ensure the security of the sensitive data.

1. **Modifications made to any of the various components of the software stack should not adversely affect any of the existing functionality of any of the components.**

In order to implement our required functionality it will be necessary for us to change or add significant amounts of code to the software components we are using. These modifications should not lead to any of the components malfunctioning.

## Non-Functional Requirements

1. The modified software stack should run at a usable speed.
2. The modified software stack should be stable and run reliably.
3. Modifications to QEMU, Fiasco.OC and L4Re should be written in C and C++ programming language on a Linux platform.

Our software stack should allow other TPM services to be added later.

## Concept Sketch

The important components we are working with namely the ARM TrustZone, Trusted Platform Modules and QEMU are described in this section. This is followed by a discussion of what our project broadly seeked to accomplish.

**TrustZone:** The ARM TrustZone is a secure execution environment. TrustZone provides two virtual processors on a single processor core backed by hardware access control. This facilitates the creation of two separate parallel execution worlds: a non-secure “normal” execution environment and a trusted secure world. Hardware logic present in the TrustZone-enabled AMBA3 AXI bus fabric ensures that no secure world resources can be accessed by the normal world components.

A simple overview of how TrustZone operates is demonstrated below. A secure monitor is in charge of the switching between worlds and is in place to not allow programs that are in the normal world access the secure world. The functionality of the secure monitors is similar to a traditional operating system context switch thereby ensuring that the state of the world that the processor is leaving is safely saved and the state of the world that the processor is switching to is correctly restored. The SMC instruction in the ARM instruction set provides the main route change worlds.



**Trusted Platform Module:** The Trusted Platform module is a hardware component specified by the Trusted Computing Group to facilitate several important cryptographic functions and act as a root of trust for storage and reporting. A TPM provides four basic classes of functions:

* Securely storing and reporting platform configurations
* Storage of protected keys and data
* Cryptographic functionality
* Initialization and management functionality

Given the capabilities of a TPM, it can serve as an important part of a trusted computing system. The following figure summarizes the architecture and functionality of a typical TPM.



What we were primarily interested in is the cryptographic services provided by the TPM. The mandatory cryptographic functions of a TPM are:

* Either an algorithmic or a true hardware-based pseudo random number generator
* An RSA signing, encryption and key-generation unit.
* A message digest function SHA-1 and corresponding message authentication code (HMAC) engine.

The functionality we focused on is the pseudo random number generation and RSA key generation.

**QEMU:** QEMU is a machine emulator. It can be used to run an unmodified target operating system and all its applications on a virtual machine. Several different host operating systems are supported including Android, Linux, Mac OS X and Windows. The host and target CPUs can be different. QEMU is primarily used for three things:

* Running one operating system on another, for example Android on Linux.
* Debugging since the virtual machine can be stopped and its state can inspected, saved and restored.
* Simulation of specific embedded systems can be achieved by adding new machine descriptions and new emulated devices.

QEMU uses a dynamic binary translator to perform runtime conversions of the target CPU instructions into the host instruction set. The resulting binary code is stored in a translation cache so that it can be reused.

**Project concept**

The overall goal of our project was to use QEMU to emulate an ARM processor that implements the TrustZone and to use this emulated TrustZone environment to provide TPM cryptographic services. The services we intended to provide are pseudo random number generation and RSA key generation. Given its widespread popularity and open source nature we decided to use the Android operating system to interact with the underlying software stack that will have QEMU at its base.

## High-Level Design

The main objective of this project was to provide a way for Boeing and other application developers to test their secure programs before actually putting them on a device. To accomplish this task we were told to extend the already implemented Android emulator QEMU to accept ARM TrustZone instructions.



