7-2006

Evaluation of MILS and reduced kernel security concepts for SCADA remote terminal units.

Brent L. Guffey 1982-
University of Louisville

EVALUATION OF MILS AND REDUCED KERNEL SECURITY CONCEPTS FOR SCADA REMOTE TERMINAL UNITS

By
Brent L.Guffey
B.S., University of Louisville, 2005

A Thesis
Submitted to the Faculty of the
University of Louisville
Speed School of Engineering
in Partial Fulfillment of the Requirements
for the Professional Degree

MASTER OF ENGINEERING

Department of Computer Engineering and Computer Science
University of Louisville

July 2006

EVALUATION OF MILS AND REDUCED KERNEL SECURITY CONCEPTS FOR SCADA REMOTE TERMINAL UNITS

By
Brent L.Guffey
B.S., University of Louisville, 2005

A Thesis Approved on

_____

July 2006

by the Following Reading and Examination Committee:

_____
Dr. James H. Graham, Thesis Director

_____
Dr. Dar-Jen Chang

_____
Dr. Patricia Ralston

ACKNOWLEDGMENTS

The author would like to thank Dr. James H. Graham for his guidance and support throughout the life of this project, including the initial ideas that got it started. The author would also like to thank Dr. Dar-Jen Chang and Dr. Patricia Ralston for serving on the thesis reading and examination committee.

The author would also like to greatly thank Jeff Hieb, as without his insight and support this project would not have been possible. Mr. Hieb is currently conducting research on a related project and provided vital code that comprised the RTU used in this project, along with an MTU application to connect and interact with the RTU.

LynxOS is a registered trademark of LynuxWorks. The author wishes to acknowledge the contributions of the open source microkernel research community for providing freely available implementations of the L4 microkernel, such as the Pistachio kernel utilized for this project created by the System Architecture Group at the University of Karlsruhe.

ABSTRACT

The purpose of this project is to study the benefits that the Multiple Independent Levels of Security (MILS) approach can provide to Supervisory Control and Data Acquisition (SCADA) remote terminal units. This is accomplished through a heavy focus on MILS concepts such as resource separation, verification, and kernel minimization and reduction. Two architectures are leveraged to study the application of reduced kernel concepts for a remote terminal unit (RTU). The first is the LynxOS embedded operating system, which is used to create a bootable image of a working RTU. The second is the Pistachio microkernel, the features and development environment of which are analyzed and catalogued to provide the basis for a future RTU.

A survey of recent literature is included that focuses on the state of SCADA security, the MILS standard, and microkernel research. The design methodology for a MILS compliant RTU is outlined, including a benefit analysis of applying MILS in an industrial network setting. Also included are analyses of the concepts of MILS which are relevant to the design and how LynxOS and Pistachio can be used to study some of these concepts. A section detailing the prototyping of RTUs on LynxOS and Pistachio is also included, followed by an initial security and performance analysis for both systems.

LIST OF TABLES

LIST OF FIGURES

Table of Contents

CHAPTER I: Introduction

1.1 Background Information

This project falls under the umbrella of a larger SCADA research project at the University of Louisville. The focus of that project is to mitigate the effects of threats caused by electronic attacks on SCADA systems and has risen from ongoing research at the University of Louisville that has centered around both SCADA security and improving the monitoring process for chemical process control systems. The project is in collaboration with industrial groups such as Hexion Specialty Chemicals, Eon-US Energy Corp., and the Louisville Water Company. It is funded as part of the Kentucky Critical Infrastructure Project.

1.2 Problem Statement

This project aims to produce a framework for a more secure SCADA RTU (remote telemetry unit or remote terminal unit). At the outset of the project, the focus was on applying the Multiple Independent Levels of Security (MILS) standard to these RTUs. However, those plans fell by the wayside as it became apparent that obtaining an acceptable MILS specific operation system or kernel would not be possible within this project's time frame. As full MILS compliance is not currently available, this project will instead focus on analyzing the security benefits that MILS could provide RTUs and

provide an implementation of a hardened RTU which will exhibit some of these security related features.

One such attempt was made by creating a minimal kernel using features provided by LynxOS, a real time embedded operating system. This operating system provides the means to modify the kernel and custom build bootable images to suit the users needs. This allowed the creation of a minimal bootable system, providing only the functionality needed for the RTU and nothing extraneous.

The second attempt utilized the concept of a predefined microkernel to run the RTU software. This project uses the Pistachio microkernel created by the L4Ka group at the University of Karlsruhe, and is based on the L4 microkernel specification. The reasons each of these alternate approaches was pursued will be explained in detail in later sections of this document.

1.3 Motivation

SCADA systems are used in critical infrastructure. Increasingly, communication between the central SCADA master system and the RTUs is occurring over the Internet or other network connections. The exposure of these RTUs to the network opens them up to the problems all other networked computers face: attacks from malicious or malignant entities that desire and attempt to harm the system. Network based attacks such as intrusion, denial of service, and worms are all serious risks that must be considered when connecting critical infrastructure equipment to the Internet.

SCADA systems use the DNP-3 protocol for communication between the master unit and the RTUs. DNP-3 provides data fragmentation, error checking, and other

features and operates between the physical layer and the networking layer of the network. Higher level networking functions are increasingly transmitted using TCP/IP, tying in to SCADA's increasing reliance on the Internet. There has been research that falls under the same umbrella as the parent project of this MILS project, namely the overlying SCADA research at the University of Louisville, which has focused on enhancing the DNP-3 protocols [36]. However, even if these protocols are perfect and perfectly secure, as long as communication occurs over the Internet between the RTU and master unit, there will be a risk of attacks such as denial of service (DOS) that can bypass DNP-3.

Since it is inevitable that many SCADA systems will utilize the Internet for its familiarity, ease of use, and wealth of available tools and documentation, a researcher into security for SCADA systems must focus on ways to make the entire communication process more secure. The tasks that arise from this need can range from basic networking security principles such as setting up routers and disabling remote logon, all the way up to the overall security of the underlying operating system. The latter is the focus of this project, which began as a way to research the benefits that the MILS standard would supply to the operating system of the RTU.

The MILS standard provides a verifiably secure kernel, as well as many other features which, upon inspection, provide security synergy with the functional aspects of SCADA RTUs. The most important feature to provide this type of synergy is partitioning. MILS defines a method of brick-wall partitioning of memory, kernel access, and other system resources. "Brick-wall" means the operations of one partition cannot affect another partition, meaning any errors occurring in, for example, a dedicated

networking partition could not cascade and affect monitoring applications in another partition.

A verifiably secure kernel can also provide a huge enhancement to the confidence in security of the overall system. Mathematical verification of most kernels is impossible, given the size of the code base of many kernels can approach millions of lines of code. The MILS kernel should provide all base functionality needed for system operations, but will be small enough for mathematical verification. This principle removes all extraneous functionality from the kernel, removing with it the problems of complexity, bloat, and potential security flaws in extraneous methods or code.

These two elements of the MILS standard, namely resource partitioning and a minimal, verifiable kernel, are what lead to this project's focus on applying the standard to SCADA RTUs. If any benefits can be gleaned from such a pairing, infrastructure monitor and control systems would be safer even with communications occurring over the Internet.

1.4 Organization

Chapter Two focuses on the literature available covering SCADA security, the MILS architecture, and microkernel design and security. Chapter Three outlines the concepts of designing a hardened RTU, explain why the MILS standard is appropriate for such devices, and detail the ideas which produced the two related, but ultimately non-MILS prototypes created during this project. Chapter Four discusses the details and implementation of these prototype RTUs, one created using the LynxOS kernel image creation tools and the basis of another utilizing the L4 Pistachio microkernel. Chapter

Five provides analysis and applies metric measurements and performance evaluation on each of the two base systems, addressing security and performance related questions. Chapter Six discusses any conclusions determined through the course of this project, as well as future directions for hardened RTU research, including a fully MILS compliant RTU and what further benefits that might bring.

CHAPTER II: Literature Review

2.1 SCADA Systems

2.1.1 Overview

Supervisory Control and Data Acquisition (SCADA) systems have been used for years by industry and infrastructure providers such as power and water plants as a means of controlling distributed systems from a master location [1]. The typical SCADA system consists of a centralized master control unit (sometimes called an MTU, or master terminal unit) and distributed Remote Terminal (or Telemetry) Units (RTUs) that control and monitor physical equipment and machines such as pumps, latches, and other control machines. SCADA master units typically run on licensed operating systems such as Unix and variants, with Microsoft Windows also becoming more widely used. These operating systems provide the human users with an interface to the SCADA system in order to monitor and control the overall distributed system from the master unit.

2.1.2 SCADA RTUs

The RTUs provide the distributed backbone of a SCADA system. RTUs are often hardened computers with some type of connection back to the master via either a serial port, on board modem [2], or, increasingly, via the Internet or a Local Area Network (LAN). The functions of the RTU include acting as an arbiter for control functions at a location remote to the master unit, and monitoring and collecting data from the local site.

There are two types of RTUs: ‘single board’ and ‘modular’. A single board RTU contains all chips and circuitry necessary for I/O and processing on a single board while a modular RTU has separate modules for CPU, memory, I/O, and other functions [2]. This project will focus on using a hardened PC to act as an RTU, implying that a modular RTU approach will be required.

RTUs collect data called points to be sent back to the master unit for storage inside a database. Each point represents a monitored input or output controlled by the overall SCADA system. A so-called “hard point” is data that is directly input or output by the system, while a “soft point” is data derived from mathematical operations, usually carried out at the RTU. [1] Complex SCADA systems can have as many as 30,000 to 50,000 points at once, indicating the necessity of computers for monitoring and control purposes. Graphs of the status of certain points are often created to improve understanding of system states by human administrators and users. [3] The volume of data also suggests the need for high bandwidth communication from the distributed RTUs back to the master control unit.

2.1.3 Recent SCADA Security Breaches

SCADA security is currently a hot-button issue, and many organizations both industrial and governmental have studied methods for improvement. First of all, a number of articles have been published in relation to security breaches in plants providing power and other infrastructure needs. Brown’s article “SCADA vs. the hackers” [4] provides insight into a number of attacks that would be possible, including

wireless leaks and overall OS security breaches. However, there are more concrete examples of breaches in SCADA security.

In 2003, the Slammer worm impacted the network of the Davis-Besse Nuclear Power Plant in Ohio. The worm infected the internal network of the plant via an unpatched vulnerability in MS-SQL and caused slowdown of the plant network, ultimately leading to the shutdown of the plants safety monitoring system. [16] In a separate incident in 2001, hackers attempted, albeit unsuccessfully, to breach the network of CAL-ISO, the company overseeing much of California's power grid [17]. These incidents paint a disturbing picture of the need for a security focus for SCADA systems.

2.1.4 Current SCADA Security Research

Many groups have conducted research and published findings on SCADA security in recent years. For example, a paper presented to the 2005 IEEE Systems, Man and Cybernetics Information Assurance Workshop outlines best practices for next generation SCADA security systems. Topics include the use of demilitarized zones (DMZ) between the SCADA system and the Internet, as well as combating denial of service (DoS) attacks by using a modified TCP protocol for transport level communication. [18]

One article that underlines the emerging trend to integrate SCADA systems with the Internet is [19], which outlines the design of a web-based SCADA system using Java and XML. This article does not focus on security, which brings forth the need for a security paradigm to possibly be separate from the implementation of the SCADA system, such as within the underlying OS and kernel.

Other works covering similar themes as these two articles include [20], [21] and [22], all articles from the past two years dealing with security, both long- and short-term, for SCADA systems. There are many other such articles in, for example, the IEEE Xplore article database, indicating the need and desire from the research community to find better solutions to SCADA security quandaries.

2.2 SCADA and the Internet

2.2.1 Usage

The Internet has become a crucial component of most SCADA systems today. In fact, it has been stated that, "…it is almost impossible today to buy remote terminal units or control systems that are not Web- or network-enabled." [4] The benefits that the Internet provides to SCADA systems are obvious. First of all, the protocols used in Internet communication are commonly known, as well as easy to implement and use. This allows operators setting up SCADA distributed networks to focus on the functions of the separate components and not how they communicate. It also provided the means for reliable high-bandwidth data transfer that is essential for large, critical SCADA systems.

