Enhanced Security of Building Automation Systems Through Microkernel-Based Controller Platforms

Xiaolong Wang[*], **Richard Habeeb**[*], **Xinming Ou**[*]
Siddharth Amaravadi[†], **John Hatcliff**[†], **Masaaki Mizuno**[†], **Mitchell Neilsen**[†]
S. Raj Rajagopalan[‡], **Srivatsan Varadarajan**[‡]

[*]University of South Florida, Department of Computer Science and Engineering
[†]Kansas State University, Department of Computer Science
[‡]Honeywell

***Abstract*—A Building Automation System (BAS) is a complex distributed Cyber-Physical System that controls building functionalities such as heating, ventilation, and air conditioning (HVAC), lighting, access, emergency control, and so on. There is a growing opportunity and motivation for BAS to be integrated into enterprise IT networks together with various new "smart" technologies to improve occupant comfort and reduce energy consumption. These new technologies coexist with legacy applications, creating a mixed-criticality environment. In this environment, as systems are integrated into IT networks, new attack vectors are introduced. Thus, networked non-critical applications running on the OS platform may be compromised, leaving the control systems vulnerable. The industry needs a reliable computing foundation that can protect and isolate these endangered critical systems from untrusted applications.**

This work presents a novel kernel-based approach to secure critical applications. Our method uses a security-enhanced, microkernel architecture to ensure the security and safety properties of BAS in a potentially hostile cyber environment. We compare three system design and implementations for a simple BAS scenario: 1) using the microkernel MINIX 3 enhanced with mandatory access control for inter-process communication (IPC), 2) using seL4, a formally verified, capability-based microkernel, and 3) using Linux, a monolithic kernel OS. We show through experiment that when the non-critical applications are compromised in both MINIX 3 and seL4, the critical processes that impact the physical world are not affected. Whereas in Linux, the compromised applications can easily disrupt the physical processes, jeopardizing the safety properties in the physical world. This shows that microkernels are a superior platform for BAS or other similar control environments from a security point of view, and demonstrates through example how to leverage the architecture to build a robust and resilient system for BAS.

I. Introduction

A Building Automation System (BAS) is a network-based control system that monitors and manages physical security, fire safety, air conditioning (HVAC), humidity, lighting, *etc.*, for modern buildings. As the industry is moving towards Cyber-Physical Systems (CPS) and the Internet of Things (IoT), there is a growing motivation to develop new technologies for BAS to improve occupant comfort and at the same time reduce energy consumption and operating costs. Thus, more and more "smart building" devices are entering the market. These commercial products leverage various sensors and actuators to better understand the living context, automate the control environment, and respond to customer needs. The goal of CPS is to enable more advanced sensing, actuation, and controls for better energy and operational efficiency. In turn, the resulting requirements of interconnection and integration increasingly sophisticates computing and networking within the BAS, opening new attack surfaces that endanger the safety of the physical world under that BAS's control. State-of-the-art BAS have many networked entities, which were separated in the past from the IT systems in the environment.

Today's control network for building automation is outdated. It is well-known, the security of BACnet, one of the most popular communication protocols in BAS, is vulnerable to diverse, common network-based attacks such as denial-of-service (DoS) attacks, replay attacks, spoofing attacks, *etc.* More importantly, BAS devices, such as PLCs, global controllers, and management computers, are themselves potentially vulnerable to attacks. Modern BAS heavily depend on PLCs for monitoring and controlling building facilities. A PLC is an embedded real-time computer that connects with various sensors and actuators; it can be remotely monitored and programmed through the BAS. PLCs have traditionally been on isolated internal networks, so developers have focused on maintaining functionality rather than security. Prior to the discovery of Stuxnet attack in June 2010 [1], the security of PLCs received little attention.

Since then numerous recent incidents have been reported regarding cyber attacks on industrial control systems. For example, in 2012, ICS-CERT issued an alert that documented various PLC vulnerabilities, including the possibility of uploading unauthenticated configuration changes to the PLC with arbitrary code [2]. Recent work [3] clearly demonstrated that the technique required to attack a PLC is no more difficult than attacking any other computer. In another example, Charlie Miller [4] demonstrated a remote exploit for advanced vehicles. In 2014, a cyberattack on a German steel mill plant control system caused physical damage [5]. All of these demonstrate the possible threat and potential impact of cyber attacks on comparable systems. What is there to prevent a similar attack from happening in building automation systems?

