5-2011

TOWARDS SECURING VIRTUALIZATION USING A RECONFIGURABLE PLATFORM

Tushar Janefalkar
Clemson University, tushar.janefalkar@gmail.com

TOWARDS SECURING VIRTUALIZATION USING A RECONFIGURABLE PLATFORM

A thesis
Presented to
The Graduate School of
Clemson University

In Partial Fulfillment
Of the Requirements for the Degree
Master of Science
Computer Engineering

---

by
Tushar Janefalkar
May 2011

Abstract

Virtualization is no longer limited to main stream processors and servers. Virtualization software for General Purpose Processors (GPP) that allow one Operating System (OS) to run as an application in another OS have become commonplace. To exploit the full potential of the available hardware, virtualization is now prevalent across all systems big and small. Besides GPPs, state-of-the-art embedded processors are now capable of running rich operating systems and their virtualization is now a hot topic of research. However, this technological progress also opens doors for attackers to snoop on data that is not only confined to storage servers but also transferred to and used in important transactions on mobile platforms.

This work focuses on side channel attacks that arise due to hardware resource sharing between two concurrently running processes. These attacks can be in the form of monitoring cache accesses of a process or monitoring the power consumption of the system to determine the operation being performed. These attacks are seemingly harmless as the attacking process does not perform any illegal operations to snoop on the information available through side channels.

Side channel attacks have been used to easily decipher encryption keys for AES and RSA algorithms that are the two most commonly used encryption techniques. Software based solutions against these side channel attacks have been documented but do not guarantee a complete solution as they are either too specific to one aspect of an attack or demand changes to the Instruction Set Architecture (ISA) or static hardware designs. Implementation of such solutions is not always feasible.

In this project, we explore the virtualization of a PowerPC processor embedded on a Field Programmable Gate Array (FPGA) using the Kernel-based Virtual Machine (KVM). Then, we propose solutions that make use of the surrounding FPGA fabric to implement security measures that would make execution of side channel attacks difficult.

Lastly, this work provides detailed discussion on how to setup a development platform for FPGA-enabled hardware security, which involves cross compilation.

Dedication

I would like to dedicate this work to my family for their support throughout my Master’s program.

Acknowledgement

First and foremost, I would like to thank my advisor Dr. Melissa C. Smith for guiding me through this research. Her patient advice and encouragement has helped my professional growth beyond just research work. I would also like to thank Dr. Richard Brooks and Dr. Stanley Birchfield for being on my committee and reviewing my work.

I thank Mark Hur from Pico Computing, Inc. for providing us with the development board and Jeremy Chritz for his valuable technical support and guidance.

I would also like to thank my past and present colleagues in the Future Computing Technologies Lab for making the lab an enjoyable place to work and learn.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

Most processors run some form of operating system to manage tasks. Recently, embedded processors have become highly capable and run rich operating systems like Linux and Windows. Virtualization of embedded processors, where multiple Operating Systems (OS) run on the same processor via a hypervisor, is becoming a hot topic of research. These hypervisors are being widely implemented to develop applications and OSes that are device independent. Running on hypervisors, multiple OSes share hardware resources (cache, peripherals, RAM, etc.), which can lead to leakage of sensitive data through various side channels. In these side-channel attacks, the snooping process does not necessarily perform any illegal activities to recover sensitive information from the victim, but rather the attacker gathers information from side-channels that emerge due to the sharing of resources or physical properties of the hardware. This information can be in the form of tracking the number and location of cache accesses [1], [2], analysis of system power consumption while performing different operations [3], [4], or retention of data in DRAM even after power is turned off [5]. These attacks are difficult to detect since the attacker gathers information by performing seemingly harmless tasks, presenting a major challenge since countermeasures cannot be taken until an attack has been identified most likely during or after the attack.

In spite of these prevalent threats, cloud computing is on the rise as it is an attractive avenue to allow more users access to a service without having an onus of local processing. It is also easier to maintain a service in a centralized location rather than solving multiple issues at the user's local machine. As this practice becomes more widespread, the threat to the user's data through side-channel attacks becomes more prevalent given the current state of hypervisors and virtualization hardware. Currently, the U.S. Government ensures protection against side-channel attacks using Red-Black separation that dictates the use of separate power sources and hardware for different security levels. These kinds of security measures are expensive and their widespread implementation is difficult, especially in resource-constrained environments and in a cloud system where the allocated resources are constantly changing.

Defensive designs against such attacks are logically the best solutions. Although currently used hypervisors take advantage of hardware extensions provided by vendors like Intel [27] and AMD [28], they do not provide protection against side-channel attacks as these attacks exploit vulnerabilities at a different level beyond software control. Our research aims to develop effective defensive designs by implementing security measures in hardware using FPGAs. These designs are aimed at assisting hypervisors, or other concurrently running processes, to protect their sensitive data from side-channel attacks.

Traditionally, FPGAs have been used to create custom logic that allows efficient implementation of parallel algorithms and acceleration of parallel computation algorithms. Recent FPGAs have an embedded processor in the fabric presenting an attractive prototyping platform where customizable hardware and the embedded processor co-exist in the same device. Not only does this make interfacing of peripherals easier, it also gives the developer more control for monitoring every operation using sophisticated tools available from the FPGA vendors. In this project, we explore designs where security measures are implemented in the FPGA fabric with the embedded processor. Our design strives to eliminate the root cause that facilitates cache-based side-channel attacks - cache interference, by providing non-cacheable memory regions on the FPGA fabric that can hold sensitive data. This facility of storing sensitive data outside the general purpose RAM also provides protection against basic cold boot attacks. Further, we also propose designs exploiting the runtime reconfigurability of the FPGA [6] to protect against side-channel attacks that monitor the system's power consumption while performing different operations. This work is part of a larger plan to provide robust solutions to thwart various side-channel attacks on virtualized systems through hardware enabled security measures.

There are two kinds of hypervisors currently in use. One enables para-virtualization, in which a guest OS runs on top of an existing OS and does not depend on support from the underlying architecture [8]. Second, is the hardware-enabled hypervisor that runs on bare-metal hardware and makes use of the hardware extensions provided by hardware vendors like Intel and AMD. We envision a hardware-enable hypervisor system that utilizes security measures implemented in hardware (as proposed by this project) to protect the

Virtual Machines (VMs) from side-channel attacks. The proposed FPGA platform allows us to explore hardware-based techniques for thwarting cache-based, cold boot, and power analysis attacks. The hypervisor used to virtualize the hardware must be modified to make use of the proposed security features. Hence we use the open source hypervisor: KVM. This work provides a design for kernel modules that must be inserted in the kernel in order to make use of the hardware extensions. It will be necessary to modify the KVM code to provide access to these hardware extensions through the kernel modules. For example – it can allocate one page of memory from a Block Ram (BRAM) on the FPGA that is designed to store sensitive data, to one of the virtual machines that indicates the need to protect its data.

The prototype system used in this research is Xilinx's Virtex-5 FXT platform [7] that has an embedded PowerPC-440 processor. This embedded processor is capable of running a Virtual Machine Monitor (Hypervisor) such as the KVM. For our prototype designs, we port a Linux Kernel to the PowerPC and include custom Kernel modules to control the peripheral hardware instantiated in the FPGA fabric and connected the PowerPC.

The remainder of the thesis is organized as follows - Chapter 2 presents an overview of some known side-channel attacks and the proposed counter solutions. Chapter 3 gives an overview of the system used for prototyping and along with Appendix A, serves as a guide for setting up the development system. In Chapter 4 we present one of the proposed solutions to counter cache-based side channel attacks. In Chapter 5, we look at the results of the conducted tests and how they fit into the ultimate goal of hardware enabled security. Chapter 6 offers conclusions and suggestions for potential future work.

Chapter 2

Background

Side-channel attacks exploit information gained from monitoring a system’s physical environment or the physical properties of the system hardware. Although the mathematics of cryptographic algorithms is analyzed minutely to avoid any information leakage, side-channel attacks exploit implementation details of the protocol that inadvertently leak information about keys and/or cleartext data. Well known side-channels include timing, power consumption, electromagnetic emanations (tempest) and cold boot attacks. Recent research reveals how to maneuver Virtual Machines (VMs) in cloud computing onto the same server as a potential victim, where, a side-channel can be exploited to examine the cache and determine when a keyboard is being used [9]. In this chapter, we provide an overview of the side-channel attacks that we aim to thwart. A review of the solutions proposed by other researchers is also provided followed by a discussion of how our design compares to and differs from some of the existing solutions.

2.1 Cache-based side-channel attacks

Percival et al. [1], Bernstein et al. [2] and Osvik et al. [10] demonstrated the use of cache-based side-channel attacks. These attacks exploit the fact that hardware resources, like caches, are shared between concurrent processes running on the same processor. In [1], Percival et al. demonstrated an attack against the RSA algorithm using concurrent threads running on the same processor. The attacker thread runs concurrently with the victim thread executing the RSA algorithm. The attacker sequentially loads all the cache lines by repeated access to fixed memory locations. During the execution of the RSA algorithm, the victim processes will evict some of the cache lines. The attacker measures the time required for each access and can infer a cache miss based on these delays. This attack is analyzed briefly in [11] and following is an excerpt:

> *"The core operation used in RSA is modulo exponentiation. It is often implemented with a series of squarings and multiplications. The encryption key is also divided into a series of segments. For each multiplication, a multiplier is selected from a set of pre-computed constants stored in a table. During the table lookup, a segment of the encryption key is used to index the table. As the table is stored in memory, the attacker can detect the cache evictions caused by the victim (the encrypting process) for the table lookup. Based on which line is evicted, the attacker can infer which table entry is accessed. This tells the attacker the index used for this table lookup, which is a segment of the encryption key."*

In [2], Bernstein performed an attack to obtain the AES algorithm's encryption key. In this scenario, the attacking thread might not be executing on the same machine as the victim thread that implements the encryption algorithm. The attacking thread only monitors the total execution time of the encryption process. This attack is performed in three steps - first, the attacker monitors the timing for encryption of plain-text data with a known key. In the second step, the attacker uses an unknown key and monitors the encryption time on plain-text data with this key. Lastly, using a correlation algorithm that analyzes the timing corresponding to cache misses, the AES encryption key is obtained. In both of these attacks described above, the main source of information leakage was cache interference.

