Chapter 10

DESIGNING SECURITY-HARDENED MICROKERNELS FOR FIELD DEVICES

Jeffrey Hieb and James Graham

Abstract Distributed control systems (DCSs) play an essential role in the operation of critical infrastructures. Perimeter field devices are important DCS components that measure physical process parameters and perform control actions. Modern field devices are vulnerable to cyber attacks due to their increased adoption of commodity technologies and that fact that control networks are no longer isolated. This paper describes an approach for creating security-hardened field devices using operating system microkernels that isolate vital field device operations from untrusted network-accessible applications. The approach, which is influenced by the MILS and Nizza architectures, is implemented in a prototype field device. Whereas, previous microkernel-based implementations have been plagued by poor inter-process communication (IPC) performance, the prototype exhibits an average IPC overhead for protected device calls of 64.59 μs. The overall performance of field devices is influenced by several factors; nevertheless, the observed IPC overhead is low enough to encourage the continued development of the prototype.

Keywords: Distributed control systems, field devices, microkernels, security

1. Introduction

Field devices employed in distributed control systems (DCSs) connect sensors and actuators to control networks, providing remote measuring and control capabilities. Early DCSs were isolated proprietary systems with limited exposure to cyber threats. However, modern DCSs often engage commercial computing platforms and network technologies, which significantly increase their vulnerability to cyber attacks. While major disasters have thus far been averted, incidents such as the 2003 Slammer worm penetration of the Davis-Besse nuclear power plant network in Oak Harbor (Ohio) and the 2006 hacker

attack on a water treatment facility in Harrisburg (Pennsylvania) underscore the significance of the cyber threat.

Field devices are attractive targets for cyber attacks on control systems. Since these devices are used for measurement and control of physical systems, preventing these attacks is essential to securing DCSs and, by extension, the critical infrastructures assets they operate. Unlike early field devices, which were highly specialized systems, modern field devices use commercially-available hardware and software and can be attacked quite easily.

The need to secure field devices and their operating systems has been discussed by several researchers (see, e.g., [8, 10]). Guffy and Graham [2] have applied multiple independent layers of security (MILS) to creating security-hardened remote terminal units (RTUs). Hieb and Graham [6] have investigated techniques for creating security-hardened RTUs with reduced commercial kernels or microkernels. Hieb, Patel and Graham [7] have discussed security enhancements for DCSs involving protocol enhancements and a minimal kernel RTU (a reduced version of LyxnOS, a commercial RTOS).

This paper describes an approach for creating security-hardened field devices by applying elements of the MILS and Nizza microkernel-based security architectures. The approach protects field device operations and data by isolating them from less trustworthy application software that may be network accessible. Field device performance is an important issue because DCSs are much less tolerant to delays and jitter than traditional IT systems. To enhance performance, the approach leverages the inter-process communication (IPC) primitive provided by the microkernel. Preliminary results indicate that the observed IPC overhead is low enough to warrant further development of the security-hardened microkernel.

2. Microkernel-Based Security Architectures

Multiple independent levels of security (MILS) [1] and Nizza [5] are two microkernel-based security architectures. The MILS architecture, which was developed for high assurance and high performance computing, is based on Rushby's separation kernel [9]; it enforces strict security and separation policies on data and processes within a single processor [14]. The Nizza architecture is based on the L4 microkernel and protects security critical code. The MILS and Nizza architectures are presented in Figures 1 and 2, respectively.

MILS and Nizza employ isolated partitions, each with its own protection domain, that allow software and data of different security levels or sensitivity to be decoupled from potentially less secure software. Secure compartmentalization of components and IPC allow the trusted computing base (TCB) to remain small, comprising only the kernel and security-critical code; application software resides outside the TCB. In the MILS architecture, this enables high assurance application layer reference monitors to be inserted between application software components [1, 4]. In Nizza, security-critical code is removed from commercial applications and placed in a protected isolated compartment,

