A Secure Software Platform for Real-Time Embedded Systems

Eric James Lorden

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Masters of Science
in
Computer Engineering

Dr. Peter M. Athanas, Chair
Dr. Mark T. Jones
Dr. Cameron D. Patterson

December 15, 2006
Bradley Department of Electrical and Computer Engineering
Blacksburg, Virginia

Keywords: FPGA, configurable, security, real-time, embedded system, PLB
Copyright 2006 ©, Eric James Lorden

A Secure Software Platform for Real-Time Embedded Systems

Eric James Lorden

(ABSTRACT)

Embedded systems are becoming nearly ubiquitous, found in a plurality of devices ranging from everyday cars and dishwashers to sophisticated spy satellites and remote sensing equipment. As the applications for embedded systems increase in number and diversity and continue to pervade our lives, a need arises to secure these systems. Whether the need arises from a desire to protect personal, proprietary, sensitive, or classified information, the security of the embedded system seeks to maintain the confidentiality and integrity of data contained within the system. Research into securing embedded systems is in its nascent stages. The generally accepted methodology of securing embedded systems involves techniques that either modify an embedded system's processor or entail custom ASIC hardware. This thesis presents a novel embedded system architecture for secure software processing that does not involve processor modification, but rather processor augmentation to ensure the confidentiality and integrity of information contained within the embedded system. Specifically, configurable logic placed at the processor periphery provides just-in-time cryptographic transformation of instructions, data, and I/O of a running embedded application. In addition to presenting the embedded secure software platform, this thesis provides a characterization of the data protection architecture of the platform.

Acknowledgements

I must first thank Dr. Peter M. Athanas for serving as my committee chairman, advisor, mentor, and instructor. The knowledge and experience I gained through his classroom instruction and through his guidance during my time in the VT Configurable Computing Lab has proven invaluable and will serve me well in the future. I cannot thank him enough for introducing me to FPGAs and allowing me to pursue my academic interest in secure software architectures.

I would like to thank Dr. Mark T. Jones for serving as a committee member. His ability to distill complex arguments and his knowledge of all things computer architecture has been instrumental in the development of the work contained herein.

I would like to thank Dr. Cameron D. Patterson for serving as a committee member and instructor. His superlative knowledge of the inner workings of Xilinx FPGAs has saved me many a late night of debugging.

The work presented in this thesis could not have been accomplished without their phenomenal support and world-class expertise, and I am honored to have worked with them.

I would like to extend my gratitude to fellow members of the Virginia Tech Secure Software Group: Benjamin Muzal, Anthony Mahar, Justin Stroud, and especially Joshua Edmison. Their provisions of technical guidance and constant moral support helped make the project a resounding success.

I appreciate the joint research funding provided by Luna Innovations, Incorporated. With-

out its financial and objective oversight, the conceit and realization of the Virginia Tech Secure Software Architecture would not have been possible.

I must also thank my lab mates who have helped to make my time in the VT Configurable Computing Lab a welcome and memorable experience: Matthew Blanton, James Webb, Jorge Surís Pietri, Alexander Marschner, Meghan Quirk, Yousef Iskander, Stephen Craven, Neil Steiner, Justin Rice, Ting-Ting Meng, Todd Fleming, Vivek Venugopal, Kipp Bowen, and Tim Dunham.

I would not be the driven and successful individual I am today if it were not for Kimberly G. Lorden and Ronald R. Lorden. I am blessed for having such wonderful parents, and I am eternally grateful for having their unwavering support, love, and counsel.

Last but not least, I would like to thank my fiancé, Miss Kristen Michelle Robbins. She has put up with me throughout the trials and tribulations of my graduate career, but more importantly, she has loved me unconditionally. I thank her for her constant encouragement, her sympathetic ear, and her unshakable confidence in me.

I dedicate the work in this thesis to those who are no longer with us: my maternal grandmother, Mrs. Barbara Norford Gentry, and my paternal grandfather, Mr. Roger Arthur Lorden.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Contributions 3

1.3 Thesis Organization 4

2 Background **5**

2.1 Security Principles 5

2.2 Attacks on Embedded Systems 6

2.2.1 Physical (Hardware) Attacks 7

Invasive Attacks 7

Non-invasive Attacks 8

2.2.2 Software Attacks 10

2.2.3 Cryptographic Attacks 11

2.3 Embedded System Security 13

2.3.1 Software Mechanisms for Software Security 13

2.3.2 Hardware Mechanisms for Software Security 16

ASICs 16

ASIPs 17

Co-processors 18

FPGAs 19

Custom Secure Processors 20

2.4 Secure Software Platform 21

2.4.1 History and Perspective 21

2.4.2 Assumptions 22

2.5 eSSP Identity 23

3 Implementation **25**

3.1 eCos 25

3.1.1 eCos Kernel Modifications 26

3.2 SELFGEN 26

3.3 Platform 28

3.4 System Architecture 29

3.4.1 Instruction Encryption Management Unit 30

IEMU Bridge Unit 30

IEMU Data Tables 31

IEMU to DEMU Bus 35

AES Galois-Mode Encryption 36

Controller FSM 36

Modifications to IEMU 38

3.4.2 Data Encryption Management Unit 38

Transaction FIFO 40

Data Capture Module 41

Data Process Module 44

DEMU Data Tables 48

Data Translation Unit 52

4 Results **53**

4.1 Methodology 54

4.2 Latency 56

4.2.1 Read Latency 58

4.2.2 Write Latency 61

5 Conclusion **64**

5.1 Conclusion 64

5.2 Future Directions 67

Bibliography **69**

A Source Code **77**

A.1 PPC405 Startup Assembly 77

A.2 Benchmark Caller Code 78

A.3 Read Benchmark Assembly 80

A.4 Write Benchmark Assembly 83

List of Figures

2.1 Embedded System Attack Classification System 7

3.1 eCos SELF Layout 27

3.2 eSSP System Architecture 31

3.3 IEMU Architecture 32

3.4 Page-to-Key and Page-to-Counter Mapping Hardware in IEMU Tables . . . 34

3.5 DEMU Architecture 39

3.6 DEMU FIFO Interface Signaling and Fields Reference 41

3.7 DEMU PLB Transaction Capture Unit Finite State Machine 43

3.8 DEMU PLB Transaction Process Unit Finite State Machine 46

3.9 PLB Write Transaction Timing Protocol 47

3.10 PLB Read Transaction Timing Protocol 48

3.11 Page-to-Key Mapping Hardware in DEMU Tables 49

3.12 DEMU Supplementary Data Table Fields Reference 50

3.13 Data Tagging Scheme 51

4.1 Read Latency Characteristics of the DEMU 59

4.2 Perceived PPC405 Write Latency Characteristics 63

List of Tables

3.1 Resources available on the Xilinx XC2VP30 Virtex-II Pro FPGA 29

Glossary of Acronyms

AES Advanced Encryption Standard

API Application Program Interface

ASIC Application-Specific Integrated Circuit

ASIP Application-Specific Instruction Set Processors

CAM Content-Addressable Memory

CCR Core Configuration Register

CORBA Common Object Request Broker Architecture

CPU Central Processing Unit

DCC Data Cache Controller

DCM Data Capture Module

DDR Double Data Rate

DEMU Data Encryption Management Unit

DES Data Encryption Standard

DIMM Dual Inline Memory Module

DPA Differential Power Analysis

DPM Data Process Module

DTU Data Translation Unit

eCos Embedded Configurable Operating System

ELF Executable and Linking Format

eSSP Embedded Secure Software Platform

FIFO First In First Out

FPGA Field Programmable Gate Array

FSM Finite State Machine

GCM Galois Counter Mode

HAL Hardware Abstraction Layer

IBM International Business Machines

ICC Instruction Cache Controller

IDL Interface Description Language

IEMU Instruction Encryption Management Unit

IPIF Intellectual Property Interface

ISR Interrupt Service Routine

JTAG Joint Test Action Group

LAN Local Area Network

MAC Message Authentication Code

MIMO Multiple Input Multiple Output

MISO Multiple Input Single Output

MMU Memory Management Unit

NIC Network Interface Controller

OPB On-chip Peripheral Bus

OS Operating System

PCI Peripheral Component Interface

PLB Processor Local Bus

POSIX Portable Operating System Interface

PPC405 IBM Power PC 405 Processor

RAM Random Access Memory

RISC Reduced Instruction Set Computer

RTOS Real-Time Operating System

SDRAM Synchronous Dynamic Random Access Memory

SELF Secure ELF

SELFGEN Secure ELF Generator

SEU Secure Execution Unit

SKU Secure Key Management Unit

SLU Secure Loading Unit

SPA Simple Power Analysis

SSP Secure Software Platform

TLB Translation Look-aside Buffer

UART Universal Asynchronous Receiver/Transmitter

Chapter 1

Introduction

1.1 Motivation

Embedded systems are becoming nearly ubiquitous, found in a plurality of devices ranging from everyday cars and dishwashers to sophisticated spy satellites and remote sensing equipment. As the applications for embedded systems increase in number and diversity and continue to pervade our lives, a need arises to secure these systems. Whether the need arises from a desire to protect personal, proprietary, sensitive, or classified information, the security of the embedded system, at a minimum, ensures the confidentiality and integrity of information contained within the system. The confidentiality of a piece of information restricts access to that piece of information to a certain subset of authorized entities. The integrity of information relates to its correctness, completeness, and compliance with the intention of its creator(s). Therefore, the security mechanisms of embedded systems should restrict access to data and establish that the data is original and unaltered.

Research into securing embedded systems is in its nascent stages. Kocher et al. [1] provide a general hardware taxonomy of secure embedded system architectures that includes Application-Specific Integrated Circuit (ASIC) devices, application-specific instruction set

processors, and embedded general-purpose processors used in conjunction with FPGAs, hardware acceleration, embedded co-processors, or with extended instruction sets. A common characteristic shared amongst all of the architecture,s with the exception of the embedded general-purpose processor used with an FPGA, is the need to in some way modify the hardware of the processor to support security mechanisms. The modifications required may prove prohibitively expensive or technically infeasible. Thus, there exists a need for a secure embedded system that requires absolutely no modification to the hardware of the embedded processor.

Most embedded processing systems are custom-constructed systems in that they contain custom hardware to support data processing. If the custom hardware is implemented in an ASIC technology, then the manufacturing costs alone may be an insurmountable obstacle, even at high production volumes [2]. There also exists a time-to-market constraint as well as a system testing constraint when developing the custom hardware of a viable embedded system. The faster an embedded system can be developed and verified, the more competitive it can be in its niche market. These constraints generate the need for an extensible platform on top of which embedded systems can be quickly constructed and easily tested.

The above factors combine to form the motivation for the Virginia Tech Embedded Secure Software Platform (eSSP). The eSSP is constructed using FPGA fabric surrounding an embedded IBM PowerPC-405 (PPC405) general purpose processor that is available on a single chip, in the form of a Xilinx XC2VP30 FPGA. No modifications are made to embedded processor, but instead custom hardware is designed on-chip in the FPGA fabric that provides the cryptographic services necessary to ensure that information processed on the PPC405 remains confidential. The ease of development and the reconfigurable nature of the FPGA fabric make the eSSP an ideal platform for creating secure embedded systems.

1.2 Contributions

This thesis makes the following contributions to the field of secure embedded systems:

1. Presentation of an embedded secure software platform based on the Virginia Tech Secure Software Platform [3].

2. Introduction of a method for protecting the confidentiality and integrity of embedded application data in the form of the the Data Encryption Management Unit (DEMU).

3. Implementation of a novel PLB-to-PLB bridge that supports all PLB transactions on its slave interface and contains a custom, lightweight PLB master on its master interface.

4. Introduction of a method for securing the confidentiality and integrity of embedded real-time application data, SELFGEN.

5. Modification of a real-time embedded operating system, eCos, to provide a compatible execution environment with the eSSP and the embedded PPC405 processor.

6. Characterization of the DEMU's performance.

7. Determination of the impact the DEMU has on the guarantees provided by an RTOS is negligible through bounding the performance behavior of the DEMU.

1.3 Thesis Organization

The remainder of this thesis is organized as follows: Chapter 2 provides the background information necessary to understand the remainder of the thesis. It outlines hardware and software mechanisms used to secure embedded systems as well as introductory material on the related Secure Software Platform. Chapter 3 introduces the eSSP's system architecture and constituent components, the IEMU and the DEMU. Chapter 4 characterizes the performance of the eSSP in addition to providing a security analysis of the final system. Lastly, Chapter 5 reiterates the contributions of this thesis, summarizes the important performance results, and provides future directions for eSSP work.

Chapter 2

Background

This chapter provides the background material necessary for understanding the work presented in the remainder of this thesis. First presented are the six fundamental principles of security. A subsequent discussion provides an overview of the different types of hardware and software attacks that an embedded system can face as well as mechanisms and architectures for obviating the attacks. The chapter concludes with an introduction to the predecessor platform of the eSSP, the Secure Software Platform (SSP).

2.1 Security Principles

To understand the functional requirements of a secure embedded system, six primary tenants of security must be characterized. The six principles of security defined by Parker in [4] are availability, confidentiality, integrity, authenticity, possession, and utility. Together, the principles form the so-called Parkerian Hexad. An embedded system makes assurances on the availability of information by providing mechanisms, such as high availability protocols and redundant network architectures without single points of failure [5], to authorized users that ensure access to information in timely and reliable manner. Confidentiality restricts

information access to a certain subset of authorized entities, typicalling using cryptographic algorithms [6]. The integrity of information relates to its correctness, completeness, and compliance with the intention of its originator(s). Typical methods of ensuring the integrity of information are cryptographic hashes and digital signatures [7]. Information authenticity ensures the correct provenance and authorship of a piece of information. Mechanisms for providing means to asses the authenticity of a piece of information include Message Authentication Code (MAC)s and challenge-response protocols [8]. The possession of a piece of information relates to its authorized custodial entity. Consider confidential information sealed or even unsealed in a secure container. If that container is stolen, the security of the system is compromised because that information is no longer under the control of its authorized custodial entity. Security systems, armed guards, and other such *physical means* are used to protect the possession of a piece of information. The utility of a piece of information considers its usefulness when compromised. If other unauthorized information can be gleaned from a piece of compromised information, then it has utility; otherwise, the piece of information has no utility. Utilizable information is usually kept separated or protected from compromise by some physical means, such as hardware partitioning [9].

2.2 Attacks on Embedded Systems

To compromise any or all of the tenants of security described in Section 2.1, embedded system adversaries employee various means of attack. The types of attack can be broadly categorized into physical and logical attacks. Physical and side-channel attacks assail an embedded system's hardware components, and logical attacks aggress upon an embedded system's software and cryptographic services. Software attacks focus on an embedded system's software components, and cryptographic attacks concentrate on breaking the cryptographic mechanisms protecting the information security of an embedded system. The physical and side-channel attacks can be further subdivided into invasive attacks and non-invasive attacks. This classification system is illustrated in Figure 2.1. The remainder of this section

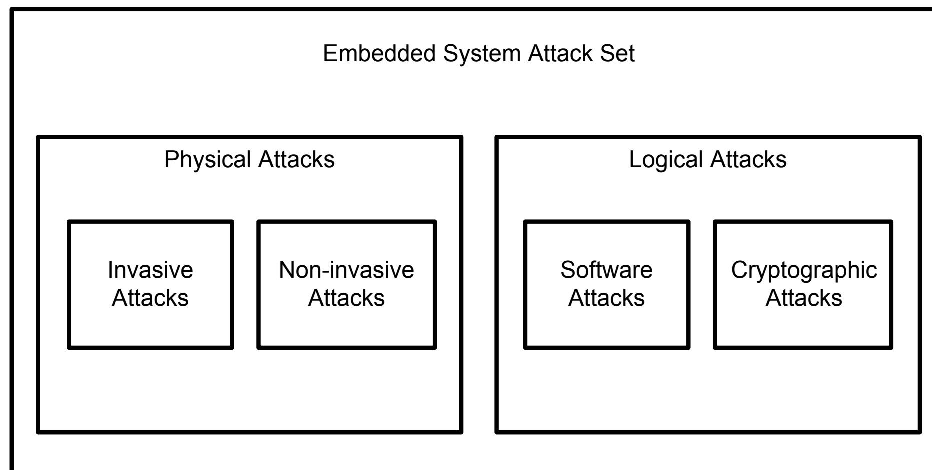

Figure 2.1: Embedded System Attack Classification System

provides an overview of the various types of attacks that appear at the leaves of the attack classification hierarchy. This section is not intended to serve as a complete reference to all attack methods, but rather characterizes the plurality of attack methods that exist for an embedded system.

2.2.1 Physical (Hardware) Attacks

Invasive Attacks

Invasive physical attacks require direct access to the silicon and substrate layers of chips within an embedded system in order to observe or modify the behavior of the chips [10]. An invasive attack by definition implies that the possession security principle has been broken, since observing the silicon layer of an embedded system requires physical control of the system by an unauthorized user. Invasive attacks typically require expensive equipment such as sensitive abrasion machines, microelectronic probes, and high resolution cameras. As such, they are typically difficult to mount or even repeat once successfully accomplished. There exist companies, such as Chipworks [11], that have invested vast resources to mount

invasive attacks to reverse engineer chips. The process begins by removing a chip's packaging using acid or sensitive milling machines [12]. The process continues with a succession of microscopy and the milling away of the various layers comprising the chip. At any point during the process, microprobes can be used to physically observe the values on buses or values at the boundaries of memories and processing units. Invasive attacks are difficult but not impossible to thwart since an attacker has physical access to the device. Techniques such as bus encryption and randomization, discussed further in [13] mitigate the success of invasive attacks.

Non-invasive Attacks

Non-invasive attacks, unlike invasive attacks, do not require direct physical access to the device's silicon; however, they do require physical access to an embedded system, just not its physical silicon chips. Non-invasive attacks tend to require more ingenuity and less financial support to execute than invasive attacks. As such, they are the most prevalent set of physical hardware attacks and highly repeatable [14]. One type of non-invasive attack is called a *timing analysis attack*. The attack works by considering a system with s stages of execution that each mix a bit of a known input. The s stages should each take non-constant time and be a function of each input bit b_i. By repeatedly introducing different input vectors, measuring the time to a change in the outputs, and correlating these results with a statistical timing model of the system, the key of the system can be exposed without resorting to the cryptographic techniques discussed in Section 2.2.3 [15]. There are techniques, such as message blinding that transform the input with a random value before encryption and invert it before decryption, that can thwart a timing analysis attack [16] [17].

Another type of non-invasive hardware attack is the *power analysis attack* that relies on correlations between an embedded system's operating characteristics and its power consumption. The operation of an embedded system can be correlated to the computations it performs. Specifically, the switching characteristics of CMOS technology causes it to con-

sume more power as more logic gates switch from low-to-high or high-to-low in the system. Simple Power Analysis (SPA) takes advantage of any power-sensitive implementation stages of an embedded system to reveal confidential information, such as cryptographic keys, without the use of statistical correlation [18]. Differential Power Analysis (DPA), similar to timing analysis, relies on correlating measured responses of an embedded system to an advanced statistical model of the system. DPA measures the dynamic power consumption of the system, while timing analysis measures the time it takes outputs to change provided an input. Similar to a DPA attack are *electromagnetic (EM) attacks* that attempt to correlate the electromagnetic measurements of an embedded system to a statistical model of its electromagnetic operation [19].

The last major class of non-invasive embedded system attacks involved *fault induction techniques*. These techniques vary widely from manipulating the voltage to raising or lowering the primary clock frequency of the system to induce faults. Another fault injection technique is to expose the embedded system to radiation. Dusart et al. induce faults into the rounds of the DES algorithm to reveal confidential information [20]. Irradiative fault injection techniques can be used to break AES. With just 10 faults injected into the rounds of AES, Dusart et al. were able to recover a 128-bit key within minutes of the attack's introduction [21]. Carrying out non-invasive fault injection attacks is inherently difficult because most successful fault injection techniques rely on a differential analysis of an embedded system, and a detailed understanding of the algorithms in the embedded system must first be known. This may or may not be the case with a newly confiscated embedded system. Moreover, the exact point of fault introduction affects the differential analysis used to reveal the weaknesses of algorithms. These points may or may not be readily available to an attacker without further invasive attacks and reverse engineering.

2.2.2 Software Attacks

Software attacks on embedded systems are just as diversified as hardware attacks. The attacks generally rely on exploiting the weaknesses of software protocols or function arguments. The most common software attack on an embedded system is a *buffer overflow attack*. According to CERT [22], buffer overflow attacks account for 66% of the vulnerabilities in trusted programs. The buffer overflow attack occurs when a buffer has poor bounds checking, arising from incorrect loop bounding, format string errors, bad buffer length parameter passing, or any number of other such sources. By overflowing the buffer, an attack can effectively gain control of a system. For example, by overflowing the call stack (a buffer) during function invocation and placing a return address of a malicious subroutine in the overflowed area, the malicious subroutine could begin execution when the currently executing function returns.

