4-2011

# A prototype security hardened field device for SCADA systems.

Bradley Alan Luyster
*University of Louisville*

Recommended Citation
Luyster, Bradley Alan, "A prototype security hardened field device for SCADA systems." (2011). *Electronic Theses and Dissertations.* Paper 869.
https://doi.org/10.18297/etd/869

# A PROTOTYPE SECURITY HARDENED FIELD DEVICE FOR SCADA SYSTEMS

By

Bradley Alan Luyster
B.S., University of Louisville, 2009

A Thesis
Submitted to the Faculty of the
University of Louisville
J. B. Speed School of Engineering
as Partial Fulfillment of the Requirements
for the Professional Degree

MASTER OF ENGINEERING

Department of Electrical and Computer Engineering

April 2011

A PROTOTYPE SECURITY HARDENED FIELD DEVICE FOR SCADA SYSTEMS

Submitted By:__________________________________
Bradley Alan Luyster

A Thesis Approved On

___________________________________
(Date)

by the Following Reading and Examination Committee

___________________________________
James H. Graham, Thesis Co-Director

___________________________________
Jeffery L. Hieb, Thesis Co-Director

___________________________________
John Naber, Committee Member

# ACKNOWLEDGMENTS

I would like to thank Dr. Jeffery Hieb for his assistance in guiding me through the miasma of academia, as well as Jane Tanner for her invaluable advice in navigating the shores of both University bureaucracy and the greater world. I must also thank my Uncle, Bill Luyster, for providing the initial spark which eventually led me to Electrical Engineering. Finally, I cannot thank enough my parents, Thom and Wendy Luyster, for providing unflagging support, no matter what endeavor I have taken upon myself.

# ABSTRACT

This thesis describes the development of a prototype security hardened field device (such as a remote terminal unit) based on commodity hardware and implementing a previously developed security architecture. This security architecture has not been implemented in the past due to the difficulty of providing an operating system which meets the architecture's isolation requirements. Recent developments in both hardware and software have made such an operating system possible, opening the door to the implementation and development of this new security architecture in physical devices attached to supervisory control and data acquisition (SCADA) systems. A prototype is developed using commodity hardware selected for similarity to existing industrial systems and making use of the new OKL4 operating system. Results of prototype development are promising, showing performance values which are adequate for a broad range for industrial applications.

TABLE OF CONTENTS

Page

# Table of Contents

ACKNOWLEDGMENTS.....iii
ABSTRACT.....iv
CHAPTER I – INTRODUCTION.....1
1.1 BACKGROUND.....1
1.2 PROBLEM.....2
1.3 MOTIVATION.....3
1.4 ORGANIZATION.....3
CHAPTER II – LITERATURE REVIEW.....5
2.1 SCADA SECURITY.....5
2.1.1 SCADA SECURITY ISSUES.....7
2.1.2 SECURING SCADA SYSTEMS.....8
2.2 ARCHITECTURAL SECURITY IN KERNEL MODELS.....9
2.2.1 SEPARATION KERNELS AND MICROKERNELS.....11
2.2.2 THE OKL4 MICROKERNEL.....15
2.2.3 THE SEL4 KERNEL.....18
2.3 A SECURE INDUSTRIAL SYSTEM USING MICROKERNELS.....19
CHAPTER III – DESIGN APPROACH.....21
3.1 ARCHITECTURE MODEL AND SECURITY FEATURES.....21
3.2 PRIOR WORK.....25
3.3 ISOLATING COMPONENTS WITH CELLS AND THREADS.....26
3.4 INTER-CELL COMMUNICATION USING IPC.....29
3.5 SECURITY FEATURES.....34
3.5.1 A SIMPLIFIED SCADA PROTOCOL.....35
3.5.2 USER AUTHENTICATION AND HANDSHAKING.....40
3.5.3 ROLE BASED ACCESS CONTROL.....41
3.6 DESIGN SUMMARY.....43
CHAPTER IV – IMPLEMENTATION.....44
4.1 HARDWARE, BUILD SYSTEM, AND WORKFLOW.....44
4.2 IO HARDWARE AND SOFTWARE.....50
4.3 UTILITY FUNCTIONALITY.....55
4.3.1 HARDWARE MANAGEMENT.....56
4.3.2. SERIAL DEBUG.....57
4.4 NETWORK IO.....58
4.5 SECURITY CELL.....61
CHAPTER V – PERFORMANCE ANALYSIS.....66
5.1 TEST GOALS AND CONSTRAINTS.....66
5.2 TEST METHODOLOGIES.....68

5.3 INITIAL PERFORMANCE MEASUREMENTS....70
5.4 SECURITY FEATURE PERFORMANCE MEASUREMENTS....73
CHAPTER VI – CONCLUSIONS AND FUTURE RESEARCH....76
6.1 SUMMARY OF RESULTS....76
6.2 FUTURE DIRECTIONS....78
REFERENCES....81
APPENDIX I – LAB MANUAL....83
APPENDIX II – SELECTED CODE....95
APPENDIX 2.1 – UTILITY CELL ELFWEAVER XML FILE....95
APPENDIX 2.2 – DATA STRUCTURES IN ROLE-BASED ACCESS CONTROL. 95
APPENDIX 2.3 – PHYSICAL IO INTERFACING....97
VITA....99

# GLOSSARY OF TERMS

Capabilities – A security construct which contains both the reference required to access an object, as well as access rights assigned to the capability holder. Ownership of a capability implies permission to access a given resource.

Cell – A virtual construct in the OKL4 operating system. A logical set of a running thread, a memory space, and a zone.

Dnp3 – Distributed Network Protocol, a common communication protocol in industrial control systems.

I2C – Inter-IC bus

IDE – Integrated development environment

IPC – Interprocess communication

Kernel – An operating system's lowest level abstraction code, interfacing between hardware and application code, often performing other duties such as memory management, task scheduling and file system management, among others.

MTU – Master terminal unit

Macrokernel – An operating system kernel which places code such as file system management, device drivers, and server daemons in kernel-mode code.

Memory space – An OKL4 construct which separates a given section of virtual memory from all other sections of virtual memory.

Microkernel – An operating system kernel which seeks to minimize code which runs in kernel-mode.

Modbus – A common communication protocol used in industrial control systems.

OKL4 – A commercially available microkernel operating system, currently the primary focus of research on the L4 family of microkernels.

Points – A distinct input or output on a field device. Examples include digital inputs and analog outputs.

Pre-shared Secret – a secret shared previously shared between two systems using a secure communications channel, used for authentication and verification of credentials.

RBAC – Role based access control

RTU – Remote terminal unit

RPC – Remote procedure call

SCADA – Supervisory control and data acquisition

SDK – Software development kit

SHA-256 – A cryptographic hashing function which encodes a given set of data as a unique 256 bit value.

TFTP – Trivial file transfer protocol

TCP – Transmission control protocol, a core protocol of the Internet protocol stack.

Thread – A piece of code executing concurrently with other pieces of code, switched in and out in a “timesharing” manner.

UDP – User datagram protocol. A simpler alternative to TCP which sacrifices reliability for speed and simplicity.

XML – Extensible markup language

Zone – A construct in OKL4 defining which memory spaces may access each-other.

LIST OF TABLES

Table 2.1: Cost of Minimal L4 IPC Transfer on Various Architectures[28]......................18

Table 3.1: Summary of Simplified SCADA Communications Protocol............................37

Table 5.1: IPC Performance measurements......................................................................71

Table 5.2: Performance Summary of Security Operations................................................74

# LIST OF FIGURES

Figure 2.1: Typical SCADA System Architecture..............................................................6

Figure 2.2: Monolithic and Micro- Kernel Architectural Differences..............................11

Figure 3.1: Target Architecture from [33].........................................................................22

Figure 3.2: IPC messages during Start-up.........................................................................31

Figure 3.3: IPC from Network Event to Physical IO, given proper RBAC credentials….32

Figure 3.4: IPC from network event to physical IO, given improper RBAC credentials..33

Figure 3.5: Full state diagram for IPC...............................................................................34

Figure 3.6: Decision tree for SCADA packet reception...................................................39

Figure 4.1: Hierarchy of ElfWeaver Compilation.............................................................48

Figure 4.2: Pinout of theMCP23009 Digital IO IC...........................................................50

Figure 4.3: Pinout of the MCP4725 Analog Output IC....................................................51

Figure 4.4: Pinout of the AD7997 Analog Input IC..........................................................51

Figure 5.1: IPC Communications Testing Paths...............................................................70

# CHAPTER I

# INTRODUCTION

This thesis describes the design, development, and testing of a security hardened field device, such as a remote terminal unit (RTU) for supervisory control and data acquisition (SCADA) systems. This prototype RTU is developed using several previously researched, but as-of-yet unconstructed, architectural features which are designed to provide additional layers of security to industrial control systems which control this nation's energy, water and fuel supplies (among many other systems). This thesis will review the pitfalls of existing SCADA systems, explore the possibilities of security hardened systems using new architectural structures, and describe the development and testing of a prototype RTU utilizing these structures.

## 1.1 Background

SCADA systems are used in the operation of many industrial systems, from the power grid to potable water distribution: large, monolithic systems which are critical to the health and well-being of the citizenry. These systems are widely distributed, and require a large number of remote terminals, each controlling a small number of devices, and gathering data from a small number of sensors. Many of these systems are connected to a central control location by a variety of possible commodity communications systems, ranging from radio links to industrial Ethernet connections. Unfortunately, as these systems have grown larger, the pervasiveness and public awareness of these commodity communications systems has also grown. Corporations and utilities can no longer rely on security through obscurity to protect these systems. Further, with the growth of the

Internet, these systems have become increasingly connected, forcing industries to defend against security threats well outside these systems' design parameters.

Network security and an understanding of these communications problems can only provide a modest amount of security. Drop in modules can provide additional security, but these control systems have product lifetimes measured in decades. The security solutions used in these modules may find themselves outmoded or broken in a few months or years, becoming a huge expense to any company maintaining a device's security.

## 1.2 Problem

Part of the security problem is endemic to the architecture utilized by these devices. The monolithic kernels used by the operating systems integrated into the RTUs currently on the market are fundamentally incapable of providing the high assurance security required of such critical control devices. Formal verification is a key component of high assurance systems. For many years, this verification has been out of reach for operating-system kernels. Several operating systems since the 1970's have attempted to claim the crown of verified security, and several certifications exist which provide some subjective measurement of security. For many systems, these measures are sufficient – such operating systems are used in the aerospace and defense industries.

Due to the important role kernels play in providing device security, building an RTU with high assurance requires the use of a trusted kernel. A verified operating system allows for the development of verifiable security structures. If these structures operate as intended, then a deterministic level of security can be architected, transcending the security of any drop-in device or network security solution. As a result, it is possible to create an RTU whose security is absolutely guaranteed over the lifetime of the device.

## 1.3 Motivation

The creation of a prototype would be impossible without the existence of a verified secure operating system. Until now, many architectures for verified operating systems have been proposed, and many have endured rigorous testing and certification procedures, only to fall just short of completely verifiable performance. Recently, a research group in Australia has reportedly achieved the rigorous mathematical verification of a microkernel based operating system. Using this operating system, a prototype can be developed using previously researched architectures, and its performance measured in order to determine the viability of such an architecture implemented with modern hardware and software. A security architecture developed by Hieb and Graham [33] considers the features provided by a microkernel operating-system, and the possibilities of a formally verified kernel. Until the development of this new kernel, the implementation of a prototype based on this architecture has been impossible.

## 1.4 Organization

The second chapter of this thesis presents a detailed review of related literature and research, more clearly defining and developing the summary provided in this chapter. Chapter three presents the design architecture, explores the security features developed in prior work, and examines how such an architecture can be applied to the verified operating system. Chapter four is an in-depth exploration of the development environment and the prototyping process, reviewing the implementation of the prototype from hardware to software. Chapter five describes the methods and results of performance testing the RTU prototype. Chapter six summarizes the findings of this thesis, and explores some possible future directions for continued research in this area.

# CHAPTER II

# LITERATURE REVIEW

This review of existing literature provides an overview of the current state and history of SCADA security research. The following sections will detail the work and research which has led to the need for the development of a prototype as described in the following chapters. Section 2.1 provides an overview of SCADA systems and their associated security deficiencies. Section 2.2 examines the insufficiencies and failures of existing solutions. Section 2.3 explores separation and microkernels in the context of security solutions, and Section 2.4 explores the existing work done in using microkernels in industrial systems.

## 2.1 SCADA Security

Supervisory Control and Data Acquisition (SCADA) is a term that has come to refer to any and all devices involved in a computerized, automatic industrial control system. The IEEE defines SCADA systems as "A system operating with coded signals over communication channels so as to provide control of remote equipment (using typically one communication channel per remote station). The supervisory system may be combined with a data acquisition system, by adding the use of coded signals over communication channels to acquire information about the status of the remote equipment for display or for recording functions[2]." These systems rose to prominence concurrent to microcomputers, beginning in the 1960s. As SCADA became more popular, the different architectures used became codified in IEEE Standard C37.1-2008 [3].

Principally, there are three different SCADA system architectures, the first two being primarily of historical interest.

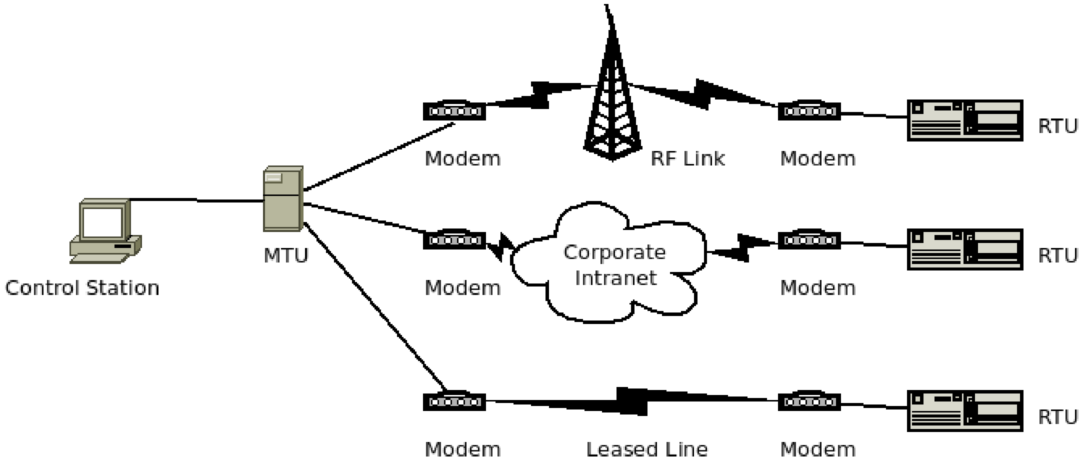

*Figure 2.1: Typical SCADA System Architecture*

Initially, SCADA systems were completely centralized, with a single master controller station connected directly to sensors, actuators, and other intelligent electronic devices (IEDs). These systems had no connectivity to a greater network of sensors or devices. Eventually, these SCADA systems developed into decentralized systems, with one or more master terminal units (MTUs), and one or more remote terminal units (RTUs) [4]. These RTUs were connected to a larger network of MTUs using proprietary, industrial communication links. Following a period of decentralization in SCADA architecture, these networks transitioned from using proprietary hardware, software and communication links, to using commodity, commercial and public hardware, software and communications links, gaining larger connectivity to a network of SCADA devices. These networks gradually became integrated with larger corporate intranets, and the Internet [5].

As these SCADA networks became part of a larger network of standardized hardware and software, the communications protocols used were also standardized. The amount of information that these early links were able to carry limited the amount of information that could be expressed using these protocols. As a result of the initially proprietary nature of the links, the bandwidth limitations of the links, and the real-time nature of the processes being controlled by SCADA systems, these protocols were not designed with any built-in security features. The protocols currently in broad use, including ModBus, and IEC 60870-5-101 do not include any inherent security features, while DNP3 includes provisions for security which remain in draft form [6, 7].

### *2.1.1 SCADA Security Issues*

SCADA security hasn't been a large concern through the history of SCADA systems. SCADA systems initially used proprietary hardware and software on closed networks. As a result, security was provided by the obscurity of the system, and the lack of general knowledge as to how these systems operated. As a result, a majority of SCADA security threats came from employee sabotage. As SCADA system have shifted from obscure, closed systems, to open, commodity-based systems on open networks, the attack vectors have appropriately shifted. A study conducted by Byres and Lowe in 2003 found that attacks from insiders represented 38% of SCADA security breaches in 2000, with external attacks accounting for 31%, while in 2003, internal attacks represented only 5% of security breaches and external attacks accounted for another 70% [8].

In summary, network connectivity and standardization has infiltrated the industry in a bid to increase awareness and control. While this has decreased costs, it has also resulted in the pervasive distribution of SCADA systems which are susceptible to electronic attack [9]. In addition to using hardware and software which contains known and unknown vulnerabilities, the documentation for these systems is now freely available [10], showing that security through obscurity is no longer a trustworthy method for maintaining the integrity of SCADA networks.

### *2.1.2 Securing SCADA Systems*

With the general acceptance that SCADA systems have inherent insecurities which are acting as vectors for outside attack, it is crucial that strategies be developed in order to solve these problems. Because of the commodity hardware being used in these SCADA networks, many sources suggest using common information technology (IT) based security practices, such as enabling WEP on wireless devices [11], increasing network connectivity in order to provide more resiliency to attack [10, 11], implementing role based access control [12], and segmenting the SCADA networking using firewalls and virtual private networks [10, 11, 13].

While these methods provide some security, they are insufficient by themselves. They fail to properly differentiate between the security needs of a SCADA control network and a typical corporate network. Although the resources used in creating and managing these SCADA networks are increasingly similar to the solutions used in traditional corporate IT environments, these solutions fail to understand the key differences between SCADA systems and IT based systems.

SCADA systems require high data integrity, high up-time and high security at all points in the network. If any leaf of the SCADA network (any RTU, or MTU sub-tree) fails, then the entire SCADA system may have its operation compromised. The IT based solutions focus on high data throughput, with a tolerance for data corruption or loss, and principally focus on the confidentiality of data. As a result, adopting these IT based solutions in a SCADA environment can be costly, and may not provide the necessary protection. IT based solutions generally focus on making the core of the network secure, reliable, and robust, while security among the leaves of the network is allowed to languish. In the SCADA network, this cannot be tolerated because the leaves of the network connect to physical control systems. Unfortunately, the vast majority of the solutions being deployed and developed today focus on these IT based solutions.

