View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by University of Louisville

University of Louisville

ThinkIR: The University of Louisville's Institutional Repository

Electronic Theses and Dissertations

7-2012

Microkernel security evaluation.

Kevin C. Kurtz

University of Louisville

Follow this and additional works at: https://ir.library.louisville.edu/etd

Recommended Citation

Kurtz, Kevin C., "Microkernel security evaluation." (2012). *Electronic Theses and Dissertations.* Paper 784.
https://doi.org/10.18297/etd/784

This Master's Thesis is brought to you for free and open access by ThinkIR: The University of Louisville's Institutional Repository. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of ThinkIR: The University of Louisville's Institutional Repository. This title appears here courtesy of the author, who has retained all other copyrights. For more information, please contact thinkir@louisville.edu.

MICROKERNEL SECURITY EVALUATION

By

Kevin C. Kurtz
B.S., University of Louisville, 2011

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

July 2012

MICROKERNEL SECURITY EVALUATION

Submitted By:___________________________________

Kevin C. Kurtz

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

Acknowledgements

I would like to thank Dr. James Graham for providing me with the opportunity to undertake this research as well as his patience and time during the experiment. I must also thank Dr. Jeffrey Hieb, without whom this thesis would have never materialized. His endless patience throughout the last several months and great guidance have been the basis for my motivation. My surrogate father, John Pensy, is responsible for my choice of majors and research. He got me interested in electrical experimentation long before I knew what an engineer was or how far I would eventually take my education. I have him to thank for my love of experimentation and all of my interest in the electronic and programming worlds. Finally, I could not have come this far without my mother and father, Rhonda and Kevin Kurtz. My mother's belief that I can do whatever in life I choose, no matter the hurdles, got me here today and my father's work ethic and push towards higher education. He would settle for nothing less than for me to receive a college degree and live a happy life. His help and guidance will never be forgotten, may he rest in peace.

Abstract

This thesis documents the successful development and testing of a more secure industrial control system field device architecture and software. The implementation of a secure field device has had limitations in the past due to a lack of secure operating system and guidelines. With the recent verification of OK Labs SEL4 microkernel, a verified operating system for such devices is possible, creating a possibility for a secure field device following open standards using known security protocols and low level memory and functionary isolation. The virtualized prototype makes use of common hardware and an existing secure field device architecture to implement a new level of security where the device is verified to function as expected. The experimental evaluation provides performance data which indicates the usefulness of the architecture in the field and security function integration testing to guarantee secure programs can be implemented on the device. Results of the devices functionality are hopeful, showing useful performance for many applications and further development as a fully functional secure field device.

Contents

Acknowledgements ... iii
Abstract ... iv
List of Tables ... 1
List of Figures ... 2
Chapter I ... 3
Introduction ... 3
Chapter II ... 6
Literature Review ... 6
SCADA Security for Field Devices ... 6
OPSAID ... 7
LEMNOS ... 8
Microkernel Architecture ... 9
Microkernel History ... 9
Microkernel Design ... 11
Microkernels for Secure Field Devices ... 13
OKL4 Microkernel ... 14
SEL4 Microkernel ... 16
Chapter III ... 17
System Design ... 17
Security Model ... 17
Prior Work ... 22
Memory Isolation ... 23
Communication ... 25
Summary of Design Consederation ... 25
Chapter IV ... 27
System Implementation ... 27
Hardware Emulation ... 28
Software ... 29
Security Isolation and Communication ... 31
CHAPTER V ... 35

Performance Evaluation ... 35
Cell IPC ... 35
Linux VPN Server ... 39
Linux VPN Client ... 42
Linux IPC Program ... 43
CHAPTER VI ... 45
Conclusion ... 45
Summary ... 45
Future Research ... 47
References ... 50
Appendices ... 51
Build Configurations ... 51
Config.mk ... 51
system_two_wombats.h ... 52
Custom Code ... 53
main.c (for IPC Cell) ... 53
ipc.c (IPC inside Linux) ... 58

List of Tables

Table 5.1 – IPC Message Performance.. 38

List of Figures

Figure 2.1: Layered kernel architecture.. 10

Figure 2.2: Microkernel architecture... 12

Figure 4.1: Communication in SEL4 and Memory Segmentation 30

Figure 4.2: Isolated cell approach, including security cells.......................... 33

Chapter I

Introduction

This thesis documents the design, development, and testing of security software in a secure microkernel device. The device could alter the way industrial control system field devices are currently implemented, adding many layers of additional security to common devices without the need for hardware upgrades by using a modern architecture system and open source security software. The software tested will be shown to have more secure access to other devices, its own hardware, networking resources, and communication between its own local software.

Field devices are a critical component of industrial control systems. They are used in many industries, especially utilities, and historically these devices have lacked cyber-security features. In the past, physical access to the device was necessary to attack it; in the last decade there have been networking advances that allow high speed networking to almost any location, no matter how remote. Advances currently available for devices, previously isolated, mean many of them now have remote access and are connected to the Internet. Field devices and subsequently industrial control systems are vulnerable to malicious attacks that could damage their physical systems and have serious environmental impacts, without additional security.

Added security to the networking interface of field devices is not enough. A separate hardware firewall could be added to deter attackers, but this raises additional hardware costs, another device to support, and does not comprehensively secure the device. It is suggested that the device must be secured from every logical point of attack, not just the network interface. The device must be secured from physical access via its own terminal or directly connected serial programming devices, from unwanted network access by a firewall or other discriminating software for both outward and inward communications, and the device must be secured from its own internal software that may have been modified for malicious intent.

A secure device would be an unreasonable goal without guarantee of secure software. This starts at the most basic level of operation; the device kernel. A secure microkernel for which to build the other software systems is a requirement for the entire project. A verified correct microkernel exists for the use in a more secure experiment. The SEL4 microkernel has been formally verified and will be used for the experiment. The kernel is open source and is able to be modified if necessary. However, any modifications will not be verified and therefore should be avoided if at all possible.

This thesis presents a review of literature and research related to this experiment. Chapter II details the extent of the literature review describing the architecture of the system based on previous work, the security features hoping to by implemented in the system, and the microkernel used. Chapter III

discusses the design architecture of the build system, a more in-depth analysis of the security features necessary of a secure device, and the use of memory isolation in the secure microkernel. Chapter IV is an overview of the experiment and how it was designed, showing the operation of the system and the implemented software. It details the use of the software and how it should be implemented to best secure the device. Chapter V shows the software that was implemented before the end of the experiment, testing, performance, barriers overcome through the experimentation process, and how software verification might be used to complete the project. The final chapter explains the outcomes of the experiment, presents conclusions drawn about the project and secure devices, and indicates future research and experimentation directions that may be beneficial for the project.

Chapter II

Literature Review

SCADA Security for Field Devices

SCADA (Supervisory Control And Data Acquisition) systems are currently vulnerable to cyber-security attacks. Many SCADA systems are insecure by today's Internet standards; they have chronic and pervasive vulnerabilities [1]. Many of the current efforts in security assessment involve searching for known vulnerabilities [2]. Computer controlled systems should be subjected to scrutiny and this is often ignored at the management level. With the upgrade of electrical grids (smart grids); transportation systems; and water distribution systems; now is the time to upgrade the security scheme as well. [1]

Field devices are small embedded computers running their own operating system, discussed in a later section. The recent Stuxnet attack shows the importance of securing these devices [3]. SCADA devices control major processes in utilities and industry. An attack on these systems could be devastating. Hijacking of SCADA and field devices can disrupt processes and shut down utilities. In some systems, a simple technique known as SQL injection can be a successful weapon. This is inadequate protection and needs to advance as devices join IP networks.

SCADA systems can have remote vulnerabilities, but can also be affected by inside users that are trained improperly or are malicious. Only authorized

personnel should have access to the interior features of the devices. This shows a need for IP security (firewalls), authentication, secure remote access, and intrusion detection without a significant cost upgrade for vendors. [1]

OPSAID

OPSAID (Open PCS Security Architecture for Interoperable Design) is a program intended to overcome security issues in the short term for SCADA devices. PCS (Process Control Systems) weren't designed with adequate regard for security issues. Communication was typically through serial links at a single location segregated from the outside network. As industry has evolved, so has the need for remote control and diagnostics of systems. PCS devices are now moving to using TCP/IP as the standard communication and off the shelf software for their firmware. Without an added layer of security, anyone with knowledge of the widely used software can control the system. Typical IT systems incorporate secure event logging, authentication, and firewall services; PCS rarely uses any of these.