2.2.2 Risks

However, placing SCADA communication over the public Internet has opened up the potential for many security risks. The possibility of intruders into a SCADA system cannot be ignored. Indeed, one security survey of a nuclear power plant which used

modems for dial-in access revealed several unregistered modems had accessed the network [4]. Intercepted messages can also be used to gain information about system operation, and masquerade attacks can be used to plant false commands into an unsecured communication channel [4]. Basically, any security problem that afflicts the Internet can be applied to SCADA networks using this resource. RTUs can be taken over, data can be intercepted, intruders can determine network topography and weak points, and so on.

The use of public standards such as Ethernet and TCP/IP in SCADA systems has made the possibility of attack more likely. Studies have found that more than one third of external security incidents use the Internet as the remote point of entry [30]. The same study found a dramatic increase in the number of security incidents starting in 2001, with an ever increasing percentage of such incidents being external in nature [30]. The authors attribute this increase in external incidents to three factors. The first is the rise, beginning with Code Red in 2001, of the automated worm attack. Such attacks likely don't target SCADA systems specifically, but are still a major risk to system stability if infection were to occur. Second, the use of operating systems and software designed for business requirements within the network of critical systems, exposing these systems to common IT attacks such as viruses and backdoor exploits. Third is the increasing interconnection of SCADA systems, creating interdependencies that can, without proper study and design, open up new avenues for attack. [30]

Intent by hackers to break into critical infrastructure systems has also been recently expressed. Such individuals often produce tools to aid in the breach of security procedures, enabling those who do not have the skills to directly make an attack to do so with the help of this software. Such tools, combined with an increasingly computer

literate world population and the opening of SCADA systems to public standards, make the difficult task of hacking into a SCADA system slightly easier and will no doubt increase the frequency of attempts at such attacks. [31]

2.3 The MILS Security Standard

2.3.1 Definition

The acronym MILS stand for Multiple Independent Levels (or Layers) of Security. MILS is a standard that has been designed to allow mathematical verification of the security of core systems software by separating security functionality into four security policies: Information Flow, Data Isolation, Periods Processing, and Damage Limitation [5]. These functionalities are controlled by a partitioning kernel that oversees resource and security management for the system. The concept of a separation kernel used to divide memory into separate partitions for application use was first outlined in [6] and [7] by John Rushby in the early 1980's. The four security policies are described below (paraphrased from [5] and [8]):

- Information Flow control ensures that the flow of data between partitions is authenticated end-to-end between sender and receiver. Information must arrive only where intended by the sender.
- Data Isolation ensures that each partition's data is accessible only by that partition, and that private data cannot be publicly accessed.
- Periods Processing ensures that exploits using the processor and networking hardware cannot give an intruder access to the system.

- Damage Limitation (or Fault Isolation) ensures that a failure in one partition will not affect the performance or security of another partition. All failures must be detected, contained, and recovered from locally.

By ensuring that a kernel is built from the ground up with these principles in mind, the amount of security critical code can be drastically reduced. This in turn provides the ability to apply rigorous mathematical tests and inspections to the kernel [8], something that is not possible with current operating systems whose critical features may contain millions of lines of code [9].

2.3.2 Evaluation of the MILS Kernel

One of the main objectives of MILS is for the separation kernel to be evaluated at EAL 7 [9]. The Evaluation Assurance Levels (EALs) are defined by the Common Criteria, a set of internationally defined and recognized standards for evaluation of secure software [10]. EAL 7 is the highest possible level of assurance, and states that software must be formally (mathematically) verified, designed, and tested to attain an EAL 7 rating. By using small code for the separation kernel and security functions, as well as enforcing *NEAT*ness (see following section), components used in MILS implementations can be evaluated at EAL 6+ [8].

Evaluation consists of the process of verifying and testing, either formally or informally, a given piece of software to ensure its compliance with the internationally defined EALs. The following table outlines each EAL from 1 to 7 and provides a brief description of each. Each EAL builds on the previous, so any tests performed at a lower

level of assurance will be included in higher level assurance tests. These criteria are formally defined and outlined in [10].

Table 2.1 Evaluation Assurance Levels (EAL)

EAL	Requirements for Verification
1	Functionally tested. Analyzes behavior and system documentation.
2	Structurally tested. Analyzes more detailed high-level design specifications.
3	Methodically tested and checked. More complete testing process.
4	Methodically designed, tested, and reviewed. Analyzes more detailed design specifications and an implementation subset.
5	Semi-formally designed and tested. More structured architecture and semi-formal design description required.
6	Semi-formally verified, designed, and tested. Requires improved analysis of architecture and improved design description over EAL 5.
7	Formally verified, designed, and tested. More comprehensive testing required. Formal (mathematical) proof of design and formal representations must be used to assure security.

The formal verification required to meet EAL 7 is not currently possible with today's monolithic kernels, which can contain millions of lines of source code. The MILS kernel seeks to alleviate this quandary using two methods. The first is to reduce the amount of code in the kernel by removing functionality that can be obtained by higher level applications while still enforcing the principle of NEATness. The second is to provide brick wall partitioning, allowing for different levels of assurance for applications running in separate partitions. [9] These concepts will be discussed in more detail later in this document.

2.3.3 Properties of NEATness

*NEAT*ness is essential for an operating system to ensure security. *NEAT* is an acronym for Non-Bypassable, Evaluatable, Always Invoked, and Tamperproof [8].

These properties allow a high assurance level for overall systems security. These attributes of secure operating systems are explained below (adapted from [5] and [8]):

- Non-Bypassable means security functions cannot be avoided.
- Evaluatable means that security functions can be formally verified and tested.
- Always Invoked means security functions are invoked each time they are needed.
- Tamperproof means poorly written or subversive code cannot modify security functions and security related data.

MILS is the first publicly available approach to operating system design that allows for *NEAT*ness in Commercial Off The Shelf (COTS) software and operating systems [8].

2.3.4 Partitioning Communication Systems

A Partitioning Communication System (PCS) is a middleware component of a MILS OS which has authority over all communication between MILS partitions. While those familiar with SCADA systems may recognize the acronym "PCS" to stand for process control systems, this PCS is a MILS specific concept and the two shouldn't be confused. The four main policies that MILS is based on (Information Flow, Data Isolation, Periods Processing, and Damage Limitation) are extended to cover end-to-end communication between MILS partitions [5]. The MILS PCS uses the separation kernel's control over these four policies and applies them to communication between partitions, and also communicates with network protocols and drivers to ensure the policies are applied to incoming and outgoing network data [8]. This PCS, combined with isolation of network components in a single partition, allows a high-assurance backbone for distributed systems by ensuring tight application of security policies on

transmitted data. The following diagram illustrates the function of PCS middleware in a MILS RTU.

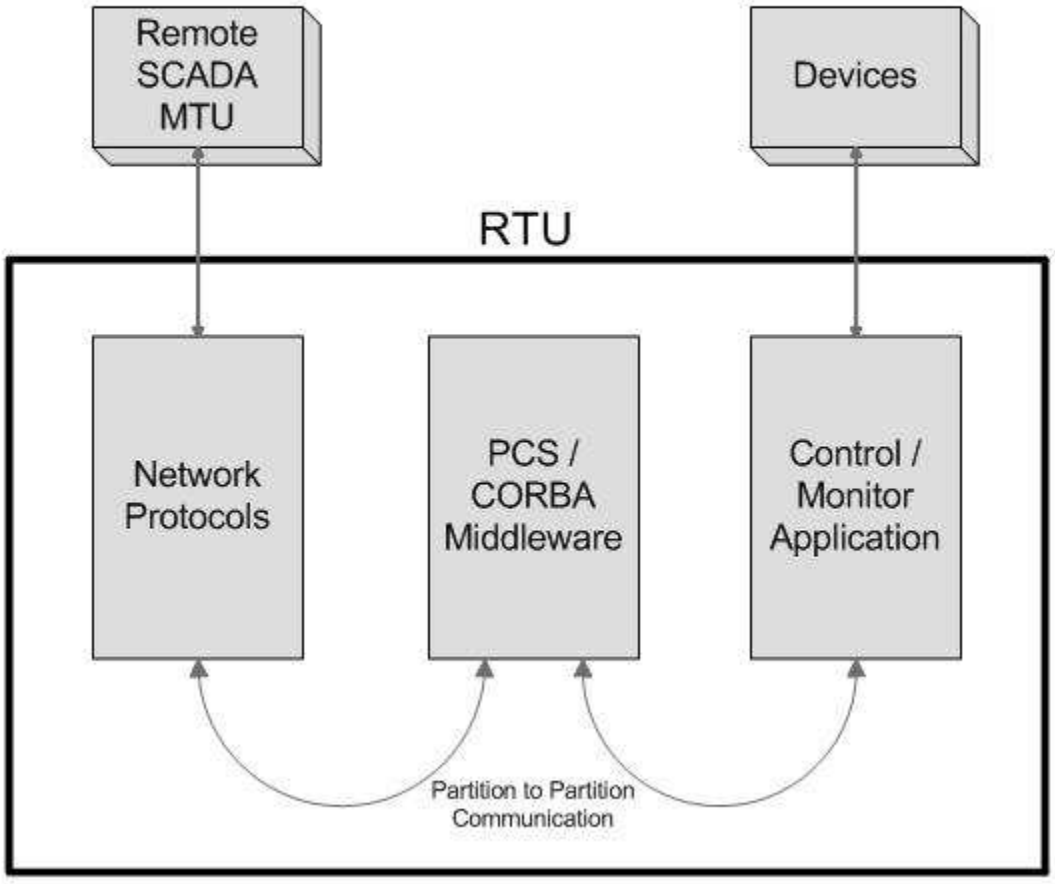

FIGURE 2.1: A PCS for a MILS RTU

2.3.5 Vendors Offering MILS or MILS-Like Products

A number of companies offer operating systems which implement, or will implement, the MILS separation kernel. Green Hills software offers the INTEGRITY line of operating systems utilizing MILS [11]. Wind River has been working on its own real time operating system using the MILS architecture. This implementation uses a

Memory Management Unit (MMU) to protect the kernel from applications and to partition system memory with their VxWorks OS [12].

LynxOS-178 from LynuxWorks is a popular embedded real-time Linux variant which implements MILS partitioning [13]. Via correspondence with the company, it has been learned that they plan on releasing an OS next year called either LynxSecure which will add networking functionality out-of-the-box to their MILS compatible Linux variant operating systems. It is worth noting that none of the above MILS implementations have yet been verified at an EAL 6 or above, so they are not considered fully MILS compliant. However, LynuxWorks is expecting an EAL 7 assurance level for their upcoming product LynxSecure [14], which should allow for full MILS compliance of that operating system.

Objective Interface Systems provides MILS based middleware components. PCS*express*, their PCS implementation [15], is their major current contribution to MILS middleware and could be combined with an aforementioned MILS OS to enable secure networking capabilities on a PC.

2.3.6 Current MILS Research

To date, much of the work done on the MILS standard has been research and theory oriented. There are a number of papers and articles outlining the design of a MILS system and explaining the benefits such a system should provide for security. The article by Van Fleet et al [8] is the quintessential example of such an article. This article is the basis for much of the general knowledge about MILS' applications and benefits. The authors mention the current trend of combining MILS with other security

applications such as the aviation standard DO-178B and ARINC-653, the Avionics Application Software Standard Interface. MILS is often combined with other standards to create a synergy of operation and enhance security. One example of such a combination is the LynxOS-178 real time operating system (RTOS), which combines a partitioning kernel with DO-178B certification and uses ARINC-653 style resource partitioning [23].

The University of Idaho's Center for Secure and Dependable Systems (CSDS) is working on MILS partnered with such groups as the NSA, Lockheed-Martin, and the aforementioned Objective Interface Systems. The objectives of the CSDS are to provide mathematical foundations for concepts for MILS, as well as providing architectural design guidance [24]. While most of the articles published on the CSDS website are from 2002 and 2003, some of the Center's members are still active in the MILS research community.

2.3.7 Current Research on Applying MILS to SCADA

Little has been published in the way of applying MILS to SCADA systems, so this project would seem to be leading the way in that regard. However, there has some cross-pollination of interest in the two areas. Jim Alves-Foss, a prominent member of the aforementioned CSDS, has also co-authored articles such as [25], which looks at applications of security to real-time control and SCADA systems. This shows that other researchers at least have some cross interest in the study of MILS and SCADA systems, though they still have not yet combined the two.