As there is no public API for the ARM TrustZone we were not be able to just implement this functionality. To accomplish our goal we looked at using a variety of other pieces of software that we have found. The first is an extended version of the processor emulator QEMU, it is extended so that it not only emulates the hardware processor but also accurately emulates the TrustZone hardware. The extensions were released as an experimental version of QEMU by Mr. Johannes Winter’s group. Mr. Winter is a researcher involved with Trusted Computing at the Institut für Angewandte Informationsverarbeitung und Kommunikationstechnologie (IAIK) at Technische Universität Graz (TU Graz) in Austria. The next layer necessary is that of a microkernel that will sit above the hardware and provide basic resource management as well as support for TrustZone. Based on the recommendation of Mr. Winter we are using the Fiasco microkernel. The next layer of the software stack we need is that of a run-time environment. L4Re will communicate between the Android operating system and the Fiasco kernel. Above the run-time environment is the operating system layer. We will be using the L4Android operating system since it supports the L4Re. This will allow us to develop Android applications that will test our implementation of the TrustZone. Using this layered approach we should be able to run applications that use TrustZone.

## Low-Level Design

* 1.
	2.
	3.
	4.
	5.
1.
2. 1.
	2.
	3.
	4.
	5.

### Software Stack

**QEMU**: For a more complete overview of this project we must first start at the lowest level which is of course the QEMU emulator with the TrustZone extension. This is a very powerful and complex piece of software as it is able to emulate many different processors using dynamic binary translation. This is the process of emulating an instruction set through translation of code. This piece of code allows us to emulate the ARM processor of our choosing. We found a modified version of QEMU online from an Austrian researcher; this version implemented the hardware emulation needed for TrustZone.

**Fiasco.OC**: It is a microkernel developed by a group at TU-Dresden. A microkernel is a component at the lowest level of the software stack. It is the only piece of software that is running in the privileged mode of the processor. The kernel provides primitives to execute programs in tasks, to enforce isolation among them, and to provide means of secure communication in order to let them cooperate. As the kernel is the most privileged, security-critical software component in the system, it is a general design goal to make it as small as possible in order to reduce its attack surface. It provides only a minimal set of mechanisms that are necessary to support applications. It does not include complex services such as program loading, device drivers, or file systems; those are implemented in user-level programs on top of it (a basic set these services and abstractions is provided by the L4Re).

**L4Re**: The small kernel offers a concise set of interfaces, but these are not necessarily suited for building applications directly on top of it. The L4 Runtime Environment aims at providing more convenient abstractions for application development. It comprises low-level software components that interface directly with the microkernel. L4Re consists of a set of libraries and servers. Libraries as well as server interfaces are completely object oriented.

**L4Android**: L4Android is derived from the L4Linux project, which is developed at the Technische Universität Dresden. L4Linux is a modified Linux kernel, which runs on top of the Fiasco.OC microkernel. It is binary compatible with the normal Linux kernel. L4Android combines both the L4Linux and Google modifications of the Linux kernel and thus enables us to run Android on top of a microkernel.
The L4Android kernel supports the x86 and ARM platform and both Froyo (2.2) and Gingerbread (2.3) releases are supported.

### TPM Services

**Pseudo Random Number Generation:** Pseudo random number generation for security applications have some additional requirements besides passing basic statistical tests for randomness. Ideally they should provide extremely long periods and low correlation thereby increasing the probability that an adversary attacking the random number generator that did not know the seed for the generator would have a more difficult time trying to distinguish the generator’s output sequence from a random sequence. We intended to implement a C++ pseudo random number generation program that would be an extension to the Fiasco microkernel. We intended to use the freely available GNU Scientific Library that provides many random number generators. We planned on using the gsl\_rng\_ranlxs2 generator. This generator is a second-generation version of the RANLUX algorithm developed by M. Lüscher. The RANLUX algorithm has mathematically proven properties and can provide truly decorrelated numbers at known levels of randomness. The generator provides 24 bit single precision output that passes all known tests for randomness. The period of the gsl\_rng\_ranlxs2 generator is 10^171. The program will run in the secure world of our stack since it will be implemented as an extension to the Fiasco microkernel. The random numbers generated will be available for use by other applications.

**RSA Key Generation:** RSA is a widely used algorithm for public-key cryptography. The RSA algorithm involves three steps: key generation, encryption and decryption. Our focus is on key generation. We need to generate two keys: a public key for encryption and a private key for decryption. We intended to implement a C++ RSA key generation program that would be an extension to the Fiasco microkernel. The program would use the random numbers generated by our random number generation program to get two distinct prime numbers required for the key generation process. The program would then complete the remainder of the key generation procedure and then make the public key available and store the private key in the secure world of our software stack. The entire program will run in the secure world since it will be implemented as an extension to the Fiasco microkernel.