Other systems architectures advocate securing the communication links between SCADA devices with drop-in encryption modules [14]. These modules provide a reasonably high level of protocol-layer security without adversely affecting performance, as the IT based solutions might. Unfortunately, these modules only encrypt the communication link, and do not guard against vulnerabilities in the hardware or the commodity operating systems running on these SCADA devices. Furthermore, these devices add a focus on data confidentiality, while data integrity is usually a higher priority for SCADA: an attacker need only understand that these link encryption modules block intrusion at only one layer of the SCADA system stack.

## 2.2 Architectural Security In Kernel Models

Unfortunately, the basic design architecture of using off-the-shelf operating system software adds an inherent level of risk to the system in question. Traditional operating systems are hugely complicated pieces of software. The operating system itself can be logically divided into the sections which operate in user mode, and the sections which operate in kernel mode. The sections which operate in kernel mode provide direct abstractions to the hardware that the operating system is running on. As a result, the kernel mode code has special access to the hardware of the system, and operates without the safety of memory segmentation and protection. The kernel mode code of any operating system must be part of that system's trusted code – that is, the kernel mode code must be trusted to perform its duty with certainty and integrity.

Unfortunately, the majority of commodity operating systems today use a monolithic kernel architecture, including massive amounts of code in the kernel of the operating system, such as file system code, device driver code, and many many other services. The Linux kernel, as of version 2.6.35, contained over 13.55 million lines of code[15]. Some sources estimate the number of lines of code in Windows XP at 40 million[16]. More code is a direct vector for taking advantage of a system. Hackers with an understanding of the systems being used can exploit bugs, both known and unknown, and larger systems inherently contain more bugs. Various sources place the average number of bugs in a given piece of code between 2 and 75 bugs per 1000 lines of code. [17, 18]. Even conservative estimates at the number of errors in a code-base this large place the number of bugs in the tens of thousands. As a result, there are inherent insecurities and instabilities in operating system kernels of this magnitude.

### *2.2.1 Separation Kernels And Microkernels*

There is an alternative to this monolithic kernel architecture. A microkernel is a kernel architecture which seeks to provide minimal kernel level abstractions to the hardware of a given platform. As a result, these kernels provide minimal services, and force items like file systems and device drivers to operate in user mode, without privileged access to all the resources of a system.

Dr. Jochen Leidtke formalized this concept of minimalist design thusly:

> *A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e., permitting competing implementations, would prevent the implementation of the system's required functionality.*[19]

As a result, modern microkernels provide very few services familiar to monolithic kernels. Figure 2.2 shows some of the differences between monolithic kernels and microkernels.

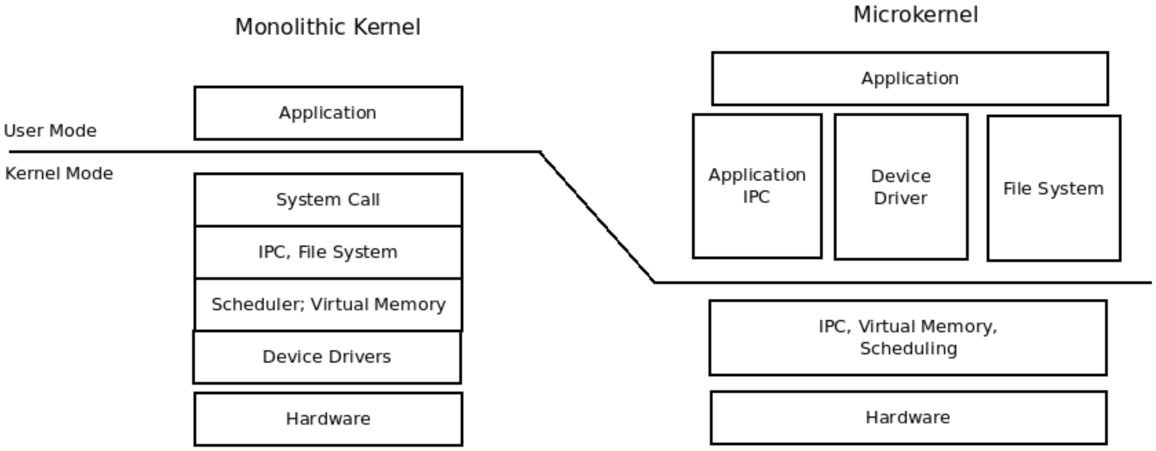

*Figure 2.2: Monolithic and Micro- Kernel Architectural Differences.*

In Liedtke's implementations, these services were limited to management and protection of memory spaces; thread creation, destruction, management and scheduling; interrupt management; and inter-process communication. Due to these cutbacks, a microkernel can be implemented in under ten thousand lines of code. This immediately reduces the number of possible bugs which can be used as an attack vector by several orders of magnitude. Additional services must be implemented in user-mode, but much of this code need not operate in a completely error-free manner in order to provide a secure system. Tanenbaum, original author of the MINIX operating system, advocates the implementation of microkernels as the basis of a secure and reliable system[20].

Although the concept of placing critical operating system services in user space has existed since at least the 1960s [21], UNIX and BSD remained the dominant operating systems of the era, and these operating systems made a design decision to place services such as device drivers and file systems in the kernel. As a result of the popularity of UNIX and BSD, many other operating systems followed suite in this monolithic design.

In the 1970's, Tymshare Inc. began development of a new operating system called GNOSIS, (The Great New Operating System in the Sky) which would evolve into KeyKOS, and later EROS, as the concept and source code were bought and sold. The EROS operating system, a self-claimed "nanokernel" operating system, was designed to be an extremely reliable and secure operating system. This operating system provides security and separation through the communication of "capabilities." These capabilities are unforgeable representations of the rights which a program has for a specific operating-system object.  Although the foundations of EROS were laid in the 1970s, development and implementation of these concepts took place primarily in the late 1980s and early 1990s.  EROS has two successor projects, CoyotOS and  CapOS.  These projects are lead by two of the principal developers of the EROS project..  As of 2010, development for both these projects remain inactive.  Despite being proclaimed a "nanokernel," the source tree for the final release of EROS is over twice as large as modern microkernels.

To block all layers of possible intrusion, an operating system known as a "separation kernel" was proposed as early as 1981 by  Dr. John Rushby [22].  This kernel would treat software components as if they were pieces of a physically distributed system.  Each piece of software must have no interaction with software extant to itself, save for a known quantity of communication paths.  Thus, the flow of information between components of a separation kernel system may be completely known.  As of 2010, only a single separation kernel has received any sort of formal certification [23].

In 1984,  development began on the Mach kernel at Carnegie Mellon[24]. This kernel is one of the first examples of a kernel designed with microkernel principals throughout.  Unfortunately, the purpose of this kernel was to support operating system research, primarily in the areas of parallel and distributed computing.  As a result, this kernel was not designed with minimality and security in mind, but rather sought to emulate the UNIX kernel in whole.  As a result, the source tree became extremely large, the result of attempting to fit monolithic kernel attributes into the microkernel architecture.  Additionally, during the early 1990s, CPU speed grew at over 60% per year, while memory speed grew at only 7% during this same period.  The necessity of switching in and out of kernel mode in order to perform routine microkernel tasks like memory space management and interprocess communication requires more access to memory than a comparable monolithic kernel. As a result of this rising difference between CPU and memory speed, performance of microkernels waned. Between the increasing size of code, and the waning kernel performance, the interest in microkernels spurred by Mach and other projects declined.

Roughly concurrent to the development of the Mach kernel, Tanenbaum released MINIX, a microkernel operating system designed to be a minimal implementation of a UNIX operating system. This operating system was designed to be a companion for a textbook on operating systems. As a result, security and minimality were not design goals. The first version of the operating system, however, was implemented in only 12,000 lines of code, which is comparable with modern microkernels.

The QNX operating system is another example of a microkernel based operating system. This operating system began as a research project at University of Waterloo in 1980. This project has evolved, and changed hands many times, however. Some versions of the QNX operating system have been available as open-source products, while others have not. The most recent version, which has received a certificate of security evaluation, is currently closed source and owned by Research in Motion[25]. QNX has seen used in many high reliability systems. Although this operating system has been proven commercially, and achieves similar performance to the L4 Microkernel, its certificate of verification is based on informal criteria of evaluation.

Following the decline of the Mach microkernel, Liedtke began research on a new generation of microkernels. Throughout the mid 1990s, until his death in 2001, Leidtke developed several new microkernel paradigms that would prove that microkernels were not required to perform badly [21, 26]. The L3 and L4 microkernels achieved their performance goals by tightly coupling the algorithms used to perform memory space management and IPC to the target system architecture. Earlier work had focused on creating a microkernels which weren't bound to a specific set of hardware, as well as creating a kernel which could place important services in user mode. Liedtke's kernels were the first to be designed specifically with performance in mind. The L4 operating system was written primarily in assembly language and C. Porting this operating system to other architectures was a daunting task. The L4::Hazelnut project was the first to successfully translate the L4 operating system into C++ without a great performance penalty. The L4::Pistachio project would further build on the success of the L4 kernel, and generalize the architecture specific algorithms and programming interface, decoupling the kernel code from the target architecture, with the exception of only a few required assembly-language function implementations [29].

### *2.2.2 The OKL4 Microkernel*

The OKL4 Microkernel is a descendant of the L4::Pistachio kernel, implementing a capability based security system similar to the one implemented by EROS. This microkernel is a good example of a third-generation microkernel [27]. In order to more fully explain the operating principals of a modern microkernel, OKL4 will be used as an example. The current version of OKL4 is Version 3.0, and is available as open-source software from NICTA.

The OKL4 microkernel, as a direct descendant of Liedtke's L4 kernel, implements minimal operating system abstractions, including memory space management, thread management, and interprocess communication (IPC). Additionally, the OKL4 kernel exports management of kernel resources to user mode processes using a modified, simplified capability system from the EROS operating system. These capabilities represent a non-forgeable token of an object, as well as the program's permissions to operate on or with the object. As a result, memory spaces, thread, and IPC can be controlled from user mode processes with a measurable assurance of security [28].

In the OKL4 operating system, threads may have one or more memory spaces mapped to them. These memory spaces present themselves as virtual memory to the thread. This virtual memory, however, must be backed by physical memory allocated to the OKL4 kernel or a delegating user-mode program. The specific mapping of this memory is known only to the kernel, or a user-mode memory delegation implementation. As a result, in order to learn about this specific memory mapping, a thread must have appropriate capabilities, ensuring that threads cannot inspect or modify memory to which they have no capabilities. Threads are switched in a round-robin fashion, based on priority.

Interprocess communication (IPC) is the basis of interrupt management, remote procedure call (RPC) implementations, and data transfer between threads. In OKL4, IPC is abstracted depending upon its specific use. In all cases, in order for IPC to take place, the communicating cell must have capabilities to the recipient cell. If so desired, the communicating cell can transmit temporary “reply” capabilities to the recipient cell in order to receive a response. These reply capabilities are temporary. If IPC takes place between threads in the same memory space, then a full kernel-mode switch need not occur, saving CPU cycles. If an IPC takes place between threads existing in different memory spaces, then the OKL4 operating system uses a system-specific implementation of the IPC algorithms pioneered by Leidtke in developing the L4 microkernel, and later L4::Pistachio. Determining the fastest method of transmitting information between threads is very important in a microkernel environment.

Performance completely drives the design decisions, since in a microkernel environment, every IPC transfer between memory spaces requires a transition into kernel mode and back again, which requires a large number of CPU cycles. Interrupts are handled just like any other IPC. The cell which is configured to receive a given interrupt must have capabilities to this interrupt. As a result, whenever this interrupt is triggered, the kernel will send an IPC message to the thread receiving the interrupt. Requiring IPC to trigger interrupt service further drives the need for fast IPC. It is also worth noting that particular implementations of the IPC API and algorithms used by Liedtke are strongly architecture dependent. A presentation by Gernot Heiser in 2008 summarizes the actual L4 IPC performance for different architectures as shown in table 2.1.

*Table 2.1: Cost of Minimal L4 IPC Transfer on Various Architectures[29]*

| **Architecture** | **Intra Address Space** | **Inter Address Space** |
|---|---|---|
| Pentium | 113 Cycles | 305 Cycles |
| AMD-64 | 125 Cycles | 230 Cycles |
| Itanium | 36 Cycles | 36 Cycles |
| MIPS64 | 109 Cycles | 109 Cycles |
| ARM Xscale | 170 cycles | 180Cycles |

### *2.2.3 The SeL4 Kernel*

In addition to providing fewer possible vectors for software-level attack, these second generation microkernels are small enough to go through a process of formal verification, whereby the code implemented by the operating system is proven to operate without any bugs by a set of mathematical proofs. With only a few thousand lines of code, and a minimal number of operating system hooks into user space, this verification is no longer as daunting as it would be if performed on a code base of 15 million lines of code. Researchers at National Information and Communications Technology Australia (NICTA) have developed an operating system based on L4::Pistachio called the seL4 kernel. After over seven man-years of labor, they have reportedly performed a formal verification on this microkernel, with over two hundred thousand human-and-computer generated proofs.[30]

This verification is a tremendous leap forward in creating secure and reliable computer systems. However, it is not without assumptions. One must trust the compiler, the hardware, and the proof generator. The compiler used for this verification is a derivative of the freely available GCC C-compiler, which is monolithic enough that a formal verification would be daunting, at best. Gerwin Klein has performed some initial work at NICTA in the use of a verified C compiler to compile the seL4 kernel [26], which shows promise at being a viable alternative to GCC. The seL4 kernel is designed to adhere to most ANSI C standards. This avenue is one road that could be taken to extend the research being performed at NICTA. Klein suggests that another avenue is to begin the implementation of large-scale trusted systems, such as the long-theoretical MILS architecture. [31]

As a result of this verification, secure, trusted software can be built on top of the kernel, with the separation between user-mode memory spaces abstracted as trustworthy. This makes the creation of a minimal trusted computing base (the parts of code which can bypass and compromise system security [32]) possible. Now, the formal verification of a minimalist TCB can build on the verification of the kernel itself, to create a larger secure system.

### 2.3 A Secure Industrial System Using Microkernels

As a result of the minimalist nature of these microkernels, and the formal verification to which they can be subjected, they can be used to create secure embedded systems. Demanding real-time requirements exist in almost all SCADA applications. IEEE and IEC standards state that many SCADA applications require a response time as low as 2-4 ms [33]. As a result, the microkernel based solution cannot be performance bound. Initial research in the implementation of a microkernel based industrial control system suggest that this is not an impossible task, with raw IPC times as low as 69.54

microseconds [1]. Additionally, the seL4 microkernel represents a basis upon which a secure separation kernel can be built, implementing with measurable certainty, a mathematically proven method for implementing such an architecture.

# CHAPTER III

# DESIGN APPROACH

This chapter details the design goals of the security hardened RTU prototype based on the architecture described by Hieb and Graham [1]. Many of these design goals are derived from earlier work, as described in Chapter II. While the body of work contained in Chapter II describes the security vulnerabilities in existing SCADA systems and explores the possibilities for a security hardened RTU based on microkernel operating systems, this chapter details a specific target architecture, designed with consideration for the needs of actual development, as well as the hardware available. Section 3.1 outlines the architectural model, developed by Hieb and Graham [1] of a security-hardened RTU, with respect for the possible avenues of implementation, and Section 3.2 details some of the prior work performed in creating an RTU with this architecture. Section 3.3 details how OKL4 design paradigms may be utilized for for the specified implementation, Section 3.4 details a simplified SCADA communications protocol created for testing the security features and performance of the RTU, Section 3.5 explains the operation of interprocess communication (IPC) and the communications paths through the RTU, Section 3.6 explains the usage and design of a challenge-response algorithm, and Section 3.7 describes the role based access control model utilized in this prototype.

## 3.1 Architecture Model And Security Features

The security hardened RTU is based on an iterative design model proposed by Hieb, Graham and Patel in a series of papers on possible security enhancements for

SCADA systems. Through several papers and dissertations, this design model became increasingly specific to the architecture which could be implemented under the OKL4 operating system, as described in section 2.3.1. The model targeted at the beginning of the project was described by Hieb and Graham in [1], as shown in figure 3.1.

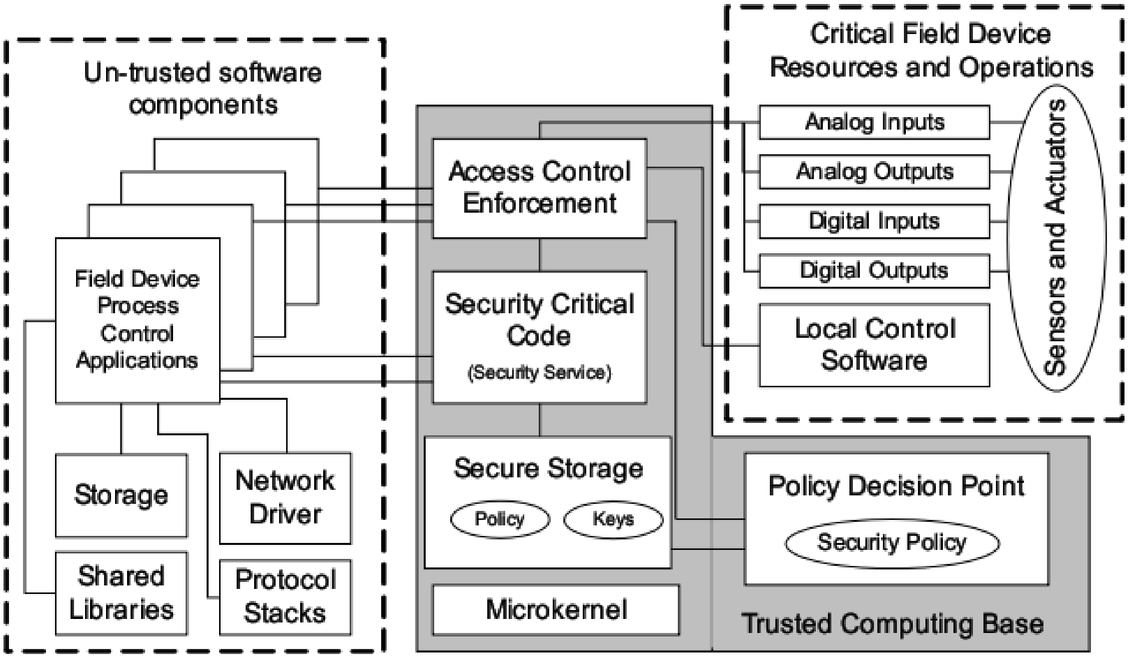

*Figure 3.1: Target Architecture from [1]*

Research performed by Graham, Hieb and Patel, among many others (See Section 2.1) has shown that the current monolithic kernel architectures contain inherent insecurities which cannot be easily mitigated through software. The above architecture, developed by Hieb and Graham is an alternative to the predominant architectures. In this new architecture, all components need not reside in the trusted computing base (TCB). The security benefits of such an architecture have been reinforced by Klein, among the primary developers of the OKL4 family of operating systems [31], in describing the use of a system very similar to this for providing security to embedded devices in a more general sense (not just SCADA systems).