In [10] Osvik et al. present two types of cache-based attacks – asynchronous and synchronous. In synchronous attacks, the attacker process runs concurrently with the victim process on the same hardware and shares resources with the victim process. For example, the authors have executed an attack where a user who has write permissions to any file on a file system can trigger encryption of a known plain-text and using the authors' attack techniques (monitoring the cache line eviction times) the AES encryption keys are deciphered. In asynchronous attacks, the interactions between the attacker process and victim process are significantly different. The attacker process only monitors patterns of memory accesses for its program execution and assumes a non-uniform distribution of plain-texts or cipher-texts rather than their specific values. The authors claim that this form of attack is more constrained in the hardware and software platforms to which it applies and is very effective on certain platforms with simultaneous multithreading.

2.2 Power analysis side-channel attacks

Power analysis side-channel attacks use variations in power consumption to distinguish different operations. The power consumption of an operation depends on the inputs. Different operand values cause different switching activities in the memory, buses, datapath units (adders, multipliers, logical units) and pipeline registers of processors. Among these components, the processor datapath and buses exhibit more data-dependent energy variation than memory components [12]. Power analysis attacks have varying degrees of sophistication. Simple Power Analysis (SPA) [3] uses only a single power consumption trace for an operation. From this power trace, an attacker can identify the operations performed (e.g. whether or not a branch at point *p* is taken, or if an exponentiation operation is performed). Combining power consumption information with knowledge of the underlying algorithm can be used to reveal the secret key. Differential Power Analysis (DPA) is a common higher-order power analysis approach that uses power profiles from several runs and the data-dependent power consumption variation to break the key [13].

2.3 Cold-boot attacks

Cold-boot attacks [5] are a different type of side channel attack in which the attacker needs to have physical access to the victim's machine. These attacks exploit the fact that DRAMs are used for storing data during runtime and retain information for a sufficient amount of time even after power is turned off. The attacker can press the reset button on the computer without prior warning and subsequently boot, using a custom bootloader, to dump the entire contents of the RAM into an external storage. Or, the attacker can physically remove the RAM from the victim's computer and access its contents on some other machine. Most of the commodity disk encryption softwares store the encryption keys on the RAM. If the attacker gains access to such a system even in a suspended state, he can obtain all the contents of the RAM and identify the encryption keys from the data obtained.

2.4 Existing Solutions

Researchers have suggested various solutions to counteract side-channel attacks. In [11], Wang et al. propose hardware and software solutions to thwart software-based side-channel attacks that observe cache access patterns. Their first solution, called Partition Locked Cache (PL Cache), needs modification of the cache structure by adding two new fields – the Locked bit (L) and an ID field, to each cache line. Also, in order to access the security feature provided by these two fields, they suggest extensions to the existing ISA in the form of commands to set and reset the L bit and to make use of the ID field. Their second solution, called Random Permutation Cache (RP Cache), randomizes the cache interference between concurrent processes so that no useful information can be inferred about the evicted cache line. Like the first solution, even this requires modification to the cache structure and the ISA. Though this work presented the solution designs in great detail, the implementations are at present possible only in simulation. It is difficult to make modifications to the cache structure and the ISA of a processor without aid from the vendor.

Similarly, [10] and [14] present design modifications to defend against side-channel attacks on AES encryptions. [10] proposes multiple solutions that are only theoretical. Some of the suggested solutions are alternative lookup tables, application specific algorithmic masking, dynamic table storage, etc. The authors also briefly list cache-based security measures similar to our work. However, their work focuses more on designing attacks than their mitigation and the authors offer no implementation details about the mitigation techniques. In [14], Brickell et al. propose software solutions aimed to mitigating only the type of attack executed by Bernstein [2]. Their first solution suggests using compact substitution box (S-Box) tables available on modern processors. Their second solution suggests randomizing the use of these S-Boxes. Their third solution suggests preloading the relevant cache lines. These solutions are too attack specific and may be overcome in the near future by simply modifying a few aspects of the attack. On the other hand, the solutions we propose target the cause of cache-based side-channel attacks and are not attack specific.

Besides these processor directed efforts, solutions have been proposed to make FPGAs resistant against side-channel attacks in [15]. This work focuses on how implementations of custom hardware on an FPGA can be made secure against information leakage and not how information leakage between two concurrent processes can be avoided. Although they propose security measures for systems using FPGAs, their solutions are not applicable to general-purpose processors that are commonly used as hardware for VMs.

Our work significantly differs from this as the FPGA in our system is used to implement the security measures rather than some custom hardware.

From the attacks studied above, we see there are multiple ways to execute side-channel attacks if the attacker process has sufficient knowledge of the hardware resources shared with the victim process. More often than not, the hardware that is virtualized is a general-purpose processor whose characteristics are well known. Hence side-channel attacks assume certain underlying procedures that conventional processors perform during the execution of code. The most common assumptions are the caching of data during execution and prior knowledge of the processor architecture. Although it is a known fact that data is leaked through side-channels, inclusion of defensive features in processor design and coding practices are often dictated by conventional attacks because it is not always possible to change the underlying architecture while designing a system with commodity processors. Also, there is more than one way of monitoring the status of a system with regards to its caching of data or consumption of power.

We believe that designing a system against such unconventional attacks demands exploration of new avenues. Therefore, in this project, we explore a new dimension in securing virtual machines using the reconfigurability of FPGAs. We propose changes to the memory structure regarding how and where the encryption data (encryption keys and lookup tables) should be stored in order to avoid cache based side-channel attacks and cold-boot attacks. We also propose solutions to avoid side-channel attacks based on monitoring power consumption, using the partial reconfiguration capabilities available in modern FPGAs.

Conclusion

In this chapter, we reviewed the three most prevalent side-channel attacks – cache-based side-channel attacks, cold-boot attacks and power analysis attacks. We also reviewed existing solutions proposed by other researchers and how they compare to our proposed design. In the next chapter, we provide an overview of the prototyping platform that is used in this project to implement hardware solutions for the side-channel attacks discussed above.

Chapter 3

System Setup

In this chapter, we provide an overview of the system starting with the hardware and software tools used for development. Then we walk through the important steps for setting up the target system comprised of a Linux kernel running on an embedded PowerPC440 processor.

3.1 Kernel-based Virtual Machine (KVM) and QEMU

KVM is an open source Hypervisor (Virtual Machine Manager) developed by Avi Kivity et al. [326]. Using KVM, multiple VMs can be run simultaneously on a Linux system. On x86 processors, KVM makes use of virtualization hardware extensions that are provided by hardware vendors like Intel [27] and AMD [28]. Recently, developers of KVM have also added support for running KVM on embedded PowerPC processors including the PowerPC-440 architecture available on our target FPGA. The VMs launched with KVM are normal Linux processes, which enables us to avail of existing Linux functionality of process management to manage these VMs. The guest OS, which is a VM created by opening a device node (/dev/kvm), has its own memory separate from the user space that created it. We plan to make use of this feature for isolation of multiple VMs.

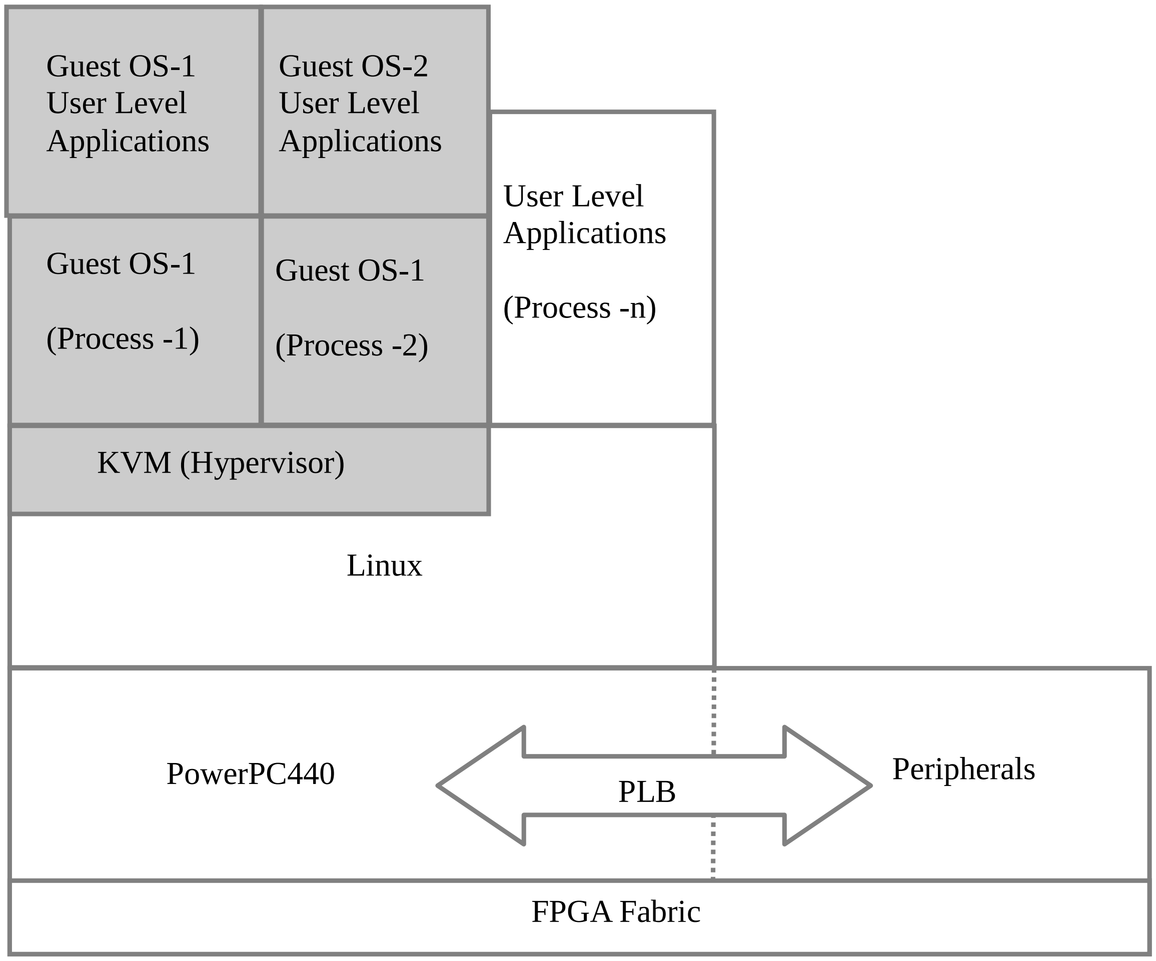

Figure 3.1 : KVM on PowerPC440 embedded FPGA

In addition to the kernel being configured for virtualization, the host kernel also needs an executable of QEMU [29]. QEMU is a userspace emulator that is widely used for emulating a wide range of processors on other host processors. KVM however uses

QEMU for hardware I/O emulation. The QEMU executable for the PowerPC can first be cross-compiled on the development station (Host x86 PC) and then transferred to the host kernel on the board. A guest kernel image must also be loaded into the host kernel. Using this guest kernel image, QEMU then launches the guest kernel as the VM.