BAS hardware components are often expected to stay in place for a decade or more, and any approach to secure BAS

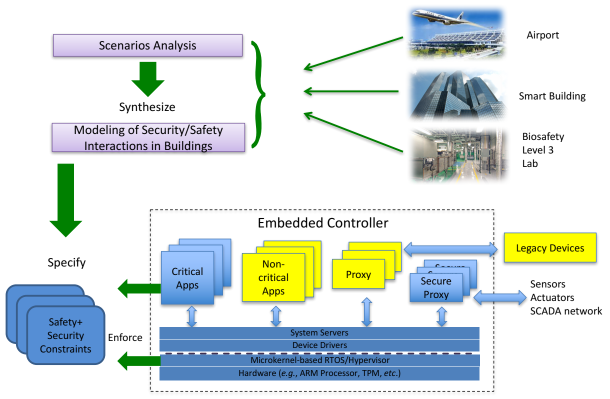

Fig. 1: Building Security/Safety Control Framework

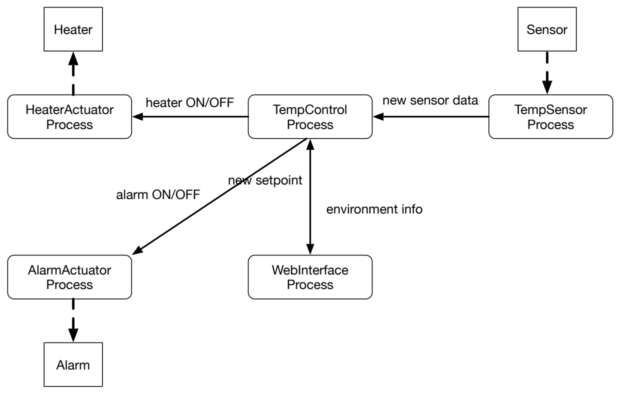

Fig. 2: Logic Illustration of Temperature Control Scenario

must accommodate the long field life of control hardware. Thus, new building technologies must coexist with legacy systems. This situation creates a mixed criticality environment. In this environment, less critical applications often share the same resources as critical control processes on the same device. This is problematic because the network functionality also opens up an additional attack surface. This calls for a solution that can regulate the communication between critical applications and non-critical applications.

We believe one promising direction is to transform BAS microcontrollers into security anchors that protect the critical control components. Even if part of the overall system is compromised, the critical functionalities of the controllers should not be jeopardized. This requires a simultaneous top-down and bottom-up approach as illustrated in Figure 1. At the top level, the control environments are decomposed into components through a modeling approach. From the bottom, a re-designed computing foundation of the microcontrollers provides a succinct set of security primitives that support the diverse set of safety and security properties that are required by the high-level environment. This framework will enable decomposing the domain-specific security/safety properties into the various isolated modules with strongly enforced timing, spatial, and networking isolations that run on the embedded controllers. Ultimately, we want to guarantee the critical tasks a controller must fulfill will never be jeopardized even in a potentially malicious environment. Through our investigations, we believe that microkernel-based real-time OSs are a good approach to build this underlying computing foundation.

In this paper, we document our initial experiment towards the proposed framework. We implemented a simplified temperature control scenario that is extracted from our case study of the Biosecurity Research Institute at Kansas State University [6]. We designed and implemented this scenario using two microkernel-based operating systems: MINIX 3 [7] with our newly added mandatory access control security mechanism, and the formally verified seL4 [8]. In the following sections, we will explain the system and the experiment in detail.

II. Real Scenario

In this section, we present a simplified subset of a building automation control scheme. We extracted this temperature control scenario from our case study of the Biosecurity Research Institute at Kansas State University in our previous work [6]. The simplified temperature control scenario involves a single controller, one temperature sensor, one heater actuator, and one alarm actuator. The goal of this controller is to maintain the room temperature within a predefined range. The controller should allow an administrator to adjust the desired room temperature within this range through a web-based interface. If the controller fails to achieve the desired temperature within certain time interval(*e.g.*, 5 minutes), the alarm will be triggered to alert the occupants.

The control scenario contains multiple components that can be implemented into five processes using a microcontroller. The logic illustration and control components can be seen in Figure 2.