Another common example of a software attack on an embedded system involves *debuggers*. Debugger applications can be used maliciously to attach to a secure process within a mixed security operating system to observe any characteristics of the running process. The utility of a debugger increases if the security mechanisms of the system are all implemented in software.

Another example of a software attack on an embedded system involves vulnerabilities introduced through *typecasting*. Typecasting is a programmatic way of changing how software handles a variable, and typecasting usually has a compiler and operating system specific implementation. Using typecasting, an attacker can pass false information to a function and inadvertently corrupt an algorithm. For example, consider a cryptographic function Λ that expects to work with floating point numbers, but the parameter passed to Λ is an unsigned integer or signed integer. The result of Λ is not guaranteed to work since it expected a floating point number but received an integer of some sort. Some binary representations of a signed or unsigned integer can, after being typecast to a floating point number, be considered NaN (not a number), positive infinity, or negative infinity in floating point that can affect the operation of Λ. Any other number of software attacks exist that exploit the weak-

nesses of protocols and function calls. For further information regarding software attacks on embedded systems, refer to Bond et al. [23].

2.2.3 Cryptographic Attacks

In addition to software and hardware attacks on an embedded system, there are three primary mathematical attacks used against an embedded system's cryptographic services. The first and by far the least sophisticated technique is called a *brute-force attack*. Given ciphertext and a cryptographic algorithm, a brute-force attack attempts to exhaustively search the key space of a cryptographic algorithm to find the correct key to decrypt the ciphertext. This generally proves computationally infeasible for large key sizes, for the magnitude of the search space required to unlock a binary key of length n is 2^n. In a brute-force attack, the expected number of random selections of a full-length key before finding the correct selection is equal to half the magnitude of the search space. For example, given a 64-bit key, the search space is the finite field $\mathbb{F}_2^{64}$ and $|\mathbb{F}_2^{64}| = 2^{64}$. Thus, the expected number of random key selections before the correct one is found is 2^{63}.

The second attack exploits the mathematics underlying the birthday paradox and is aptly named the *birthday attack* to reduce the search space of the brute-force attack. The birthday paradox uses the pigeonhole principle and statistical analysis to prove that given two random elements of a discrete uniform distribution with a range of $[1,d]$ the probability $p(d;n)$ that at least two numbers are the same is given by

$$p(d;n) = \begin{cases} 1 - \prod_{i=1}^{n-1}(1 - \frac{i}{d}) & \text{if } n \leq d \\ 1 & \text{if } n > d. \end{cases} \tag{2.1}$$

For large d, this formula can prove computationally cumbersome; however, the Taylor Series expansion of $e^x = \sum_{n=0}^{\infty}(\frac{x^n}{n!})$ can be used as an excellent approximation to the $n \leq d$

case of Equation 2.1, yielding

$$
\begin{aligned}
p(d;n) &= 1 - \prod_{i=1}^{n-1}\left(1 - \frac{i}{d}\right) \\
&\approxeq 1 * e^{\frac{-1}{d}} * e^{\frac{-2}{d}} * e^{\frac{-3}{d}} * \cdots * e^{\frac{-(n-1)}{d}} \\
&= e^{\frac{-n*(n-1)}{2*d}} \\
\therefore & \\
p(d;n) &\approxeq e^{\frac{-n*(n-1)}{2*d}} \text{ and, solving for n,} \\
n(d;p) &\approxeq \sqrt{2 * d * ln\left(\frac{1}{1-p}\right)}.
\end{aligned}
\tag{2.2}
$$

The results of Equation 2.2 indicate that on average, $p = 0.5$, it takes approximately $1.177\sqrt{d}$ random pairings to find a collision in a cryptographic algorithm or hash function, where for a binary key $d = 2^{\text{key length}}$. Once a collision is found, it may prove to be the key used to decrypt ciphertext. If it is not the key and the encryption method is known, the collision can still be used as a starting point from which to work backward to find the key used to generate the ciphertext. The ultimate goal of this attack as well as the other cryptographic attacks is discovering the plaintext, i.e. the protected information. This mathematical result is an extremely powerful cryptanalytic technique.

The third primary mathematical attack used against embedded cryptographic systems is the *preimage attack*. Given a cryptographic function $\mathfrak{F}$ and a message m_1, a preimage attack seeks to find a second message m_2 such that $\mathfrak{F}(m_1) = \mathfrak{F}(m_2)$. A preimage attack differs from a birthday attack in that a message m is known for a preimage attack and a second image is sought for a collision; whereas, in a birthday no messages are known but random pairings of messages are sought for a collision. The probability that a preimage attack will yield a

collision in a discrete uniform distribution with a range of $[1,d]$ is

$$\begin{aligned} q(d;n) &= 1 - \left(\frac{d-1}{d}\right)^n \text{and solving for n,} \\ n(d;q) &= \frac{ln(1-q)}{ln(\frac{d-1}{d})}. \end{aligned} \tag{2.3}$$

With the same goal as the birthday attack, the preimage attack can be used to find a key to crack ciphertext by working backward given the collision and the function $\mathfrak{F}$.

2.3 Embedded System Security

To address the hardware and software attacks on embedded systems described in Section 2.2, the systems must be constructed with hardware and software mechanisms to prevent the attacks or abate their success. This section provides a conspectus of hardware and software mechanisms used to secure applications running on embedded systems.

2.3.1 Software Mechanisms for Software Security

There are a wide range of software-based techniques used to protect embedded system applications. These techniques reside in the three primary software layers of a general embedded system: the kernel layer, the middleware layer, and the application layer. The kernel layer of an embedded operating system is the lowest level of the software layers and provides hardware access abstractions to higher software layers, in addition to coordinating the embedded system's hardware and software services. The middleware layer of an embedded operating system is also known as the Application Program Interface (API) layer, and it serves to further abstract kernel software services to applications as well as introduces its own services. The topmost layer, the application layer, is the software layer in which embedded

applications run. Since the application layer relies on services provided by the middleware and kernel layers, software security solutions do not generally exist for this layer.

A middleware technique for ensuring the utility and possession information security tenants for an embedded system is the Middleware-based Intrusion Detection for Embedded Systems (MIDES) [24], and it is built atop a Common Object Request Broker Architecture (CORBA) platform. CORBA allows different programs on different platforms to interoperate by bundling code with descriptors identifying capabilities and invocation methods, and passing the bundled code across an interconnection network or Local Area Network (LAN). CORBA provides an Interface Description Language (IDL) to expose the code's methods and maps the exposed methods to known programming languages on the various platforms, making them universally accessible to any connected platform. MIDES uses a set of software-based *interval* and *procedural* sensors. The interval sensors periodically profile method invocations on an embedded system using CORBA mechanisms. The procedural sensors monitor the execution trace of a running embedded application and compare the results to a known execution trace. Results indicate that the usage of interval sensors to monitor embedded system activity increase execution overhead by only 2.5% to 6.9% on embedded systems running the Linux operating system. The procedural sensors incur substantially more overhead of 17.4% to 17.8%, because they involve tracing the execution of a running embedded system application. This technique for securing an embedded system against attack proves insufficient due to fundamental security flaws in CORBA [25].

Another software-based security mechanism, program shepherding, is introduced by Kiriansky et al. [26]. The authors propose a kernel-level mechanism that monitors control flow during an embedded application's execution to restrict the execution of code based on its *origin*. The code origin categories are original pristine from disk, dynamically generated but unmodified, and dynamically generated but modified. By categorizing (shepherding) code, the authors seek to protect against the disclosure of confidential information on an embedded system through authentication techniques. The categorization of code is implemented as a series of flags in a custom data structure (segment) prepended to an embedded application

during compilation through the Runtime Introspection and Optimizer (RIO) tool. This data structure is protected by the kernel and is only writeable by code contained in the segment itself. In addition to categorizing an embedded system's application code, RIO attempts to trace all branch statements in an application and store them for a run-time comparative analysis by the shepherding components built into the kernel. The authors assert that the security policy created using code flagging and trace execution prevent a whole host of attacks including malicious instructions masquerading as data and code injection attacks. However, the shepherding mechanisms in the kernel are themselves (software) code and are susceptible to tamper.

Oikawa et al. propose to co-locate multiple operating systems atop a secure microkernel [27]. It is assumed that each operating system and its running applications will be in the same security domain and therein no security mechanisms will be required of operating systems, only the trusted microkernel. The implementation involves a singular trusted *TL4 microkernel* with multiple $\mu ITRON$ operating systems (kernels) running on the microkernel. The *TL4 microkernel* provides the general hardware abstractions, virtual memory, interprocess communication, and security domain features. Both kernels are modified to handle the layering of interrupt calls and fundamental multi-operating system process scheduling. The authors describe a 70% increase in the volume of the code as a result of the modifications, but only a $1\mu s$ increase in latency of tasks tested on the system. The major drawback from this approach is that segregation of applications occurs *virtually* through co-located operating systems, without hardware enforcement of the segregation. Thus, software attacks against the *TL4 microkernel* could allow a process on one operating system to read or modify the instruction or data of another process in a different operating system because all operating systems in this configuration share the same physical memory.

The methodologies just described share a common characteristic with all software-based embedded system application security mechanisms. Purely software mechanisms, though they aid in providing security to an application running on embedded system, do not ensure the security of the applications running because the mechanisms themselves are software. As

such, they are extremely susceptibile to modification just as any piece of software is by its very nature. They can be successful, however, when coupled with complementary hardware mechanisms. Such hardware mechanisms are described in Section 2.3.2.

2.3.2 Hardware Mechanisms for Software Security

This section discusses hardware mechanisms used by embedded system designers to thwart or mitigate the success of security attacks on embedded systems. A taxonomy of hardware mechanisms for software security is introduced followed by a detailed discussion of each, with associated examples from literature. Kocher et al. in [1] provide a general taxonomy of architectures used to secure applications running on embedded systems. The taxonomy includes Application-Specific Integrated Circuit devices (ASICs), Application-Specific Instruction Set Processors (ASIP), secure embedded processors, and embedded general purpose processors used in conjunction with hardware acceleration, co-processors, or field-programmable gate array (FPGA) technology.

ASICs

Certain subclasses of embedded systems with power-sensitive needs use ASIC technology to implement security protocols or algorithms. In [28], Burg et al. present an ASIC design of an AES cryptographic chip. The ASIC is designed to balance the data path delays of the AES encryption and decryption implementation in a 0.25 μm fabrication process. The balance is required because the decryption data path can incur a 30% greater delay than the encryption path. This arises because an inverse column mix operation used during the decryption process involves a more complex algorithm than the column mix operation used during encryption. Overall, the chip sustains a throughput exceeding 2.0 Gbps with a 256-bit key while only consuming 600 mW of power. Work at Sandia National Labs by Wilcox et al. [29] describes a Data Encryption Standard (DES) implementation that achieves through-

puts exceeding 10Gbps and uses less than 2 mW of power at 1 MHz operation. Sen et al. [30] claim that implementations for AES and DES require large amounts of hardware or software, especially for power-sensitive embedded systems. They develop a novel cryptographic algorithm using cellular automata that consists of a number of identical cells arranged in a regular pattern. The next state of any cell is a function of the present states of its left and right neighbors. Using Shannon's security quotient that measures the entropy of an algorithm, Sen et al. prove that their cryptographic algorithm is stronger than DES and only slightly less strong than AES.

The goal of ASIC cryptographic hardware in embedded systems is to provide a means to cryptographic services. However, adversaries rarely attack the theoretical strength of well-designed cryptographic hardware but instead attack some other weakness in the system. An analogy is a burglar (embedded system attacker) attempting to burgle a house (an embedded system) having titanium doors with electromagnetic locks (cryptographic hardware). The burglar, upon noticing such a formidable obstruction to securing his spoils, might opt instead to break in through a window or disconnect the electricity, disabling the electromagnetic locks. A truly secure embedded system is architected with more than just cryptographic ASIC hardware.

ASIPs

Kocher's taxonomy of embedded system hardware security architectures also includes application-specific instruction set processors with extensible instruction sets. Commercially available ASIPs include Tensilica's Xtensa [31], Altera's NIOS [32], and a collaborative effort between Hewlett-Packard and STMicroelectronics named Lx [33]. Academic examples of ASIPs include CHIMERA [34] by Ye et al. and Adaptive Explicitly Parallel Instruction Computer (AEPIC) [35] by S. Talla. Using the extensibility of the instruction set architecture, certain steps of cryptographic algorithms such as the Feistel rounds of DES or the mix columns step of AES can be transformed into atomic instructions implemented in configurable hard-

ware [1]. Yu et al. present heuristic methods to identify and select basic blocks of code to transform into custom hardware instructions [36]. The problem of automatically identifying and selecting code to transform into hardware proves difficult because code must be profiled for Multiple Input Single Output (MISO) and Multiple Input Multiple Output (MIMO) patterns that can be exponential in number in any given basic block. Ragel et al. [37] suggest using ASIPs to generate microinstructions (micro-operations) to perform integrity checking on instructions and bounds checking on stack and application boundaries in order to prevent instruction injection, buffer overflow, and data path hijacking attacks. Simulation reported an average performance penalty of 1.93% across ten benchmarks though ASIP generation of the microinstruction monitoring mechanisms. A major drawback of the ASIP approach is that implementation tools used to facilitate the generation the custom instruction hardware on the various ASIP platforms are currently immature, casting doubt upon the feasibility and correctness of implementations.

Co-processors

Another hardware methodology in Kocher's taxonomy of embedded system security hardware architectures and mechanisms is the use of general embedded processors with secure co-processors. Vandorn et al. [38] propose a secure co-processor that is completely independent of an embedded system's (host's) main processor. The duties of the secure co-processor involve booting the operating system of the main processor into a known, secure state where *invariants*, the security properties of the operating system, can be subsequently monitored for malicious behavior. The authors propose a methodology in which key data structures within an operating system's kernel are monitored periodically for malicious modification using differential analysis against a known tamper-free execution. The primary problem with this approach is that malicious behavior can take place inside the time interval in which the co-processor monitors the data structures. Any compromise of security defeats the purpose of having a security system in the first place: to prevent such attacks.

Arora et al. propose a similar security co-processor [39]. Instead of monitoring kernel data structures, a secure co-processor monitors *inter-procedural* as well as *intra-procedural* calls by building a function call graph and control-flow graph, respectively, of an embedded application. The traces of the running embedded application are compared against the traces of a normal execution of that application for detection and possible reaction to malicious software behavior. As with a majority of secure co-processors, no implementation is made, but results are gathered through simulation. Simulation indicates a 2% maximum performance penalty for using the secure co-processor to check the function call and control-flow graphs of an executing embedded application.

In addition to random number generation, Su et al. propose a co-processor that accelerates the AES, RSA, SHA-1, and MD5 cryptographic algorithms [40]. A new instruction on a host processor turns control over to the cryptographic co-processor to carry out the cryptographic algorithms. Their implementation in $0.18\mu m$ CMOS technology operates at 83 MHz and consumes 384 mW of power while producing throughput rates of 1 Gbps for AES, 81 Gbps for RSA, 120 Mbps for SHA-1, 130 Mbps for MD5, and over 6.4 Gbps for random number generation. The Integrated Monitoring for Processor REliability and Security (IMPRES), a software/co-processor architecture developed by Ragel and Parameswaran [41], attempts to verify the integrity of an embedded system application's instruction memory using a secure co-processor that compares a known hash value of instruction blocks to a hash of the application at runtime. Secure co-processors, though they accelerate the performance of cryptographic algorithms or verify the integrity of a running embedded application, can easily be circumvented by replacement unless implemented on-die with the host processor.

FPGAs

Field programmable gate array (FPGA) technology used in conjunction with general purpose embedded processors is another mechanism mentioned by Kocher in his taxonomy of embedded system protection architectures. FPGAs are programmable logic devices that use

configurable look-up tables and routing resources for creating hardware designs quickly and efficiently. Rapid implementation is often offset by slower operational speeds and higher power requirements when compared to ASICs of the same design. FPGAs are primarily used in the embedded system realm to accelerate cryptographic hardware. High performance FPGA implementations of the DES cryptographic algorithm in feedback mode are presented by Kaps and Paar in [42] and by Patterson in non-feedback mode in [43]. In addition to cryptographic hardware acceleration, secure embedded system architectures built atop FPGA platforms are emerging. The SafeOps architecture for embedded software security uses compiler techniques in conjunction with FPGA hardware to achieve the integrity protection of embedded system applications [44]. The compiler technology tags register access sequences that are compared at run time by FPGA hardware to a known *pristine* register sequence, i.e. a sequence known to be trusted. If any incongruities between the running application register sequence and the *pristine* register sequence are found at run time, the FPGA hardware halts the execution of the embedded general purpose processor. Other secure embedded systems developed on FPGAs include the Secure Software Platform, discussed further in Section 2.4, and the work presented in this thesis detailed in Chapter 3. As with nearly every implementation architecture, there exist vulnerabilities associated with FPGAs. Bitstream reverse engineering, SRAM cloning, readback attacks, and other vulnerabilities with respective countermeasures are detailed by Wollinger et al. in [45].

Custom Secure Processors

The last mechanism mentioned in Kocher's taxonomy is the custom secure embedded processor. Best introduces the concept of a custom secure processor to decrypt memory that has been encrypted [46]. The XOM (eXecute-Only Memory) architecture developed by Lie et al. [47] attempts to further the idea of protecting an embedded system application's code confidentiality. XOM uses a tagging scheme to label all architectural components used by an executing embedded application with a tag unique to that application. The tag is mapped to

a key that can be used to encrypt and decrypt data. This tagging scheme cryptographically segregates executing processes by allowing access to architectural components (registers, etc.) based on the tags, and the scheme uses the idea of Best to decrypt code only-the-fly. The concepts developed by Best and Lie et al. are further refined in the AEGIS architecture proposed in [9] [48]. The AEGIS architecture uses an architectural tagging scheme to exclusively bind running embedded applications to processor features such as cache lines and general-purpose registers, providing confidentiality of instructions and data. However, the AEGIS architecture further ensures the integrity of code and data by storing hash values of the data and performing a comparative analysis at run time to detect tampering or other such malicious behavior. Both the XOM architecture as well as the AEGIS architecture require modifying existing processor architectures to incorporate their respective security mechanisms.

2.4 Secure Software Platform

This section provides perspective on the eSSP by introducing its predecesor, the Secure Software Platform (SSP). Assumptions made to scope the security requirements of the SSP subsequently serve as the basis for the eSSP.

2.4.1 History and Perspective

The Secure Software Platform began development in early 2004. Edmison [3] architected the platform and modified the Linux operating system to interface with the platform. The SSP architecture consists of three logical components: an Instruction Encryption Management Unit (IEMU), and a Data Encryption Management Unit (DEMU), and a Secure Key Management Unit (SKU). The function of the IEMU is to ensure the confidentiality and integrity of applications running on the platform and was implemented by Mahar in [49]. The Secure Key Management Unit (SKU) serves to provide a key management scheme us-

ing hierarchical security credentials to generate one or more instruction and data keys for a secure application and was implemented by Muzal [50]. The DEMU serves to ensure the confidentiality and integrity of data of applications running on the platform and serves as a fundamental contribution of this work, since they both share the same author. The eSSP is based on the SSP; however, contains a different operating system and different hardware. Modifications to the primary hardware of the SSP for the eSSP are discussed in Sections 3.4.1, 3.2 and 3.4.2.

2.4.2 Assumptions

The assumptions made by the authors of the SSP and subsequently of the eSSP to scope the security attacks and mechanisms addressed by the platforms include the following statements.

1. The attacker does not have physical silicon access to the FPGA internals and the embedded PPC405 processor.

2. The embedded processor, the IBM PowerPC-405 processor, is correctly implemented as the result of being constructed by a trusted foundry.

3. All cryptographic algorithms used in the platform have been correctly implemented with appropriate parameters, e.g. key length and truly random numbers, such that the theoretical strength of the cryptographic algorithms is upheld during implementation.

4. A key management scheme is correctly implemented for the platform.

5. Any software developed to facilitate the interaction of the running embedded application with the platform is not trusted.

6. Pre-processing of protected applications is necessary to ensure the confidentiality and integrity of that application's instruction and data segments.

7. The attacker cannot mount any invasive or non-invasive physical attack, with the exception of the probing of bus signals external to the FPGA.

8. Data and Instructions are perfectly segregated by pre-processing techniques into contiguous segments and remain segregated through the Harvard architecture provided by the PPC405.