The OPSAID project was designed to address security issues using established and available IT standards for a corporate network. Using mini-itx computers and the open source Linux operating system Ubuntu, the project has confirmed that it is possible and cost effective to build a more secure PCS field security appliance using open source software with thorough testing. OPSAID is not meant to be a standard for security, as networks and security needs will change in the future. An "all or nothing" standard is inappropriate. The purpose

is to provide a roadmap and proof of concept for vendors to address their own security issues and maintain interoperability with other OPSAID components. [4]

The security features in the OPSAID implementation include:

- Virtual Private Networking/Encryption
- Firewall Services
- Network Intrusion Detection Systems
- Host Intrusion Detection Systems
- Event Logging
- Event Database Storage, Alert Generation & Visualization
- End-device Configuration Session Logging
- Authentication
- Device Management [4]

LEMNOS

The Lemnos project was built upon the OPSAID projects component modules for interoperability. The purpose of the project is to output artifacts referred to as Interoperable Configuration Profiles. The asset is defined by the needs of the owner, both functionality and security. The Lemnos approach is focused on interoperability for secure modules. Much like the OPSAID project, Lemnos is built on open source software, but allows for “best in class” cyber security solutions for various points in their infrastructure. [5]

Microkernel Architecture

A kernel is the lowest level of software abstraction on hardware. Its duties include managing system resources and connecting applications to actual data processing on the hardware level. In a monolithic kernel device drivers, file systems, and many other features are a part of the operating system kernel. These services require privileged access to system resources, usually access to physical memory , and only kernel code can accesses these resources.

In microkernels, most of the features, such as device drivers, file IO, etc. are implemented outside of the privileged mode of the processor. This allows for improved security since these software services are limited to only specific resources. The drawback of this approach is performance. A microkernel will implement the smallest set of operations and abstractions in the kernel and the drivers, file systems, and other functions in user-space. [6]

Microkernel History

In monolithic kernel design, programs in the kernel can access any resources the kernel has access to, all of the physical memory. They are “trusted” not to violate their memory boundaries. This structure grew beyond usability as operating systems grew to enormous proportions. To help calm this growing beast in kernel space, layered operating systems were developed. Modular programming techniques helped to handle the scale of software development. Functions in layered operating systems are organized in a structure

to allow communication and interaction between adjacent layers only. Still, most layers were implemented and executed in kernel mode.

The layered approach, shown in Figure 2.1, helped simplify programming and the size of the kernel, but each layer possessed a great deal of functionality. A change in one layer could cause undesirable effects in adjacent layers, difficult to trace bugs, and numerous other problems. The interaction between these layers made it exceptionally difficult to build in security due to every layer being able to access all functions of the adjacent layers. A bug in one layer could allow malicious code to gain control of the hardware or disrupt operation of the device entirely.

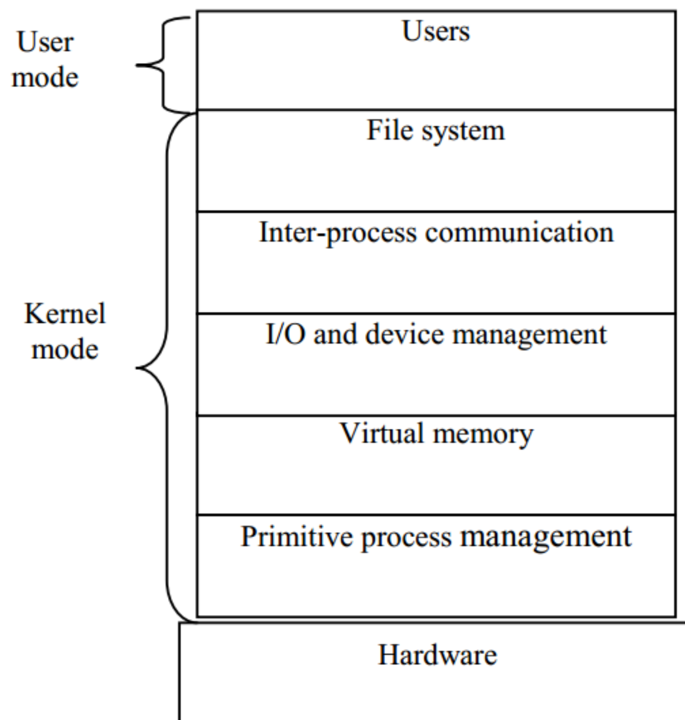

Figure 2.1: Layered kernel architecture [7]

The microkernel was created under the philosophy that only the essential functions of the operating system were implemented in the actual kernel. Less essential functions and applications are built on top of the microkernel. These functions operate in user mode as opposed to the more secure kernel mode. There is no concrete rule as to what is essential and should be compiled into the kernel, but the common definition is for most services that were previously part of the operating system are now external to the kernel as a separate module or subsystem that interacts with the kernel and with each other; these services can include security services, windowing systems, virtual memory managers, file systems, and device drivers. [7]

Microkernel Design

The design of a microkernel is implemented to solve some of the problems mentioned in monolithic kernels and layered operating systems mentioned above. A microkernel architecture is a horizontal implementation of the abstraction system, as opposed to the vertical model of a layered architecture. All operating system components external to the microkernel are implemented as server processes that interact with one another on a peer basis in user mode, shown in Figure 2.2. To communicate, typically they will send messaged through the microkernel via IPC calls. This allows bugs and unintended actions to be more easily traced since layers are not talking to each other, but instead can only interact by way of loggable messages through the microkernel. This allows for a higher level of security, accountability, and more controllable operation.

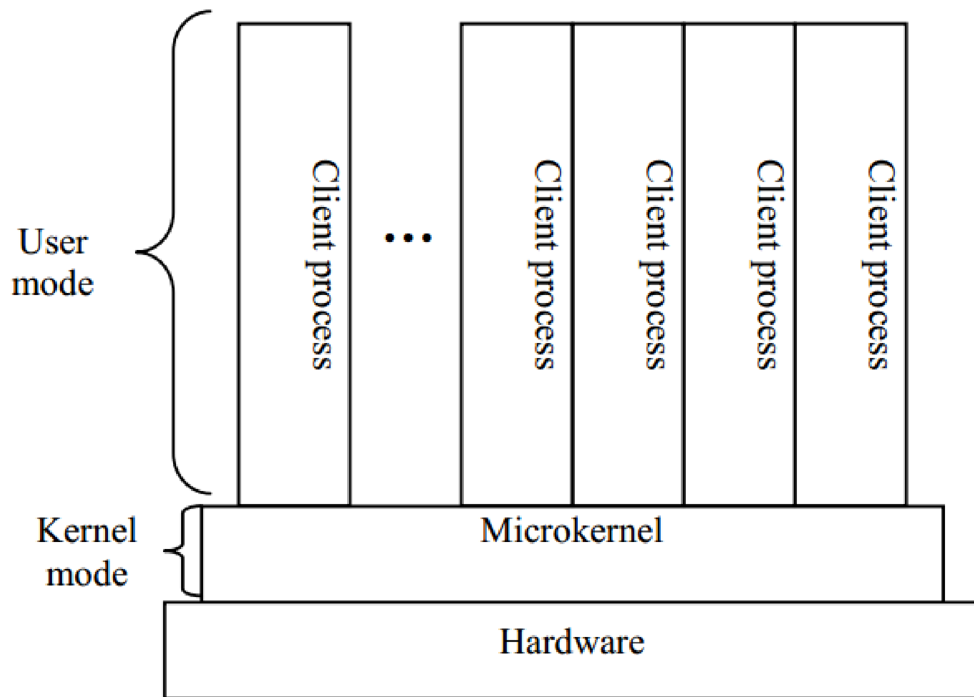

Figure 2.2: Microkernel architecture [7]

The microkernel is required to act as the message exchange between the user mode components. The microkernel will validate the messages, relay them to the user mode recipients, and grants access to hardware. The microkernel adds extra security by performing message transfers through a protection function; it prevents messages from being passed unless exchange between the components is allowed. This prevents hijacking of drivers or other system resources by unauthorized components. This is a client/server architecture within a single computer, where each component can be thought of a peer client on a network and they can only transmit messages, which can be filtered, through the server. These messages can be sent to other components and request the primitive functions compiled into the microkernel.

Microkernel design creates a uniform interface for processes to make requests. A component does not need to make a distinction between kernel-level and user-level services as all requests will be processed by the kernel. This allows for easy extensibility; newer components or modules can be installed on the microkernel to allow for the use of newer hardware, alternative file systems, and new software techniques come to light. Allowing a modest microkernel to be programmed once and used even after computer upgrades or software technologies change. The upgraded services do not require all the services to be updated.

A Microkernel architecture is more efficient by design. Components can be easily removed for a smaller footprint or replaced for a system lacking powerful hardware. The memory manager can be easily replaced to deal with small amounts of RAM and a lack of swap space if the hardware requires it. [7]

Microkernels for Secure Field Devices

Field devices connect sensors, actuators, and other input/output peripherals to a control network. This provides remote measuring and control capabilities. These devices must be secured to avoid unauthorized control of utilities and other applications. The security of these devices directly reflects to the safety of the operators and everyone involved with the use of the utility. Unauthorized access to field devices can also cause massive physical damage to expensive equipment.