The architecture modeled in figure 3.1 visualizes the software infrastructure of a security hardened RTU. There are three isolated components: Network IO, Security, and Physical IO. This architecture decouples principal RTU functions from each other. The Network IO layer is part of the untrusted computing base (UTCB), and is a generic interface to the physical method of communicating with the RTU, whether this is industrial Ethernet, RS485, wireless data, or some other medium. This portion of the UTCB is only capable of sending inter-process communication (IPC) to the Security segment. There is no direct path of communication or memory manipulation between the Network IO and the Physical IO.

The use of a microkernel based operating system opens the possibility of creating a trusted computing base (TCB) which can be formally and rigorously verified. The security layer contains all of these necessary trusted components, including the kernel code. The security layer is capable of communicating with both the Network IO and the Physical IO. As a result, this trusted layer must mediate all communications between these two untrusted components. This layer ensures the integrity of clients across the network layer with features such as hashing and handshaking. The security module also ensures that clients posses the appropriate permissions to perform a given action, using role based access control. Although these functions are linked logically, they are, in fact, separate; hashing and handshaking verifies the integrity of the communication path, ensuring resistance against threats such as man-in-the-middle and replay attacks, while the role based access control layer ensures that the end user is performing appropriate actions (a vendor, for example, cannot change set-points, but may be able to view operations). Since the security layer is part of the trusted component of the system, it must be fast. In order to be feasibly verifiable, it must be small. The security layer must be built to facilitate later verification of functionality.

Finally, the physical IO layer consists of software which interfaces with a set of external hardware in order to generate outputs and accept inputs from the connected control system. The code to operate this subsystem is untrusted, and may only communicate with the security layer. Otherwise, the hardware and functionality of this portion is comparable to currently available and legacy commercial RTUs.

This architecture may be realized using the OKL4 operating system described in Chapter 2. As described in section 2.2.2, the OKL4 operating system makes use of memory segments, protection domains, and zones. These are combined in a logical unit called a "cell." These cells are logically separate– in fact, during the compilation process, they are compiled and linked as though they were independent programs. A utility called "`Elf Weaver`" combines these separate programs with the kernel code to create a bootable system image. Thus, when considered with the verification model of the OKL4 kernel describe in section 2.3.2, we can understand a cell as a realization of the kind of separation architecture described above, with the exception of two trusted "cells"– in OKL4, the kernel remains in a separate memory space and protection domain.

Each of the segments described above resides in its own cell. The physical IO resides in a cell with the remainder of the untrusted computing base. This includes utility functionality, such as hardware management, and debug interfaces. Much of this functionality is a side-effect of creating a device which must be intricately monitored for development reasons. Security must reside in its own cell to perform trusted IPC with network and physical IO. Network IO resides in a third, separate cell. Utility functionality, such as hardware management, debug interfaces, and test code reside separate from network IO due to the complexity of the network IO code and the desire to keep possible bugs as deconvolved as possible. This design decisions simplifies development, and is otherwise arbitrary.

### 3.2 Prior Work

This section highlights and details some sections of Chapter II which are of particular import to the development of the prototype using the OKL4 operating system, based on an architecture described by Hieb and Graham [1]. In 2007, Graham, Hieb and Patel investigated the SCADA security issues described previously, and reached a number of conclusions regarding possible directions for future research, briefly investigating the feasibility of minimizing an existing real time operating system kernel, developing enhanced SCADA protocols, and implementing role based access control on top of existing access control and security-hardening layers [34]. Hieb and Graham continued this line of research in 2008, exploring the benefits that an isolation kernel (described briefly in 2.3.2) would provide to a security enhanced SCADA system[1]. Importantly, they also performed initial experimentation using the OKL4 operating system, exploring the speed of IPC calls on real hardware. These investigations served as initial steps at developing the architecture, demonstrating that IPC overhead would not be a limiting factor in development. Hieb further described the architecture modeled above, and continued to describe (in great detail) a role based access control system  for SCADA systems, providing a framework for a role based access control system suitable for the prototype RTU considered herein [35].  A patent application has been filed for this technology. [36]

### 3.3 Isolating Components With Cells And Threads

As mentioned above, there are two primary methods of dividing up the computing resources of a system running the OKL4 operating system – cells and threads.  Threads in OKL4 are identical to threads in other operating systems.  A program executed as a thread shares time on the CPU by well-understood principals of context switching.  A cell, on the other hand, is a combination of many other constructs in the OKL4 Operating System. “Cell” is a glossary term, and does not actually exist as a programmatic interface in the operating system.  A cell is a combination of a memory segment and protection domain, within which runs one or more threads.

A memory segment is simply an allocation of virtual memory, carved from the larger physical memory of the system.  The address spaces of this virtual memory maps in a one-to-one basis with the physical memory, and must be page-aligned with the physical memory, but its addressing schema is different.  A protection domain is a set of memory segments which are isolated from all other memory segments on the system.  Memory segments inside a given protection domain have no way of being accessed from outside, and threads executing inside a protection domain have no access to memory segments outside their protection domain.  As a general rule, memory segments cannot be mapped into more than one protection domain.  Although there are exceptions to this, the utilization of these features is not necessary for the completion of this project.

The distinction between physical and virtual memory is important. In the OKL4 operating system, all non-kernel operations exist in virtual memory space, which allows for the levels of isolation which make it possible to implement critical security components. The hardware utilities and various low-level faculties of the device are accessed using memory-mapped hardware. The configuration and data registers for these various devices reside in physical memory. This physical memory can be mapped into the virtual memory space, and the running program can discover its location at runtime without a great deal of trouble. Unfortunately, the same set of physical memory cannot be mapped in a read/write manner to two protection domains, which enforces this verified memory segment protection. As a result, there are some utility functions, such as power and clock management, which must reside in a central location. In the case of functions like these, a hardware manager thread is necessary. This hardware manager responds to requests over IPC from other cells, configuring common memory-mapped peripherals, and responds when these actions are complete. Compare this to a traditional (macro) kernel in which all driver operations have unfettered access to the entirety of physical memory.

For the purpose of debugging during development, it is necessary to create a thread with access to a common serial output. This serial output provides a way to display debug messages from multiple cells concurrently. This debug data can then be transmitted through IPC to the serial debug thread, which then utilizes the hardware of the device to generate actual output. Otherwise, the physical serial device would map to a single memory space, limiting the debug output. This becomes an issue when trying to debug IPC communications calls across multiple threads and protection domains.

Threads in OKL4 must execute within the context of a protection domain. More than one thread can exist within a protection domain, and if this is the case, each thread has unfettered access to the virtual memory mapped within that protection domain. From 3.1, a deliberate decision was made to place utility functionality separate from network functionality, in the same cell as the physical IO. This is partially due to the complexity of the network code. For the development phase of the project, it is desirable to minimize the possible source of multiply conflicting, convoluted bugs in the network code.

Threads in the OKL4 operating system can either be declared and initialized at compile time, using the elfweaver utility, or created at run-time. Creating threads at compile time is relatively simple. These threads begin at the start of execution, and continue until they terminate themselves. Capabilities, required to send IPC calls to these threads, can be declared at compile time, and assigned to other threads with which they must interact. Unfortunately, there is no built-in manner to spawn, fork, or externally terminate new threads of this type. Run time threads are better suited for applications requiring these features. Run time threads are created by another thread. This originating thread must carve out a memory space for these new threads, and must take care of their operation and termination. The major down side to this method is the creation of capabilities, which are required to communicate with the new threads. Upon starting, only the originating thread has any knowledge of the capabilities required to communicate to these threads. As a result, in order for disparate cells and threads to communicate, the originating thread must express this capability through IPC. As the features of run time threads were not required for the construction of the target architecture, they were not used in favor of the easy of compile time threads.

In summary, for the purposes of development, there are three cells implemented in the prototype. The network IO and security cells run a single thread each, while the utility cell operates three threads-- one for network IO, one for hardware management, and one for serial debug. All threads, across all cells run at the same priority level. These threads are context switched in a round-robin fashion, without respect for their given protection domains or memory segments.

### 3.4 Inter-Cell Communication Using IPC

All communication between components must occur through IPC, and the IPC structures in the OKL4 operating system are critical to the implementation of more complex structures, such as remote procedure call (RPC). The rules of communication must be well planned out and understood before development. Due to the complexity of initializing hardware in an operating system which provides such strong protection across memory segments, the startup sequence must be planned separate from the steady-state IPC methodologies. There is a large amount of synchrony between and among cells which must be planned for and understood in order to avoid race conditions which could cause the RTU to deadlock, either in operation or during startup.

There are two types of IPC calls in OKL4: blocking and non-blocking. Blocking calls will halt a process until the conditions for sending or receiving a given IPC are met. Non-blocking calls will attempt to deliver or receive an IPC message, and if the parter in the communication is not ready, the attempt will fail. There are two communications primitives: send and receive. In order for a thread to send data to another thread, it must have a capability to the target thread. The capability is an object which can be passed from thread to thread, or declared at compile time. This capability is a combination of both the location of the receiving thread, as well as permissions to access this thread. The sending thread may opt to send "reply" capabilities alongside an instance of IPC. These reply capabilities are temporary capabilities to return data to the sending thread. They are invalid after the conclusion of the IPC.

These send and receive primitives combined with the ability to transmit reply capabilities can be combined in complex ways in order to achieve the security goals of the project. For the security hardened RTU, only blocking primitives are necessary. The flow of IPC through cells can be modeled as a finite state machine. Because of this, threads have no need to continue operating while they are awaiting a message,

As mentioned above, there are two "time periods" of interest for IPC. During startup, the hardware manager must enable hardware and alert cells to the initialized hardware in an order which does not halt the system. Race conditions must be avoided. During startup, the communication taking place is as shown in Figure 3.2.

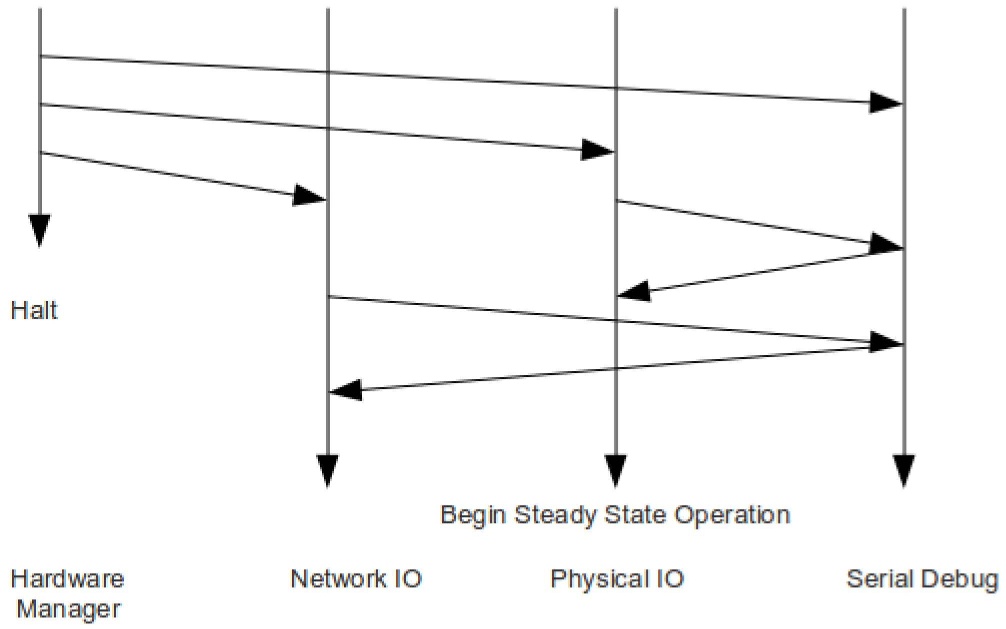

*Figure 3.2: IPC messages during Start-up*

The hardware manager initializes hardware for serial debug, physical IO, and network IO. The hardware manager then permanently halts. Subsequently, the physical IO and Network IO communicate with the serial debug thread. All threads then enter a wait state. The serial debug may receive an IPC message from any thread, and will output this message to the terminal. Blocking "send" calls are used in order to ensure that all debug output will be seen. If a cell cannot deliver its output, it will wait until the serial debug is available. As a side-effect, however, the serial debug must be stable. Any crashes in the serial debug thread will halt the system. This ensures that testing and debugging remain consistent and simple, but provides a method for halting the system abruptly. In a production device, there would be no serial debug, so this is not a security issue. The serial debug is simply a reality of development. During steady-state, the chain of IPC events can be modeled as in Figure 3.3.

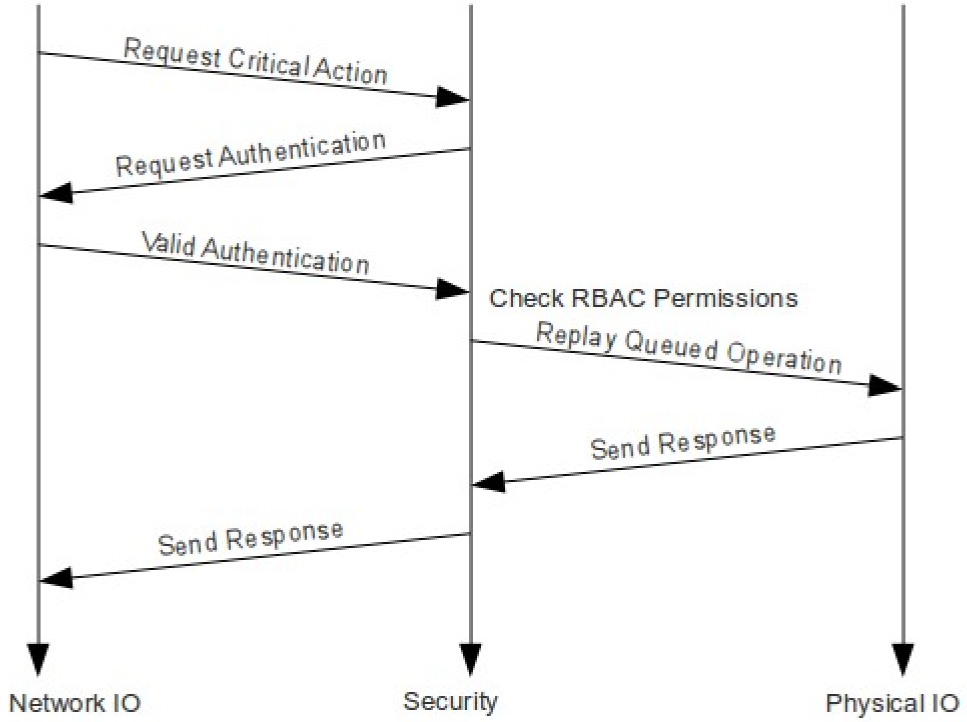

*Figure 3.3: IPC from Network Event to Physical IO, given proper role based access control credentials*

In figure 3.1, a message is delivered to the network cell, and relayed to the security cell. The security cell flags this as a critical point, and demands that the request be authenticated. It relays this demand to the network IO layer. A valid authentication is returned and verified by the security cell. The queued operation is then internally recalled, and its permissions are checked inside the security cell. These permissions are valid, and the security cell passes the command to the Physical IO layer, which returns some data. This data is then passed through the security layer to the network IO cell for relay to the original client.

Figure 3.4 shows a similar attempt to request some data. This time, the given RBAC credentials disallow the operation. As a result, the security cell replies to the network cell to re-enter its wait state, without sending a response. From these two examples, it's very easy to see the statefullness of the system. Figure 3.5 shows all possible states.

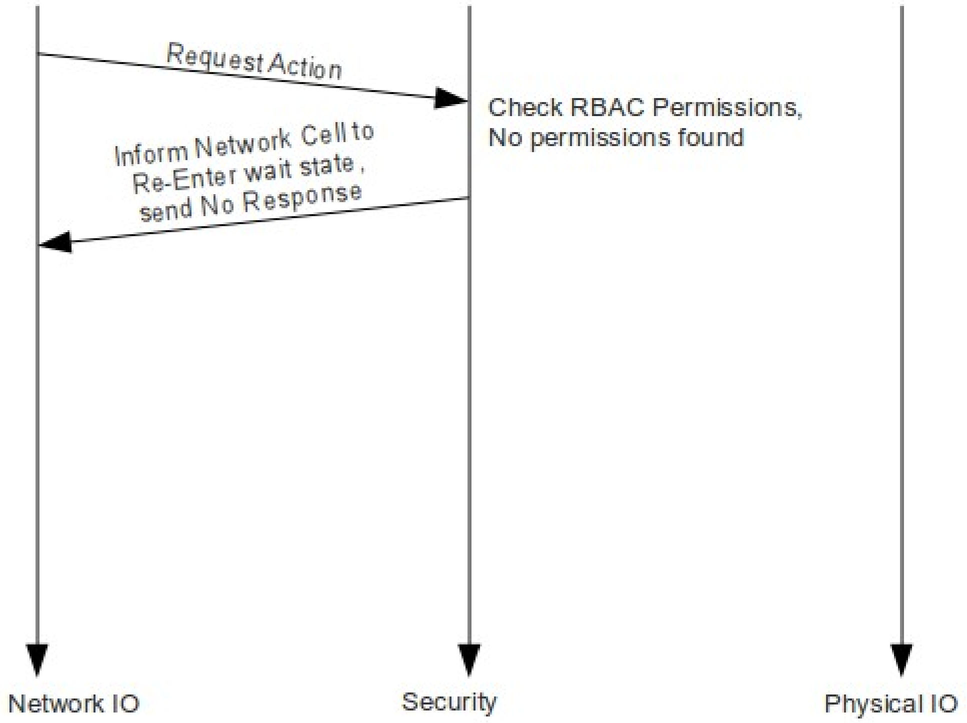

*Figure 3.4: IPC from network event to physical IO, given improper role based access control credentials*

This IPC ties the handshaking and role based security features together. All of these operations are performed in the security cell, after being received by the network cell.