- **Temperature Control Process** is the major control logic responsible for maintaining the room temperature according to the desired temperature setpoint. This process periodically receives the current room temperature sensor data from the sensor process. Based on the sensor data, it sends control commands to the heater driver and to the alarm driver. The temperature control process also listens for setpoint updates from web interface.
- **Temperature Sensor Process** is the temperature sensor driver. This process periodically samples the room temperature and sends the data to temperature control process.
- **Heater Actuator Process** is the heater actuator driver. This process controls the physical heater actuator. It listens for commands from temperature control process and adjusts the actuator accordingly.
- **Alarm Actuator Process** is the alarm actuator driver. This process controls the physical alarm actuator. It listens for the control command from temperature control process and enables or disables the alarm accordingly.
- **Web Interface Process** simulates the building automation management interface. This interface also provides

administrators a way to change the desired room temperature setpoint. Realistically, the control interface might support other control systems, such as lighting, air-conditioning, fire alarm, *etc*.; however, for the sake of simplicity in our mockup scenario we only consider the room temperature control system.

In this scenario, we assume the drivers are implemented correctly without vulnerabilities, and the control logic of the temperature control process is functionally correct with intuitive implementation. On the other hand, the web interface process does not hold any security guarantee, which means it might be vulnerable to various attacks, such as spoofing, DoS, and buffer overflows.

If the scenario is implemented on a typical Unix-like system, an adversary has many available attack options. First, all the inter-process communication is potentially vulnerable to spoofing attacks. 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. This mechanism conveniently provides a uniform way to manage IPC, and if configured correctly, it can satisfy basic security requirements. However, it cannot prevent attacks with root privilege. Secondly, a monolithic kernel architecture includes most of its services, such as process management, virtual memory management, file system, *etc*. in the kernel space as a whole. These monolithic systems have few techniques to restrain a process with root privilege in those environments. For example, a process with root privilege can invoke system calls to kill other processes.

III. System Overview

In order to evaluate the proposed microkernel architecture, we implemented the temperature control scenario, as shown in Figure 2, on three different platforms to test the strength of the proposed secure OS architecture, and the advantage of the microkernel approach. Through our research, we developed our prototype on two generic microkernel-based platforms, the MINIX 3 [7] and seL4 [8]. In this section, we present the security enhancement on MINIX 3, which we call access control matrix (ACM), and how we translate the control scenario into seL4's capability system by using the CAmKES development framework [9]. We illustrate the implementation details and explain the key differences as well as the security benefits of our approach.

Microkernel architecture, in contrary to monolithic kernel architecture, where all OS services and drivers execute in the kernel address space, is designed in a modular flexible structure. In typical microkernel based operating systems, kernel only handles low-level functionality and process primitives such as interrupts, process control blocks (PCBs), and IPC. All other OS services and device drivers are running on top of the kernel 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 them, thus they are well isolated. Although the microkernel approach generally underperforms the monolithic due to the multiple context switches, from the security point of view, the kernel's absolute control over inter-process communication provides an advantage.

One important feature of microkernel based OS is the kernel facilitated IPC mechanism called message passing. In most Unix-like systems, the IPC options are either Unix domain sockets or message queues. Both of these Unix options use the virtual file system; thus are protected through file access controls. The solution though 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) [10]. In microkernel architecture, each data transfer operation across process boundaries goes through kernel, and the kernel can monitor each of those operations and control the requests. For example, each system call can be implemented to subject to strict permission checking. A process can be constrained to which system calls it can invoke and to which process it is allowed to exchange information with.

A. MINIX 3

MINIX is a well-known, ever-evolving standard microkernel based OS that emphasizes on reliability [7]. The latest version is MINIX 3 which is a full-fledged POSIX-compliant OS that targets embedded devices. In MINIX the kernel is only about 4000 lines of code. All other OS functionalities such as process management and virtual memory are implemented as modules running in user space. Its modular design means that each process is well isolated, and OS can be easily customized by adding or removing different servers and drivers to meet various requirements.

Because of the isolation provided by MINIX 3, the kernel handles the routing of all messages among all processes. Each message sent out will go through multiple context switches among caller process to kernel and kernel to callee process. In fact, in MINIX 3 all POSIX-compliant system calls such as *fork*, *kill*, *exit*, *etc*. can only be invoked by sending a message through kernel IPC primitives between the caller process and the process management (PM) process.

MINIX 3 IPC directly supports synchronous and asynchronous message passing, and memory grants. In MINIX 3, 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 PCB. The synchronous message passing uses a rendezvous-style mechanism. The synchronous *ipc_send() ipc_receive()* system call blocks until the message is delivered to the receiving process. In the current version, synchronous message passing is reserved for device drivers and system server components with designed communication protocols.