2.5 eSSP Identity

Using the taxonomy of embedded system attacks shown in Figure 2.1 and the assumptions made in Section 2.4.2, the scope of the security claims of the eSSP can be ascertained. Assumptions 1 and 7 preclude the possibility of any physically invasive attack. Therefore, the eSSP does not contain any mechanisms for defending against physical attacks, with the exception of the non-invasive probing of bus signals external to the FPGA. Cryptographic algorithms are used to encrypt/decrypt data at the processor boundary, making any information probed external to the FPGA nonsensical. *Fault induction* attacks are also not considered in the scope of the eSSP, as detailed in assumptions 2, 3, and 4. However, the sheer complexity of the eSSP system significantly mitigates the success of invasive attacks such as *timing analysis* attacks, *power analysis* attacks, and *EM analysis* attacks. The feasibility of the cryptographic attacks outlined in Section 2.2 is obviated by assumptions 3 and 4. Preventing the last class of attacks on embedded systems, software-based attacks, is the primary focus of the eSSP architecture. These attacks are addressed by the cryptographic segregation of protected applications, as detailed in [3].

The salient characteristic that differentiates the eSSP from the other embedded system hardware mechanisms is that it provides data and instruction confidentiality and integrity without the need to modify the embedded application processor. This distinguishes it from XOM and AEGIS discussed in Section 2.3.2. The eSSP architecture relies on processor augmentation, not modification, to achieve its security claims of instruction and data confiden-

tiality and integrity. Moreover, the variable granularity of instruction and data protection, currently set at a page size of 4KB, sets it apart from existing architectures. The eSSP is a holistic approach to a secure architecture rather than just an amalgamation of cryptographic hardware, contrasting it to the ASIC and ASIP mechanisms described in Sections 2.3.2 and 2.3.2, respectively. An FPGA implementation further differentiates it from an ASIC. The eSSP can be viewed as a co-processor of sorts; however, unlike the co-processor mechanisms described in 2.3.2 it is implemented on the same die (chip) as the host general-purpose embedded processor. The implementation of the eSSP is outlined in Chapter 3.

Chapter 3

Implementation

This chapter presents the implementation of the Embedded Secure Software Platform (eSSP). The software components of the eSSP, the real-time Embedded Configurable Operating System (eCos) and the Secure ELF Generator (SELFGEN), are presented. The overarching system architecture is subsequently introduced at a high level of abstraction followed by detailed descriptions of the system's two primary components. The first component introduced is the Instruction Encryption Management Unit (IEMU). A detailed discussion of its Bridge Unit, Key Tables, Encryption Unit, and Controller Finite State Machine (FSM) follows. The second key component of the eSSP architecture, the Data Encryption Management Unit (DEMU), is next presented. The chapter closes with a subsequent analysis of the DEMU's Key Tables, Capture FSM, Process FSM, Data Translation Unit (DTU), and PLB-to-PLB bridge unit.

3.1 eCos

By virtue of its name, the eSSP necessitates the inclusion of an embedded Operating System (OS) in order to run the secure embedded applications on the embedded IBM Power PC

405 Processor (PPC405). The Real-Time Operating System (RTOS) selected to run on the eSSP is the eCos (embedded configurable operating system) [51] [52]. eCos is selected as the RTOS to run Secure ELF (SELF) executables on the eSSP platform because it is specifically designed to run as an embedded operating system with soft real-time scheduling. The eCos kernel is highly customizable with over 200 configuration points that can be included or removed for a given build. The ability to exclude unnecessary functionality in the kernel allows its memory footprint to be extremely small.

3.1.1 eCos Kernel Modifications

The kernel used to run on the eSSP was constructed using the default eCos kernel settings with the addition of device driver support for the ML310's Field Programmable Gate Array (FPGA) Universal Asynchronous Receiver/Transmitter (UART) as well as custom startup assembly code. In order for the eCos kernel to communicate with the FPGA UART, linkage code bridging the kernel generic UART driver to the specific memory-mapped ML310 FPGA UART driver was constructed. This involved modifying several eCos Hardware Abstraction Layer (HAL) files to have function pointers and data structures properly setup to communicate with the ML310 FPGA UART driver code. Moreover, custom PPC405 assembly listed in Section A.1 of Appendix A was added that set reserved bits of the PPC405 Core Configuration Register (CCR) [53]. The assembly code then disables all instruction page caching and data page caching. Finally, all instruction and data cache lines are cleared and control is returned to the eCos startup routines. This assembly code was added to make the PPC405 stable while executing eCos.

3.2 SELFGEN

A method for securing real-time embedded software applications is necessary if confidentiality of both instructions and data is to be achieved. Such a method is presented as SELFGEN,

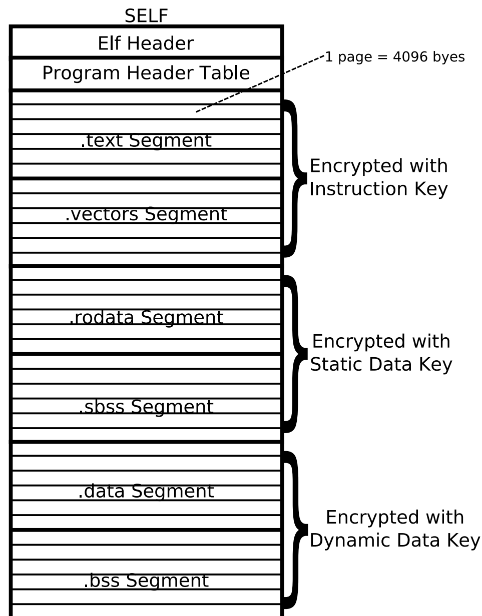

Figure 3.1: eCos SELF Layout

a Secure Executable and Linking Format GENerator. The Executable and Linking Format (ELF) is an open-source, extensible format used to store and structure executables, shared libraries, and object code [54]. Since ELF is an open-source format, it is favored in open-source consumer operating systems such as Linux as well as open-source embedded operating systems such as eCos. SELFGEN for embedded real-time systems is based off the original work of J. Edmison et al. [3]. The original SELFGEN only supports protection of the instruction segments of an ELF; however, protection of the data segments of an ELF is required for comprehensive application confidentiality. Therefore, the eSSP SELFGEN utility contains full instruction protection as well as full static and dynamic data protection.

Figure 3.1 shows the layout of a typical eCos monolithic kernel-user application SELF. The eSSP SELFGEN utility accepts an unprotected eCos monolithic kernel-user application ELF as its input. The utility then traverses the various segments of the ELF and encrypts the pages contained within each segment according to the characteristics of the segment. If the segment is marked as executable, a single 128-bit instruction key is used during the encryption process of that segment's pages. The particular encryption algorithm used to

encrypt both instructions and data within SELFGEN is the Advanced Encryption Standard (AES) Galois Counter Mode (GCM) elucidated further in Section 3.4.1. If the segment is marked as both data and read-only, the static data key is used to encrypt the pages of that segment. If the segment is marked as data and it is not marked as read only, the dynamic data key is used to encrypt the pages of that segment. Otherwise, as the encryption process continues, SELFGEN maintains a mapping of keys to pages in a custom header structure appended to the end of the ELF. This data structure is generally passed to the operating system, which updates hardware tables reflecting the keys to use for execution after a SELF has been loaded for execution. The result of the encryption process is a secured ELF (SELF).

In the eSSP SELFGEN implementation, the custom data structure listing page-to-key mappings featured is disabled. This mapping structure, generally speaking, allows for the segregation of the instruction and application sets of one process from the other processes in the system. As eCos is a single process multi-threaded operating system, there is no need for the segregation of processes, as only the monolithic kernel-application is running on the eSSP embedded system at a time. However, the framework exists both in the hardware implementation of the system as well as the eSSP SELFGEN utility to enable different instruction keys to be used to encrypt the pages of an executable segment of an ELF. Currently, the tagging scheme used to associate the instruction pages of a SELF to the data pages of that ELF, discussed in Section 3.4.2, only allows for a single static data key and a single dynamic data key to be used for the protection of data pages.

3.3 Platform

The development platform for the eSSP is the Xilinx ML310 Development Board. The ML310 is an embedded development platform featuring a Xilinx Virtex-II Pro XC2VP30-896-6C FPGA [55]. A summary of the resources available within the FPGA configurable fabric is provided in Table 3.1 [56].

RocketIO Transceiver Blocks	PowerPC Processor Blocks	Logic Cells	CLB		18b*18b Multiplier Blocks	BRAM		DCMs	Max User I/O
			Slices	Max Distr. RAM (Kb)		18 Kb Blocks	Max Block RAM (Kb)		
8	2	30,816	13,696	428	136	136	2448	8	644

Table 3.1: Resources available on the Xilinx XC2VP30 Virtex-II Pro FPGA

In addition to the FPGA, the Xilinx ML310 Development Board contains a plurality of FPGA-connected peripherals including a Double Data Rate (DDR) Synchronous Dynamic Random Access Memory (SDRAM) Dual Inline Memory Module (DIMM), a System ACE^{TM}CompactFlash Controller, a 10/100 Ethernet Network Interface Controller (NIC), a Joint Test Action Group (JTAG) module controller, various Peripheral Component Interface (PCI) slots, and an RS232 connection. As Table 3.1 indicates, there are two PowerPC 405-D5 (PPC405) Reduced Instruction Set Computer (RISC) Central Processing Unit (CPU) cores built into the configurable fabric of the XC2VP30. Each PPC405 core can run at 300 MHz and has a Harvard Architecture that physically separates the pathways for instruction and data [53]. The PPC405 Instruction Cache Controller (ICC) manages a 16 KB two-way set associative instruction cache, while the PPC405 Data Cache Controller (DCC) manages a separate 16 KB two-way set associative data cache with a five-stage data path pipeline. The Memory Management Unit (MMU) on the PPC405 contains a unified 64-entry Translation Look-aside Buffer (TLB), and the MMU supports variable page sizes. For the purposes of the eSSP implementation, a fixed page size of 4 KB is used. The minimal platform requirements for the eSSP are the Xilinx XC2VP30-896-6C FPGA including the two embedded PPC405 cores, the FPGA UART, and the DDR SDRAM with a 256-MB DIMM.

3.4 System Architecture

The implementation architecture of the embedded secure software platform appears in Figure 3.2. The primary components of the platform include two PowerPC 405 processors, an Instruction Encryption Management Unit (IEMU), a Data Encryption Management Unit (DEMU), and a DDR SDRAM Controller. One PowerPC 405 processor, the Secure Loading

Unit (SLU), is used to load a secured ELF (SELF) into external DDR SDRAM for execution by the second PowerPC 405 processor, the Secure Execution Unit (SEU). A secured ELF, or SELF, is an executable application whose instruction text and data segments are encrypted before execution. Details of the SELF generator application used to encrypt the data and text segments of an ELF are discussed in Section 3.2. The purpose of the IEMU is to provide just-in-time decryption of the instruction stream of a SELF as instructions from main memory are fetched from external memory for execution by the PPC405 SEU instruction cache controller (ICC). The DEMU is similarly responsible for providing just-in-time decryption of data arriving from main memory en route to the data cache controller (DCC) on the PPC405 SELF Execution Unit. Unlike the IEMU, the DEMU must perform just-in-time encryption of data returning from the PPC405 back to external memory. Instruction text is read-only, and consequently, the stream of instructions traveling through the IEMU is unidirectional. However, data is readable as well as writable, and thus, the data stream transmitted through the DEMU is bidirectional. This bidirectional data flow necessitates an architecture for the DEMU that is different than that of the IEMU. The architectures of both the IEMU and the DEMU are discussed further in Section 3.4.1 and Section 3.4.2, respectively, as well as the salient features of each architecture that facilitate the security requirements of the eSSP.

3.4.1 Instruction Encryption Management Unit

IEMU Bridge Unit

The block diagram illustrating the functional units of the Instruction Encryption Management Unit (IEMU) is shown in Figure 3.3. The IEMU implemented in the eSSP is adapted from [49]. The CPU PLB Bus Interface and the DDR SDRAM PLB Bus Interface prune the full International Business Machines (IBM) Processor Local Bus (PLB) specification [57] down to subset of signals that can be directly manipulated by the three state finite state machine (FSM) controller. Directly connecting the signals from the CPU interface to the corresponding signals in the memory interface forms a direct pass-thru bridge of signals

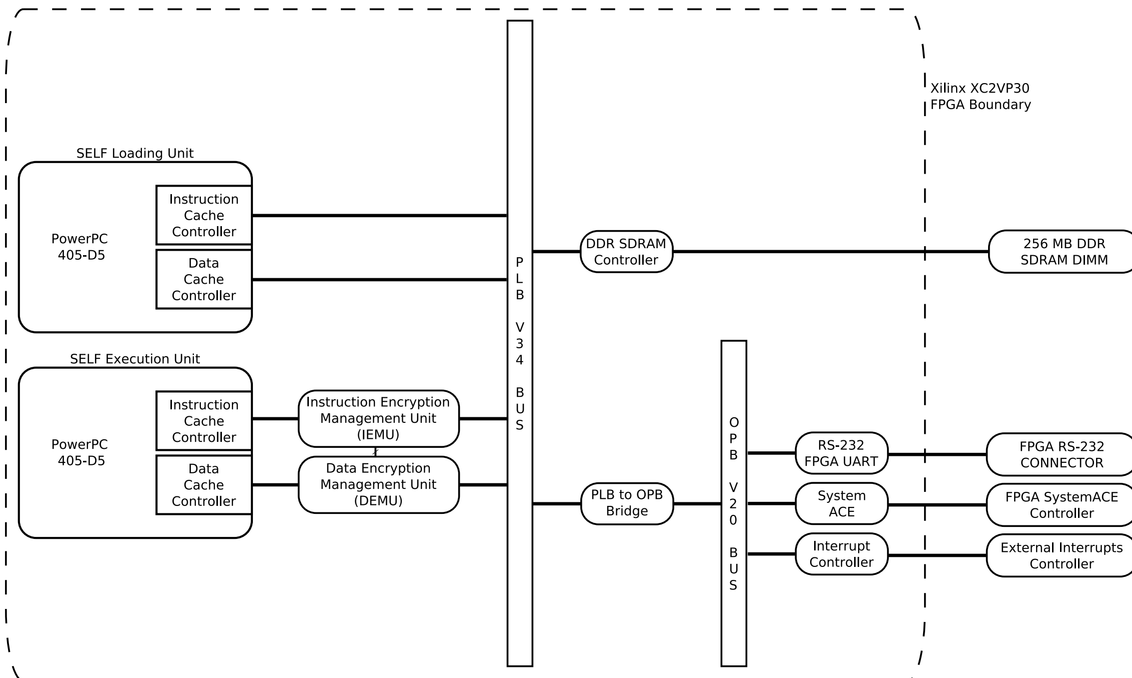

Figure 3.2: eSSP System Architecture

between the PPC405 SEU ICC and main memory.

IEMU Data Tables

The IEMU Data Tables are a key component to the IEMU architecture, as they provide the important data segments to encryption keys mapping. In a fully functioning multi-user, multi-tasking operating system such as Linux, the entries of the IEMU Page Table are updated by routines in the OS kernel. Since the PPC405's MMU contains a unified software-controlled translation look-aside buffer (TLB), both instruction and data page information are stored in the TLB [53]. Thus, whenever the OS encounters a virtual page fault caused by instruction or data while executing a process, a hardware interrupt is generated. The interrupt invokes the TLB miss handler Interrupt Service Routine (ISR) in the OS kernel. The TLB miss ISR first translates the virtual page address to a physical page address of memory. The ISR then updates the entries in the software TLB by inserting the faulted page and subsequently updates the hardware Page Table in the IEMU. Finally, the TLB

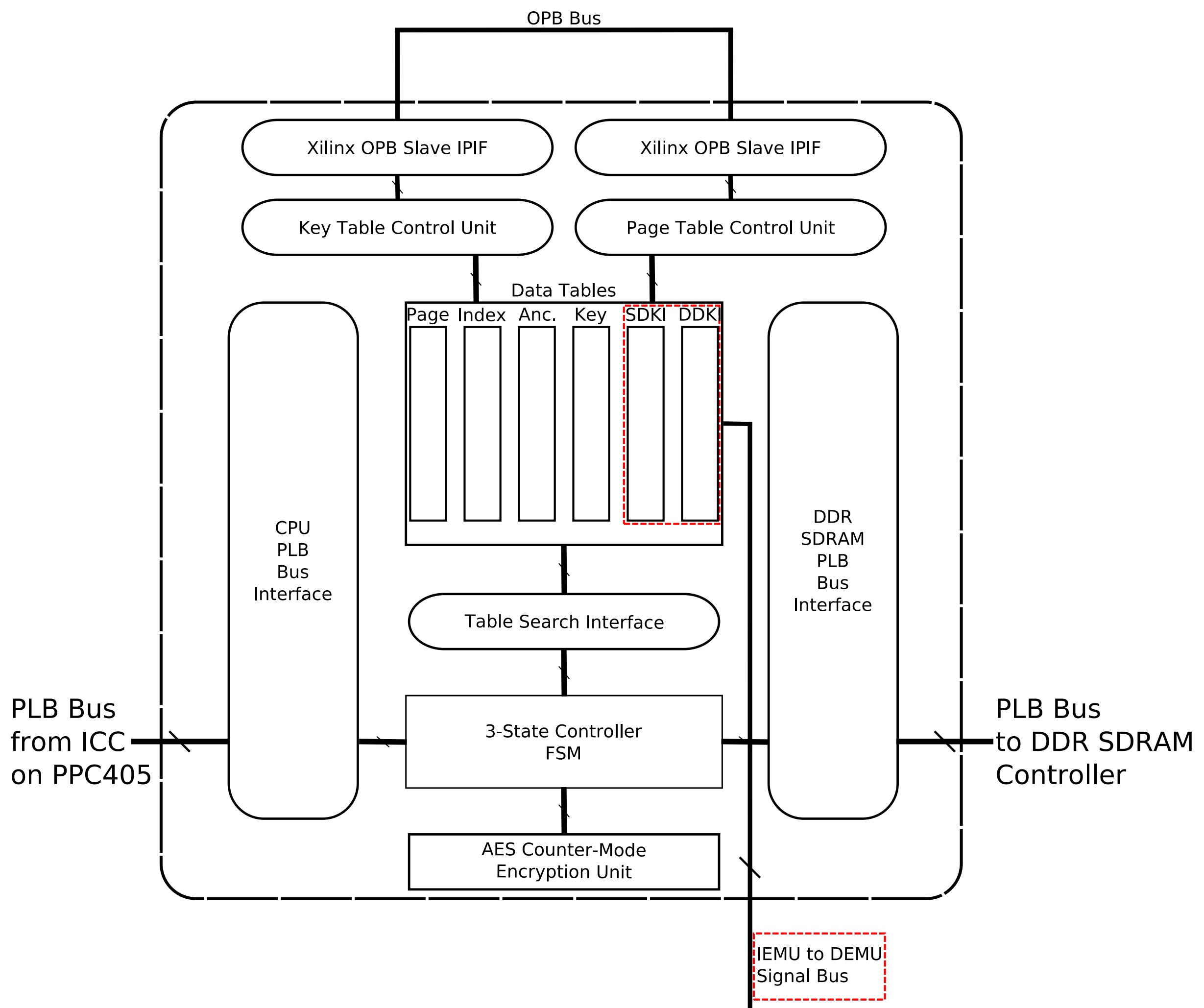

Figure 3.3: IEMU Architecture

miss ISR issues a request to memory for the data on the faulted physical page. In this way, the IEMU hardware Page Table is fully coherent with the software-controlled TLB.

The IEMU Page Table is used in conjunction with the IEMU Key Table and the IEMU Ancillary (AES Counter) Data Table to provide a page-to-key mapping for IEMU decryption purposes. During the SELF generation process detailed in Section 3.2, each virtual page (4096 bytes) of instruction text is encrypted using a 128-bit key and a 128-bit counter. These two pieces of information are required for the implementation of full AES Galois-mode encryption [58]. SELFGEN records which virtual pages of data are encrypted with each key-counter combination. The operating system uses this information to populate the IEMU Index data table when the SELF is loaded for execution. Routines within the operating system's kernel are responsible for translating the virtual page information of the SELF to physical page information and re-associating the keys and counters with physical pages. The IEMU Index table allows for a many-to-one mapping of physical pages to encryption keys. A one-to-one mapping between physical pages and counters is used. Figure 3.4 shows this relationship between pages, keys, and counters (Ancillary Data). This figure also shows the hardware implementation of the IEMU Data Tables in conjunction with the tables' search and write interfaces.