Currently, only kernels built for PowerPC-440 are supported as guests on the host PowerPC-440 kernels. Also, since the PowerPC-440 does not come with hardware extensions for virtualizations like currently available general-purpose processors (GPPs), virtualization is achieved by 'trap and emulate' of certain instructions that are executed by the guests [16]. As embedded virtualization becomes more prevalent, we expect to see hardware extensions similar to those available for GPPs being provided on embedded processors.

3.2 Field Programmable Gate Array (FPGA)

An FGPA is an SRAM, flash or antifuse-based device that can be configured using bitstreams generated from hardware description languages (HDLs). FPGAs can be customized to implement any logic in hardware and are often used as an alternative to application specific integrated circuits (ASICs) due to their configurability and reduced time to market. Modern FPGAs are also capable of partial-reconfiguration, where part of the FPGA fabric can be modified during runtime while the remaining part remains functional.

3.3 Pico – E17 development board

The platform selection for the hardware-enabled hypervisor was dependent on a number of factors. First, to implement the hardware-based security features, an FPGA with an embedded processor was desired. A number of FPGA devices fall into this category including currently available FPGAs from Xilinx and Altera, the two most prominent names in FPGA development. All of these FPGAs are capable of instantiating a soft-core processor; namely Microblaze on Xilinx devices and Nios on Altera. However, these are considered low-end processors with minimum capabilities. Xilinx also has a series of high-end FPGAs that have a hard-core PowerPC processor embedded in the FPGA fabric. The PowerPC architecture is widely used as an embedded processor and designs developed on this platform can be generalized and adopted in other mainstream processors. Xilinx provides two versions of the PowerPC on different FPGAs – The PowerPC405 and the PowerPC440. The PowerPC440 was selected since the target hypervisor – Kernel-based Virtual Machine (KVM) supports the PowerPC440 architecture [16]. With these constraints in mind it was determined that the E-17 platform from Pico Computing [30], which is based on a Virtex-5 FXT FPGA that contains a PowerPC440 core connected to 256MB of DDR2 RAM, would meet the requirements for running KVM. The E-17 is a highly compact board designed for use with laptops. It interfaces via the PCI slot on the laptop and the FPGA can be accessed through the PicoUtil software [17]. However, this software is available only for the Windows operating system. As our development platform was Linux, the PCI connection was used only as a power supply and the FPGA was accessed through an extension board that has a

JTAG connection, a serial port and an Ethernet port. More details about the board can be obtained from [30].

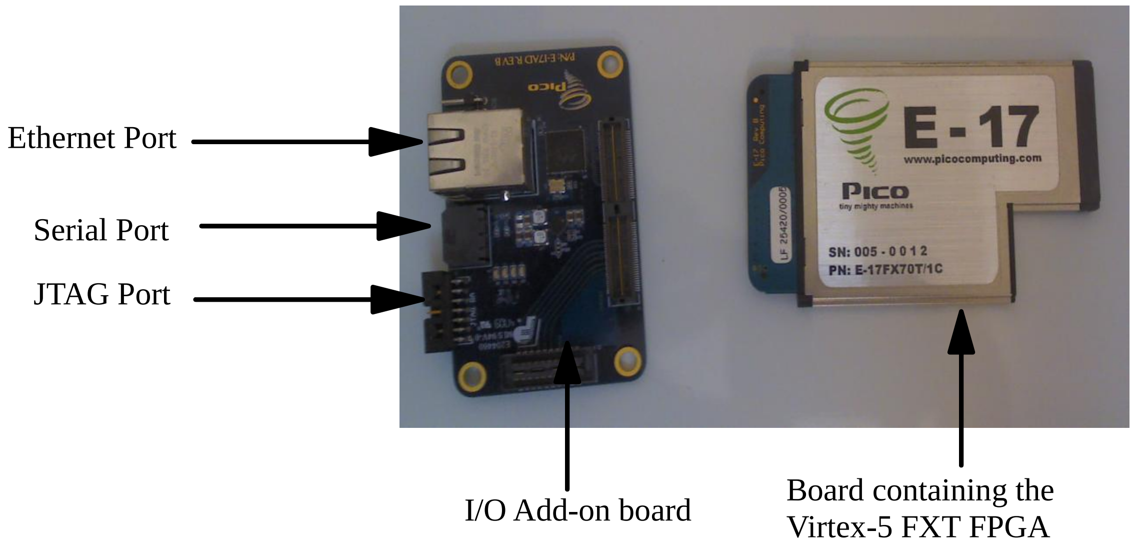

Figure 3.2: Pico E-17 development board and I/O Add-on board

3.4 PowerPC440 embedded processor

The PowerPC-440 embedded processor on the Virtex-5 FPGA is a 32-bit Fixed-Point subset of IBM's Book-E: Enhanced PowerPC architecture [18]. It is a superscalar RISC architecture with support for 32, 64 and 128-bit Processor Local Bus (PLB). It is highly customizable as various peripherals can be attached to the PLB. It provides interfaces for custom co-processors and floating-point functions as well as separate 32KB instruction and 32KB data caches. The processor runs at a speed of 400 MHz while the peripherals attached to the bus operate at a speed of 100 MHz. One of the basic requirements for running a proper Linux Kernel on a processor is the Memory Management Unit (MMU). The MMU enables the use of Virtual Addressing by translation of physical addresses to virtual addresses. User-level Processes use virtual addresses in their operation and are seldom aware of the physical addresses of the memory locations that they access. The PowerPC-440 that is available on the FPGA is equipped with a Memory Management Unit (MMU) to support Virtual Addressing thereby enabling the operation of a rich Linux kernel. The processor has a TLB implemented entirely in software, which enables complete control over its operation [19].

3.5 Xilinx Embedded Development Kit (EDK)

Xilinx's Integrated Development Environment (IDE) for an embedded system consists of the Xilinx ISE [20] and the Embedded Development Kit (EDK) [20]. Most of the work in this project is performed using the EDK since it provides all the functionality needed to develop the hardware as well as software for an embedded system using either a hard-core or a soft-core processor. New versions of the software tools are frequently released by Xilinx and the decision to choose a particular version should be made based on the support available for the target hardware. For developing our system, we have used version 12.2 of Xilinx’s IDE installed on the Linux-based operating system Ubuntu 9.10.

3.6 Cable Drivers Installation

To communicate with the FPGA and the embedded processor, Xilinx provides a JTAG connection to the board through its Platform USB Cable. On an Ubuntu Linux development station, the drivers provided with the ISE and EDK installation do not suffice to enable use of the USB cable and it is necessary to download the USB drivers package available on Xilinx's website. Also, in Linux it is necessary to perform all installations with root privileges. Once the drivers are correctly installed, the Platform Cable box will illuminate a green LED confirming proper connection to the computer’s USB port, if an orange LED is present, a physical connection has been made but it does not indicate a communication link has been established.

3.7 Project setup in EDK

To set up a project in the EDK, it is advisable to follow a few sample tutorials available on the Xilinx website. Please refer to Appendix A.1 for detailed instructions to set up a basic system on the Pico E-17 board.

3.8 Generate the Device Tree

The device tree compiler generates a Device Tree Source (.dts) file [21] that has information about the hardware setup of the board. This information is converted from text form into a compact flattened-tree representation of the system's hardware based on the device tree supplied by Open Firmware. To generate the .dts file, the device tree compiler can be downloaded from Xilinx's wiki. The folder named 'bsp' contains the device tree compiler and should be placed in the project's parent directory. Next, perform the following steps to generate the .dts file:

1. Go to Software → Software Platform Settings

6. Select device-tree as the OS.
7. In the OS and Libraries tab, enter the name of your serial console device (e.g. RS232) and type 'console=ttyUL0' in bootargs.
8. Now select Software → Generate Libraries and BSPs.

This will generate a *xilinx.dts* file in the <project folder>/ppc440/libsrc/device-tree/ folder. Place this file in <Linux>/arch/powerpc/boot/dts/ folder of the source code and rename it to *virtex440.dts*.

3.9 Serial Connection

During the system configuration in EDK, we set the console output as an RS232 connection allowing the serial output from the board to be connected to the serial port of the computer. On the prototype platform, we use a serial-to-USB converter and connect the serial port of the board to the USB port of the development station. To capture the input obtained on the USB port and display it meaningfully on the screen, a terminal application such as 'Kermit' [22] is used. On a Windows development platform, the hyperterminal can be used and its parameters are set to those that were used during the system build in the EDK. To start the Kermit application, at the command prompt execute:

```
$ ./kermit
```

Make the connection between the board and the USB port. Obtain the device name that appears in /dev directory on making the connection. Assuming the entry in the /dev directory for the serial connection is 'ttyUSB0' (Tele-type USB0), input the following parameters:

```
> set line /dev/ttyUSB0
> set speed 9600
> set carrier-watch off
> set parity none
> set flow-control none
> set handshake none
> robust
> connect
```

This will establish the serial connection between the board and the computer and the terminal window running kermit will act as the terminal window for the Linux system running on the board.