To secure field devices, the microkernel controlling the hardware has to be secure. No device is impenetrable, but care can be taken to make it as difficult as possible. This was less necessary when devices were on isolated networks; many devices are now accessible via the Internet and therefore must the security scheme must change. Modern secure microkernels employ isolated partitions, each with its own isolated memory and contact to other hardware, software, or instrumentation; This is referred to as the partitions protection domain. This protection domain allows for software of varying levels of security to be decoupled from the microkernel and other less secure applications. [8]

A microkernel for a secure field device will compartmentalize components and allow the trusted computing base (TCB) to consist of only the kernel and trusted security-critical code in kernel mode. All other applications reside in user mode with limited access to the secure areas of the device. The microkernel will determine what trusted resources can be accessed from these less secure compartments and only allow access to security critical code or data when absolutely necessary, all through IPC calls to the kernel [9]. This small amount of secure code and the small size of the kernel allow for the code to be thoroughly checked for possible errors and unintended operation, even allowing for mathematical proof of the code's operation. [8]

OKL4 Microkernel

To use a microkernel for security purposes, the IPC must be fast and efficient or else other process communications might be used, bypassing the

security of the kernel. Most microkernels exhibit poor IPC performance. The L4 microkernel is built to improve the IPC performance of L3 and other pre-existing microkernels [10].

The OKL4 microkernel was designed by OK Labs as a highly flexible, high performance microkernel. Providing a minimal layer of hardware-abstraction on which modules can be built. Each component is isolated in the system from programming errors or malicious code introduced to the system by other components. A feature is only implemented in the kernel if it was impossible to provide the service outside of the microkernel with the same level of security.

OKL4 provides a trust and security implementation using hardware and software mechanisms to enforce security. The API provides time, resource/memory, communication protection, and fault isolation. Using address space control and IPC for each component or thread to communicate the kernel can create separate memory spaces for each component that are independent of one another. These cells are completely isolated and can only communicate through IPC. Each call is verified through the security model that the component has access rights to the hardware, data, or other component it is trying to communicate with, therefore containing malicious code or activity to an individual cell. [11]

SEL4 Microkernel

SEL4 is a mathematically verified version of the OKL4 microkernel. Since the microkernel is the only part of the operating system that executes in the privileged mode of the hardware, there is no protection from faults occurring in the kernel. Every bug could potentially cause physical damage. The kernel is a major part of the TCB that can bypass security.

Using an interactive, machine-assisted and machine-checked proof the SEL4 microkernel was formally verified. This does not mean it is necessarily secure, but that it has been mathematically verified that the C code operates specifically as the kernel should behave. The verification was run on the C code itself. Therefore, the kernel itself is not verified, but the code that it was compiled from was verified to operate as specified. To declare the kernel as verified, one must also assumed the correctness of the C compiler, linker, assembly code, hardware, and boot code. [12]

Chapter III

System Design

It is shown in Chapter II that operating system security in field devices cannot be guaranteed. To help deal with this issue, the microkernel was designed as a horizontal approach to abstraction based on modules instead of a layered monolithic kernel approach. This allows the kernel to segregate memory and permissions based on the needs of the module and can help unwanted access by a corrupt piece of software from accessing all areas of the device.

The SEL4 microkernel is designed and verified as correct, and operates discussed in Chapter II. The experiment is designed around the SEL4 microkernel to determine speed in which communication can be achieved between processes in a secure environment.

Security Model

The security for this project is based off the OPSAID [4] security project discussed in Chapter II. The several aspects of field device security provided by OPSAID are listed below and their relevance to the project is stated. Using these security upgrades for field devices the security and performance of devices can be tested.

Virtual private networking and encryption is necessary in modern devices. Any device operating on the Internet should be located behind a firewall preventing unwanted access. To enter these networks and gain full access to

their resources it is necessary to virtualize a private network. A private network is intended to be used only by the devices physically connected to the local network, where security should be less of a concern. These networks were common before the Internet. Now, with the invent of virtualized networking, it is possible to encrypt a tunnel into a network and allow remote devices to be included in a private network. Only devices on the private, hopefully encrypted, network are authorized to communicate with the device. This does not need to be supported by the device directly, but by some gateway device on the network, although, it can be internalized to the device as it is in the OPSAID project.

Firewall services are included in the OPSAID standard architecture. A firewall is a software tool used to deny communications from certain processes, programs, ports or specific devices. This is useful for blocking unwanted programs from finding an open port on a device or a network. Only approved processes have access to the network. Malicious software running on the device will not be able to create an outside connection without meeting the policy of the firewall. Most home computers and home routers will have some firewall policy that will try and protect the machine. This is common in the PC network environment and should be used for field devices as well. A custom policy can be implemented to not hinder the current functions of the device, but protect from dangerous software and DDOS attacks.

Network intrusion detection systems are used to detect and notify the network administrator of an unauthorized program or user on the network. Typically, these systems operate by analyzing network traffic for suspicious or malicious activity. The use of these systems can be beneficial to any network, but particularly to those that control expensive/dangerous equipment or contain sensitive information. The software can potentially detect an intruder and in some cases severe the connection to the network. In the event that a program or user has bypassed the virtual network security and the firewall for the network, this additional software could be the added layer necessary to finally stop communication with the device.

If a user can get on the network and send commands to a device, the network intrusion detection has a chance to stop them. However, if they penetrate software on the device itself, there may not be suspicious network activity. Host intrusion detection systems cover this base by monitoring and analyzing internals of the system. Host security typically consists of watching memory for unverified modifications, be it to a database or program memory depending on the device and its uses. Using checksums of file sizes, date attributes, and permissions, the system can check for changes that were not authorized. Host security may not prevent access to sensitive information, but it can warn the owner of the information that it has been accessed and possibly distributed. Detection systems are a notification system, not necessarily a prevention mechanism.

Event logging can be useful in determining the fault when a problem does occur. Problems in electronic devices occur for many reasons; sometimes these reasons cannot be determined. For security, the device or the owner needs to know if the error was a software/hardware fault or was caused by mischievous activity. Event logging allows for the device to keep records of errors. These records can contain timestamps, user with access, the process that initiated the error, and other vital information that can help the owner determine if the device's security has been compromised. Event logging is also helpful if an accident occurs in the vicinity of a piece of equipment. The device may have logged activities in the error, from its current operating level, process, or duty to sensor data at the time of the accident; which can be useful for troubleshooting or to determine user error.

Event storage and alert generation are related to event logging. The logged data can be stored in a database for later analysis and any error logs can generate an alert for the device owner. The owner can be notified in real time of an error instead of having to discover the event own their own. If an error occurs the device can instantly send an alert via the Internet or internal network so that the error can be dealt with without unnecessary downtime, be it an intruder, hardware failure, or software error.

Session logging is used to determine what commands were sent to the device and when. When the device is accessed remotely or physically, all actions are logged to determine when a configuration problem was caused. The device

can benefit from logging changes, as the owner knows what was changed and can quickly and successfully reconfigure the device. The reconfiguration can be automated to return the device to a previous configuration with ease when a fault is detected.

Field devices, which originally were accessed only via an isolated a physical network, are now connected using Internet technologies. When isolated, access to the device was granted by lock and key; now these devices can be remotely accessed, potentially through corporate networks, from anywhere in the world. No need to identify a user was necessary if they had physical access to the device, but this is no longer standard. Every electronic network needs some sort of authentication to keep access limited to only privileged users. A minimalistic user authentication and password protection is standard on most PC networks and that is the level of security trying to be reached by the OPSAID project for field devices. Therefore, there needs to be a built in software mechanism to authenticate the user on the device, whether this be a username and password or some other means of verified user authentication.

Remote device management already exists, but could be much more secure. Device management is an important part of having a device, occasionally it will need to be reconfigured or the logs accessed. This should be done through secure software that has access rights to all of the software listed above. This software should be used to view or download the logs, sessions, alerts, firewall

policy, and any other device configuration from one easy resource that can be accessed securely and remotely.

All of this software already exists. It is waiting to be installed and configured for use in field devices and SCADA systems to help protect the device and ensure normal operation. Without these additional software products, a device is vulnerable to many types of attacks and the results can be devastating. Every device should be secured in some way, but Internet enabled devices or any that can accessed remotely need to have many added layers of security to prevent unwanted access and malicious software actions.

Prior Work

Security has been a concern in field and SCADA devices for some time. This is not a new area of research. The OPSAID project sets a plan for implementing better security in these devices. The Lemnos project adds to the security of the modules installed on the device and interoperability between the devices without the added level of security impeding the functionality of the device.