One of the few mentions of the combination of MILS and SCADA comes in the form of a newsletter from the Process Control Security Requirements Forum (PCSRF), a group which focuses on applications of security for SCADA systems and is supported by the National Institute for Standards and Technology (NIST). The newsletter in question announces a conference in April of 2006 presenting a demonstration of the MILS architecture. The newsletter notes that the security potential of MILS could be directly transferable to critical systems such as SCADA networks, as well as financial, medical, and consumer electronic areas. [26] This again shows that others are thinking along the same lines as this project, though they also have still not attempted any implementation or research directly.

2.4 Microkernels

2.4.1 General Microkernel Principles

Microkernels have been around for years in many forms. The idea behind such constructs is to minimize the complexity of the kernel, providing greater security and performance than more traditional kernels. This is accomplished by removing all functionality that can be implemented at a higher level from the kernel code, leaving only essential components inside the kernel. Higher level functionality is controlled by programs known as servers which run on top of the kernel to provide services that do not require direct access to the underlying hardware. The server approach provides greater

flexibility to microkernel based systems, as any additional functionality can be added by the creation of a new server for the task, so no modification to the kernel is required. [27]

2.4.2 The L4 Microkernel

The L4 microkernel is currently at the forefront of microkernel research and development. This kernel was developed in the mid-nineties by Jochen Liedtke, one of the main proponents of microkernel-based systems. Prior to L4, microkernels suffered severe performance hits compared to traditional, monolithic kernels. L4 changed this by dramatically improving inter process communication (IPC) overhead compared to previous microkernels such as Mach. L4 does contain some drivers and a scheduler, the functions of which could be handled by servers. They are included to make development and setup easier for L4 programmers and users. [28]

The L4::Pistachio implementation of this microkernel was utilized in this project as the basis of a secure RTU. Pistachio is maintained by the System Architecture Group at the University of Karlsruhe, where Liedtke had worked prior to his death in 2001 [28]. Pistachio is the L4 implementation that supports the greatest amount of hardware, and is licensed under the BSD license [29]. Pistachio was chosen as the L4 implementation for this project because of the relatively large amount of documentation for it, as well as the community that provides support for it at both the l4ka.org and l4hq.org websites.

CHAPTER III: Designing a Hardened RTU with MILS

3.1 Applying MILS to SCADA RTUs

3.1.1 Current Operating System Security Paradigm

Most operating systems that are currently in wide use employ a "Penetrate and Patch" [9] a.k.a. "fail-first patch-later" [5,8] system to provide security functions. Under this paradigm, an operating system is released "into the wild" to be used by businesses and home users. If a security failure is discovered, for example a hole or backdoor that has been exploited by a hacker, a patch is released to repair the problem and prevent future exploits using that particular security exploit. This paradigm allows the multitudes of worms, viruses, and Trojans that are consistently on the news to cause serious damage [9].

This approach to OS security is not acceptable for SCADA RTUs. SCADA systems were created for use in critical systems and infrastructure. By their nature, they can cause serious damage to property and persons if they are compromised. Therefore, if a "Penetrate and Patch" situation were to occur, it would already be too late: failures that cause critical infrastructure to cease operations cannot be tolerated. A security policy must be enforced that will maximize prevention of security flaws and minimize the damage any unforeseen flaws can cause. This is where MILS, with its focus on core security policies and partitioning of system resources can excel.

3.1.2 Communication over the Internet

Since communication between RTUs and SCADA master units is occurring increasingly over the Internet, this project will focus on using MILS to bolster the security of networked RTUs. MILS is well-designed to allow for enhanced security of networking functions if networking components are placed inside their own partition controlled by the separation kernel's security policies [8]. This means attacks on networking protocols and features cannot spread damage to other partitions which are running critical monitoring and control tasks. From [8], the benefits of this approach are the following:

- "Network facilities can be used by multiple application partitions.
- "Network data is processed in unprivileged user mode, eliminating a vulnerability that is a common avenue of attack.
- "Complex protocol code such as Internet Protocol (IP) Ver. 6 can be evaluated and certified independent of the applications using the code, enabling reuse of the evaluation artifacts."

The core idea here is to place any communication with the outside world (i.e. the network) in its own partition, effectively cutting off all access to other applications except through the middleware and kernel, both of which should be verifiably secure in a true MILS system. The following diagram illustrates the separation of applications into their own partitions on a MILS RTU.

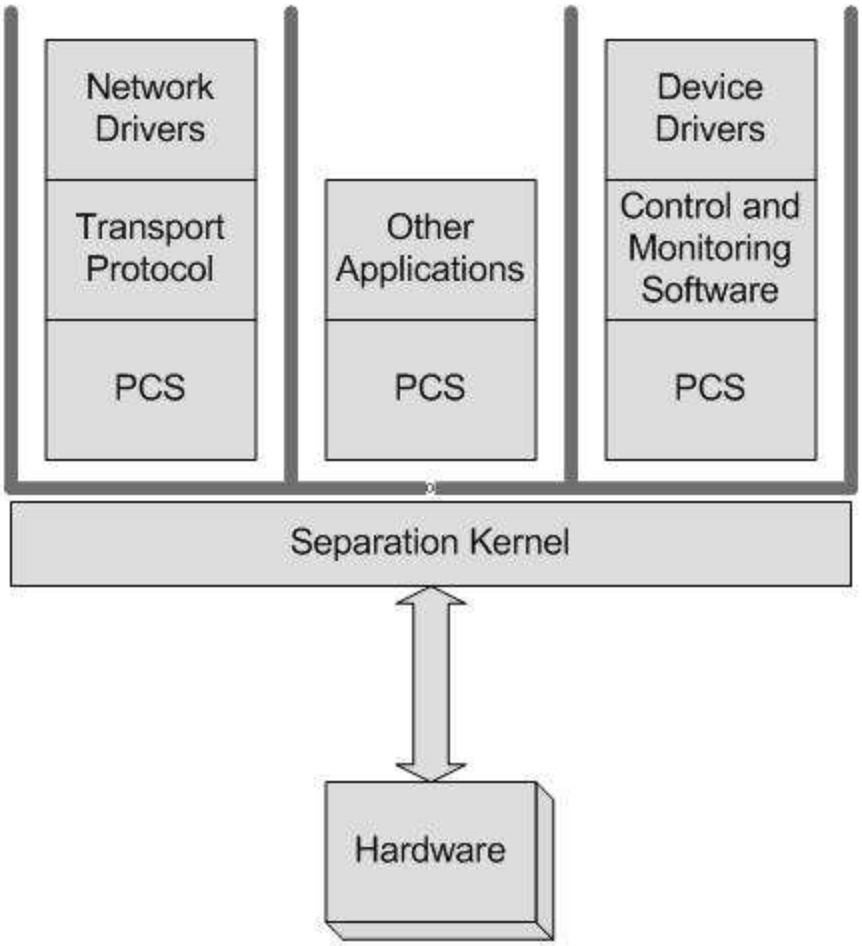

FIGURE 3.1: A MILS RTU

Figure 3.1 illustrates the overall security environment for a MILS RTU. Each separate functionality can be placed in its own partition with its own user permissions and security settings. We see here that networking functionality is contained in a single partition. Likewise with the control and monitoring software that makes up the core of the RTU is in its own, separate partition. Since all communication takes place through the PCS and the secure separation kernel (see figure 2.1 for an illustration of the PCS system), errors and security breaches cannot cascade from, for example, the networking partition to the control partition. Also, information from the monitoring software cannot

be compromised directly through the network connection. Also, as illustrated in the middle partition, separate applications, and even operating systems, may also run on the system with no threat to the control and monitoring software even if they are not verified or known to be totally secure.

3.1.3 Handling of Network Errors in MILS

MILS provides another important feature to PCs used as RTUs in that severe networking errors or attacks will not cause a cessation of operations throughout the entire machine. Since MILS ensures that errors are detected and fixed locally in each partition, monitoring and control software running in other partitions will not be affected by the problems in the network-centric partition. The information obtained by these partitions while the network partition is recovering can be sent to the SCADA master unit when networking operations have resumed normal functionality, preventing loss of data due to network component failure. This is especially important if an RTU is providing control to critical components. As long as the RTU can act independently until the networking partition has recovered, the hardware and infrastructure controlled by that RTU will not function any differently than if the connection had stayed constant. Also, the networking partition can run in unprivileged user mode, removing the potential for administrative exploits [8].

A partitioning communication system can be used on both ends of the network transaction to ensure end-to-end enforcement of security policies. Using a PCS in this manner provides the benefits outlined in Table 3.1 (from [8]). This setup can be used to securely synchronize and communicate over a network, as well as reduce the risk of networking errors by removing backdoors into the communication system.

Table 3.1 Benefits of Networking via PCS

1. Strong identity of each MILS node in the network
2. Separation by level or community of interest, MILS nodes connected by MILS cross-domain servers
3. Secure configuration and validation of consistency of all security databases
4. Secure image loading
5. Secure clock synchronization
6. Provisioning of bandwidth / quality of service
7. Suppression of covert channels (i.e. backdoors)

3.1.4 Attempts to Obtain a MILS Product

As is evident, this project at the onset had the objective of obtaining a MILS compliant operating system and using that OS as the basis of an RTU. However, this did not occur for a number of reasons, and the project has instead shifted to a related focus that will be explained in sections following. First, an explanation of why a MILS product could not be obtained in time for the completion of this project.

In September of 2005 LynuxWorks, the current leader in supplying MILS based technology, was contacted about acquiring one of their operating systems for research in this project. After several emails detailing requirements for the research, the LynuxWorks representative determined that the product we would need is their LynxSecure OS. This real-time OS is LynuxWorks' attempt at creating a verified MILS kernel at EAL 7, which would be the first of its kind. LynxSecure will also feature built-in secure networking, removing the need for the customer (or, in this case, researcher) to implement his or her own networking solution.

This product would indeed provide a solid basis for this project's research, but unfortunately LynxSecure is still not available at the time of this writing. An analysis of the market shows that no other comparable MILS-based product is yet available from

LynuxWorks' competitors, companies such as Green Hills and Wind River. LynxSC, an interim version of LynxSecure designed to be verifiable at EAL 4, was scheduled for a spring 2006 release. As of June 2006, neither LynxSecure nor LynxSC is available to the market.

These complications led the researcher and his advisor to consider alternatives to MILS that could be implemented and researched within the given time frame. This research would focus on applying some of the security techniques that are defined in MILS to RTUs. This project will provide a basis for secure RTUs which can be expanded upon or modified when proper MILS products become available in the future.

3.2 Implementing an RTU Utilizing MILS Concepts

3.2.1 Alternatives to MILS

Since a MILS product could not be obtained in time, this project was refocused into researching what benefits MILS would provide an RTU and discovering how to emulate those benefits with a different system. There were determined to be two major benefits that a MILS compliant system could provide, and these were abstracted into two different areas of research.

First of all, MILS provides hard partitioning of resources, the management of which is handled by the separation kernel. As discussed earlier, this resource management scheme allows the system to contain any errors in a single partition, nullifying the effects of such errors on any applications running in other partitions. If inter-partition communication is needed, a PCS provides verified means of secure

communication between applications in separate partitions by directly invoking the kernel's security methods every time communication is required.

The second major benefit of MILS is the use of a small, hardened, verifiable kernel to manage the system. The benefits of having a minimal kernel are many, not the least of which is the reduction of complexity. This reduction of complexity means that fewer errors should occur in the kernel creation process, as number of errors generally correlates directly with the number of lines of code (LOC) in the kernel. Coding errors residing in the kernel source often lead to exploits being discovered by malicious individuals after the product is released 'into the wild'. It is therefore desireable to reduce the potential for errors up front, and a kernel with the minimum amount of functionality (and therefore LOC) required is a step in this direction. Having a small code base for the kernel also allows mathematical verification of the system, so a quantifiable figure can be placed on the security of the system, a feature which is not possible in a kernel with millions of lines of code.

With these two features of MILS in mind, a search for technology that would provide these benefits was conducted. First of all, a version of LynxOS (version 4.0) was obtained to determine if it was MILS compatible. Unfortunately, it is not as it does not have a separation kernel. This OS distribution did provide one extremely useful feature, which will be discussed in the following section.

Through extensive research, it was determined that brick wall partitioning would be extremely difficult, if not impossible, without a MILS compliant operating system. Resource separation has been mentioned in a few placed as a worthy goal even apart from MILS, but there are few implementations of such available to the public. With this

in mind, the project's focus was shifted to exploring the benefits of a minimal kernel for RTU security.

The benefits of such a kernel have already been listed. It should be noted, however, that this project did not attempt to formally verify either kernel presented at any EAL as the process is lengthy and, as a proof of concept and first step, the work presented here was deemed sufficiently constructive for now.