B. Security Enhancement in MINIX 3

In order to fully leverage the security advantages of micro-kernel architecture, we extended the MINIX 3 OS. First, we modified 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, we added three system calls in the process management server for improving IPC related operations.

Our second modification to MINIX 3 is on the process control block (PCB) data structure. We added a field called access control ID (ac_id) as well as added two more related system calls. *fork2()*, and *srv_fork2()* can uniquely 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 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.

Our final modification was to design and implement a fine-grained mandatory access control mechanism called access control matrix (ACM). 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, 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. We implemented 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. The 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 3. There are three processes in the example, App1, App2, and App3, two of which provide public remote procedure calls (RPCs). For App1 the RPCs *app1_f1()*, *app1_f2()*, and *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,

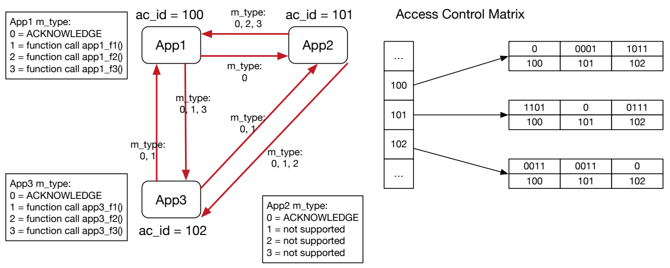

Fig. 3: Fine-grained IPC Using Access Control Matrix

message type 0 is reserved to indicate an acknowledgment to the caller.

We want to allow App2 access to App1's *app1_f2()*, *app1_f3()* functions, and we want *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, 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.

C. seL4

The seL4 kernel is the first mathematically verified microkernel [8]. All of seL4's 10,000 lines of code have been formally proven for functional correctness. While seL4 isn't a complete operating system, it provides an important layer of reliable task switching and virtual memory management. Eventually a completely POSIX-compliant operating system based on seL4 may emerge; however, such functionality isn't required for this case study.

We didn't need to make any modifications to seL4 for our experiment, since it already exposes the functionality needed to enforce IPC policy. With seL4, all access control policy, including IPC policy, is managed with *capabilities*. At a high level, a capability is a token which allows access to special kernel objects. These kernel objects could be page tables, thread control blocks, IPC endpoints, or many other types [8]. Capabilities are typically owned by executing threads, and the kernel enforces that no thread without the proper capability can access the corresponding object. When a thread makes a system call, it passes an argument identifying which capability should be invoked; the kernel then checks if that thread has that capability. Capabilities have wide general usage for seL4 systems; however, further discussion will only focus on aspects which are important to IPC.

The seL4 designers implemented IPC through capability primitives [8]. For example, two processes each with a capability to a common endpoint can communicate.[1] Some

[1]MINIX 3 endpoints are different from seL4 endpoints. In seL4, endpoints are implemented as wait queues or semaphores.

capabilities can also have specific rights associated with them, improving their flexibility. *read*, *write* and *grant* are the three rights allowed, and they can be used to regulate IPC communication. For instance, if a process has a read-only capability to an endpoint, it can only receive messages from that endpoint. The inverse is true for a write-only capability. In current seL4 revisions, the *grant* privilege only applies to endpoint capabilities; it allows any capability to be sent to another thread via IPC.

There are several different system calls used for seL4 IPC [8]. The pair *seL4_Send* and *seL4_Recv* will send and receive messages, but they will block if no other process is ready to send or receive. On the other hand *seL4_NBSend* and *seL4_NBRecv* are non-blocking variants of *seL4_Send* and *seL4_Recv*. If a thread is given *grant* access to an endpoint it can use *seL4_Call*. This system call invokes the kernel to attach a one-time reply capability to the message, so *seL4_Call* is an atomic send and receive. The receiving thread of a message with a reply capability can use *seL4_Reply* to send a reply message. These two system calls, *seL4_Call* and *seL4_Reply*, can be used to build RPC functionality.

The designers of seL4 wanted a minimal kernel where all access-control policy was specified in user space [11]. To do this, the kernel simply hands over all capabilities to the bootstrap process. In this way, a system developer can design her bootstrap process to implement the necessary IPC policy. This bootstrap process can create new processes and distribute capabilities to them to setup the architecture.

