November 2018

A Secure Computing Platform for Building Automation Using Microkernel-based Operating Systems

Xiaolong Wang
University of South Florida, xiaolongw@mail.usf.edu

Follow this and additional works at: https://scholarcommons.usf.edu/etd

Part of the Computer Sciences Commons

Scholar Commons Citation

This Dissertation is brought to you for free and open access by the Graduate School at Scholar Commons. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Scholar Commons. For more information, please contact scholarcommons@usf.edu.
A Secure Computing Platform for Building Automation Using
Microkernel-based Operating Systems

by

Xiaolong Wang

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Ximning Ou, Ph.D.
Jarred Ligatti, Ph.D.
Srinivas Katkoori, Ph.D.
Nasir Ghani, Ph.D.
Siva Raj Rajagopalan, Ph.D.

Date of Approval:
October 26, 2018

Keywords: System Security, Embedded System, Internet of Things, Cyber-Physical Systems

Copyright © 2018, Xiaolong Wang
DEDICATION

In loving memory of my father,

to my beloved, Blanka, and my family.

Thank you for your love and support.
ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Dr. Xinming Ou for his guidance, encouragement, and unreserved support throughout my PhD journey. Dr. Xinming Ou introduced me to the field of security, and helped me identify important security problem in the domains of Cyber-Physical Systems and the Internet of things. His thorough scientific approach and research philosophy – “start from real problems, create solutions that last” - have been inspirational in the years of my research. He has set an example of excellence as a researcher, mentor, and role model.

I would like to thank Dr. Raj Rajagopalan for his insightful ideas, feedback, and inspiring discussions that helped me shape this research; Dr. Masaaki Mizuno, Dr. Mitchell Neilsen for their guidance and support with their tremendous knowledge in microkernel and embedded systems; and Dr. John Hatcliff for his advice and vision in the design of high assurance systems. Without them I would not have been able to carrying out this work.

I’m deeply indebted to Dr. Jarred Ligatti, Dr. Srinivas Katkoori and Dr. Nasir Ghani for serving on my dissertation committee. This dissertation would not be possible without their numerous constructive feedback and invaluable inputs and suggestions.

Finally I would like to express my gratitude to my fellow Argus Cyber Security Lab colleagues for the many memorable times, and fruitful discussions. I’m grateful to Richard Habeeb, who has closely worked with me in the past 3 years, and responsible for implementing and testing the seL4 version. With his help I could not possibly be finished with my thesis by now. I would also like to thank my wife, Blanka, for proofreading drafts of this thesis.

This research was supported the Department of Homeland (DHS) Security under contract number D15PC00303, and National Science Foundation under Grant No. 1136040.
## TABLE OF CONTENTS

### LIST OF TABLES

| LIST OF TABLES | iii |

### LIST OF FIGURES

| LIST OF FIGURES | iv |

### ABSTRACT

| ABSTRACT | v |

### CHAPTER 1: INTRODUCTION

| 1.1 Building Automation Systems | 1 |
| 1.2 Safety and Security Challenges for BAS | 2 |
| 1.3 Existing BAS Security Solutions | 4 |

### CHAPTER 2: VISION

|  | 7 |

### CHAPTER 3: CASE STUDY

| 3.1 BRI Scenario | 9 |
| 3.1.1 Biocontainment Facility | 9 |
| 3.1.2 BRI Policies | 10 |
| 3.1.3 Airflow and Air Pressure Control | 13 |
| 3.1.4 Other Control Tasks | 15 |
| 3.2 Smoke Evacuation Scenario | 16 |
| 3.2.1 Smoke Evacuation Policies | 16 |
| 3.2.2 Daily Operation | 17 |
| 3.2.3 Smoke Evacuation | 19 |

### CHAPTER 4: BUILDING AUTOMATION SECURITY ISSUES

| 4.1 Building Automation Network | 21 |
| 4.1.1 Building Automation Network Protocols | 22 |
| 4.1.2 Field Sensors and Actuators in BAS | 23 |
| 4.1.3 Automation Controllers in BAS | 24 |
| 4.1.4 Management in BAS | 25 |
| 4.2 Security of Building Automation Network | 26 |
| 4.3 Security of BAS Controllers | 27 |
| 4.4 Security of BAS Management | 29 |
| 4.5 BAS Threat Summary | 30 |

### CHAPTER 5: DESIGN OVERVIEW

| 5.1 Microkernel Architecture | 32 |
| 5.1.1 Microkernel History | 33 |
| 5.1.2 Microkernel Design | 34 |
LIST OF TABLES

Table 4.1  Known potential threats 31
Table 7.1  IPC RTT for Linux and MINIX 3 on a BeagleBone Black Board 58
Table 7.2  Network RTT for Linux and MINIX 3 on a BeagleBone Black Board 59
Table 8.1  Lmbench results for seL4 microvisor on TK1 clocked at 696 MHz 76
**LIST OF FIGURES**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 2.1</td>
<td>Envisioned building security/safety control framework</td>
<td>8</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>Biocontainment laboratory scenario layout</td>
<td>12</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>Biocontainment laboratory scenario structure</td>
<td>12</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>Airflow and air pressure control</td>
<td>14</td>
</tr>
<tr>
<td>Figure 3.4</td>
<td>HVAC control system</td>
<td>18</td>
</tr>
<tr>
<td>Figure 5.1</td>
<td>Microkernel vs. monolithic kernel</td>
<td>35</td>
</tr>
<tr>
<td>Figure 5.2</td>
<td>Each controller enforces global policy locally, allowing process A to communi-</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>cate with process B over remote proxies</td>
<td></td>
</tr>
<tr>
<td>Figure 6.1</td>
<td>Fine-grained IPC using access control matrix</td>
<td>44</td>
</tr>
<tr>
<td>Figure 6.2</td>
<td>Three devices, four process scenario demonstrating the usage of cryptographic</td>
<td>48</td>
</tr>
<tr>
<td></td>
<td>tunneling between the proxies</td>
<td></td>
</tr>
<tr>
<td>Figure 7.1</td>
<td>Single device temperature control scenario logic structure</td>
<td>50</td>
</tr>
<tr>
<td>Figure 7.2</td>
<td>Temperature control testbed</td>
<td>51</td>
</tr>
<tr>
<td>Figure 7.3</td>
<td>Single device temperature control scenario process communication</td>
<td>52</td>
</tr>
<tr>
<td>Figure 7.4</td>
<td>Multiple device temperature control scenario logic structure</td>
<td>52</td>
</tr>
<tr>
<td>Figure 7.5</td>
<td>Single device temperature control scenario process communication</td>
<td>53</td>
</tr>
<tr>
<td>Figure 8.1</td>
<td>Secure execution environment using seL4 microvisor</td>
<td>68</td>
</tr>
<tr>
<td>Figure 8.2</td>
<td>Cloud connector in conventional systems</td>
<td>73</td>
</tr>
<tr>
<td>Figure 8.3</td>
<td>Cloud connector in secure execution environment</td>
<td>74</td>
</tr>
</tbody>
</table>
ABSTRACT

Building Automation System (BAS) is a complex distributed control system that is widely deployed in commercial, residential, industrial buildings for monitoring and controlling mechanical/electrical equipment. Through increasing industrial and technological advances, the control components of BAS are becoming increasingly interconnected. Along with potential benefits, integration also introduces new attack vectors, which tremendous increases safety and security risks in the control system. Historically, BAS lacks security design and relies on physical isolation and “security through obscurity”. These methods are unacceptable with the “smart building” technologies. The industry needs to reevaluate the safety and security of the current building automation system, and design a comprehensive solution to provide integrity, reliability, and confidentiality on both system and network levels.

This dissertation focuses on the system level in the effort to provide a reliable computing foundation for the devices and controllers. Leveraged on the preferred security features such as, robust modular design, small privilege code, and formal verifiability of microkernel architecture, this work describes a security enhanced operating system with built-in mandatory access control and a proxy-based communication framework for building automation controllers. This solution ensures policy-enforced communication and isolation between critical applications and non-critical applications in a potentially hostile cyber environment.
CHAPTER 1: INTRODUCTION

1.1 Building Automation Systems

Building Automation Systems (BASs) are large-scale distributed control systems that aim to improve autonomous control and the customized management of building environments. A typical BAS controls every aspect of a building’s operation from lighting, heating, ventilation and air conditioning (HVAC), to fire safety, access control, video, and more. BAS operations directly impact the energy consumption of building facilities. According to the Department of Energy, buildings account for about 40% of total energy consumption and 75% of electricity consumption in the U.S., and those numbers are increasing each year [1]. Naturally, there is growing motivation to develop new technologies for BASs to improve occupant comfort and at the same time reduce energy consumption and operating costs.

Traditionally, BASs consist of multiple standalone application-specific subsystems. Each subsystem only provides a single service, such as temperature control, ventilation, etc. Devices in each subsystem are connected through building automation networks (BANs), which used to be separated from the IT systems in the environment. The control logic is managed through central management servers that are called Building Management Systems (BMSs) that connects with Programmable Logic Controllers (PLCs) in a master-slave pattern. Different subsystems are coupled into different control domains (e.g., safety domain, security domain), and are managed separately.

With the rapid commercial expansion of Cyber-Physical Systems (CPSs) or popularly known as the Internet of Things (IoT) in industry, developers and researchers have sought to embrace the concept of “smart buildings”, which leverages various sensors to better understand living context, provides IP-enabled actuators to control the environment, and interconnects existing control systems to be more responsive to customized needs. Buildings are undergoing a transformation to better serve customers and occupants through advanced automation and networking. This, in turn,
requires integrating increasingly sophisticated computing and networking capabilities within BASs; the robustness and stability of which is crucial to support occupants’ daily routine and ensure both security and safety for normal operations as well as during emergencies. These advances have made life easier for occupants and helped developers find new ways to, e.g., reduce energy consumption.

In order to achieve the goal of energy conservation, a tight integration among sensors, controls, and intelligence is indispensable. Today, a state-of-the-art BAS integrates multiple subsystems at the device level. This integration not only means interconnecting these existing systems but also means the enabling of remote control and data acquisition. Because of these changes, cloud-based machine learning solutions and Internet-accessible management systems are starting to become common in BASs. Although, benefits, such as agile reaction and cascaded control are unarguable, like other domains that are embracing the CPS perspective (such as smart grid, distributed robotics, autonomous vehicles and avionics systems), the development of BAS as a system of systems is now facing multiple obstacles to progress, particularly the nexus of safety-security and the reliability challenges associated with the threat of cyber attacks.

1.2 Safety and Security Challenges for BAS

The integration between the control network and IT network dramatically increases the attack surface of BAS, offering adversaries more opportunities to attack than ever before. Cyber attacks not only increase the energy consumption by tampering with carefully laid plans, but also can alter the functionalities of control components, thereby impacting the overall safety of the buildings. The lack of appropriate interoperability and secure architecture standards leaves potential control hazards and threats in play, especially for safety critical facilities.

BASs are already connected to the Internet. As shown by Billy Rios at Black Hat USA 2014 [2], there are 21,000 Tridium Niagara Systems (one of the most popular platforms for building control) connected to the Internet. It has been identified in over 50,000 buildings which are exposed to the Internet either intentionally or due to misconfiguration and can be publicly searched using tools such as Shodan [2,3]. Besides, BAS widely uses outdated low-level protocols, such as BACnet, KNX, Modbus, which send data in plaintext and lack proper authentication mechanisms. As previous works [4,5] show, attackers can easily sniff control packets, modify Programmable Logic Controller
(PLC) arbitrarily, and use carefully crafted low-level data gathered through PLCs to inject high-level control software. Moreover, many commercial off-the-shelf BAS, like MetaSys and Niagara are based on outdated Windows operating systems. The discovery of the first cyber-weapon, Stuxnet [6], suggests that specifically crafted malware can be easily launched against control networks such as BAS to sabotage high security-concerned institutions.

Recent high-profile attacks have demonstrated the possible threats and potential impacts of cyber attacks in building environments. For example, in 2013 researchers infiltrated Google’s Australia office through their building automation network [7]. Mirsky et al. have created a proof-of-concept malware, HVACKer, that demonstrated how attackers can use HVAC systems to bridge air-gapped networks with the outside world [8]. While the U.S. Department of Energy works together with alliances launching a new Net Zero Energy Installation (NZEI) initiative for achieving self-sustaining buildings, building infrastructure is integrating with different internal and external infrastructures [9]. Hacking into these systems have also been shown to be trivial. For example, in winter 2016, a rookie hacker launched a Distributed Denial of Service (DDoS) attack against an HVAC system which resulted in the loss of heating for two buildings in Finland [10]. Such attacks not only can help attackers gain control of BAS, but might also provide a stepping stone for further infiltration of other critical infrastructures such as power, water, transportation, health service, etc.

The challenge of securing building automation has multiple dimensions. First, buildings are heterogeneous. Buildings are designed for different purposes, and therefore have distinct requirements. For example, the automation system for a stadium might focus on how to manage lighting, temperature, and increased ventilation to maintain low carbon dioxide concentration. On the other hand, the main concern for a biocontainment facility or a chemical plant could be minimizing air exchange between different areas to reduce the risk of cross-contamination. The design of the next generation BAS has to consider the details of the varying use cases and understand the distinct requirements therein.

Secondly, a BAS is hierarchical. Different sub-hierarchies have different views of the system. Where and how each sub-hierarchy enforces the various policies has a significant impact on the overall security, safety, and reliability of the building. It is vital to have a global view of the
system from top to bottom with a proper abstraction. Formalizing the various security and safety requirements and analyzing threats from a global view of the system, and designing policies and appropriate security mechanisms to uphold them should be a key step in BAS design.

Last but not least, the interoperability of BAS is challenging. Not only are highly integrated subsystems openly connected through networks but they are also interacting through the physical features that they control (e.g., open doors change airflow patterns). When designing a BAS, it is often impossible to completely isolate one control system from another.

Considering the tremendous upcoming changes in next-generation BAS, and the potential risks it will cause, it is critical to rethink and reevaluate how the building automation systems are designed and organized together. In order to provide the necessary safety/security guarantees in the distributed environment, security cannot be an afterthought, but should be one of the critical considerations of the entire design and implementation process. The security evaluation includes not only mechanisms for secure network communication, authentication, but should also include the embedded platforms, system architecture, operating systems etc. Furthermore, for practical reasons, the legacy subsystems and standard industrial control protocols in existing BASs have to be considered.

1.3 Existing BAS Security Solutions

The related work of this dissertation can be summarized in three categories: (1) the studies of safety and security in building automation system; (2) secure operating systems for embedded devices; (3) secure virtualization for embedded systems.

The security and safety issues of building automation systems is a broad research topic and although still in its early stages, has attracted a lot of attention from academia. For example, the Automation Systems group at the Vienna University of Technology (TU Wien) has been working on security and safety related topics on home and building automation. W.Granzer et al.’s works on BAS security issues provides good insight into the security challenges in BAS [11, 12, 13]. They studied safety and security threats at each level of the system, and proposed a security extension on the KNX/EIB protocol. Much of their work focuses on network protocol related security. In a 2009 paper, they discuss the use of cryptographic technology to secure building automation com-
communication over IP networks [14]. This method is very similar to the network tunneling approach, I proposed in this work. However, in this work the network tunneling approach is part of the proxy-based network communication in an attempts to apply communication policies in existing BAS.

Furthermore, S. Dünhaupt’s work on a study of vulnerabilities in industrial automation systems has systematically reviewed the known security attack vectors in automation systems in general [15]. F. Praus et al. in 2016 provided an extensive survey of the security requirements for distributed control applications in BAS [16]. This work discussed various security protection techniques, such as static analysis, formal verification etc. and proposed a secure control application architecture at a high level. This work shares a similar vision with those researchers. However, this work views the safety and security issues from a system perspective, and focuses on securing critical control applications, enforcing global policies, through trustworthy operating systems. To my best knowledge, this work is the first attempt to design a secure computing foundation for the next-generation BAS embedded devices.

Over the years many efforts have been carried out in the design of secure operating systems. One of the most widely-adapted ones is Security-Enhanced Linux (SELinux) [17]. It has been applied in Android OS and lives in millions of commercial smart phones [18]. SELinux enforces mandatory access control by monitoring and checking process operations against kernel-stored policy. SELinux provides comprehensive process control and flexible authorization models. It leverages the Linux Security Module (LSM) hooks to implement checks in the Linux kernel. However, SELinux resides in the monolithic Linux kernel. The effectiveness of the reference monitor hinges on whether the whole Linux kernel, including that all the device drivers are free of security bugs. The underlying Linux OS adopts a monolithic kernel architecture. Moreover, due to the complexity of Linux, the system requires complex protection states representation that involves extensive amounts of explicitly defined type labels and policies to express all the necessary access relationships [19]. As a result it is hard, if not impossible, to understand and reason about the correctness of a SELinux policy. The proposed approach takes advantage of the microkernel architecture. The modular design in microkernel architectures reduces the chance of kernel space vulnerability. Also, since the policy
only needs to concern the IPC primitive, the policies are much simpler and easier to reason about. This better suits the embedded system domain. Additionally, there are many microkernel-based OSs that have been proposed. Many of them belong to the L4 family, such as L4Ka::Pistachio [20], OKL4, etc. but when it comes to availability and portability to ARM architecture, MINIX 3 and seL4 show great advantages, hence this work choose those two. Recently R. Gu et al. proposed a certified kernel, CertiKOS. Although in its early stage it shows another promising approach [21].