### Android Security Application

We intended to implement an Android application that will use the TPM services running in the secure world to encrypt a small amount of sensitive data specifically usernames and passwords. The application will be able to access the public encryption keys generated by our RSA key generation program to encrypt data and will be able to use the private decryption keys to decrypt data. This application will run on the secure world.

* 1.
	2.
	3.
	4.
	5.
	6.

## Alternative Designs

**Hypervisor Solution**: In this design a hypervisor would switch between two instances of the Fiasco microkernel. Each microkernel would be in one world. The hypervisor would be responsible for identifying the nature of each individual instruction and then forwarding it to the microkernel in the correct world. If a context switch needs to be made the hypervisor takes care of the necessary details. The hypervisor would be responsible for any communication between the worlds.

Two hypervisor were considered for our stack. NOVA, is a microhypervisor released by the group at TU Dresden that built Fiasco and L4Re. Naturally, NOVA is built to work with the other two components. This would make it an attractive choice. A second option would be to use Choices, a small object-oriented operating system released by a group at the University of Illinois at Urbana-Champaign. Choices was used in the past to implement similar use cases to our project for older ARM architectures that predated the TrustZone extensions.

The hypervisor solution is attractive since it isolates the two kernels and consequently the two worlds in a complete manner. Implementing the solution maybe more time consuming since another software component will need to be understood, compiled, run and possibly modified.

The design is described in the picture given below:



**Dual MMU Solution:** In this design QEMU would be modified to have two memory management units (MMU) with each managing a microkernel. Like the previous design each microkernel would be in one world. The normal MMU would be subservient to the secure MMU. The nature of each instruction would be polled in QEMU by the secure MMU and would then be processed by the correct MMU. The secure MMU would be responsible for administering context switches. The secure MMU would have access to the memory of the normal world as well and this will allow it communicate any results to the normal world by writing in a portion of the memory that the normal MMU has access to.

The design has the advantage of not introducing additional software components to the stack. The two MMUs should also be able to isolate the two worlds effectively. Implementing modifications to the QEMU MMU would however be a very difficult task owing to the complexity of the QEMU project in general and the sensitive nature of a module like the MMU.

The design is described in the picture given below:



**Static Memory Division:** In this comparatively simple design, there will be only one microkernel and one MMU. The separation of the worlds will be achieved by statically allocating a section of memory in QEMU to the secure world and the rest to the normal world. The Fiasco microkernel will monitor the nature of each instruction and would forward each instruction to the correct area of memory. The microkernel will be responsible for administering context switches. The secure world would have access to the memory of the normal world as well and this will allow it communicate any results to the normal world by writing in a portion of the memory that the normal world has access to.

This design is much simpler than the other two discussed and would be easier to implement. The task is however not inconsiderable since any modifications to the memory in QEMU would need to be done carefully. This design also provides a potentially less secure separation between the two worlds.

The design is described in the picture given below:



# System Analysis and Design tradeoffs

1.

## Android OS vs. Other Mobile Operating Systems

Mobile security is a concern no matter what operating system we are looking at. Almost all smartphones on the market today run either Android, iOS, BlackBerry OS, or Windows Phone. All of these operating systems run on ARM chips as well. The decision to focus on Android was motivated by a number of factors. Android is open source unlike the other three operating systems mentioned. This ensures that we have the capacity to make any and all the changes to the operating system source code that we need to implement our system and negates intellectual property concerns. The open source nature of Android has encouraged many people to start working with the system and means that there is already a large group of developers sharing resources and knowledge. This will be invaluable in the future. Android also happens to be the most popular smartphone operating system at the moment with a nearly 53% share of the world smartphone market. This high market share increases impact that our project can have out in the real world.