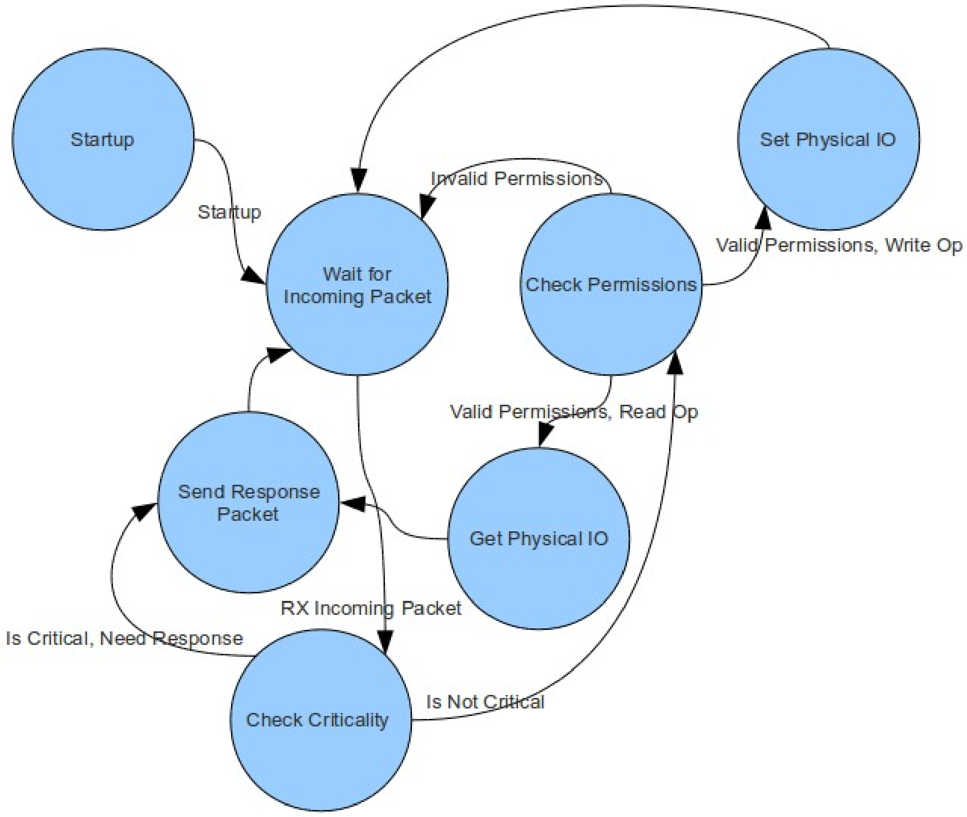

*Figure 3.5: Full state diagram for IPC.*

### 3.5 Security Features

The security hardened prototype will utilize several layers of security which have been implemented on devices in the past. With the ability to place these features inside a verified trusted computing base, these features become much more trustworthy. First, a simplified SCADA protocol was developed. This protocol reasonably emulates commercial SCADA protocols, allowing for easy expansion with new security features, which can then be tested for security, performance and reliability. This protocol is used internally and externally. External transit is provided by the UDP protocol. A standard handshaking authentication method is used to verify the integrity of users attempting to use the communication link, and a role base access control layer ensures that they users are permitted to perform the actions they are requesting.

### *3.5.1 A Simplified SCADA Protocol*

Modern SCADA devices utilize a large number of different communications protocols over a large number of different potential communications media. The RTU prototype will communicate solely over Ethernet. In order to test the security and performance of this RTU, a full suite of SCADA functionality is unnecessary, and developmentally burdensome. Protocols like ModBus and DNP3 can very quickly become complicated, and the prototype RTU will not support the majority of these protocols' features, in any case. For development and testing, a very basic SCADA protocol is needed.

This protocol must implement the minimum number of features necessary to properly emulate a typical RTU, and remain extensible for the addition and testing of security features. SCADA operations affect control points, and control points may be either read, selected, or operated upon. Modern SCADA protocols require a select followed by an operate (with identical data payloads) in order to alter the setting of a given point. In addition to this, we must generate responses to read request, and implement our hashing and handshaking algorithm. Operate and select operations do not require a response. In order to verify the operation of an operate or select, the user performs a  read operation. This write-then-read paradigm is common to many SCADA systems, and is not unique to this prototype. With this description, our simplified SCADA protocol must implement only five operations. Protocols such as ModBus and DNP3 derive their respective complexities through support for complex addressing, CRC and data integrity verifications, and standardized maps of points and operations and implementations of a variety of  standardized data expression techniques.

In order to implement the role based access control mechanisms described in section 3.7, each operation must be associate to a specified user. Beyond this, the data required for an operation is dependent on the operation. A read operation requires only the point which must be acted upon. A select or an operate operation, meanwhile, requires both a point and data. Read responses include only return data, without notification of which point the read response is for. Challenge requests include a payload of a server nonce, while  challenge responses include a client nonce, as well as a SHA-256 hash.

For the prototype, the payload size for user identification will be one byte, point identification will be one byte, point data will be one byte, nonces will be 4 bytes, and the SHA-256 is 32 bytes. The total size of the packet payloads, as well as operation ID is summarized in Table 3-1.

*Table 3.1: Summary of Simplified SCADA Communications Protocol used in RTU Design*

| Operational Type | Operation ID | Data |
|---|---|---|
| Read Point | 0x00 | 3 bytes |
| Select Point | 0x01 | 4 bytes |
| Operate Point | 0x02 | 4 bytes |
| Demand Challenge-Response | 0x03 | 4 bytes |
| Challenge-Response | 0x04 | 36 bytes |
| Read Response | 0x05 | 4 bytes |

The data size indicated above includes the user and point ID bytes, but does not include the operation ID byte.

For reading a point, the data format is:

<Operation ID>+<User ID>+<Point ID>,

each a one-byte field.

For select or operate operations, the format is:

<Operation ID>+<User ID>+<Point ID>+<Data>

For initial testing, Select simply operates as a "write" operation, although typical SCADA systems require a select followed by an operate, containing identical data.

For demand-response operations, the format is:

<Operation ID>+<Server Nonce>

with the server nonce being a 4 byte field.

For challenge-response operations, the format is

<Operation ID>+<Client Nonce>+<Hash>

The client nonce is a four byte field, while the hash is a four byte field. Although the SHA-256 hash is 32 bytes in length, many commercial SCADA devices compare only the first four bytes for authentication. In this implementation, neither the client nor the server require the originating message to be hashed along with the nonce. This was excluded for simple testing and rapid development, but is a trivial addition given the current state of the code base.

For read response operations, the format is

<Operation ID>+<User ID>+<Point ID>+<Data>.

In all of the above cases, '+' indicates bit concatenation. There is no delimiter of information, other than the knowledge that each field is a strictly fixed length.

Below is an example of a “Read Response” packet, sent using user ID two, from point ID ten:

| 0x05 | 0x02 | 0x0A | 0x01 |
|---|---|---|---|
| Read | User ID | Point ID | Returned Data |

This information will be wrapped in the payload of a standard UDP packet. The port 1200 is selected arbitrarily. UDP is selected due to its simplicity and ubiquity in modern Ethernet networks. A stateless UDP engine can be trivially constructed, and many, many programming languages and development environments include methods for interacting with standard UDP packets. Additionally, these UDP packets are routeable across the larger Internet.

On the RTU, the code which sends and receives these simplified SCADA packets is very simple. The decision tree for this functionality is shown in Figure 3.6.

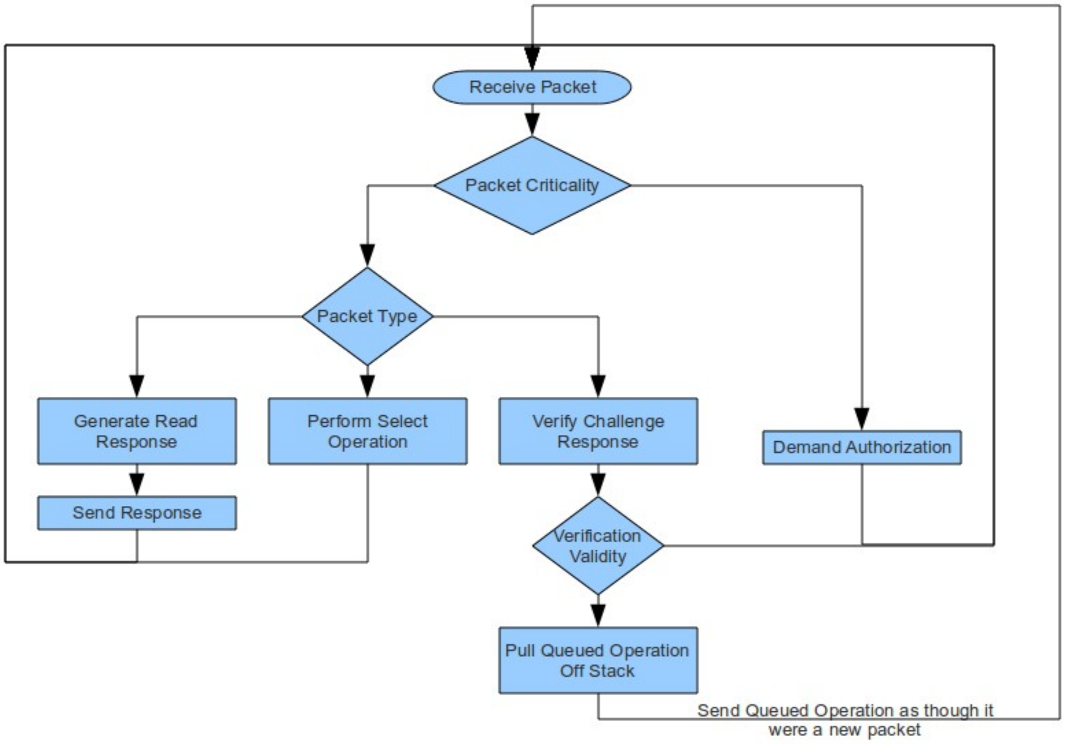

*Figure 3.6: Decision tree for SCADA packet reception.*

As can be seen, there are only a handful of decisions points in the packet reception mechanism. This greatly simplifies the network layer code required. This decision tree is simplified and does not represent the role based access control decisions being made by the security layer. This diagram also does not represent the role the security cell must play in generating hashes and performing handshaking. The simple SCADA packet reception code does not need any awareness of state in order to perform its duties– it is not required to keep track of authentications, points read, or any other state. For read requests, the RTU will send a response, and allow the client to sort out which packet for which response is directed. For handshaking, the requested operation is placed on a queue, and if the client responds with valid authentication, this action is removed from the queue, and placed through the simple SCADA decision tree as though it were any other packet. Failed attempts to authenticate are simply ignored.

### *3.5.2 User Authentication And Handshaking*

The method for determining the authenticity of the clients is a simple challenge-response handshake using a pre-shared secret. This is computationally simple, and actively used in commercial SCADA devices. The only prerequisites for this method of authentication are the ability to generate random numbers, and the ability to perform a SHA-256 hash. The algorithm itself is trivially simple.

For the purpose of the prototype, it is sufficient to demonstrate that hashing can be performed, and that it is both fast and reliable. For the prototype, hashing is only performed on points which are declared "critical." If a critical point is manipulated, an authentication is demanded of the client, regardless of how recent a previous authentication may have occurred. Future implementations and expansions of this system may consider more complex methods of determining when a challenge-response handshake should occur, and include features such as caching of credentials.

If a critical point is requested, this operation is cached, and the server (RTU) demands a response from the client. This response includes a nonce generated by the server. The client receives this request, and generates its own nonce. These values are bitwise-concatenated with a pre-shared secret. A SHA-256 hash is generated on this bit concatenation, and the hash value, along with the client nonce are returned to the server. Upon reception, the server verifies the authenticity of the hash, and if the hash is valid, pulls the operation off the queue, and executes the operation as though it were a new request. If the hash is invalid, no response is sent, and the RTU re-enters a wait state.

### *3.5.3 Role Based Access Control*

The role based access control to be implemented on this RTU prototype is modeled after the access control system described in Hieb[35]. In order to facilitate prototyping, this system has been simplified from the system Hieb describes. These changes facilitated the rapid development of a system which can demonstrate the capability of a limited RTU to perform complex security operations using IPC of the sort found in the OKL4 operating system. The principal sacrifice is that of future expandability. Although the role based access control system described herein implements most of the features described by Hieb, they will not be expandable without some sacrifice in performance.

There are several object types in this role based access control system. "User" refers to the credentials passed to the RTU and "Points" represent the physical IO to be manipulated or read. Users maintain membership to one or more roles, and these roles have access to one or more sets of permissions. A permission is a set consisting of a point combined with point access controls. A point is a physical IO (or abstraction of physical IO) whose state may be altered, while a permission represents the actions that may be performed on the IO. In the target implementation, permissions may be read-only access or full-access.

Points have associated point types, which are labels used to more broadly control permissions. Roles have, in addition to associated permissions, point type controls, which govern whether or not a role may have access to a given point type regardless of permission. Roles further have point access constraints, which are further restrictions on how a role may access a given point. In the target implementation, only time-of-day and day-of-week point access constraints are considered, while Hieb describes further restrictions, such as terminal location, among others.

Users have a similar constraint in the form of role access constraints. Users may have restrictions placed on how they may use a particular role to which they are assigned. The goal targeted only temporal restrictions, while Hieb further expands on the possible implementations of such restrictions. Permissions in this manner are assigned in a logical OR fashion: If any of the set of permissions assigned to a user allow an action, this action is allowed to occur. If the action is not allowed to occur, the client is not notified of this failure. Only later reinspection will inform the user that their access permissions have been denied.

In order to simplify development, roles, users, points and all other controls relating to the RBAC feature are hard-coded into the system, and cannot be modified at run-time. Any access constraints which depend on information other than time of day have not been designed into this system. In order to check permissions and easily iterate through sets of permissions, users, and points, many access controls are stored as bitfields. This allows the use of native bitwise logic operations in order to determine positions. Since these permissions are permissive by default, their inspection is simple.

### 3.6 Design Summary

The security hardened prototype architecture makes use of several security features available with a microkernel like OKL4, which provides the ability the separate thread and memory spaces. Figure 3.1 provides a high level overview of this architecture, which will be implemented in the OKL4 operating system using thread and cells as units of separation between the zones defined in this figure. These cells will consist of a network IO cell, a security cell, and a physical IO cell. The physical IO cell will also include all utility functionality which is local to the RTU prototype, including debug outputs and hardware management. This is the bulk of the untrusted computing base. The security cell contains the entirety of the trusted computing base, with the exception of the operating system kernel. This kernel is abstracted from user-mode access, but for architectural purposes can be visualized as a separate cell running a separate thread. The remainder of the untrusted computing base lies in the network communication cell. Although this code is in the UTCB just as the physical IO, the network and physical IO reside in different memory spaces. The only method of transferring information and messages from physical IO to network IO (and vice versa) is through the security cell. This is where the verifiable security of the prototype RTU is defined. Chapter IV will continue to detail the implementation of the features described above.

# CHAPTER IV

# IMPLEMENTATION

This chapter describes the implementation of the design described in Chapter III. This will include an overview of the hardware selected, and the OKL4 development environment in Section 4.1, a description of the IO hardware and software in Section 4.2, and the necessary implementation of some utility functions in Section 4.3. Network hardware and software is discussed in Section 4.4, and finally, security software descriptions are located in Section 4.5. These subsections will briefly review the architectural choices from Chapter III, then discuss the actual implementation, along with any necessary architectural changes.

## 4.1 Hardware, Build System, And Workflow

The primary hardware platform for this project is the Gumstix Verdex Pro XM4 COM board. The Gumstix platform is a simple, embedded computer module which includes an Intel Xscale PXA microprocessor, 64 megabytes of RAM, and 16 megabytes of flash memory. The module also includes expansion connectors which are used general purpose IO (GPIO) and network connectivity. This module includes a PXA270 chip clocked at 400MHz. The PXA270 is a 32-bit microcontroller based on the ARMv5 architecture. Although dated (from 2007), the processor compares favorably to existing RTU devices. Datasheets and application manuals are commonly available from Marvell and Intel [37].

The Gumstix Platform was selected for its expandability and compatibility with available OKL4 Distributions, as well as its similarity to existing RTU hardware. The OKL4 microkernel is currently targeted towards the cell phone and mobile device market. Although the PXA270 was designed principally as a mobile device chip, its specifications are not dissimilar from RTUs already on the market. This chip is compatible with both Linux and the OKL4 Development Environment, allowing for testing in a real-world environment.

Along with the Gumstix, the console-VX and netpro-VX expansion boards are used. The console-VX provides pin headers for all three universal asynchronous receiver/transmitters (UART) on the PXA270 Chip, as well as pin-headers for Audio, inter-IC Bus (I2C), and several GPIOs. The netpro-VX includes an SMC9118 network PHY/MAC interface for network connectivity.

In order to boot the Gumstix board, a bootloader (Uboot) resides in flash memory. The remainder of flash is utilized by the JFFS2 file system. This file system is designed for flash storage, and in the gumstix environment, is pre-loaded with a Linux distribution. Uboot executes a startup script, which loads a program image from flash, microSD, or over the network into RAM, and then begins execution at that location. RAM begins at memory address 0xA2000000 and ends at address 0xA3F00000. In order to properly boot the Gumstix board, an image must be loaded into memory in either ARM ELF or Intel Hex format. The factory default bootloader script has been modified for this project to load an image over the network, via TFTP, load it into RAM, and execute.

The OKL4 operating system used for this prototype is version 3.0. The verified kernel which branches from the OKL4 project is based on version 3.0 of the operating system. The operating system is provided as a precompiled ELF file. The devleopment kit includes a C and C++ compiler chain based on GCC. This GCC compiler is modified to cross-compile for the ARM chip on the Gumstix board, and the SDK includes Make files which abstract the inclusion of OKL4 Libraries and ElfWeaver calls. GCC and its linker produce output ELF files individually for each cell in the source tree structure. Each cell has associated XML files which define characteristics such as default priority, heap and stack sizes, and capabilities. There are further XML files which describe the structure of a complete project, and still more XML files which define the hardware available on the development platform. These XML files are used after the compilation process by the ElfWeaver tool.