3.2.2 RTU Based on Embedded LynxOS

As mentioned in the previous section, LynxOS 4.0 provides some interesting features that proved beneficial for this project. First of all, LynxOS is an embedded, real time operating system (RTOS) created by LynuxWorks, the company that will release LynxSecure at some point in the near future.

LynxOS 4.0 was originally obtained because the researcher was attempting to determine if it was MILS compliant. It is not, but some interesting ideas arose from the study of this product. An RTU is much like an embedded device in that it performs specific tasks, often on specific hardware, and reliability of operation is one of the most important features of both devices. Also, LynxOS provides real-time scheduling support, an obviously useful feature for monitor and control in an industrial setting.

Upon further inspection, an even more enticing possibility was discovered. Included with the LynxOS demo is LynuxWorks' Kernel Downloadable Image (KDI) package. This package allows the user to make any desired modifications to pre-built kernels or to create his or her own kernel. This KDI can then be booted from in a number of ways, including from distribution media or over the network. The KDI package was

used to strip all unnecessary functionality from the kernel, with RTU code created by Jeff Hieb used to simulate the monitoring and communication of the RTU. The specific implementation details of this RTU will be discussed in Chapter IV.

3.2.3 RTU Based on L4::Pistachio Microkernel

The Pistachio microkernel is currently the most widely used and supported implementation of the L4 microkernel. The MILS principle of having a small, minimized, verifiable kernel is similar to the idea behind the microkernel. The microkernel paradigm seeks to strip all functionality from the kernel that can be handled at a higher level. Applications called servers provide any functionality that does not need direct access to the hardware, or that can utilize the already-existing kernel primitives for such access. While the microkernel is not designed to be verified at an EAL, its emphasis on minimalization and strict kernel control of all calls made to the hardware provide an interesting parallel to MILS based kernels. For this reason, it was determined that a microkernel based RTU could provide insights into the benefits of a minimal kernel for RTU security, and therefore speak to the benefits of MILS for RTUs.

Pistachio was chosen as the microkernel to be studied in this project first for the amount of documentation and support available. Also, there are user-friendly tools, such as a comprehensive build system, that enhance and speed up development for Pistachio systems. This build system, called Kenge, will be discussed in the following chapter, along with all other implementation details for the Pistachio based RTU.

CHAPTER IV: Prototyping RTUs using LynxOS and Pistachio

4.1 A LynxOS RTU

4.1.1 The LynxOS 4.0 Operating System

LynxOS 4.0 is a real-time embedded POSIX-compliant operating system that was designed for use in critical systems where deterministic real-time performance is essential. It was created by LynuxWorks, the company that is working on bringing the first MILS compatible OS to market in the near future. LynxOS is compatible with UNIX and is similar in architecture. Like UNIX, LynxOS provides full process and thread support with fork and exec system calls. This means applications run in their own protected address space, protecting the kernel from errors created by erroneous application behavior. Because if this similarity to UNIX, many UNIX applications can run with little need for alteration on a LynxOS machine with only a recompilation. LynxOS can also run Linux binaries without the need for recompilation using a built in Linux ABI compatibility layer.

The following table lists some of the features of LynxOS that led to the current research on its suitability for an RTU. These features are listed in this form in the documentation provided with the LynxOS 4.0 demo which was used in this project. These features deal with both the operation of the kernel and the development tools provided with the operating system.

Table 4.1 LynxOS Features

Multiprocess and multithreaded environment
Hierarchical, UNIX-like file system
Kernel threads
Industry standard Networking (TCP/IP)
Support for diskless clients
Industry standard GNU tools, UNIX-like utilities and UNIX-like shell scripts
ROM-able kernel
Modular scalable architecture

Some of these features, such as support for diskless clients and ROM-able kernel, could be helpful for small, embedded RTUs to be designed in the future. While this type of RTU is not the basis for this project, it is a potential future benefit of utilizing this OS. Features such as industry standard networking, as well as GNU and UNIX tools and utilities, should ease the transition from more traditional development to development for a real-time OS. These features, combined with the ability to create Kernel Downloadable Images, make LynxOS a strong candidate for the basis of our prototype RTU.

4.1.2 Kernel Downloadable Images

A Kernel Downloadable Image (KDI) provides the basis for the bootable RTU studied in this project. A KDI is an image that contains the LynxOS kernel, a file system, and any application code specified by the user. This application code consists both of system utilities that are included in any OS, such as network and file managing utilities, as well as any code or applications that is desirable for use within the system represented by the KDI. The KDI, once created, is bootable from flash memory, a disc, or over the network.

The creation of a KDI allows the developer to remove any unnecessary modules from the OS image to ensure that only the necessary applications and utilities are

included. This allows the RTU developer to remove bloat, thereby reducing the memory footprint of the system, and to remove any unforeseen security vulnerabilities that may arise from unnecessary modules. Using a KDI also allows the RTU developer to instruct the system to boot directly into the RTU code. This feature was used in this project and will be discussed in more detail later.

A KDI is created by utilizing the mkimage utility provided by LynxOS. The mkimage utility leverages a .spec (specification) file which contains information on how the utility should configure the KDI. This .spec file contains attributes detailing the initial set of files that should be included in the RAM disk memory image created by the mkimage utility. Table 4.2 lists some of the more important attributes that were employed in the .spec file used for this RTU prototype. These attributes are listed in the mkimage.spec man page and in the LynxOS 4.0 demo documentation, with some slight modification here for readability.

Table 4.2 Important Attributes of a .spec File

Attribute	Description
target=[x86|ppc]	The target system
osstrip=[local|all|none]	Causes local symbol definitions to be stripped from the kernel text file.
ostext=[ram|rom]	Designates where the kernel resides in the running system.
kernel=<*path*>	The path of the LynxOS kernel to be used in the image.
nodetab=<*path*>	The device node table corresponding to the kernel
root=[ram|rom]	Specifies that the root file system is either resident in RAM, ROM, mounted from the device, or that there is no file system.
directory=	A directory on the target file system
file=	A file on the target file system
source=	Designates a fully qualified path name to a source file to be copied into the target file system as the file specified in the file=.
symlink	Designates a symbolic link of <*pathname1*> to <*pathname2*>.

4.1.3 The Developer.spec Specification File

The demo for LynxOS 4.0 that was used for this project was shipped with a few pre-built KDIs, including their corresponding .spec files. For this project, one of these pre-built KDIs called "developer" was modified to provide the basis of our RTU KDI. The developer KDI came with a specification file that included all networking components and other utilities for a fully functional OS. For this reason, the .spec file for this KDI, called developer.spec, was modified to conform to the desired functionality of an RTU prototype. The mkimage utility creates an image ending in a .kdi file extension, so to the image created in this project is named developer.kdi.

As the developer.spec file contained attributes related to providing full OS functionality, including and Apache server, this file was modified to remove unneeded modules from the KDI completely. Therefore, anything unrelated to networking or basic system functionality was removed. Using the directory= and file= attributes, the directory structure and included files for the KDI were specified. The original developer.spec file contained a great many such attributes. Many of these were edited out of the file in order to exclude them from the KDI. Exclusion of such attributes is accomplished by commenting out the line in the developer.spec, with a #, that contains the attribute that should be removed. Appendix I lists the developer.spec file used to specify the attributes for this project's KDI. The lines commented out with the # symbol were attributes that were included in the original developer.spec file. There were actually many more attributes that were removed for this project, but many of these commented

lines were edited out of the file shown in the appendix to reduce the length of the .spec file, allowing easier modification and readability.

The directory= attribute signifies a directory to be included in the image, and the file= attribute does the same for a file. The main additions to the developer.spec file were made in the directory=/net attribute. This attribute specifies what files and binaries should be included in the /net directory of the KDI. The following four lines are the only file= attributes specified for this directory:

```
file=rc.network    source=/tmp/newproj/60.developer/rc.network $(BIN_PERM)
file=rtuDevice     source=./rtuDevice              $(BIN_PERM)
file=rtu           source=./rtu                    $(BIN_PERM)
file=rtuDevice.h   source=./rtuDevice.h            $(BIN_PERM)
```

The purpose of the file=rc.network attribute is to include the file indicated by the path in the corresponding source= attribute. This source= attribute indicates the path to the file on the development machine. The file indicated by the source= attribute will be copied into the /net directory with the name corresponding to the file= attribute, in this case rc.network. The $(BIN_PERM) flag indicates that the file that is to be included in the image in a binary and is permanent, i.e. will not be deleted after the system is fully booted and all modules indicated to begin on startup have are running.

The rc.network file indicated above is essential for the operation of networking capabilities for the system. The purpose of the rc.network file, as well as the modifications to it that were necessary for this project, will be outlined in the following section. The other three targets of the file=attributes in the line above indicate the files needed to run the RTU on the machine and will be discussed in the section following the rc.network section.

4.1.4 The Rc.network File

On UNIX and similarly styled systems, rc files are used to specify startup commands for certain applications or the operating system itself. In this case there are two rc files included in this projects KDI, rc.d and rc.network. The rc.d file contains scripts designed to run any services that are desired upon system startup. This file is was included in the pre-built developer KDI that was used as the basis for this RTU, and its significance to the current topic is that it calls rc.network to run with the following lines of code, which tells the kernel to look at the rc.network file if no scripts from that file are currently running:

```
if [ -x /net/rc.network ]; then
      /net/rc.network
fi
```

Of more significance to this project is the rc.network file. This file was also included in the pre-built developer KDI. However, this file was modified in small but significant ways that allow the RTU to function. The rc.network file contains scripts to setup and enable networking on the machine booted with the developer KDI. It runs the ifconfig and dhclient commands to configure networking for the machine, which explains the attributes indicating the need to include these files in the developer.spec file. The following lines of code are used to start the RTU portions of the code:

```
#
#     Start RTU Server
#
/net/rtuDevice &
start_it /net/rtu
```

The '/net/rtuDevice &' start the rtuDevice portion of the code. The & indicates that the application should run in the background, freeing the kernel up to run the next script in the rc.network file. The 'start_it /net/rtu' command starts the rtu application,

which passes connection information to rtuDevice, which handles monitoring and control for pumps in an industrial plant. The origin and functionality of these applications will be discussed in the following section. To see the complete listing of the rc.network file, see Appendix II.

4.1.5 RTU Code

The code created for the RTU in this project was written by Jeff Hieb, a researcher at the Intelligent Systems Lab at the University of Louisville who is currently researching SCADA and RTU security. The RTU project created by Jeff was written in C++ and makes use of several functions and custom headers spread out through many files. The three code files most important to this project are rtu.cpp, rtuDevice.cpp, and rtuDevice.h.

The rtu.cpp code creates a process that will listen for incoming connections using a server socket. Incoming connections are communicated using the DNP3 protocol for SCADA communication, the support code for which is also included in the RTU project. A master control unit (MTU) can connect to the RTU via this method.

The rtuDevice.cpp code contains the bulk of what would be considered typical RTU functionality. It contains data structures and functions used for monitoring and controlling the levels of pumps in an industrial plant. It accepts connection to an MTU via the connection initiated with the rtu process. Of course, the program does not actually monitor or control any physical objects, but instead methods are included to simulate typical values an RTU in this situation would encounter. There is a .NET based MTU that can remotely connect to this rtuDevice process through the rtu process. This

MTU provides a GUI for providing logon information to the rtu to allow the remote user to monitor and modify values in the tanks. It also provides information on the latency of communication signals, which will be discussed in Chapter V.

The rtuDevice.h file is a header file that provides a definition of message structure for inter-process communication between the rtu and rtuDevice processes. This code ensures that communication is standardized by providing structures for writing and reading messages, as well as user and permission lists, among other features.

4.2 A Pistachio Based RTU

4.2.1 The Pistachio Microkernel

The Pistachio microkernel is an implementation of the L4 microkernel, a high performance microkernel that improves upon previous attempts at microkernel design. The L4 API is the design document that defines the requirements for implementing a microkernel, and the Pistachio microkernel is the first version to implement the L4 Version 4 API. This API provides improvements in a number of ways over its predecessors, such as 32 and 64 bit support, multiprocessor support, and fast local inter-process communication (IPC). Table 4.3 lists some of the features provided by an L4 microkernel that could prove beneficial to an RTU, including information from [28] and [32].