With advancement in hardware, there have been many proposals of microkernel-based hypervisor. Among those microvisor, the NOVA, L4Re, OKL4 stands out. The NOVA is based on a third-generation microkernel with 9000 lines of code in kernel space [22]. The NOVA hypervisor uses a similar capability-based security model and implements VMM in userspace. However, NOVA applies instruction emulation approach for virtualization, which complicates the VMM design. Furthermore, NOVA is currently only available for X86 architecture. L4Re is another microkernel-based operating system framework that focuses on real-time schedulability [23]. Similar to seL4 microvisor, L4Re provides isolated user-land domains. L4Re supports both para-virtualization and hardware assisted virtualization with multiple platforms, including X86, ARM, and MIPS. Another one is the OKL4 microvisor, which is is the predecessor of seL4. OKL4 supports para-virtualization and has been optimized for high performance for commercial usage with mature VMM and mature driver frameworks [24]. Each of these have their own benefits, however, when it comes to security none of them can provide the same level of guarantee of functional correctness and isolation, as the seL4.
CHAPTER 2: VISION

Buildings are containers of many different types of human activities supported by various types of technologies. The type and utility of a building determine its unique security/safety needs. The lack of appropriate interoperability and secure architecture standards leaves potential control hazards and threats in play, especially for safety critical facilities. On the other hand, buildings are heterogeneous. Different buildings are designed for various purposes, therefore, have distinct requirements. For example, the automation system for a stadium might focus on how to manage lighting, temperature, and increased ventilation to maintain low carbon dioxide concentration, while the main concern for a bioccontainment facility or a chemical plant could be minimizing air exchange between different areas to reduce the risk of cross-contamination.

Instead of relying upon traditional security approaches such as perimeter control and continuous monitoring and patching, we believe that embedded controllers need to adopt a radically new architectural approach to safety and security, so that core primitives supporting those properties can be built in as opposed to being bolted on later. Due to the innate dynamics of building environments, the controllers must be able to guarantee reliable safety behavior under hostile conditions, even when part of it are untrusted or compromised. Figure 2.1 shows the envisioned solution framework. The very important first step is to research the multitude of security/safety issues that may arise and properly model them so each embedded controller in the environment has clearly defined security/safety requirements it has to uphold. Based upon scenario analysis, we can construct models of the specific building automation system, and extract safety/security constraints that should be enforced by different layers of a system running on those embedded controllers. Finally, build a layered architecture for embedded platforms so that applications can use these properties to achieve security and safety specifically for their contexts.
Based on the above positions, we envision a microkernel-based OS forms the core of the embedded controller platform, due to the advantages of microkernel mentioned in chapter 1. The core, with underlying hardware support, provides the following functionalities: (1) Process isolation (control logic integrity); (2) scheduling with real-time constraint (responsiveness and availability); (3) Inter-process communication and synchronization (data flow integrity and authenticity); (4) Network communication (secure communication, confidentiality).
CHAPTER 3: CASE STUDY

In order to fully understand the nature of building automation, analyze safety/security risks, and balance the trade-offs between usability and security, we conducted case studies on two realistic scenarios equipped with state-of-the-art building automation systems. One is for the automation control system of a biocontainment laboratory that is modeled according to the field study of Biosecurity Research Institute (BRI) at Kansas State University. The other is for an HVAC smoke evacuation control system based on a campus building at the University of South Florida. For safety consideration, all examples discussed in this paper are assumptive. This section documents the mock-up scenarios.

3.1 BRI Scenario

BRI is a unique biocontainment research and education facility recently built at Kansas State University. It is designed to conduct research on highly infectious animal diseases, pathogens, and pests that may cause serious or potentially lethal disease in humans through inhalation. As a CDC approved BSL-3 (biosafety level 3), ABSL-3 (animal biosafety level 3), and BSL3-Ag (biosafety level 3 agriculture) facility, BRI follows strict federal regulations and has a set of rigorous requirements for building management policies. The proper management of the facility must protect laboratory workers, persons outside of the laboratory, and people and/or animals in the neighbor community from infectious agents that could be accidentally released from the laboratory.

3.1.1 Biocontainment Facility

An important use case of BAS is one for controlling and monitoring a biocontainment facility. Biocontainment facilities are designed to do research on infectious diseases, pathogens, quarantined pests, invasive alien species and living organisms [26]. The nature of this kind of laboratory makes

---

1Part of this chapter is published in ACM CCS 2015 Workshop on Cyber-Physical Systems Security and Privacy (CPS-SPC2015) [25] paper. Permission is included in Appendix A.
it highly hazardous. Since some samples might be contagious for human beings, once reaching the public space it could cause tremendous danger that even jeopardize human lives. In this case, protecting just the worker is oftentimes not enough. Systems must also be in place to protect the environment and the facility from possible contamination. In the United States, the Centers for Disease Control and Prevention (CDC) have specified different biosafety levels of biocontainment precautions. They range from the lowest level 1 (BSL-1) to the highest at level 4 (BSL-4) for isolating dangerous biological agents in an enclosed laboratory facility [27]. Biosafety levels increase safety and security measures as the dangers associated with the agents’ increase. For example, BSL-3 is applicable to clinical, diagnostic, teaching, research, or production facilities in which work is done with indigenous or exotic agents. The agents may cause serious or potentially lethal disease after inhalation. BSL-4 is specifically for fatal diseases such as the Marburg or Ebola viruses [27].

For guiding the design and maintenance of laboratories, the CDC and the U.S. Department of Agriculture - Agricultural Research Service (USDA-ARS) have published Biosafety in MicroBiological and Biomedical Laboratories (BMBL) and USDA-ARS Facilities Design Standards respectively. They define various policies and standards for HVAC, air pressure, temperature, and the procedure of handling special situations [27, 28]. Moreover, The National Institutes of Health (NIH) details design requirements and guidance manuals for biomedical research facilities [29]. In all of them, BAS is regarded as critical and responsible for enforcing biosafety level and personnel security.

3.1.2 BRI Policies

In BRI, one of the high-priority concerns is the control of airborne hazards and to avoid the potential risk of cross-contamination. A BSL-3 facility generally consists of a separate building with clearly demarcated and isolated zones within the building, each of which is equipped with independent ventilation and waste management systems. The BAS must support multiple institutional policies. According to the Biosafety in Microbiological and Biomedical Laboratories (BMBL) standards published by HHS and CDC, each zone must enforce access control to the laboratories. The laboratories must ensure directional airflow, negative air pressure with respect to its surrounding environment, and complete isolation between each other. Access to the laboratory is through two doors for isolation from unrestricted traffic flow. The laboratory doors must be self-closing and
locked by default. Proper evacuation control for incidents and emergency situations, such as fire and flood, that may result in exposure to infectious materials must be considered.

For the sake of simplicity, in this scenario, we are only considering a suite with five rooms in a biocontainment facility. A biocontainment facility is usually separated into different isolated zones. Each zone is totally independent of the others and has its own ventilation system. In our scenario, the suite is composed of an isolated zone. It contains four laboratories for conducting different research and a chamber as a public area for researchers. As shown in Figure 3.1, rooms start from No. 1 to No. 4 are biocontainment laboratories and room No. 5 is the public chamber. The logic structure of the suite BAS is shown in Figure 3.2. The suite must comply with basic biocontainment laboratory standards. In this case, the suite should only allow limited authorized researchers to have access. Every authentication should be logged for audit purposes. Each laboratory within this suite has to constantly maintain a negative differential air pressure relative to the chamber room in order to avoid airborne hazards exposures. Meanwhile, different laboratories maintain independent air pressure among each other and airborne hazards exposures are prevented through controlling doors by using interlock system installed in the chamber. Each laboratory has to keep the temperature in a certain range. When a laboratory is operational, the fume hood fan in the room has to start working steadily. Furthermore, aside from normal operations, laboratories need to be decontaminated when hazards have been detected as exposed. In order to handle this situation, laboratories usually have two modes: normal mode and decontamination mode (DECON mode). For this purpose, each room has a strobe that is used to indicate emergency situations including power failure, DECON mode, door-open-too-long alarm, etc. When in DECON mode, laboratories would only allow researchers to exit and reject any ordinary user’s entering requests except requests from users with administrator privilege. No matter in what emergency situations, certain functionalities of a laboratory have to work – for example, user authentication and the door access system, directional airflow control, fire alarm, etc. Besides, in order to prevent cross-contamination, no two doors in an isolated zone should be allowed to open at the same time. This is enforced by the magnetic door interlock subsystem, except when the fire alarm is being triggered. When the fire is being detected, the fire control subsystem will override the interlock subsystem and unlock all doors at the same time.
Figure 3.1: Biocontainment laboratory scenario layout

Figure 3.2: Biocontainment laboratory scenario structure
3.1.3 Airflow and Air Pressure Control

According to the policies, the critical responsibility of the BRI BAS is to isolate every laboratory from the others through access and air pressure controls. To provide negative air pressure and prevent unwanted air transfer, proper room pressurization is essential. The design of the overall BRI building automation is largely directed by this goal. To comply with BMBL’s two-door policy, each zone in BRI consists of a number of laboratories and an out-facing chamber. All doors in the zone are self-closing and self-locking. To avoid cross-contamination, all doors are equipped with magnetic interlocks that only allow one door to open at any time. To maintain relative negative air pressure and directed airflow, all rooms in the zone use the Volumetric Airflow Tracking Control strategy (also called offset control) [30]. Airflow tracking preserves the desired relative differential pressure between rooms or spaces by measuring the supply airflow and the exhaust airflow through a Variable Air Volume (VAV) ventilation system and monitoring the exhaust airflow of each fume hood in the laboratory. The control must ensure that the total amount of air constantly exhausted from the room exceeds the amount of air supplied to the room by an offset. This design creates a vacuum effect and causes the air to flow into the room through leakage areas such as door gaps, instead of the other way around.

The airflow tracking is simple to manage and reliable in a relatively static situation. However, it does not guarantee that a specific relative differential pressure value will be reached and maintained. The offset value has to be measured and calculated ahead of time, based on the size of room and leakage area, which is guided by the infiltration curve for pressure difference vs. flowchart, and the related equations [31].

The challenge in maintaining relative differential pressure for laboratories in BRI is related to the door. An open door will instantly increase the leakage area of the laboratory and therefore equalize the air pressures. In practice, while the door is open, it is impossible to maintain an appreciable differential pressure, even when the supply airflow is completely shut off. At the same time, the VAV fume hoods that are equipped in the laboratory also complicate the differential pressure since they are dynamically controlled, and when they are operating, the room airflow changes
rapidly. To prevent the cross-contamination, there is still a need for an additional mechanism to protect and monitor the differential pressure level.

In each laboratory, another control loop, the Cascaded Pressure Control, is applied to monitor the differential air pressure value and adjust the supply and exhaust airflows, especially during entering and exiting the room (Figure 3.2). The cascaded pressure control scenario is similar to the volumetric airflow tracking, but instead of using a fixed offset setpoint for calculating airflows, it also monitors the differential pressure through differential pressure sensors and uses the air pressure data to periodically adjust the airflow tracking offset to maintain the laboratory’s differential pressure at desired level [32].

The chamber controller monitors the differential pressure of all rooms through wired sensors whereas each laboratory controller manages its own airflows. Periodically the differential pressure value is sent to each laboratory controller to reset the offset. The differential pressure data is also sent to a global controller, which is a central control device in a zone deployed for managing and monitoring control tasks. The global controller constantly audits the differential pressure data to detect contamination. Additionally, if required, the cascaded pressure control can be equipped to
monitor the door position sensors. When the door is opened, the door position sensor signals the laboratory controller to switch to a boost exhaust airflow that is pre-measured within the safe range.

Failing to maintain the differential pressure relationship between rooms and spaces for an extended period of time is considered a safety violation due to the potential airborne hazard exposure. Either triggered manually by the managers or automatically detected by the global controller through auditing, the zone will be turned into the decontamination (DECON) mode. In DECON mode, the amber strobe in every room of the zone will be triggered. Occupants will be required to exit the zone following the institutional procedures and the access control will be changed to allow only decontamination experts’ access.

3.1.4 Other Control Tasks

In addition to the airflow and differential pressure control tasks, there are other control tasks that coexist on the same controllers in a zone.

- Security control task, for enforcing access control policies. Each room is locked through a self-closing door and only allows authorized occupants to access. The access control system in each zone is managed by a local security controller and is centrally managed by a physical access server. In operation mode, occupants are required to badge in and badge out, except during decontamination mode, when access will be denied until the hazard is decontaminated.

- Interlock control task, for isolating each laboratory. To avoid cross-contamination between laboratories, every door in the zone is guarded with a magnetic interlock that managed by an interlock controller. It is programmed to enforce that only one door can open at any time, except during the fire incident the magnetic interlock will be disabled for personnel’s safety.

- Mode control task, for coordinating different building operation modes. The mode control is coordinated by the global controller (Network Automation Engine) and room controllers. There are 4 modes for a zone: the unprovisioned mode, which indicates the initialization stage of a zone before the physical environment, such as temperature, air pressure reach to the desired level; the operation mode, which indicates that all laboratories are ready to be used; the fire mode, which signals fire incident; the decontamination (DECON) mode, which
is triggered, when there is a procedural violation or the physical environment drops to an undesired level below threshold.

- Fire alarm control task, for managing fire incident. When a fire incident is being detected, the local room controller is responsible for responding to the fire mode and turning on the amber strobe.

- Temperature control task, for managing temperature in each room. This task controls and monitors the heating coil, the thermostat, and temperature sensor in the room.

The control network for the zone is pictured in Figure 3.2. Each zone is managed relatively independently by a global controller (NAE in the diagram), which is managed by the Building Management System (BMS) and Energy Management System (EMS). In the zone, the access control for all doors is realized by a security controller that managed by the physical access server, while the interlock control is controlled by a single interlock controller with magnetic interlock actuators, and door position sensors that applied to every door. Each room has a local controller for all of the other control tasks with corresponding sensors and actuators. Those controllers communicate with each other through BACnet protocol.

3.2 Smoke Evacuation Scenario

In contrast to the biocontainment facility scenario, university campus buildings such as libraries, residence halls, student centers, etc., are designed to provide easy and comfortable public access for large groups and to conduct various teaching and research activities. In campus buildings, isolation of spaces and restriction of unauthorized access are not big concerns. Faculty, staff, and students spend a substantial amount of their waking hours inside university buildings. Improving the safety and the comfort level of occupants and the energy efficiency of campus buildings are among the top priorities of the BASs.

3.2.1 Smoke Evacuation Policies

The main goal of HVAC management for campus building BAS is to efficiently increase the air exchange. Unlike in the BRI scenario where the supply air is completely from outside air due to
biosafety considerations, for a campus building, it is more energy efficient to reuse the circulating air as much as possible. Normally the fresh outside air pulled into the building has to be processed to remove excess moisture and cooled/heated to a set temperature before being supplied to the building. Cooling, heating, and dehumidifying the air accounts for a large portion of the total cost of HVAC operation, especially in the southern U.S. states where the weather tends to be hot and humid. Therefore, reducing the need of processing outside air by cleaning and re-using the inside air is an important energy saving technique.

Reusing recirculating air increases the indoor carbon dioxide (CO₂) level. CO₂ is a natural component of air. The normal outside atmosphere contains about 300 to 500 parts per million carbon dioxide [33]. The American Society of Heating, Refrigeration and Air Conditioning Engineers (ASHRAE) standard recommends 1000–1200 ppm CO₂ for indoor sedentary spaces [31]. Elevated CO₂ concentrations can cause drowsiness, lethargy, and sick building syndrome. A recent study has shown a direct link between indoor carbon dioxide level and cognitive function [34]. Hence, maintaining a comfortable level of CO₂ concentrations in campus buildings, where large groups of people are constantly exhaling carbon dioxide, is critical.

For high-density occupancy zones, emergency evacuation during fire and smoke is another challenge. The HVAC management subsystem of BAS is responsible for detecting such incidents, reducing potential damage, and facilitating the evacuation procedures for public safety and security, by measures such as opening or unlocking fire exit doors, turning on the fire alarm bullhorns, and reducing fresh air supply to the area on fire.