The operation of the compiler and linker are unremarkable. The OKL4 libraries included make use of basic kernel structures and system calls, and may be compiled into each cell independent of the kernel code. Each cell is compiled as its own individual program. Thus, the compilation tools and methods are independent of the OKL4 operating system. It is the ElfWeaver tool which manipulates these compiled ELF files in order to create a working OKL4 system. Elfweaver uses the XML files in order to determine the manner in which the resulting ELF files must be linked to the kernel. This linking takes place in an inside-out order. First, the compiled output files from GCC and its linker are combined with the per-cell XML file, relocating the memory space of the program, and making note of required cell characteristics such as capabilities, threads, memory sections, and other attributes.

Next, the cells are linked to the kernel, with the kernel getting assigned memory sections with low addresses, and cells coming in above that. Next, Elfweaver takes note of the earlier XML file specifications combined with a description of the machine architecture, and statically writes kernel memory to inform the kernel thread of the capabilities and permissions of each memory space, the location of hardware memory segments, as well as starting threads. This is how the final ELF file is generated.

The result of these operations is a single ARM ELF file which contains the operating system and user code. This is loaded onto the RTU prototype using the trivial file transfer protocol (TFTP) and the bootloader mentioned previously. The code then begins immediate execution. The "debug" kernel mode is used to provide additional information on the working state of the system. This includes a kernel debugger which can interrupt all running threads to provide state information. Threads may invoke this debugger, which provides a simple mechanism to generate breakpoints while the program is running on the prototype hardware.

The prototype RTU must be physically connected to a serial port for debug, and utilizes its network connection for loading compiled code. In order to facilitate development, a dedicated machine is used for this serial debug interface, as well as compiling the code. This machine also serves as a code repository, allowing multiple people to seamlessly contribute code to the project, and synchronizing development across several computers.

The architecture description describes the logical cell breakdown. The development tree is laid out in a similar fashion: Each cell resides in its own directory, with its own Make script and XML ElfWeaver description. This XML tree is shown in figure 4.1. This make script is capable of compiling the files within the cell independent of other cells. Above all these cell directories lies a directory which contains a Make script and a project description XML file. This Make script recursively calls the Make scripts for each cell, then assembled the appropriate ElfWeaver calls, resulting in a single output ELF file.

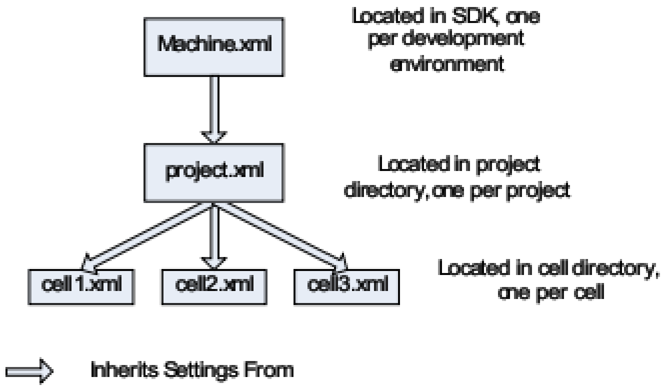
Machine.xml
Located in SDK, one per development environment
project.xml
Located in project directory, one per project
cell1.xml
cell2.xml
cell3.xml
Located in cell directory, one per cell
Inherits Settings From

*Figure 4.1: Hierarchy of ElfWeaver Compilation*

As an example, the cell XML file for the network IO cell is as follows:

```
<okl4 priority="255" clists="256" file="access"
kernel_heap="0x400000" mutexes="256" name="access" spaces="64">
      <use_device name="serial_dev"/>
      <use_device name="timer_dev"/>
      <use_device name="cs_dev"/>
      <environment>
            <entry cap="/security/main" key="SECURITY_CAP"/>
      </environment>
      <heap size="0x100000"/>
      <commandline/>
</okl4>
```

The first line declares configuration set-points for the OKL4 operating system, such as thread priority, heap size, and arbitrary names by which the cell will be referred in the Elfweaver utility, and which must correspond to the output ELF file generated by the C compiler. Under this declaration exist other declarations. Among these, device tags tell the Elfweaver utility to search the machine.xml file for the memory space and interrupts which correspond to a given device name, and assign them to virtual memory with the cell.  Envrionment entrys with the keyword “cap” indicate capabilities to communicate with another cell.  These capabilities are given a location within the build tree (in this example, the “main” thread in the “security” cell), and a key to which these capabilities can be referred by the running thread.  This is a small example of the sort of declarations which can be made within cell-specific XML file.

In order to access specific hardware devices, the running thread must have an awareness of how the device-specific memory registers get mapped into the thread's virtual memory.  As code changes, these virtual memory locations may be moved around, although pages are always mapped in a one-to-one manner.  Although the start location of a given page of memory may change, locations within that page will retain the same offset from the start of the virtualized page.  This code, from the physical IO cell, shows how this mapping takes place.

```
okl4_memsec_t           * i2c_memsec;
okl4_env_segment_t      * i2c_seg;
okl4_static_memsec_t    * i2c_static_memsec;
okl4_static_memsec_attr_t i2c_static_memsec_attr;
okl4_virtmem_item_t     i2c_virtmem;

okl4_init_thread(); //this sets up okl4 lib, call for each thread
//   may not need this due to weaver call
L4_KDB_SetThreadName(L4_Myself(), "I2C");

i2c_seg=okl4_env_get_segment("MAIN_I2C_MEM0");
assert(i2c_seg);

okl4_static_memsec_attr_init(&i2c_static_memsec_attr);

okl4_static_memsec_attr_setsegment(&i2c_static_memsec_attr,i2c_se
g);


i2c_static_memsec=malloc(OKL4_STATIC_MEMSEC_SIZE_ATTR(&i2c_static
_memsec_attr));
assert(i2c_static_memsec);


okl4_static_memsec_init(i2c_static_memsec,&i2c_static_memsec_attr
);

i2c_memsec=okl4_static_memsec_getmemsec(i2c_static_memsec);

i2c_virtmem=okl4_memsec_getrange(i2c_memsec);


//Need these offsets, since OKL4 Page-aligns everything to the
nearest
//Smallest page size (which is 0x1000).
IBMR = okl4_range_item_getbase(&i2c_virtmem)+ (okl4_word_t)0x680;
IDBR = okl4_range_item_getbase(&i2c_virtmem) +
(okl4_word_t)0x688;
ICR = okl4_range_item_getbase(&i2c_virtmem) + (okl4_word_t)0x690;
ISR = okl4_range_item_getbase(&i2c_virtmem) + (okl4_word_t)0x698;
ISAR = okl4_range_item_getbase(&i2c_virtmem) +
(okl4_word_t)0x6A0;
```

In the last block of code, virtual memory is assigned to global variables which represent device memory registers. These are declared by finding a base memory location from the given virtual memory object, and applying an offset which is determined by the specific device, and can be found in the data sheet for the processor. This process is repeated for each hardware device which a cell must implement.

### 4.2 IO Hardware And Software

While the Gumstix is a powerful and versatile computing platform, it is not suitable for directly generating the sort of digital and analog output typical of SCADA systems. For this, the project requires additional hardware and interfacing software. The existing digital IO utilizes 3.3v logic levels with no isolation. With the exception of a sound driver, the Gumstix has no analog operating capability. A survey of existing commercial RTUs shows a number of isolated nine volt digital IO, as well as analog in and analog out. In order to safely generate this output, additional hardware is necessary.

In order to provide flexibility in the selection of components, as well as reduce possible avenues of catastrophic failure, an I2C-based IO system was devised. The I2C bus is an Inter-IC communication bus, with many thousands of available peripherals in many thousands of configurations. The bus has a maximum speed of 400KHz, and can support up to 127 slave devices. Using an I2C based input/output system for the project had multiple advantages. Changing the specifications of the IO no longer required retooling of complex analog components, and the PXA270. Rather, a cheap commodity IC could be replaced and some minor code changes made. Furthermore, GPIO were conserved, and the risk of destroying the PXA270 is greatly reduced as a result of placing the I2C chips between the PXA and the field components.

This also allows for the simple isolation of IO software. All built-in GPIO, including the IRQ lines, and GPIO function registers, exist in the same memory space. As a result, only a single OKL4 cell may have access to these functions. This is a disadvantage, as many hardware devices require interaction with these registers. The net result would be requiring the IO code to exist in the same execution space as the utility/hardware manager. The I2C memory space, however, exists as its own page. As a result, the architecture of the prototype as-written can more closely match the target architecture in meeting segmentation goals, allowing for a more verifiably secure device.

All development and hardware build-out has been performed on a breadboard. Initial development of the I2C hardware was tested with an Arduino Microprocessing Environment. The Arduino is a simple, easy-to-use microcontroller development kit, with a simplified IDE. Although this device primarily targets the hobbyist market, it was useful in this case to get development started quickly. Without a complex understanding of the OKL4 operating system, and the PXA270 chip, the IO breakout board could be completed and tested. The code generated to test the devices on this breadboard greatly sped up the process of transferring the IO code to the PXA platform.

The project as-built has eight analog inputs, two analog outputs, and eight digital IO spread across three chips. Without modification, all of these chips operate at the 3.3 volts provided by the PXA270 and the Console-VX breakout board. The Serial Data and Serial Clock lines of the I2C bus must be pulled to a high logic level. Nominally, 4.7 kOhm resistors are used. Steps must also be taken to minimize bus capacitance. With another layer of signal conditioning, these I/O can be easily converted into the 0-9v/0-20mA signals needed for interaction with control devices. The chips were selected for their speed, accuracy, and availability. The digital IO chip used the Microchip MCP23009. This chip provides 8 GPIO at up to 400KHz. The pinout of this chip is shown in figure 4.2

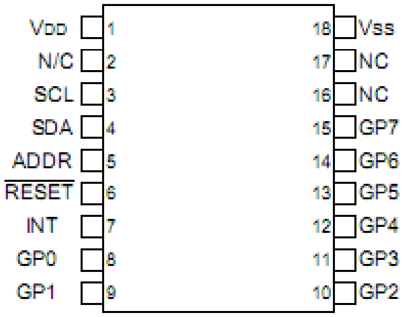

*Figure 4.2: Pinout of theMCP23009 Digital IO IC.*

The analog output chips are both Microchip MCP4725.  Two are included, each  provides a single 12-bit DAC. The output is provided by a high-speed, high-accuracy resistor ladder.  The pinout of this chip is shown in figure 4.3.

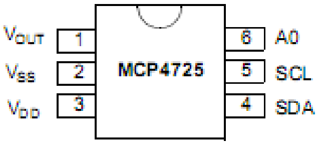

*Figure 4.3: Pinout of the MCP4725 Analog Output IC.*

The analog input chip  is the Analog Devices AD7997. This chip provides eight 10-bit analog inputs.  The pinout of this component is shown in figure 4.4.

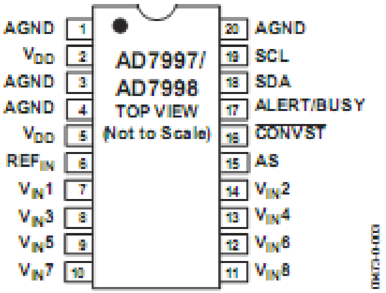

*Figure 4.4: Pinout of the AD7997 Analog Input IC.*

These components are mounted on a breadboard, along with appropriate external passive components (such as decoupling capacitors and pull up resistors).  This breadboard is connected to the console-VX breakout board, which includes standard pin headers for the I2C bus, as well as serial outputs.  The circuit diagram of all IO hardware is shown in figure 4.5.

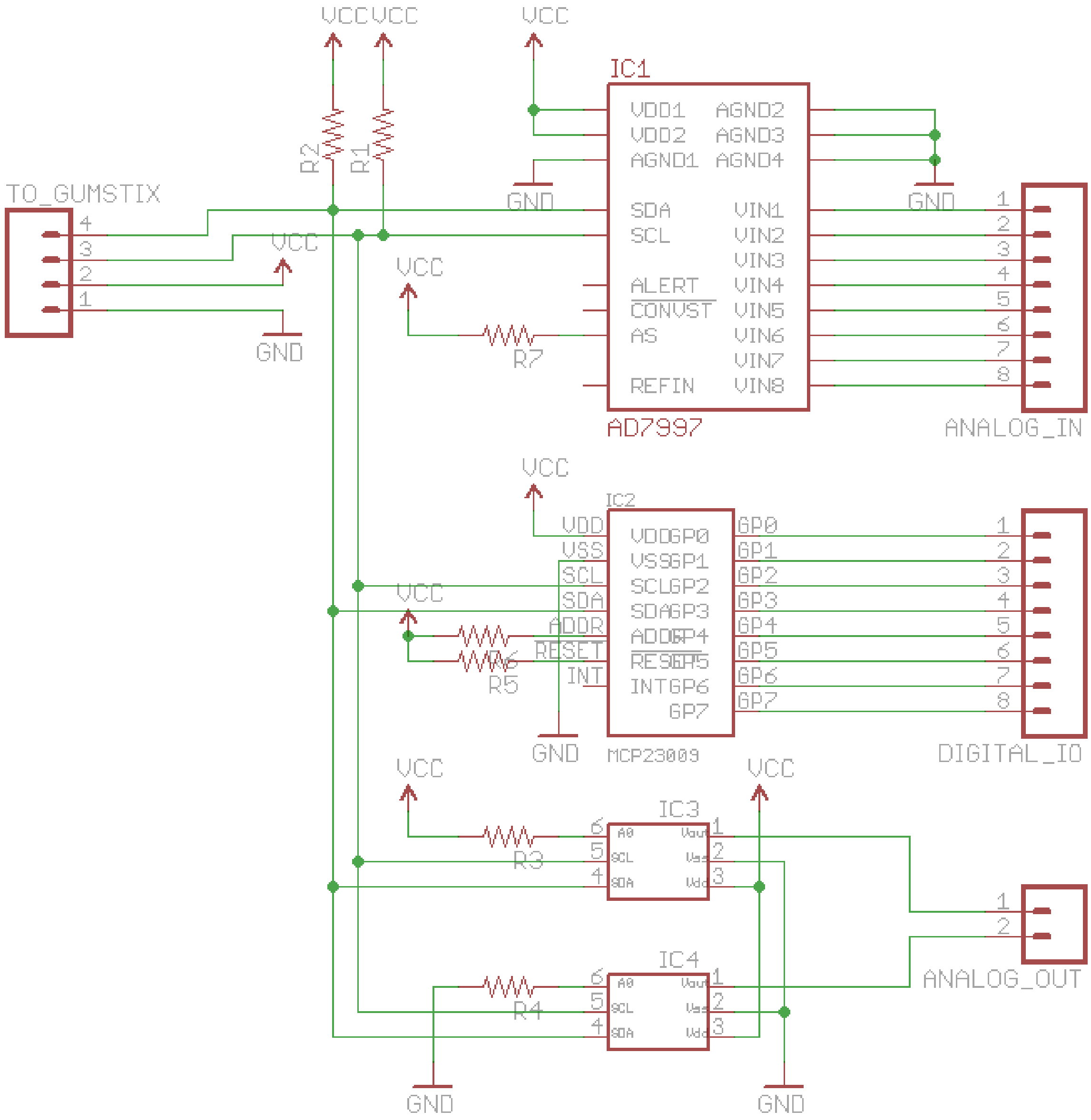

*Figure 4.5: Circuit Diagram for IO Hardware*

As mentioned earlier, the bit-level interfacing for the IO hardware was initially developed using a simple 8-bit microcontroller platform. Transferring this to the Gumstix required driver code for the I2C hardware of the Gumstix. This peripheral device has two main operating modes: buffered and unbuffered. While buffered mode provides greater reliability in an environment with many sensors and devices operating in a highly threaded environment, unbuffered mode provides more direct access to the hardware. In this mode, a single byte is loaded into a transmit register, the appropriate configuration bits are set, and the byte is shifted onto the communication media. The configuration register is polled to determine the transmission status of the data. In the case of all the hardware described above, no data transmission requires more than three bytes of data, to either receive or transmit a given piece of data.

The I2C bus operates a clock speed for 400khz, with a single acknowledge bit to each byte of data, and a single address byte for each transmission of data. This results in a maximal channel bandwidth of 44 kilobits per second. For the purpose of the prototype, this is an acceptable method for generating IO.

The driving software for the I2C peripheral controller was simply generated. The major hurdle through this part of development was the discovery of clock distribution and power saving hardware within the Gumstix processor. This hardware shut down non-critical systems by default in order to save power. This was only discovered after sifting through the Linux driver for the I2C peripheral on the Gumstix board. The functionality required to initialize the I2C peripheral was added to the utility functions described in Section 4.3.

### 4.3 Utility Functionality

For the purposes of development, utility functionality includes any functionality which is not specifically related to the task of accepting control input from a communication device (in this case, Ethernet), and generating output through a security layer. The OKL4 microkernel takes care of the large part of this debug functionality.

Unfortunately, some additional features are required specific to this hardware and target architecture which necessitate additional utility threads.  During development, these threads consist of a hardware manager, and a serial debug thread.

### *4.3.1 Hardware Management*

During the architectural planning phase, it was discovered that the memory layout of the Gumstix processor would require a hardware manager in order to facilitate the configuration of  various peripherals on the gumstix board, including the I2C hardware, the network hardware, timer functionality, and serial ports.  This could not be done by the individual cells due to the complexities of mapping memory into each cell on a read/write basis.  Such a mapping is not only difficult to implement under OKL4, but it undermines the principals of separation which allow such a device to remain secure.  The mechanisms of IPC within the OKL4 operating system allow us operate this hardware from a separate thread and memory space.  The original design placed the hardware management thread within its own cell, for even stronger segmentation.  Unfortunately, bugs in the build system rendered this solution untenable.  The total number of cells in the operating environment is limited by errors in the memory allocation subsystem of the Elfweaver program.  As a result, the hardware management functionality is placed in the untrusted computing base, in the same memory space as the physical IO.  This is not a critical architectural change, and does not affect the security or performance features of the RTU. The IPC calls required are abstracted by the libraries.  The function call do not behave differently based on the locality of the memory space to which the communication is destined.