Fast IPC improves performance of the system, so this would be beneficial to any operating system or embedded device. The small number of fundamental mechanisms and kernel defined policies, combined with the small image size for the kernel, provide

an opportunity to study the MILS principle of a small, reduced kernel containing only essential functionality. The privileged threads sigma0, sigma1, and root task are the only threads that can make certain system calls. Sigma0 and sigma1 deal with memory related requests, while the root task helps control overall system operation. These privileged threads separating higher level threads from certain system calls behave similarly to how MILS is designed to operate as well, as only highly trusted threads and processes have direct access to the MILS kernel.

Table 4.3 L4 and Pistachio Features

Fast IPC with low overhead
Small number of fundamental mechanisms built in to kernel
Almost no kernel defined policies (such as memory, protection, and process management)
Small size (from 40 to 200KB)
Privileged threads sigma0, sigma1, and root task

4.2.2 The Basis for a Pistachio RTU

The success of the LynxOS based RTU did not transfer fully to the Pistachio microkernel. While the RTU code from Jeff Hieb is a solid foundation on which to build, it is not yet compatible with Pistachio. A number of difficulties led to this situation, not the least of which is the complexity of development for Pistachio due to the build system it uses. Many of the calls used in the RTU code are not available in the C libraries that are built in to the Pistachio build environment, and networking support is not included in the kernel as was the case with the previous RTU based on the LynxOS kernel.

Instead of reinventing the wheel and creating services to run on top of the microkernel to allow full networking support as is required by the currently available RTU code, this project has instead focused on unraveling the build system and providing a simple example of how to build programs for Pistachio. A basic code example was used that was inspired by a previous, greatly simplified version of an RTU created by Jeff Hieb. It generates some random binary values, transmits them, and simulates some network delay to compensate for the fact that networking is not currently supported. Transmission, in this case, it to standard output, but it would be simple to modify this to transmit over a network connection should one be devised in the future. In this manner, the basis for a Pistachio based RTU has been established, even if the final product is not feasible within this project's time frame.

4.2.3 The Pistachio Development Environment

The Pistachio development environment, much like the Pistachio microkernel itself, is primarily a research project. Whereas Pistachio was created and is maintained by the System Architecture Group at the University of Karlsruhe, the development environment for Pistachio is a product of the Embedded, Real-Time, and Operating Systems (ERTOS) group funded by National ICT Australia (NICTA). ERTOS, as the name implies, focuses on researching various embedded and real-time operating system technologies. They have also modified the Pistachio microkernel to their own embedded version, entitled NICTA::Pistachio-embedded, which focuses on embedded system concerns such as resource utilization and performance. This L4 implementation would be a strong basis for an RTU, but the greater amount of support and documentation

available for regular Pistachio led the project in the more general direction. The build system developed by ERTOS can be used either on their embedded variation or the original Pistachio, and was therefore selected to provide the build environment for this project. This build system and development environment is called Kenge, and consists of four major components which will be discussed in the following sections.

4.2.4 The Kenge Build Environment

The Kenge build system is designed to create a bootable image based on the target machine by compiling user-generated programs together with the Pistachio microkernel, sigma0, roottask, and any other code specified for the image. The Kenge build system utilizes four major components, including a scripting language, toolchains, and two outside applications. Kenge uses the SCONS build system to build projects, the first of the four components. The build scripts that are used by SCONS based on the Python scripting language, the second Kenge component. If building occurs on the target machine, the standard gcc version 3.3 toolchains are used for compilation of code. This is the third component. The fourth and final component is the QEMU simulator, a full IA-32 system simulator. [33]

4.2.5 The SCONS Build System

The SCONS build system is based on the Python programming language. Builds are controlled by a top-level SConstruct file, which contains functions to set up the build environment and combine all the separate modules into a final, bootable image. Lower level SConstruct files are also used to specify how the system should and link build

individual modules such as applications and libraries. Two SConstruct files were essential to the build of the simple example rtu code. As both files are titled SConstruct, they will be referred to as the high-level SConstruct and the low-level SConstruct.

The low-level SConstruct used for this project is a Python file which specifies that this module contains an executable program, in this case the C code file rtu.c. Using the env.MyProgram() function targeting rtu.c, the low-level SConstruct file specifies to the build environment that rtu.c should be compiled and treated as an executable file. Using the information from this low-level specification, the high-level SConstruct can properly handle the rtu module once the high level build begins. The low-level SConstruct is listed in Appendix III.

The high-level SConstruct is more complex, as it specifies the overall build environment as well as any applications that should be included in the final bootable image. In this file, the first step is to create the build environment for the machine, called env as indicated in the low-level SConstruct explanation above. This environment determines the compilers and flags necessary for the build process on the specified machine. There are then calls to specify the kernel to be used, in this case the Pistachio microkernel. The rootservers, special processes that begin at boot time, are specified next, along with any libraries that are needed to compile programs in the build environment. This program includes the “c” and “l4” libraries, as the simple rtu code is written in C and the “l4” library contains code relating to compilation for the L4 microkenel. Any applications that are required to start at boot time are then specified. In this case, the sigma0 memory manager process and the rtu application are indicated. The final two scripts combine the kernel and applications into a single bootable image, and set

that image as the default boot image for the system. The high-level SConstruct file is listed in Appendix IV.

4.2.6 The QEMU Simulator

Kenge uses the QEMU simulator to simulate the runtime operation of bootable images configured by SCONS. QEMU is a full IA-32 system simulator, meaning it can run inside another operating system while simulating the functions of the created boot image. The use of such a simulator eases the development process of a bootable operating system by running the system within the development environment, eliminating the need to hard boot the system from a disk, ROM, or a hard drive.

The Kenge environment is setup to simulate bootable images directly after a build by setting a "simulate" flag when calling SCONS. One problem with using this method for simulation was an inconsistency with how Kenge calls QEMU. When the simulate flag is set, Kenge calls QEMU after the bootable image has been created with the following command:

```
qemu –hda build/c.img –nographic –nics 0
```

The problem here is the current version of QEMU does not use the –nics flag to specify the number of network interface cards to simulate, but instead uses a –net nic flag. Even though this operation ends in an error, the simulate step is necessary to build the c.img bootable image as specified in the command. This image contains all files necessary for system boot. After the simulate command has been executed and stopped with an error, the QEMU simulator can be manually started targeting the created c.img. The command for this execution will be specified in the next section.

4.2.7 Building, Booting, and Running the System

Once the SConstruct files are created and all needed support tools and files are in place, Kenge is ready to build the system. The following command builds and links all modules into a single bootable image:

```
scons machine=pc99
```

The machine=pc99 flag tells SCONS that the target machine is an IA-32 based PC. To create and simulate the c.img bootable image, the following command is issued:

```
scons machine=pc99 simulate
```

As discussed in the previous sections, this command ends in an error but correctly produces the c.img file that is needed for simulation. QEMU now should be called manually with the following command:

```
qemu –hda build/c.img –nographic
```

This command boots c.img in the QEMU simulator. The –hda flag indicates that the target is a hard disk image, while the –nographic flag starts QEMU without graphical support. This saves memory and processor resources, as the bootable image created for this project is purely text driven and has no graphical interface. The –net nic flag is not needed because the default value is sufficient for this simulation.

Once the simulator has started, the kernel is booted along with the applications specified in the high-level SConstruct file. The rtu code is started and begins its operation by asking the user to press a key to begin. The code then generates and outputs a pseudo-random binary value represented by hexadecimal, simulates network delay, and attempts to measure the time needed for the operation. The user can then choose to press

x to exit or any other key to end the program. The time measurement does not work correctly when booted on Pistachio because the system calls needed for the functions in the time.h header file are not available from the L4 microkernel to this process. The system boots correctly, processes user input and processes and generates binary output. Further work that could be done with this microkernel and build system will be discussed in Chapter VI. The code for the rtu.c file is listed in Appendix V, and a sample output listing for the run-time operation of this code is provided in Appendix VI. The code was running in the QEMU simulator on top of the simulated Pistachio kernel.

CHAPTER V: Performance and Security Evaluation

5.1 Quantitative Security Analysis

5.1.1 Objectives of Analysis

The goal of this project from the outset was to determine what benefits, if any, MILS concepts could provide to RTU security. The LynxOS and Pistachio systems were utilized for the similarities they shared with certain MILS concepts. This section will attempt to scrutinize precisely what security vulnerabilities can be closed for each of these approaches, what vulnerabilities are still open, and which of these open vulnerabilities could be solved with a fully MILS compliant RTU.

5.1.2 Vulnerabilities Closed Using LynxOS

Table 5.1 outlines the security vulnerabilities that were closed by implementing a prototype RTU with the LynxOS KDI development tools. These closed vulnerabilities are mostly related to MILS concepts, meaning that they are also vulnerabilities that should be closed by a fully MILS compliant OS and kernel.

Table 5.1 Vulnerabilities Closed with LynxOS

Small kernel, reducing complexity and risk
Increased periods processing protection
RTU applications run in unprivileged user mode
Removal of shell from KDI
Enforcement of remote communication using DNP3 protocol

Most of these security improvements are specific to MILS concepts of secure communication and processing. First of all, MILS calls for a kernel that is reduced in complexity from more traditional kernels in order to reduce the complexity, and therefore unforeseen risks, involved with a kernel containing hundreds of thousands or millions of lines of code. MILS takes this a step farther and calls for a kernel to be formally and mathematically verified. While such verification is not possible for the kernel used in this RTU setup, the kernel nevertheless takes a step in the direction of MILS compliance compared to more traditional kernels, and therefore increases confidence in the security performance of the system.

Periods processing is a MILS concept which states that exploits using the processor or networking interface cannot grant access to the system to an intruder (for more, see Chapter II). Periods processing is one of the four security policies that defines the basis of the MILS standard. The LynxOS based RTU increased assurance in periods processing in two ways. First, all applications started after boot time run in unprivileged user mode. This means that an intruder who manages to connect with the code from rtu.cpp that allows the remote connection to the MTU should not be able to exploit this connection to cause system changes that require root access.

One argument against this notion would be that an intruder could somehow gain account information for root and achieve an apparently legitimate login with the stolen information. This concern is addressed by the removal of the UNIX like shell from the KDI. The shell is the command line user interface that allows a user to input commands to the system. By modifying the developer.spec and startup scripts for the KDI, this functionality was removed from the image in order to reduce system complexity and

prevent exploits such as the one outlined above. While the removal of the shell ensures that the system is secure from command line exploits, it also reduces the overall usefulness of the operating system for everyday tasks. This was deemed to be a fair trade off, because RTUs are highly specialized machines with the singular purposes of monitor and control.

All network communication after the initial DHCP acquisition of an IP address is handles via the rtu process. This process utilized the DNP3 protocol for SCADA communication, effectively shutting out any connection attempt not conforming to the protocol. While attempts by outside forces to connect could cause errors in the operation of the rtu and rtuDevice processes, functionality could be built in to auto-reboot after fatal errors or to allow reboots or re-running of applications with faults to be specified over the network connection with the MTU. See Chapter VI for more on such potential additions. While this security improvement is not related to MILS concepts, it is a side effect of the way RTUs and MTUs communicate and is implicit within the operation of this RTU, and is therefore included in the security analysis.

5.1.3 Vulnerabilities Closed Using Pistachio

The similarity of the microkernel concept to the MILS concept of a secure, minimized kernel is significant. For that reason, microkernels address many of the concerns that form the basis of the MILS standard. While this project did not create a prototype RTU complete with full networking capabilities, many security improvements resulting from the use of this microkernel setup become apparent on analysis of the

features of L4 and Pistachio. Table 5.2 gives a brief overview of these security improvements, which will be discussed in detail following the table.

Table 5.2 Vulnerabilities Closed with Pistachio

Small, minimally complex kernel
Most functionality provided by user level modules
Privileged threads
Address spaces separate services from one another

The use of Pistachio as the basis of an RTU, similar to the use of the LynxOS kernel, builds on the concept of a minimal, secure MILS kernel. Pistachio goes beyond the LynxOS approach to abstract out all non-essential functionality from the kernel and replaces such functionality with higher level system and user services. As the complexity of this kernel is greatly reduced compared to monolithic kernels that handle everything from file systems to networking, this kernel can provide the RTU designer with greater assurance that kernel functionality will not lead to errors. Pistachio improves over the LynxOS kernel in this regard, as even that kernel contains much built in functionality and is more similar to a monolithic kernel, albeit reduced and streamlined for embedded, real-time use.