The IEMU Table Search Interface provides a means for the IEMU Controller FSM to query the IEMU Data Tables for any information relevant to the address of the transaction that it is currently processing. The address of the pending transaction is presented to the *iPageAddress* line of the IEMU Table Search Interface. This value is used as an input to a Content-Addressable Memory (CAM) [59]. The output of the CAM, *sMatchAddr*, is non-zero if the address is contained in an entry within the CAM. This output is used as the address line for both the IEMU Index Data Table as well as the IEMU Ancillary Data Table, which are implemented as a SelectRAM core and BRAM core, respectively. The output of the Ancillary data, *oAncillary*, is driven by *sMatchAddr*. The output of the IEMU Index data table is used as the address line to drive the output of the IEMU Key Table, *oKey*. The IEMU Key Table is implemented using a BRAM core. The last output of the IEMU

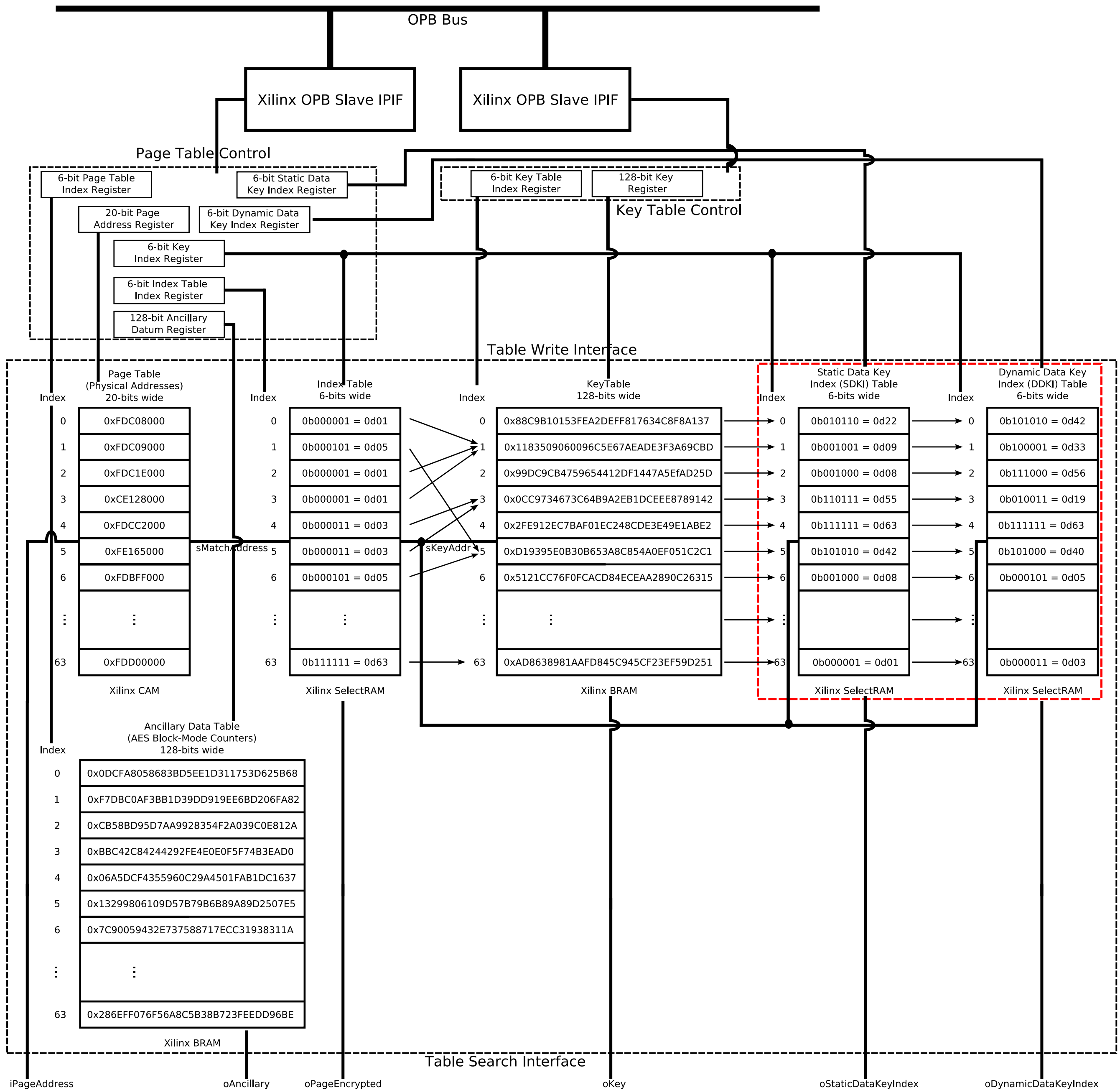

Figure 3.4: Page-to-Key and Page-to-Counter Mapping Hardware in IEMU Tables

Table Search Interface, *oPageEncrypted*, is constructed using a simple comparator circuit. If *sMatchAddr* is zero, then *oPageEncrypted* is set to a logic zero, else *oPageEncrypted* is set to a logic one.

The IEMU Table Write Interface provides a mechanism by which the IEMU Data Tables can be populated. As previously discussed, the TLB miss ISR in the OS kernel is responsible for updating the IEMU Data Tables, specifically the IEMU Page Table and the IEMU Index Table. The TLB miss ISR writes to the registers within the Page Table Control unit via the IBM On-chip Peripheral Bus (OPB) [60]. The Xilinx OPB Slave Intellectual Property Interface (IPIF) decodes the address and transaction lines of the OPB Bus and directs the data arriving on the OPB bus to the registers within the Page Table Control unit. There is a sequence of register write operations necessary in order to update the IEMU Page and Index tables and this sequence is detailed in [49]. The Key Table is updated independent of the PPC405 by a Secure Key Management Unit (SKU). One possible implementation of a SKU is outlined in [50]. For the eSSP, however, no SKU was implemented, and the reasoning behind this design choice is elaborated in Section 3.4.1.

IEMU to DEMU Bus

The two tables outlined by dashed lines in Figures 3.4 and 3.3 are additions to the original IEMU architecture. The IEMU Static Data Key Index Table and the IEMU Dynamic Data Key Index table are used to help associate a SELF's instruction and data pages, and they are implemented as SelectRAM cores. The complete tagging scheme associating instructions and data was developed by Edmison [3] and is elaborated upon in Section 3.4.2. The same *sKeyAddr* output of the Key Index Table that drives the read address lines of the Key Table also drives the address lines of the Static and Dynamic Data Key Tables. When an address is presented to the IEMU Page Table CAM, its output drives the address lines of the IEMU Index Data whose output in turn drives the IEMU Key, Static Data Key Index, and Dynamic Key Index address lines. The Table Search Interface contains

two signals, *oStaticDataKeyIndex* and *oDynamicDataKeyIndex*, that are the read output of their respectively named tables. These two signals are bundled together along with a tertiary signal, *sClockInNewIndexes*, and are transmitted to the Data Encryption Management Unit (DEMU) for processing as outlined in Section 3.4.2.

AES Galois-Mode Encryption

The cryptographic method selected for implementation in the IEMU is the Advanced Encryption Standard (AES) defined by the National Institute of Standards and Technology (NIST). The AES (Rijndael) encryption algorithm is selected for three primary reasons. Firstly, the algorithm is readily implementable in FPGA hardware [6]. Secondly, the Galois- or Counter-mode of operation is well suited for high-speed streaming data when compared to the other operational modes of AES such as CBC-MAC, CCM, EAX, OMAC, and CWC [58]. Thirdly, AES is the acknowledged encryption standard. GCM encryption is accordant with the streaming behavior of the instruction text of a SELF as the instructions pass through the IEMU while being executed on the PPC405 SEU. A 128-bit counter (random value) is encrypted with a 128-bit key using the block mode of AES encryption [61]. The encrypted counter is subsequently XOR-ed against a 128-bit portion of plain text to form the cipher text. Since AES is a symmetric cipher, the same 128-bit data key used to encrypt a given counter can be used during the decryption process. To decrypt a 128-bit block of cipher text, the 128-bit key used during encryption is again used to encrypt the 128-bit counter used during encryption to arrive at the encrypted counter value. Since the inverse operation of an XOR is itself an XOR, the encrypted counter is XOR-ed against the cipher text to reveal the plain text.

Controller FSM

The controller FSM orchestrates the IEMU, making use of the IEMU Data Tables, CPU and Memory Interfaces, and the AES Galois-Mode Encryption Unit. The finite state machine

begins in a state, *ST_Wait*, in which it waits for a request for instruction text from main memory to be presented on the CPU PLB Bus Interface from the ICC. If the transaction size of the request is less than 128-bits, the FSM up-converts the transaction to 128-bits, since GCM requires 128-bits of clear text or cipher text for proper operation. The types of PLB transactions that the IEMU supports are simple 64-bit data lines arriving in one (single), two (double), or four (quad) data beats.

After receiving a request from the ICC, the FSM enters into the *ST_Encrypt* state in which it queries the IEMU Data Tables for any information relating to the address presented in the pending read transaction. In this same state, the FSM passes the transaction qualifiers to the IEMU Memory PLB Interface to query the requested data from external memory in parallel to querying the data tables. The results of the query appear in one clock cycle, and the key and counter are directed into the AES Galois-Mode Encryption unit. The FSM waits for the results of the encryption to complete before it proceeds from the *ST_Encrypt* state to the *ST_Write* state.

In the *ST_Write* state, the *oPageEncrypted* signal is used as the control line to a multiplexer whose two inputs are a 128-bit zero string and the encrypted counter. If *oPageEncrypted* is true, then the encrypted counter is XOR-ed against the data fetched from memory, else zero is XOR-ed against the data fetched from memory. A property of the XOR operator is that $A \oplus 0 = A$, and thus if the page is not protected then the multiplexer will pass the 128-bit zero string through to be XOR-ed against the data, making the operation pass the data through with no translation. In addition to multiplexing the XOR value used in the GCM translation, the *ST_Write* state waits for the data to be returned from external memory and performs the XOR operation and finally returns the XOR-ed data. The FSM then returns to the *ST_Wait* machine for the next transaction from the PPC405 ICC. For further details on any portion of the IEMU's original architecture or implementation, refer to [49].

Modifications to IEMU

As discussed in Section 3.1, the eCos operating system is the real-time operating system used as the software component of the eSSP. Because eCos does not contain native memory management, the MMU hardware on the PPC405 SELF Execution Unit (SEU) is disabled. There is no notion of separate addresses spaces or virtual memory in eCos as there would be in other multi-user, multi-tasking operating systems such as Linux. The IEMU was originally developed to support the Linux multi-user multi-process operating system. Separate physical pages of memory could belong to any process on the system. As a result, the IEMU Data Tables originally included support for segregating the instruction and data segments of each SELF from the rest of the processes in the system, if desired. As eCos is a single process, multi-threaded operating system there is no need for segregation, as only the monolithic kernel-application is running on the eSSP embedded system at a time. Therefore, the entries of the IEMU Index table for the eSSP all point to a single 128-bit key in the IEMU Key Table, a single 128-bit counter in the IEMU Ancillary Data Table, a single static data key index in the IEMU Static Data Key Index Table, and a single dynamic data key index in the IEMU Dynamic Data Key Index Table. Generally speaking, this need not be the case. If eCos were ever to support a multi-process environment, the full feature set of the IEMU Data Tables could be exploited. All keys in the IEMU are statically coded into the hardware, as opposed to being loaded by a secure key management unit. Again, this need not be the case as hardware exists for a key management mechanism to dynamically load key values.

3.4.2 Data Encryption Management Unit

The block diagram illustrating the functional units of the Data Encryption Management Unit (DEMU) is shown in Figure 3.5. Though the DEMU is similar to the IEMU in that both must decrypt transactions passing through their data path, the DEMU has a fundamentally different implementation architecture than the IEMU. The set of PLB transactions that the IEMU must support is limited to read transactions of 64-bit line widths arriving in a

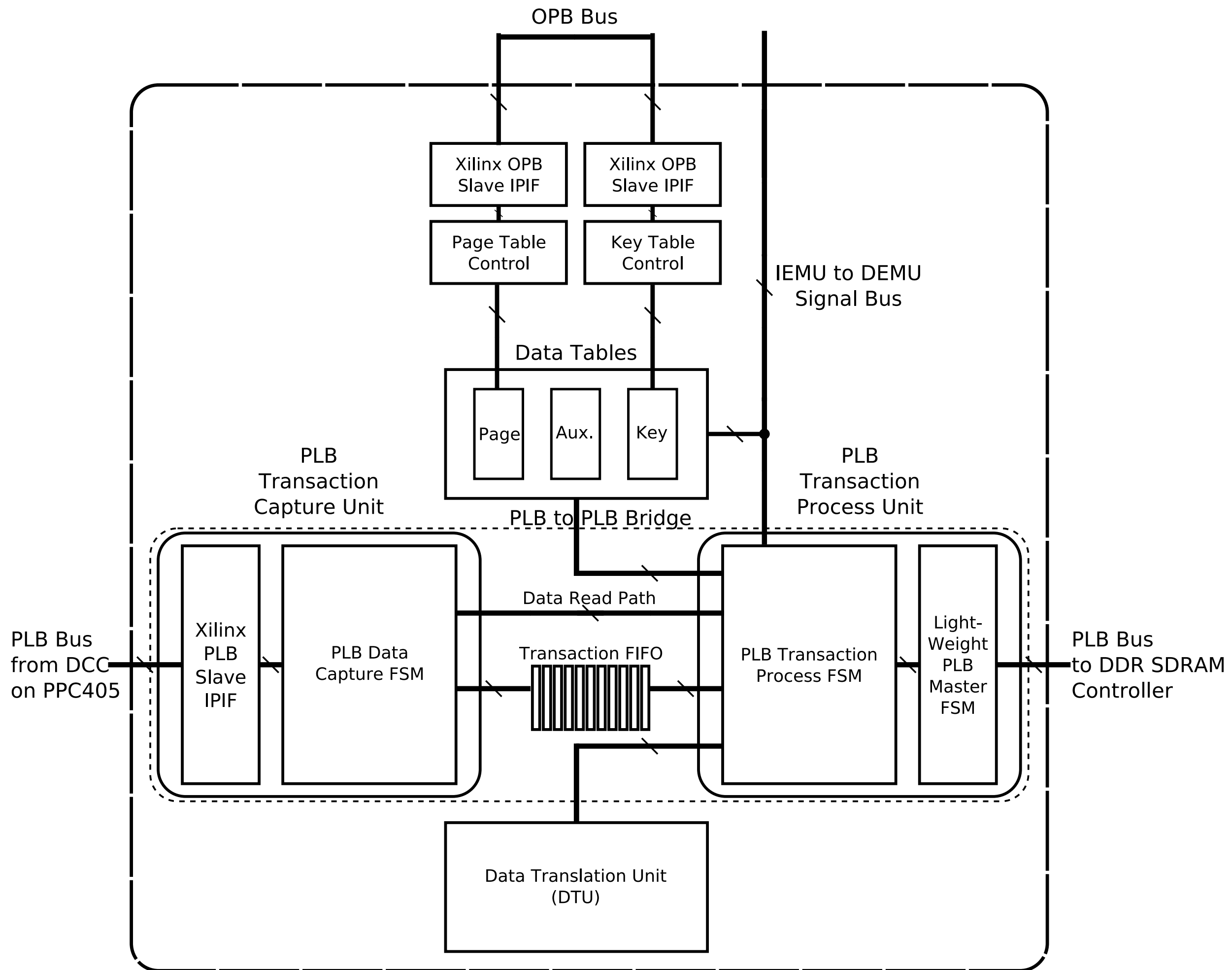

Figure 3.5: DEMU Architecture

single beat, double beat, or quad beat fashion. This set of PLB transaction types is only a small subset of the types of PLB transactions that the DEMU must support. The DEMU must support a wide range of burst transfers, pipelined back-to-back transfers, and locked transfers in both read as well as write mode. The fact that the DEMU must support write transactions makes it fundamentally different than the IEMU.

The primary functional component of the DEMU is the full PLB-to-PLB bridge. Unlike the bridge of the architecture in the IEMU, the PLB-to-PLB bridge of the DEMU is fully functional, conforming to the entire PLB specification on its slave interface. The slave

interface of the PLB-to-PLB bridge is implemented as the DEMU Data Capture Module (DCM). The DEMU DCM receives commands from the PPC405 SEU DCC that are queued into the DEMU Transaction First In First Out (FIFO) for execution by the master interface logic. The master interface of the DEMU PLB-to-PLB bridge is implemented as the DEMU Data Process Module (DPM). The DEMU DPM dequeues transactions in the FIFO and executes them, whether they are read or write transactions. The other important architectural features of the DEMU are the DEMU Data Tables and the DEMU Data Translation Unit (DTU). All abovementioned components of the DEMU implementation architecture are discussed in subsequent sub-sections of this chapter.

Transaction FIFO

The transaction FIFO is the central piece of the DEMU PLB-to-PLB bridge. The purpose of the FIFO is to provide an interface for the DEMU DCM to enqueue data for pending transactions and to provide an interface for the DEMU DPM to dequeue the pending transaction data. The FIFO is implemented as a LogiCORE FIFO [62]. According to [62], LogiCORE modules are fully optimized to take advantage of the underlying FPGA architecture of Xilinx's various device families. The FIFO was generated to have a depth of 64 entries and it was made dual ported, so that the write interface (push) and read interface (pop) could operate in a fully parallel fashion.

The LogiCORE FIFO was also generated to contain *Full*, *Empty*, *AlmostFull*, and *DataValidToRead* qualifier signals in addition to the standard *Push*, *Pop*, *DataIn*, and *DataOut* signals. The *AlmostFull* signal was programmed to go high when only when there were 49 or more entries in the FIFO. This decision to make 49 entries the threshold value for the *AlmostFull* signal was predicated upon knowing that the largest contiguous transaction that the Xilinx PLB Slave IPIF could present to the DEMU Data Capture Module would be a fixed-length burst transaction consisting of 16 single transactions. For this largest contiguous transaction to fit into the FIFO, 48 entries would be needed in the FIFO, and thus 49 entries was selected

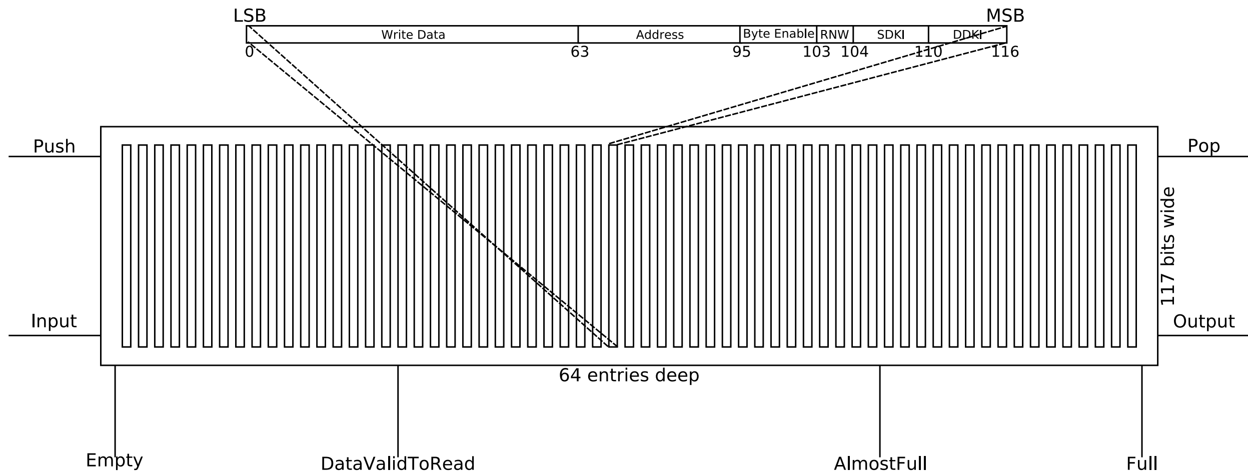

Figure 3.6: DEMU FIFO Interface Signaling and Fields Reference

as the threshold value for the *AlmostFull* signal. Figure 3.6 shows the width of the FIFO to be 117 bits of which 64 bits are reserved for data, 32 bits are reserved for address, 8 bits are reserved for byte enables, 1 bit is reserved for *RNW*, 6 bits are reserved for the Static Key Index, and the last 6 bits are reserved for the Dynamic Data Index.

Data Capture Module

The Data Capture Module shown in Figure 3.5 is comprised of two components: a Xilinx PLB Slave IPIF and a finite state machine. The purpose of the Xilinx PLB Slave IPIF [63] is to translate the full specification of PLB transaction types to a singular, uniform transaction type as well as communicate with the PLB bus to the PPC405 SEU DCC. The PLB Slave IPIF provides an abstraction layer needed to cross-communicate with the PLB using as few signals as possible. When the PPC405 SEU DCC initiates a transaction on the PLB bus connected to the PLB Slave IPIF, the IPIF intercepts the transaction and decodes the signals. From these decoded signals, it creates a simple transaction that consists of a target address (*Bus2IP_Addr*), target data (*Bus2IP_Data* for write transactions), byte enable signals (*Bus2IP_BE*), and the transaction type that is either read or write (*Bus2IP_RNW*). This simple transaction along with other various transaction qualifiers used to communicate

with the PLB bus are presented to the finite state machine for processing.