3.2.2 Daily Operation

In this university building scenario, there are six controllers – the outside air controller, the returning air controller, the air handling unit (AHU) controller, the smoke evacuation controller, and two-floor damper controllers (as shown in Figure 3.4). The HVAC system’s daily operation sequence is as follows. First, based on the outside temperature and humidity, the building manager starts the operation in heating or cooling mode with the desired moisture level. When a workday begins, at the scheduled time the AHU controller will set the system in the occupied mode. The other controllers periodically (e.g., 5 second cycle) check with AHU controller for the mode change.
The returning air controller watches the humidity, temperature, CO₂ as well as the fire alarm status. It regulates the return air damper G and opens/closes it according to the manager’s request. When in occupied mode, the outside air controller is constantly adjusting the outside dampers (damper B, E, F) by referring the CO₂ sensor data from the returning air controller.

In the occupied mode, the AHU controller processes the air supply. The chilled water control valve of the cooling coil is maintained at a certain degree based on the sensor feedback and mode setpoint (e.g., 55F adjustable) to lower humidity from the intake air. The supply fan speed is automatically regulated through the differential pressure sensor (DP sensor) feedback between the duct and the hallway. There is also an air filter that helps control dust. A static DP sensor is equipped between the two sides of the air filter to determine when the filter is clogged enough to be replaced.

The two-floor damper controllers are deployed in the lower part of the air duct. The floor damper controllers sense the differential pressure between air duct and floor hallway and use the feedback to manage inlet guide vanes and damper Cᵢ and Dᵢ (ᵢ = 1, 2), where C₁ and C₂ are for hallway supply, and D₁ and D₂ are for office supply. The smoke evacuation controller is responsible
for pulling smoke out of the building. In the occupied mode, the smoke evacuation controller is usually set in sleep mode.

### 3.2.3 Smoke Evacuation

During fire and smoke incidents, the fire alarm system is triggered and the AHU controller is notified. This changes the mode setpoint from the occupied mode to the smoke evacuation mode. The mode change is then spread to all controllers in the HVAC system. First, the outside controller will adjust outside air dampers $B, E, F$ to 100% open. The returning air controller will completely close damper $G$ for preventing smoke recirculation in the building. The supply fan in the air handler unit will be running in full capacity. The goal is to reduce smoke from spreading to the entire building while providing adequate oxygenated fresh air for personnel safety.

In the lower part of the duct, the floor damper controllers will open damper $C_i$ to 100% and completely close damper $D_i$ for diverting air supply to the smoke evacuation discharge plenum. The reason for doing this is to reduce oxygen supply in the office spaces where a fire is most likely to happen and increase oxygen concentration in hallways to facilitate evacuation. Meanwhile, the smoke evacuation controller will open the damper $H$ 100% and operate the exhaust fan to pull out the smoke as much as possible until the fire alarm is turned off either by manual reset or EMS override.

Analysis of these two scenarios shows that the devil is in the details. BASs are very complex and data-oriented systems. There are all kinds of analog, digital, and network devices with data from a variety of sensors. Therefore it is very easy to be overwhelmed and get lost in the technical details. On the other hand, the interoperability of BAS subsystems makes the design of the integrated BAS more complicated. Not only are different components interacting with each other through the control network, but changes in the physical environment also indirectly influence other subsystems' decision making. For example, change in temperature can cause fluctuation in air pressure, hence the gauging of differential pressure has to also consider the room temperature. Often the condition and success of a control task, such as air filter condition or damper opening are indirectly indicated by the physical status captured by sensors. Additionally, the logic of various control tasks changes according to the institutional policies under different situations. Errors in control logic and wrong
assumptions in the BAS can cause serious safety hazards. For example, two mutual contradicting
tasks could be implemented in the mechanisms for detecting decontamination. If implemented
without careful thinking of all possible situations in the BRI scenario, the BAS’s control could
result in contamination never being correctly detected.
CHAPTER 4: BUILDING AUTOMATION SECURITY ISSUES

The two case studies capture the basic structure, functionality, and requirements of the state-of-arts BAS. Based on the case studies, this chapter illustrates the abstract logic structure of BAS control network, and then it analyzes the security issues in each logic layers.

4.1 Building Automation Network

A modern BAS involves a vast amount of communication among all kinds of field devices and controllers through a layered industrial control network called a Building Automation Network (BAN). BANs are the backbone of Building Automation Systems. A standard enterprise BAN can be generally divided into three hierarchical layers. At the bottom is the field level, where small embedded devices measure and control the physical environment. The field level typically consists of networked sensors (e.g., motion or temperature sensors) and actuators (e.g., fan, locks, cooling coils). Devices communicate with each other through wired or wireless signals using protocols such as LonWorks [35], KNX [36], etc. The middle layer is where automation control is performed. Programmable logic controllers (PLCs), data acquisition units, and supervisory controllers live in this level and conduct distributed control tasks through network protocols, such as BACnet [37] and Modbus [38]. Monitoring and configuration tasks are realized at the management level. This level is where facility management systems, energy management systems, and cloud-based third parties integrate together.

Through the BAN, devices meticulously coordinate together to perform the daily operations of the facility. Buildings come in a variety of sizes and functions; hence the control components in BAS are highly heterogeneous and often come from various vendors with proprietary communication solutions (e.g., Johnson Controls N2, Siemens Building Technologies). Those communication solutions often lack security [13]. Also, due to their long life cycles, BASs often consist of numerous
legacy subsystems. This diverse range of communication mediums and devices presents a complex challenge to system-wide security.

4.1.1 Building Automation Network Protocols

Network protocols are the backbone of BAS. The communication protocol BACnet is one of the most popular protocols in building automation and control networks. BACnet is a protocol designed by ASHRAE (American Society of Heating, Refrigerating and Air-Conditioning Engineers) for controlling HVAC systems. BACnet as an open protocol communication standard has been widely applied to different types of practical systems in many fields, such as HVAC, lighting, life safety. It is the de facto standard in building automation due to its openness, flexibility, and interoperability. BACnet as a communication standard supports several different physical and link layer protocols including PTP, MS/TP, ARCNET, LonTalk, and Ethernet. It can be used to integrate diverse types of plants and vendors without having to use special hardware. The protocol supports poll-response and publish-subscribe communication patterns and services including Who-is, I-Am, Who-Has, I-Have for device and object discovery. Data inside a BACnet device is organized as a series of objects. Every piece of information in BACnet is encapsulated within a BACnet object. Each object has a type and a set of properties. However, the major focus of BACnet lies on the interoperability and functions, due to historical reasons, in practical, BACnet lacks proper cryptographic support and authentication mechanism.

Besides BACnet, it is also common to see other protocols such as LonMark, Modbus, etc. LonMark is a networking platform that is created to address the needs of control applications. It is a proprietary communication standard includes multiple protocols in the family developed by the Echelon Corporation in conjunction with Motorola in the early 1990s. The LonMark standard is based on the low-level proprietary communications protocol called LonTalk. As a rival protocol standard of BACnet, LonTalk offers a different type of solution to the interoperability issue. LonMark supports different types of transmission media, such as twisted pair cables, power line, RF, fiber optics or IP (both TCP/IP and UDP/IP). LonTalk requires specialized communications microprocessor called the Neuron, and together with its supporting software, the protocol estab-
lishes communication and exchanges information between devices. On top of the LonTalk, another protocol, LonWorks defines the content and structure of the information that is exchanged.

Another widely used open protocol in building automation is Modbus. The Modbus protocol was developed during the 1970s by Modicon, Inc. for use in industrial automation systems with programmable controllers. It is one of the most widely used means for connecting electronic equipment in industrial applications. Modbus supports traditional serial and Ethernet protocols and has simple, minimum hardware requirements. The benefit of Modbus is that it uses TCP/IP transport protocol, hence can be easily integrated with the Internet. However, Modbus protocol itself provides no security against unauthorized commands or interception of data.

Other than those three protocols mentioned above, there are other communication standards in building automation domain. Some example includes: KNX is an open network communications protocol for building automation that has been used more than 20 years and supported by hundreds of vendors. It is widely applied in Europe. DALI is a standardized interface for lighting control. EnOcean is an energy harvesting wireless technology that used primarily in building automation systems in response to the smart home concept. It is not uncommon to have more than one protocols coexist in a building automation system. Different parts of the BAS subsystem especially provided by different vendors may use different protocols. It is also common to see different layers of the system use different protocols, e.g., within the HVAC system between master and slave devices, they might communicate using Modbus, but the communication between automation devices and management system might use BACnet.

4.1.2 Field Sensors and Actuators in BAS

In the field level of building automation network resides various sensors, and actuators, e.g., temperature sensor, air pressure sensor, electricity meter, thermostat, heater, damper, airflow valve etc. Sensors and actuators were connected to controllers through direct-connected interfaces or simple field networks, such as, wired field buses that support Twisted Pair (TP) or Powerline (PL) as network media. The sensors translate analog signals to digital input for controllers. The actuators take output control signals to adjust physical environment.
4.1.3 Automation Controllers in BAS

Traditionally Programmable Logic Controllers (PLCs) play a critical role in control systems. A PLC is a digital embedded device used for automation of industrial processes. They perform real-time control of electromechanical processes. A PLC works by continually scanning a program, which is called the scan cycle. This involves reading sensor measurements repeatedly, executing control logic program to calculate output, and actuating output with electromechanical processes. PLCs are responsible for constantly adjusting physical machinery based on the sensor measurement as well as functions as a gateway between the machinery and human operators. PLCs translate continuous analog signals into digital values. Processing is performed using a cycle of input, processing, and output. The control logic for processing is typically represented using relay ladder logic written in a graphical language. The most widely used PLC control structure is a Proportional Integral Derivative (PID) controller. This controller algorithm has three separate constant parameters: the proportional, the integral, and the derivative values. PID controllers are used to dynamically adjust output by comparing setpoint (desired value) and actual value of the process variable from the process under control. However, such control logic can be easily simulated using the software nowadays.

The trend towards more intelligent controls in BASs necessitates data integration and new capabilities in the control devices. In order to satisfy the requirement, modern BAS demands more computing power. New generation embedded controllers running processes with rich functionalities are replacing PLCs. With the advancement of hardware, many vendors of automation controller are gradually switching to ARM architecture as their primary computing platform of current and future building controllers. One example is the Tridium ARM-based JACE controller, a widely used building automation controller. These devices use an ARM Cortex A8 processor and usually run a UNIX-like operating system.

Along with software control logic, cloud clients, web server, analytics are often deployed on the same embedded controllers, which increases the system complexity and makes it prone to potential vulnerabilities and exploits. Traditional operating systems (OSs) for embedded controllers are similar to commercial OSs (e.g., Linux, Windows) which encompass all OS services as privileged-mode
tasks in kernel space and allocate resources through discretionary access control. Compromising any code running in privileged-mode (for example device drivers) gives attackers the highest privilege in a system, thus bypassing any security controls. In order to protect safety-critical building applications in such mixed-criticality systems, a robust distributed embedded architecture must isolate applications and regulate local resource usage, so that even when the less critical applications are compromised by attackers, the critical ones are still functional in a safe way.

4.1.4 Management in BAS

Similar to the Industrial Control System (ICS), BAS may use Supervisory Control and Data Acquisition (SCADA) software to monitor automation facilities. SCADA often is used in distributed systems for remote monitoring and control that operates over communication networks. A SCADA performs centralized monitor and control for field sites over long-distance, such as electric power grids, a university campus, including monitoring alarms and processing status data. Based on information received from remote devices, supervisory commands either automated or issued by operators through Human Machine Interface (HMI) can be pushed to remote control devices, which are often referred to as “field devices”. SCADA servers that are used in BAS often no much different compared to an IT computer workstation. The majority of SCADA systems using in BAS often run standard Windows or Linux operating systems with proprietary software on a physical host or in the virtual infrastructure. Some vendors provide their customized operating system, However, according to the Guide to Industrial Control System Security from NIST, it is “often without security capabilities built in” [39]. Nowadays, due to the computing power has dramatically increased and available at an economically viable price, more powerful automation and supervisory devices have been deployed in the automation level that provides remote access through a web-based interface. A typical example is the Metasys solution from Johnson Controls, which have most of the management functionality in the network automation engine (runs embedded Windows). As a result, BASs starts to integrate with enterprise IT infrastructure and often managed by the IT teams. Consequently, BAS controllers are interconnected by a common backbone TCP/IP network through dedicated gateways and VLANs.
4.2 Security of Building Automation Network

The current de facto standard BAN protocols, such as BACnet and Modbus, were designed in the 80’s. They do not support modern security mechanisms, and often do not consider security in their design [40]. In 2012 the latest BACnet network specification [37] was published, which introduced guideline support for standard cryptographic protection (ANSI/ASHRAE 135-2012 Clause 24). However, talking with vendors we found that even newer BAN devices do not support these specifications for backward compatibility reasons. As a result, BAS operators have to rely on “air gaps” and “security by obscurity.” Ostensibly, “air gapped” networks are isolated from outside influence because they are not connected to outside networks. The Stuxnet attack is a prime example of this approach failing to defend against attacks that utilize non-network means (e.g., USB drives) to break the air gap [41] [42]. Furthermore, contrary to common belief, numerous BASs are not “air gapped” at all [3]. According to the search engine Shodan [43], in the U.S. there are over 50,000 exposed buildings on the Internet that can be directly accessed due to unintended misconfiguration [43]. With modern BASs moving towards more intelligent controls that heavily rely upon communication with the outside world, such air-gapping approach may not even be feasible. Once the air gap is broken, the whole BAN is vulnerable to attack. As Molina demonstrated at Black Hat USA 2014, once the attackers find their way into the control network, they can arbitrarily control the building facility [5].

Many facilities rely upon gateway firewalls for BAN security. Recent years also see much research effort on developing intrusion detection system (IDS) for Industrial Control Systems (ICS) [44], which share many similarities with BAS. While they add an additional layer of protection, firewalls and IDS alone are not sufficient. Firewalls must allow certain traffic through and cyber attacks can be successfully carried through legitimate communication paths. Due to real-time constraints in most ICS, it is hard to act upon suspicious traffic detected by IDS [45], IDS and firewalls are necessary to stop a range of attacks with known signatures, but are not sufficient to stop sophisticated attacks without knowing the semantics of various control logic, especially given the fact that traditionally IDS tends to produce a large amount of false positive [46]. This may be particularly problematic in a BAS environment where devices are highly heterogeneous. Furthermore, system
firewalls typically use coarse-grained discretionary access control models. Applications are either fully allowed or denied the network access based on the identity of the applications. This model is not sufficient since compromised applications with authorized identity will allow attackers to arbitrarily abuse the network resources. The tight integration among subsystems requires fine-grained access control.

4.3 Security of BAS Controllers

Threats not only come from the network but can also come from software vulnerabilities in the automation controllers. Study of software reliability shows that industrial software system, in general, contains 2-75 bugs per 1000 lines of executable code [47], which is considerably higher than normal code. On the other hand, patching of security vulnerabilities in control systems is difficult and usually have to wait when maintenance is performed. These issues make control systems like BAS an attractive target for attackers. Now in light of the advance of CPS, control systems are highly automated and digitally interconnected with high-level applications and the Internet. Considering the easy accessibility, poor protection, and potential benefit the hackers can gain, automation controllers have become the low hanging fruits for cyber attacks. The environment and assumption of real-time operating systems for BAS has changed. We can no longer confidently assume that all applications running on top of the OS are benign. For example, with the new setup, a program with buffer overflow vulnerabilities could possibly allow attackers to inject malicious code remotely, deceiving the scheduler to plunder CPU time slots, therefore intentionally starving critical applications, forcing BAS to violate real-time constraints.

Traditionally automation controllers run simple real-time operating systems (RTOSs), although those RTOSs are sufficient, they have mainly focused on build functionalities and real-time requirement rather than safety and security. Those legacy controllers often lack modern OS features, such as virtual memory units, and protection rings. Security has been relying on physical separation, using a different microcontroller for each function. This solution is not expendable. As shown from the case studies, there are already many microcontrollers for a relatively small scale scenario. When more and more microcontrollers are deployed in the BAN, the network becomes complex and hard to manage. They require a lot of cabling, consume a lot of energy, most importantly those
microcontrollers are not cheap. This problem is compounded by the increasing number of sensors that must be shared across devices.

Nowadays, modern automation controllers often contain a powerful processor and hosts multiple control tasks. Those often run commercially operating systems that are very similar to operating systems for general purpose computers, such as embedded Windows, VxWorks, QNX, and customized Linux etc. Most Commercial OSs are designed based on monolithic kernel architecture. In a monolithic architecture, all OS functions (including interrupt handlers, device drivers, and system call functions) are implemented as functions in one big kernel binary, runs in the same address space at the highest privilege level. Therefore, any vulnerabilities in one part of the kernel, such as a third-party device driver, can allow an attacker to issue any sensitive operations and gain kernel privilege. Furthermore, the monolithic architecture stuffs all system functionalities in the same address space, which makes the kernel code bloated. A typical Linux kernel contains over 2 million lines of code. Which makes it impossible to guarantee that reliability of the most important component of automation controller, hence jeopardize the safety and security of the building automation system.