The privileged threads sigma0, sigma1, and roottask provide another level of abstraction between high level modules and certain system calls. As long as these threads cannot be exploited, the system calls they are associated with cannot be exploited either. This plays to the MILS idea that small, verifiable applications can run on top of the kernel to provide another degree of security between non-verified modules and the system. If the code behind these privileged threads is written correctly and is free of security holes, intruders cannot gain access to any system calls these threads are associated with.

The idea of address spaces also supports a MILS concept, that being the idea of resource partitioning to prevent error propagation throughout the system. Through the microkernel's inter-process communication (IPC) system, programs can reside in separately assigned address spaces and still communicate through certain kernel IPC calls. This also brings to mind the concept of a partitioning communication system (PCS) for a MILS based OS. However, the microkernel IPC system is built in to the kernel while the PCS is a middleware component that runs between user level processes and the kernel. Also, while address spaces protect processes from damaging and being damaged by errors in other memory spaces, there is no support for brick wall partitioning of processor resources. Therefore this feature, while useful from a preventative security standpoint, is not a substitution for the partitioning provided by a fully MILS compliant system.

5.1.4 Remaining Vulnerabilities

Though these two approaches close many security vulnerabilities extant in current networked RTUs, there are vulnerabilities still open that could be addressed by a full MILS system. The first of these is the absence of resource partitioning between the networking code and the monitor and control applications. An attack on system memory could effect the LynxOS based RTU, while an exploit of the processor could effect either system. The use of address spaces protects the Pistachio based system from some memory errors, but is not as rigidly enforced as MILS brick-wall partitioning.

Another vulnerability is the lack of a verifiably secure means for end-to-end network communication such as a partitioning communication system (PCS). While the

DNP3 protocol provides a network interface between the RTU and MTU, it is not verified like a PCS could be. A security verified PCS would provide the additional benefit on a MILS system of securely handling partition-to-partition communication, providing an additional layer of separation between the networking partition and monitor and control partitions.

5.2 Performance Analysis

5.2.1 Latency Analysis for LynxOS RTU

As mentioned in Chapter IV, the creator of the RTU code used on the LynxOS prototype also created an MTU to remotely connect to the rtuDevice process via the rtu process. This MTU provides a graphical interface that allows the user to provide logon information, logon to the RTU, and remotely begin the operation of the RTU. The user can then monitor and change values in pumps simulated by the program. The MTU constantly polls the rtuDevice to determine if any changes have taken place, and it provides mechanisms to record and store latency measurements. Sample output from the latency measurement of this program for a short connection session is listed in Appendix VII.

Ignoring outliers due to loss of packets, which occurred only once during the data gathering process, some analysis on the latency data is possible. The latency measurement is time between a request for information from the MTU and the arrival of the response from the RTU. It takes into account transmission time to and from the RTU

as well as any processing that is required of the RTU once the request has arrived. The output listed by the MTU is measured in milliseconds.

The data for this test ranges from 1.504661ms to 1.808051ms, with a median of 1.559416ms. The mean value for the latency represented by this data is 1.580219ms with a standard deviation of 0.050565. The small standard deviation suggests that the data is clustered closely around the mean, an assumption which is borne out by the proximity of the mean and median values. These round-trip latency times fall below the sampling time in most SCADA systems, showing the performance of this system is sufficient at least over a LAN connection.

This test was conducted via a local area connection with the MTU and RTU machines in fairly close proximity. A system distributed at greater distances would affect performance, as would a connection over the Internet as opposed to intra-network communications. However, the data obtained suggest efficient performance for the LynxOS based RTU, performance which should be more than sufficient for most SCADA communication.

5.2.2 Overhead Incurred from Use of L4

Microkernels, in their early days, were stigmatized for poor performance stemming from the design principals that guided there creation. The L4 kernel was created as a response to such criticisms as a way to determine if microkernel architecture truly could provide performance comparable to that of other kernels. Studies conducted on the performance of L4 can be applied to this project to determine if use of the

Pistachio microkernel is feasible in terms of performance for the overall RTU architecture.

Studies conducted by porting Linux to the L4 microkernel showed throughput to be only 5% lower than that recorded from native Linux. Load testing determined that the Linux L4 port, titled L4Linux, is 8.3% slower than native Linux if all loads are averaged and 6.8% slower if only maximum load is accounted for. This is a huge improvement over previous microkernels, which could perform up to 60% slower than native Linux, but this data still does not lend itself well to high-performance applications. [34]

An empirical study of a full featured RTU would need to be conducted to determine if a Pistachio can provide the required performance under run time conditions. As with all improved security measures, the trade off of performance for security may be necessary to ensure confidence in RTU operations over a network. An average 8.3% performance hit should theoretically not be a problem for most RTUs as much of the processing of data is handled by the centralized MTU, requiring less processor power and fewer hardware resources than would be required otherwise. This is a reasonable trade off for enhanced RTU security, but again an empirical analysis of performance, not to mention analysis of costs if better hardware is needed for the L4 based RTU, would be required before the deployment of such an RTU to the field.

5.2.3 Kernel Size

The size of the kernel used in an RTU can be a vital statistic in certain situations. For example, an RTU based on a LynxOS KDI can be booted from flash memory or over a network on a diskless system. This means the kernel and all application and support

data must fit into the system's flash memory, ROM, or RAM. The memory footprint required for the kernel should therefore be minimal while still providing all required functionality for the system. This brings to mind again the MILS concept of a minimalized kernel containing only basic system calls and operations.

The size of the Pistachio kernel for IA-32 systems is 163,983 bytes, or about 160.14KB. This is within the typical 40 to 200KB range of microkernels sizes. The size of this kernel should not be an issue, as it could fit on many cell phones with extra room for application data. Of course, enough room must be provided for any servers providing functionality not in the kernel as well as RTU code, so kernel size is not the only size measurement that could be applied before deployment of an RTU design. It should also be noted again that, with the decreased size of this microkernel, a certain performance penalty is incurred which should be considered whenever obtaining a hardware configuration for the RTU.

The LynxOS kernel is configured for the hardware on which it is running. For the machine used in this project, the size of the LynxOS kernel is 1,246,244 bytes, or 1.1885MB. This is a significant increase over the size of the Pistachio microkernel caused by the more traditional monolithic architecture of the LynxOS kernel. Even with an increase of over 7.5 times, this kernel is still small and should easily fit within most, if not all, memory configurations feasible for an RTU machine. It is also a significant improvement over the size of more traditional monolithic kernels, such as the Linux kernel.

As an indirect comparison, it can be noted that the latest full version of the source for the 2.4 Linux kernel comprises almost 37MB worth of code and build rules in a

compressed format. When uncompressed, the source measures over 167MB. While this is not a direct comparison of kernel size, it demonstrates the difference in scale and scope of a monolithic kernel compared to an embedded kernel or microkernel.

CHAPTER VI: Conclusions and Future Directions

6.1 Conclusions

This project, upon initiation, set out to analyze what security benefits, if any, MILS could provide for SCADA RTUs. Although a fully MILS compliant system was not studied, the shift of focus toward architectures that provide features similar to MILS allowed the researcher to study some of the security concepts that form the basis of MILS. Because of this, the project has generally accomplished its initial goals. Several vulnerabilities were closed by these systems, indicating that MILS could indeed prove useful for RTU security, and that these alternatives could serve productively in the interim.

The greatest success of this project came in the form of a fully functional RTU running on top of the embedded LynxOS kernel. This RTU demonstrates how reducing underlying system complexity, a MILS concept, can provide the basis for a fully functional RTU while improving security by closing certain vulnerabilities. This system allowed a quantitative analysis of closed security vulnerabilities as well as performance analysis via a remotely connected MTU application. Since this RTU is based on a kernel developed for embedded real-time applications, performance is high and overall required resources are low, both desirable qualities for an RTU.

Because of the research presented in this project, the future RTU researcher has a clear base from which to build upon to develop an RTU running on top of the Pistachio microkernel. The concept of a microkernel is similar to the MILS concept of basing

security around a minimal, secure kernel. The L4 microkernel goes so far as to provide some small amount of resource protection via address spaces for resident memory applications.

Even with the success of much of the research conducted during this project, much remains to be done. There are many further avenues of study available such as expanding the LynxOS RTU, developing a full-fledged RTU on top of Pistachio, and studying a fully MILS compliant system, all of which would address valid questions not answered by this project. Such future work is discussed in detail in the following section, which should provide some guidance for future research on this topic, whether or not that research builds upon this project.

6.2 Future Directions

6.2.1 Expanding the LynxOS-based RTU

While the creation of a prototype RTU utilizing the LynxOS kernel downloadable image (KDI) creation tools was a success, there are many avenues still open for research and expansion. First and foremost, a strong series of tests should be conducted to determine the exact security strength of the networking code native to the operating system. LynxOS uses many customized functions such as its TCP stack to provide networking capabilities. Because any KDI inherits the portions of the LynxOS kernel and major functionality as specified during KDI creation, thorough analysis of such features would be required before any deployment of a LynxOS based RTU.

A more robust performance evaluation would also be a beneficial addition to the knowledge base for this RTU. Better access to kernel documentation and kernel performance analysis from LynuxWorks would greatly aid this process. Currently, LynuxWorks is rather closed regarding information about the inner workings of the LynxOS demo on which the RTU is based. Advanced kernel performance evaluation tools would allow measurements on the number of clock cycles needed for certain operations as well as the amount of context switching, memory, and transfer time needed for such operations. This information would allow the RTU designer to choose the optimum hardware for each RTU design, thereby improving efficiency, utilization, and performance yield for the unit.

As the RTU running on this system has no shell for command line user input, mechanisms will need to be created to allow the system to recover from errors. For example, the system could auto-reboot if the kernel encounters a fatal error, or a remote reboot of a faulty application could be an option from the MTU controlling the RTU. It would be impractical to perform a hard reboot every time a process, or even the entire system, encounters a fatal error it cannot handle on its own, so such options would be vital additions for future researchers to study. Another feature that would need to be built into the communication code between the RTU and MTU would be the ability to retransmit or recover dropped packets, as drops can occur even over a LAN as demonstrated by the data in Appendix VII.

A future path of study for this machine would be customization of the LynxOS kernel itself, as opposed to simply creating KDIs utilizing the default LynxOS kernel. LynxOS provides the ability to accomplish this by modifying the kernel directory and

supporting scripts and makefiles. According to the LynxOS 4.0 demo documentation, the kernel can be modified for performance, size, and functionality. Such modifications could reduce the complexity of the kernel and bring this system a step closer to the target MILS concepts upon which this project is based.

6.2.2 Expanding the Pistachio-based RTU

Of the two RTU architectures researched for this project, the RTU based on the Pistachio version of the L4 microkernel demonstrates the most room for improvement. First and foremost, a robust networking interface for Pistachio must be developed as the basis of communication for the RTU. The networking features should be in the form of servers, separate modules running on top of the microkernel, instead of integrated in the kernel itself. This setup, the basis for microkernel design, ensures future extensibility and eases the maintenance and expansion of the networking modules in the future. It also follows the MILS specification of a secure, minimally complex kernel to control the system.

The code created for this project to work with the Pistachio build system and environment is not an RTU, but instead simulates some operations of an RTU as a proof of concept for microkernel use. If research in this area is to continue, it will need to build upon the design and prototyping work presented in Chapters III and IV of this document to create a robust, fully featured RTU based on Pistachio. A difficulty encountered during the research into the Pistachio microkernel is the lack of clear, efficient documentation to outline the features of the system. This is true even for Pistachio, which appears to have the greatest amount of documentation support from the

microkernel development community. For this reason, the work presented in this document will help the future researcher by demystifying development for a Pistachio based system.

A more thorough performance evaluation should be conducted on this microkernel to determine its suitability for monitor and control applications. Since the inception of microkernels, there has always been concern over their performance. Many of these issues have been addressed in one way or another, but precise measurements should be recorded to determine if the overall greater amount of context switching required for operations with a microkernel could prove too detrimental to the performance of the system. The reverse should also be researched to determine if microkernels could provide performance increases in certain areas.

There are many implementations of the L4 microkernel and, as progress with this kernel is currently carried out through research and academic institutions, each implementation differs and can provide better functionality in certain areas. The Pistachio implementation was chosen because of its current popularity for use in research and its relatively strong amount of documentation. As mentioned previously, the NICTA::Pistatchio-embedded L4 implementation would be a good fit for an RTU, as its focus in on further reducing kernel complexity and memory footprint. The Fiasco microkernel is also popular in the research world. While this microkernel is not a direct L4 implementation, it was designed to be compatible with L4 as it was created to serve as a new basis for a pre-existing L4 project focusing on operating system quality of service requirements. The l4hq.org website is the central hub for obtaining information about

current and legacy L4 projects, and should be consulted in the future if a desire for furtherance of the use of L4 in a SCADA RTU context exists.