Thus, the operation and implementation of the hardware management cell is simple. The hardware manager alters some configuration registers according to the needs of the remote cell. This alteration is statically defined– the cell which requires this notification cannot request specific changes in system state. The remote threads simply wait until they are given a message. This message is sent after all hardware has been initialized. In reality, these messages are not sent concurrently, and if they are not sent in a specifically calculated order, race conditions could result. For prototyping purposes, these race conditions are avoided by inserting calculated execution delays prior to thread operation. This allows for simple insertion and removal of test frameworks.

### *4.3.2. Serial Debug*

The need for serial debug was immediately apparent after some initial tests and research into the operation of the OKL4 microkernel. The existing kernel debugger operated alongside the `stdout` structure in the development environment. This required the assignment of a serial device to a specific and single cell. Although this is useful for debugging a single thread, in many instances the interaction of threads and the communications between them was the interesting subject of debugging. In order to generate usefully verbose output from more than one cell, messages would be passed to a central thread which would implement and use the serial device. IPC calls will pass the desired ASCII data to the serial debug cell, which will be printed to the terminal, along with the originating cell name.

The implementation of such functionality is simple. ASCII message must fit within a single IPC call, and are thus limited to approximately 160 bytes. The client thread must first call an initialization function which declares the cell's name in a human readable format. A new function, `lprintf`, is written which implements `vnsprintf` to generate a string output based on a variable argument input. This output byte string is then passed to the serial debug cell. The serial debug cell receives this communication, and displays the message along with the previously declared thread name.

From the client cell, the calls are blocking. As a result, output to the debug cell is guaranteed. If the debug cell is in the process of printing data from another thread, the second thread simply waits until the debug cell is able to process the request. While this results in decreased performance for cells printing data, this is not critical. An operational device would not include such debug functionality.

### 4.4 Network IO

Network IO is a critical component of the RTU, providing greater connectivity which allows for the demonstration of the security and performance capabilities described herein. The Gumstix board includes a network device based on the SMC9118 chipset. This chipset provides access to the physical and MAC network layers. All other layers of communication must be implemented in software. This network device is designed for high throughput performance in a monolithic kernel operating system such as real-time Linux. As a result, the hardware includes faculties for deep packet buffers and interrupt based functionality. For the development of the prototype, these high performance features are toilsome rather than helpful. Unfortunately, the existing device drivers implement the complete feature set of capability included for this hardware.

In order to boost progress, the least comprehensive drivers available were used in the development of the RTU software. These drivers are derived from a dated version of the Uboot bootloader. This program is normally used as a minimal bootloader for embedded systems. In fact, the Gumstix itself loads boot code from a newer version of this software package. This device driver is neatly stratified into layers which support raw Ethernet frames, and higher level program structures which implement TCP/IP, UDP and other standard protocols. This is beneficial to development of the prototype RTU, as we have principal need of a driver stack which implements hardware drivers to send and receive Ethernet frames.

Although the uboot drivers operate in privileged mode in their native environment, the page structure of the physical memory associate with the network device allows seamless virtualization of the physical memory with minimal modification. This memory becomes an offset subsection inside the thread's virtual memory space. Other changes are more endemic to the structure of the driver. The uboot code operates in an interrupt driven mode, and maintains a sense of statefulness between packets. In order to minimize points of failure while modifying the driver, both of these features are undesirable. It is necessary to remove the interrupt drive structures, and circumvent the packet buffers built into the driver code. This results in no net performance detriment when implementing the simplified SCADA protocol described in Section 3.4, as the data throughput during testing is not large enough to necessitate such buffers.

In order to change from interrupt driven mode to polled mode, each piece of code which would block for an impending interrupt would instead inspect configuration registers in an endless loop. Throughout these changes, it is necessary to remain vigilant of memory which must be declared volatile. In order to circumvent the packet buffers, the packet reception code, which would rotate through these memory locations in a round-robin fashion instead simply deposits all data in the first buffer. Data must be read from this buffer as soon as it is made available. This is not appropriate for high-throughput links, but this has no effect on the performance of the prototype RTU.

Following these modifications, it was possible to send and receive raw unicast Ethernet frames. In order to interface with more complex systems, some basic network layers must be implemented at a higher level. In order to establish socket connections with Windows and Linux machines, it is necessary to implement address resolution protocol (ARP), a standard protocol which allows IP addresses to be associate with MAC addresses in the link-local context. In order to deliver data in a more standards compliant manner, UDP is preferred due to the simplicity of implementation.

The micro-IP project is a comprehensive source of networking code designed for small real-time systems. The basic micro-IP stack includes an implementation of ARP, Ping and UDP. With careful efforts, this stack was modified to operate within the OKL4 operating system, utilizing the send and receive primitives of the polled-mode network driver. Unfortunately, the network driver as-implemented does not support addressing by means of multicast MAC addresses. In order to reduce network traffic on large local area networks, many implementations of ARP will only request responses via a specified local multicast address. As a result, the prototype RTU cannot respond to ARP request of this nature. In order to circumvent this issue, the prototype generates gratuitous, unrequested ARP responses. The responses are discovered by the link-local network devices, and retained for a short period of time (5-15 minutes). The UDP implementation does not pay attention to checksum or header information, excepting that all data must fit within a single Ethernet frame (including IP overhead).

Once a packet is received by the prototype RTU, it is inspected for utility. If it is an ARP or a Ping packet, it is immediately acted upon, and the network IO thread reenters a packet wait state. If the packet is UDP, it must be addressed to the correct port (1200, chosen arbitrarily, from Section 3.4). User code then inspect the packet for the proper formatting, according to the simple SCADA protocol specification in Section 3.4. If the request is properly formatted, the packet is passed up to security code as IPC. Security code processes this request, and informs the network code of the proper response. If the proper response if no response, the network code again enters a wait state. If the proper response is a packet, the packet is generated and transmitted. The UDP layer generates the UDP and IP header, as well as any necessary checksums.

### 4.5 Security Cell

The Security Cell is the only portion of code which may transmit information between the physical IO and the network IO. This layer must ensure that the network IO can be trusted, and is authorized to perform as commanded. This layer is the only portion of code which must be trusted to some degree. All other code (with the exception of the kernel) may be untrusted. Furthermore, operations performed by SCADA systems are time sensitive. While the project need not conform to strictly deterministic timing requirements, the performance must not be unsuitable for any SCADA application. The security cell must therefore be designed and written with an eye towards transparency, verifiability, and speed. In further pursuit of these goals, the security cell maintains no permissions to any hardware faculties, apart from RAM. In all cases, less code is better than more code.

Aside from passing message between the physical and network IO, the security cell has two main functions: performing the necessary network authentication required to trust the client messages, and perform role based access control to ensure that the remote user has permission to perform the operations desired. While these two functionalities are related, they are functionally independent. As implemented, all functions are highly layered, with data passing between layers, allowing additional functionality, or alteration of functionality at every level.

Data enters the security cell through the network IO cell. The network IO cell delivers the simple SCADA contents of the incoming packet to the security cell in an IPC message. If the security cell  is currently preoccupied processing another request, the incoming request is denied. No response is sent to the remote machine. This minimizes the amount of time spent in a state which cannot receive packets. This is a deliberate decision to mitigate the risk of denial of service attacks. The security cell firsts passes this message into a layer which determines whether it is necessary to demand an authentication handshake with the client. Currently, necessity is determined based on a set list of “critical points.” If any user attempts to perform any operation on a critical point, handshaking authentication is demanded. This logic is easily expandable to include other system states. If this handshaking is necessary, the operation is placed on a one-deep queue, and the process of authenticating the client is performed. If this handshaking is performed successfully, the operation is passed up to the role based authentication layer. If handshaking is unsuccessful, the operation is not passed up.

Next, if authentication is successful, or no authentication is necessary, the packet is passed up to the role based authentication layer. This consists of a single monolithic function which compares the desired operation and associate user, combined with current system state, with the role based authentication policy. If this operation is allowed by role based authentication controls, then it is passed to the physical IO cell. If the operation necessitates a response, the security cell waits for the response, and passes it back to the network IO cell. This is the entire operation  of the security cell.

If handshaking is necessary, the operation is placed on a one-deep queue. The security cell then generates a server nonce value. This nonce value is transmitted to the network IO cell. The network IO cell formats this into a demand-response packet, and transmits to the client. The client must generate its own nonce, and perform a SHA-256 hash of the server nonce, the client nonce, and a pre-shared secret. The client nonce and the first 4 bytes of the calculated SHA-256 hash are transmitted back to the server. Upon reception, the network IO cell transmits this information to the security cell. The security cell then generates its own SHA-256 hash based on server nonce, client nonce, and pre-shared secret. If the first 4 bytes are identical the authentication is successful and the queued packet is passed to the role based authentication layer. If the authentication is not successful, the queued operation is simply ignored. No response is sent to the client regarding the success or failure of the operation.

The next step taken by the prospective operation is then subject to the role based access control layer. The role based access controls are programmed statically (hard coded) for simplicity and speed. The functions which make use of these controls are agnostic to the source of access control data. Expanding on the descriptions from section 3.7, points represent physical IO or system state. Points have a unique ID, and an associated point type. This point type is an arbitrary descriptor with no programmatic meaning. Permissions are a combination of a point and an operation. Operations are coded as bit fields in order to allow easy inspection by logic operations. (For example, read is 0x01, operate is 0x02 and select is 0x04. A logical AND operation can easily mask out desired operations.)

Roles are buit of permissions, combined with point type controls, and permission access constraints, along with a unique role ID. Point type controls exist as unique bitmasks, similar to operations within permissions. These point types indicated which points may be manipulated, regardless of permissions. For example, a role might have permission to read and operate on point three, but point three is of point type two, and this role does not have access to point type two. As a result, no operations are allowed, despite permissions. This allows for larger redefinition of points without altering individual permissions. Permission access constraints are descriptors which consider system state when allowing a role access to its permissions. Although Hieb's description of an RTU based RBAC configuration includes consideration for many types of permission access constraints, the prototype RTU only considers constraints based on time of day and day of week. Hieb also adds constraints based on location within factory, location of terminal, and the data payload of the operation.

Users are the highest level construct in this hierarchy. Users contain information about roles, as well as unique pre-shared secrets (used in the authentication step described above), and permission access constraints. Role access constraints are logically identical to the permission access constraints described above, but instead limit a users ability to access roles. Role ID numbers are assigned in a bitmask fashion, as described above. As a result, a user may have access to multiple roles. If any single role gives a user access to a point, the user is allowed to perform the permitted operation on the given point, even if another role would result in failed permissions.

In order to determine access for a given incoming operation, access controls are inspected in order of granularity. First, the incoming user is inspected for membership to any role. If the user has an associated role, this role is inspected for access to the given point. The role is then inspected for point type access. If the role has access to the given point type, its associated permission is inspected for access to the desired operation. If the permission has access to the desired operation, role access constraints are then inspected. If no role access constraint prevents access, permission access constraints are inspected. If no permission access constrains prevent access, the operation is allowed to proceed. Failures at any level in this hierarchy prevent continuation into higher levels of inspection. As a result, failed access is the fastest possible operation. This prevents bogus requests from occupying too much processing time in the security cell.

If permission is granted, the data is then passed into the physical IO cell for further processing. If permission is not granted, the security cell and network cell re-enter a wait state, and no communication is sent to the client.

# CHAPTER V

# PERFORMANCE ANALYSIS

After implementing the basic role based access controls, testing was carried out in order to determine the performance overhead of the implementation of such an architecture. This testing provides a baseline level of performance which could be improved and optimized. The intent was to develop an understanding of the performance of a device which security enhancements based on the isolation of threads and memory spaces. Section 5.1 will provide and overview of the test goals and constraints, Section 5.2 will describe the test methodologies, Section 5.3 will review initial performance tests, and Section 5.4 will detail the performance testing of the entire software configuration.

## 5.1 Test Goals And Constraints

This performance testing was designed to provide an overview of the overhead that should be expected when implementing the security architecture as built. It was not intended to certify or verify the prototype or the architecture's suitability for a specific control system application.

In general, it is desirable to spend a minimum amount of time performing security operations. It is not so critical for timing to remain consistent across points. It is also not critical for all points to remain constant in response time. Initial performance measurements were taken between the network IO cell and the physical IO cell, with a dummy cell in the middle, playing the role of the security cell. This provides a baseline level of performance which will allow for insight into the specific time overhead of the security operations. In this initial testing, the dummy “security” cell merely passes the messages on to the physical IO cell, without any modification or inspection. The time overhead required for this operation consists of the time required to inspect the incoming packet, pass the data to the security cell via IPC, combined with the time required to pass this IPC through the security cell, to physical IO, and for the physical IO to return via the same path. Testing for this phase measured performance based on digital operation, an analog read, or and analog write, due to the differing amounts of data, and different software drivers required to manipulate each of these physical IO components. The SCADA protocol described in Chapter IV was further simplified in order to eliminate as many variables as possible. Messages were injected into raw Ethernet packets, without the overhead of a UDP protocol stack. For testing, the RTU did not perform verification or inspection of packet contents.

Further testing probed the performance penalties imposed by the SCADA protocol processing and security features. These measurements focused primarily on the end-to-end performance of the device, without consideration for outside factors such as client code, or network conditions. The important timing data is the time between the reception of a request and the response sent. Testing for this phase will cover a broad region of possibilities, in order to explore possible bottlenecks. Considerations were made for the acquisition of time data for the challenge-handshake authentication, as well as the role based access control. Due to the nature of the RBAC algorithm used for the prototype RTU, different points respond in differing amounts of time depending on which user and which role is accessing the given point. The timing data for failed requests is also considered.

### 5.2 Test Methodologies

Performance testing as described above takes place within the software of the RTU. In order to generate this timing data, a stable time base which remains consistent across cells will be required. The Gumstix processor has multiple counter peripherals. The OKL4 operating system makes use of one of these timing peripherals in order to handle operating system tasks such as context switches and IPC. The other timer peripheral is unused. In order to track timing across multiple cells and in multiple situations, the memory space assigned to this timer peripheral must be mapped into multiple cells simultaneously. This is impossible in to do if both cells must have read-write access to the memory segment in question. Fortunately, it is possible to map the same memory segment into a single cell in a read-only fashion. In order to make use of the timer, it is only necessary to read the current state of the timer. Although other timer features are useful, for the purposes of generating performance results, this will suffice.

The remaining timer is configured by the hardware manager as a 3.25 MHz free-running 32-bit counter. The counter timebase is generated from the main CPU clock, which is driven by a 13MHz crystal oscillator, which is fed into a PLL circuit which multiplies the input frequency to a CPU clock of 416 MHz. The total error in this system is +/- 50PPM, amounting to +/- 50 microseconds over a one second measurement. A 3.25 MHz free-running counter which increments a 32-bit register results in a maximum timebase of approximately 22 minutes without overflowing. There is a slight chance of overflow when simply observing the before-and-after state of the counter. As the timebase of measurement is very small compared to the full range of measurements, overrun data will appear be calculated as negative. These negative measurements will simply be discarded if they appear.

A single inline function was written to make note of the timer value at the beginning and end of the testing period. This inline function simply writes the value of the timer to a local variable. After the timer is read, the amount of time can be calculated without worry of altering the results. Creating this function as a C Code inline insures that it is compiled without requiring the use of the stack pointer, reducing the required CPU overhead. In the ARM architecture, this function compiles to three assembly instructions, requiring 12 cycles of execution at 400 MHz. This is below the minimum time increment detectable on the prototype RTU.

During the testing process, all serial output was disabled. The design of the serial output in both the kernel debugger, as well as the included serial debug cell require bits to be loaded sequentially into a shift register. As a result, generating serial IO requires a large number of CPU cycles, because the speed of serial output is a mere 19200 bits per second. Additionally, it is necessary for any non-running tasks to cede processing time as early as possible. For example, the hardware management cell should cease execution immediately following the configuration of peripherals. Failure to cede unused CPU time to time-critical processes could lead to as much as a third of the CPU time doing absolutely nothing. In implementation, C preprocessor defines are used to enable and disable these features at compile time, as this code is necessary for debugging.

Initial performance test measured the performance from the reception of an incoming packet to the response packet. This considers the time spent reading the packet, sending the message through two IPC links, generating the appropriate IO, and responding. This provided a baseline level of performance for I2C IO operations, as well as IPC messages between cells.

Additional performance data was measured from the reception of a packet to the response packet. This considers the decoding of the UDP payload, IPC between cells, security operations, and generating IO. This was performed using a set of test users. These users have access to between zero and two simultaneous roles, and have different associated role and permission access constraints. Performance of the handshaking algorithm was measured as well. For handshaking, only the time spend processing the request on the RTU was considered. The time required for the client to generate an appropriate response is not considered as part of a performance test for the prototype RTU.

### 5.3 Initial Performance Measurements

Hieb showed that performance could be as good as 65 microseconds for an IPC request.[35] This high level of IPC performance on a similar platform implies that it should not be impossible to implement an RTU using verified IPC calls of such speed. Hieb's example, however, was reduced in scope and complexity compared to what would be required for a complete RTU. In order to properly simulate the calls necessary to implement a functional RTU prototype, IPC is routed through multiple cells, and used to generate a physical IO response before responding to the original cell.

For the purposes of initial performance measurement, IPC was routed from the network IO cell, through a dummy cell, into the physical IO cell. During this test, the performance of security algorithms was not considered. This test displayed the performance of the OKL4 operating system in an architecture similar to that defined in Chapter 3, establishing a baseline of performance on top of which security could be added.

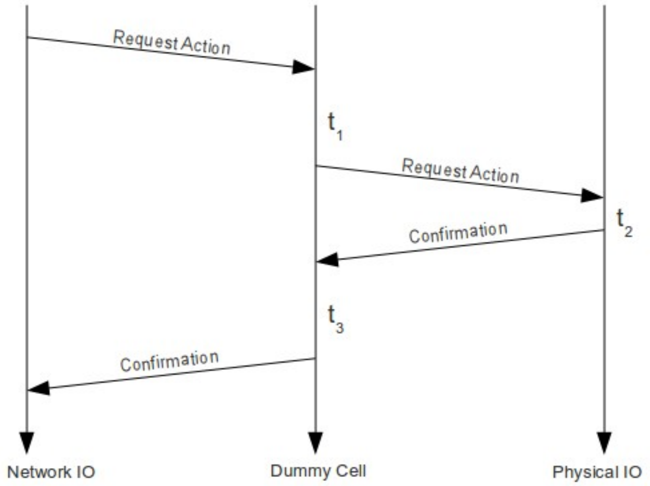

*Figure 5.1: IPC Communications Testing Paths*

The IPC message passing is illustrated in Figure 5.1.