Another important issue that is related to the operating system of automation controller is the process isolation. In most Unix-like systems, inter-process communication (IPC) is conducted through either message queues or Unix local sockets, which are all implemented through the file system. Therefore, the authenticity of the message is protected through file permissions and granted using role-based discretionary access control. The solution through flexible provides fairly limited security. If not properly configured, these file system handles could be exploited by adversaries. In fact, several such recent vulnerabilities have affected both Linux and Android (e.g., CVE-2011-1823, CVE-2011-3918, CVE-2016-9793). If a process is compromised by attackers, it is possible that the compromised process might be able to use IPC to interact with peer process and injects spoofing messages.

On the other hand, traditional Unix-like OSs are built to expose hardware resources for fair use by applications. Any process has an equal privilege to request services from device drivers. To conduct network communication, processes can open ports by simply requesting a socket and
writing to a memory mapped file. In essence, any compromised processes can send out outbound communication and conduct DoS attacks on the network. Moreover, if a malicious process gains more privilege, it can control the whole device and arbitrarily manipulate the inbound and outbound network traffic. In contrast to general purpose computing platforms, embedded devices such as BAS controllers often co-hosts applications with mixed criticality and various real-time requirements. Constructing such systems using the traditional OS resource sharing model will create situations where compromised processes in BASs can easily abuse network resources and spread malware throughout a network [48]. Recent massive DDoS attacks that utilized compromised IoT botnets are emblematic examples of this problem [49,50]. From both safety and security perspectives, in safety-critical systems network communication should be a privileged operation.

### 4.4 Security of BAS Management

The management system for building automation system is very similar to a enterprise IT system. Hence, the security threats are also similar to IT system. First, servers are often prone to computer viruses and backdoors. Many BAS management engines run embedded Windows due to vulnerabilities in software and system service. Control systems have a high-reliability requirement, it is easy to oversee the system update and apply software patches.

On the other hand, modern BAS management system often supports web-based interface and can be accessed through a browser. If not configured properly the system is vulnerable to typical web-based attacks. For example, if the certificate of the web server is not set up correctly attackers can conduct eavesdropping, man-in-middle attacks, and other common Internet attacks. Furthermore, for convenience, many system administrators set up remote access in BAS management network. Although such network access often protected by VLAN, VPN, etc., if there are misconfigurations in networks or the administrators had weak passwords, it is possible to expose weak point and allow attacks to gain access. Last but not least, the authentication between client and BAS server is critical. High-profile system breaches often start from compromising password either through exploiting the system, such as weak password cracking, authentication bypass or through exploiting the users, such as phishing attacks and other social engineering methods.
4.5 BAS Threat Summary

Based on the discussion above the security issues of modern building automation system can be categorized as the following three attack surfaces:

- Attack on the management system
- Attack on the internal network and protocol
- Attack on automation controllers and field devices

And for each surface, known potential threats are summarized as shown on table 4.1.
Table 4.1: Known potential threats

<table>
<thead>
<tr>
<th>Threats on Management System</th>
<th>Threats on Building Automation Network</th>
<th>Threats on Automation Controller</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Backdoor and Virus</td>
<td>1. Network Sniffing (Interception)</td>
<td>1. Physical Tampering</td>
</tr>
<tr>
<td>- Communication Hijacking</td>
<td>- Insert Malformed Data</td>
<td>3. Deception Attack</td>
</tr>
<tr>
<td>- Man-in-the-Middle Attack</td>
<td>- Insert Fake Commands</td>
<td>- Replay Attack</td>
</tr>
<tr>
<td>- Network Spoofing Attack</td>
<td></td>
<td>- Device Impersonation</td>
</tr>
<tr>
<td>5. Attack on Software and Service Vulnerabilities</td>
<td>5. Denial-of-Service Attack ( Interruption)</td>
<td>6. IPC False Data Injection</td>
</tr>
<tr>
<td>7. Social Engineering</td>
<td></td>
<td>8. Bogus Command</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9. Resource Monopoly Attack</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10. Resource Consumption Attack</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11. Code Injection</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12. Exploit Algorithm Weakness</td>
</tr>
</tbody>
</table>
CHAPTER 5: DESIGN OVERVIEW

In an effort to address the aforementioned security issues, this work focuses on the design of a robust computing platform that follows security-by-design principle to mitigate the aforementioned problems at the root. For a robust computing platform to support a mixed-criticality system, the design boils down to isolation, authenticity, and secure communication. A computing platform can be viewed as a layered architecture. The bottom layer is the hardware; the middle layer is kernel; the top layer is the user interface and runtime environment. In the traditional approach to embedded systems, all of those layers are part of the trusted computing base (TCB), which means that they are critical to system security [52]. If we want to create a secure platform, we need to make sure all of those elements are secure. That is a huge portion of a system to secure considering the lines of code involved. Therefore, this work chooses the alternative microkernel architecture with minimal TCB as the core technology. This chapter documents some of the choices of the design rationale.

5.1 Microkernel Architecture

In modern software, it is fair to say that operating systems are the core of any computing system. Bridging the boundary between hardware and software, OSs manage hardware and physical resources, such as memory, peripheral, processor usage, and energy, etc. for isolation and collaboration. From an application’s perspective, OS abstracts the complex details of various hardware mechanisms away and provides services through easy-to-use interfaces. Hence, the security and stability of a system greatly rely on the robustness and security of the operating system. The heart of an operating system is called a kernel. A kernel is the first program executed in a system. Running in the most privileged mode, it is responsible for initializing the platform and bootstrapping other services.

1Part of this chapter is published in IEEE International workshop on Communication, Computing, and Networking in Cyber Physical Systems (CCNCP 2017) [51] paper. Permission is included in Appendix A.
5.1.1 Microkernel History

Generally speaking, there are two main kernel architectures, the monolithic kernel architecture and the microkernel architecture. Microkernel was first proposed in the 1970’s. Designed by Per Brinch Hansen, the Nucleus real-time OS is considered to be the root of microkernel [53]. Being a minimalistic system, Nucleus features inter-process communication primitives using message passing and process control, and dedicates I/O, scheduling to external processes. Following Hansen’s work, microkernel became a very appealing solution in response to the existing security and reliability challenges in the monolithic kernel-based operating systems. Historically, Mach is considered as one of the classic examples of the first generation microkernel. Originally intended to be a replacement of monolithic UNIX kernel, Mach was designed with a simplified kernel structure with a small set of functionalities in the kernel space that concentrate on inter-process communication. Although the legacy of Mach still lives in many modern operating systems, such as GNU Hurd, and Apple’s MacOS, Mach failed to replace UNIX due to the performance impacts caused by the frequent context switch between kernel and user processes. During this period of time, many experimental microkernels emerged, some of the notable ones being MINIX, QNX, AmigaOS that are still under active development.

For a long time, the term microkernel has been associated with poor performance. However, this perception is not necessarily true. In fact, issues with performance are not inherent in microkernel architecture, but are due to the complicity of existing design choices. For example, many microkernels provide both synchronous and asynchronous IPCs and support more than 100 system calls in order to be compatible with the UNIX system. Later on, as proven by Liedtke, the first L4 microkernel shows that microkernels do not have to be slow. Written in assembly code and specifically focused on high performance, he improved the IPC by making it 20 times faster than Mach’s [54]. Liedtke formulated his famous minimality principle stating, “A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e., permitting competing implementations, would prevent the implementation of the system’s required functionality.” [55]. Since the success of the first generation of L4 microkernel, many L4 family microkernels have emerged in both academic experimental and commercial deployments. Those implementations include but are
not limited to: L4::Hazelnut, which is a proof-of-concept microkernel constructed in high-level language C++ [56]. L4/Fiasco, which is designed to support hard real-time and focuses to achieve low interrupt latency. L4Ka::Pistachio, which supports multiprocessor [57]. Most especially, NICTA (now Data61) released a third-generation microkernel, called seL4. SeL4 is the first formally verified microkernel in the world that focuses on high assurance and reliability. Recently, with the advancement in hardware and the increasing concern of security, the idea of microkernel has made a second comeback, as many experimental microkernel-based OSs, such as Redox, KasperskyOS, NOVA, L4Re have been proposed.

5.1.2 Microkernel Design

Microkernel’s design matches the principle of least privilege in information security (the structure of microkernel as shown in Figure 5.1). Contrary to monolithic kernel architecture, where all OS services and drivers are executed in the kernel address space, Microkernel architecture is designed in a modular flexible structure. In a typical microkernel-based operating systems, the kernel only handles low-level functionality and process primitives, such as low-level address management, hardware interrupts process control blocks (PCBs), and IPC. All other OS system services, such as file system, virtual memory management, network stack, and device drivers that traditionally resides in the kernel space, are removed from the kernel and running as processes in the user mode. The architecture is more robust compared to the monolithic kernel approach. This is caused by the reduced size of code running in privileged mode, which dramatically minimizes catastrophic errors. Furthermore, all processes in user mode lack the authority to directly access memory that doesn’t belong to those processes, thus they are well isolated.

In microkernel architecture, inter-process communication is the backbone. Different to the monolithic kernel, where all system functions reside in the most privileged processor mode, different parts of the system can directly invoke each other through a function call. Microkernel is organized through IPC invoked remote procedure call. Because different services are running in isolated memory space, data is explicitly passed through either registers or shared memory.
5.1.3 Microkernel Security Benefits

The motivation that drives the design choice of the microkernel is how we view the kernel. Traditionally, the kernel has been viewed as the central decision maker that aims to manage resources, hides hardware complexity, and provides fair services as efficient as possible with the assumption that every piece of code is benign. With every system services running in the kernel space, the result is a bloated monolithic kernel. This review is not practical, as we know that from device drivers to file system, each component of the system can contain vulnerabilities. As Tanenbaum pointed out [58], current operating systems are complex software. The Linux kernel has over 2.5 million lines of code (LOC) and Windows systems are even larger. It is impossible to guarantee the functional correctness of a monolithic kernel. The microkernel structure extenuates the kernel’s responsibility, and views kernel as the organizer, similar to the functionality of a Switch on network, that only provides communication mechanisms to loosely couple different modules.

This structure provides the following outstanding benefits compared to the monolithic kernel structure. First, isolates system services and device drivers dramatically reduce the potential of a security breach in the kernel space. It is well-known that device drivers have error rates 3 to 7 times higher than other code [59]. In general-purpose OSs, device drivers contribute to a large percentage of vulnerabilities. In monolithic kernel architecture, compromises device drivers will give attackers the highest privilege, while in the microkernel device drivers are user space processes; unless the
kernel is compromised the breach is well-confined within the vulnerable processes. Exploits of driver
code won’t lead to privilege escalations and the rest of the functionality remains intact.

Second, microkernels are more robust against failures, due to the reduced size of the code
in the kernel space. The functional correctness has a lot to do with the kernel’s size. A typical
microkernel consists about 4000 - 6000 lines of code. The small code size in microkernel makes it easy
to audit the code. As shown by seL4, it is even possible to apply formal verification to its code base
and mathematically prove the specification and implementation [60]. Furthermore, microkernels
generally do not dynamically allocate memory. This dramatically reduces the probability of memory-
based attacks such as buffer overflow and use-after-free exploits in the kernel space. Finally, the
microkernel’s unique modularity guarantees process isolation and ensures that there is only one way
to communicate between different processes [61]. The only inter-process communication (IPC) in a
microkernel system is via kernel primitives such as message passing. From the security perspective,
this makes it easy for developers to monitor process behaviors, system calls etc and detects potential
exploits. Also, because of this exclusivity, the kernel is in a unique position to allow, deny, or
otherwise referee all communication. Compared to a monolithic OS, all functions in the OS are
executed on behalf of the user process and no process switching takes place. It is hard to keep track
these function calls once the control is transferred to the kernel.

5.2 Mandatory Access Control

In a microkernel architecture, user-space subsystems and components are modularized into
isolated processes. The inter-process communication (IPC) under microkernel architecture is similar
to communication in distributed networks, where the kernel behaves like a router. For processes to
send data or signal across memory space, the processes must use kernel primitive, message passing.
Because of this exclusivity, the kernel is in a unique position to allow, deny, or otherwise referee
all IPC. On the other hand, in embedded systems IPC generally follows predefined patterns among
user-space processes. By regulating the use of IPC primitives, the kernel arbitrates the interaction
among isolated processes. It can enforce customized control policies for all data flows based on a
system specification. Thus, to enforce the communication policy for mixed-criticality systems, the
design of an access control mechanism to regulate the IPC in microkernel architecture is an ideal solution. This work follows such an approach.

A common IPC primitive in microkernel architectures is synchronous message passing. Using a system call, a process requests the kernel to send or receive a message; at this point, the kernel can determine the validity and authenticity of this request on the basis of pre-defined IPC policies. Once the message is validated, the kernel transfers the message across address-space boundaries through dedicated registers and memory. In addition, the kernel appends metadata with the source and destination information which cannot be forged. Because virtual memory mapping is also overseen by the kernel, shared memory between processes can likewise be subject to the pre-defined policy. By regulating the use of IPC primitives, the kernel arbitrates the interaction among isolated processes. It can enforce customized control policies for all data flows based on a system specification.

Taking advantage of the aforementioned unique feature in the microkernel architecture, this work applies mandatory access control to the message passing mechanism. Mandatory access control is a system-wide policy enforcement mechanism. Different from discretionary access control which is widely implemented in commercial OSs that let users decide the access control policies on their data regardless of global policies, mandatory access control is globally enforced and cannot be altered by users or applications themselves. In this work, the fine-grained mandatory access control is implemented as kernel module called Access Control Matrix (ACM) and only resides in the kernel space, which is inaccessible from user space hence it is immune from potential attack as long as the kernel is trustworthy.

5.3 Proxy-based Network Communication

Another part of the system that needs to be regulated is the network communication. Contrary to the general purpose OSs, in which the systems focus on providing fair service to all processes, in a mixed-criticality embedded systems network access should be treated as a privilege and only be granted to an application if it is needed to achieve its functionality. This is because in a complex distributed system many controllers cooperate together for a control task. A compromised process can open ports by simply requesting a socket, and spoof or conduct DoS attacks to other critical applications on the other devices.
Constraining network resources of a subset of processes are not trivial in the general purpose OSs due to the monolithic kernel architecture. However, it is easy to achieve under the microkernel architecture especially with the ACM mechanism, because in microkernel architectures system services and device drivers are running as user processes, and system calls use the kernel IPC primitives in the same way. Hence, to extend the policies with network communication control, this design deprives processes’ capability to communicate with network services, by default using ACM. For applications that need network access, this design applies the proxy pattern by introducing a lightweight special proxy process for applications with network communication requirements. Proxy pattern is a well-known software design pattern. In essence, proxy design pattern decouples application control logic and interface in an attempt to provide a surrogate for access and protect the control components from undue complexity. In this work, a proxy process is a local delegate for a remote process on a separate controller. All network communication to remote processes is routed through dedicated proxies (one-to-one) via IPC, allowing IPC policy to dictate valid remote communication. When BAS control processes need to use the network, the traffic is forwarded by a dedicated light-weight proxy process. During this process, the kernel vets the IPC request according to specific application communication policy. Figure 5.2 demonstrates a simple two-controller, two-process example of this framework. $A_{proxy\_2}$ is a proxy for Process A and $B_{proxy\_1}$ is a proxy for Process B. Process A sends messages to $B_{proxy\_1}$, which forwards the traffic across the network to $A_{proxy\_2}$, which scrutinizes and forwards the message to Process B.

![Diagram](image_url)

Figure 5.2: Each controller enforces global policy locally, allowing process A to communicate with process B over remote proxies
On the other hand, building automation network security is two-fold. The lack of authentication and encryption in communication protocols, such as BACnet and Modbus have a big impact on building automation network security. Even if all trustworthy devices are restricted with network access, there is nothing to stop attackers from plugging malicious devices into the network through physical access. This problem is non-trivial. Although it is feasible to develop new protocols with built-in strong authentication and state-of-arts encryption algorithms, it is not likely to be applied in practice due to the compatibility consideration with existing legacy devices and systems. In fact, the latest BACnet network specification [37] introduces guideline support for standard cryptographic protection (ANSI/ASHRAE 135-2012 Clause 24). However, talking with vendors we found that even newer BAN devices do not support these specifications for backward compatibility reasons.

In order to provide a solution that is compatible with existing BAS, but also has the capability to protect those critical control tasks and devices from potential attacks, this work relies upon the deployment of paired proxies. Since each network communication has a pair of proxies deployed on both ends, the proxy pair can negotiate security tunnels in the application layer regardless of the communication protocol and authenticate each other. The data would flow from a BAS process to a proxy process via IPC; then it would be encrypted and forwarded to the network driver. The receiving controller would forward incoming traffic to the listening proxy, which then decrypts and dispatches the message to the destination BAS process through IPC.