One upcoming L4 project should prove of interest to the researcher willing to investigate the use of L4 for RTUs. That project is the Secure Microkernel Project (seL4) from ERTOS and NICTA [35]. The goal of this microkernel is to use formal methods and computations to provide a high degree of security assurance in the kernel itself to provide the basis of trustworthy embedded systems. This seems to take a page directly from the MILS concept of kernel verification, and could be looked at in the future if research in this direction continues.

6.2.3 Toward a MILS Compliant RTU

This intent of this project at the outset was to study the benefits gained by applying the MILS standard to SCADA RTUs. As the focus of the project was forced to shift away from directly applying MILS to an RTU, the objective of applying MILS concepts to an RTU remained unchanged. This project has demonstrated how some of these concepts may improve RTU security, but the work in this direction has only just begun.

This project was not able to study the impact of brick-wall resource partitioning for RTU security. This feature of MILS would appear to be its strongest aspect in the face of network security threats, as discussed in Chapter III of this document. Such partitioning would ensure networking errors could not propagate to effect control and monitoring applications and would provide a greater amount of confidence in the overall system.

As mentioned in Chapter II, a number of vendors are offering products currently or in the near future that are designed to utilize MILS concepts. Green Hills and Wind River are two embedded OS developers offering products with MILS-like separation kernels. LynuxWorks also offers such products, and plans to introduce the world's first verified MILS kernel in the near future. The researcher who desires to study a true MILS system should watch for this product.

REFERENCES

[1] Fact Index, SCADA Systems. http://www.fact-index.com/s/sc/scada.html Accessed August 2005

[2] Tek Soft Consulting. "SCADA RTU's." http://members.iinet.net.au/~ianw/rtu.html Accessed August 2005

[3] Tek Soft Consulting. "SCADA Primer" http://members.iinet.net.au/~ianw/primer.html Accessed November 2005

[4] Brown, A.S. "SCADA vs. the hackers." *Mechanical Engineering*. Dec. 2002. http://www.memagazine.org/backissues/dec02/features/scadavs/scadavs.html Accessed August 2005

[5] Objective Interface Systems. "MILS White Paper." http://www.ois.com/mils/ Accessed August 2005

[6] Rushby, John. "The Design and Verification of Secure Systems." *ACM SIGOPS Operating Systems Review*. v 15, n 5, Dec. 1981, pp. 12-21.

[7] Rushby, John. "Proof of Separability: A Verification Technique for a Class of Security Kernels." *Computer Science* 137: (1982) 352-367.

[8] Vanfleet, M.W., J.A. Luke, R.W. Beckwith, C. Taylor, B. Calloni, G. Uchenick. "MILS:Architecture for High-Assurance Embedded Computing." *Crosstalk: The Journal of Defense Software Engineering*. Aug. 2005. http://www.stsc.hill.af.mil/crosstalk/2005/08/0508Vanfleet_etal.html Accessed August 2005

[9] Singh, I.M. "Homeland security and embedded software." *Embedded Computing Design*. 2004. http://embedded-computing.com/articles/singh2/ Accessed August 2005

[10] "Common Criteria." http://www.commoncriteriaportal.org/ Accessed November 2005

[11] Green Hills Software, Inc. "Products." http://www.ghs.com/products.html Accessed November 2005

[12] Joint Tactical Radio System (JTRS). "Technology Awareness Bulletin." Vol. 2 no. 6. June 2004.

[13] LynuxWorks. "Embedded Systems." http://linuxworks.com/products/overview.php3 Accessed November 2005

[14] "LynuxWorks to Demonstrate LynxSecure Separation Kernel at Embedded Systems Conference (ESC)" http://www.lynuxworks.com/corporate/press/2005/lynxsecure-demo.php Accessed November 2005

[15] Objective Interface Systems. "PCSexpress for Military and Aerospace." http://www.ois.com/images/PCS_BROCHURE.pdf Accessed September 2005

[16] Poulsen, Kevin. "Slammer worm crashed Ohio nuke plant network." *SecurityFocus*. August 2003. http://www.securityfocus.com/news/6767 Accessed March 2006

[17] "Hackers attempted to breach California power grid." http://archives.cnn.com/2001/TECH/internet/06/09/california.energy.hackers/ Accessed March 2006

[18] Bowen, C.L. III, T.K. Buennemeyer and R.W. Thomas. "Next generation SCADA security: best practices and client puzzles." *Proc. IEEE Systems, Man and Cybernetics (SMC) Information Assurance Workshop 2005.* June 2005. pp 426-427.

[19] Fan, R., L. Cheded and O. Toker. "Interned-based SCADA: a new approach using Java and XML." *Computing & Control Engineering Journal*. v 16, n 5, Oct.-Nov. 2005, pp 22-26.

[20] Whitlock, Steve. "Harnessing SCADA without undermining security." *Journal of the American Water Works Association*, v 96, n 7, July, 2004, p 51-53+111.

[21] Abshier, J. and J. Weiss. "Securing control systems: what you need to know." Control, v 17, n 2, Feb. 2004, p 43-8.

[22] Hauser, C.H., D.E. Bakken and A. Bose. "A failure to communicate: next generation communication requirements, technologies, and architecture for the electric power grid." *IEEE Power & Energy Magazine*, v 3, n 2, March-April 2005, p 47-55.

[23] LynuxWorks. "RTOS for Software Certification: LynxOS-178." http://www.lynuxworks.com/rtos/rtos-178.php Accessed September 2005

[24] CSDS MILS Project. http://www.csds.uidaho.edu/mils.shtml Accessed October 2005

[25] Oman, P., A. Krings, J. Alves-Foss and D.C. De Leon. "Analyzing the security and survivability of real-time control systems." *Proceedings from the Fifth Annual IEEE System, Man and Cybernetics Information Assurance Workshop, SMC*, 2004, p 342-349

[26] Stouffer, Kieth. "{PCSRF} Upcoming meetings." *PCSRF Newsletter*. April 05, 2006.

[27] National Information and Communications Technology Australia (NICTA). "Microkernels." http://ertos.nicta.com.au/research/l4/microkernels.pml Accessed February 2006

[28] NICTA. "About L4." http://www.ertos.nicta.com.au/research/l4/about.pml Accessed May 2006

[29] L4HQ. "Kernel APIs." http://l4hq.org/kernels/ Accessed May 2006

[30] Byres, E. and J. Lowe. "The Myths and Facts behind Cyber Security Risks for Industrial Control Systems." *VDE Kongress*, Berlin, Germany. 2004.

[31] Fernandez, J.D. and A. E. Fernandez. "SCADA systems: vulnerabilities and remediation." *Journal of Computing Sciences in Colleges*. v 20, n 4, April 2005, pp 160-168.

[32] System Architecture Group. "The L4Ka::Pistachio Microkernel white paper." May 1, 2003. http://l4ka.org/projects/pistachio/pistachio-whitepaper.pdf Accessed May 2006

[33] ERTOS. "Externel tools needed." http://www.ertos.nicta.com.au/software/kenge/build-tools/latest/required_tools.pml Accessed May 2006

[34] Liedtke, J. et al. "The Performance of µ-Kernel-Based Systems." *16th ACM Symposium on Operating System Principles*. October 5-8, 1997. http://os.inf.tu-dresden.de/~jork/papers/sosp97.pdf Accessed June 2006

[35] ERTOS. "Secure Microkernel Project (seL4)." http://www.ertos.nicta.com.au/research/sel4/ Accessed June 2006

[36] Patel, Sandip Chunilal. *Secure Internet-Based Communication Protocol for SCADA Networks.* Doctoral Dissertation. Department of Computer Engineering and Computer Science, University of Louisville. May 2006.

APPENDIX I: Developer.spec File

The developer.spec file is used to specify the desired components for a bootable image created by the mkimage utility and kernel downloadable image (KDI) creation tools. This is the file that allows the developer to include any applications or code that should be executed on the target machine. Some comments were edited out for length and readability. Attributes attached to unwanted files are commented out to exclude those files from the final bootable image. This file is modified from the developer.spec file included with the demo version of LynxOS 4.0 that was used for this project.

```
# Kernel
target=$(TARGET_ARCH)
osstrip=false
ostext=rom

kernel=$(BSP_DIR)/a.out
nodetab=$(BSP_DIR)/nodetab

# Boot-up method
#flags=a

# File System
free=200
inodes=10
root=rom

#Text Files:
strip=none
text=rom
resident=false

# KDI files

directory=/       $(DIR_PERM)
 file=init        source=$(ENV_PREFIX)/init     $(BIN_PERM)
 file=KDI.sh      source=$(PROJECT_DIR)/KDI.sh  $(FILE_PERM)


directory=/bin    $(DIR_PERM)
# file=netstat    source=$(ENV_PREFIX)/bin/netstat    $(BIN_PERM)
# file=tcpdump    source=$(ENV_PREFIX)/bin/tcpdump    $(BIN_PERM)
 file=ps          source=$(ENV_PREFIX)/bin/ps         $(BIN_PERM)
# file=vi         source=$(ENV_PREFIX)/bin/vi         $(BIN_PERM)
# file=chmod      source=$(ENV_PREFIX)/bin/chmod      $(BIN_PERM)
 file=hostname    source=$(ENV_PREFIX)/bin/hostname   $(BIN_PERM)
```

```
 file=ifconfig    source=$(ENV_PREFIX)/bin/ifconfig   $(BIN_PERM)
 file=drivers     source=$(ENV_PREFIX)/bin/drivers    $(BIN_PERM)
 file=devices     source=$(ENV_PREFIX)/bin/devices    $(BIN_PERM)
 file=ping        source=$(ENV_PREFIX)/bin/ping       $(S_PERM)
 file=dhclient    source=$(ENV_PREFIX)/bin/dhclient   $(BIN_PERM)
# file=ls          source=$(ENV_PREFIX)/bin/ls         $(BIN_PERM)
# file=login       source=$(ENV_PREFIX)/bin/login      $(S_PERM)
 file=reboot      source=$(ENV_PREFIX)/bin/reboot     $(BIN_PERM)
 file=sh          source=$(ENV_PREFIX)/bin/bash       $(BIN_PERM)
 file=tset        source=$(ENV_PREFIX)/bin/tset       $(BIN_PERM)
 file=mount       source=$(ENV_PREFIX)/bin/mount      $(BIN_PERM)
# file=drinstall  source=$(ENV_PREFIX)/bin/drinstall  $(BIN_PERM)
# file=devinstall source=$(ENV_PREFIX)/bin/devinstall $(BIN_PERM)
 file=umount      source=$(ENV_PREFIX)/bin/umount     $(BIN_PERM)

directory=/usr/bin      $(DIR_PERM)

directory=/etc          $(DIR_PERM)
# file=passwd           source=./passwd               $(FILE_PERM)
# file=starttab         source=./starttab             $(FILE_PERM)
# file=hosts            source=$(PROJECT_DIR_PORT)/hosts
      $(FILE_PERM)
 file=dhclient-script   source=$(ENV_PREFIX)/etc/dhclient-script
      $(FILE_PERM)
 file=dhcpd.conf        source=$(PROJECT_DIR_PORT)/dhcpd.conf
      $(FILE_PERM)
 file=inetd.conf        source=./inetd.conf           $(FILE_PERM)
# file=protocols        source=$(ENV_PREFIX)/etc/protocols
      $(FILE_PERM)
# file=services         source=./services             $(FILE_PERM)
# file=resolv.conf      source=./resolv.conf          $(FILE_PERM)
# file=hosts.equiv      source=./hosts.equiv          $(FILE_PERM)
# file=fstab            source=./fstab                $(FILE_PERM)

directory=/etc/rc.d
 file=rc    source=$(PROJECT_DIR_PORT)/rc $(BIN_PERM)

directory=/net                                        $(DIR_PERM)
# file=inetd      source=$(ENV_PREFIX)/net/inetd      $(BIN_PERM)
 file=rc.network  source=/tmp/newproj/60.developer/rc.network
$(BIN_PERM)
# file=rtu_server source=./rtu_server                 $(BIN_PERM)
 file=rtuDevice   source=./rtuDevice                 $(BIN_PERM)
 file=rtu         source=./rtu                       $(BIN_PERM)
 file=rtuDevice.h source=./rtuDevice.h               $(BIN_PERM)
# file=rshd       source=$(ENV_PREFIX)/net/rshd       $(BIN_PERM)
# file=telnetd    source=$(ENV_PREFIX)/net/telnetd    $(BIN_PERM)
# file=routed     source=$(ENV_PREFIX)/net/routed     $(BIN_PERM)
# file=ftpd       source=$(ENV_PREFIX)/net/ftpd       $(BIN_PERM)
# file=rlogind    source=$(ENV_PREFIX)/net/rlogind    $(BIN_PERM)
# file=irshd      source=$(ENV_PREFIX)/net/irshd      $(BIN_PERM)
# file=portmap    source=$(ENV_PREFIX)/net/portmap    $(BIN_PERM)
# file=rc.local   source=./rc.local                   $(BIN_PERM)
# file=unfsio     source=$(ENV_PREFIX)/net/unfsio     $(BIN_PERM)


directory=/mnt          $(DIR_PERM)
```

```
directory=/tmp          $(TMP_PERM)
directory=/usr/tmp      $(TMP_PERM)

symlink /bin/sh /bin/bash
symlink /etc/rc.d/rc /bin/rc
```