D. seL4 and CAmkES

Developing reliable, secure code for seL4 can be a challenging task because it is laborious to implement a system using capabilities. System designers need to translate a process architecture into a capability model. Using the capability model, designers must implement a bootstrap process to distribute the capabilities to the child processes. Boilerplate code must also be written for each process to use the capabilities to perform IPC. This entire process is prone to error, and it is also highly mechanical. In a recent paper, researchers theorized about using a tool called CAmkES (Component Architecture for microkernel-based Embedded Systems) [9] to automate the development of seL4 systems [12]. This tool, CAmkES, will generate all the boilerplate code that implements a specified process architecture. This boilerplate code, also called glue code, abstracts away seL4 capabilities from the developers, and it allows them to think about high-level design.

The glue code generated by CAmkES has two major components. The first is the bootstrap process; this part is generated using a language called CapDL. CapDL is a domain specific language used to describe capability-based systems [13]. For CAmkES, CapDL is used to describe the state of all the capabilities after bootstrap. With this language, then, a bootstrap process can be generated to implement the desired architecture [14]. The second part of the glue code is the user-level libraries which abstract IPC communication into RPCs.

Using CAmkES and seL4 we can build a system where the kernel enforces IPC policy. An important difference between seL4 and MINIX 3 then is where the IPC policy is defined. Because the IPC policy for MINIX 3 is defined in kernel space at compile time it cannot change at runtime (unless the kernel is exploited). Alternatively, seL4's IPC policy is defined in user space at runtime [11]. This means that the architecture could change at runtime if one process sends its capabilities to another process. SeL4 has one primary way for independent processes to share capabilities among them, the aforementioned grant access right.[2] The effects of this right depend on the scenario and will be discussed in the following section.

IV. Experimentation

We implemented the extracted temperature control scenario of Figure 2 on both the security enhanced MINIX 3 and seL4. Additionally, we use a similar implementation on Linux for comparison. This section explains the implementation details on all three OSs, and it also discusses our experiment to measure the impacts of various attacks. The first step in our experiment is to correctly interpret the control scenario. To properly study the scenario's communication scheme and operational attributes, we decided to construct a formal model of this simple scenario using AADL (the SAE Architecture Analysis Design Language) [15].

For this specific scenario, we model the logic components in the process level, and focuses on the IPC patterns. The whole scenario is modeled as a *system implementation* in AADL semantics. The system includes five *process components*: *tempProc*, *tempSensProc*, *heaterActProc*, *alarmProc*, and *webInterface*. Each of these corresponds to the five processes and three *devices*: *alarm*, *heater*, *tempSensor* as described in Section 3. The allowed IPC is modeled as AADL *data* and *event ports*. These ports have a predefined data format and direction (*in* or *out*) which indicate how data can enter or leave from a source or to a sink. Additionally, the data flow is modeled using AADL *connections*, which allows users to specify the allowed IPC. In the definition of each process, we use the *properties* keyword in AADL to annotate each process's unique *ac_id*, for example in this scenario: *TempSensorProcess.imp* is 100, and *TempControlProcess.imp* is 101 *etc*.

The AADL model helps us to better specify the exact responsibilities of each process. However, using AADL to build systems is a mechanical task of interpreting the specification and then translating that into the domain specific implementation. To reduce the amount of work for system designers and the potential for errors, we partially automate this process by creating an AADL to C compiler. This source-to-source compiler can automatically generate the ACM for the AADL specification. Its job is to traverse AADL models, extract various processes and their unique *ac_id*, generate the

[2]Independent processes are not ancestors of each other. If one thread starts another, it has access to its child's capability space.

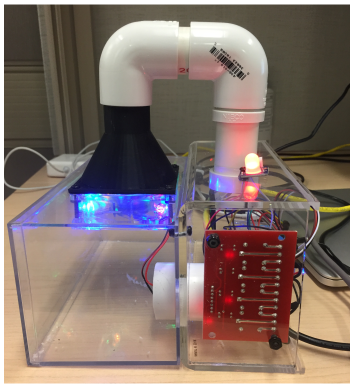

Fig. 4: Temperature Control Testbed

matrix data structure in C language based on the specified connections.

A. *Implementation on Security Enhanced MINIX 3*

In this experiment, we use the BeagleBone Black as the prototype development platform. BeagleBone Black is an open-source single-board computer hardware produced by Texas Instruments. Using a BeagleBone together with a sensor and actuator we assembled a simulated control environment. For the temperature sensor, we choose the BMP180 barometric pressure sensor, which can measure both air pressure and temperature. For the heater actuator, we simply use a fan actuator and manually heat up the environment for emulation. For the alarm actuator, we use the on-board LED light instead. Figure 4 shows the our testbed implementation.