Following the philosophy of proxy design pattern as well as adding encryption through tunneling, we expect the following benefits of both these design features (proxies and tunneling) for our framework:

1. It isolates control logic and regulates usage of network resources. Compromised applications cannot monopolize the network, and incoming network threats will first be vetted by the communication framework.

2. It separates control functions from communications. Network communication becomes transparent for applications, which can then be redeployed to independent devices without worrying about interconnectivity.
3. It shields critical applications from direct network exposure and allows system designers to specify application-layer policies. Rather than simply forward network packets from/to processes, various domain-specific rules can be inserted in the kernel for scrutinizing incoming/outgoing information.

4. It enables system designers to consider security in the design phase at a high-level. The framework maps system-wide access control to local access control in multiple devices.

5. It guarantees end-to-end security even in complex BAN environments where diverse protocols and legacy devices are deployed. All communication can be encrypted based on the negotiation of the pair of dedicated proxies and tunneled through standard control protocols, such as BACnet and Modbus.
CHAPTER 6: IMPLEMENTATION

This chapter discusses the implementation details of the aforementioned design. For the prototype, this work chooses two mature microkernels as the development platforms, MINIX 3 and seL4. Through investigation, the ARM architecture stands out as the primary computing platform of current and future building controllers. One example is the ARM-based JACE controller, a widely used building automation controller. The implementation is targeted on a development board, BeagleBone Black. The BeagleBone Black uses a TI AM335x ARM Cortex A8 chip (1GHz), equipped with 512 MB of DDR3 RAM, and has a host of physical interfaces.

6.1 MINIX 3 Microkernel

MINIX is a well-known microkernel-based OS that is designed to exemplify the microkernel approach [62]. The latest version is MINIX 3, which is aimed for embedded devices with a focus on high reliability. From the user's point of view, MINIX 3 looks very much like a traditional UNIX-style system. In fact, most libc and NetBSD userland applications have been ported into MINIX 3. However, the architecture of MINIX 3 is totally different from traditional systems that are driven by a typical microkernel. The kernel of MINIX 3 is only about 6,000 lines of code. The kernel code contains hardware abstractions, interrupts, process control blocks (PCBs), timers, and IPC primitives. The OS is built in three layers: the device driver layer, system service layer, and user application layer. On top of device drivers, MINIX 3 consists of multiple servers running as isolated user space processes including, process manager, virtual file system, virtual memory manager, system information service, device manager, etc. MINIX 3 is the most mature open-source microkernel-based OS with the most device driver support. For those reasons, I use it as one of the platforms to develop the prototype.
6.2 The seL4 Microkernel

The seL4 is the latest member of the L4 microkernel family. Following the L4 kernel philosophy, the seL4 kernel supports abstractions for virtual address spaces, threads and inter-process communication with high performance. Most importantly, seL4 is the first mathematically verified software kernel [63]. Implemented and verified by the University of South Wales and NICTA, the formal verification proofs the executable machine code compiled from seL4’s 10,000+ lines of code is functionally correct against its high-level specification (through theorem prover), which implies that the kernel code is free of vulnerabilities such as buffer overflows, undefined behaviors etc.

Different from MINIX 3 and traditional Unix-like system, seL4 uses the capability-based security model. From the kernel’s point of view, resources are viewed as different types of kernel objects. The ownership or access rights of a kernel object (e.g., unused memory regions, page tables, task control blocks, IPC endpoints, etc.) is accounted by capabilities. Capability-based access control directly ties to the virtual memory management through MMU. A capability is an unforgeable token that represents the owner’s explicit authority and directly managed by the kernel. This model provides a flexible mechanism for reasoning and enforcing the control policies.

6.3 Implementation of Access Control Matrix

In MINIX 3, kernel IPC primitive is synchronous message passing. The synchronous message passing uses a rendezvous-style mechanism. When the IPC primitives are called, the calling process will be suspended until the message is copied from the sender to receiver. The messages are fixed-size 64-byte buffers, which includes a 4-byte endpoint identifier, a 4-byte message type field, and 56-byte payload. A destination endpoint has to be explicitly supplied to send or receive a message. An endpoint identifies a process uniquely among the operating system. It is composed of the process slot number concatenated with a generation number for IPC addressing which is stored in the process control block. There are 3 system calls: `ipc_send()` `ipc_receive()`, and `ipc_sendrec()`. The `ipc_send()` system calls block until the message is delivered to the receiving process. The `ipc_receive()` system calls block until the message is received from the target process. The `ipc_sendrec()` is an atomic operation for a round trip send and receive communication. In the current
version, synchronous message passing is reserved for device drivers and system server components with designed communication protocols.

The MINIX 3 IPC primitives provide an effective vehicle to implement the policy-enforced mandatory access control for process isolation and communication regulation. This work modifies the MINIX 3 kernel to bring the message passing primitives to all user processes. Because the kernel facilitates all of the IPC, it is the ideal location to enforce IPC policy. By directly exposing the IPC primitives to all user processes, we also simplify the communication paths and information flow. Additionally, I added three system calls in the process management server for improving IPC related operations. The `getendpoint(pid)`, which translates a process ID and returns the corresponding endpoint; `getendpoint_name(proc_name)`, which retrieves process’s endpoint by name; `select()` system calls that allows process to query all the pending messages.

Secondly, I modified PCB data structure, to add a field called access control ID (ac_id) and related system calls: `fork2()`, and `srv_fork2()`. Those system calls can assign each process, server, a unique number during booting period. They are designed to replace the original `fork()`, and `srv_fork()` system calls for loading process and system servers with specified ac_id. Process IDs are randomly assigned and can change, so we needed this ac_id to assist building definitions of IPC policy. We use the added ac_id field to uniquely identify each process and enforce the control policy.

Finally, I implemented the policy checking mechanism, Access Control Matrix (ACM) in the message passing primitive. The kernel now checks the ACM for each IPC to determine if the two processes are allowed to communicate. As the only code running in privileged mode, the kernel has the absolute authority over IPC. Because the ACM is stored in kernel space, it cannot be easily modified without recompiling the kernel source code. Thus, correct implementation of the ACM in kernel space can guarantee the enforcement of mandatory security checking.

As the name suggests, the ACM is a tabular data structure. The ACM using a sparse matrix data structure for fast lookup and space efficiency. Each ac_id indicates an index entry in the matrix. Each row in the matrix defines which processes the sending process can communicate with through message passing, and what type of message is allowed. A message type is a number
indicating what type of communication is allowed. The interpretation of message type is reserved for the individual processes and it is assumed by the kernel that it is pre-negotiated between the sender and receiver. In our experiment, we use the message type field to represent different remote procedure calls a certain process provides to the other process to invoke.

To explain how the mechanism functions, a simple example is illustrated in Figure 6.1. There are three processes in the example, App1, App2, and App3, two of which provide public remote procedure calls (RPCs). For App1 the RPCs \texttt{app1\_f1()}, \texttt{app1\_f2()}, and \texttt{app1\_f3()}, are represented by message types 1, 2, 3 respectively. App3 also provides three RPCs like App1; App2 has no publicly available procedures. For all processes, message type 0 is reserved to indicate an acknowledgment to the caller.

We want to allow App2 access to App1’s \texttt{app1\_f2()}, \texttt{app1\_f3()} functions, and we want \texttt{app1\_f1()} only be invoked by App3. We want all confirm messages between processes be allowed. With this example model, we can define the ACM as shown in the figure and compile this matrix together with kernel binary. During runtime, the kernel will use the ACM to audit the information flow. Suppose App2 tries to send a message with message type 2 to App1. The kernel will lookup the App1’s and App2’s ac\_id in the ACM. Since the bitmap is defined as 1101 the message will be allowed. On the other hand, if the message type is 1 the message will be denied and the request will be dropped instead.
6.4 Implementation of Proxy Framework and Security Tunneling

With ACM, the proxy design can be easily implemented. By default, all processes are blocked from network communications. This is done by restricting IPC communication between processes and network servers. In the MINIX 3 prototype, a proxy is assigned the same `ac_id` as the remote process its delegate. Hence, it is being subject to the same ACM policy. This allows a common, unified ACM policy to be distributed to all kernels in the controllers on the network. By using the same ACM policy among all controllers, processes can be migrated to different controllers but still provide the same level of regulated isolation and integration. This way, global security enforcement can be done in a completely distributed manner on each local controller. From the system’s point of view, the information flow is between process A and proxy process $B_{proxy-1}$, which is conveyed by kernel IPC primitives, which are subjected to the ACM. Therefore we map the remote IPC to local IPC, to which we can enforce policies.

By using proxies, we can effectively reduce the potential risks caused by malicious internal processes. However, another set of attack vectors for BAS devices are common network attacks, such as man-in-the-middle attacks, DDoS attacks, sniffing attacks, spoofing attacks, etc. For example, if an outside malicious device gains access to the networking medium, that device isn’t subject to any microkernel policy, therefore posing a threat to the BAN. Furthermore, BANs often involve multiple industrial control protocols at the same time which are often vendor dependent. The complexity and dependency of BAS make it impossible to switch to secure protocols in practice. A more practical solution would be to provide a VPN-like secure network tunnel between devices on the BANs through authentication and encryption mechanisms in the application layer. This is a well-studied problem in computer networks; protocols such as TLS have been widely used on the Internet and enterprise networks. Although, using proxy-based communication provides an ideal medium to establish secure network tunnels on an application-to-application basis. In the proposed framework, we use pre-distributed public/private key pair in proxies. A symmetric session key is negotiated between proxy pairs using their asymmetric key pair through standard Diffie-Hellman key exchange algorithm [64].
This solution can be extended with special cryptographic hardware support, such as using a Trusted Platform Module (TPM) to provide a stronger chain-of-trust. For example, with TPM support an authorized root key pair with a certificate can be stored the TPM hardware. A TPM driver could be used to provide service for proxy processes for key derivation, data sealing, and other cryptographic functionality. In that case, asymmetric key pairs for proxies can be derived at runtime and the proxies will never need to store keys in nonvolatile memory.

Adding all these lightweight proxy processes allow system mapping network communication into local IPC, and enables the microkernel to regulate the inter-device IPCs through arbitrating local IPCs between local process and proxy using ACM policies. Global security is obtained by uniform local security checks enforced in the lowest layer in each device.

When a process sends a message to the proxy, the communication conduct the following logic steps:

1. BAS control processes issue `ipc_send()` or `ipc_sendrec()` to initiate communication.  
2. The request is trapped into the kernel, the kernel verifies the validity of this communication against ACM.  
3. The kernel forwards the message to the waiting proxy.  
4. The proxy receives the message through `ipc_receive()`.  
5. The proxy checks the validity of the application specific message content.  
6. The proxy encrypts and forwards the outgoing message to the network server.  
7. The proxy replies to the BAS control process indicating the message was sent.

To transfer a large amount of data, shared memory can be used. MINIX 3 supports system V shared memory operations, `shmat()`, `shmgdt()`. But the implementation is realized using the userland process IPC server. To set up shared memory, processes communicate with the IPC server using message passing. This work modified the IPC server to fit the policy enforcement. Processes that initialize shared memory explicitly specifies the process endpoint who are able to access this.
memory segment. Successfully creating the shared memory segment returns a random secret key. The process can pass the secret key to the target peer using standard IPC which can only happen if they are allowed to communicate. Only the explicit specified process with the random secret key can attach the shared memory. The way the shared memory is also mediated by ACM.

6.5 An Illustrative Example

An illustration of how the system works in a distributed environment is provided by the following example, as shown in Figure 6.2. Assume in a control scenario that involves three devices, Controller A, Controller B, and Controller C. Controller A hosts a client process \( C_1 \), Controller B hosts a server process \( S_1 \), and Controller C hosts a server process \( S_2 \) and a client process \( C_2 \). Client process \( C_1 \) requests service from both server processes \( S_1 \) and \( S_2 \). Client process \( C_2 \), a legacy process, requests service from the local server process \( S_2 \).

Using the proposed application communication framework we deploy server proxies, \( S_1_{proxy\_A} \) and \( S_2_{proxy\_A} \), on Controller A on behalf of \( S_1 \) and \( S_2 \) respectively. On Controller B we deploy a client proxy, \( C_1_{proxy\_B} \), on behalf of \( C_1 \). Similarly, for communication between \( S_2 \) and \( C_1 \) on Controller C a client proxy, \( C_1_{proxy\_C} \) is deployed. Additionally, because \( C_2 \) is a legacy process it poses potential security risks for other control processes, so \( C_2 \) will be installed in the untrusted partition. Direct communication between \( S_2 \) and \( C_2 \) will be prohibited, and a cross-partition proxy \( C_{2\_trusted\_proxy} \) will be deployed. On each controller, the communication policy is enforced through OS policy enforcement primitive (ACM in security-enhanced MINIX 3) and (capability in seL4) in the kernel. The remote proxies share the same access control ID as the corresponding processes they represent. Those proxies are assumed to be trusted and have access privilege to the network infrastructure services.

When client \( C_1 \) needs service from server \( S_1 \) (either for data sharing or invoking remote procedure calls), an IPC will be issued in the form of message passing. From \( C_1 \)'s point of view the message is sent to server \( S_1 \), but actually the message will first be vetted by the kernel according to policy and then sent to proxy \( S_1_{proxy\_A} \). The proxy then forwards the message to proxy \( C_1_{proxy\_B} \) through the network using a secure network tunnel between those two proxies. Eventually, the proxy \( C_1_{proxy\_B} \) forwards the authenticated and decrypted the message to process \( S_1 \). The reply follows
the same route. A similar procedure applies to communication between client process $C2$, proxy $C2_{trusted\_proxy}$, and server process $S2$ locally on Controller C.

We assign the same access control ID as server process $S1$ to proxy $S1_{proxy\_A}$ (the proxy working on behalf of process $S1$ on Controller A). This allows a common, unified ACM policy to be distributed to all kernels in the controllers on the network. By using the same ACM policy among all controllers, processes can be migrated to different controllers but still provide the same level of regulated isolation and integration. This way, global security enforcement can be done in a completely distributed manner on each local controller.

![Diagram of network communication](image.png)

Figure 6.2: Three devices, four process scenario demonstrating the usage of cryptographic tunneling between the proxies

The prototype using seL4 microkernel has similar implementation and achieve the same goal. Thanks to the capability-based access control model, the seL4 microkernel already has strong mandatory access control enforcement. No modification in the kernel space is needed. The policy is realized in the form of endpoint capability distribution, which happens in the root (init) user process. The proxy-based network communication works the same way. The development of the seL4 prototype is lead by Richard and the detailed information is documented in [65]. This shows that the chosen design works well with microkernel architecture in general.
CHAPTER 7: EVALUATION

This chapter explains the experiment I conducted using the security-enhanced MINIX 3 prototype. These experiments are designed to evaluate the system security, the effectiveness of the proposed solution, as well as performance in comparison with an unmodified microkernel, as well as the widely applied Linux. The following sections are organized as three steps: (1) introduction of the scenario; (2) threat model and security analysis; (3) performance analysis.

7.1 Temperature Control Scenario

In order to evaluate the design in a realistic way that simulates a realistic use case that captures the gist of building automation system operations. I developed two temperature control scenarios based on the case study of a biological research laboratory HVAC system.

7.1.1 Single-device Temperature Control Scenario

The first scenario is a single-controller control scenario. The goal of the scenario is to maintain the laboratory temperature within the desired range. Failed to maintain the laboratory temperature within the extended time will trigger an alarm. Administrators can monitor the control environment status, and adjust the desired laboratory temperature from a web portal. The logic structure of the simple scenario is represented in Figure 7.1.

Following the implementation of the prototype, this experiment uses the BeagleBone Black to simulate the building controller. The temperature sensor chooses the BMP180 barometric pressure sensor, which can measure both air pressure and temperature. The heater actuator is simply a fan actuator together with a small electrical heating mat for emulation. For the alarm actuator, I use the onboard LED light instead. The controlled environment is represented using a 3D printed PVC container. The container includes two partitions. One is used to simulate the laboratory. The other
Figure 7.1: Single device temperature control scenario logic structure

is used to simulate the HVAC room, where the fan is mounted. The airflow from HVAC room to the laboratory. Figure 7.2 shows the our testbed implementation.