The finite state machine begins in a state after reset in which it waits for a transaction to arrive from the Xilinx IPIF slave. When a transaction arrives, an assertion of the *Bus2IP_RdReq* signal or *Bus2IP_WrReq* signal by the PPC405 SEU DCC indicates that a read transaction or write transaction, respectively, is pending on the DCM Xilinx PLB Slave IPIF. The transition to a next state depends on the *Bus2IP_Burst*, which if asserted indicates that the transaction is a burst or a multi-entry FIFO transaction. If *Bus2IP_Burst* is not asserted, a single entry FIFO transaction is pending for consumption by the FIFO. If the pending transaction is a write (*Bus2IP_WrReq*) and non-burst transaction (!*Bus2IP_Burst*) and there is a single entry available in the FIFO (!*Full*), then the transaction qualifiers, discussed in the DEMU FIFO Section 3.4.2, are sampled and placed into the queue. Simultaneously, the FSM acknowledges the completion of the transaction to the Xilinx PLB Slave IPIF that relays the write completion acknowledgement to the PPC405 SEU DCC. The same sequence of events occur in the case of a write request (*Bus2IP_WrReq*) with a burst transaction qualifier (!*Bus2IP_Burst*), except the FSM waits to sample and enqueue the transaction signals until there is enough space for the burst write transaction to be enqueued (!*AlmostFull*).

The sequence of events for enqueuing a read transaction is different than for enqueuing a write transaction. A write transaction can be immediately acknowledged as completed by the DEMU DCM because the PPC405 SEU DCC is not waiting for data during a write transaction; the DEMU DCC is the supplier of the data involved in a write transaction. During a read transaction, however, the PPC405 SEU DCC is waiting for the requested data to be returned from external memory. As such, any read requests must be serviced immediately by the DEMU DPM. Therefore during either a burst or non-burst read transaction, the DCM FSM does not immediately acknowledge the PPC405 SEU DCC because if it were to do so, the DCC would expect the returned data immediately. Thus, when there is space for either a burst read or a single beat data read transaction, the DCM FSM enqueues the transactions and effectively tells the PPC405 SEU DCC to wait for the results from the

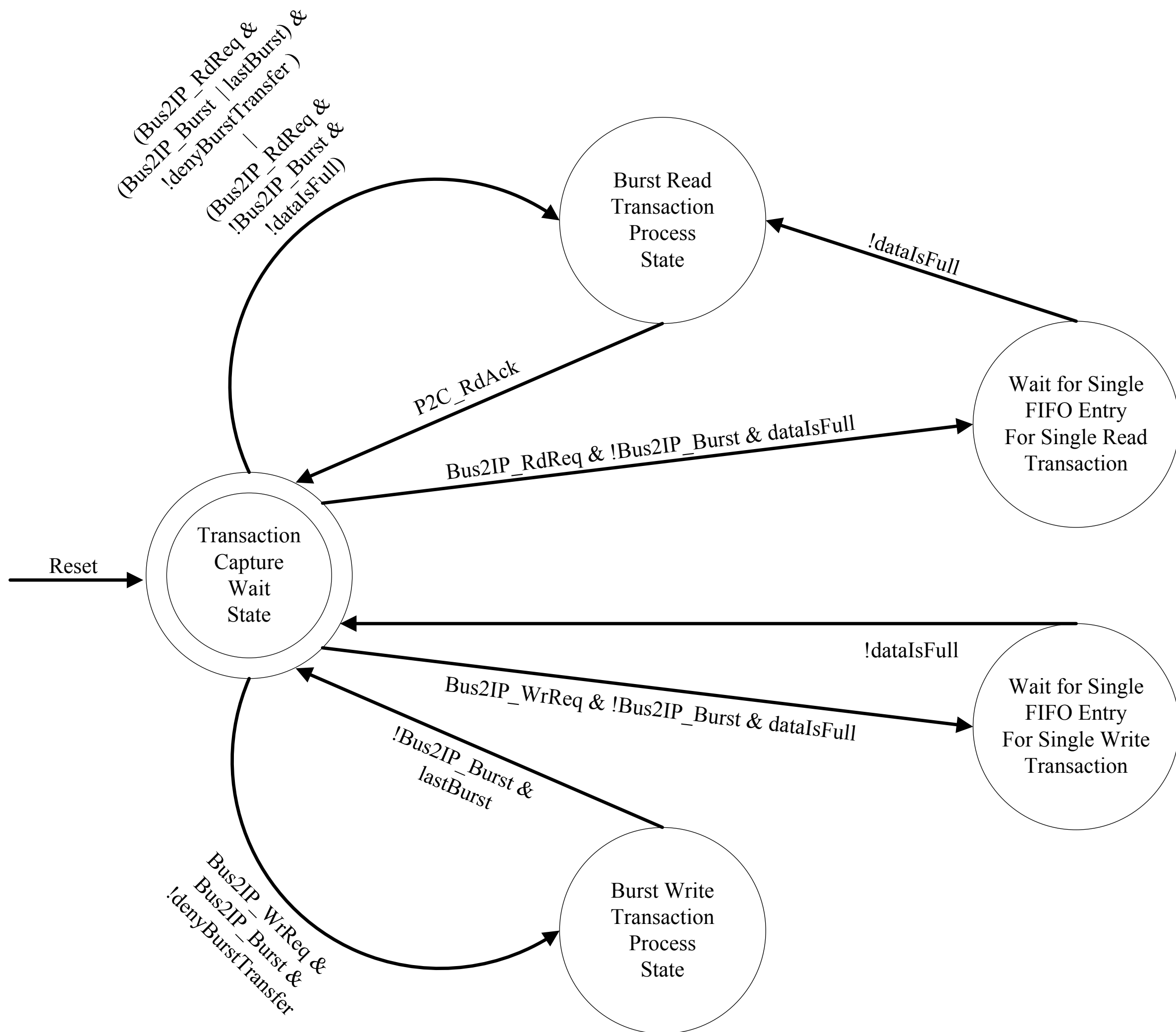

Figure 3.7: DEMU PLB Transaction Capture Unit Finite State Machine

Xilinx PLB Slave IPIF proxy. This wait continues until all transactions enqueued in the DEMU FIFO prior to the pending read request have been popped and processed by the DEMU DPM. The request is then forwarded to external memory and the result is returned to the DEMU DCM. Upon receipt of the data, the DCM acknowledges completion of the read request to the Xilinx PLB Slave IPIF module that delivers the acknowledgement and the returned data to the PPC405 SEU DCC.

Data Process Module

The DPM shown in Figure 3.5 is comprised of two components: a transaction process FSM and a lightweight PLB master FSM. The portion of the FSM outlined by the dashed lines is the set of states that define the PLB-to-PLB bridge portion of the DPM. The purpose of the transaction process finite state machine is to dequeue pending read and write transactions, encrypt protected data for writes, decrypt protected data for reads, and communicate with the lightweight PLB master FSM to perform the actual writes and reads to and from external memory.

The transaction process FSM begins by waiting for a transaction to arrive in the DEMU transaction FIFO. While the FIFO is not empty, the process FSM begins by popping a transaction off the DEMU FIFO and waiting for that data to setup on the DEMU FIFO output port. Once the data is valid to read, the process FSM progresses to either the read state if dequeued RNW value is a logic one, or to one of two write states. If the RNW is a logic zero and the page is protected, then a transition occurs from the branch state to a table search state; otherwise, a transition occurs to a state in which a plaintext write occurs via the lightweight PLB master write FSM. A page is considered to be protected if it falls within the memory-mapped range of external memory, it has a non-null Dynamic Data Key Index, and it is marked as protected in the DEMU Supplementary Data Table described in Section 3.4.2. If the page referred to in the dequeued transaction address is protected, a transition occurs from the Branch State to a Table Search State. This DEMU

table search request queries the DEMU Data Tables for all data related to the address page of the pending transaction. Once the results of the table have returned and setup, the data to be written out to external memory and the Dynamic Data Key are loaded into the Data Translation Unit (DTU) for encryption purposes. The DTU encrypts the data with the key and signals that the data on its output port is valid for reading by asserting its done signal. The process FSM waits for this signal before it continues onward to the next state.

At this point in the process FSM, the paths of the encrypted and plain text write converge to the lightweight PLB master write FSM. This finite state machine implements the PLB Master Write protocol shown in Figure 3.9. The proper transaction qualifiers are presented to the PLB bus connected to the external DDR SDRAM controller. The PLB master write FSM waits for the address acknowledgement from external memory (PLB_MAddrAck) and then waits for the write completion acknowledgement from external memory (PLB_MWrDAck) before returning to the process FSM start state (wait state).

The sequence of state transitions for a pending read transaction are different than those of a pending write transaction. After the FSM recognizes the presence of request in the DEMU FIFO, the transaction is popped off and the FSM waits one clock cycle for the data on the DEMU FIFO's output port to setup. Recognizing that a read transaction was popped from the FIFO involves the branch state checking the popped RNW value. If this value is a logic one, then the transaction is a read transaction and a transition in the process FSM from the branch state to the lightweight PLB master read FSM occurs. This FSM implements the PLB Master Read protocol shown in Figure 3.10. The proper transaction qualifiers are presented to the PLB bus connected to the external DDR SDRAM controller. The PLB master read FSM waits for the address acknowledgement from external memory (PLB_MAddrAck) and then waits for the read completion acknowledgement from external memory (PLB_MRdDAck). A transition then occurs from the lightweight PLB read FSM back into the process FSM to another branch state. This branch state decides whether the data just returned from external memory is returned immediately with no translation (a plain text read) or is returned after translation (a decrypted read). The qualifications

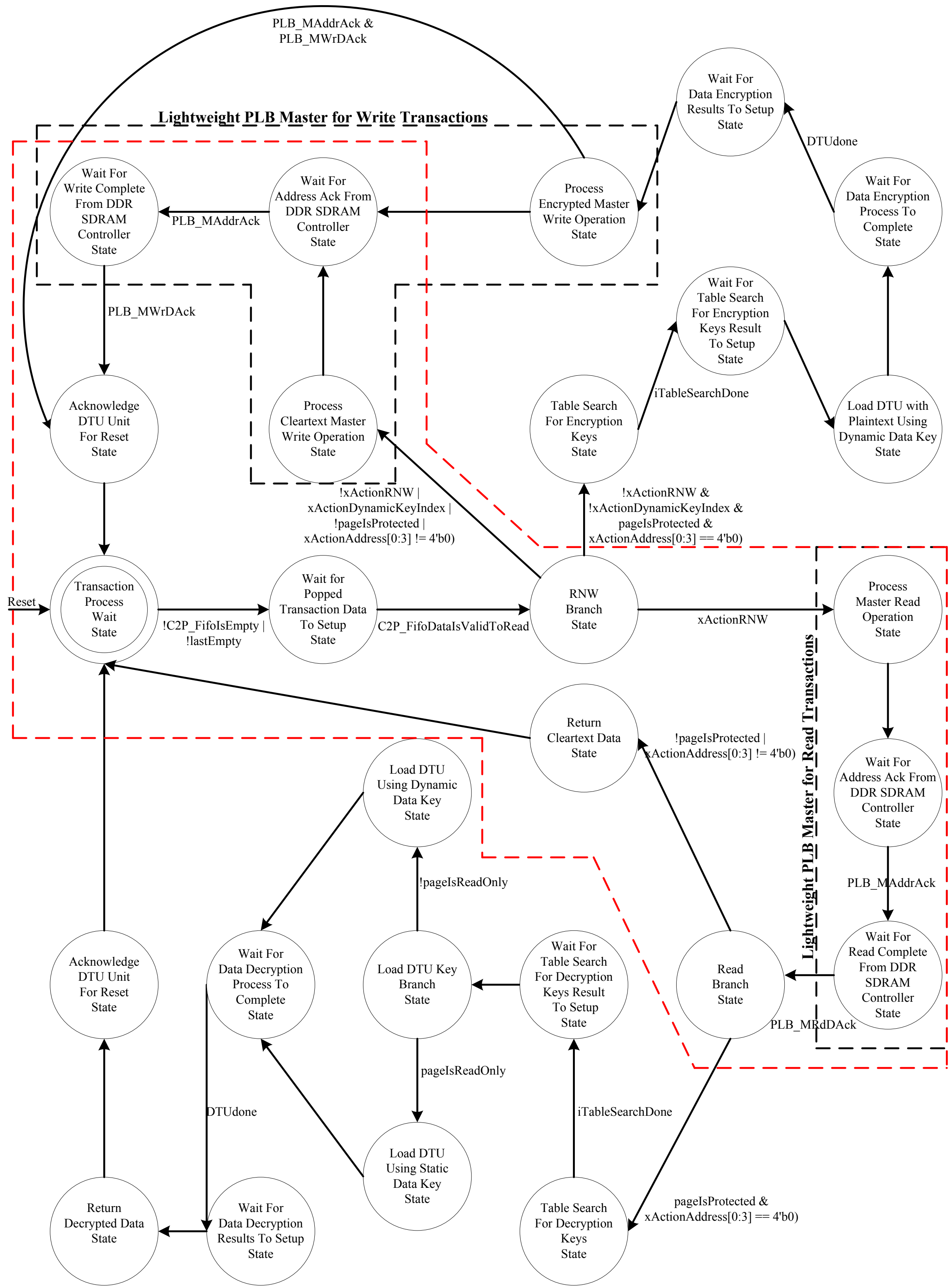

Figure 3.8: DEMU PLB Transaction Process Unit Finite State Machine

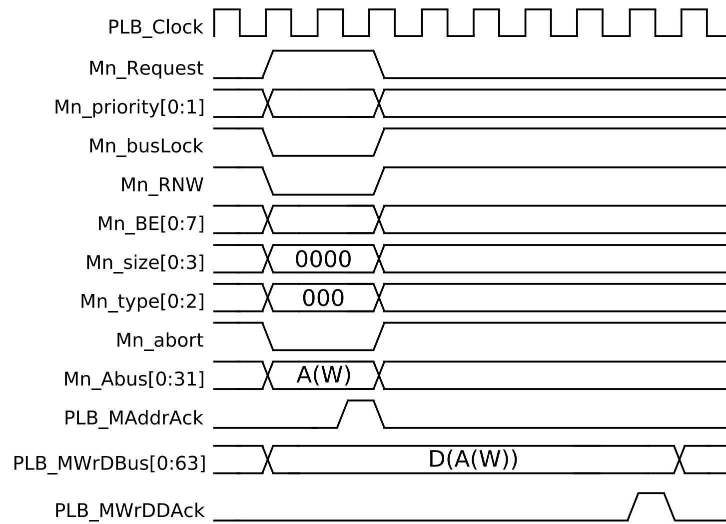

Figure 3.9: PLB Write Transaction Timing Protocol

necessary for a decrypted read are the page address of the pending read transaction is within the memory-mapped range of external memory, and the page is marked as protected in the DEMU Supplementary Data Table. If either of these conditions are false, then the data is returned immediately to the Data Capture Module FSM.

If both conditions are true, however, a transition occurs from this protected/not-protected branch state to a state in which the DEMU data tables are searched for all data related to the address page of the pending read transaction. Once the data from the table search returns, a transition to yet another branch state occurs. The reason for the existence of this third branch state in the process FSM is to distinguish read-only data from dynamic data. If the data were dynamic data and had been written before, it would have been encrypted with the Dynamic Data Key in the write portion of the process FSM. If the *pageIsReadOnly* field of the DEMU Supplementary Data is set to a logic one, then the page address refers to a page of memory that is marked as read-only and the DTU is loaded with the data read in from external memory and the Static Data Key. If the *pageIsReadOnly* field of the DEMU Supplementary Data is set to a logic zero, the page address refers to a page of memory that

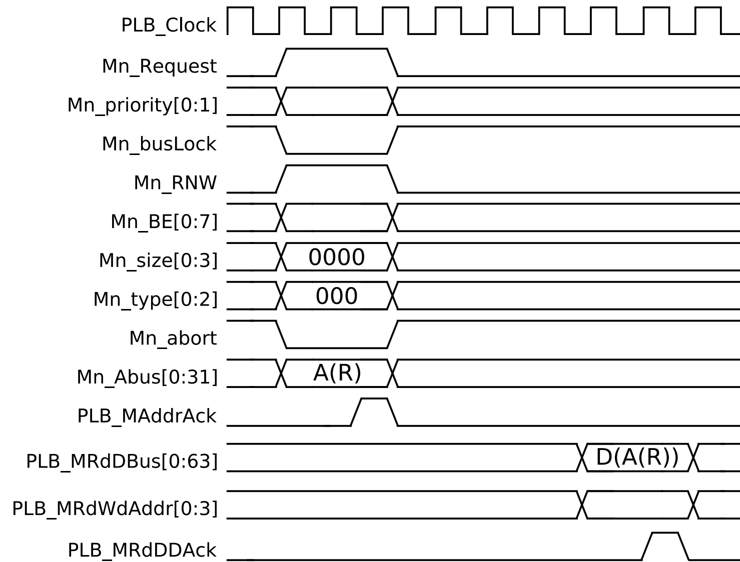

Figure 3.10: PLB Read Transaction Timing Protocol

is marked as read/write and the DTU is loaded with the data read in from external memory and the Dynamic Data Key. The Data Translation Unit then decrypts the data that is subsequently returned to the DEMU DCM FSM for routing to its ultimate destination, the DEMU SEU DCC.

DEMU Data Tables

The premise behind the DEMU Data Tables is the same as that behind the IEMU Data Tables characterized in Section 3.4.1: to store cryptographic and other pertinent information related to a page (4096B) of external memory on-chip for consumption by other on-chip hardware modules. Because the software-controlled TLB on the PPC405 SEU is unified, it contains entries of both instruction pages as well as data pages. Therefore, the DEMU Page Table exists in an effort to mirror the software-controlled TLB. Whenever an entry of the software-controlled TLB is updated by the TLB miss handler ISR, the ISR also updates the DEMU Page Table by writing the faulted entry to the registers inside the DEMU Page Table Control Unit.

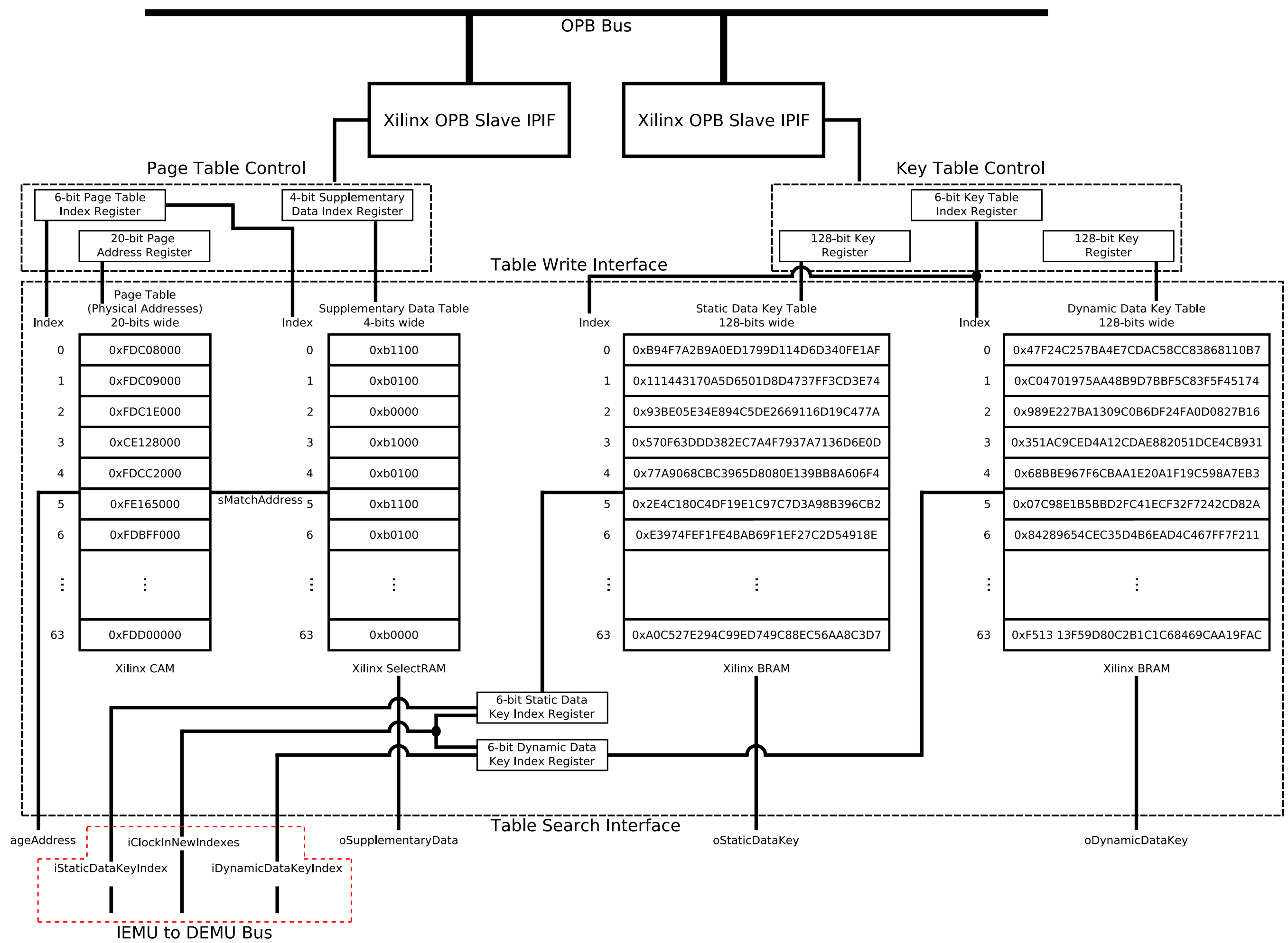

Figure 3.11: Page-to-Key Mapping Hardware in DEMU Tables

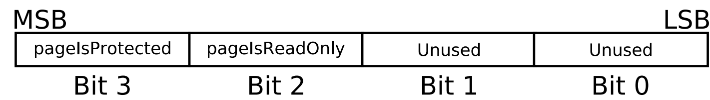

Figure 3.12: DEMU Supplementary Data Table Fields Reference

The Page Table is the only common table found in both the IEMU and DEMU Data Tables. Aside from the DEMU Page Table, the DEMU Data Tables contain the Supplementary Data Table, the Static Data Key Table, and the Dynamic Data Key Table. As Figure 3.12 indicates, the DEMU Supplementary Data Table is 4-bits wide. The two least significant bits of the table remain unused. The most significant bit of the table is designated as the *pageIsProtected* bit of information and the next significant bit of the table is designated as the *pageIsReadOnly* bit. A page is marked as read only by setting the *pageIsProtected* bit to a logic one, otherwise, it should be interpreted as an unprotected page of data. Similarly, if a data page is to be designated a read-only page, then the *pageIsReadOnly* bit of the entry relating to page in question should be set to a logic one.