On the MINIX 3, we implemented the sensor driver and fan driver as the temperature sensor process and heater actuator process respectively. Besides those five processes, a scenario process is included to initialize the control environment. The scenario process acts as a process loader that forks the other five processes, tells kernel each process's ac_id, and loads the correct binaries for each of them. In a real system, new process binaries and their ac_ids would be loaded by the init server based on the specification. 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.

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.

The remaining processes are relatively simple. 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 heater actuator process are implemented to passively wait for commands from temperature control process. Lastly, the web interface 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.

B. *Implementation on seL4*

For the seL4 experiment, identical to the MINIX 3 design, we used a BeagleBone platform with sensor and actuator. Additionally, the process architecture also follows the AADL architecture description. We used CAmkES to implement the AADL description and to generate all the necessary glue code. The bootstrap process initializes the temperature control process, the alarm actuator process, the temperature sensor process, the heater actuator process, and the web interface process. We also added two additional timer driver processes for demonstration purposes. All of the processes follow similar implementation patterns as the MINIX 3 and Linux implementations.

AADL and CAmkES are similar languages; both describe high-level component behavior. Translating between them is relatively simple because AADL *processes* and *systems* are like CAmkES *components* and *assemblies*. In both languages, we first define *components*, then instantiate them, and finally describe the connections between them. CAmkES, like AADL, allows for many different connection types. For example, data ports and RPC connections are allowed in both. We have begun development of an AADL to CAmkES source-to-source compiler, but in the meantime, we manually translated our AADL model into a CAmkES description.

The *seL4RPCCall* connection type generates glue code to implement remote procedure calls using *seL4_Call* and *seL4_Reply*. We chose to use this type for our connections to avoid a scenario where the malicious web interface could indefinitely block one of the temperature controller's threads. This specific IPC vulnerability is caused by the asymmetric trust between a client and server [16]. Because we use this connection type, this means that the untrusted web interface will be given a capability with the grant privilege. While the web interface will be able to modify the capability distribution slightly with the grant privilege, we argue that it cannot ever gain additional capabilities. Intuitively, if an untrusted process can only send away capabilities to trusted processes, the untrusted process could never gain more capabilities.

C. *Implementation on 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 real-time library. Similarly to MINIX 3 and seL4, we use a scenario process to facilitate the loading procedure. The scenario process in Linux spawns all other processes and creates 6 message queues that are needed for various communications.

D. Attack Simulation

In our temperature controller scenario, we suppose that an attacker would want to manipulate the critical functionality of the device. This would mean an adversary would want to gain access to the drivers, either to send arbitrary data, kill their execution, or otherwise incapacitate them. With our Linux, MINIX 3, and seL4/CAmkES temperature system implementations, we simulated two types of attacks. In the first simulation, we assume the web interface process can execute arbitrary code, and have enough knowledge about other control processes. In the second simulation, we also assume the web interface process has root privilege gained through a privilege escalation exploit or through miss-configuration.

1) Linux: In the first simulation on our Linux implementation, with the assumption that the adversary can execute arbitrary code in the web interface process, the attacker can easily spoof messages to all message queues. We successfully used the web interface process to impersonate the temperature sensor process, and we deviated the temperature control process's behavior by sending fake sensor data. Even when the environmental temperature is lower than desired temperature, we were able to get the temperature control process to still turn the fan on. Additionally, the LED controlled by alarm actuator process showed everything is normal. Similarly, we were able to send commands to the heater actuator process and the alarm actuator process to arbitrarily control the fan and LED. Since all five processes are running under the same user account, the file access control mechanism allows the web interface process to read and write all message queues. Unless each process runs under a unique user account, and the message queue is specifically configured to only allow the correct user account, the problem will still remain.

In the second simulation, we assume the adversary who controls the web interface process also has root privilege through privilege escalation exploit. In this simulation, the attacker can send spoofing message to all message queues even when all processes are running under different user accounts and the access control of message queues are well configured. Furthermore, the attacker can kill the temperature control process to incapacitate the whole control scenario, disable the alarm control for good and take over the control completely.