The control task is realized in five processes, temperature sensor driver, fan actuator driver, alarm actuator driver, temperature control process, and webinterface process respectively (Figure 7.3). Each process communicates via synchronous message passing. Additionally, each process has a set of predefined message types it accepts, while the ACM mechanism restrains which kind of message types can be sent from which process. The temperature sensor process periodically samples the environment temperature and sends the fresh data using nonblocking send system call to the temperature control process. Both the alarm actuator process and the fan actuator process are implemented to passively wait for commands from temperature control process. Lastly, the webinterface process acts as a basic human-machine interface. It is a static HTTP web server with 5 fixed child threads. The process maintains TCP socket on port 8080 and supports HTTP GET and HTTP POST. In the scenario, the most important process is the temperature control process. When the process starts, it executes the initialize function to retrieve endpoint of each process it needs to communicate with. Then the process enters a while loop, waiting for the new sensor data from temperature sensor process. When data arrives, the sensor data will be compared with temperature setpoint to decide whether to turn on or turn off the fan, meanwhile, a timer will be checked if the
temperature is out of the range of the setpoint. If within a certain time the temperature control fails to maintain the range within the setpoint, the alarm will be triggered. Then the process will check if there are pending messages from web interface process for updating new setpoint. At the end of the while loop, environment information will be written in a log file.

7.1.2 Network-based Temperature Control Scenario

The second experiment is a multiple-device temperature control scenario. Similar to the former example the goal of this control task is to regulate the temperature of the laboratory. In an effort to make the simulation realistically reflexing BAS, and test the network security of the platform, this temperature control scheme is implemented in controllers. Controller A represents the building controller and hosts the webinterface process, temperature control process, and alarm actuator driver. Controller B represents the Air Handler Unit, and hosts the temperature sensor driver, and fan actuator driver as shown in Figure 7.4.

Between those processes that communicate through the network, proxy pairs with pre-distributed public/private key pairs. Those proxies includes temperature control proxy in Controller B, webin-
Figure 7.3: Single device temperature control scenario process communication

Figure 7.4: Multiple device temperature control scenario logic structure
terface proxy, temperature sensor proxy, and fan actuator proxy in Controller A. This logic process communication is shown in Figure 7.5.

7.1.3 Implementation using Linux

The implementation on Linux is very similar to the implementation on MINIX 3. The only major difference is that on Linux the interprocess communication is conducted through POSIX message queues. Message queues provide an asynchronous communication protocol. A majority of real-time operating systems encourage the use of message queuing as the primary inter-process communication mechanism for real-time applications. On Linux, message queues are first in first out. They are implemented through the virtual file system and are supported by the real-time library. Also, there are no proxy processes for network communication. Each process can directly open a network socket.

7.2 Security Evaluation

7.2.1 Threat Model and Security Analysis

To fully examine the viability of the proposed application communication framework, we simulated possible attack vectors and compared the outcomes with a traditional-style implementation. Based on existing real-world attacks, the threat model assumes adversaries can have access to the BAN network, and some of the processes may contain vulnerabilities that can be exploited. According to the threat model, I simulated three categories of attacks from least to most starting attacker
privilege: 1) network attacks, 2) compromised network interface attacks, 3) and compromised control process attacks. In each of these scenarios, I illustrate what possible privileges attackers can gain to test the safety/security properties of the communication framework design.

The analysis presumes the alarm actuator process, the temperature control process, are critical applications for the system to defend. Also, I presume proxies are trusted components that are functionally correct. In practice, I believe this is a safe assumption due to proxies’ simple functionality and potential to be automatically generated according to a high-level specification.

7.2.1.1 Attacks from Network

Network-based attacks pose great challenges for distributed BASs. Many attacks specifically target weaknesses in industrial control and communication protocols. This analysis executed three kinds of attacks: identity spoofing attacks, man-in-the-middle attacks, and DoS attacks. To the best of my knowledge these attacks represent the majority of threats to BANs. Given the two device scenario, I installed a third device on the network with Kali Linux to perform the network attacks.

Many industrial communication protocols can support routing traffic over TCP/IP and Ethernet (e.g., BACnet/IP, Modbus TCP/IP). In the traditional implementation, an attacker on the same network can arbitrarily send spoofing packets to control devices. Without the deployment of proxy, false pressure differential data sent to the airflow control process will interfere with the control logic.

However, this attack was prevented using the framework prototype. This is because the proxy handles inter-device authentication and encryption. Those spoofing packets that weren’t encrypted with the correct session key used between temperature control proxy and temperature sensor proxy, hence the spoofing message cannot be authenticated and was simply be dropped or reported by the proxy. Similarly, with the correct implementation of the secure network tunnel, assuming the cryptographic infrastructure is implemented correctly, there is no way attackers can conduct man-in-the-middle attacks without obtaining a certified asymmetric key pair. At most the attacker can do is simply forward the messages. For those broadcast proxy that handles legacy processes, depends on the specific implementation, the message might not be encrypted. Furthermore, the proxy can incorporate with a white-list approach to prevent from receiving spoofing broadcast message.
If the controller supports Trusted Platform Module (TPM), the proxy can take advantage of the unforgeable device identity and use it to verify the legibility of the senders. The key point is that with the proxy design provide a layer of abstract for authentication and system level policy enforcement and can be tailored as needed with emerging technologies. Therefore, the integrity of control logic and confidentiality of device data/command are preserved with the proposed framework.

On the other hand, one of the most influential attacks on a BAN is DoS attacks. DoS attack seeks to exhaust computing and network resources in a system. With the access of the network, attackers can flood the network and temporarily disrupt the connection between Laboratory Controller and Chamber Controller. Although the attackers might not be able to infiltrate into the control system, through DoS they might prevent or delay the sensor data to be delivered by the temperature control process across the network. Without a proxy, the airflow control process would be busy handling all those false requests from the network. However, using the proposed communication framework, only proxies will be directly impacted, the control loop in the airflow control process would be still intact. Although it cannot receive the current real-time laboratory temperature, it still maintains control and can turn on the alarm or adjust the fan. Of course, if the alarm actuator process is also a remote process, it might be impacted as well, this guarantee depends on how the system is integrated and deployed. But in general, the proxies work like application-specific firewalls and filter out invalid requests with high accuracy. Furthermore, although to completely rule out the impacts of DoS attack, additional network level protection are needed, the proxy can help detect unauthorized network traffic against the system policy and report spurious behaviors from the BAN.

### 7.2.1.2 Compromised Interface Process

As most of the network-facing applications, cyber attack is a constant threat. If an attacker can find misconfiguration or vulnerabilities in the web interface process, the attacker can arbitrarily control the process. With this assumption, we simulated the behavior of the compromised web interface process. First, the attacker might attempt to send spoofing message to peer processes *e.g.*, the alarm actuator process and fan actuator process. With the proposed communication framework and the underlying mandatory access control, the spoofing attempt that targets any processes
other than the explicitly authorized process with explicitly authorized communication type would simply fail. Hence, the integrity of high critical applications is preserved even in the presence of compromised processes.

Furthermore, assuming the temperature control process is a high assurance application, therefore, the disturbance can cause by the compromised web interface is very limited. However, in the traditional system, the attacker might be able to invoke system calls, such as \texttt{kill}, \texttt{fork}, \texttt{etc.}, but due to the isolation provided by the ACM policy, restriction on system calls would be imposed even the web interface with root privilege (which, is only a concept in file system for microkernel architecture). Although in order to fully minimize the impact of this level of compromise, a legacy application should be running in a resource-restricted partition with a real-time scheduler. Furthermore, the proxy can be developed with monitoring mechanism, in that case even if the user interface keeps spoofing the proxy with redundant messages in a fast speed, the invalid communication can be simply dropped and the violation can be logged and reported. At most the proxy would be bogged down with false messages but the main control logic would remain intact.

### 7.2.1.3 Compromised Field Level Control Process

Finally the field level processes, such as the temperature sensor process and alarm actuator process in this case, also have chances to be compromised. Such breaches would be destructive in any situation. It is the same with the proposed prototype. This problem requires additional solutions (\textit{e.g.}, through the deployment of redundant sensors and actuators). However, in practice, the negative laboratory pressure control logic is likely co-hosted with other control logic on the same device, such as fire alarm. Even in the situation where the alarm actuator process is compromised, the prototype application communication framework does prevent the attacker further escalating his/her privileges on the device and network.

In general, the experimentation shows how potential risks of common network-based attacks can be prevented or minimized without replacing existing industrial control protocols and legacy applications immediately. Even if attackers are able to compromise some of the processes, they won’t have total authority. Hence, no single device is solely in charge of the security of the whole system.
7.3 Performance Evaluation

In BAS, the response time and latency are of great importance in real-world usage. Thus, any new features or designs much be weighed in terms of performance with the comparison of existing systems. Security does not come without cost. The design adds a number of proxy processes and cryptographic tunneling, and multiple context switches during operations. Hence, it is reasonable to see a modestly increased overhead comparing to a system without using encryption as long as the performance is acceptable in the BAS control environment.

Historically, one of the main concerns for adopting a microkernel architecture was its lower performance when compared to monolithic architectures. However, this concern has been negated by modern hardware and advancement of OS design. Modern microkernels, such as seL4 and L4/Fiasco, achieve high performance through myriad code optimization techniques. Hence, these specific disadvantages may not apply in the same way to microkernel systems in BAS domain. In particular, the hallmark benchmark of microkernel systems, IPC latency, has been demonstrated to be less than 100 nanoseconds for some implementations. Overall, IPC performance is the most critical metric for microkernel performance, since all component interactions occur through IPC. Because microkernels are highly tuned for IPC performance, this analysis hypothesized that the additional overhead of proxy processes would be within tolerance for building automation controllers.

Based on the observations of BAN traffic from local university BAS networks, devices exchange data on the network using BACnet protocols over UDP/IP. In the application layer, BACnet Application Protocol Data Unit (APDU) is encapsulated in the UDP data portion. The maximum APDU length can be as high a 1476 bytes; however, for backward compatibility, the size of BACnet packet is usually limited to 480 bytes, which is specified in each APDU’s “MAX APDU Length Accepted” field. Therefore, this evaluation only evaluates the performance of network latency up to 1024 bytes in each situation. Furthermore, we see that the BAS traffic is relatively sparse (around 35 packets/second for a trunk of BAN involves around 40 devices), although more data is needed to confirm this. Because of these low-bandwidth observations, the evaluation didn’t measure bandwidth systematically (although we were able to exceed the 35 packets/second without any difficulty). Using
Table 7.1: IPC RTT for Linux and MINIX 3 on a BeagleBone Black Board

<table>
<thead>
<tr>
<th>Size (bytes)</th>
<th>Linux Message Queue</th>
<th>Linux Unix Socket</th>
<th>MINIX 3 IPC</th>
<th>MINIX 3 shmem</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30.5</td>
<td>79.6</td>
<td>18</td>
<td>-</td>
</tr>
<tr>
<td>64</td>
<td>33.4</td>
<td>79.3</td>
<td>18</td>
<td>-</td>
</tr>
<tr>
<td>128</td>
<td>35.2</td>
<td>80.4</td>
<td>-</td>
<td>93</td>
</tr>
<tr>
<td>256</td>
<td>37.3</td>
<td>80.7</td>
<td>-</td>
<td>93</td>
</tr>
<tr>
<td>512</td>
<td>36.1</td>
<td>83.2</td>
<td>-</td>
<td>95</td>
</tr>
<tr>
<td>1024</td>
<td>36.6</td>
<td>82.5</td>
<td>-</td>
<td>95</td>
</tr>
</tbody>
</table>

these observations, this evaluation models the workload of the building automation system. For comparison, the Linux network performance is measured as the baseline.

On the BeagleBone Black board, this evaluation compared two Linux IPC mechanism, the POSIX message queue and UNIX local domain Socket, against MINIX 3 message passing (see Tables 7.1 and 7.2). MINIX 3 `clock_gettime()` on BeagleBone Black is only accurate to 100 microsecond; the results shown here are the average of scaled-up tests. I observed that MINIX IPC is faster than Linux IPC when transferring small chunks of data. MINIX 3 IPC is limited to 64 bytes maximum; however, larger transfers are most optimal through shared memory. For network communication using UDP, MINIX 3 is generally slower. We believe this is due to the lack of overall performance optimization in MINIX 3 and an inefficient network driver for the BeagleBone. However, this performance is still acceptable in the BAS domain. For network latency, the proxy design only adds about 4% overhead without tunneling, and roughly 10% to 30% overhead with tunneling (similar to Linux).

This evaluation is not a rigorous benchmark for systematically comparing different OSs’ kernel performance. However, the result provides a preliminary evaluation of the approach for the BAS domain. The proposed communication framework shows promising results with tangible performance overhead cost for the overall communication security benefits. Based on these measurements and collected real-world BAN dataset, the solution is sufficient for BANs’ performance requirements.
Table 7.2: Network RTT for Linux and MINIX 3 on a BeagleBone Black Board

<table>
<thead>
<tr>
<th>Size (bytes)</th>
<th>Network RTT (microseconds)</th>
<th>Network RTT with Encryption (microseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Linux</td>
<td>MINIX (Proxies)</td>
</tr>
<tr>
<td>64</td>
<td>459.0</td>
<td>958.0</td>
</tr>
<tr>
<td>128</td>
<td>513.3</td>
<td>984.0</td>
</tr>
<tr>
<td>256</td>
<td>534.6</td>
<td>1040.0</td>
</tr>
<tr>
<td>512</td>
<td>638.7</td>
<td>1156.0</td>
</tr>
<tr>
<td>1024</td>
<td>840.6</td>
<td>1386.0</td>
</tr>
</tbody>
</table>
CHAPTER 8: BACKWARD COMPATIBILITY AND VIRTUALIZATION

The trend towards more intelligent controls in BASs necessitates data integration and new capabilities in the control devices. In order to satisfy the requirement, modern BAS demands more computing power. With the advancement of hardware, many vendors of automation controllers are gradually switching to ARM architecture as their primary computing platform of current and future building controllers. Responding to and driving these developments, hardware technology for BAS changed dramatically from 8-bit to 64-bit processors, from single core to multi-core [66]. Along with software control logic, cloud clients, web server, analytics are often deployed on the same embedded controllers, which increases the system complexity and makes it prone to potential vulnerabilities and exploits [11].

Although there are many promising proposals in the research of CPS security, it is very rare to see those solutions get adopted in actual products [12,13]. The challenge of upgrading the existing BAS to satisfy the requirement of the mixed-criticality system is two-fold. First, directly applying security in the legacy system is infeasible due to hardware limitation and the lack of security consideration in the initial design. Many existing building controllers (e.g., Programmable Logic Controllers) have been deployed over the last 10 years. Those controllers often do not distinguish between user mode and privilege mode. Also, within these older models various control tasks are built into the same memory space without an operating system to help isolate each other. Considering their age, it is essentially impossible to patch or upgrade such devices. Exposing those devices on the Internet will post great risks.

Secondly, completely replacing the legacy system is cost-prohibitive. Porting existing software is a costly and time-consuming process. A study shows in businesses, porting an existing software to other platforms generally costs 30% of the total development cost of the product [67]. Many vendors have their own hardware platforms with customized operating systems to support their software.
stacks. For example, for the well-known Tridium automation solution they use the Honeywell JACE embedded IoT controller, which built its software on a relatively new ARM Cortex-A8 processor, and the QNX real-time operating system which has been in use over the last decade. This solution is probably one of the most similar commercial uses compared to our proposed security-enhanced OS prototype in terms of the hardware platform and OS architecture [68]. Despite the similarities, it is non-trivial to port the existing codebase to the proposed security enhanced OS, let alone port many of that old software on other solutions. On the other hand, new OSs often lack device driver support from third-party vendors which further complicate the adoption process.

Those challenges are confirmed by Honeywell Sentience production team. Honeywell Sentience supports the consumer and industrial IoT building automation solutions. In an effort to integrate Honeywell control products with cloud platforms for enabling “smart” connected assets, and advanced analytics, the core of this solution is the sentience IoT gateway. The gateway is responsible for collecting device data and communicating with cloud platforms. Hence, the security of the gateway is crucial since compromising the gateway will allow attacks to access cloud assets and control actuators. It is therefore preferable to construct such a gateway with the aforementioned security microkernel platform. The Sentience team has been developing the IoT gateway target a customized Linux over the years, although they are interested in our previous work. However, it is unrealistic to abandon existing work in favor of better security in a foreseeable future. Hence, any practical solution must be non-intrusive and facilitate this transition in a gradual incremental process.

The lessons we learned from this experience:

- It is cost-prohibitive to abandon legacy software stacks for businesses.
- It is infeasible to expect businesses to build applications from scratch in favor of security.

8.1 Virtualization

One of the solutions for converging legacy software and the new platform is through virtualization [69]. Virtualization refers to the technologies designed to provide a virtual runtime environment for hosting software that traditionally runs on physical computers. Virtualization has been the key
technology to cloud computing [70]. It allows different operating systems to share the same hardware resources efficiently and maintain independence. This method divides physical resources into multiple shares which makes applications more accessible, reduces the number of resources required, provides better management, and thereby cut costs. Most importantly, programs do not need to depend on specific OS.