The Static Data Key table and the Dynamic Data Key tables contain the data keys that are associated with a protected SELF's instruction keys. This association is necessary in order to link data pages to instruction pages. This relation effectively correlates instruction and data to form a coherent protection boundary of a SELF. The DEMU implements a data tagging scheme developed in [3]. The conceptualization of this scheme is illustrated in Figure 3.13. The scheme involves tagging a set of instruction keys with a value denoting the associated static data key and a value denoting the associated dynamic data key. The modifications to the original IEMU architecture, the IEMU Static Data Key Index Table and IEMU Dynamic Data Key Index Table, provide the means for the association in the eSSP. When an IEMU Table Search request is finished, the *sClockInNewIndexes* IEMU to DEMU Bus signal is asserted. This signal causes the registers sourcing the read address lines to the DEMU Static and Dynamic Data Key Tables to clock in the current values of the *oStaticDataKeyIndex* and *oDynamicDataKeyIndex* that originate as the results of a IEMU Data Table search. In this manner, the IEMU is able to communicate to the DEMU which

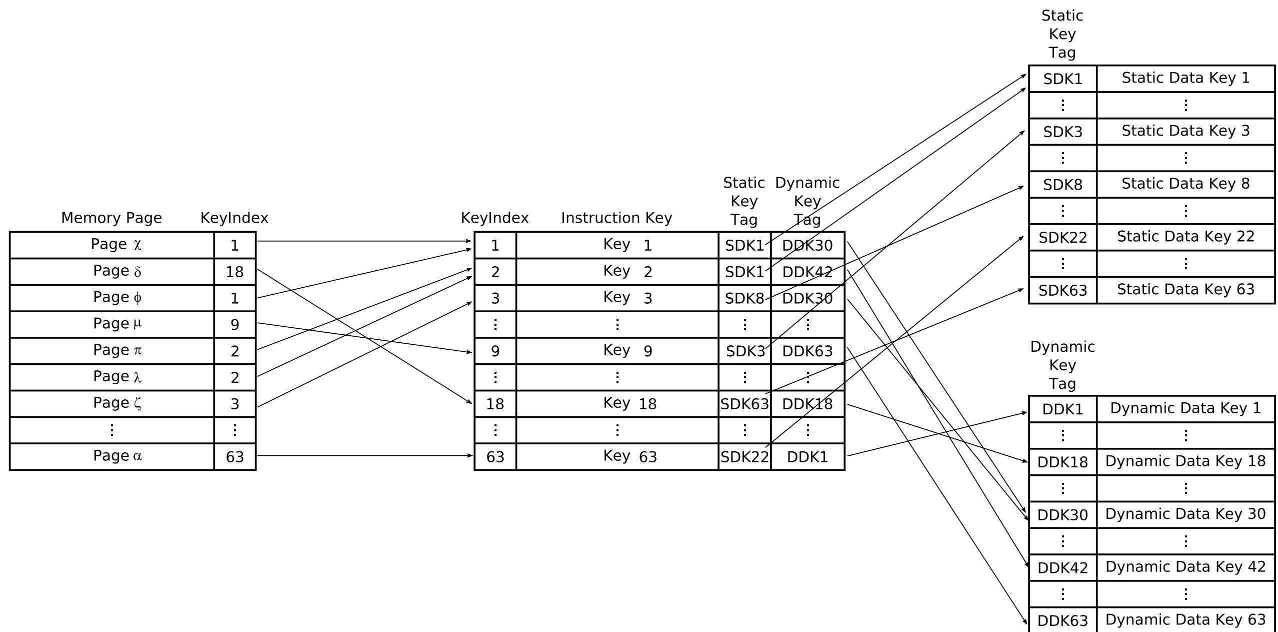

Figure 3.13: Data Tagging Scheme

application is currently running and therein which set of data keys should be used.

The DEMU Table Search Interface provides a means for the DEMU process FSM to query the DEMU Data Tables for any information relevant to the address of the pending transaction. The address is presented to the *iPageAddress* line of the DEMU Table Search Interface. This value is used as an input to a Content Addressable Memory (CAM) [59]. The output of the CAM, *sMatchAddr*, is non-zero if the address is contained in an entry within the CAM. This output is used as the address line for the DEMU Supplementary Data Table, which is implemented as a SelectRAM core. The search result outputs of the DEMU Static and Dynamic Data Key Tables are driven by registers that sample the *oStaticDataKeyIndex* and *oDynamicDataKeyIndex* when *sClockInNewIndexes* is asserted high. A graphical representation of the DEMU Data Tables hardware is displayed in Figure 3.11.

The DEMU Table Write Interface provides a mechanism by which the DEMU Data Tables

can be populated. As previously discussed, when a TLB miss interrupt occurs the TLB miss ISR in the OS kernel is responsible for updating the DEMU Data Tables, specifically the DEMU Page Table and the DEMU Supplementary Data Table. The TLB miss ISR writes to the registers within the Page Table Control unit via the IBM On-Chip Peripheral Bus (OPB Bus) [60]. The Xilinx OPB Slave Intellectual Property Interface (IPIF) decodes the address and transaction lines of the OPB Bus and directs the data arriving on the OPB bus to the registers within the Page Table Control unit. The sequence of register write operations necessary in order to update the IEMU Page and Index tables is examined in [49]. For the eSSP, every entry in the DEMU Static Data Key Table is populated with the same static key value. Likewise, every entry in the DEMU Dynamic Data Key Table is populated with the same dynamic data key value. All keys are statically coded into the Random Access Memory (RAM) hardware, as opposed to being loaded by an SKU. The lines of reasoning behind this design decision parallel those discussed in Section 3.4.1.

Data Translation Unit

The DEMU Data Translation Unit serves to encrypt and decrypt data using the same AES Galois-Mode encryption unit described in Section 3.4.1. Another physical instantiation of the AES GSM module is created so as not to create contention between the IEMU and DEMU for control of a cryptographic module when processing protected transactions. The interface to the DTU is generic enough that any cryptographic algorithm implementation can be used instead of AES GCM.

Chapter 4

Results

This chapter presents a characterization of the Data Encryption Management (DEMU) of the embedded secure software platform (eSSP). As the DEMU lies architecturally between the embedded PPC405 processor and main memory, it adds extra cycles of latency to main memory read and write transactions. The purpose of the characterization is to show the latency of the DEMU under various eSSP operating conditions. The first scenario is a hardware system containing neither a DEMU nor an IEMU, used as base profile against which application runtimes involving EMUs can be compared. The second scenario is a hardware system containing only a DEMU that is in *pass-thru* mode of operation, in which the data is unaffected. This condition characterizes the latency that occurs by inserting the DEMU into the PPC405 DCC data path. The last scenario is a hardware system containing only a DEMU that is in *encrypt/decrypt* mode of operation, transforming data of an encrypted SELF at the processor boundary. This condition characterizes the extra latency that the encryption hardware incurs beyond the PLB-to-PLB bridge. These metrics show the general effect the DEMU has on performance, since latency and performance slowdown are directly related.

Typical open-source benchmarking suites such as *LMBench* [64] and *miBench* [65] are implemented using Portable Operating System Interface (POSIX)-compliant operating system calls, despite containing operating system agnostic algorithms. This precludes their use on

the eSSP platform that contains eCos, as eCos is not a POSIX-compliant OS. In addition, no benchmarking suites were found that target the eCos operating system. As a result, custom benchmarks are created to reveal the latency of the DEMU as it performs write operations to memory and read operations from memory. These benchmarks are detailed further in Section 4.1. The decision to characterize only the performance of the DEMU hardware in the absence of IEMU hardware is predicated upon prior existing performance characterizations of the IEMU in [49] and the SSP system in [3]. It should be noted that the latency measurements described in this chapter should not be construed as actual application performance, since the benchmarks constructed are artificial in nature and serve only to provide a general upper bound on the degradation of performance due to the DEMU.

4.1 Methodology

The source code for the benchmarks constructed to expose the latency of DEMU read and write transactions is listed in Sections A.2, A.3, and A.4 of Appendix A, and are adapted from a benchmark, *lat_mem_rd*, found in the *LMBench* suite [64]. The *LMBench lat_mem_rd* benchmark uses a circularly linked list of nodes constructed from the entries of an array. Each node of the linked list is a pointer to the next node in the list with the last node in the list pointing to the first. The nodes of the list are separated in the array by *stride_size* widths of memory, and the total size of the array is *array_size*. This implies that there are exactly $\left\lfloor \frac{array_size}{stride_size} \right\rfloor$ nodes in the linked list than can fit into the array. The benchmark issues one-hundred sequential *load* instructions that walk the linked list structure inside a loop that iterates *num_100s* times, assuring that the linked list is traversed at least once. The value of *num_100s* is given by

$$num_100s = \left\lfloor \frac{array_size}{stride_size * 100} \right\rfloor + 1. \tag{4.1}$$

This loop of one-hundred *load* instructions is iterated over a range of *stride_sizes* defined

by the recurrence relation

$$stride_size_{k+1} = \begin{cases} stride_size_k * 2 & \text{if } 0 \leq stride_size_k < 1024 \\ stride_size_k + 1024 & \text{if } 1024 \leq stride_size_k < 4096. \\ stride_size_k + 2^{11} \prod_{i=15}^{i<=log_2(k)} 2 & \text{if } stride_size_k \geq 4096, \end{cases} \quad (4.2)$$

assuming that $\prod_{i=n}^{m} |n > m = 1$. The *stride_size* loop is in turn iterated over various memory sizes that adhere to the same stepping function defined in Equation 4.2. The average latency over all memory sizes incurred at each $stride_size_i$ is calculated and graphed as a function of *stride_size*. The premise behind varying the stride size and the overall array size is to reveal the latencies embedded at the various levels of the memory hierarchy. For small memory sizes, the entire circularly linked list can fit into the L1 data cache of the PPC405. Therefore, the loop that iterates through the list can execute quickly, with negligible latency since most accesses end up as L1 data cache hits. However, as the array continues to grow or the stride size becomes large, access to memory while traversing the linked list will turn from L1 data cache hits to L1 data cache misses. This manifests itself as the second plateau of latency shown in Figure 5.1 of [49]. Finally, the last plateau of operation is created when the array or stride size becomes large enough to cause a miss in the unified TLB of the PPC405, requiring an access to external memory. In this way, the *lm_mem_rd* benchmark profiles the entire memory hierarchy, including on-chip cache latency, external cache latency, TLB miss latency, and main memory latency.

To tailor the general algorithm of *lat_mem_rd* for the eSSP platform, several changes are made that result in the code listing *lat_mem_ecos* found in Appendix A.2. Firstly, because eCos does not support memory management, the MMU on the PPC405, the L1 Data Cache, and the L1 Instruction cache were all disabled. This implies that the expected latency measurements taken of the baseline operating condition or any of the operating modes of the DEMU should be a flat line. The plateau effect only manifests if there is an actual memory hierarchy; disabling the MMU and caches effectively removes the hierarchy, resulting in a flat

line of latency. Because there is only one latency time in any operating condition targeted, only one size of data memory is considered in the *lat_mem_ecos* benchmark. The *lat_mem_rd* benchmark seeks to characterize the average latency behavior of a system at each level of that system's memory hierarchy, but with no memory hierarchy on the eSSP, the memory size is kept constant in the benchmark *lat_mem_ecos* while varying the stride size. Even for varying stride sizes, the expected variance between measurements of latency should be relatively constant for a given operating condition because of the removal of caches. The one size of memory selected for benchmarking purposes is 8MB, which is the upper bound on memory considered by *LMBenchlat_mem_rd*.

In order to tailor the *lat_mem_rd* algorithm to the eCos operating system and the eSSP platform, the PPC405 high-performance timer hardware was used. The general *lat_mem_rd* algorithm is specifically implemented for any POSIX-compliant operating system such as Unix or Linux. Therefore, it makes Linux- or Unix-compliant system calls, especially those involving timing such as the *gettimeofday()*. There is no such refined timing infrastructure provided by eCos. Instead, custom assembly involving the high-performance PPC405 timer listed in Appendices A.3 and A.3 is used to benchmark the traversal of the circularly linked list to obtain read and write latency measurements. The high-performance PPC405 timer hardware is exposed to software through special registers, Timebase Upper (TBU) and Timebase Lower (TBL). Timebase lower is incremented every clock cycle by the PPC405 clock, and when it wraps after 2^{32} clock cycles, Timebase Upper is incremented. These registers are captured and stored to a known range in external memory to collect the start time and the ending time of the benchmarked read loop. For further reading on the PPC405 timer hardware, refer to [53].

4.2 Latency

A methodology for computing the average latency of DEMU hardware for read and write accesses is necessary if the latency is not directly measured but rather is calculated indirectly

from benchmark execution times. Generally speaking,

$$benchmark_execution_time = benchmark_instruction_execution_time + memory_latency_time. \quad (4.3)$$

Thus, the equation to calculate the memory latency value based on a benchmark's execution time is

$$memory_latency_time = benchmark_execution_time - benchmark_instruction_execution_time. \quad (4.4)$$

The instruction execution times depend specifically on the number of instructions benchmarked. In the case of the benchmark developed in this thesis, *lat_mem_ecos*, the time in cycles it takes to execute the instructions in the benchmarked loop listed in Appendix A is equal to the number of instructions contained in the loop

$$benchmark_instruction_execution_time_in_cycles = 6 + \left((100 + 3) \left(\left\lfloor \frac{memory_size}{stride_size * 100} \right\rfloor + 1 \right) \right). \quad (4.5)$$

This equation assumes that one instruction can be executed every cycle, which is the case for this benchmark and the PPC405 processor. The PPC405 documentation [53] states that one instruction is executed every cycle with the exception of some multiply and divide instructions that may take more than one cycle. The benchmarked loop contains only *load*, *store*, *decrement*, *compare*, and *branch* instructions that can each be executed in one clock cycle. The one-hundred *loads* used for read latency benchmarking or the one-hundred *stores* used for write latency benchmarking dominate the execution time and fetching time of the benchmark, making the benchmark good for measuring average memory latency. The first six cycles shown in Equation 4.5 refer to the three instructions executed after the lower timebase register is captured for the start time and the three instructions executed before the lower time base is captured for the ending time. The second term of Equation 4.5 is derived by noting that the benchmarked loop contains one-hundred *load* or *store* instructions with three

additional instructions used to control loop iteration yielding a total of 103 instructions in the loop. The loop is iterated $\left\lfloor \frac{array_size}{stride_size*100} \right\rfloor + 1$ times, and this value is multiplied by 103 to arrive at the total execution time in cycles of the benchmarked *load* instructions that are used to discern the read and write latency of the DEMU for various operating conditions. Because the benchmarked loop iterates a variable number of times based on the *stride_size*, each latency measurement taken must be normalized against the number of memory accesses of the loop, $100 \left\lfloor \frac{array_size}{stride_size*100} \right\rfloor + 1$. The final equation used to compute average latency is

$$avg_memory_latency_time = \frac{benchmark_execution_time - \left(6 + \left((100+3)\left(\left\lfloor \frac{memory_size}{stride_size*100} \right\rfloor + 1\right)\right)\right)}{100 \left\lfloor \frac{array_size}{stride_size*100} \right\rfloor + 1}. \tag{4.6}$$

4.2.1 Read Latency

The average memory latency equation is used in conjunction with raw benchmark execution times obtained for the three eSSP operating conditions to construct the average read latency trends as a function of stride size depicted in Figure 4.1. The figure shows three distinct straight lines, as expected by the absence of the PPC405 MMU hardware and L1 cache. The bottom-most line represents the baseline system configuration containing no EMU units. The average read latency time for this configuration is 240 ns and this accordant with the average UTLB miss value of 250 ns observed by Mahar in [49]. The middle line of the graph represents the latency induced by inserting the DEMU in *pass-thru* mode into the data path between the PPC405 DCC and memory. The average read latency for this eSSP operating mode is 444 ns. This implies that the addition of the *pass-thru* DEMU module introduces $444ns - 240ns = 204ns = 21$ cycles at 100MHz of extra latency beyond the average base read latency.

These 21 cycles of latency are explained by following the data flow through the DEMU Data Capture and Data Process Module FSMs. The transaction first arrives from the PPC405 DCC and travels through the PLB Slave IPIF logic, consuming 7 cycles of the

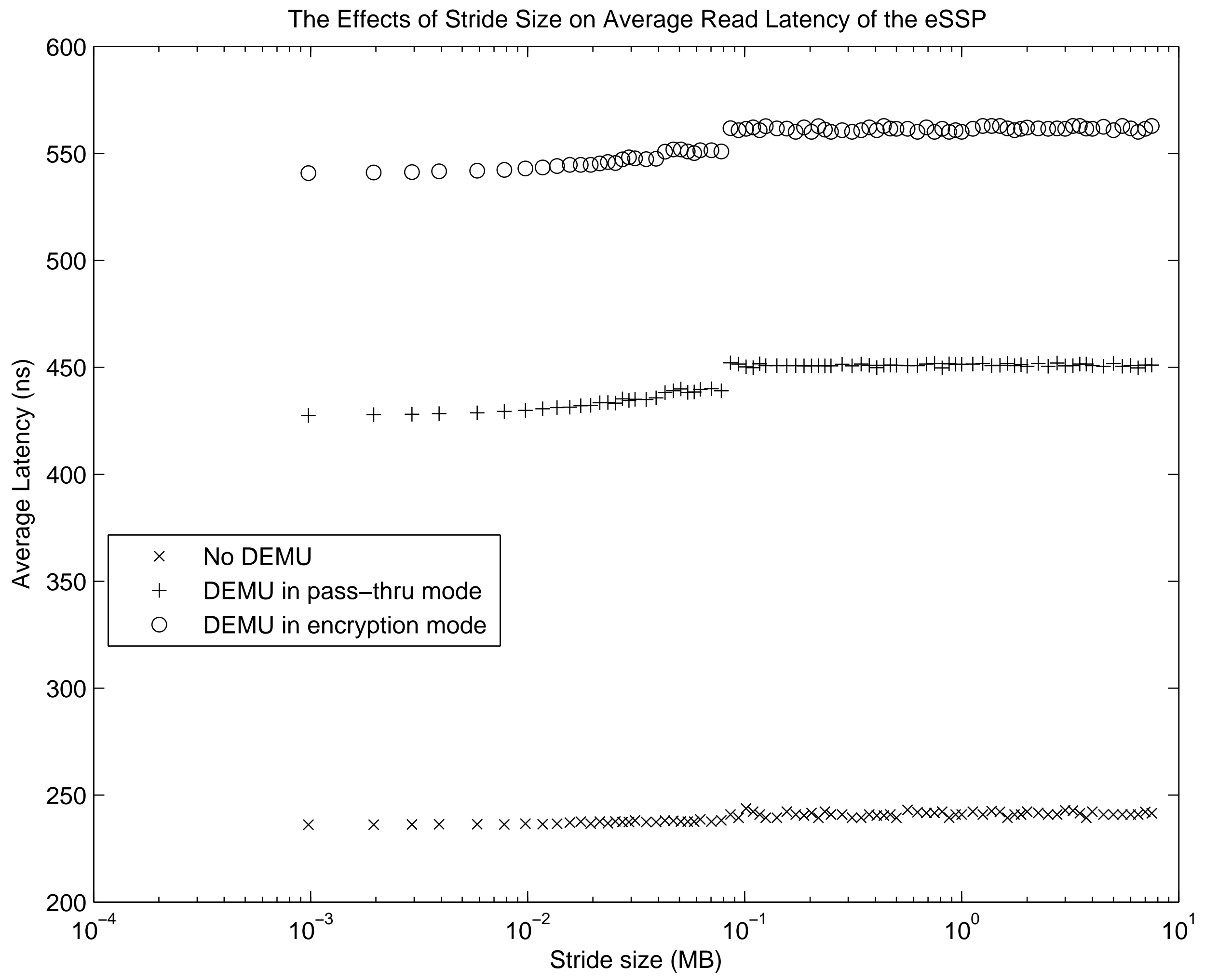

Figure 4.1: Read Latency Characteristics of the DEMU

21. Next, one clock cycle is needed for the Data Capture FSM to recognize the pending transaction followed by two clock cycles to push the transaction into the FIFO. It takes one clock cycle after the write completes for the internal FIFO logic to then assert the signal that indicates the presence of a transaction in the FIFO to the Data Process Module (DPM). The Data Process Module subsequently pops the transaction off the FIFO, which takes one clock cycle to complete. The DPM FSM then waits one clock cycle for the data to setup. The DPM FSM decides in the RNW state that it is a read transaction, and this again, takes one cycle. It takes three cycles to then issue the read request, wait for the address acknowledgement signal, and finally wait for the read completion signals from memory. The DPM FSM transitions to the state where it branches and consumes one cycle if the DEMU state is in *pass-thru* or *encryption* mode. Upon branching to the return clear-text data, transferring data to the blocked Data Capture FSM takes one clock cycle. The Data Capture FSM then transitions to a state where it returns the read data back to the PPC405 DCC. This transaction takes one clock cycle. Finally, the Data Capture FSM takes one clock cycle to transition back to its wait state to process the next read request.