2) MINIX 3: Following the experiment on Linux, we tested the same two simulations on the security enhanced MINIX 3. In the first simulation the attacker cannot do much damage because IPC communication in MINIX 3 is through kernel-facilitated message passing. The web interface process in user land cannot change a process's identity stored in the kernel PCB, hence spoofing by trying to fake one's identity cannot work. Even if the temperature control process has design flaws, like failing to check the message type and sender's identity, the kernel will audit each round of communication using the ACM to stop spoofing attacks.

In the second simulation, we give the web interface process root privilege; however, the result is the same. For the same reasons mentioned above, with root privilege web interface still cannot spoof message to other process by impersonation. Unlike Linux, in microkernel based architectures user privilege is not directly tied with access control and IPC. Our experimental attack also tried to kill other processes with root privilege with the intention to incapacitate the control scenario. However, we incorporated the process management server with ACM auditing mechanism, and the policy explicitly disallowed the web interface process to use kill system call.

Other types of attack might exist for MINIX 3 systems. For example, because web interface process has the privilege to fork children processes, it can potentially launch a fork bomb to eat up system resources. This is problematic; although Linux is in the same situation. This issue could be solved by using the ACM to give each system call a quota. We will explore this in future research.

3) seL4/CAmkES: In a worst-case scenario, the web interface process is compromised remotely, allowing arbitrary code execution. To simulate this, we assume the web interface is already compromised, and we designed it to demonstrate attack possibilities. Since the seL4 kernel and CAmkES generated code have no concept of user or root, the attack surface is limited to system calls into the seL4 kernel and communication to other processes. Given the formal verification of the seL4 [8], we assume that the kernel isn't exploitable. Thus, the only attack surface is available to the compromised web interface is via IPC, and IPC policy is gated by capabilities. In our implementation, the web interface has only one capability, to communicate with the temperature controller process.

We attacked the seL4 implementation in two primary ways, by attempting to kill processes and spoof data via IPC. Both methods of attack are implemented in the same way since both require capabilities to reach out of the malicious process's virtual memory space. In addition to arbitrary code execution, we assume adversary has access to the capability distribution information. At compile time, CAmkES generates a CapDL file with this information, and we expect this file to be correct (for high-assurance systems this file can also be machine verified with the correlating source code [14], as mentioned in the implementation section above). Per the CapDL file, our malicious process, the web interface, should only have access to one capability in our implementation; it can only initiate RPCs to the temperature controller process. We also tested this with a simple brute-forcing program which attempts to enumerate all the seL4 capability slots. This brute-force program was unsuccessful in finding any additional capabilities, so it never could send arbitrary data nor kill any other processes.

The comparison of a seL4/CAmkES system to MINIX 3 and Linux is complicated by the fact that our CAmkES system is not a complete operating system with a root user. However, we argue that our comparison provides evidence about how a full seL4 OS would behave. If a file system and process manager were implemented in an seL4 OS, they would also be user-level processes. In this case, capability distribution could still manage access to the critical systems.

V. Related Work

The security of Cyber-Physical Systems is a broad research topic that is still in its early stage. One research project that shares a similar vision is DARPA's High-Assurance Cyber Military Systems (HACMS) program for using formal methods to enable more secure vehicles. The project aims at pursuing a clean-slate, formal methods-based approach to the creation of more secure vehicles by designing high-assurance, networked embedded systems from bottom up. Similar to our work, this project sees the advantage of microkernel architecture. In the HACMS project, researchers developed a UAV called SMACCMCopter that achieved high security against penetration testing. Their designs leverage existing high-assurance technologies, such as formally verified seL4 kernel, and conducted domain specific model using AADL [17].

On the other hand, the design and development of secure and reliable operating systems have been an ever-evolving journey for a long time. In response to the increasingly complex code base in monolithic kernel, microkernel architecture has been a popular topic in recent years once again. One of the most widely used commercial microkernel operating system is QNX Neutrino RTOS from BlackBerry [18]. QNX is a commercial Unix-like RTOS that is mainly used on Internet routers, Remote Terminal Units (RTUs) and in-car infotainment systems. Last year, anti-virus firm Kaspersky also launched its own microkernel-based Kaspersky OS; it made a debut on a Kraftway Layer 3 Switch with inbuilt security related mechanisms [19]. However, since both QNX and Kaspersky OS are commercial and closed source, their technical details and availability are limited.