There has been work done in security hardened field devices and operating system security for the devices. Some of the work mentioned in Chapter II needs to be detailed further as it is important to the development of the experiment and the need for the research. Graham and Hieb [8] have researched the need for SCADA security and the inherent issues of securing the devices. Through

their research they have concluded a number of possible future research directions for the field. Using an isolated kernel, the device could possibly be much more secure. Using the OKL4 microkernel as a means of isolating separate processes, the experiment explored IPC communication on hardware and its effectiveness for security applications.

The research of Graham and Hieb was continued by Luyster [9]. His research involved developing a prototype based on the hardened security research using the OKL4 microkernel for RTU control devices and industrial embedded systems. The research suggested that the added layers of security added 20 to 100 milliseconds of delay to IPC calls that typically took 500 microseconds to complete. The research suggests there is a need to test how security additions and IPC calls function on a more secure verified kernel, such as the SEL4 microkernel, now commercially available.

Memory Isolation

The SEL4 microkernel allows for all cells to be isolated from each other within memory. No process may read or modify the memory space of another process. The only communication between the processes is via IPC through the kernel. This allows for a more secure environment than one user space for all of the components of the system to access.

A cell is a concept unique to security software. The OKL4 microkernel isolates specific memory for usage in the cells and allocates these addresses to

only one process. The memory used in these cells is stored in the large bank of memory for the device, typically RAM. This is virtual addressed by the processor and then allocated to a thread, process, or to an entire cell. This allocation is referred to as a protection domain. This is a memory segment that is completely isolated from all other memory segments virtually. There is no way for memory within a protection domain to be accessed by any other processes than the one to which it was allocated.

For this isolation to be possible, no software can have access to direct memory mapping except the kernel. Any other software must use virtual memory addresses that are mapped one to one with physical memory. These virtual memory addresses are then translated by the kernel. The processes and cells have no way of determining absolute memory locations, this is vital to the security of the memory segments. This is in contrast to typical monolithic kernels which allow device drivers and other software modules in the kernel space unrestricted access to the entirety of the system memory.

Like cells, threads operate within a protection domain. However, multiple threads can exist within a single protection domain, whereas multiple cells cannot. A cell can be thought of more like a program, that can have a single running process or consist of multiple threads within its protection domain.

Communication

In the OKL4 and SEL4 microkernels, all communication between different components must be done through IPC calls. There are two types of IPC calls in the OKL4 architecture. These calls are referred to as blocking and non-blocking calls. Blocking calls will stop a process or thread until the corresponding IPC call has completed, either sending or receiving. This is useful if the thread is waiting on access to a locked component and needs the information to continue, however, the possibility exists for a race condition in this situation and it should be avoided where possible. Non-blocking calls will immediately attempt to send or receive the desired IPC call, but may fail if the other component is locked or unready for the call. The failure can be handled in software and the component can wait to send or receive again or continue with the process. This is important to avoid race conditions as a failure is easy to deal with, but a locked process waiting on a device that may never be ready can be devastating to a system unless designed to operate where these situations cannot exist, such as a state machine.

Summary of Design Consederation

It has been suggested that using existing security software available now that the security of field and SCADA devices can be greatly enhanced. By using some or all of the layered security described by the OPSAID project above, a device may be updated/upgraded to be compatible with corporate secure

computer networks without posing significant security risks from outside users and malicious software on the network.

However, these network and device protocols are not secure enough. The device must also protect itself from malicious components installed on the device itself. Using microkernels and software designed for memory isolation the device can protect from unwanted access to hardware and secure data. These components must be able to communicate securely with the device and the other components in the system. This is all possible and an experiment will be designed in Chapter IV to show some of the features of these more secure devices and their performance.

Chapter IV

System Implementation

The purpose of this thesis is to evaluate whether, or not, while using proper security protocols and methodologies, reasonable performance can still be achieved in kernel communication. The design is based on the OKL4 wombat Linux kernel and its provided image. This para-virtualized distribution of Linux runs on the OKL4 kernel, more specifically on the SEL4 microkernel in this experiment, and is an entire functional operating system with virtualized hardware that can communicate with the actual hardware through the base microkernel. The operating system can also communicate with other system components that are running in separate cells.

Using two wombat distributions and the SEL4 microkernel, it can be shown that secure communication can be achieved, that its performance is reasonable, and that isolated components can operate independently of each other without risk of corruption from the other component.

The SEL4 system image was built on an Ubuntu Linux machine using a dedicated cross compiler for the x86 architecture designed for compiling for a generic x86 machine using only the most basic generic hardware. The tool is Crosstool-ng and has cross compiling capabilities for many architectures. The cross compiler compiled the wombat supervisor, timing server, and all other

SEL4 components to be run on the kernel. The kernel was delivered precompiled and was not rebuilt during this experiment.

The kernel modules for this experiment were compiled using the same cross compiler. The default module settings were used with the exception of one. Net filter was not compiled due to compiler errors and was unnecessary for the experiment, therefore was excluded. The module configuration is included in the appendix.

Many different IPC and other SEL4 programs were compiled. The wombat supervisor, the program used to load a paravirtualized Linux image into memory, was compiled and configured to load two identical images in parallel. The timing server was compiled using the default settings and is included with every Wombat Linux image as Linux cannot function without a system clock and the hardware clock is unavailable to the SEL4 kernel.

Hardware Emulation

The Ubuntu Linux machine used to compile the test system was emulated using Sun Virtualbox. This was for convenience as it is entirely portable between the different locations and computers used for the testing. The test system itself was emulated using Qemu as the generic x86 system. An emulated system was chosen, originally, to easily start and stop the system multiple times during testing. It was shown in later testing that the system could be booted on actual

hardware, however serial output was garbled, and efforts to correct this were unsuccessful.

The Qemu machine must be operated in a Linux environment, because the Windows build of Qemu cannot display the debug output from the SEL4 kernel. As no Linux machines were easily available for the experiment and the Virtualbox Ubuntu machine used for compiling was already configured for testing and included the compiled image, it was used for the Qemu test machine as well. The Qemu SEL4 hardware is therefore emulated in the Ubuntu virtual machine. Meaning, for the tests, the program is running on an emulator in an emulator. This will affect the overall performance of the results. If the experiments are compared to each other, the performance should be affected equally and therefore the relative results will show a feasibility and performance increase or decrease even in a doubly emulated test bed.

Software

The software for this experiment includes the SEL4 microkernel, the Wombat Linux image, and custom IPC examples running on the microkernel. The code for IPC examples is included in an appendix. The Wombat images will be used for in operating system testing of IPC speeds and cell to cell IPC communication between two Wombat images.

The IPC examples will show the speed and effectiveness of IPC on the lowest level of the SEL4 kernel. Building a program directly on the kernel will

yield the best results. The IPC program will make use of multiple threads to communicate through the kernel between the memory segregated threads via IPC.

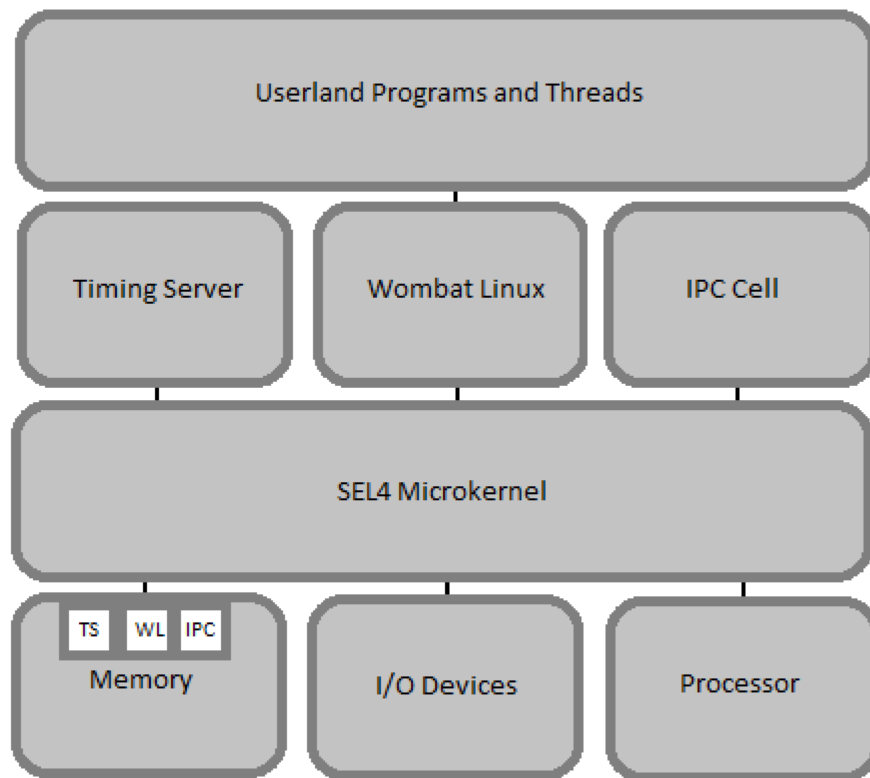

Figure 4.1: Communication in SEL4 and Memory Segmentation

The experiment will contain IPC calls from the userland built on the Wombat Linux cell. This program will run within the paravirtualized Linux environment and make IPC writing and reading calls to the kernel. The program can only communicate with the Linux API, as if it was a standard distribution, and cannot access directly any other cell or hardware. The Linux cell must then relay the information to the kernel that will write the IPC call to the appropriate

register on hardware. The IPC cell, if it wished to read this message, would contact the kernel and request an IPC message. This is the only means of communication between cells. All programs and threads in the userland are dedicated to the Linux cell and cannot communicate with or be aware of the memory space of other cells.