3.10 Cross-Compilation

Since the development platform is a Linux system running on an x86 processor and the target platform is an embedded PowerPC440 processor, a cross-compilation tool-chain to allow generation of binaries on the x86 that can be executed on the PowerPC is needed. The Embedded Linux Development Kit (ELDK) [23] is a well-established cross-compilation platform and has been chosen for this purpose. To set up the tool-chain, download the ELDK source files from [23] and follow the installation instructions on the webpage.

The executables required for cross-compilation are located in the folder <ELDK>/usr/bin.

To make use of these executables, this folder must be included in the PATH environment variable by typing the following at the command line:

```
$ export PATH= <ELDK parent folder>/usr/bin:$PATH
```

3.11 Linux Kernel – Build and Port

The Linux kernel used is Xilinx's version of the Linux kernel 2.6.34 and can be obtained from Xilinx's wiki [24]. This kernel image comes preloaded with the KVM source since KVM was merged with the Linux kernel in version 2.6.20. However the kernel must be customized for use on the PowerPC440 by selecting and deselecting kernel features and setting up the environment variables. Please refer to Appendix A.4 for instructions on configuring the Linux Kernel and environment variables.

To mount a file system when the Kernel boots, an Initial Ram Disk (initrd) image must be included as part of the kernel. A prebuilt image is available from Xilinx's wiki. Place this image 'ramdisk.image.gz' in the <Linux>/arch/powerpc/boot/ folder.

To build the image, type at the command line in the top level directory of your Linux source :

```
$ make simpleImage.initrd.virtex440-mine
```

In the above command, the .initrd string instructs the build process to include the initial ram disk as part of the kernel image. The 'virtex440' string points towards the device tree file that was included earlier and information from that is used during compilation. This will generate an image in the Executable and Linkable Format (.elf) in the <Linux>/arch/powerpc/boot folder.

This image can be downloaded onto the board using the 'dow' command on the Xilinx Microprocessor Debugger (xmd). Launch xmd through EDK by selecting Debug → Debug Using XMD. Navigate to <Linux>/arch/powerpc/boot directory in xmd and type the following at the command prompt of xmd :

```
$ dow simpleImage.initrd.virtex440-mine.elf
```

This will load the kernel image to the RAM and the Kernel can be seen booting on Kermit's terminal that receives data from the serial port.

3.12 Busybox

Busybox [25] is called the 'swiss army knife' for embedded Linux. It is a multi utility binary that is customizable by the user. Busybox generates a single binary executable that can perform the command line operations selected during its configuration. It also generates executables with the names of the command line operators; all of which have symbolic links to the busybox executable. It is an extremely useful tool to have in an embedded system since it saves a lot of space.

Our Initial Ram Disk image contains a build of busybox with most of the command line operations needed for a basic system to run. For example: ls, vi, grep, ifconfig, etc.

The utility needed to connect the board to the network is 'udhcpc', which stands for 'micro dhcp client'. When the board is connected to an Ethernet port using an Ethernet cable, it must obtain a lease to get an ip address and be connected to the network. 'udhcpc' is used to obtain the lease by negotiating with the DHCP server. To achieve this, udhcpc runs a predefined script called 'default.script' that is placed in /usr/share/udhcpc/ folder in the file system (initrd). To execute the DHCP client, at the command prompt type:

```
$ udhcpc
```

After obtaining a lease from the DHCP server, we are connected to the network and can receive files using 'wget', a command that is also built as part of the busybox binary.

Conclusion

In this chapter, an overview of the development system was provided including the important steps for setting up the target system. Now that we have a Linux kernel running on the PowerPC, we delve into implementation of the designs proposed for thwarting side-channel attacks. The next chapter will explain how the BRAMs can be made non-cacheable using a kernel module. These BRAMs can be used to store sensitive data that will not be moved to the cache and hence avoid monitoring by a snooping process. This technique is also a measure against basic cold-boot attacks.

Chapter 4

Implementations

This chapter provides the details for implementing a BRAM that is mapped to a specific user process and is also marked non-cacheable as a measure against cache-based side-channel attacks as well as basic cold-boot attacks. The procedure will first map a page of memory to a user process using the mmap() system call and then extend the idea by using the kernel module to mark that page as non-cacheable. The latter part of the chapter provides an overview of our attempts to set up a prototyping system for partial-reconfiguration at runtime and to virtualize the PowerPC440 processor using KVM on the selected platform.

4.1 Implementation of Non-Cacheable BRAMs

This section explains how the BRAMs in the system are allocated and their properties modified through a kernel module.

4.1.1 BRAM on FPGA fabric

The purpose of using an FPGA-based system for prototyping was the ability to design custom hardware around a processor in the FPGA fabric. Through Xilinx's EDK, it is possible to add a variety of hardware components to an existing base system consisting of the PowerPC440 processor. As shown in Figure 4.1, the system has three Block RAM (BRAM) areas in the FPGA fabric. In Figure 4.2, we see how the BRAMs are connected to the Processor Local Bus (PLB) of the PowerPC through a BRAM controller for each BRAM.

The BRAMs are not part of the main memory that is mapped by the Linux Kernel during bootup. However, since it is connected to the PLB and its physical address is known, it can be mapped to a virtual address in the user space, making it available to a user process like any other memory region. To map one page (4KB) of memory from any one of the BRAMs into a user process, we can execute a user-level application that will use the system call *mmap()* and pass as an argument the physical address of the BRAM that we want to access. While the physical address being available at the user level is acceptable during development, it is not advisable in production code. Hence, in our implementation, the user does not have the liberty to choose which physical address he wants to have access to. The mapping of the page from BRAM is done by our kernel module and is beyond the purview of the user level code. A pointer to such a page is returned by the *mmap()* system call.

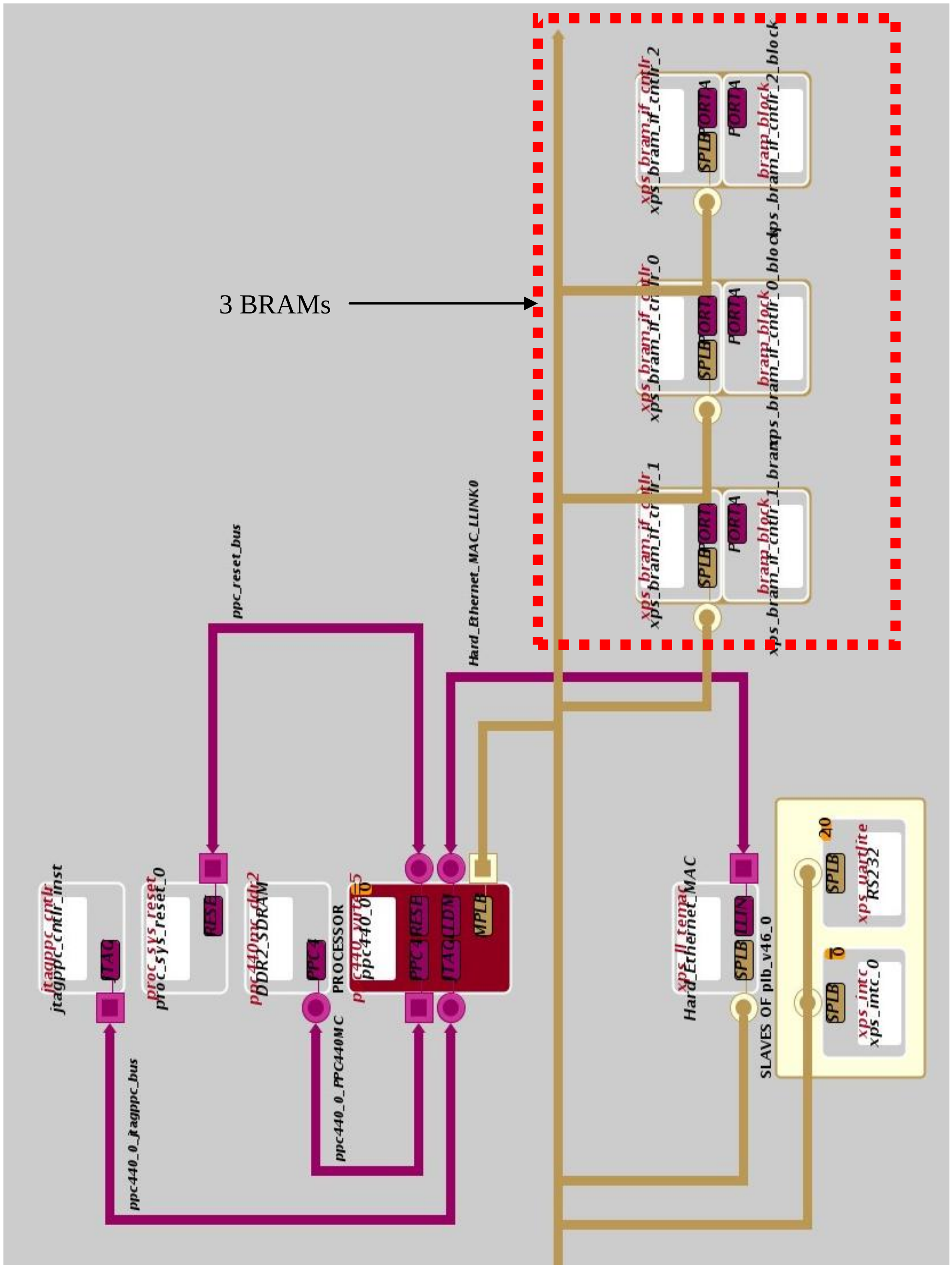

Figure 4.1: Block Diagram of the entire system

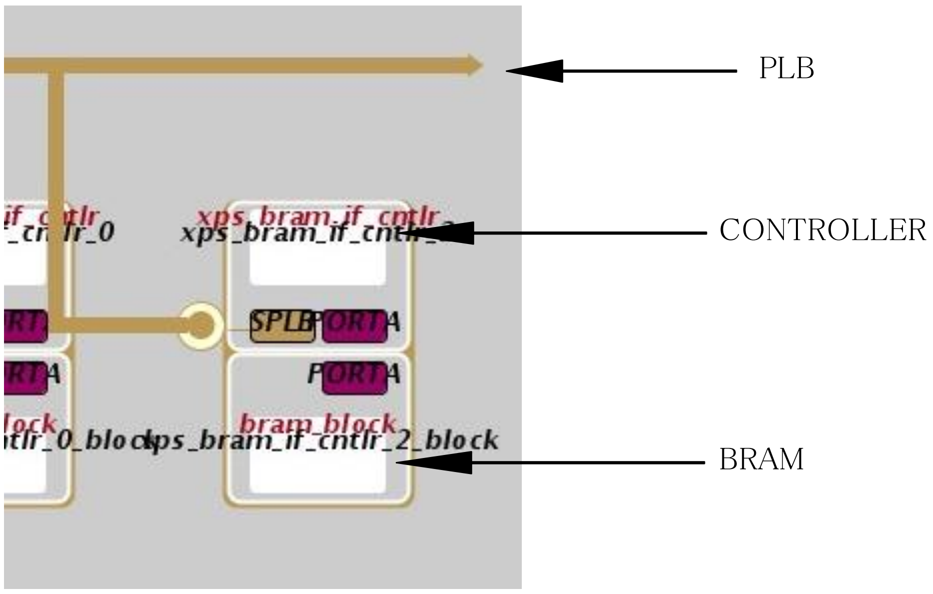

Figure 4.2: BRAM connected to PLB through BRAM controller

```
bram_address = mmap(0, 4096, PROT_READ|PROT_WRITE, MAP_SHARED, memfd, 0);
```