Initial results were several orders of magnitude different from the results discovered by Hieb in 2008. The time required by the IPC routines (represented in figure 5.1 as $t_1$ and $t_3$ ) consists of 95-98 percent of the total time required for the operation. This was suspicious, as the complexity of the software tested was not significantly greater than that tested by Hieb in 2008 [35]. Investigation provided a great deal of insight into the operation of the OKL4 operating system, and revealed that these performance measures were invalid. An errant thread (operating the hardware manager, in this case) failed to cede its CPU time after its critical operations had been completed, resulting in large delays while the hardware manager utilized the CPU by idling. Fixing this issue required major changes in the architecture of the utility functionality. These changes are reflected by the results in Table 5.1

*Table 5.1: IPC Performance measurements*

| | Metric | **Analog Read Performance** | | **Analog Write Performance** | | **Digital Read-Write Performance** | |
|---|---|---|---|---|---|---|---|
| | | Value | Units | Value | Units | Value | Units |
| I2C + IPC | Mean Time | 533.65 | µs | 334.39 | µs | 441.66 | µs |
| | St. Deviation | 3.12 | µs | 2.41 | µs | 3.03 | µs |
| | N | 999 | | 999 | | 999 | |
| | Max | 553.85 | µs | 344.92 | µs | 458.77 | µs |
| | Min | 528.92 | µs | 329.75 | µs | 436.62 | µs |
| | | | | | | | |
| I2C | Mean Time | 503.46 | µs | 304.24 | µs | 411.39 | µs |
| | St. Deviation | 2.88 | µs | 2.32 | µs | 2.58 | µs |
| | N | 999 | | 999 | | 999 | |
| | Max | 520.92 | µs | 314.15 | µs | 421.85 | µs |
| | Min | 499.69 | µs | 300.31 | µs | 407.38 | µs |
| | | | | | | | |
| | IPC Overhead | 5.66% | | 9.02% | | 6.86% | |

With the removal of the errant thread, which utilized a large number of CPU cycles, the IPC performance was vastly improved, with round-trip results on the order of 30 µs. Considering the time required to generate the physical IO, IPC overhead accounts for a mere five to ten percent of total required overhead. This performance is well within the time performance requirements of many SCADA system applications.

### 5.4 Security Feature Performance Measurements

Following the initial performance measurements, which show the capabilities of the OKL4 operating system in the context of the target architecture, the security layer was developed and inserter between the network IO and the physical IO. Although the security layer is designed to minimize performance penalties, the many checks and parsing layers imposed by the system as developed in chapter IV imparts a delay in the passing of messages. It should be noted that the performance results described herein do not include any amount of optimization beyond what is described in chapter 4. In order to gain adequate insight into the operation of this cell, several outside cases were tested.

The testing is performed in the same manner as described in figure 5.1, with the dummy cell replaced by the actual security code. The communications which take place between the network IO, security, and physical IO processes is described more fully in Chapter III. The performance measurement is taken from the moment of packet receipt to the moment of reply packet transmission. If no packet is to be transmitted to the remote host, the end point of time measurement is taken as the moment the network IO cell re-enters a wait state, awaiting the reception of new packets. This performance measurement considers the overhead of parsing and generating UDP packets, disassembling the data payload and forwarding the message to the security layer, performing any necessary security operations, generating the physical IO and/or generating a response. This method of testing deconvolutes the potential network delays from the performance of the RTU hardware.

Four test cases are used in this test measurement. User one is used for the test cases involving operations which are permitted to occur. Due to the ordering of roles and users within the security data structures, user one must undergo the maximal possible number of inspections and iterations in order to generate a “permission granted” response. For tests involving operations which are not permitted, user five is utilized. User fivve has no permissions to any point or operation, and due to its placement in the security data structures, undergoes the maximal possible number of inspections and iterations for an operation which is not permitted.

The test cases used are user one reading and writing to point one, user one reading point two, and user five reading point one. Point one does not require handshaking authentication, and is placed as an outside case among point which do not require handshaking authentication, while point two is an outside case of points which do require handshaking authentication. In order to properly test the time required to generate and verify a challenge-handshake demand and response conversation, the time required from the original request to the authentication demand is summed with the time required from the reception of the response to the transmission of the operation response. The removes from consideration the amount of time required for the client to receive a challenge request and reply appropriately.

*Table 5.2: Performance Summary of Security Operations*

| | User One, Point One, Read | Units | User One, Point One, Write | Units | User One, Point Two, Read | Units | User Five, Point One, Read | Units |
|---|---|---|---|---|---|---|---|---|
| Average | 44.98 | ms | 44.99 | ms | 67.34 | ms | 22.36 | ms |
| St. Deviation | 0.35 | ms | 0.33 | ms | 0.33 | ms | 0.28 | ms |
| N | 1000 | | 1000 | | 1000 | | 1000 | |
| Min | 45.22 | ms | 45.23 | ms | 67.852 | ms | 22.54 | ms |
| Max | 40.49 | ms | 43.09 | ms | 66.69 | ms | 21.61 | ms |

Pursuant to design goals, user five experiences a minimal delay from request to response, corresponding with the very fast denial of permission associated with the design of the security cell. Predictably, access to point two, which requires handshaking authentication, requires more time than access to point one. As points one and two take approximately the same number of inspections and iterations to work through the role based access control system, these results show that the handshaking algorithm imposes an additional delay of approximately 23 milliseconds. The representative numbers summarized in table 5.3 are acceptable for a wide variety of SCADA applications.

# CHAPTER VI

# CONCLUSIONS AND FUTURE RESEARCH

This thesis presented an overview of the design, implementation, and testing of a prototype based on a new security architecture for remote terminal units attached to SCADA systems. Using the OKL4 microkernel operating system, it is now possible to implement this architecture in hardware, and begin to perform security and performance testing on this prototype device. The preceding chapters have detailed this process. Section 6.1 will summarize the procedures and findings in this prototype development, while Section 6.2 will briefly discuss possible future directions to this line of development.

## 6.1 Summary Of Results

The architecture developed by Hieb and Graham [33] is uniquely applicable to RTU control devices and similar industrial embedded systems. This architecture separates the physical IO from the communications layers with a security piece which provides trusted isolation. Monolithic kernels cannot provide this trusted isolation, due to their architecture. The architecture of the microkernel lends itself to this development very well by providing a minimal basis upon which a secure system can be built. The security pieces of the target architecture aim for minimal interaction with untrusted pieces, which can be provided through a microkernel. This minimization of the security components simplify the procedure of formal verification, which requires an operating system which can be objectively trusted.

Fortunately, the OKL4 operating system has reportedly undergone the rigors of formal verification, providing an ideal microkernel platform upon which such a system can be developed. The verified kernel discussed in Chapter II is not yet commercially available, though this piece can easily be replaced. The verified version of the kernel is almost identically similar in code-base and operation [29]. The basic libraries used to build the prototype device have not yet withstood the rigors of formal verification, but the task of building a minimal library necessary to implement the trusted features of this device is trivial compared to the implementation and verification of a specific construction of the target architecture.

This OKL4 microkernel operates on modern hardware which is not dissimilar to hardware being used for commercial RTU devices today. As a result, this prototype can be built with reasonable similarities to existing control devices, with the added benefits of the proposed security architecture.

The actual implementation of the prototype closely follows the target architecture, with some shifting of utility functionality within the untrusted components of the system. The memory management architecture of the OKL4 operating system requires the careful placement of utility functionality in order to maintain the separability requirements of a secure industrial system. Minor issues in the Elfweaver tool have reduced options for placement of utility functionality for the purposes of building a prototype, but this is neither a permanent issue, nor one endemic to the concept. This prototype makes use of commodity hardware for both computational power and the generation of physical IO. Throughout the development, effort and care was taken to minimize contact between cells, and simplify the costly inter-process communications (IPC) calls which have plagued microkernel operating systems in the past. The net result is a prototype which closely follows the target architecture, implemented with the OKL4 microkernel operating system, paired with commodity hardware. The device created is not dissimilar from existing RTU devices, and can be used to judge performance, security and feasibility of the proposed architecture when applied to real hardware with one example of a secure microkernel based operating system.

During basic performance testing of the prototype device, as described in Chapter V, the results are acceptable for a wide variety of SCADA applications. The IPC interactions and IO hardware require on the order of 500 microseconds to perform their duties, while the security layer imposes a delay between 20 and 100 milliseconds. These tests have sought to eliminate external interference, and solely represent the time required by the prototype software. The device meets a basic level of performance necessary to move on to more complex testing and development of security and performance features within the target architecture.

### 6.2 Future Directions

The development of a security architecture and prototype which can achieve an objective degree of security opens up a realm of many possibilities. Moving forward, it will become necessary to perform additional testing with the devices in a simulated plant environment, for a more complete understanding of performance. This can also be extended into the control of a real plant, utilizing commodity control devices in a larger control network. This will require the implementation of industry standard SCADA communications protocols, implementing the role based access control features of the prototype device.

Additional development and testing could explore the use of the device as a security layer in an existing SCADA network.  In this case, the security layer would remain unchanged, while a network IO layer resides on either side of the security layer. One network interface would be exposed to the larger "untrusted" network of control devices, while the second network interface would connect directly to an existing control device, allowing for the retrofitting of verifiable security into networks of existing, legacy control devices.

Currently, the prototype firmware is loaded directly into RAM on the flash device. While this is useful for development, future prototypes may choose to optimize long term performance by placing firmware on the Gumstix's built in flash, or implementing execution-in-place from a secure digital memory card.  This memory card adds the benefit of being field replaceable, resulting in a device which can easily be updated, without compromising network security with "over-the-air" update features.

While the OKL4 team has reached a major milestone in formally verifying a kernel, there are many additional avenues of research which must be followed before the OKL4 "system" can lay claim to total verification.  The verified kernel makes two basic assumptions:  The compiler is trustworthy, and the hardware performs deterministically. Although some initial research has been performed in the area of compiler verification, no verified compiler yet exists.  As the OKL4 operating system is written in standard C, a minimal verified compiler will compile the OKL4 operating system.  The Elfweaver tool is another component of the compilation stack which has not been verified. As an integral part of the OKL4 build system, it must be a trusted component. The most difficult assumption to follow through the path of verification, however, is the hardware. The microprocessor, and particularly the memory management unit (MMU) must behave precisely as designed in order for the verified software pieces to achieve trustworthiness.

Following further performance testing, security testing utilizing a broad range of possible attack vectors will verify the architecture fundamentally, as well as its implementation. With the use of a formally verified kernel, the eventual goal of the architecture and its prototype should be to achieve a similar level of formal verification. This process is uniquely cumbersome, especially when using the unverified kernel derivative, with unverified libraries. While replacement of these components with verified analogs is eventually necessary, the first steps of verification can begin before this point. Performing attacks is simply one step in this process which should provide high-level feedback about the implementation of the architecture.

# REFERENCES

[1] J. Hieb and J. Graham, "Designing Security-Hardened Microkernels For Field Devices", in *IFIP International Federation for Information Processing*, Volume 290; Critical Infrastructure Protection II, eds. Papa, M., Shenoi, S., (Boster: Springer), pp. 129-140., 2008

[2] IEEE, *The Authoritative Dictionary of IEEE Standards Terms*, 2007.

[3] Anonymous, "C37.1-2007 IEEE Standard for Scada and Automation Systems."

[4] W.J. Ackerman and W.R. Block, "Understanding supervisory systems," *IEEE Computer Applications in Power*, vol. 5, 1992, pp. 37-40.

[5] D.J. Gaushell and W.R. Block, "SCADA communication techniques and standards," *IEEE Computer Applications in Power*, vol. 6, Jul. 1993, pp. 45-50.

[6] Y. Wang, B.-tseng Chu, and U.N.C. Charlotte, "sSCADA : Securing SCADA Infrastructure Communications," *City*, 2004, pp. 1-13.

[7] M. Majdalawieh and F. Parisi-Presicce, D, "DNPSec: Distributed Network Protocol Version 3 (DNP3) Security Framework," *Advances in Computer,*, vol. 3, 2006.

[8] E. Byres and P. Eng, "The Myths and Facts behind Cyber Security Risks for Industrial Control Systems The BCIT Industrial Security Incident Database ( ISID )," *Security*, 2004, pp. 1-6.

[9] M.T.O. Amanullah, a Kalam, and a Zayegh, "Network Security Vulnerabilities in SCADA and EMS," *2005 IEEE/PES Transmission & Distribution Conference & Exposition: Asia and Pacific*, 2005, pp. 1-6.

[10] J. Pollet, "Developing a solid SCADA security strategy," *2nd ISA/IEEE Sensors for Industry Conference,*, 2001, pp. 148-156.

[11] M. Naedele, "Addressing IT Security for Critical Control Systems," *40th Annual Hawaii International Conference on System Sciences (HICSS'07)*, 2007, pp. 115-115.

[12] E.B. Fernandez and M.M. Larrondo-Petrie, "Designing Secure SCADA Systems Using Security Patterns," *2010 43rd Hawaii International Conference on System Sciences*, 2010, pp. 1-8.

[13] A. Miller, "Trends in Process Control Systems Security," *IEEE Security and Privacy Magazine*, vol. 3, Sep. 2005, pp. 57-60.

[14] A.K. Wright, J.A. Kinast, and J. Mccarty, "Low-Latency Cryptographic Protection for SCADA Communications," *System*, 2004.

[15] T. Leemhulis, "What's New in Linux 2.6.35," *The H Online*, 2010.

[16] V. Marala, *The Build Master: Microsoft's Software Configuration Management Best Practices*, Addison-Wesley Professional, 2005.

[17] T.J. Ostrand and E.J. Weyuker, "The distribution of faults in a large industrial software system," *ACM SIGSOFT Software Engineering Notes*, vol. 27, Jul. 2002, p. 55.

[18] V.R. Basili and B.T. Perricone, "Software errors and complexity: an empirical investigation0," *Communications of the ACM*, vol. 27, Jan. 1984, pp. 42-52.

[19] J. Liedtke, "On micro-kernel construction," *ACM Symposium on Operating Systems Principles*, vol. 29, 1995, p. 237.

[20] A.S. Tanenbaum, J.N. Herder, and H. Bos, "Reliable and Secure ?," *Computer*, vol. 39, no. 5, pp. 44-51, 2006.

[21] M.T. Alexander, "Organization and features of the Michigan terminal system,"

*Proceedings of the November 16-18, 1971, fall joint computer conference on - AFIPS '71 (Fall)*, 1971, p. 585.
[22] J.M. Rushby, "Design and verification of secure systems," *ACM SIGOPS Operating Systems Review*, vol. 15, Dec. 1981, pp. 12-21.
[23] NIAP CCEVS, "Green Hills Separation Kernel Verification," *Evaluation*, 2008, pp. 2007-2007.
[24] R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, a Forin, D. Golub, and M. Jones, "Mach: a system software kernel," *Digest of Papers. COMPCON Spring 89. Thirty-Fourth IEEE Computer Society International Conference: Intellectual Leverage*, 1989, pp. 176-178.
[25] Research In Motion, "QNX Neutrino RTOS Secure Kernel Brief," 2009.
[26] J. Liedtke, "Improving IPC by kernel design," *ACM Symposium on Operating Systems Principles*, vol. 27, 1994, p. 175.
[27] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: formal verification of an OS kernel," *ACM Symposium on Operating Systems Principles*, 2009, pp. 207-220.
[28] D. Elkaduwe, G. Klein, and K. Elphinstone, "Verified Protection Model of the seL4 Microkernel," *Proceedings of the 2nd international conference on Verified Software: Theories, Tools, Experiments,* 2008.
[29] G. Heiser, "Do Microkernels Suck?," *Linux.conf.au,* 2008.
[30] G. Klein, J. Andronick, and K. Elphinstone, "seL4: formal verification of an operating-system kernel," 2010, pp. 107-115.
[31] G. Klein, "A Formally Verified OS Kernel . Now What ?.", *Proceedings of the First International Conference on Interactive Theorem Proving*, 2010, pp. 1-7.
[32] G. Klein, "The L4 . verified Project — Next Steps.", *Proceedings of the 3rd International Conference on Verified Software: Theories, Tools, Experiments*, 2010, pp. 86-96.
[33] C. Beaver, D. Gallup, and W. Neumann, M, "Key management for SCADA," *Cryptog. Information Sys*, 2002.
[34] J. L. Hieb, S. C. Patel, and J. H. Graham, "Security Enhancements for Distributed Control Systems," in *IFIP International Federation for Information Processing*, Volume 253, Critical Infrastructure Protection, S. Shenoi and E. Goetz, Eds. Boston: Springer, pp. 133-146, 2007.
[35] J.L. Hieb, "Security Hardened Remote Terminal Units for Scada Networks," Ph.D dissertation, Dept. Elect. Eng., Univ. of Louisville, Louisville, KY, 2008.
[36] J. Graham, J. Hieb, "Security Enhanced Network Device and Method for Secure Operation of Same", U.S. Patent Application 12/726 105, 2010.
[37] Intel, "Intel PXA27x Processor Family Developer's Manual". 2006, Available: http://www.marvell.com/products/processors/applications/pxa_family/pxa_27x_dev_man.pdf

## APPENDIX I

## LAB MANUAL

Intelligent Systems Research Laboratory

# Secure RTU Prototype

# Lab Manual

3/29/2011

Brad Luyster

### Section 1 – Introduction

This laboratory manual will serve as an introduction to the prototype hardware and firmware for the security hardened remote terminal unit for SCADA systems. This provides a “state of the project” type of overview, and is not meant to outline any strict rules which must be adhered to. Rather, this document should serve as a reference to the state of the build tools as of Spring 2011. All information is subject to change, and if it is developmentally convenient to effect such a change, it would be wise to deprecate any information contained herein. Section 2 will provide an overview of the hardware used, including the custom IO hardware assembled for the prototype. Section 3 will provide configuration details for the OKL4 build system. Section 4 contains a short overview of the code versioning software used for tracking this project.

All the information contained herein is accurate to April, 2011. With the exception of the version of the OKL4 operating system used, all other information contained in this document is freely mutable. This document is intended to provide a brief introduction to the systems used in the past and present of this project, so that informed choices may be made regarding future development.