An IPC cell will also be designed to communicate with the Linux environment in the manner described above. This will test both the IPC performance in an operating system environment and directly on the kernel. Tests will show both performance of cell to cell, Linux to cell, and Linux to Linux IPC communication.

Security Isolation and Communication

In SEL4 memory is semi-isolated in threads and completely isolated in cells. The only communication between memory isolated processes is by IPC through the kernel API as described in Figure 4.1. For the system to be considered more secure than its predecessors, the software must take advantage of this isolation and operate within its dedicated memory space.

Software running in the user-space of the Linux cell will be considered insecure on the basis that Linux is a robust monolithic operating system, even the small version of embedded Linux used for Wombat. There are many areas of the operating system that could have memory overflows or other vulnerabilities allowing attack or control of the system. For this reason, no program operating

within the Linux cell shall have access to vital data or I/O devices unless specifically designed and allowed to control said device. In this situation, the device should be dedicated to the Linux cell and not accessible from other cells to avoid communication bypassing IPC.

All security software should be kept minimal as to not introduce bugs and operate directly on the kernel in a cell parallel to the IPC cell in Figure 4.1. See Figure 4.2 for visualization of security software. These secure programs can operate completely independent of the other cells and can control hardware, through microkernel calls of course. For example, a firewall cell might operate on the kernel and have complete control of any networked devices. The firewall would guard all incoming and outgoing communication. Other cells may use the network by IPC calls to the firewall cell. Any incoming communication would have to meet strict security protocols implemented in the firewall policy and be restricted from communication without formal authentication. This firewall cell could be referred to as a secure network manager.

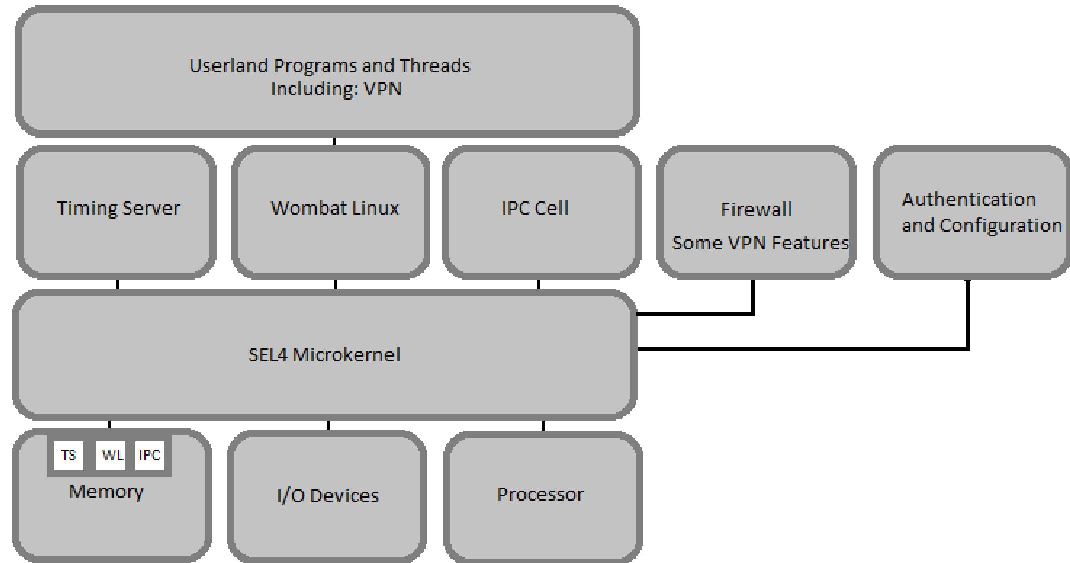

Figure 4.2: Isolated cell approach, including security cells.

Authentication and configuration of the device can also be isolated to its own secure cell. This makes the likelihood of introducing a security flaw into the software much less than running all of the security programs on the same system where they must interact with each other. Complexity of the programming can be exploited to introduce security bugs. This configuration cell will allow a user to log into the device, perhaps remotely via the firewall/limited VPN features, and configure the device. No configuration should be possible except through this cell if it is to be a secure device. Logging programs can also be implemented in the same way, as parallel cells.

A less secure VPN could be possible into the Linux environment. There would be no security risk in allowing this unless critical data or processes are implemented within the Linux environment. Any VPN access to Linux would not

be able to communicate outside of the cell. This should be done only through a dedicated networking interface or virtualized networking interface to avoid contact with other cells or hardware they are using. The VPN features in the firewall or networking manager could be programmed to allow for a virtualized device that would be able to reach Linux via IPC. This would be complex, but more secure than allowing Linux direct access to the network where other device might be less secure and on a closed network.

This experiment was conducted to test the IPC performance and usability of these secure cells. As stated above, all cells will communicate by IPC. If this system of communication is too slow or suffers from excessive use and the number of possible cells must be limited, the whole security scheme of this device may be impractical. The architecture example used above hinges on the ability to isolate every security component in its own virtual space. Without a secure, readily available, and efficient method of communication, the architecture would be useless.

CHAPTER V

Performance Evaluation

To test the security of the software, the added layers of security must first be integrated into either a cell or the Linux image. The cell programs, running directly on the microkernel, are written in the C programming language and compiled using the SEL4 library. They are then loaded into memory immediately after the microkernel. Dite is a memory mapping program used to integrate the cells into the kernel image. Dite is used to make the kernel executable from a boot loader such as Grub. Grub was used in all of the testing in this experiment to load the kernel, which then took possession of the hardware and loaded the security cells or Linux paravirtualized kernel and image. Four security programs were tested in the experiment and are described in the following sections.

Cell IPC

IPC was implemented in a separate security cell for performance testing. Running IPC straight on the kernel has added benefits to performance and shows how cell to cell communication will be handled and preform in a secure field device. The IPC program uses two threads within the same cell to communicate via IPC through the SEL4 microkernel.

The IPC test was modified from the included IPC cell example distributed with the SEL4 microkernel and was understood using the SEL4 microkernel manual explaining all of the available kernel calls in the API. Modifications were

made to include timing for performance testing. The added timing component was never completed due to the faux calls in the API. The time.h and clock.h files included with the kernel never contact the hardware clock available to the system. This created complications. The first attempt was to write code that would display the time passed between sending and receiving IPC messages in microseconds. This yielded a return of 0, which was obviously incorrect. The code was then modified to return the clock cycles past between sending and receiving. Knowing that the system was running at 100MHZ allowed the user to calculate the time passed between calls. This as well returned 0. Upon further investigation it was found that the clock and time functions existed in the API, but were set to do nothing but return 0. It appears that these functions were included for completeness so that compiling errors were not caused by lack of proper available headers, but the code in the headers was never actually connected to the hardware. This is not documented in the API, but was discovered when viewing the header files in the API itself. Without a hardware clock, cell timing performance data could not be gathered.

After completing the cell code, the code is then compiled using the standard GCC C compiler on Linux configured to cross compile for the target system. Dite then integrates the compiled program with the microkernel. The image is put onto a bootable memory stick or hard drive containing the GRUB boot loader. GRUB is instructed to boot the image on hardware, which consists of placing the microkernel and cell data into memory and passing off rights to the processor

and other hardware to the microkernel. The microkernel prepares itself for operation and starts the secure cell program. The IPC program runs and outputs the test message transmitted from one thread to another. The first thread writes a message to the IPC register, which is hardware dependent, and then the second thread reads the register through kernel calls. The message is displayed and if transmitted properly should be the same from sending to receiving. This is done three times and then the program exits.