In the above example of the *mmap()* system call, 4096 is the size of the requested memory, *PROT_READ* and *PROT_WRITE* are protection flags indicating that data can be read from and written to this memory region. *memfd* is the file descriptor of the device to which this page of memory is mapped.

For example,

```
memfd = open("/dev/mem", O_RDWR);
```

will map the BRAM's page through the generic *mem* device that is listed in */dev* directory. When *mmap()* is called using this file descriptor, it will call the corresponding kernel level *mmap()* routine written in that device's driver. We will delve deeper into this procedure when we discuss our custom driver for the BRAM. The *mmap()* call will return a pointer to the starting address of that page and we can use that page of memory area like any other memory allocation. This page will be confined to the process's address space and no other process will be able to access it. Sample user code is included as Appendix C.

4.1.2 Making the page non-cacheable

Although the page obtained is secured from illegal direct accesses, there are a few issues associated with the above implementation that do not fit our requirements. First, since the techniques developed through this project are targeted for use on a hypervisor that will have other operating systems as user level processes, it is not feasible and certainly not recommended for the physical addresses of BRAMs to be known at the user level. Second, security from illegal accesses of other processes is not necessarily security from side-channel attacks. As we have previously discussed, software cache-based side-channel attacks monitor cache access patterns of victim processes. Since developing a measure against this type of attack is desired, we go a step further and create our own device driver (Appendix B) to map the BRAMs into user space with the condition that data accessed from the page allocated in the BRAM will not be cached. First, we create a device named BRAM in the */dev* directory of the Initial Ram Disk (initrd) image and mark it as a character device with MAJOR number 50 and MINOR number 9.

For this, we first unzip and mount the ramdisk.image.gz to a suitable location and then in its */dev* directory type:

```
$ mknod /dev/BRAM c 50 9
```

This will create a device named BRAM in the */dev* directory. When we are done building our kernel module, we will transfer that into initrd and not build it as part of the kernel image. We plan to insert the device driver at runtime using the *insmod* utility that is built as part of the busybox binary. To build the kernel module, a Makefile is required with the same modifications to the environment variables as were made at the time of building the Linux kernel for the PowerPC architecture. Namely,

```
$ export ARCH=powerpc
$ export CROSS_COMPILE=ppc_4xx-
$ export PATH=<Path to ELDK installation folder>/usr/bin:$PATH
```

Executing the Makefile with *make* command will create a kernel module with extension .ko. We transfer this kernel module to the initrd so that it will be available on the board after bootup. To insert the kernel module, at the command prompt for the PowerPC system type:

```
$ insmod <module_name>.ko
```

After inserting the kernel module, the function *bram_hello()* is executed as it is registered to be the module initialization function. It registers the driver for the device we intended through MAJOR and MINOR numbers, which are 50 and 9 respectively.

The device driver for any device has a predefined set of functions that must be provided.

These functions are listed in the *file_operations* structure and are essentially mappings of user space system calls. The type and number of functions provided by the driver depends on the functionality required by the user. Our file_operations structure lists support for the following system calls on this device:

1. *.open = bram_open()*
2. *.release = bram_release()*
3. *.mmap = bram_mmap()*
4. *.ioctl = bram_ioctl()*

This structure is registered with the driver in the initialization function through the *cdev_init()* call. When the user code executes the system call to open our device, control will be transferred to the *bram_open()* routine of our device driver and the user code will obtain the file descriptor for the device.

As mentioned earlier, our system contains three BRAM blocks. For testing purposes, our user code will make four *mmap()* calls that will be mapped to *bram_mmap()* routines in our driver code. Each *mmap()* call will return a pointer to one page of memory allocated from the three BRAMs and one from the DDR2 SDRAM so that the user code will have four pointers, one to each of the four physically different memories of our system. As a test, we loop through all the memory locations reading the data stored at every address on the page multiple times in order to cache these memory locations. We perform this test on all four pages obtained from all four memories. We record the time required by each loop and store these as reference times for accesses to memory locations that are cached. Based on the access times, it can be concluded whether or not the memory location was cached since a cached memory location will require less access time. This test and the timings will be discussed in detail in the next chapter.

The user-level *mmap()* call has the form:

```
bram_address = mmap(0, 4096, PROT_READ|PROT_WRITE, MAP_SHARED, memfd, bram_base);
```

While the kernel-level *bram_mmap()* routine has just two parameters -

```
int bram_mmap(struct file *fp, struct vm_area_struct *vma)
```

Of these two parameters, we are interested in *struct vm_area_struct *vma*. Every memory region mapped by the Linux kernel has an associated *vm_area_struct* structure that keeps a record of all the properties of that memory region. The various arguments that we pass through the user level *mmap()* call are internally assigned to different members of the *vm_area_struct* structure before it is used in the *bram_mmap()* routine of our device driver. The various flags associated with a memory region are set in the *vm_page_prot* member of the *vm_area_struct* structure. The *PROT_READ* and *PROT_WRITE* flags that we pass as arguments in the user level *mmap()* call, are also recorded in *vm_page_prot*. One of these flags also marks the memory region as cacheable or non-cacheable. The memory region can be marked as non-cacheable by setting this flag through the *pgprot_noncached()* routine as follows :

```
vma->vm_page_prot= pgprot_noncached(vma->vm_page_prot);
```

It takes in the *vm_page_prot* member as its argument, modifies it by marking it as non-cacheable and returns the modified value back to the *vm_area_struct* structure.

We perform this operation for one of the BRAMs during its *mmap()* call so that one of the four pointers returned to the user code points to a page that is marked as non-cacheable. For subsequent mmap() calls, we allocate a page from the remaining BRAMs and the DDR2 RAM and mark them as non-cacheable in a similar fashion.

4.2 KVM on PowerPC440

The goal of this project is to provide security against side-channel attacks on virtualized hardware. As mentioned earlier, KVM is the selected hypervisor since it is open source and can be modified according to our needs.

As mentioned in Chapter 3, KVM was included in the Linux kernel source code since version 2.6.20 (the kernel source used for this project is 2.6.34). To run KVM on the PowerPC, the configuration file of the kernel source is modified to enable virtualization according to the directions given in Chapter 3 and the kernel is recompiled. To verify whether or not the recompiled kernel supports KVM, we can observe the messages printed on the console during boot-up and shutdown. They should contain messages confirming startup and/or shutdown of KVM.

To start a virtual machine on the PowerPC using KVM, an executable of QEMU is required for I/O emulation. This executable is built from the source code of QEMU obtained from its website. This source code is cross-compiled on the x86 development station and then transferred to the PowerPC by including it in the initrd image. To build this cross-compiled image, installed copies of ‘libfdt’ and ‘zlib’ that are built for the PowerPC are needed. In other words, the libraries must also be cross-compiled during their installation. To obtain an executable of QEMU, set the environment variables:

```
$export cross = ppc_4xx-

$export PATH=<ELDK>/usr/bin:$PATH
```

Then execute the following command:

```
$ ./configure --disable-sdl  --target-list=ppcemb-softmmu \
  --cross-prefix="$cross" \ ; make
```

This will create an executable named qemu-system-ppcemb in the <QEMU>/ppcemb-softmmu folder. This executable is transferred to the PowerPC.

In addition to the QEMU executable, a device tree blob (.dtb) file is also transferred to the PowerPC. This file gives QEMU information about the hardware of the system. This file can be obtained from the .dts file that was earlier included in the kernel source during its compilation. The device tree compiler converts the .dts file to .dtb file when executed in the following way:

```
$ ./dtc –O dtb –o virtex440.dtb virtex440.dts
```

As of now, we have not run a virtual machine on the embedded processor for multiple reasons. First, the KVM port for the PowerPC440 is in a nascent state and has only been tested on one platform (Bamboo Board). As a result, it is not clear what library dependencies must be satisfied before the QEMU executable can run on our board's embedded processor. Second, the Pico E-17 development board is a compact board and is highly resource constrained with only 256MB of RAM. There is also no provision for attaching external memory to the board. This prevents a static image of the QEMU executable from being loaded on the board's RAM.

4.3 Partial reconfiguration at runtime

In earlier chapters, we briefly discussed that side-channel attacks that monitor power consumption of a system rely on the hardware being static. In a system with static hardware, each operation will have a unique footprint of the power consumed for a particular input. The attacker process monitors the power consumed by the system during its runtime and by correlating it with power consumption traces previously obtained, deciphers the encryption keys.

We propose to thwart this kind of attack by introducing variations in the power consumption traces. To achieve this, new hardware is connected to the processor's bus during runtime and used as a substitute for an operation in the encryption algorithm. This hardware will have a completely different power consumption footprint from that of the same operation performed on conventional static hardware. This deviation will confuse the attacker preventing correlation to previous traces for the same operation. We believe this will prove a significant hurdle for the attacker attempting to implement power-based side-channel attacks.