The top line of average latency data shown in Figure 4.1 represents the encryption overhead incurred when the DEMU is placed in *encrypt/decrypt* mode to execute a protected application. This configuration represents the longest delay path through the DEMU, and so the latency incurred by placing the DEMU in *encryption/decryption* mode between the PPC405 DCC and main memory is an upper bound on the DEMU hardware and consequently on the system. The average read latency for the eSSP configuration containing only a DEMU in *encryption/decryption* mode is 557 ns, implying that the decryption hardware introduces $557ns - 444ns = 113ns = 12$ cycles at 100MHz of extra latency beyond the *pass-thru* mode of DEMU operation. This is concordant with the 12 ns of encryption time that Mahar observed in [49] for the same open-source AES cryptographic hardware used in the implementation of Chapter 3. All three trends show a point of discontinuity at a stride size of approximately 0.1MB. This gap is attributable to the 12 ns of prefetch that are lost when the SDRAM DIMM incorrectly attempts to prefetch the next address based on the current

address [66]. With such large stride sizes, the SDRAM DIMM cannot correctly prefetch the data of the next read transaction; however, with smaller stride sizes, it is able to do so not incurring the 12 addition cycles to activate the correct row and column in the correct bank of memory on the chip. Thus, this prefetch latency is not exposed for small stride sizes, but becomes apparent and remains apparent for large stride sizes.

4.2.2 Write Latency

The average memory latency equation provided in 4.6 is used in conjunction with raw benchmark execution times obtained for the three eSSP operating conditions to construct the average write latency trends as a function of stride size depicted in Figure 4.2. The figure shows a straight line and two curves. The straight line represents the baseline system configuration containing no EMU units. The two curves represent the PPC405's perceived write latency of memory. Unlike read transactions that are blocking, write transactions passing through the DEMU are non-blocking and return immediately if there is space in the FIFO for that transaction, or else they block until space becomes available. Because of this non-blocking behavior, the trends shown in Figure 4.2 are not the *actual* write latency of the system but rather the write latency that the PPC405 processor would perceive. Unless hardware is changed to make writes blocking like reads, the *actual* write latency is difficult to benchmark because of the possibility of the FIFO being full and accounting for blocked times and patterns of execution behavior when extracting the write latency from the benchmark execution. Moreover, the FIFO could still be full when the benchmark ends since writes are non-blocking, making it difficult to get a precise measure on write latency because of possibly inaccurate reported completion times. Despite this fact, general trends can be gleaned from Figure 4.2. The most important piece of information the figure conveys is that the write latency of the DEMU is at least bounded, denoted by the asymptotic behavior of the write latency for large numbers of write operations. The curvature of average write latency trends of the DEMU in its two modes of operation is explained using the step function of Equation 4.2 and the transaction FIFO. As the number of write operations increases, the probability

of the FIFO being full also increases. This increase in the probability of the FIFO being full in turn increases the average latency, since transactions will have to wait longer to become enqueued than if there were immediately available space. The graph shows that as the number of write operations increases, the average write latency increases due to the FIFO being full more often. The increase in latency approaches an asymptote as the FIFO becomes nearly always full. The last important piece of information this graph conveys pertains to the startup behavior of the two DEMU modes of operation. The DEMU in *pass-thru* mode starts up at a lower average write latency because write transactions take less time, meaning that the FIFO is less full. The magnitude of the latency is less than the base latency because there is enough FIFO space for transactions, since the non-blocking return takes less time to process than an actual write. The DEMU in *encryption/decryption* mode starts up at an average latency time whose magnitude is larger than the average latency time for *pass-thru* operation. This is again expected since the FIFO will be full more often as a result of write transactions taking more time to execute than in *pass-thru* mode.

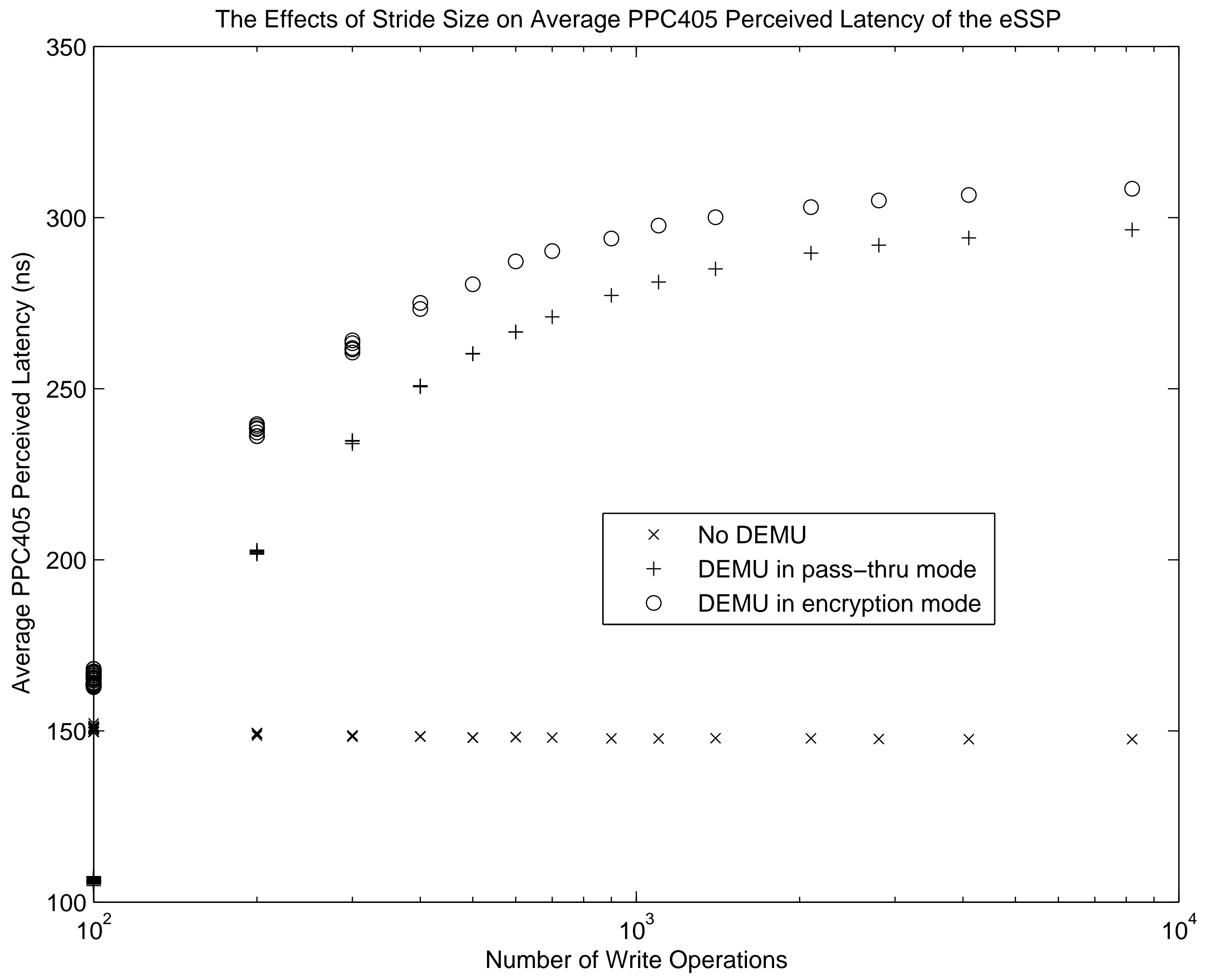

Figure 4.2: Perceived PPC405 Write Latency Characteristics

Chapter 5

Conclusion

This chapter summarizes the work presented in this thesis and provides possible future directions for the Embedded Secure Software Platform.

5.1 Conclusion

This thesis has presented a novel embedded system protection architecture titled the Embedded Secure Software Platform (eSSP), based on its predecessor platform, the Virginia Tech Secure Software Platform architected by Edmison in [3]. The eSSP is a novel approach to hardware-based embedded software security in that it relies on processor augmentation mechanisms rather than the traditional processor modification mechanisms inherent in most existing embedded system protection solutions to realize its security claims. The eSSP claims to protect the confidentiality and integrity of a secured application's instructions and data as the application executes in real-time. These claims are substantiated in the platform's implementation, comprised of an Instruction Encryption Management Unit (IEMU) that protects the embedded application's instruction privacy, and a Data Encryption Management Unit (DEMU) that protects the embedded application's data privacy. The eSSP platform is implemented with the Xilinx ML310 Evaluation Board's Xilinx XC2VP30-896-6C FPGA. The FPGA contains two embedded PPC405 processors surrounded by configurable logic fabric.

The IEMU and DEMU are implemented inside the configurable fabric of the FPGA.

The IEMU was implemented originally by Mahar in [49], but was modified to communicate with the DEMU. The IEMU is comprised of a controller FSM, a set of data tables, an AES cryptographic unit, and PLB interface units. The data tables contain information mapping the various physical pages of memory to encryption keys and counters that are applied to selectively decrypt data traveling through the IEMU en route to the PPC405 Instruction Cache Controller. Because the instruction PLB bus is read-only, the data traveling through the IEMU is unidirectional, traveling from main memory to the processor. The unidirectional nature of the IEMU precludes its use on the data PLB bus because transactions on the data PLB bus are bidirectional, allowing reading and writing of data. Therefore, a new architecture was synthesized for the DEMU's operation. The design and characterization of the DEMU is the primary contribution of this thesis. The DEMU is comprised of a Data Capture Module, a Data Process Module, a transaction FIFO, data tables, and an AES core (DTU). Like the IEMU, the data tables of the DEMU serve to map physical pages of data memory to data keys. This mapping occurs indirectly using the tagging scheme developed by Edmison [3] in which sets of instruction keys for secure applications (SELFs) are mapped to a singular static data key and a dynamic data key. The Data Capture module enqueues incoming read and write transactions originating in the PPC405 DCC into the FIFO where the transactions are popped and processed by the Data Process Module (DPM). The DPM uses the data tables to selectively encrypt outgoing write operations and selectively decrypt incoming read operations based on whether the data tables tag a page of memory as protected or not.

The secure applications (SELFs) whose data run through the IEMU and DEMU are compiled to run on the eCos real-time operating system, an open-source RTOS. Modifications were made to the eCos kernel to establish communication with the Xilinx hardware drivers. This bridge was necessary in order to allow simple applications involving I/O to run on the ML310 evaluation board. A pre-processing utility, SELFGEN, was constructed to encrypt the various instruction and data segments of a secured embedded application using AES

counter-mode encryption.

A custom set of benchmarks, *lat_mem_rd_ecos* and *lat_mem_wr_ecos*, were constructed to run on the eCos operating system based on the algorithm defined in the $LMBenchlat_mem_rd$ benchmark. The benchmarks time the execution of a loop containing one-hundred *load* or *store* instructions to use in the computation of the average read and write latency values, respectively. The results for benchmarking the average read latency introduced by the insertion of the DEMU into the data path are shown in Figure 4.1. The results indicate that the read latency introduced by the DEMU's longest read data path is bounded at approximately 33 additional cycles compared to the baseline system containing no DEMU. The results for the average write latency introduced by the insertion of the DEMU into the data path are depicted in Figure 4.2. Like the average read latency, the average write latency is bounded. These two facts imply the impact on the real-time scheduling algorithm implemented in the eCos kernel should be negligible as long as there exists room for the addition of the extra latency in the task set deadline. Suppose before inclusion of the DEMU hardware, there existed a schedule of a set of tasks $\mathfrak{T}$ such that they are schedulable under their combined deadline ζ. Furthermore, let the upper bound on the latency introduced by the DEMU be Δ. Then, because the latency is bounded as suggested by Figures 4.1 and 4.2, after the DEMU is introduced to the system the set of tasks $\mathfrak{T}$ will be schedulable under the new deadline of $\zeta + \Delta$.

In addition to arguing the neglible real-time operating impact of the DEMU, this thesis has

1. contributed a new secure software execution architecture to the field of secure embedded systems.

2. pioneered a novel approach to securing the confidentiality and integrity of embedded application data in the form of the Data Encryption Management Unit (DEMU) containing a novel PLB-to-PLB bridge that supports all PLB transactions on its slave interface and contains a custom, lightweight PLB master on its master interface.

3. characterized the bounds on the latency added to an executing embedded application by the insertion of the DEMU into the PPC405-to-memory data path.

4. contributed a secure software pre-processing methodology, SELFGEN, for the eCos RTOS.

5. addressed modifications made eCos to provide a compatible execution environment with the eSSP and the embedded PPC405 processor.

5.2 Future Directions

One possible future direction the eSSP platform could take involves pipelining read and write transactions in the DEMU hardware. Currently, the DEMU does not make the distinction between executing a PLB burst transaction and a PLB regular transaction. Burst transactions are useful when transferring large data sets to or from memory. They save in the overhead involved in having to go through the 3-phase protocol of regular read or write transactions for every memory address in the large data range. A burst transfer for read transactions is characterized by presenting a starting address to memory and a length and then letting memory return all memory values for addresses in that range. A burst transfer for write transactions is characterized by presenting a starting address to memory and a length followed by a stream of data, letting memory be written for all memory addresses in that range. The DEMU could pipeline burst transactions in its future implementations, and in doing so could significantly reduce its average memory latency overhead.

A second future direction the eSSP platform could be an evolution to include a large, highly granular L2 cache on the data path between the DEMU and main memory. Even though the current implementation uses cryptographic mechanisms to secure the confidentiality of data and instructions, attackers can still probe the bus signals traveling from the FPGA boundary to main memory. Attackers can use *timing* attacks where they single-step the processor in an attempt to correlate the execution of the application and that applica-

tion's memory access patterns. Even though they don't have access to the currently executing instructions or data, the correlation between execution and memory access may provide sufficient insight to attack the application in a malicious way. To mitigate the success of such a correlating attack, the large L2 cache would significantly reduce the amount of memory traffic traveling off-chip to memory. Moreover, a large L2 cacheline size would make it more difficult to correlate off-chip transactions. By reducing data and address observability, we increase the strength of our system through obfuscation. Though obfuscation is not as strong an approach to security, it has some effectiveness.

Although the Virginia Tech eSSP platform is in its nascent stages of development, it provides considerable security against attacks seeking to compromise the confidentiality or integrity of a protected embedded system application's instructions or data. The platform provides a unique perspective on embedded system security by augmenting the functionality of a processor rather than taking the traditional approach of processor modification. The platform's configurable, modular nature makes it easily adaptable to the emergent security concerns of the future.

Bibliography

[1] P. Kocher, R. Lee, G. McGraw, A. Raghunathan, and S. Ravi, "Security as a new dimension in embedded system design," *DAC*, pp. 753–760, 2004.

[2] P. Koopman, "Embedded system security," *IEEE Computer*, vol. 37, no. 7, pp. 95–97, 2004.

[3] J. N. Edmison, "Hardware architectures for software security," Ph.D. Dissertation, Virginia Polytechnic Institute and State University, Bradley Department of Electrical and Computer Engineering, Blacksburg, VA, June 2006.

[4] D. B. Parker, *Toward a New Framework for Information Security*, New York, 2002.

[5] F. Lau, S. Rubin, M. Smith, L. Trajkovic, and S. Fraser, "Distributed denial of service attacks," in *IEEE International Conference on Systems, Man, and Cybernetics*. Los Alamitos, CA, USA: IEEE Computer Society, 2000, pp. 2275–280.

[6] M. McLoone and J. V. McCanny, "High performance single-chip fpga rijndael algorithm implementations," in *CHES '01: Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems*. London, UK: Springer-Verlag, 2001, pp. 65–76.

[7] *The MD5 Message Digest Algorithm*, RFC 1321, April 1992.

[8] *The AES-CMAC Algorithm*, RFC 4493, June 2006.

[9] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "Aegis: architecture for tamper-evident and tamper-resistant processing," in *ICS '03: Proceedings of the 17th annual international conference on Supercomputing.* New York, NY, USA: ACM Press, 2003, pp. 160–171.

[10] A. Raghunathan, S. Ravi, S. Hattangady, and J.-J. Quisquater, "Securing mobile appliances: New challenges for the system designer," *Design, Automation and Test in Europe*, vol. 01, p. 10176, 2003.

[11] *Chipworks: Semiconductor Manufacturing. Reverse Engineering of Semiconductor components, parts and processes for Semiconductor Distributors*, http://www.chipworks.com, 2006.

[12] S. Ravi, A. Raghunathan, and S. Chakradhar, "Tamper resistance mechanisms for secure, embedded systems," *International Conference on VLSI Design*, p. 605, 2004.

[13] R. Elbaz, L. Torres, G. Sassatelli, P. Guillemin, C. Anguille, M. Bardouillet, C. Buatois, and J. B. Rigaud, "Hardware engines for bus encryption: A survey of existing techniques," in *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe.* Washington, DC, USA: IEEE Computer Society, 2005, pp. 40–45.

[14] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady, "Security in embedded systems: Design challenges," *Trans. on Embedded Computing Sys.*, vol. 3, no. 3, pp. 461–491, 2004.

[15] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *CRYPTO '96: Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology.* London, UK: Springer-Verlag, 1996, pp. 104–113.

[16] J.-F. Dhem, F. Koeune, P.-A. Leroux, P. Mestré, J.-J. Quisquater, and J.-L. Willems, "A practical implementation of the timing attack," in *CARDIS*, 1998, pp. 167–182.

[17] D. Boneh and D. Brumley, "Remote timing attacks are practical," in *Proceedings of the 12th USENIX Security Symposium, August 2003.*, 2003.

[18] M. Aigner and E. Oswald, "A power analysis tutorial," *Institute for Applied Information Processing and Communication*, 2000.

[19] W. van Eck, "Electromagnetic radiation from video display units: an eavesdropping risk?" *Computer Security*, vol. 4, no. 4, pp. 269–286, 1985.

[20] E. Biham and A. Shamir, *Advances in Cryptology - CRYPTO '97: 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 1997. Proceedings.*, ser. Lecture Notes in Computer Science. Berlin, Germany: Springer Berlin / Heidelberg, 1997, vol. 1294–1997, ch. Differential Fault Analysis of Secret Key Cryptosystems, pp. 513–525.

[21] P. Dusart, G. Letourneux, and O. Vivolo, *Applied Cryptography and Network Security*, ser. Lecture Notes in Computer Science. Berlin, Germany: Springer Berlin / Heidelberg, 2003, vol. 2846–2003, ch. Differential Fault Analysis on A.E.S, pp. 293–306.

[22] *Vulnerability notes database*, CERT coordination center: http://www.kb.cert.org/vuls/, 2006.