It was found that the Qemu machine allows for the ttyS0 output data to be redirected to a telnet server. This was used to analyze the performance of the IPC calls. Since it was not feasible to time the calls directly in on the microkernel, timing data was taken in between the written string data that attempted to send the message and the string data that successfully received the IPC message. This is less accurate due to the overhead required for writing the messages, sending the data across the virtualized network interface to the telnet client, and the extra components of the IPC message program that had to run between sending and receiving messages, but is favorable to not receiving any performance data. Table 5.1 shows the results collected from the telnet log. Putty, an open source telnet client, was modified to write timestamp data to the log file allowing for timing calculations. Putty was modified for the experiment to timestamp in seconds, this was not accurate enough and Windows operating systems do not have a timer that is more accurate than milliseconds. It was attempted to accurately time to the microsecond level, but the microsecond file

timer in Windows calculates accurately, but only updates every 10 or more milliseconds rendering it useless for timing calculations. It was found that a precompiled fork of Putty named ExtraPutty timestamps correctly on the millisecond level, therefore this program was used to collect the streaming data from the microkernel and log it in text file for the experiment. Since millisecond timing was not preferable, the IPC program was again modified to complete more than three iterations between the threads and the average time was calculated for IPC messages.

Table of IPC Message Performance			
Iterations	Total Time	Time (ms)	Average Time (ms)
1,000	00:02:165	2165	2.165
10,000	00:20:647	20647	2.065
128,000	04:02:565	242565	1.895
128,000	04:03:692	243698	1.904
500,000	17:29:266	1049266	2.099
1,000,000	34:11:824	2051824	2.052

Table 5.1 – IPC Message Performance

As shown in Table 5.1, the average time to send and receive an IPC message is between 1.895 ms and 2.165 ms. The performance can be increased if timing took into account only the sending and receiving of a message. However, this data, including the overhead for timing and the running IPC program, shows that the performance on a 100MHz field device is not too

constrained to obstruct the cells from operating in a useful manner and finding an alternative means of speedier communication between cells is unnecessary.

The IPC cell operates as expected. The messages transmitted are identical to the messages received. However, the actual speed performance data was not gathered in the virtualized machine due to microkernel constraints; we must rely on the external timing data. This proves that it is possible to write a secure cell and implement it on the microkernel, that IPC functions do work as expected, and that other security software should be written from scratch and not rely on the API if at all possible as there may be other unexpected functions that are unavailable.

Linux VPN Server

Other security features are implemented inside the Linux image to be run on a higher level operating system. A VPN program was compiled and installed within Linux. For the purposes of this experiment it is unnecessary to be able to access the VPN from the actual network, it was compiled and configured to allow access only from the loopback networking device. This security addition shows it is possible to add LEMNOS/OPSAID suggested programs within the embedded Linux installation making access to the device easier and more secure.

The VPN server program, PeerVPN 0.023, is open source and distributed in C. It was compiled on the Ubuntu testing machine using the standard GCC compiler configured to cross compile for the target architecture. A configuration

file is also necessary for the program to run successfully. This is configured from the test machine. The program and config file are then entered into the static Linux image and can be run from the Linux environment.

The Linux image used in this experiment contains a basic root file system structured as a standard Linux operating system, known as "/". Programs that need to accessed by Linux must be located in this file system. It was decided that any added programs would be put in the folder located at "/bin" which is already included in the root users "path" so that it can be called from the command line without added navigation through the file system. Almost all of the commands available to the root user are located in the "/bin" directory, or included as a symbolic link if they are elsewhere located. This is standard practice in Linux environment configuration.

To add an item to the "/bin" directory in the already preconfigured image requires one of two processes; either the entire image is recreated for the target architecture using a program called Bitbake for embedded systems and testing, or the image must be unpacked, modified, and repacked in a format that Linux will recognize. The first process was initially tested, however several of the source servers for Bitbake (of which there are 165) were consistently unavailable. This made compiling Bitbake for testing impossible, causing this process to be abandoned for the second option of unpacking and manually modifying the image.

The image is packed in a CPIO image. Using the CPIO tool in Linux, it is possible to unpack the image into a directory. The compiled program and configuration file can then be moved to the "~/image/bin" directory if the image directory is located in the users home directory. It is necessary to copy the VPN program with root privileges, as all of the image directory will have root only access rights and a normal user will be unable to add programs. This can be done from the command line by tediously typing the directories out or from a GUI program if the test machine has a program file exploring program with root access. The Ubuntu test machine does not have such a program, but if the user is comfortable on the command line it should not be a problem. Almost everything, including programming, in this experiment was done from the command line. After the program is successfully copied to the images binary directory, the image can be repacked. Repacking is slightly more difficult than unpacking, it must be done with root access rights, find all files recursively in all subdirectories, and be put into "newc" format. Using the MAN pages of the CPIO command will instruct the user how to pack into the "newc" format required for a Linux image. The format for images was not documented anywhere for SEL4, but was discovered later when images refused to boot. The provided static image for use with the Wombat Linux kernel was received with no documentation or explanation for modification.

With a successfully modified file system image, the system can be instructed to MAKE the wombat supervisor, timing server, and use Dite to

package them both along with the microkernel and the file system image just created. The location of the image is specified in the make file for the whole wombat image. The microkernel image will be made and can be booted similarly to the IPC image in the previous section, by placing it in memory accessible from the GRUB boot loader.

Once the microkernel has finished setup and boots the paravirtulized Linux kernel, the Linux kernel will find the root file system and leave the user at a login prompt. The only user in the Wombat kernel is “root”. Once logged in, the user is left at a standard BASH command prompt. The “PeerSVN” program can then be accessed by calling it directly from the command line. It will start and leave the server waiting for a connection on the loopback Ethernet device.

Linux VPN Client

Running a VPN server on the field device is useful if it is a primary SCADA device, however if it is used for collection and sensors it may have to respond to an outside server for instruction and reporting. In this case, it would be necessary for the device to contain a VPN client to connect to the main servers VPN server to create a secure connection.

Similar to the previous section, an open source VPN client package was cross compiled for the target device and configured. It was then placed in the file system image exactly as the VPN server was implemented. This yielded successful results proving that both client and server programs may be ran on

the device as required. A successful VPN connection was not tested for the experiment, but the client ran and reported correctly. It is believed that the client is fully functional, but more testing is required to make certain that it was implemented correctly and can be used to create a secure connection from the device to an OPSAID server.

Linux IPC Program

An IPC program was written to be operated from within the Linux environment, in hopes of testing cell to cell communication between two Linux Wombat kernels operating in separate cells. The SEL4 API library was completely modified to allow it to compile a standard Linux application. The modification of the SEL4 library was successful from a compilation standpoint, however the microkernel has denied access to many of these functions from higher operating systems. The software was written similar to the IPC cell program, it would send an IPC message to the register specified for the target architecture and then try to read the message.

The program was successfully compiled and placed into the Linux image via the same process listed in the previous two sections. The system was then booted and the program was tested. The timing functions, being that the program is now in an environment with proper time and clock functions using the Linux timing server, now work as expected. The IPC message is sent, but the microkernel responds with an message interpreted as an access denied error and the read function finds a 0 instead of the intended message. It is believed that it

is impossible to access microkernel functions from within Linux, at least not by the method described above. Little help could be found on the subject and the Linux IPC cell to cell experiment was abandoned. The code, along with all other custom code for the experiment can be found in the appendix.

CHAPTER VI

Conclusion

This thesis demonstrated the design, implementation, and testing of a secure field device based on a verified microkernel. The SEL4 microkernel and software can be implemented on real hardware and further tested using the previous chapters as a guide for set-up and software configuration. It is hoped that this research and experimentation has built the foundation for a more secure device.

Summary

The virtual prototype presented in this thesis has shown that a secure, memory isolated field device can be implemented. Using open source software, the device can be affordable and configured for any data collection or control device. The OPSAID system requirements can be met or exceeded without any additional hardware. The x86 hardware architecture used in the prototype could be ported to ARM without much difficulty using similar methods to the building and compiling of the current x86 system.

The security software implemented in the current design shows that nearly any necessary software can be implemented as well. If the secure rules outlined in Chapter II for communication are followed most software could be ported to the device. It was shown that programs can be compiled for either the higher Linux operating system or to run directly on the microkernel for an efficient and

secure program with its own isolated memory space. Programs were created for and tested in both the Linux environment and directly on the microkernel.

It was observed that if the security program required any outside libraries, or was not fairly straight forward to implement, that it was much less labor intensive to create the program for the Linux environment. The microkernel libraries are limited and incomplete in some places such as access to any sort of clock or timer. Programs created to run in their own cell directly on the microkernel should have a specific intended purpose and perform that purpose in the simplest possible way to avoid creating security bugs. Only security critical processes, or those that require direct hardware access, should be implemented in their own cells. A program or process that is convoluted and has additional, unnecessary features would be best placed in the Linux environment. The Linux system offers a full-fledged operating system API for a virtualized set of hardware and interaction with the operating system. This allows for less challenging programming and cross compilation.