Although this project does not aim to provide a completed design using this technique, we attempted to set up a basic prototyping platform for further research on these topics. Xilinx's partial reconfiguration guide [31] serves as a good starting point for understanding the details of a partially reconfigurable system.

Xilinx’s design flow for a partially reconfigurable system is a combination of multiple project files combined to form a complete system. Apart from Xilinx’s ISE and EDK softwares, this system needs an installation of PlanAhead for floorplanning the design, which includes defining a reconfigurable partition for the reconfigurable region. Using this tool, multiple configurations can be designed for loading at different times. This tool is also needed for generation of full and partial bitstreams for the system. Once a partially reconfigurable partition is defined, custom logic blocks can be designed to mimic one or more of the operations in the encryption algorithm (e.g. adder, multiplier, multiply and accumulate unit, etc.).

After the full bitstream for the static system and partial bitstream for the reconfigurable partition have been generated, the full bit stream can be transferred to the FPGA and the static system designed using the EDK, consisting of a PowerPC processor can be functional. The Xilinx documentation [31], [32] instructs that partial bitstreams be stored on a Compact Flash (CF) card and the card be inserted into the CF slot on the board. The partially-reconfigurable system is concieved such that the partial bitstreams are loaded from the CF card to the Internal Configuration Access Port (ICAP) that is connected to the PowerPC’s PLB. The design in the partial bitstream is then configured on the FPGA fabric in the partition defined earlier through the PlanAhead tool.

Our prototyping platform is extremely compact and does not have a CF card slot, which prevented us from implementing this technique on the prototyping platform. In Chapter 6, in the section on Future Work, we suggest ideas that can be implemented when a board with required peripherals is available.

Conclusion

In this chapter, we presented details of implementing our design to thwart cache-based side-channel attacks using BRAMs that are marked as non-cacheable using a kernel module. We also explain our attempts to virtualize the PowerPC available on the platform as well as set up a prototyping system for partial reconfiguration and the platform constraints that made it difficult to implement them successfully. In the next chapter, we present the results of the tests that were performed to confirm that memory locations on the BRAMs are indeed non-cached.

Chapter 5

Results

In this chapter, we present the tests that were performed to verify the effect of marking different memory regions non-cacheable. We measure the time required to access memory locations that are marked as non-cacheable and compare those to the time required to access memory locations that are cached. Also, we present our ideas on how this would contribute to the overall goal of making a system secure from cache-based and cold-boot side-channel attacks by applying this proof-of-concept to a non-prototype platform virtualized using KVM.

5.1 Test conducted

As discussed in Chapter 3, our prototyping system consists of DDR2 memory and three BRAMs connected to the PLB through custom BRAM controllers. A kernel module is inserted in the Linux kernel at runtime using 'insmod' command and linked to the device 'BRAM' that is created in /dev directory. The device is opened in the user level code and using its file descriptor, one page of memory (4KB) from each of the three BRAMs and the DDR2 RAM is allocated using the 'mmap()' system call. This mmap() call in turn calls the bram_mmap() routine of the kernel module.

To conduct tests on each of the BRAMs and DDR2 RAM, we designed four separate kernel modules each having the capability of marking a page from each memory as non-cacheable. In the user code, we then write an integer value to each of the memory locations in each page obtained from all the four memories. In a loop, we sequentially read from every memory location of a page. We execute this loop 10 times for every page and record the execution time for the entire duration of 10 runs using the function gettimeofday(). The timing for the tests performed for the page from each memory is listed in Table 5.1 below.

Marked As Non-Cacheable / Access Timings	BRAM-1 (usec)	BRAM-2 (usec)	BRAM-3 (usec)	DDR2 RAM (usec)
BRAM-1	**112**	37	37	37
BRAM-2	38	**113**	38	38
BRAM-3	37	37	**113**	37
DDR2 RAM	67	67	67	**180**

Table 5.1: Average access timings of memory regions

As seen from Table 5.1, the regions marked as non-cacheable required significantly more access time compared to the regions that were not marked. This time difference confirms that the memory regions with the Non-Cached flag set in the the vm_page_prot field of the vm_area_struct structure, indeed are not cached by the processor.

We verify the data written to each memory region in two ways. First, we simply write the data, execute the test and read the data to verify it is the same. In a second test, we make use of the JTAG connection provided by Xilinx to the PowerPC processor. Using JTAG, it is possible to directly write to a physical address through XMD without going through an operating system. Initially, data is written to the first few addresses of each BRAM that are to be included in the page mapped from the kernel module. Then, we verify the data read from the user code with what was written through XMD. This also confirms that the pages mapped from the kernel modules are indeed from the BRAMs attached to the PLB.

5.2 Protection from cache-based side-channel attacks

In Chapter 2, we saw that in a cache-based side-channel attack, the attacking process monitored the cache accesses of the victim process when performing the encryption. As a countermeasure, if the encryption data (encryption keys and lookup tables) is stored in a memory region like one of the BRAMs that is marked non-cacheable, cache accesses for obtaining encryption data will not take place and the attacker will not be able to access this data and infer segments of the encryption keys. Although this will incur a performance cost (discussed in Section 5.3), the trade-off between security and performance is something that the developer has to weigh. Additional analysis of the encryption algorithms (beyond the scope of this work) may reveal that only a portion of the key/sensitive data would need to be stored in these non-cached BRAMs, since they are slower, and still provided ample protection against cache-based attacks.

5.3 Performance cost

Amdahl's law [33] is used to predict the improvement/decline in performance after modifying a program. Using Amdahl's law, we can predict the impact on performance of our system when memory locations are marked as non-cached. Amdahl's law states that if a fraction 'p' of a program is modified and it results in a speedup of 's' in that fraction of code execution, the overall speedup of the program can be calculated by:

$$\text{Speedup} = \frac{1}{(1-p)+\frac{p}{s}}$$

From table 5.1, it is seen that after a page is marked as non-cached, the time required to access a memory location on that page increases approximately 3 times. That translates to a negative speedup (slowdown) of 3 times. The fraction p of the program code is variable as it depends on the implementation. Therefore, we can approximate the impact on performance in our system as:

$$\text{Slowdown} = \frac{1}{(1-p)+\frac{p}{3}}$$

Where p is the fraction of code that accesses the non-cached memory.

5.4 Protection from basic Cold Boot Attacks

In Cold boot attacks [5], the attacker retrieves all data using the physical properties of the system's RAM since the data in RAM is not erased immediately after the power is removed. The attacker uses this knowledge to his advantage and retrieves the data by either physically dismounting the RAM chip from the computer or booting the system with his own startup code after a sudden reset. There are techniques [5] by which the attacker can identify the encryption keys from the data in RAM and decrypt all the information thereafter.

If however, the encryption keys are stored on memory regions not in the general purpose RAM, the attacker cannot obtain them using the above mentioned cold boot methods. It will take significantly more effort to retrieve data stored in BRAMs on the FPGA fabric. Thus, storing sensitive data on the FPGA's BRAM, either cacheable or non-cacheable, is a good protection against basic cold boot attacks.

Conclusion

In this chapter we presented the result of the test conducted to measure the time required to access data from memories marked as cacheable or non-cacheable. This data enables us to assume that the processor does truly not cache pages marked as non-cacheable in the kernel module. We also present methods that can make use of this feature to avoid data leakage to cache-based side-channel attacks and cold boot attacks. In the next chapter, we will see how the reconfigurability of the FPGA can be further exploited by designing systems that can be partially reconfigured at runtime.

Chapter 6

Conclusions and Future Work

In this chapter, we present a summary of the completed work and conclusions based on the results and analysis collected. We also propose and discuss ideas that exploit advanced features of FPGAs and can be implemented for developing defensive designs against other forms of side channel attacks in future work.

6.1 Conclusions

This project proposes and prototypes changes to existing architectures to enforce security measures in systems that can be utilized by hardware-enabled hypervisors. We discussed different types of prevalent side-channel attacks and previous efforts to develop countermeasures against them. The results of this literature review are summarized in Chapter 2. From the literature review, we characterized the different aspects of a side-channel attack and what are the difficulties faced in developing designs to protect against them. The most significant features are that attacks that monitor the cache accesses and power consumption of different operations exploit the inherent functionality of the system and cold boot attacks exploit the physical properties of a computer's RAM. These attacks are difficult to detect prior and during the event and even more challenging to prevent during runtime.

Our work is unique because to our knowledge, this is the first attempt to explore the use of reconfigurable hardware against side-channel attacks. Through this work, we demonstrated proof-of-concepts that can be further developed to provide robust security measures. In Chapter 4, we explain in detail how memory regions on the FPGA can be integrated into a running Linux Kernel and how it can be made non-cacheable so as to protect encryption keys against cache-based side-channel attacks and basic cold boot attacks.

As virtualization of modern processors became common, vendors like Intel [27] and AMD [27] provided hardware extensions that aid virtualization software. We believe that the outcome of this and related work could make a significant contribution toward developing robust security measures in hardware. These solutions can potentially be applied to main-stream architectures by either introduction of processors with access to an onboard FPGA or dedicated hardware on the processor to aid virtualization in the manners presented in this work.

6.2 Future Work

6.2.1 Partial reconfiguration

Reconfigurability is one of the prime motivations for using FPGAs in most cases. Moreover, Xilinx's FPGAs are capable of being partially reconfigured during runtime [31] meaning that part of the FPGA can be reconfigured while the remaining portion of the FPGA is still performing its operations. This flexibility is possible through the FPGA's Internal Configuration Access Port (ICAP). Xilinx's Hardware ICAP (HWICAP)

IP [34] enables the PowerPC to read and write FPGA configuration files through the ICAP. This port can be used to write programs that can add hardware on the PLB during runtime. Support for this feature can be enabled by the configuration option CONFIG_XILINX_HWICAP in the Linux kernel's configuration file. This feature has already been used in a wide range of applications [35], [36] to modify filters and algorithms during runtime. The partial bitstream needed for reconfiguration is stored separately and can be accessed at runtime for configuration.

Side-channel attacks are easier when the hardware is static. The attacker generates patterns of cache accesses, power consumption during multiplier accesses or any other units of the architecture assuming that the hardware unit being used is known. To offset the repetitive patterns generated by frequent accesses of hardware units, new hardware units that perform one or more functions in any encryption algorithm that is being snooped upon can be generated during runtime and can be accessed intermittently. This will confuse the attacker as accesses to these new hardware units will generate different patterns of power consumption and foil attacks that perform power consumption analysis for each operation.