Before reading this manual, it is assumed that the reader has a basic understanding of the C programming language, and has previously read the Summer and Fall 2010 progress reports, as well as “A Security Hardened Field Device for SCADA Systems.” This document expands and extends on the information presented previously. More detail on various progression caveats and stumbling blocks may be found in those aforementioned documents.

## Section 2 – Hardware

There are two primary components to the hardware used in the development of the prototype RTU. The Gumstix boards and associated accessories are commercial development tools which provide an out-of-box turnkey solution for ARM development. Although these tools are intended for embedded Linux development, the OKL4 operating system supports the Gumstix hardware out-of-the-box. The second major component is the in house developed IO hardware. This IO hardware generates IO signals not dissimilar from existing commercial devices. The Gumstix board is unable to generate analog IO out of box, and the additional hardware provides a further separation layer for IO, in both software and firmware. Should the custom IO hardware fail, it is unlikely to also destroy the more expensive Gumstix board.

### *Section 2.1 – Gumstix*

The Gumstix board is the basis of the prototype RTU. This board contains a 400MHz ARM Xscale processor, with 16 MB of Ram and 64 MB of Flash memory. A micro sd socket is also included. Importantly, the Gumstix board includes provisions for expansion through two external connectors. These connectors allow boards to be attached to the front and back of the Gumstix board using ZIF sockets and screws with spacers. Care must be taken to align the modules and the screws holding the assembly together.

The prototype Gumstix stack includes the Netstix expansion board, which provides an SMC9117 ethernet adapter chip. The prototype firmware implements a minimal number of a features for this network chip, forgoing software or hardware based queuing, or interrupt receive and transmit management. Although these features would enhance the performance of the prototype device, they have not yet been implemented due to the difficulty of doing so.

A Console VX extension is also attached to the prototype device. This extension provides headers for the three serial ports provided by the Xscale processor, and breakout pins for the I2C IO devices described in Section 2.2. Additional digital IO may be provided by this board, if desired. More information about the configuration of the prototype hardware, including pinouts and documentation of which UART port it connected where may be found in the Summer 2010 progress report. Resources pertaining to the Gumstix board are located in Section 6 of this document.

### *Section 2.2 – IO Hardware*

For IO generation, the prototype RTU uses four ICs. These devices are connected to the Inter-IC bus of the Xscale. The I2C bus is designed to be a simple, easy to implement communication standard for local IC communications. This bus is pulled high by two 4.7 kilo-ohm resistors, allowing bus contention to be automatically discovered by the attached devices. If a device attempts to send a logic high, but observes a logic low on the bus, contention has occurred, and the contending device ceased communication. As a result, devices transmitting the most zeros will tend to "win" contentions.

These IO devices have been chosen for their similarity to existing commercial examples of RTUs. These include a digital IO chip, an analog input chip, and two analog output chips, for a total of eight digital IO, eight analog in, and two analog out. Although the Gumstix board can generate its own digital IO, a digital IO IC is utilized to keep all external IO in the same memory space, and to isolate digital IO from the Xscale chip itself. These IO chips operate at a bus speed of 400kHz.

I2C is an extremely well documented method of communicating with embedded devices. Some resources to this effect are found in Section 5 of this document. In summary, a master device is attached to several slave devices. When the master device wishes to communicate, it sends an I2C Start command, followed by a seven bit address, with a single bit indicating the direction of desired communication (read or write). If a device with the given address is attached to the bus, it sends an acknowledgment. If the direction is read, the slave device begins transmitting data, and the master acknowledges the data. If the master is done reading data, it simply ceases acknowledging incoming bytes, and sends a stop command. If the data direction is write, the master begins sending data, followed by an acknowledgement from the slave device. If the slave cannot handle any further incoming data, it does not acknowledge. If the master does not wish to send any more data, it simply sends a stop. All transactions must terminate in a stop command to avoid bus contention.

Each device implements different command sets on the I2C bus, requiring different arrangements of reads and writes. The data sheet for each component will include multiple examples of interfacing with the device.

### *Section 2.2 – Development PC*

The development PC is running the Ubuntu Linux operating system, and contains all the utilities required to actively write code within the OKL4 / Gumstix environment, as well as code versioning and debug interface software. Although the build tools for OKL4 are specific to Linux, the choice of PC and Linux distribution are arbitrary. The build system consists of a custom GCC cross compiler, the ElfWeaver software, and OKL4 SDK. Other necessary tools include a TFTP server, and the git version tracker.

After code is compiled, the ElfWeaver program generates a bootable ELF file. The prototype hardware loads code into RAM upon each system reboot by using TFTP. The Make files in the project automatically upload a bootable ELF file to the TFTP server root directory. The bootloader on the prototype requests this file, and boots. More information is provided in Section 4.

## Section 3 – The Build System

The tools required to build the firmware for the prototype RTU are available on the OKL4 Website (http://www.okl4.org/). The OKL4 SDK for Xscale and the EABI toolchain are required, while the OKL4 base system is helpful when trying to determine the operation of some base functionality. These tools must run under Linux. Documentation is provided, with detail on how to install and integrate with the build environment. Section 3.1 will detail the specifics of the Linux environment currently being used, Section 3.2 will provide an overview of the GCC tool-chain used to compile individual cells. Section 3.3 discusses the OKL4 SDK, and Section 3.4 reviews the Elfweaver tool, and all its caveats.

### *Section 3.1 – Build PC*

Currently, the Linux configuration used for development includes a base of Ubuntu 10.10. Aside from the OKL4 tools, the only necessary tools are git, a TFTP server, and a serial port terminal such as minicom. The configuration of git and the serial port terminal are trivial and self-evident. The TFTP server is slightly more complicated, and it would be prudent to test remotely using a TFTP client located on the local network. Problems tend to arise in unix file permissions for the TFTP directory, and its contents. Currently, permisisons for the TFTP server directory and all files therein are 777, although this is traditionally bad practice.

With the build system configured in this manner, it is possible to develop code from any environment, commit this code to the server using git (see Section 5), and remotely compile the code using this development machine. Simply rebooting the prototype RTU will refresh the code loaded into RAM. If local compilation is desired, it must be done under Linux.

### *Section 3.2 – Uboot and TFTP*

The Gumstix board uses the Uboot boot loader to load code into the RAM of the Gumstix and begin execution. The bootloader automatically executes a script which loads code from the custom development PC via TFTP. In order to stop this process, follow the on screen instructions before boot. From the factory, this bootloader uses the following commands to load a Linux operating system from Flash memory:

```
setenv bootargs console=ttyS0,115200n8 root=1f01 rootfstype=jffs2

reboot=cold,hard

fsload a2000000 boot/uImage

bootm a2000000
```

Runing this code at the uboot prompt restores the factory default linux installation.

The modified boot script loads code over TFTP.

```
ipaddr = 192.168.1.77

serveriip = 192.168.1.16

tftpboot a2000000 image.boot

bootelf a2000000
```

The server IP is the IP address of the PC containing the boot code. “image.boot” is the file name in the TFTP directory of the development PC. If either of these things change, the boot script must be changed. This can be done from the uboot command line.

### *Section 3.2 – Code Organization and Compilation*

Within the project source tree, there are directories for each cell. There is also a central directory of libraries which can be utilized by all cells. Under the cell directory, there is a central Makefile, and an XML file which determines how the cell is weaved with all other cells. A “src” directory contains the “src” tree, while the “inc” directory includes headers and includes files. The “build-debug” directory contain previously compiled source code. The exception to this rule is the “Utility” cell.

The utility cell is designed to spawn multiple child threads through ElfWeaver. These are compiled as a single cohesive unit, and each threads' start point is loaded at run-time after ElfWeaver generates the appropriate initialization code. As a result, each thread has its own directory which is analogous to the “cell” subtree described above. The Makefile located in the Utility directory handles the traversal of all sub-tree directories.

Code is compiled using the GCC cross compiler. Each cell is compiled as through it were completely independent from all other cells: its own independent elf is created by GCC and the linker. These are weaved together with the kernel code later. As a result,the make scripts located throughout the source tree simply traverse deeper levels of the source tree until they encounter a single cell, which is compiled singly. The next cell is then similarly compiled, until all cells are compiled.

### *Section 3.3 – The OKL4 SDK*

The OKL4 SDK contains precompiled kernel images which include different features of the kernel. The prototype uses the micro-debug kernel. This kernel includes provisions for a kernel debugger, which can be accessed by hidding the “Escape” key during operation. All commands are accessed through a menu, which is nagivated by various upper and lower case alphanumeric characters. A question mark character (“?”) will display all available options under the current menu tree.

All versions of the kernel use the same machine.xml. This machine.xml is used to define the hardware available to the operating system. This includes physical memory sections, their start addresses and size, and the available memory page sizes. The machine.xml file also defines all memory mapped hardware, and their associated interrupts. These device defines allow the use of the “use_device” tag in cell-specific XML files. This simply maps the memory spaces and interrupts to a particular cell without requiring any complicated configuration of memory access constraints. The cell is simply granted unfettered access to the resources alloted to that device, and no other cells may map the same device. Interestingly enough, these cell's memory spaces may be mapped in a read-only manner to other cells, which is used in the prototype for timing testing. Timing operations simply observer a free-running counter value, and calculate the difference in time.

### *Section 3.4 – ElfWeaver*

The ElfWeaver tool gathers the ELF files generated during the compilation process, and combines them with the selected kernel image, using the XML files in each cell's directory to generate the correct cell environment (priority, threads, heap and stack size, and so on). ElfWeaver is executed immediately following the linking proceess, and gathers all the XML files in the directory structure (as defined by the first lines in the main Makefile), and begins to divvy up system resources. These XML files must exist in the root directory of each cell pointed by the make file. The ELF files for each cell may be located elsewhere, as is indicated by the XML file header.

There are dozens of possible XML tags that may be used to configure the environment generated for each cell. The ElfWeaver tool performs a remarkably complicated job, essentially providing the separation and security that makes the OKL4 operating system appealing. This process requires a lot of CPU time, and is not enhanced by multiple cores. Unfortunately, the ElfWeaver tool is a miasma of python scripts, and contains quite a few bugs, ranging from annoying to critical. Among the bugs encountered through the development of the prototype RTU, two required significant effort to work around.

The first bug causes secondary threads to spawn improperly. In the prototype RTU, utility functionality and physical IO are combined in a single cell. All of these processes operate as separate threads. Although the OKL4 SDK includes examples of spawning threads from within a cell, this process is not trivial, and would require that capabilities be passed around through IPC, in order for remote cells to communicate with these run-time generated threads. Threads which are spawned in ElfWeaver are much simpler to execute. All that is required is a simple XML tag which passes the star locatoin of a thread to the ElfWeaver. Unfortunately, this does not work properly with the base release of Elfweaver. Instead, ElfWeaver simply writes the start location of these newly spawned threads as the same locatoin as the Idle thread. This results in a group of threads which appear to do nothing. This problem was caused by a function call in the ElfWeaver failing to pass the start location defined in the XML file. The default start location is the location of the idle thread. Simply adding this declaration in the functino call solved the bug.

The remaining bug has been far more insidious. This bug causes a failure to compile if more than three cells are built in ElfWeaver. Regardless of heap and stack assignments, no more than three cells will make it through the weaving process. The error halts weaving, and results in no new image being generated. This bug is located in the virtual to physical memory mapping process. This file, /tools/pyelf/weaver/allocator.py , handles the segmentation of physical memory, and the division into virtual memory segments. This segmentation results in "slices" of memory which do not neatly fit the page boundary. These slices cannot be used, and are essentially wasted. For some reason, excessive waste is occurring, resulting in insufficient memory to support more than three cells.

## Section 4 – Using GIT

GIT is a tool for managing revisions and changes to source code. Devleoped by Linus Torvalds for the Linux Kernel, this tool is uniquely suited for projects with many disparate contributors, operating with many disparate workflows. The basic unit of trade in GIT is a change. A change made to a file is tracked, and when this file is committed, the change is recorded. A local copy of all files is always maintained. If GIT is not desired, it may simply be disregarded. For a much better explanation of GIT's usage, see the ProGit ebook listed in Section 6.

## Section 5 – Resources

Microchip, Datasheet for MCP23009, http://ww1.microchip.com/downloads/en/DeviceDoc/22121b.pdf

Microchip, Datasheet for MCP4725, http://ww1.microchip.com/downloads/en/DeviceDoc/22039c.pdf

Analog Devices, Datasheet for AD7997, http://www.analog.com/static/imported-files/data_sheets/AD7997_7998.pdf

Marvell/Intel, Datasheet for Xscale PXA 270, http://www.marvell.com/products/processors/applications/pxa_family/pxa_27x_emts.pdf

Scott Chacon, “Pro Git”, E-book, http://progit.org/book/

OKL4 3.0 Documentation and Downloads, http://wiki.ok-labs.com/Microkernel

# APPENDIX II

# SELECTED CODE

This appendix contains selected code samples from throughout the build system, including more examples of XML declarations, a summary of the data structure and algorithms used in the role-based access control system, and the physical IO interfacing.

## Appendix 2.1 – Utility Cell Elfweaver XML File

```
<okl4 priority="255" clists="256" file="utility" kernel_heap="0x400000"
mutexes="256" name="utility" spaces="32">
      <use_device name="clock_manager_dev"/>
      <use_device name="gpio_dev"/>
      <use_device name="stuart_dev"/>
      <use_device name="i2c_dev"/>
      <heap size="0x100000"/>

      <thread name="serial" start="serial_main" priority="255"/>
      <thread name="i2c" start="i2c_main" priority="255"/>
      <thread name="test" start="test_main" priority="255"/>
      <thread name="hwman" start="hwman_main" priority="255"/>
      <memsection cache_policy="uncached" name="timer_vaddr"
phys_addr="0x40A00000" size="0x1000" attach="r" />

      <environment>
            <!-- Need both these caps for backwards compatibility /-->
            <entry cap="/utility/serial" key="SERIAL_CAP"/>
            <entry cap="/utility/i2c" key="I2C_CAP"/>
            <entry cap="/utility/serial" key="SERIALSERVER_CAP"/>
      </environment>
      <commandline/>
</okl4>
```

## Appendix 2.2 – Data Structures Use In Role-based Access Control

```
struct point
{
      bitfield ppt;
      pointID pointid;

} point;

struct permission
{
      bitfield operations_allowed;
      struct point *ppoint;
} permission;
```

```
struct permission_access_constraint
{
      int pacType;
      struct point *pacpoint;
      struct timeconstraint *timec;
      struct dayconstraint *days;
      //locationconstraint location;
} permission_access_constraint;

struct role_access_constraint
{
      bitfield roles;
      struct timeconstraint *timec;
      struct dayconstraint *days;
      //struct locationconstraint location;
} role_access_constraint;

struct role
{
      bitfield point_type_controls;
      struct permission permissions[NUMPOINTS];
      int numPacs;
      struct permission_access_constraint *PACS;
      int testArray[NUMPOINTS];

} role;

struct user
{
      userID myId;
      uint32_t *secret;
      bitfield roles;
      int numRacs;
      struct role_access_constraint *RACS;

} user;

struct authedUser
{
      userID myUser;
      struct authedUser * nextUser;
} authedUser;
```

## Appendix 2.3 – Physical IO Interfacing

```
char requestDigital(char selected)
{
      char retval;

      retval = getAllDigital();

      switch(selected) {
            case 0x00:
                  return retval & 0x01;
            case 0x01:
                  return (retval & 0x02)>>1;
            case 0x02:
                  return (retval & 0x04)>>2;
            case 0x03:
                  return (retval & 0x08)>>3;
            case 0x04:
                  return (retval & 0x10)>>4;
            case 0x05:
                  return (retval & 0x20)>>5;
            case 0x06:
                  return (retval & 0x40)>>6;
            case 0x07:
                  return (retval & 0x80)>>7;
            default:
                  return 0xff;
            }
}

char setDigital(char selected, char state)
{
      char current = getAllDigital();
      char result;
      if(state == 0)
      {
            result = current & ~(1<<selected);
      } else {
            result = current | (1<<selected);
      }
      i2c_start(MYGPIO);
      i2c_send(0x09);
      i2c_send((okl4_word_t)result);
      i2c_stop();

      return 0x00;
}
```

```
okl4_u16_t getAnalog(char selected)
{
      okl4_u8_t data1;
      okl4_u8_t data2;
      okl4_word_t channel;

      if(selected > 0x07)
      {
            return 0;
      } else {
            channel = (selected << 4) | 0x80;
            i2c_start(MYADC+0);
            i2c_send(channel);
            i2c_start(MYADC+1);
            data1 = (okl4_u8_t)i2c_get_ack();
            data2 = (okl4_u8_t)i2c_get_nak();
            i2c_stop();

            return data1<<8 | data2;
      }

}

char setAnalog(char selected, okl4_u16_t data)
{
      if(selected == 0x00)
      {
            dac(MYDAC1, (okl4_word_t)data);
      } else if(selected == 0x01)
      {
            dac(MYDAC2, (okl4_word_t)data);
      }
      return 0x00;
}
```

# VITA

BradLuyster@gmail.com

## Brad Luyster

1679 Trigg St.

502-533-9013

Louisville, KY 40213

## Education:

**University of Louisville** **2005 - 2009**

***Bachelor of Science*** in Electrical and Computer Engineering.

Highest Honors. GPA 3.76

## Experience:

**University of Louisville: Research Assistant** **January 2010 - Present**

Research Assistant in the Electrical and Computer Engineering Department's Intelligent Systems Research Laboratory. Investigated and developed methods for securing SCADA Remote Terminal Units using mathematically verified Microkernel operating systems on low power embedded processors.

**University of Louisville: Information Technology** **April 2006 - Dec 2009**

Assisted Students, Professors and Staff of the Computer Engineering and Computer Science Department in day-to-day troubleshooting and tech support, culminating in the design and construction of the Information Technology infrastructure for the Duthie Center for Engineering.

**Federal Bureau of Investigation: Intern** **January 2007 – August 2008**

Employed for 3 semesters at the Kentucky Regional Computer Forensics lab, assisted in investigating computer crime, as well as developing methods to verify the integrity of evidence gathering software.

## Awards and Publications:

“A Prototype Security Hardened Field Device for Industrial Control Systems”, International Conference on Advanced Computing and Communications Proceedings, Orlando, Florida, September 2010.

IEEE Student Section Service Award, Spring 2010

## Leadership:

**Tau Beta Pi Engineering Honors Fraternity**
Active member, 2007-Present.

**IEEE Student Section**

Served as an officer of the University of Louisville Student IEEE from 2008-2010.