The programs tested performed well for their intended functions. The two VPN programs compiled for the Linux environment show that a secure device can be used as both a client and a server device; this shows the flexibility of the security software. Most standard Linux applications could be compiled for the device. The VPN software was standard open source software available freely on the Internet for use in any Linux system, opening the door for any security application available to be implemented in the device. The microkernel

application was a modified version of the multi-threaded IPC example included with the SEL4 microkernel. Until it become a standard architecture, it is unlikely that an implementation of this device would be able to make use of off-the-shelf software for single purpose secure cells. These cells would have to be custom created for the purposes of the device, but it was shown that the software can function in a private cell. The limitations of these security cells are determined by the hardware, the microkernel API, and the creativity of the programmer.

Future Research

There is a great need for a secure field device. This experiment and the prior work is a great start, but there is still much to be done. The road to an effective security device is not a short one. With small strides, each contributing researcher is moving the field ahead, with the goal being a completely secured, inexpensive field device with interchangeable security programs and a standardized design.

If this experiment were to continue there are a few things that should still be tested. The most important next step is implementing the device in real hardware. The virtualized environment worked great for the experiment, but it would be a leap forward to actually create the device in a useable state. A generic x86 computer was used for hardware testing, but the output was garbled for some reason. This was not a problem on the virtual console. For the design presented in the previous chapters to be useful it would have to be shown to work on hardware.

Further research into the current design of this device should also test cell to cell IPC messages. The prototype described in chapter V showed IPC communication in a single cell between multiple threads. This is a great start for microkernel secure communication, but is not of much use without continuing experimentation. For IPC to be useful, it needs to be used to communicate between memory isolated processes as it was intended. Preferably, the device would communicate between two dedicated security cells that had a purpose other than testing the IPC; for instance, the firewall cell would successfully send messages to and authentication cell.

It was not determined whether it was possible for the Linux environment to communicate with outside cells using IPC. If this is possible, two wombat Linux cells should be created side by side as detailed in chapter V, but have an additional IPC application to communicate with one another. It would be beneficial to the device to have multiple Linux environments for the programs that require Linux libraries to still be isolated, but be able to communicate with each other and all of the other secure cells directly on the microkernel. Two wombat images were designed and tested during the experiment, but the IPC communication could not be shown to function correctly inside the Linux environment.

The software used in the experiment should be ported to the ARM architecture. It seems common for SCADA devices to use ARM processors for the price and power consumption. The SEL4 microkernel is available on both ARM

and x86. It would be beneficial to test all of the software on both systems to reach the greatest possible audience for the secure device. This should not be a difficult task, but was not included in the experiment due to time constraints.

Finally, the device should be thoroughly tested for vulnerabilities. Unless all of the software on the device is tested, the device cannot truly be considered secure. It should be reasonable to assume that any intrusion into a single cell should not compromise the other software, but that is no reason not to test each component individually.

References

[1] Rautmare, S.; , "SCADA system security: Challenges and recommendations," *India Conference (INDICON), 2011 Annual IEEE* , vol., no., pp.1-4, 16-18 Dec. 2011

[2] V. Igure, "Taxonomies of attacks and vulnerabilities in computer systems," *Communications Surveys & Tutorials,*, pp. 6-19, 2008.

[3] Langner, R.; , "Stuxnet: Dissecting a Cyberwarfare Weapon," *Security & Privacy, IEEE* , vol.9, no.3, pp.49-51, May-June 2011.

[4] S. A. Hurd, J. E. Stamp, and A. R. Chavez, "OPSAID Initial Design and Testing Report," *Department of Energy*, 2007.

[5] B. P. Smith, J. Stewart, R. Halbgewachs, A. Chavez, R. Smith, and D. Teumim, "Cyber Security Interoperability - The Lemnos Project," *53rd Annual ISA Power Industry Division Symposium*, 2010.

[6] R. I. Mutia, "Evaultaion and Analysis of OKL4-Based Android," *Lund University*, 2009.

[7] W. Chengjun, "Research on the Microkernel Technology," in *2009 Second International Workshop on Computer Science and Engineering*, 2009, pp. 199-202.

[8] J. L. Hieb and J. H. Graham, "Designing Security-Hardened Microkernels For Field Devices," in *Critical Infrastructure Protection II*, M. Papa and S. Shenoi, Eds. Boston, MA: Springer, 2009, pp. 129-140.

[9] J. H. Hieb, J. H. Graham, and B. Luyster, "A Prototype Security Hardened Field Device for Industrial Control Systems," in *Proceedings of the International Conference on Advanced Computing and Communications*, 2010, pp. 95-100.

[10] J. Liedtke, "Improving IPC by kernel design," *ACM SIGOPS Operating Systems Review*, vol. 27, no. 5, pp. 175-188, Dec. 1993.

[11] OK Labs, "OKL4 Microkernel Reference Manual," no. 12, 2008.

[12] G. Klein, "The L4 . verified Project — Next Steps," *Verified Software: Theories, Tools, Experiments*, 2010.

Appendices

The appendices section contains the build configurations for the SEL4 microkernel system and any custom code developed and used during the experiment. Any code not included was not used to collect data, not original to this thesis, or a derivation of included code too similar to require a separate attachment.

Build Configurations

Config.mk

```
# This file contains build configuration variables

# Architecture and platform to build for
export ARCH?=ia32
export PLAT?=pc99
export CFLAGS=-fno-stack-protector
#export ARCH?=arm
#export PLAT?=imx31

# Comment out the line below to build a non-debug kernel and userland
export SEL4_DEBUG_KERNEL=1

# Compile in IOMMU functions
export IOMMU=1

# Tell the build where the toolchain is
ifeq ($(ARCH),arm)
export TOOLPREFIX=arm-oe-linux-gnueabi-
export TOOLSUFFIX=
CROSSBINPATH=opt/arm-2010.09/bin
DITEPATH=${PWD}/../tools/dite/build
else
export TOOLPREFIX=i386-unknown-elf-
export TOOLSUFFIX=
CROSSBINPATH=/home/kevin/x-tools/i386-unknown-elf/bin
DITEPATH=${PWD}/../tools/dite/build
endif

# Sanity check the toolchain to ensure it really does exist.
ifeq($(strip$(wildcard${CROSSBINPATH}/${TOOLPREFIX}gcc${TOOLSUFFIX})),)
$(error "Could not find your toolchain. Please check your 'config.mk'.")
```

```
endif

# Capability dumps (used by the CapDL Extrator tool) go out on this second
# serial port.
export SEL4_CMDLINE="debug=0x2f8"

# Tell the build where Wombat's prebuilt root filesystem image (in "cpio"
# format) is located. If a root filesystem image is not provided, wombat
# will panic on boot. A full path must be specified.
ROOTFS=${PWD}/misc/image.cpio

# The Linux config file. These are found in source/wombat and are named
# similarly.
LINUX_CONFIG=sel4linux_config_ia32

# The supervisor config file. There are found in:
#       source/wombat-supervisor/include/wombat-supervisor/configs/
SUPERVISOR_CONFIG=system_one_wombat
```

system_two_wombats.h

```
#ifndef _SYSTEM_TWO_WOMBATS_H_
#define _SYSTEM_TWO_WOMBATS_H_

void assemble_system(void)
{
   enum device_name wombat0_devices[] = {
      HARDWARE_NIC_0, HARDWARE_CONSOLE, 0};
   enum device_name wombat1_devices[] = {
      HARDWARE_NIC_1, HARDWARE_CONSOLE_NO_IRQ, 0};

   /* Setup a timer server, and two wombats. */
   struct component *timer = register_timer_server_component(
         DEFAULT_TIMER_SERVER_PRIO);
   struct component *wombat0 = register_wombat_component(
         0, "rdinit=/sbin/init wombat0", wombat0_devices, 200,
         DEFAULT_WOMBAT_PRIO);
   struct component *wombat1 = register_wombat_component(
         1, "rdinit=/sbin/init wombat1", wombat1_devices, 200,
         249);

   /* Connect the timer server to the wombats. */
   SYSTEM_CONNECTIONS[0] = (connection_t)
      {timer,      wombat0,      VIRTUAL_IRQ,      {.irq=0},      seL4_CanWrite,
seL4_AllRights};
```

```
    SYSTEM_CONNECTIONS[1] = (connection_t)
      {timer, wombat1, VIRTUAL_IRQ, {.irq=0}, seL4_CanWrite,
seL4_AllRights};
};

#endif /* _SYSTEM_TWO_WOMBATS_H_ */
```