We believe the outcome of this proposed work should provide robust measures against attacks that monitor power consumption of the system during execution of different operations. We also believe that there lies great potential in exploring similar ideas, as the reconfigurability of the FPGA fabric has been under-utilized when it comes to developing security measures with FPGAs. FPGAs have often been used to accelerate compute intensive applications like encryption algorithms [37], so one can argue that accelerating the encryption algorithm used for protection in the first place can compensate for the performance degradation due to the introduction of security features in the data path.

6.2.2 Virtualizing the embedded PowerPC440

In Chapter 1, we introduced this project as an effort to develop designs to protect virtualized platforms against side-channel attacks. In Chapter 3, we presented a an overview of the Kernel-based Virtual Machine (KVM) Hypervisor and QEMU that is needed to run virtual machines on the embedded PowerPC440.

That being said, the work presented here successfully lays the groundwork for further research with this development platform. The next step towards achieving the final goal as described earlier, the KVM source must be modified to access the kernel modules built to use the protected BRAMs in the FPGA. Also, if the design using Partial Reconfiguration suggested in the previous section is implemented, the KVM source must be modified accordingly to access the new hardware during runtime. A combination of these techniques will significantly strengthen the virtualized platform against cache-based, cold-boot and power analysis side-channel attacks. It will also open avenues for further research and help establish a secure platform that can be used to protect users' data in a virtualized environment.

Bibliography

[1] C. Percival, Cache Missing for Fun and Profit, http://www.daemonology.net/papers/htt.pdf

[2] D.J. Bernstein, Cache-timing Attacks on AES, http://cr.yp.to/antiforgery/cachetiming-20050414.pdf

[3] P. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS and other Systems, Advances in Cryptology, Proceedings of Crypto'96, LNCS 1109, N.Koblitz, Ed., Springer-Verlag, 1996, pp.104-113.

[4] P. Kocher, J. Jaffe, and B. Jun. Introduction to Differential Power Analysis and Related Attacks http://www.cryptography.com/public/pdf/DPATechInfo.pdf

[5] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten, Lest We Remember: Cold Boot Attacks on Encryption Keys, Proc. 2008 USENIX Security Symposium.

[6] Partial Reconfiguration of Virtex FPGAs in ISE 12, Xilinx Inc. www.**Xilinx**.com/.../wp374**_Partial_**Reconfig**_Virtex_FPGAs**.pdf

[7] Xilinx Virtex-5 Family Overview, Xilinx Inc. www.**Xilinx**.com/support/documentation/data_sheets/ds100.pdf

[8] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauery, Ian Pratt, Andrew Warfield University of Cambridge Computer Laboratory Cambridge, UK, Xen and the Art of Virtualization, SOSP 2003

[9] Webpage describing the attack: http://www.computerworld.com/s/article/9137507/Researchers_find_a_new_way_to_attack_the_cloud?taxonomyId=17&pageNumber=2

[10] D. A. Osvik, A. Shamir and E. Tromer, Cache attacks and Countermeasures: the Case of AES, Cryptology ePrint Archive, Report 2005/271, 2005.

[11] Zhenghong Wang, Ruby Lee, New Cache Designs for Thwarting Software Cache-based Side Channel Attacks, ISCA 2007

[12] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, The Design and Use of SimplePower: A Cycle-Accurate Energy Estimation Tool. Design Automation Conference, June 2000.

[13] P. Kocher, J. Jaffe, and B. Jun. Introduction to Differential Power Analysis and Related Attacks http://www.cryptography.com/public/pdf/DPATechInfo.pdf

[14] Ernie Brickell, Gary Graunke, Michael Neve, Jean-Pierre Seifert. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. IACR ePrint Archive, Report 2006/052, Feb 2006.

[15] Mokari A., Ghavami B., Pedram H., SCAR-FPGA: A novel side-channel attack resistant FPGA. , 5th Southern Conference on Programmable Logic, SPL 2009

[16] http://www.linux-kvm.org/page/PowerPC. KVM webpage for the PowerPC.

[17] http://picocomputing.com/downloads/software.php Webpage with link for downloading PicoUtil

[18] Embedded Processor Block in Virtex-5 FPGAs Reference Guide, Xilinx Inc. www.Xilinx.com/support/documentation/user_**guide**s/ug200.pdf

[19] IBM PowerPC440 Microprocessor Core Programming Model Overview https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/C41DE8664DCCFAA787256B2600799BF7

[20] http://www.xilinx.com/tools/designtools.htm webpage for Xilinx development tools

[21] David Gibson, Benjamin Herrenschmidt, Device Trees Everywhere, OzLabs, 13 February 2006, http://www.ozlabs.com/~dgibson/home/papers/dtc-paper.pdf

[22] http://www.columbia.edu/kermit/ webpage for Kermit

[23] Embedded Linux Development Kit http://www.denx.de/wiki/DULG/ELDK

[24] http://xilinx.wikidot.com/ Xilinx wiki page

[25] http://www.busybox.net/ Webpage for busybox

[26] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, Anthony Liguori, kvm: the Linux Virtual Machine Monitor, Proceedings of the Linux Synopsium, pp. 225-230, June 2007

[27] Intel Corp. IA-32 Intel R Architecture Software Developer's Manual, Volume 3B: System Programming Guide, Part 2. Order number 25366919. www.**intel**.com/products/processor/**manual**s/

[28] AMD Inc. AMD64 Architecture Programmer's Manual Volume 2: System Programming. support.**amd**.com/us/Processor_TechDocs/24593.pdf

[29] Fabrice Bellard, QEMU, a Fast and Portable Dynamic Translator, FREENIX Track: 2005 USENIX Annual Technical Conference

[30] Pico E-17, Hardware reference guide, Pico Computing. http://picocomputing.com/support/E-17.php

[31] Partial Reconfiguration User Guide, Xilinx Inc. www.**Xilinx**.com/support/documentation/sw.../**xilinx**12.../ug702.pdf

[32] PlanAhead Software Tutorial, Xilinx Inc. www.xilinx.com/support/.../sw.../**PlanAhead**10-1**_Tutorial**.pdf

[33] Amdahl, Gene (1967). "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities" . AFIPS Conference Proceedings (30): 483–485.

[34] Xilinx LogiCORE IP XPS HWICAP (v5.00a) Product specification, Xilinx Inc. www.**Xilinx**.com/support/documentation/**ip**..**hwicap**/v5.../xps**_hwicap**.pdf

[35] Llamocca D., Pattichis M., Vera A., A dynamically reconfigurable parallel pixel processing system. International Conference on Field Programmable Logic and Applications, 2009.

[36] Yi Lu, Bergmann N., Dynamic loading of peripherals on reconfigurable system-on-chip. International Conference on Field-Programmable Technology, 2005. Proceedings. 2005 IEEE

[37] Gielata A., Russek P., Wiatr K., AES hardware implementation in FPGA for algorithm acceleration purpose, International Conference on Signals and Electronic systems, 2008. ICSES '08.

APPENDICES

Appendix A

System Setup

A.1. Project setup in EDK

To set up a project in the EDK, it is advisable to follow a few sample tutorials available on the Xilinx website. Below are the key steps to set up a basic system on the Pico E-17 board.

1. Include the board support files for Pico E-17 in Xilinx's installation folder.

   Pico Computing provides the files for the EDK tools to recognize the E-17 as one of the supported boards. These files consist of the Xilinx Board Definition (.xbd) and User Constraint File (.ucf) files which provide the EDK tools with information regarding the hardware on the board. These files are to be included in the following path :

   <Xilinx Installation Parent folder>/EDK/board/Xilinx/boards

2. To set the environment variables, execute the settings32.sh script by typing the following command at the terminal in the EDK folder :

   ```
   $ source settings32.sh
   ```

3. Launch the EDK GUI by executing the following command at the terminal in the EDK/bin/lin Folder :

   ```
   $ ./xps
   ```

4. Start a new project and select the Pico E-17 board from the drop down menu listing the supported boards.
5. Select suitable features for the processor speed, bus speed and optional peripherals.

   The properties for our system are :

Processor Speed	- 400 MHz
Bus Speed	- 100 MHz
Three BRAM blocks	- 32 KB, 16 KB and 16 KB
Ethernet	
RS232	- 9600 bps, No flow control, No parity bits

6. Make sure to enable interrupts for RS232 serial connection and both interrupts and DMA for the Ethernet connection.
7. In addition to the BRAM for the initial bootloader, our system also includes two additional BRAMs.
8. Pico Computing provides a package Pico_EDK_copy.zip that is to be unzipped into the project folder. This package includes the file bitgen.ut and a custom core named marvell_set_voltage_v1_00_a in the directories etc and pcores.
9. In EDK, select Project → Rescan User Repositories to reflect the changes made by the addition of the two folders.
10. Through the IP Catalogue tab of EDK, include the marvell_set_voltage_v1_00_a IP in the system.
11. In the Ports tab of the System Assembly View, set the Clock Input port of the marvell_set_voltage_v1_00_a IP to dcm_clk_s and set the SDI port as an externel

connection.

12. The external connection needs a description in the User Constraints File (.ucf) by adding this line :

    Net marvell_set_voltage_0_SDI_pin LOC= AF10 | IOSTANDARD = LVCMOS33;

13. In the Applications tab of the project, right click on the bootloop and select 'Mark to initialize BRAM'. This loads Xilinx's bootloader as part of the bitstream and will run on the PowerPC till we load the Linux Kernel onto it.

14. This completes the hardware set up of the system. Select Device Configuration → Generate Bitstream. EDK will compile the design and generate implementation.bit file that can be loaded into the FPGA.

15. Connect the Platform USB cable to the board and connect the cable to the USB port of the computer. Once the light turns green, select Device Configuration → Download Bitstream. After this, EDK should invoke 'Impact' (Xilinx's application for transfer of configuration files to target board) and perform the procedure to transfer the bitstream into the hardware. Optionally, the bitstream can also be transferred by invoking 'Impact' independently from the ISE folder where the 'Impact executable resides.