VI. Conclusion

In this work, we present our initial experimental result on the idea of using security enhanced microkernel-based platforms for the next generation building automation system. We analyzed and extracted a real-world BAS control scenario, and demonstrated why microkernel-based platforms provide a more reliable and secure framework for control systems. We implemented the temperature control scenario in a well-controlled simulation environment using both security enhanced MINIX 3 and the seL4 kernel. We presented the prototype design of security enhanced MINIX 3, and the viability of our proposed ACM fine-grained mandatory access control. We also evaluated the formal verified seL4 microkernel and CAmkES framework. We simulated two types of attacks and demonstrated that the microkernel based approach can stop attacks that can easily be successful on a monolithic kernel (Linux) based system. Our study showed that how the microkernel-based architecture can be used to design more secure and fault-tolerant computing foundation for Cyber-Physical Systems and IoT devices.

Acknowledgment

This work is supported in part by the Department of Homeland Security (DHS) under contract number D15PC00303. The authors thank the Biosecurity Research Institute at Kansas State University for providing their building scenarios to support this work.

References

[1] N. Falliere, L. O. Murchu, and E. Chien, "W32. stuxnet dossier," *White paper, Symantec Corp., Security Response*, vol. 5, 2011.

[2] R. Spenneberg, M. Brüggemann, and H. Schwartke, "Plc-blaster: A worm living solely in the plc," *Black Hat USA*, 2016.

[3] ICSA-14-079-02, "Siemens simatic s7-1200 vulnerabilities," https://ics-cert.us-cert.gov/advisories/ICSA-14-079-02, 2014-03-20, accessed: 2015-06-14.

[4] C. Miller and C. Valasek, "Remote exploitation of an unaltered passenger vehicle," *Black Hat USA*, 2015.

[5] R. M. Lee, M. J. Assante, and T. Conway, "German steel mill cyber attack," *Industrial Control Systems*, vol. 30, 2014.

[6] X. Wang, M. Mizuno, M. Neilsen, X. Ou, S. R. Rajagopalan, W. G. Boldwin, and B. Phillips, "Secure rtos architecture for building automation," in *Proceedings of the First ACM Workshop on Cyber-Physical Systems-Security and/or PrivaCy*. ACM, 2015, pp. 79–90.

[7] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Minix 3: A highly reliable, self-repairing operating system," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 3, pp. 80–89, 2006.

[8] G. Klein, M. Norrish, T. Sewell, H. Tuch, S. Winwood, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, and et al., "sel4: formal verification of an os kernel," *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, p. 207220, Oct 2009.

[9] I. Kuz, Y. Liu, I. Gorton, and G. Heiser, "Camkes: A component model for secure microkernel-based embedded systems," *Journal of Systems and Software*, vol. 80, no. 5, p. 687699, 2007.

[10] Y. Shao, J. Ott, Y. J. Jia, Z. Qian, and Z. M. Mao, "The misuse of android unix domain sockets and security implications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 80–91.

[11] K. Elphinstone and G. Heiser, "From l3 to sel4 what have we learnt in 20 years of l4 microkernels?" *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP '13*, Nov 2013.

[12] M. Fernandez, J. Andronick, G. Klein, and I. Kuz, "Automated verification of rpc stub code," *FM 2015: Formal Methods Lecture Notes in Computer Science*, p. 273290, 2015.

[13] I. Kuz, G. Klein, C. Lewis, and A. Walker, "capdl: A language for describing capability-based systems," *Proceedings of the first ACM asia-pacific workshop on Workshop on systems*, Aug 2010.

[14] A. Boyton, J. Andronick, C. Bannister, M. Fernandez, X. Gao, D. Greenaway, G. Klein, C. Lewis, and T. Sewell, "Formally verified system initialisation," *Formal Methods and Software Engineering Lecture Notes in Computer Science*, p. 7085, 2013.

[15] P. H. Feiler and D. P. Gluch, *Model-based Engineering with AADL: an Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley, 2012.

[16] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Countering ipc threats in multiserver operating systems (a fundamental requirement for dependability)," *2008 14th IEEE Pacific Rim International Symposium on Dependable Computing*, 2008.

[17] K. Fisher, "Using formal methods to enable more secure vehicles," 2014.

[18] D. Hildebrand, "An architectural overview of qnx." 1992.

[19] N. Bene, "Finally, our own os oh yes!" https://eugene.kaspersky.com/2016/11/15/finally-our-own-os-oh-yes/, 2017-11-15, accessed: 2017-01-20.