Custom Code

main.c (for IPC Cell)

```
#include <stdio.h>
#include <sel4/sel4.h>
#include <sel4/bootinfo.h>
#include <assert.h>

#include <sel4/sel4.h>
#include <sel4/arch/syscalls.h>

#include <iwana/interrupts.h>
#include <iwana/boot_data.h>
#include <iwana/timer_server.h>

#define STACK_SIZE (1 << seL4_PageBits)
static seL4_CPtr ipc_endpoint = 0;

#define MASK(x) ((1<<(x))-1)

//The new thread will begin executing this function
static void my_other_thread(void) {
  printf("\nHello World, this is \"%s\"\n",__FUNCTION__);

  //Create a message tag that specifies that the first message
  //register should be transferred when an IPC message is sent
  seL4_MessageInfo tag = { {.length = 1} };
  seL4_Word mr0 = 0;

  //Loop forever calling the endpoint
  while(1){
    printf("%s: Sent message %d of length %d to endpoint %p.\n\n",
      __FUNCTION__,
      mr0,
      tag.length,
      (void *)ipc_endpoint);
```

```
    //Set the contents of the first message register.
    seL4_SetMR(0,mr0);
    //Make the call
    tag = seL4_Call(ipc_endpoint,tag);
    //Get the contents of the first message register. This was
    //transferred from the thread that replied to the call.
    mr0 = seL4_GetMR(0);

    printf("%s: Received message %d of length %d from endpoint %p.\n",
      __FUNCTION__,
      mr0,
      tag.length,
      (void*)ipc_endpoint);

    mr0++;
  }
}


int main(void) {

  //Get a pointer to the bootinfo structure from libsel4
  seL4_BootInfo* info = seL4_GetBootInfo();
  unsigned int i;

  printf("\n IPC Test\n\n");

  //Find the first free slot in the CSpace
  printf("Finding the first free slot in the CSpace...");
  seL4_CPtr free_slot = 0;
  for (i = 0; i < info->regionCount; i++) {
    if(info->regions[i].type == seL4_Region_FreeSlots){
      free_slot = info->regions[i].base;
      printf("found at %p.\n", (void *)free_slot);
      break;
    }
  }
  assert(i != info->regionCount);

  //Find the first empty region in the CSpace
  printf("Finding the first free empty in the CSpace...");
  seL4_CPtr empty_slot = 0;
  for (i = 0; i < info->regionCount; i++) {
```

```
  if(info->regions[i].type == seL4_Region_Empty){
    empty_slot = info->regions[i].base;
    printf("found at %p.\n", (void *)empty_slot);
    break;
  }
}
assert(i != info->regionCount);
seL4_Word vaddr = empty_slot;

//Find the first small block cap
printf("Finding the first Small Block (4K Untyped Capability) ...");
seL4_CPtr four_k_untyped = 0;
for (i = 0; i < info->regionCount; i++) {
  if(info->regions[i].type == seL4_Region_SmallBlocks){
    four_k_untyped = info->regions[i].base;
  printf("found at %p.\n",(void *)four_k_untyped);
  break;
  }
}
assert(i != info->regionCount);

//Retype the a small block to a TCB
printf("Retyping small block to a TCB...");
seL4_Untyped_Retype_t rresult = seL4_Untyped_Retype(
  four_k_untyped,
  seL4_TCBObject,
  0,
  seL4_SelfCSpace,
  free_slot >> seL4_PageBits,
  seL4_WordBits - seL4_PageBits,
  free_slot & MASK(seL4_PageBits),
  1);
printf("created %d cap(s) at %p.\n",rresult.result,(void*)free_slot);
seL4_CPtr thread_TCB = free_slot;
assert(!rresult.error);

//Go to the next small block and the next free slot
four_k_untyped++;
free_slot++;

//Retype the small block into an endpoint object
printf("Retyping small block to an endpoint object...");
rresult = seL4_Untyped_Retype(
  four_k_untyped,
```

```
    seL4_EndpointObject,
    0,
    seL4_SelfCSpace,
    free_slot >> seL4_PageBits,
    seL4_WordBits - seL4_PageBits,
    free_slot & MASK(seL4_PageBits),
    1);
  assert(!rresult.error);
  printf("created %d cap(s) at %p.\n",rresult.result,(void*)free_slot);
  ipc_endpoint = free_slot;

  //Go to the next small block and the next free slot
  four_k_untyped++;
  free_slot++;

  //Retype the small block into a 4K frame for the IPC buffer
  printf("Retyping small block to an 4K frame...");
  rresult = seL4_Untyped_Retype(
    four_k_untyped,
#ifdef IA32
    seL4_IA32_4K,
#else
    seL4_ARM_SmallPageObject,
#endif
    0,
    seL4_SelfCSpace,
    free_slot >> seL4_PageBits,
    seL4_WordBits - seL4_PageBits,
    free_slot & MASK(seL4_PageBits),
    1);
  assert(!rresult.error);
  printf("created %d cap(s) at %p.\n",rresult.result,(void*)free_slot);
  seL4_Word four_k = free_slot;

  printf("Mapping 4K frame (%p) to free vadd (%p).\n", (void *)four_k,
(void*)empty_slot);
#ifdef IA32
  int result = seL4_IA32_Page_Map(
#else
  int result = seL4_ARM_Page_Map(
#endif
    four_k,
    seL4_SelfVSpace,
    empty_slot,
```

```
    seL4_AllRights,
#ifdef IA32
    seL4_IA32_Default_VMAttributes);
#else
    seL4_ARM_Default_VMAttributes);
#endif
  assert(!result);

  //Set up new thread's IPC buffer
  printf("Setting up IPC buffer on new thread...");
  result = seL4_TCB_SetIPCBuffer(
    thread_TCB,
    vaddr,
    four_k);
  assert(!result);

  //Set up the VSpace and CSpace on the new thread
  printf("Setting TCB CSpace and VSpace..");
  result = seL4_TCB_SetSpace(
    thread_TCB,
    0,
    seL4_SelfCSpace,
    seL4_NilData,
    seL4_SelfVSpace,
    seL4_NilData);
  assert(!result);


  //Write the registers of the new thread.
  //This sets a new thread running at the
  //default priority
  printf("Starting up new thread...");
  static char stack[STACK_SIZE];
#ifdef IA32
  seL4_UserContext frame = {.regs = {.eip = (unsigned int)my_other_thread,
.esp = (unsigned int)&stack[STACK_SIZE] }};
#else
  seL4_UserContext frame = {.regs = {.pc = (unsigned int)my_other_thread, .sp
= (unsigned int)&stack[STACK_SIZE] }};
#endif
  result = seL4_TCB_WriteRegisters(
    thread_TCB,
    true,
    0,
```

```
    sizeof(seL4_UserContext) / sizeof(seL4_Word),
    &frame);
  assert(!result);

  //Wait for someone to send us an IPC
  seL4_Word sender_badge = 0;
  printf("%s: Waiting on IPC...\n",__FUNCTION__);
  seL4_MessageInfo tag = seL4_Wait( ipc_endpoint, &sender_badge);
  seL4_Word mr0 = seL4_GetMR(0);
  printf("%s: Recv'd message %d of length %d from endpoint %p.\n",
    __FUNCTION__,
    mr0,
    tag.length,
    (void *)ipc_endpoint);

  //Repeat the cycle three times
  for(i = 0; i < 250000; i++){
    //Reply to the IPC and wait for another
    seL4_SetMR(0,++mr0);
    printf("%s: Sent message %d of length %d to endpoint %p.\n\n",
      __FUNCTION__,
      mr0,
      tag.length,
      (void *)ipc_endpoint);

    tag = seL4_ReplyWait(ipc_endpoint,tag, &sender_badge);
    mr0 = seL4_GetMR(0);

    printf("%s: Recv'd message %d of length %d from endpoint %p.\n",
      __FUNCTION__,
      mr0,
      tag.length,
      (void *)ipc_endpoint);
  }

  printf("\nDone.\n\n");
  return 0;
}
```

ipc.c (IPC inside Linux)

```
#include <stdio.h>
#include <time.h>
```

```
#include "sel4/sel4.h"
#include "sel4/bootinfo.h"
#include <assert.h>

int main(void)
{
  time_t now;
  time(&now);

  printf("%s", ctime(&now));

 printf("\nHello World, this is \"%s\"\n","testipc");

  //Create a message tag that specifies that the first message
  //register should be transferred when an IPC message is sent
  seL4_MessageInfo tag = { {.length = 1} };
  seL4_Word mr0 = 129;

    printf("%s: Sent message %d of length %d to endpoint %p.\n\n",
      "test2",
      mr0,
      tag.length,
      (void *)0x0000006d);

    //Set the contents of the first message register.
    seL4_SetMR(0,mr0);
    //Make the call
    tag = seL4_Call(0,tag);
    //Get the contents of the first message register. This was
    //transferred from the thread that replied to the call.
    mr0 = seL4_GetMR(0);

    printf("%s: Received message %d of length %d from endpoint %p.\n",
      "test3",
      mr0,
      tag.length,
      (void*)0x0000006d);

  return 0;
}
```