16. On correct transfer of the bitstream, the LED on the Pico board should light up indicating that the bootloop is running on the PowerPC.

A.2. Device tree generation and setup

The folder named 'bsp' contains the device tree compiler and should be placed in the project's parent directory. Next, the following steps need to be performed to generate the .dts file

1. Go to Software → Software Platform Settings
2. Select device-tree as the OS.
3. In the OS and Libraries tab, enter the name of your serial console device (e.g. RS232) and type 'console=ttyUL0' in bootargs.

Now select Software → Generate Libraries and BSPs.

This will generate a xilinx.dts file in the <project folder>/ppc440/libsrc/device-tree/ folder. Place this file in <Linux>/arch/powerpc/boot/dts/ folder of the source code and rename it to virtex440.dts

A.3. Kermit

To start the Kermit application, execute :

```
$ ./kermit
```

at the command prompt. Make the connection between the board and the USB port. Obtain the device name that appears in /dev directory on making the connection. Assuming the entry in the /dev directory for the serial connection is 'ttyUSB0' (Tele-type USB0), input the following parameters :

```
> set line /dev/ttyUSB0
> set speed 9600
> set carrier-watch off
> set parity none
> set flow-control none
> set handshake none
> robust
> connect
```

This will establish the serial connection between the board and the computer and the terminal window running kermit will act as the terminal window for the Linux system running on the board.

A.4. Linux kernel setup

To select and deslect kernel features, set up the environment variables as:

```
$ export PATH=<ELDK installation folder>/usr/bin:$PATH
$ export CROSS_COMPILE=ppc_4xx-
$ export ARCH=powerpc
```

then type :

```
$ make menuconfig
```

at the command line in the Kernel's top level directory. This should bring up the configuration menu for customizing the Kernel for the PowerPC440.

Select the following options:

```
Processor Support →
        Processor Type (AMCC 40x)
Platform Support →
        [*] Generic Xilinx Virtex Board
```

```
Kernel Options →
        Timer Frequency (250 Hz)
Networking Support →
        Networking Options →
                [*] IP : Kernel Level autoconfiguration
                [*] IP : DHCP support
Device Drivers →
        Character Devices →
                [*] Xilinx uartlite serial port support
                [*] Support for console on Xilinx uartlite serial port
        Netowrk Device Support →
                Ethernet (10 or 100 Mbit) →
                        <*> Xilinx 10/100 Ethernet Lite support
                Ethernet (1000 Mbit)
                        <*> Xilinx LL TEMAC (Local Link Tri-Mode Ethernet MAC)
                        <*> Xilinx LL TEMAC 10/100/1000 Ethernet MAC driver
        GPIO Support →
                [*] Xilinx GPIO support
File Systems →
        <*> Second extended fs support
        [*]   Network File Systems →(If someone wishes to use NFS to mount the kernel
                                  image over the network)
```

```
                  <*> NFS client support

                          [*] Root file system on NFS

Virtualization →

        [*]  KVM support for PowerPC 440 processors
```

Appendix B

Kernel Module

```
//This kernel module is used to mark the pages as non-cached. This operation is performed in the
//bram_mmap() function
#include<asm/system.h>
#include<asm/tlbflush.h>
#include<linux/mm.h>
#include<linux/sched.h>
#include<linux/types.h>
#include<linux/init.h>
#include<linux/module.h>
#include<linux/kernel.h>
#include<linux/kernel_stat.h>
#include<linux/fs.h>
#include<linux/cdev.h>
#include<asm/io.h>
#include<linux/mman.h>
#include<asm/pgtable.h>

//Physical addresses of the BRAMs
#define BRAM_0  0xFFFF8000
#define BRAM_1  0xFFFF4000
#define BRAM_2  0x84048000
```

```
#define BOGUS_PAGE_SIZE 4096


MODULE_LICENSE("GPL");

MODULE_AUTHOR("Tushar Janefalkar");

static int count=0;

struct bram_device

{

        unsigned int *k_address;

        struct cdev bram_cdev;

        unsigned long physical_address;

        struct vm_area_struct *bram_vma;

}bram;


int bram_open(struct inode *inode, struct file *fp)

{

        printk(KERN_ALERT"\n*****************device opened**********\n");

        return 0;

}


int bram_mmap(struct file *fp, struct vm_area_struct *vma)

{

        if(count == 0)

        {

                printk(KERN_ALERT"count = %d, size = %ld", count, vma->vm_end-vma->vm_start);

                printk(KERN_ALERT"\nValue pf pageprot for BRAM 0 is %lx\n",pgprot_val(vma-
>vm_page_prot));

                vma->vm_page_prot=pgprot_noncached(vma->vm_page_prot);
```

```
			printk(KERN_ALERT"\nValue pf pageprot for BRAM 0 is %lx\n",pgprot_val(vma->vm_page_prot));
			count++;
			return io_remap_pfn_range(vma,vma->vm_start,BRAM_0>>PAGE_SHIFT, vma->vm_end - vma->vm_start, vma->vm_page_prot);
		}
		else if(count == 1)
		{
			printk(KERN_ALERT"count = %d, size = %ld", count, vma->vm_end-vma->vm_start);
			printk(KERN_ALERT"\nValue pf pageprot for BRAM 1 is %lx\n",pgprot_val(vma->vm_page_prot));
			//vma->vm_page_prot=pgprot_noncached(vma->vm_page_prot);
			printk(KERN_ALERT"\nValue pf pageprot for BRAM 1 is %lx\n",pgprot_val(vma->vm_page_prot));
			count++;
			return io_remap_pfn_range(vma,vma->vm_start,BRAM_1>>PAGE_SHIFT, vma->vm_end - vma->vm_start, vma->vm_page_prot);
		}
		else if(count == 2)
		{
			printk(KERN_ALERT"count = %d, size = %ld", count, vma->vm_end-vma->vm_start);
			printk(KERN_ALERT"\nValue pf pageprot for BRAM 2is %lx\n",pgprot_val(vma->vm_page_prot));
			vma->vm_page_prot=pgprot_noncached(vma->vm_page_prot);
			printk(KERN_ALERT"\nValue pf pageprot for BRAM 2is %lx\n",pgprot_val(vma->vm_page_prot));
			count ++;
```

```
                        return io_remap_pfn_range(vma,vma->vm_start,BRAM_2>>PAGE_SHIFT, vma-
>vm_end - vma->vm_start, vma->vm_page_prot);
            }
            else if(count == 3)
            {
                        printk(KERN_ALERT"count = %d, size = %ld", count, vma->vm_end-vma->vm_start);
                        bram.bram_vma=vma;
                        printk(KERN_ALERT"\nValue pf pageprot is %lx\n",pgprot_val(vma->vm_page_prot));
                        bram.k_address=(unsigned int*) __get_free_page(GFP_KERNEL);
                        bram.physical_address=virt_to_phys((void *)bram.k_address);
                        count++;
                        vma->vm_page_prot=pgprot_noncached(vma->vm_page_prot);
                        return io_remap_pfn_range(vma,vma->vm_start,
(bram.physical_address)>>PAGE_SHIFT,  vma->vm_end - vma->vm_start,                    vma-
>vm_page_prot);

            }
            else return -1;
}

int bram_release(struct inode *inode,struct file *fp)
{
            free_page((unsigned long)bram.k_address);
            return 0;
}

int bram_ioctl(struct inode *inode, struct file *fp, unsigned int cmd, unsigned long arg)
```

```
{
        return 0;
}
struct file_operations bram_fops=
{
        .open=bram_open,
        .release=bram_release,
        .ioctl=bram_ioctl,
        .owner=THIS_MODULE,
        .mmap=bram_mmap
};
int bram_hello(void)
{
        cdev_init(&bram.bram_cdev,&bram_fops);
        cdev_add(&bram.bram_cdev,MKDEV(50,9),1);
        printk(KERN_ALERT"\nModule Initialized\n");
        return 0;
}
void bram_exit(void)
{
        printk(KERN_ALERT"\nModule exited\n");
        cdev_del(&bram.bram_cdev);
}
module_init(bram_hello);
module_exit(bram_exit);
```

Appendix C

User Code

```
/*This user code is designed to be used with the above kernel module and tells the user how to obtain a
pointer to the page allocated from the BRAM. The user is free to use these pointers as he wishes.
If used with any other kernel module the pointer may not point to a page from the BRAM desired. Please
modify this code according to what is implemented in the kernel module */
#include<stdio.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<errno.h>
#include<sys/mman.h>
#include<stdlib.h>
#include<signal.h>
#define BOGUS_PAGE_SIZE 4096
#define PROTECT 333
#define UNPROTECT 444
#define QUIT 999
struct bogus_user_data
{
        unsigned int *u_address;
        int fd;
}bogus;
```

```
int main()
{
	int result,i,value,j,delta;
	struct timeval tstart, tend;
	volatile unsigned int *bram_0;
	unsigned int *bram_1, *bram_2,*temp;
	bogus.fd=open("/dev/BRAM",O_RDWR);
	//map 3 pages from BRAMs and 1 from DDR
	bram_2=(unsigned int *)mmap(0, BOGUS_PAGE_SIZE, PROT_READ|PROT_WRITE,
MAP_SHARED, bogus.fd,0);
	bram_0=(unsigned int *)mmap(0 ,BOGUS_PAGE_SIZE ,PROT_READ|PROT_WRITE,
MAP_SHARED,bogus.fd,0);
	bram_1=(unsigned int *)mmap(0,BOGUS_PAGE_SIZE, PROT_READ|PROT_WRITE,
MAP_SHARED,bogus.fd,0);
	bogus.u_address=(unsigned int *)mmap(0,BOGUS_PAGE_SIZE, PROT_READ|PROT_WRITE,
MAP_SHARED,bogus.fd,0);
	printf("\nbram 0 mapped at  : %p\n", bram_0);
	printf("\nbram 1 mapped at  : %p\n", bram_1);
	printf("\nbram 2 mapped at  : %p\n", bram_2);
	//these pointers can then be used like any other pointers
	*bogus.u_address=888;
	*bram_2=999;
	*bram_1=999;
	*bram_0=999;
	close(bogus.fd);
	return 0;
}
```