Driven by cost and security concerns, virtualization on embedded system-on-a-chip (SoC) is prevailing [71]. Especially, with advanced hardware assists, the impact on performance degradation is minimal [72]. The fundamental idea behind virtualization is to abstract the hardware, e.g., CPU, memory, network etc. into isolated execution environments to allow each separate environment to run as if on a private computer. Virtualization brings desirable benefits to embedded devices. Embedded virtualization allows consolidation of multiple heterogeneous hosts with rich functionalities on a single piece of silicon. Using embedded virtualization not only reduces costs but also provides a mean to host legacy applications. The strongest motivation for embedded Virtualization is that it increases the BAS security by isolating software in provisioned environments. This idea is specifically fitting for mixed-criticality systems in BAS. If the isolation between virtual environments can be guaranteed, and the communication policy can be strongly enforced according to specification, then critical processes in BAS would have their own dedicated (virtual) processor, and their own dedicated (virtual) memory space, even they are co-hosted in the same physical controllers. The impact of potential breaches can be minimized by running those control tasks in a virtual machine. Even better is that legacy software can expect to require very minimal modifications.

In general, embedded virtualization on ARM architecture can be provided in two ways, paravirtualization, and hardware assisted virtualization [73]. Para-virtualization is a virtualization technique that uses a modified guest operating system in cooperation with the hypervisor to simulate runtime environment. In para-virtualization, the guest OS runs in user-land as a normal process. The hypervisor runs in kernel space [74]. The guest OS communicates with the hypervisor through an abstract layer called virtual machine monitor (VMM). The VMM provides the guest OS with extra APIs, called hypercalls for accessing hardware related services, such as virtual memory management, interrupts, and networks. Because conventional OSs are expected to run in kernel space,
that directly manages hardware and those OSs have to be explicitly ported to use the VMM API. The benefit of this technique is that no special hardware is required, and it is relatively easier and more efficient compared to completely simulated lower level hardware since the OS and hypervisor that work together to save the work to emulate the system’s resources. Projects such as Xen, L4Re and VMware support this approach [75,76]. However, the modification of conventional OSs are not trivial. Additionally, because the guest OS cannot directly access hardware devices, the drivers in the guest OS have to be modified accordingly. Last but not least, the VMM and memory management in the hypervisor add additional overhead which increases the performance impact, in comparison with running native system directly on the hardware. As Varanasi et al. pointed out, para-virtualization has the drawback of high engineering costs for having to modify guest OSs [72].

The other approach is using hardware assisted virtualization [72]. Hardware assisted virtualization relies on additional hardware support to provide virtual environments. The benefit of this approach is that it allows unmodified conventional OSs in virtual machines with minimal overhead. In demand of virtualization requirement, both ARM and X86 architecture have provided a platform that support hardware assisted virtualization. With the release of ARM Cortex-A15, ARM now supports virtualization extensions in their architecture [77]. These extensions provide three major parts: 1) CPU virtualization extension, 2) memory virtualization, and 3) I/O virtualization extension. CPU virtualization adds a new privileged CPU mode, HYP mode (privilege level PL2). This HYP mode has a higher privilege than regular kernel mode (privilege level PL1) which allows the hypervisor to execute sensitive instructions. Additionally, this CPU extension adds an extra set of vector tables and registers for simulating dedicated virtual CPUs on which guest OSs can possess without releasing during context mode switch. With the support of memory virtualization, ARM provides multiple layers of memory address translation directly implemented in hardware. This implementation simplifies the hypervisor’s work of virtual memory management and improves the performance. Using software to emulate I/O devices is expensive, the I/O extension provides virtual interrupt handlings to facilitate this process. Using the virtual interrupt distributor, the hypervisor can pass-through specific interrupt handlings directly to virtual machines without incurring overhead for routing and tracking each I/O operations. For some latest ARM models, it even
added system MMU that allows the hypervisor to set up direct memory access (DMA) with guest OSs securely, which reduces the risk of malicious DMA drivers to temper physical memory. This approach is preferable for embedded virtualization not only because it reduces the complexity of implementing hypervisor and provides virtual machines with near minimal performance overhead. Most important, this approach improves system reliability and security.

8.2 Architecture

The hypervisor solution shows a promising direction for supporting the requirement of a secure BAS controller in practice. However, security of BAS controllers does not come naturally with virtualization. With this approach, the functional correctness and reliability of the hypervisor are crucial. Since the system depends on the hypervisor to guarantee the isolation, any malfunctions in the hypervisor will threaten the functionality of critical control tasks. Since the hypervisor is running in HYP mode, if there are vulnerabilities in the hypervisor, they can be exploited, which will give attackers higher privilege than the kernel itself.

One of the popular hypervisors is KVM [78]. KVM is a virtualization framework that turns the Linux kernel into a hypervisor. Maintained by Linux community, KVM has the advantage of rich services and drivers support. But due to the monolithic kernel design, KVM suffers the same threats as Linux. Recently researches have shown that many vulnerabilities such as the guest execution escape, the guest reads of other guest data, the ring 3 to ring 0 privilege escalation can be exploited [79].

8.2.1 seL4 as Microvisor

Given that monolithic kernels are inherently impossible to secure without compromising the usability and compatibility, the microkernel approach presents an alternative solution [80]. In order to make the security assumption sound, the underlying hypervisor must be robust, functionally correct without undefined behaviors, and preferably have a TCB as small as possible. For these reasons, the formally-verified seL4 microkernel presents an excellent solution for implementing a microkernel-based hypervisor, so-called microvisor [69].
A Microkernel is a minimal system base for providing hardware abstraction to build various system services. A hypervisor is designed solely for the purpose of managing underlying hardware for multiplexing resources among various virtual runtime environments, which includes both user-land services and kernel code. Due to the similarity functionality and structure, a microkernel can be used to implement a hypervisor. The L4 microkernel family has been built as a microvisor to support both para-virtualization and hardware assisted virtualization for a long time. Both OKL4 and L4 Fiasco (L4Re) have demonstrated preeminent results as hypervisors in industry and academia [24]. However, none of any existing microkernels can provide the same level of assurance compared to the seL4. The seL4 community has developed a proof-of-concept seL4-based microvisor for the ARM Cortex-A15 processor. Leveraging on the hardware assisted virtualization, Data61 released a library for constructing VMM. For those reasons, this research uses the seL4 microvisor as the fundamental platform for implementing the BAS controller prototype.

8.2.2 Policy-enforced Mandatory Access Control

Microvisor isolates control tasks into various virtual machines, but in the BAS domain, many control tasks also need to communicate and cooperate together to achieve the daily operations. Therefore, a mandatory access control mechanism that enables developers to enforce fine-grained communication policies only on those tasks that are allowed to cooperate is necessary. The seL4’s capability-based access model provides a flexible method to transparently mediate system-wide security policies between virtual environments. In seL4’s capability-based design, everything is an object. Each kernel object represents a share of resources, e.g., memory pages, device mapped regions, communication channels, etc. The capability is an unforgeable token with associated access rights that refers to the corresponding kernel object. Each capability can be assigned to a specific process. The kernel manages the capability token on behalf of processes. When an object is created the kernel returns an index in the owner process capability table, which can be used by the process to refer the actual kernel objects and invoke its functions. Kernel protects this capability table, therefore it is unforgeable.

For two processes to communicate with each other, processes need to possess the capability of the communication channel, an IPC endpoint. Hence, if processes do not own such a capability,
they cannot communicate. Essentially, the system-wide security policies are enforced in form of capability distribution that can be pre-configured statically. To ease the development, the seL4 community provided a domain specific language, called capDL for describing the capability distribution of seL4 systems. The capDL reads the specification and translates it to low-level C code. This work relies on the capDL to create and distribute both capabilities of IPC endpoints, and shared memory pages between processes and virtual machines. The generated capability distribution is built in to the first user-land process, \textit{init}, which also serves as a process loader during booting.

8.2.3 Secure Execution Environment

Using the seL4 microvisor and the policy-enforced mandatory access control, we can set up two runtime environments, the native seL4 partition, and the Linux virtual machine. The native partition is the runtime environment that directly runs on top of seL4 with access to kernel API. The partition provides strong isolation and higher security due to the small TCB, and high-assurance capability-based access model, enforced directly by the seL4 kernel. Applications in this partition only depend on a thin layer of code in the form of a seL4 process with the support of existing system services and device drivers. However for the same reason above, there is no POSIX-compliant system calls that exist yet, and there are limited device drivers. Hence, the application has to be implemented in a unique way. On the other hand, the Linux partition provides a rich runtime for existing legacy or low criticality applications. With the hardware assistance, the unmodified Linux kernel runs as a seL4 process. Because conventional Linux kernel does not link with seL4 libraries and executes in its own memory space, it does not aware of the underlying seL4, and therefore impossible to communicate with applications run in the native partition. Applications in the Linux environment is the same as if they are running in a conventional Linux system. They can access standard APIs, system calls, libraries \textit{etc.} and most of them can execute without porting. Furthermore, some of the device drivers in the Linux kernel can also be reused to alleviate the absence of device drivers in seL4.

The policy-enforced communication between the Linux partition and the native partition can be achieved through a loadable kernel module. Linux kernel modules are pieces of binary code that can be dynamically loaded and unloaded during runtime. Linux kernel modules run in Linux kernel
space. It extends the kernel functionality. In this setup, one can build a communication kernel module that is aware of the seL4 capability model and kernel calls. When an application in Linux needs to communicate with native services or processes, the application can invoke the registered system calls, and through this kernel module messages or data can be sent or received from an IPC endpoint.

In this way, this solution balances usability and security by hosting both Linux execution environments and the high-assurance native environment for practical purposes. Most importantly, in the effort to ease the process of migration from the traditional general-purpose Linux to the microkernel driven high-assurance system for controllers in the BAS domain, this solution enables an incremental strategy to focus on specific functions first, while allowing the rest of the system to continue to be built.

8.3 Experiment Implementation

This section documents the implementation of the proposed prototype using the seL4 microvisor. Using the open-source seL4 microvisor, this solution sets up the platform with two runtime environments, one Linux partition realized by a customized Linux virtual machine, and one native partition with seL4 native processes. Additionally, I implemented a secure boot chain system to protect software integrity. The secure boot procedure applies the cryptographic methods in the booting process, which verifies each component’s digital signature and guarantees only authorized components can be loaded in the platform.

8.3.1 Prototype

For the development, the prototype uses an NVIDIA Jetson Tegra K1 System-on-Module (TK1-SOM) from Colorado Engineering Inc. as the target platform. The TK1-SOM board carries an NVIDIA Tegra K1 processor which features ARM Cortex-A15 Quad core. Besides the Tegra K1 CPU, it also includes 8GB memory, 32GB eMMC flash, with rich I/O (USB 3.0, Ethernet, GPIO, etc.). There are two reasons why I chose this board. First, the ARM Cortex-A15 processor is the first ARM processor that supports ARM Virtualization Extensions. Secondly, the board is directly
supported by both the Linux and the seL4 community. Therefore, it saves the effort of porting and accelerates the development process.

Using the Data61’s proof-of-concept seL4-based microvisor solution, the prototype is structured as shown in Figure 8.1. There are 5 major components, the capDL-loader, a virtual machine monitor, a guest Linux OS, and a communication server. The capDL-loader is the root process and user-land initializer for seL4 system [81]. As part of the capDL toolkit, the capDL-loader contains functions that are translated directly from the capDL language during compiling time. It takes the description from generated C files from system specification and calls seL4 kernel to allocate resources, such as creating IPC endpoints, process control blocks, and distribute capability to the right owner. Another purpose of this init process, is to load programs into the system from ELFs in a combined CIPO archive. Finally, after all the tasks during the booting process are done, this init starts all other process and hibernates.

![Figure 8.1: Secure execution environment using seL4 microvisor](image)

Another important component is the virtual machine monitor (VMM). The VMM is the software that creates the virtual environment. In the prototype, the VMM is realized using CAmkES (Component Architecture for Microkernel-based Embedded Systems) runtime framework a native
seL4 process. CAmkES is designed with the intention to facilitate the process of building applications and servers using the seL4 microkernel. Due to the uniqueness of seL4, it is different from the traditional Unix-like system. Specifically, applications in the seL4-based system require the use of IPCs frequently, manually implementing all the process interactions and keeping track of capability, which is not trivial. The CAmkES follows a component-based approach which models the system as a set of interacting software and provides various predefined modules for developers to abstractly specify the communication without worry about detailed implementation [82].

The VMM monitors the system events and interrupts for the guest OS, and traps privilege instructions in order to pass it to the seL4 system. During boot time, VMM initializes the virtual GIC, and virtual CPU, configures direct memory mapping and passes through hardware. After the environment is setup, the VMM would load the guest OS kernel, the device tree, and the root filesystem (a Busybox ramdisk) in the memory and pass control to the kernel. During runtime, the VMM manages guest OS’s capabilities, virtual memory spaces, interrupts, and saves/restores CPU states for context switches, etc.

For the Linux guest OS, the system runs a customized conventional Linux that is cross-compiled for the target board. The Linux kernel and the Busybox ramdisk image are built through the Buildroot embedded Linux build system, one of the popular tools that automate the process. The Linux contains a kernel module, which when loaded in the system will register as a virtual device that can interact with Linux applications through \texttt{ioctl()} system call.

The last component of the system is a communication server. The communication server is a seL4 native process. The purpose of this process is for conducting data sharing between Linux processes that are running in the virtual environment and native applications. The communication server is also a CAmkES component. The system configures a shared memory page between the virtual machine and the communication server. For each pair of communications between the communication server and other native seL4 processes, a dedicated IPC endpoint would be used. The communication server serves as a dispatcher. When a message is received either from the shared memory or any endpoint, the message will be queued and forwarded to the corresponding receiver.
8.3.2 Secure Boot

Although the virtualization solution is good for compartmentalizing legacy and potentially vulnerable applications, it does not prevent attackers from physically tampering with the device or modifying the boot process to load a malicious OS. The countermeasure is the secure boot [83]. The secure boot is a critical security feature to ensure that only authorized (digitally signed) software can be booted from the device. The secure boot ties the software to the underlying hardware, by applying cryptographic methods such as digital signatures in each step of the system booting process, starting from the moment the hardware is initialized. Since only properly signed binary can be verified and loaded in the system, the secure boot makes sure only authorized software can execute on the specific hardware. Using this solution, the software can delegate the trust through the so-called “chain of trust”, all the way from bootloader to user-land applications. There are various standard secure boot solutions that offer both specification and implementation, such as Unified Extensible Interface specification, U-boot’s verified boot, etc. [84] In this experiment, the TK1 board uses U-boot as the bootloader. I implemented a secure boot procedure that allows each component in the system to verify the next one during the booting process and incorporate it with U-boot’s verified boot. The seL4 on ARM booting process is similar to the Linux booting process. For the specific development board, the booting procedure starts from TK1 Boot ROM when powered on. The booting process is in the following order (most ARM SoMs follow the similar steps):

1. TK1 boot ROM, a vendor owned custom boot ROM. The boot ROM initializes RAM, loads secondary bootloader into memory, and passes control to the secondary bootloader, U-boot, an open-source secondary bootloader for embedded devices.

2. U-boot reads its environment variables from the boot device, detects hardware resources, configures CPU to HYP mode, and loads the ELFloader (seL4 bootloader), the seL4 kernel, and the root task process into RAM before passes control to the ELFloader.

3. ELFloader moves the seL4 kernel, and the root task (CapDL-loader) into the right physical memory address and configures boot variables for seL4. Then, it passes control to seL4.
4. seL4 starts execution. It configures hardware resources and schedules the root task in user mode before it makes the capDL-loader runnable.

5. CapDL-loader loads other user mode applications and distributes resources in the form of capabilities. In this specific prototype, those processes are the communication server and the virtual machine monitor.

6. VMM configures the virtual execution environment for the guest OS, loads Linux kernel binary, ramdisk, and the device tree blob into memory and passes control to the Linux kernel.

Following the boot process, I implemented the proof-of-concept secure boot procedure starting from the ELFloader in the seL4 system. First, I inserted a public key in the ELFloader and used the private key signed seL4 kernel and the capDL-loader, and appended their signatures on the end of each binary. Before loading, the ELFloader calculates the hash of each binary and checks the digital signature against the hash value. The ELFloader will only continue if the signature matches. Then, I included a similar procedure in the init process, capDL-loader to make sure the capDL-loader will verify the communication server and the VMM. Finally, the verification procedure is added in the VMM, so that only signed the Linux kernel, ramdisk, and the device tree blob would be executed on the board.

Additionally, this work integrates this with U-boot’s verified boot. U-boot introduced their implementation of secure boot called verified boot, which allows developers to embed cryptographic keys in the bootloader for verification purpose. Also, U-boot’s verified boot uses a new file format, called flattened image tree (FIT), a flexible, structured container format which supports multiple kernel images, device trees, and ramdisks. With this format, it is easy to add signatures and other metadata to instruct U-boot how to load the system. I compiled the seL4 and its components into a FIT image to utilize this mechanism.