[23] M. Bond and R. Anderson, "Api-level attacks on embedded systems," *IEEE Computer*, vol. 34, pp. 67–75, 2001.

[24] E. Naess, D. A. Frincke, A. D. McKinnon, and D. E. Bakken, "Configurable middleware-level intrusion detection for embedded systems," *International Workshop on Security in Distributed Computing Systems*, vol. 02, pp. 144–151, 2005.

[25] M. Henning, "The rise and fall of corba," *ACM Queue*, vol. 4, 2006.

[26] "Secure execution via program shepherding." MIT Press, 2002, pp. 191–206.

[27] S. Oikawa, H. Ishikawa, M. Iwasaki, and T. Nakajima, "Constructing secure operating environments by co-locating multiple embedded operating systems," *IEEE Second Consumer Communications and Networking Conference 2005*, pp. 43–48, 2005.

[28] F. K. Gurkaynak, A. Burg, N. Felber, W. Fichtner, D. Gasser, F. Hug, and H. Kaeslin, "A 2 gb/s balanced aes crypto-chip implementation," in *GLSVLSI '04: Proceedings of the 14th ACM Great Lakes symposium on VLSI.* New York, NY, USA: ACM Press, 2004, pp. 39–44.

[29] D. C. Wilcox, L. G. Pierson, P. J. Robertson, E. L. Witzke, and K. Gass, *First International Workshop, Cryptographic Hardware and Embedded Systems, Worcester, MA, USA, August 1999. Proceedings.*, ser. Lecture Notes in Computer Science. Berlin, Germany: Springer Berlin / Heidelberg, 1997, vol. 1717–1999, ch. A DES ASIC Suitable for Network Encryption at 10 Gbps and Beyond, pp. 37–51.

[30] S. Sen, S. I. Hossain, K. Islam, D. R. Chowdhuri, and P. P. Chaudhuri, "Cryptosystem designed for embedded system security," *International Conference on VLSI Design*, p. 271, 2003.

[31] R. Gonzalez, "Xtensa: a configurable and extensible processor," *IEEE Micro*, vol. 20, pp. 60–70.

[32] *Literature of the NIOS Processor*, http://www.altera.com/literature/lit-nio.jsp, 2006.

[33] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood, "Lx: a technology platform for customizable vliw embedded processing," in *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture.* New York, NY, USA: ACM Press, 2000, pp. 203–213.

[34] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee, "Chimaera: a high-performance architecture with a tightly-coupled reconfigurable functional unit," in *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture.* New York, NY, USA: ACM Press, 2000, pp. 225–235.

[35] S. Talla, "Adaptive explicitly parallel instruction computing," Ph.D. Dissertation, New York University, Department of Computer Science, New York, NY, December 2000.

[36] P. Yu and T. Mitra, "Characterizing embedded applications for instruction-set extensible processors," in *DAC '04: Proceedings of the 41st annual conference on Design automation.* New York, NY, USA: ACM Press, 2004, pp. 723–728.

[37] R. G. Ragel, S. Parameswaran, and S. M. Kia, "Micro embedded monitoring for security in application specific instruction-set processors," in *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems.* New York, NY, USA: ACM Press, 2005, pp. 304–314.

[38] X. Zhang, L. van Doorn, T. Jaeger, R. Perez, and R. Sailer, "Secure coprocessor-based intrusion detection," in *EW10: Proceedings of the 10th workshop on ACM SIGOPS European workshop: beyond the PC.* New York, NY, USA: ACM Press, 2002, pp. 239–242.

[39] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, "Secure embedded processing through hardware-assisted run-time monitoring," *Design, Automation and Test in Europe*, vol. 01, pp. 178–183, 2005.

[40] C.-P. Su, C.-H. Wang, K.-L. Cheng, C.-T. Huang, and C.-W. Wu, "Design and test of a scalable security processor," in *ASP-DAC '05: Proceedings of the 2005 conference on Asia South Pacific design automation.* New York, NY, USA: ACM Press, 2005, pp. 372–375.

[41] R. G. Ragel and S. Parameswaran, "Impres: integrated monitoring for processor reliability and security," in *DAC '06: Proceedings of the 43rd annual conference on Design automation.* New York, NY, USA: ACM Press, 2006, pp. 502–505.

[42] J.-P. Kaps and C. Paar, "Fast des implementation for fpgas and its application to a universal key-search machine," in *SAC '98: Proceedings of the Selected Areas in Cryptography.* London, UK: Springer-Verlag, 1999, pp. 234–247.

[43] C. Patterson, "High performance des encryption in virtex(tm) fpgas using jbits(tm)," in *FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines.* Washington, DC, USA: IEEE Computer Society, 2000, p. 113.

[44] J. Zambreno, A. Choudhary, R. Simha, B. Narahari, and N. Memon, "Safe-ops: An approach to embedded software security," *Trans. on Embedded Computing Sys.*, vol. 4, no. 1, pp. 189–210, 2005.

[45] T. Wollinger, J. Guajardo, and C. Paar, "Security on fpgas: State-of-the-art implementations and attacks," *Trans. on Embedded Computing Sys.*, vol. 3, no. 3, pp. 534–574, 2004.

[46] R. Best, "Preventing software piracy with crypto-microprocessors," in *Proceedings of IEEE Spring COMPCON 80.* Los Alamitos, CA, USA: IEEE Computer Society, 1980, pp. 466–469.

[47] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," *SIGPLAN Not.*, vol. 35, no. 11, pp. 168–177, 2000.

[48] A. Arbaugh, D. Farber, and J. Smith, "A secure and reliable bootstrap architecture," in *Proceedings of the 1997 IEEE Symposium on Security and Privacy.* Los Alamitos, CA, USA: IEEE Computer Society, 1997, pp. 65–71.

[49] A. J. Mahar, "Design and characterization of a hardware encryption management unit for secure computing platforms," Masters Thesis, Virginia Polytechnic Institute and State University, Bradley Department of Electrical and Computer Engineering, Blacksburg, VA, June 2005.

[50] B. Muzal, "Design and implementation of a hierarchical key management system for runtime application protection," Masters Thesis, Virginia Polytechnic Institute and State University, Bradley Department of Electrical and Computer Engineering, Blacksburg, VA, 2006.

[51] *eCos User Guide*, eCosCentric, 2003.

[52] *eCos Reference Manual*, eCosCentric, 2003.

[53] *PowerPC 405 Embedded Processor Core User's Manual, Fifth Edition*, IBM, December 2001.

[54] *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2*, TIS Committee, May 1995.

[55] *ML310 User Guide: Virtex-II Pro Embedded Development Plaform. Xilinx User Guide 68*, Xilinx, January 2005.

[56] *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet. Data Sheet 83*, Xilinx, October 2005.

[57] *64-Bit Processor Local Bus Architecture Specifications, Third Edition*, IBM, May 2001.

[58] D. A. McGrew and J. Viega, "The galois/counter mode of operation (gcm)," May 2005.

[59] *Content-Addressable Memory v5.1. Data Sheet 253*, Xilinx, November 2004.

[60] *On-Chip Peripheral Bus Architecture Specifications, Second Edition*, IBM, April 2001.

[61] J. Daemen and V. Rijmen, *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002.

[62] *Xilinx LogiCORE FIFO Generator User's Manual, Second Edition. User Guide 175*, Xilinx, January 2006.

[63] *Xilinx LogiCORE PLB IPIF Product Specification. Data Sheet 448*, Xilinx, April 2005.

[64] *LMBench - Tools for Performance Analysis*, http://www.bitmover.com/lmbench/, 2006.

[65] *MiBench - A free, commercially representative embedded benchmark suite*, http://www.eecs.umich.edu/mibench/, 2006.

[66] *HYB25D256800BT-5 256MBit Double Data SDRAM Data Sheet*, Infineon Technologies, January 2003.

Appendix A

Source Code

A.1 PPC405 Startup Assembly

```
        mfccr0   0
        oris     0,0,0x50000000@h          //Handle PPC405 Errata
        mtccr0   0
        li       r3,0x00000000             // Make sure nothing is cacheable
        mticcr   r3
        mtdccr   r3
        // Force data caches to be totally clean
        lwi      r3,0
        lwi      r4,0x8000
10:     dcbf     0,r3
        dccci    0,r3
        addi     r3,r3,16
        cmpw     r3,r4
        bne      10b
        lwi      r3,0x00000000
        iccci    0,r3
```

A.2 Benchmark Caller Code

```
/*****************************************************************************
**Program: eCos data-side performance benchmark for the eSSP
**
**Author:   Eric J. Lorden
**
**Modifications: 12/05/06         ejl       Creation Date
**
**Note: adapted from LMBENCH's lat_mem_rd benchmark to stress data-side caches
*****************************************************************************/

/*Defines*/
#define LOWER    (16*64)
#define ARRAY_SIZE (8*1024*1024)
#define MAX_ITERATIONS 1

/*Includes*/
#include <stdlib.h>

/*Function prototypes*/
unsigned long int step(unsigned long int k);

/*Benchmark entry point*/
void cyg_user_start(void)
{
        /*Declare local variables*/

        unsigned long int stride_size = (16*64);                  /*the largest
        possible transaction is a 16-beat fixed length burst transaction*/
        unsigned long int array_size  = ARRAY_SIZE;               /*8 MB maximum size
        */
        unsigned long int iterations  = 1;                        /*1 iteration for
        now since we have the highest resolution timer
                                                                  -- may be greater
                                                                  than 1 if a lower-
                                                                  precision timer is
                                                                  used*/
        unsigned long int i = 1;                                  /*iterator used to
        link circular list*/
        unsigned long int hptc_base_address = 0x0FE00000;         /*the base where
        high performance timer values are placed*/
        char** array = NULL;                                      /*Dynamically
        allocate an 8 MB array -- eCos malloc() is flaky*/
        unsigned long int num_100s;                               /*Number of 100
        instructions needed to traverse the circularly linked list once*/
        register unsigned int * mem = (unsigned int *)0x0FE00000;

        memset(mem,0x00,10);
        array = (char**)malloc(ARRAY_SIZE);                                /*this is 8
        MB*/

        /*for each array size*/
        for (array_size = ARRAY_SIZE; array_size <= ARRAY_SIZE; array_size = step(array_size
        ))
        {
                /*for each stride considered between 1K and ARRAY_SIZE*/
                for (stride_size = LOWER; stride_size < ARRAY_SIZE; stride_size = step(
                stride_size))
                {
                        /*for each iteration*/
                        for (iterations = 1; iterations <= MAX_ITERATIONS; ++iterations)
                        {
```

```
                    /*Set up circularly linked list, with each link being
                    stride_size away in memory from the previous link*/
                    for (i = stride_size; i < ARRAY_SIZE; i += stride_size)
                    {
                            /*link*/
                            *(char **)&array[i - stride_size] = (char*)&array[i
                            ];
                    }
                    /*make the linking circular by pointing the last entry to
                    the first*/
                    *(char **)&array[i - stride_size] = (char*)&array[0];

                    /*Benchmark*/
                    num_100s = ((ARRAY_SIZE / (stride_size * 100))+1);
                    data_side_benchmark(hptc_base_address, array, num_100s);

                    /*GET TERMINATING TIME*/
                    hptc_base_address += 16;
                                    /*increment the address to prepare to grab
                    the END TIME*/
                }/*next iteration*/
            }/*next stride size*/
        }/*next array size*/

        /*Signal end of benchmark*/
        printf("Done with simulation...\r\n");

        /*End benchmark*/
        /*Call the infinite branch*/
        while(1);

        /*this return it never reached*/
        return;
}

/*Step function to increase the stride size in a power of two fashion.
 Adapted from lat_mem_rd*/
unsigned long int step(unsigned long int k)
{
        /*Declare local variables*/
        unsigned long int s;

        if (k < 1024)
        {
                k = k * 2;
        }
        else if (k < 4*1024)
        {
                k += 1024;
        }
        else
        {
                for (s = 32 * 1024; s <= k; s *= 2)
                ;
                k += s / 16;
        }
        return (k);
}
```

A.3 Read Benchmark Assembly

```
#define r1  1
#define r3  3
#define r4  4
#define r5  5
#define r16 16
#define r17 17
#define r18 18
#define r19 19
#define r20 20

.text
.globl  data_side_benchmark

##########################################################################################################

#Function: data_side_benchmark -- times a number of iterations of a read loop that is used
to benchmark the average
#                                  read access latency
#
#Inputs:
#       r1:     stack address (apparently gcc loads these) which to load the current values
#       r3:     starting address of current 4-tuple of data in memory
#       r4:     starting address of circularly linked list
#       r5:     numer of iterations of the read loop
#
#Used/Modified:
#       r20:  the value of the upper and lower timer bases to be transferred to the memory
location described by r19
#       r19:  the address at which to load the current values gets incremented by 4 to store
 two words (TBU and TBL)
#       r18:  register used to traverse the circularly linked list
#       r17:  register used to hold the number of iterations of the read loop
#       r16:  register used to hold the number of data 4-tuples generated
#
#Outputs: TBU and TBL to the address specified as the parameter in r31
#Returns: none
#
#Author: Eric J. Lorden
##########################################################################################################

data_side_benchmark:

        stwu    r1,-64(r1)              #Pushing the stack (in r1) to make room to place
        current values of r19 and r20
        stw     r19,0(r1)               #Preserve r19
        stw     r20,4(r1)               #Preserve r20
        stw     r18,8(r1)               #Preserve r18
        stw     r17,12(r1)              #Preserve r17
        stw     r16,16(r1)              #Preserve r16

        mr      r19, r3                 #Get the address of where we are going to store
        clocking data
        mr      r18, r4                 #Load beginning address of array
        mr      r17, r5                 #Load the number of iterations we need to do
        lwz     r16,4(r19)              #load counter of number of number of data sample
        points generated

                                        #GET START TIME
        mftbl   r20                     #Capture the lower time base and store in r20
        stw     r20,4(r19)              #Store r20 (TBL) to r19+4
        mftbu   r20                     #Capture the upper time base and store in r20
```

```
        stw     r20,0(r19)              #Store r20 (TBU) to r19+0

startl: lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
        lwz     r18,0(r18)              #Traverse the circularly linked list
```

```
lwz     r18,0(r18)           #Traverse the circularly linked list
lwz     r18,0(r18)           #Traverse the circularly linked list
lwz     r18,0(r18)           #Traverse the circularly linked list
lwz     r18,0(r18)           #Traverse the circularly linked list
lwz     r18,0(r18)           #Traverse the circularly linked list
lwz     r18,0(r18)           #Traverse the circularly linked list
lwz     r18,0(r18)           #Traverse the circularly linked list
lwz     r18,0(r18)           #Traverse the circularly linked list
lwz     r18,0(r18)           #Traverse the circularly linked list
lwz     r18,0(r18)           #Traverse the circularly linked list
lwz     r18,0(r18)           #Traverse the circularly linked list
lwz     r18,0(r18)           #Traverse the circularly linked list
lwz     r18,0(r18)           #Traverse the circularly linked list
lwz     r18,0(r18)           #Traverse the circularly linked list
lwz     r18,0(r18)           #Traverse the circularly linked list
lwz     r18,0(r18)           #Traverse the circularly linked list
lwz     r18,0(r18)           #Traverse the circularly linked list
lwz     r18,0(r18)           #Traverse the circularly linked list
lwz     r18,0(r18)           #Traverse the circularly linked list
lwz     r18,0(r18)           #Traverse the circularly linked list
lwz     r18,0(r18)           #Traverse the circularly linked list
lwz     r18,0(r18)           #Traverse the circularly linked list
lwz     r18,0(r18)           #Traverse the circularly linked list
lwz     r18,0(r18)           #Traverse the circularly linked list
lwz     r18,0(r18)           #Traverse the circularly linked list
lwz     r18,0(r18)           #Traverse the circularly linked list
lwz     r18,0(r18)           #Traverse the circularly linked list
lwz     r18,0(r18)           #Traverse the circularly linked list
lwz     r18,0(r18)           #Traverse the circularly linked list
lwz     r18,0(r18)           #Traverse the circularly linked list
lwz     r18,0(r18)           #Traverse the circularly linked list
lwz     r18,0(r18)           #Traverse the circularly linked list
lwz     r18,0(r18)           #Traverse the circularly linked list
lwz     r18,0(r18)           #Traverse the circularly linked list
lwz     r18,0(r18)           #Traverse the circularly linked list
lwz     r18,0(r18)           #Traverse the circularly linked list
lwz     r18,0(r18)           #Traverse the circularly linked list
lwz     r18,0(r18)           #Traverse the circularly linked list

subi    r17,r17,1            #Decrement iterator
cmplwi  r17,0                #Compares r17 to zero to prepare for branch
bne     startl
                             #GET END TIME
addi    r19, r19, 0x8        #Prepares for next store
mftbl   r20                  #Capture the lower time base and store in r20
stw     r20,4(r19)           #Store r20 (TBL) to r19+4
mftbu   r20                  #Capture the upper time base and store in r20
stw     r20,0(r19)           #Store r20 (TBU) to r19+0

addi    r19, r19, 0x8        #Generate address for data point counter marker
lis     r20,0xDEAD           #Value to be placed to show the end marker in memory
stw     r20,0(r19)           #store at end of list
addi    r16,r16,1            #increase counter
stw     r16,4(r19)           #store counter

lwz     r17,12(r1)           #Restore register
lwz     r18,8(r1)            #Restore register
lwz     r20,4(r1)            #Restore register
lwz     r19,0(r1)            #Restore register
addi    r1,r1,64             #Pop Stack

blr                          #return from function
```

A.4 Write Benchmark Assembly

```
#define r1  1
#define r3  3
#define r4  4
#define r5  5
#define r16 16
#define r17 17
#define r18 18
#define r19 19
#define r20 20

.text
.globl  data_side_benchmark

##########################################################################################################

#Function: data_side_benchmark -- times a number of iterations of a write loop that is used
to benchmark the average
#                                   write access latency
#
#Inputs:
#       r1:     stack address (apparently gcc loads these) which to load the current values
#       r3:     starting address of current 4-tuple of data in memory
#       r4:     starting address of circularly linked list
#       r5:     numer of iterations of the write loop
#
#Used/Modified:
#       r20:  the value of the upper and lower timer bases to be transferred to the memory
location described by r19
#       r19:  the address at which to load the current values gets incremented by 4 to store
 two words (TBU and TBL)
#       r18:  register used to traverse the circularly linked list
#       r17:  register used to hold the number of iterations of the write loop
#       r16:  register used to hold the number of data 4-tuples generated
#
#Outputs: TBU and TBL to the address specified as the parameter in r31
#Returns: none
#
#Author: Eric J. Lorden
##########################################################################################################

data_side_benchmark:

        stwu    r1,-64(r1)              #Pushing the stack (in r1) to make room to place
        current values of r19 and r20
        stw     r19,0(r1)               #Preserve r19
        stw     r20,4(r1)               #Preserve r20
        stw     r18,8(r1)               #Preserve r18
        stw     r17,12(r1)              #Preserve r17
        stw     r16,16(r1)              #Preserve r16


        mr      r19, r3                 #Get the address of where we are going to store
        clocking data
        mr      r18, r4                 #Load beginning address of array
        mr      r17, r5                 #Load the number of iterations we need to do
        lwz     r16,4(r19)              #load counter of number of number of data sample
        points generated
        lwz     r18,0(r18)              #Get the address of the second node in the linked
        list in the 8MB array

                                        #GET START TIME
```

```
        mftbl   r20              #Capture the lower time base and store in r20
        stw     r20,4(r19)       #Store r20 (TBL) to r19+4
        mftbu   r20              #Capture the upper time base and store in r20
        stw     r20,0(r19)       #Store r20 (TBU) to r19+0


startl: stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
        stw     r18,0(r18)       #(just a dummy write operation)
```

```
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)
stw     r18,0(r18)        #(just a dummy write operation)

subi    r17,r17,1         #Decrement iterator
cmplwi  r17,0             #Compares r17 to zero to prepare for branch
bne     startl

                          #GET END TIME
addi    r19, r19, 0x8     #Prepares for next store
mftbl   r20               #Capture the lower time base and store in r20
stw     r20,4(r19)        #Store r20 (TBL) to r19+4
mftbu   r20               #Capture the upper time base and store in r20
stw     r20,0(r19)        #Store r20 (TBU) to r19+0

addi    r19, r19, 0x8     #Generate address for data point counter marker
lis     r20,0xDEAD        #Value to be placed to show the end marker in memory
stw     r20,0(r19)        #store at end of list
addi    r16,r16,1         #increase counter
stw     r16,4(r19)        #store counter

lwz     r17,12(r1)        #Restore register
lwz     r18,8(r1)         #Restore register
lwz     r20,4(r1)         #Restore register
lwz     r19,0(r1)         #Restore register
```

```
addi    r1,r1,64            #Pop Stack

blr                         #return from function
```