APPENDIX II: Rc.network File

The rc.network file contains scripts associated with running networking features at system startup. The calls to the rtuDevice and rtu processes are contained within this file. Extraneous comments have also been removed from this file to reduce length and improve readability. This file was modified from the rc.network file included with the pre-build developer KDI.

```
#
#/bin/sh
#
# $Header: /cm/src/net/inetd/RCS/rc.network,v 5.46 2002/01/29 23:37:08
mooring Exp $

# A little start up routine to capture errors.
start_it()
{
      echo "Running $1"
      $@ || echo "    failed with exit code $?"
}

#################################################################
#     Network startup procedure
#################################################################
#

# 'cuz of the bash propensity to want a TERM variable at all costs,
# and the fact that the networking daemons really don't want
# one, we nuke it right here and now.

unset TERM

cd /net

#
# Configure the software loopback device "lo" with hostname "localhost"
#
/bin/ifconfig lo0 localhost

#
#     my_name is used for Ethernet interfaces
#     bplane_name is used for the SCMP interface (only on 68k, PowerPC)
#
my_name="lynxdemo"
#bplane_name="insert-your-bplane-name-here"

#     Change hostname appropriately to coincide with /etc/hosts
#
```

```
#     NOTE: If there is only the SCMP interface available, change
#     "my_name" in the next line to "bplane_name".
#
hostname "$my_name"
#echo "hostname is `hostname`"


#####################################################################
#     This starts the 3COM EtherLink XL 3C90X
start_it /bin/ifconfig elxl0 up
dhclient

echo Network interface configured

#[ -f /etc/syslog.conf ] &&\
#start_it /bin/syslogd

#
#     Start network daemons
#
#start_it /net/inetd

#
#     Start RTU Server
#
#start_it /net/rtu_server
/net/rtuDevice &
start_it /net/rtu


#####################################################################
#
# Commands for NFS Server support
# Edit the /etc/exports file to specify the directories that are to be
# exported for mounting by remote machines.
#

#start_it /net/portmap

#start_it /net/mountd

#[ -s /etc/exports ] &&\
#echo exporting directories for remote mount &&\
#start_it /bin/exportfs -av

#start_it /net/nfsd

#start_it /net/rpc.statd

#start_it /net/rpc.lockd.svc

#start_it /net/rpc.lockd.clnt

# Source a local script if it's there
#  Here's a very good place to start xntpd

#[ -x /net/rc.local ] && . /net/rc.local
```

APPENDIX III: Low-level SConstruct

This is the SConstruct file used to include the rtu.c code in the build conduced by the Kenge build environment. This was modified from a similar SConstruct file included with the Hello project from ERTOS.

```
# SConstruct file for rtu.c module
# Modified from file included in Hello program by ERTOS
Import("*")
obj = env.MyProgram("rtu",
                    LIBS=["c"],
                    CPPDEFINES=[("RTU", "\\\"%s\\\"" %
args["phrase"])])
Return("obj")
```

APPENDIX IV: High-level SConstruct

This SConstruct file is used by the Kenge build system to link together all modules with the kernel to form a single bootable image, which can then be simulated or used as a system boot option. This file was also modified from a similar SConstruct file included in the ERTOS Hello project.

```
# High level SConstruct
# modified from Hello project by ERTOS

# First step is to include our real build tools
# tools/build.py includes the KengeEnvironment
try:
    execfile("tools/build.py")
except IOError:
    print
    print "There was a problem finding the tools directory"
    print "This probably means you need to run:"
    print "  $ baz build-config packages"
    print
    import sys
    sys.exit()

# phrases detoning language
phrases = {
    "english" : "Hello, world",
    "dutch":   "Hello, wereld",
    "german":  "Hallo, Welt",
    "french":  "Bonjour, monde",
    "italian": "Ciao, mondo",
    "spanish": "Hola, mundo"
    }

add_config_help("Options:\n")
add_config_list("lang", "Which language do you want to compile for",
"english", phrases.keys())

# setup the build environment
env = KengeEnvironment()

# specify the kernel to use as "pistachio"
l4kernel_env  = env.Copy("kernel")
#l4kernel = l4kernel_env.Pistachio()
l4kernel = l4kernel_env.Application("pistachio")

# Add support libraries for rootserver
rootserver_env = env.Copy("rootserver", LINKFLAGS=["-r", "-N"])
rootserver_env.AddLibrary("l4")
rootserver_env.AddLibrary("c", system="l4_rootserver")
```

```
# setup the applications to run
sigma0 = rootserver_env.Application("sigma0")
rtu = rootserver_env.Application("rtu", phrase=phrases[lang])

# combine the kernel and applications
#  into a single boot image
bootimage = env.Bootimage(l4kernel, sigma0, rtu)

Default(bootimage) # This is the default build target
```

APPENDIX V: Rtu.c

This code simulates some simple functions of an RTU, such as outputting data and measuring transmission time. It is simply used as a stub to support the proof of concept of a Pistachio based RTU.

```
/* rtu.c */

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

uintptr_t _stack[128];
void * _stack_top = (void*) &_stack[255];
char bytes[30];
char press;
char escape;

int main(void)
{
      escape = 'x';
      press = 'y';
      for(int i = 0; i< 30; i++)
      {
            bytes[i] = 0x61;
      }
      int i = 3;
      time_t t1, t2;
      printf("Press any key to begin.\n");

      while (press != escape)
      {
          press = getchar();

          t1 = time(NULL);

          bytes[i]=bytes[i]+i;

          printf("RTU: ");
          printf("%d\n",i);
          printf("0x%x\n",bytes[i]);
          i = (i + 771) % 30;

          /*insert some delay to account for transfer time*/
          for(int j=0; j<999; j++)
          {
            (void) time(&t2);
          }

          printf("Time: %d\n", (int) t1);
```

```
            printf("Press x to exit or any other key to continue.\n");
        }

        printf("Goodbye!\n");

        return 0;
}
```

APPENDIX VI: Sample Output for Rtu.c

Listed below is some sample output created by booting the project image, which will run the rtu.c program after the kernel and all other necessary tasks have loaded. Notice the time is always output as 0. This is because Pistachio does not support the system calls and variables that allow the code in the time.h header to work properly. Also note the code prompts the user to continue or exit after each output of a generated value. This was done to give the user a feel for the execution time and input for the Pistachio based system and could easily be modified to run more autonomously. The value after RTU: is the current, pseudo-randomly generated index of the byte array that stores all hexadecimal output values. The hexadecimal values listed are the pseudo-randomly generated output values.

```
RTU: 3
0x64
Time: 0
Press x to exit or any other key to continue.
RTU: 24
0x79
Time: 0
Press x to exit or any other key to continue.
RTU: 15
0x70
Time: 0
Press x to exit or any other key to continue.
RTU: 6
0x67
Time: 0
Press x to exit or any other key to continue.
RTU: 27
0x7c
Time: 0
Press x to exit or any other key to continue.
```

APPENDIX VII: Timing_log.txt

The following is sample output of the latency measurements of communication between the RTU and MTU programs. The value denoted by the '*' indicates an outlier, and is likely caused by a dropped packet during transmission. The data has been formatted in a table for readability. Measurements are in milliseconds.

1.659708147	1.542095434	1.566679564	1.557460515	1.538184322
1.564444643	1.552431943	1.550476387	1.635962113	1.652444654
1.554387499	1.55662242	1.536508132	1.553549404	1.534552576
1.558019245	1.552152578	1.735416093	1.560254166	1.611657348
1.517511304	1.559416071	1.574781152	1.557460515	1.564444643
1.523657336	1.55662242	1.550197022	1.65440021	1.559416071
1.517231939	1.552152578	1.636800208	1.512203367	1.538463687
1.514158922	1.559416071	1.554108134	1.604393855	1.553549404
1.514717653	1.611657348	1.554666864	1.643225605	1.65384148
1.504660509	1.56807639	1.520863685	1.561371627	1.570311311
1.656914496	1.552711308	1.561371627	1.554108134	1.558019245
1.579530359	1.619479571	1.672000212	1.669485926	1.566400199
1.536787497	1.672838308	1.565841469	1.539022418	1.553828769
1.565841469	1.549358927	1.738768475	1.554108134	1.647974812
1.567797024	1.634285922	1.662781164	1.55606369	1.689041484
1.552711308	1.554108134	1.569473215	1.55773988	1.555504959
1.55718115	1.561092262	1.550476387	1.563047818	1.550755752
1.551314483	1.552711308	1.56919385	1.555784325	1.573663692
1.553549404	1.66641291	1.56807639	1.641549415	1.552990673
1.651047829	1.560254166	1.553549404	1.560812897	1.656355766
1.641549415	1.661943068	1.559974801	1.555784325	1.559695436
1.563885913	1.554946229	1.554108134	1.551314483	1.572266866
1.65495894	1.554666864	1.557460515	1.554108134	1.557460515
1.540698608	1.60467322	1.739327205	1.563047818	1.808051023
1.544889085	1.552152578	1.565282738	1.547962101	1.56807639
1.66529545	1.541816069	1.556901785	1.561092262	1.558857341
1.555784325	1.550197022	1.561371627	1.556343055	1.651885924
1.552431943	1.549638292	1.549917657	1.55606369	1.515835113
1.563047818	1.75106054	1.559974801	1.743238317	1.572266866
1.552711308	1.564165278	1.55662242	1.575060517	1.54405099
1.639873224	1.554666864	1.550755752	1.555504959	1.563606548
1.558298611	1.652724019	1.53259702	1.614171634	1.559695436
1.553270039	1.550755752	1.559974801	1.548241466	1.661104973
1.656076401	1.559416071	241.9408053 *	1.569473215	1.625904968
1.561092262	1.552711308	1.570590676	1.566679564	1.521981146

APPENDIX VIII: List of Acronyms

Acronym	Meaning
API	Application Programming Interface
COTS	Commercial Off The Shelf
DHCP	Dynamic Host Configuration Protocol
DNP-3	Distributed Network Protocol version 3
EAL	Evaluation Assurance Level
ERTOS	Embedded, Real-time, and Operating Systems
IPC	Inter-Process Communication
KDI	Kernel Downloadable Image
LOC	Lines of Code
MILS	Multiple Independent Levels of Security
MTU	Master Terminal Unit
NEAT	Non-bypassable, Evaluatable, Always Invoked and Tamperproof
NICTA	National Information Communication Technology Australia
PCS	Partitioning Communication System
POSIX	Portable Operating System Interface
RTOS	Real Time Operating System
RTU	Remote Terminal Unit
SCADA	Supervisory Control and Data Acquisition
TCP/IP	Transmission Control Protocol/Internet Protocol

VITA

NAME: Brent Guffey

ADDRESS: Department of Computer Engineering and Computer Science
University of Louisville
Louisville, KY 40292

DOB: Somerset, KY – July 5 1982

EDUCATION &
TRAINING: B.S. Computer Engineering and Computer Science
University of Louisville
2001-2005

EMPLOYMENT: CECS Co-op at Bardstown Cable Internet
Bardstown, KY
2001-2003

Software development position at Keane, Inc.
Frankfort, KY
2006-

AWARDS: Recipient of Provost Hallmark Scholarship, Graduated with High Honors