In production, many SoC vendors, such as Xilinx, NXP support secure boot specification compatible secure boot solutions (e.g., NXP High Availability Boot v4). They often provide secure boot enabled boot ROMs along with additional hardware storage (Xlinux uses BBROM storage) and eFuses for programming unpurgeable encryption key. Working with vendors, one can embed our
proprietary cryptographic key on the board in the future. Such key can be used to sign a secondary bootloader, such as U-boot to allow only our authorized U-boot with the desired configuration can be executed in the board for an end to end guarantee.

8.4 Evaluation

To evaluate the security and practicality of this solution in the BAS domain, the prototype adopted an IoT gateway scenario from Honeywell business and implemented in the aforementioned system. This section explains the scenario I implemented and the security analysis of this approach.

8.4.1 IoT Gateway

Working with Honeywell engineers revealed that one of the security-critical components in the next generation BAS is the IoT gateway. While many BAS devices get replaced by their latest counterparts for providing more “intelligent” controls, devices rely more and more on cloud-based computing services. The IoT gateway is a device that gathers and stores data from the field (office, home, factory etc.) and provides communication between local systems and the cloud. As an edge device which serves an entry point in bridging the Internet, the home, and industrial building automation systems, IoT gateway plays a key role in the convergence of traditional BAS and Internet of Services. Hence, its security is inextricably linked to the safety and security of the facility.

Among all of the functionalities in IoT gateway, I have been told that the biggest concern for business is protecting the authentication key which is used for authentication and providing secure communication between the gateway and the cloud. In an effort to demonstrate the security benefits of the proposed solution in the BAS domain, this experiment uses the key management as an example for evaluating the security benefits of the microvisor solution.

In conventional systems, the key management relies on file system and discretionary access control to protect cryptographic keys as shown in Figure 8.2. During runtime, the key management server would load the key in its space. The key management server is responsible for providing cryptographic services (e.g., encryption, decryption, digital signing, signature verification, etc.) for authorized peer process, such as the cloud client. When needed, the cloud client will communicate with the key management server using UNIX local domain socket. To protect those cryptographic
keys from malicious access, in this implementation I moved the key management from Linux partition to the seL4 native partition. During boot time, the keys are only available in the seL4 partition and are loaded in the key management server. The cloud client is hosted in the Linux partition. When the cloud client requires cryptographic services, the cloud client would issue a system call, like asking for the kernel’s or drivers’ service. The kernel module would check the privilege of the process. If the request is granted the kernel module would forward the request to the communication server in the seL4 partition by invoking the corresponding capability using seL4 IPC primitive. Eventually, the communication would be passed to the key management server and the result would be returned to the cloud client on a similar path. From the Linux application’s point of view, the actual implementation is transparent, hence very little modification in the source code is needed. Figure ?? shows the logic of architecture and communication path.

![Figure 8.2: Cloud connector in conventional systems](image)

8.4.2 Security Evaluation

To assess this approach, I analyzed possible attack vectors in comparison with conventional implementation. Based on existing real-world attacks, our threat model assumes 2 level attacks. First, attacks remotely with possible compromised processes, secondly, attack with physical access of the device. Due to the functional correctness guarantee provided by seL4’s formal verification, this work assumes the microvisor is high-assurance and does not contain software vulnerabilities.
Figure 8.3: Cloud connector in secure execution environment

Considering the IoT gateway is an edge device in the conventional system, if there are vulnerabilities in one of the applications running in Linux partition, attackers might be able to gain remote access by exploiting such weaknesses. Once the attackers gain access to the device, it is possible for attackers to access cryptographic keys directly from file system if the file permission is not configured correctly or through further privilege escalation. However, this threat is mitigated by the microvisor solution. Because cryptographic keys and the key management server are hosted in the seL4 native partition, as long as the fundamental IPC and memory management in seL4 is not breached, even attackers with Linux root privilege cannot obtain the keys.

In conventional systems, if not configured properly, a malicious process might be able to spoof local IPC and abuse the key management server without obtaining the keys. This kind of attack, although is possible, is easier to mitigate in the microvisor solution. In the microvisor solution, if a malicious application wants to request services from the key management server, it would need to invoke the customized system call. The kernel module which processes the system call acts as a proxy and would be able to scrutinize the request. Because the module is running in kernel space with high privilege, developers can implement the module to check the identity of the caller process. As long as the kernel is not compromised, and the kernel module inserts a proper security
check before authorizing any request, the attackers cannot abuse the services. If the attackers can compromise the Linux kernel, then it is possible for them to spoof request and abuse the services as if they own the key. However, if we assume the attackers can compromise the Linux kernel, they basically own the system and can do far more worst things. Hence, in comparison, the microvisor solution is still superior from the system security perspective.

On the other hand, if the attackers have physical access to the device, more attack vectors have to be considered. In the conventional system, attackers might be able to replace the booting image and launch a malicious version by which they have full control to access the storage and obtain the keys. With the support of the secure boot, such attackers can be eliminated, because unsigned kernel images and applications cannot be loaded. In order to fully protect the system from physical tampering, in both of those solutions, additional security mechanisms, such as whole disk encryption, hardware cryptographic chips, e.g., a Trusted Platform Module, have to be applied. However, such mechanisms are easier to adopt in the seL4 native partition due to the modular design in seL4, and because there is no legacy dependency in such a system.

8.4.3 Performance

In addition to the security analysis, I also conducted a performance evaluation. The seL4 microkernel has been proven to be the world’s fastest microkernel [85]. As [61] shows the IPC in seL4 on ARM architecture is within a couple hundred nanoseconds (ARMv7 Cortex-A9 353 nanoseconds for sending a single, minimal-length message between two threads). Our early implementation is on an i.MX6 board which uses the same processor. The benchmark result shows transfer of data up to 512 bytes between two threads, in round trips, takes about 3 microseconds, which is faster than conventional Linux on the same setup (around 50 microseconds for Linux). Although it is debatable whether such comparison is fair due to the architectural differences, this benchmark shows seL4 is the fastest among microkernels and microkernel architecture does not imply inferior performance.

To evaluate the virtualization performance, I compared the native Linux system performance and the virtualized Linux using a widely used system performance benchmark tool, called lmbench. The lmbench is a micro-benchmark suite that is designed to measure commonly found performance bottlenecks [86]. Table 8.1 shows the comparison on the NVIDIA Tegra K1 board. The TK1 consists
of 4 ARM Cortex-A15 cores, clocked up to 2 GHz. Because the current seL4 microvisor does not support multicore yet, the native Linux measurement is adjusted to run on one core as well. To show a fair comparison, both of those setups are clocked at 696MHz.

With the ARM virtualization extensions, the only interactions between VM and the microvisor are context switches between two partitions, VM entries, exits, and interrupts. Hence the overhead is minimal. This estimation matches our benchmark results, the VM shows nearly native performance. Many micro-benchmarks even shows negative overhead, such as write, pipe latency, and UNIX socket. Those results likely happens because the VM was using memory mapped storage, rather than MMC storage or cache related conflicts. Our result matches the result shown in [87], in which Gernot et al. used the RapiLog database system and demonstrated that on average the degradation of a virtual machine can be as low as 9%. Virtualization overheads highly depend on workload and hardware platform. This benchmark is not a rigorous performance comparison between native Linux and seL4 microvisor. This is merely to show that the performance of seL4 microvisor with ARM virtualization extensions is not going to be a bottleneck in practical adoption.

Table 8.1: Lmbench results for seL4 microvisor on TK1 clocked at 696 MHz

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Native</th>
<th>VM</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>null syscall</td>
<td>0.6043 µs</td>
<td>0.6993 µs</td>
<td>0.0950 µs</td>
</tr>
<tr>
<td>read</td>
<td>0.7984 µs</td>
<td>1.2275 µs</td>
<td>0.4391 µs</td>
</tr>
<tr>
<td>write</td>
<td>1.1256 µs</td>
<td>0.8180 µs</td>
<td>-0.3076 µs</td>
</tr>
<tr>
<td>stat</td>
<td>4.8211 µs</td>
<td>5.5214 µs</td>
<td>0.7003 µs</td>
</tr>
<tr>
<td>fstat</td>
<td>1.1115 µs</td>
<td>1.2505 µs</td>
<td>0.1290 µs</td>
</tr>
<tr>
<td>open/close</td>
<td>12.4576 µs</td>
<td>20.0 µs</td>
<td>7.7424 µs</td>
</tr>
<tr>
<td>select(10)</td>
<td>1.4831 µs</td>
<td>2.3302 µs</td>
<td>0.8471 µs</td>
</tr>
<tr>
<td>select(100)</td>
<td>7.0216 µs</td>
<td>7.8527 µs</td>
<td>0.8311 µs</td>
</tr>
<tr>
<td>signal install</td>
<td>1.3617 µs</td>
<td>1.6809 µs</td>
<td>0.3202 µs</td>
</tr>
<tr>
<td>signal overhead</td>
<td>6.0025 µs</td>
<td>7.0985 µs</td>
<td>1.0960 µs</td>
</tr>
<tr>
<td>protection fault</td>
<td>1.7337 µs</td>
<td>1.0073 µs</td>
<td>-0.7264 µs</td>
</tr>
<tr>
<td>pipe latency</td>
<td>38.4235 µs</td>
<td>20.6930 µs</td>
<td>-17.7305 µs</td>
</tr>
<tr>
<td>UNIX socket</td>
<td>44.0403 µs</td>
<td>30.4076 µs</td>
<td>-13.6327 µs</td>
</tr>
<tr>
<td>fork+exit</td>
<td>847.6667 µs</td>
<td>939.0 µs</td>
<td>91.3333 µs</td>
</tr>
<tr>
<td>fork+execve</td>
<td>4088.0 µs</td>
<td>3623.0 µs</td>
<td>-785.0 µs</td>
</tr>
</tbody>
</table>
CHAPTER 9: CONCLUSIONS

This dissertation describes a secure computing platform for the next generation building automation system. In the process of study the problem domain, I conducted an empirical study of two real-life building automation systems; analyzed the various potential security and safety risks; and proposed a microkernel-based secure computing platforms in an attempt to address those security issues for the next-generation BAS. With the consideration of compatibility with existing systems, this solution leverages the microkernel operating system architecture and applies mandatory access control and proxy-based network communication for enforcing the global policies in the automation controllers in a simple and clear way. Compared to other approaches, this solution has a sound and robust foundation, and most importantly, it provides a backward compatible method for incremental development. It is robust because of the microkernel’s minimal functionality in the kernel space, and modular architecture that can be mathematically proven to be functionally correct. It is backward compatible since legacy applications can be hosted in a virtual environment and the proxy design secures BAS network communication protocol through network tunneling. For evaluating the proposed solution, various building automation scenarios have been implemented, such as the laboratory temperature control scenario, key management scenario, and their corresponding attacks. These experiments show the security benefits of the proposed solution and tangible performance.

Additionally, in order to help businesses in the building automation domain to adopt this approach, and migrate their legacy system to the secure platform without completely changing their existing software, this dissertation presents a virtualization approach that uses the formally verified seL4 as a microvisor for hosting legacy BAS systems. This approach is rooted in facilitating business adaption while balancing the cost-security tradeoffs. Furthermore, it leverages the formally verified microkernel seL4 for its unique advantages of mathematically proven security guarantee, and recent progress in ARM virtualization technology. Using seL4 microvisor with a virtualized Linux,
this method provides two partitions: one for hosting legacy software with rich functionality, the other directly running on top of the high assurance seL4 environment for hosting safety/security-critical applications. Hosting a Linux virtual machine on top of seL4, allows developers to secure critical elements first and leave the rest of the system mostly unchanged. Most importantly, the system can evolve with time, as the products change.

There are several limitations of this approach in practice. First, the mandatory access control through an access control matrix requires kernel modification, every time new policies are introduced. This is partially due to the design of MINIX 3 kernel IPC primitives, and the microkernel structure. However, there are multiple efforts to make the process more transparent, and automatic. For example, one promising solution is to automatically generate the access control matrix, proxy process, and application code skeleton from the system model. Collaborating with researchers from Kansas State University, we used AADL to model the control tasks, policies in BAS, and applied Dr. Robby’s Slang/Logika program verifier to automatically generate the ACM, and proxy process in C. The approach is still in its preliminary stage but it has shown promising results. On the other hand, seL4’s capability-based model shows another promising approach. By applying for the capability-based access in the kernel, the policy can be implemented in user-land, which is more flexible. The proposed platform is still in the early stage, which lacks robust device drivers, and libraries. More work is needed in porting existing drivers and implementing wrapper libraries to make the system easier to use, as well as optimizing performance, and efficiency.
REFERENCES


Copyright permission below is for the use of material in Chapter 3 and Chapter 5. The publishers state that authors can reuse any portion of their own work in a new work of their own (and no fee is expected) as long as a citation and DOI pointer to the Version of Record in the ACM Digital Library are included.
ACM Information for Authors

ACM Author Rights

ACM exists to support the needs of the computing community. For over sixty years ACM has developed publications and publication policies to maximize the visibility, impact, and reach of the research it publishes to a global community of researchers, educators, students, and practitioners. ACM has achieved its high impact, high quality, widely-read portfolio of publications with:

- Affordably priced publications
- Liberal Author rights policies
- Wide-spread, perpetual access to ACM publications via a leading-edge technology platform
- Sustainability of the good work of ACM that benefits the profession

**CHOOSE**

Authors have the option to choose the level of rights management they prefer. ACM offers three different options for authors to manage the publication rights to their work.

- Authors who want ACM to manage the rights and permissions associated with their work, which includes defending against improper use by third parties, can use ACM’s traditional copyright transfer agreement.
- Authors who prefer to retain copyright of their work can sign an exclusive licensing agreement, which gives ACM the right but not the obligation to defend the work against improper use by third parties.
- Authors who wish to retain all rights to their work can choose ACM’s author-pays option, which allows for perpetual open access through the ACM Digital Library. Authors choosing the author-pays option can give ACM non-exclusive permission to publish, sign ACM’s exclusive licensing agreement or sign ACM’s traditional copyright transfer agreement. Those choosing to grant ACM a non-exclusive permission to publish may also choose to display a Creative Commons License on their works.

**POST**

Authors can post the accepted, peer-reviewed version prepared by the author-known as the "pre-print"-to the following sites, with a DOI pointer to the Definitive Version of Record in the ACM Digital Library.

- On Author's own Home Page and
- On Author's Institutional Repository and
In any repository legally mandated by the agency funding the research on which the work is based and

On any non-commercial repository or aggregation that does not duplicate ACM tables of contents, i.e., whose patterns of links do not substantially duplicate an ACM-copyrighted volume or issue. Non-commercial repositories are here understood as repositories owned by non-profit organizations that do not charge a fee for accessing deposited articles and that do not sell advertising or otherwise profit from serving articles.

**DISTRIBUTE**

Authors can post an Author-Izer link enabling free downloads of the Definitive Version of the work permanently maintained in the ACM Digital Library

- On the Author's own Home Page or
- In the Author's Institutional Repository.

**REUSE**

Authors can reuse any portion of their own work in a new work of their own (and no fee is expected) as long as a citation and DOI pointer to the Version of Record in the ACM Digital Library are included.

- Contributing complete papers to any edited collection of reprints for which the author is not the editor, requires permission and usually a republication fee.

Authors can include partial or complete papers of their own (and no fee is expected) in a dissertation as long as citations and DOI pointers to the Versions of Record in the ACM Digital Library are included. Authors can use any portion of their own work in presentations and in the classroom (and no fee is expected).

- Commercially produced course-packs that are sold to students require permission and possibly a fee.

**CREATE**

ACM's copyright and publishing license include the right to make Derivative Works or new versions. For example, translations are "Derivative Works." By copyright or license, ACM may have its publications translated. However, ACM Authors continue to hold perpetual rights to revise their own works without seeking permission from ACM.

- If the revision is minor, i.e., less than 25% of new substantive material, then the work should still have ACM's publishing notice, DOI pointer to the Definitive Version, and be labeled a "Minor Revision of"

- If the revision is major, i.e., 25% or more of new substantive material, then ACM considers this a new work in which the author retains full copyright ownership (despite ACM's copyright or license in the original published article) and the author need only cite the work from which this new one is derived.

Minor Revisions and Updates to works already published in the ACM Digital Library are welcomed with the approval of the appropriate Editor-in-Chief or Program Chair.

**RETAIN**

Authors retain all perpetual rights laid out in the ACM Author Rights and Publishing Policy, including, but not limited to:

- Sole ownership and control of third-party permissions to use for artistic images intended for exploitation in other contexts
- All patent and moral rights
Ownership and control of third-party permissions to use of software published by ACM

Have more questions? Check out the FAQ.

back to top
Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.