## Developing From Scratch vs. Using Pre-Existing Components

As we progressed with the initial research for the project it became obvious that developing a complete system that extended QEMU to support the ARM TrustZone was beyond the time constraints of this project. Using pre-existing components such as Mr. Winter’s QEMU and the Fiasco.OC microkernel would remove a significant amount of risk from our project. The team collaborated with Mr. Winter and our communications with him have been productive. Fiasco.OC, L4Re, and L4Android are well documented and since these projects are much more documented then other components we initially considered

## gsl\_rng\_ranlxs2 vs. other generators in the GNU Scientific Library

While the gsl\_rng\_ranlxs2 generator passes all known tests for randomness it is worth pointing out here that the output produced is still theoretically defective. However given the constraints on the time we have for development we have decided to not look at more cryptographically secure pseudo random number generation algorithms. Another reason for using the gsl\_rng\_ranlxs2 over some of the other generators that the GNU specific library is that it provides a good balance between performance and resource use. We are developing our stack with mobile platforms in mind and they have stricter constraints on processor power and available memory. Double precision output can be produced with other generators but they may put an unacceptable strain on mobile platforms.

## Difficulties Encountered

**Lack of Documentation**

Due to the experimental nature of this project we were expecting to run into several difficulties. The one difficulty we struggled mostly with was the lack of documentation that was available. At every level in the stack, we struggled to find documents for building components correctly. The documents that we did work with were lacking since we needed to build using ARM architecture and most documentation was for x86 architecture.

**Errors in building components**

At first we built everything using x86 architecture since we were emulating on an x86 computer. We later realized that we need to configure the stack using ARM architecture, more specifically the ARMv7A chipset. After configuring it this way, we began getting several errors about arm\_linux\_gcc not found. After doing some research we decided that we need a cross compiler to get arm\_linux\_gcc commands to work. A cross compiler would allow us to run ARM architecture instructions on an x86 computer.

**Difficulties with a cross compiler**

Before we could move on with the project we had to find a cross compiler that would work. Two cross-compilers that we first tried were CrossTool and GNUARM. CrossTool didn’t work because it would tell us that certain instructions were not ARM compatible. GNUARM failed to work for us because it simply would not compile correctly. The next cross compiler tool we used was CodeSorcery. It seemed to fix all issues that we’d been having with other cross compiler tools.

**Other difficulties**

Originally we ran on a Debian box. But when we did some test builds of Android images, the emulation would run extremely slow and would take up to 10 to 15 seconds to respond. So we decided to reformat and use Ubuntu 10.04 LTS, which fixed the problem.

It is said that failure is the necessity of invention and that it is essential for success. As Winston Churchill once said, “Success is the ability to go from one failure to another with no loss of enthusiasm.” Every success in this project was followed after a long string of failures. When we couldn’t find documentation on builds, we weren’t afraid to try several different builds that we thought would work. There were several times throughout the project where had to completely rebuild the stack and start from scratch again.

# Standards

1.

## Process Improvement Plan

Every week we met with our advisor to examine the progress of our project and to determine if any changes needed to be made to our work plan. Our advisor would challenge us by asking questions about the progress we made and also to see if we truly understood what we were working on. Besides our weekly advisor meetings we had bi-weekly meetings with our client. In the client meetings we would explain the progress, current work, and goals of our project. They would either have us continue along with our plan or request that we change it in a way that thye wanted

## Product Acceptance Plan

The acceptance plan specifies the plan for the acquirer (The Boeing Company) to accept the deliverable and have object criteria for determining acceptability of the work accomplished by our team. We have demonstrated our work to The Boeing Company showing which functional requirements were satisfied.

## Coding Standards

As with any coding project the language standards must be maintained . The following languages were used and language standards maintained:

* C
* C++
* Java

## GNU Open Source Standards

In accordance to the GNU open source standards, we have used code that falls under the GNU and credited any of its contributors. None of the code is being used for commercial profit and the GNU comments were left in as is. We will also contribute to the GNU open source library by providing our code that we added in developing this stack

# Operation Manual - Build Document

QEMU:

1. Get source at: <https://github.com/jowinter/qemu-trustzone>
2. Unzip the source files and go into that directory
3. Type the following command $./configure

Fiasco, L4RE, L4Linux:

 **Fiasco**

1. Get source at: <http://os.inf.tu-dresden.de/download/snapshots-oc/>
	1. Get the snapshot link not the core as the snapshot includes a bunch of libraries needed for future steps
2. Unzip source and cd into the src directory of the snapshot

Note -> Our source directory for this project was: /home/srdesign/Documents/l4re/ this will be used in the following examples as <homedir>

1. From the /src directory run the following command

$make setup

* 1. This will bring up a list of possible building architectures, select ARM
	2. This will automatically setup the entire project for an ARM based project, this saves a lot of time and correctly links directories
1. Create the build directory
	1. $ make BUILDDIR= <homedir>/obj/fiasco/arm-rv/
2. Go into the build directory
	1. Located at <homedir>/obj/fiasco/arm-rv/
3. Configure the kernel
	1. $ make config
		1. The configuration has a simple to use GUI for its setup

This is the automatic configuration building for Fiasco, it automatically writes the globalconfig.out file once you save and exit the file.

* 1. Proceed into the Target Configuration option



* 1. Change the target architecture from IA-32 to ARM processor family
	2. Now we need to change the platform we want to compile for, change the platform to the ARM RealView Platform as this gives you the option for TrustZone enabled processors.



1. Now we need to change the CPU we would like to compile to, change it to the ARM Cortex-A8 CPU.
2. You have now configured the kernel for our ARM build, select the Save configuration file and allow the global configuration files to be written. You are now ready to compile the microkernel
	1. Enter the command

$ make

**L4RE**

1. Traverse into the src directory located at: <homedir>/src/l4
2. Run the following command to set the build directory

 $ make B=<homedir>/obj/l4/arm-rv

* 1. This sets the build directory to the <homedir>/obj/l4/arm-rv directory
1. Go into your <homedir>/obj/l4/arm-rv directory and run this following command to configure the runtime environment

 $ make O=<homedir>/obj/l4/arm-rv config

1. This will bring up the configuration menu screen
	1. Change the Target Architecture to ARM
	2. Change the Platform Architecture to ARM Realview
	3. Change the CPU Variant to ARMv7A type CPU
	4. Select Save configuration file and allow the global configuration file to be written
2. Now that the runtime environment is configured correctly, compile it
	1. Run the command

$make O=<homedir>/obj/arm-rv

**L4Linux**

1. Traverse into your L4Linux directory located at <homedir>/src/l4linux
2. Run the command,

$ make L4ARCH=arm CROSS\_COMPILE=arm-linux- O=<homdir>/obj/l4linux/build arm\_defconfig

* 1. This sets up a build directory with ARM default configurations
1. Now we need to configure L4Linux by running the command

$ make L4ARCH=arm CROSS\_COMPILE=arm-linux- O=<homedir>/obj/l4linux/build menuconfig



* 1. Change the Target Architecture to ARM architecture
	2. Change the L4Linux configuration to your L4RE build directory, in our case that would be <homedir>/obj/l4/arm-rv



* 1. Save this configuration and exit the menu
1. To compile L4Linux run the following command

$ make L4ARC=arm CROSS\_COMPILE=arm-linux- O=<homedir>/obj/l4linux/build

* 1. If successful, vmlinuz.arm will be put under the/build directory in L4Linux

Additional Steps to make an image:

1. Once the above are configured and compiled correctly, you will need to copy the following files from this directory

<homedir>/src/l4/conf/examples/ to this directory <homedir>/obj/l4/arm-rv/bin/arm\_armv7a/

-l4lx.cfg

-arm-rv.io

-ramdisk-arm.rd

This needs to be done so that L4RE knows where all the components are to make the image.

1. Once this is done, go into the root L4RE directory located at <homedir>/obj/l4/ and run the following command

$ make elfimage

This will bring up a configuration similar to before, choose option 3.

1. If this is successful, a .elf image will be placed in <homedir>/obj/l4/arm-rv/images/

CodeSourcery Cross Compiler:

1. Download source at: <http://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/editions/lite-edition/>
2. Unzip the source files
3. Add the following command to your .bashrc file in your user login script
	1. export PATH=$PATH:/path/to/codesourcery/bin/directory

**References:**

QEMU: <http://wiki.qemu.org/Manual>

Fiasco: <http://os.inf.tu-dresden.de/fiasco/overview.html>

L4RE: <http://os.inf.tu-dresden.de/L4Re/overview.html>

CodeSourcery Cross Compiler: <http://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/editions/lite-edition/>