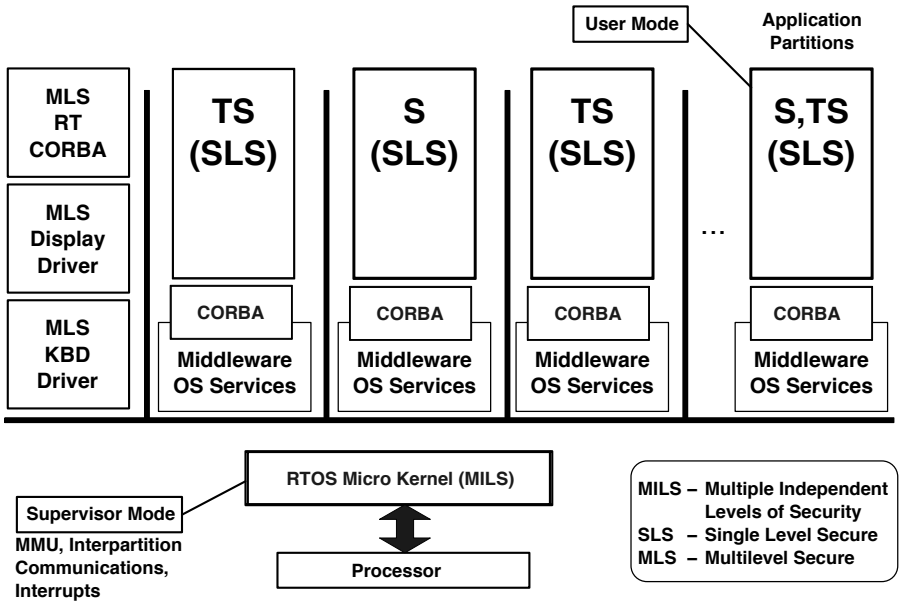

Figure 1. MILS architecture.

keeping the TCB small. Singaravelu and colleagues [13] describe an application of Nizza to the secure signing of email.

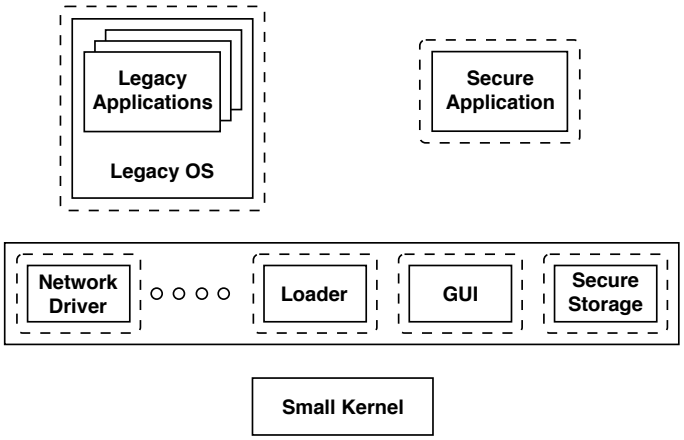

Figure 2. Nizza architecture.

MILS and Nizza primarily focus on protecting the confidentiality of data. MILS is designed for government and military systems that have multilevel security (MLS) requirements, where independent systems have historically been

used for different security levels. Nizza is aimed at desktop and commodity computing applications that require small TCBs, mainly for protecting sensitive user data. Availability and integrity – rather than confidentiality – are the principal security goals in DCSs. Our research suggests that aspects of MILS and Nizza can be used to develop security-hardened field devices that satisfy these goals.

MILS and Nizza use small kernels that provide minimal services to user processes while enforcing strong isolation of system components in separate protection domains and supporting communication between the domains. The kernel constitutes a major portion of the TCB; thus, the reliability or assurance of the overall system largely depends on the level of protection offered by the kernel. Providing strong assurance requires rigorous system testing and/or the application of formal methods. These methods do not scale well and are the motivation for using a microkernel.

One approach for creating a minimal kernel is to reduce a commercial operating system such as Linux or Windows. Unfortunately, most commercial operating systems are based on a monolithic design philosophy, which yields large kernels with poor fault isolation [14]. In a monolithic design, all the core operating system functionality – memory management, file systems, access control, network stacks, device drivers and interrupt handling – is implemented in the kernel. Thus, all the software that implements this functionality is executed in privilege mode by the processor where it is not subject to any security or protection enforcement. Although it is possible to reduce kernel size, it is difficult to cut down a monolithic kernel to a size that permits formal methods to be applied.

Another approach is to create a microkernel, a minimal kernel that implements only the services that cannot be implemented in user space [9]. Three minimal requirements exist for microkernels: address spaces, IPC and unique identifiers. Microkernel-based systems allow traditional operating system services to be moved to user space where they are run without privileges. Microkernels, by virtue of their size, tend to have significantly less code, making it possible to apply formal methods, including formal proofs. User-level services and applications exchange data using IPC, the primary abstraction provided by the microkernel. However, early microkernels were plagued by poor IPC performance, which significantly impacted overall system performance.

3. Security-Hardened Field Devices

Our approach is to design a security-hardened field device by enforcing strong isolation between: (i) critical field device resources and operations such as analog and digital input/output, (ii) software responsible for carrying out local control algorithms, and (iii) network-connected field device applications. The application code that implements network connectivity for field devices includes network drivers and protocol stacks; any exploitation of this code must not affect other components. This would enable critical field device code to continue to execute even when network components are attacked, resulting in graceful

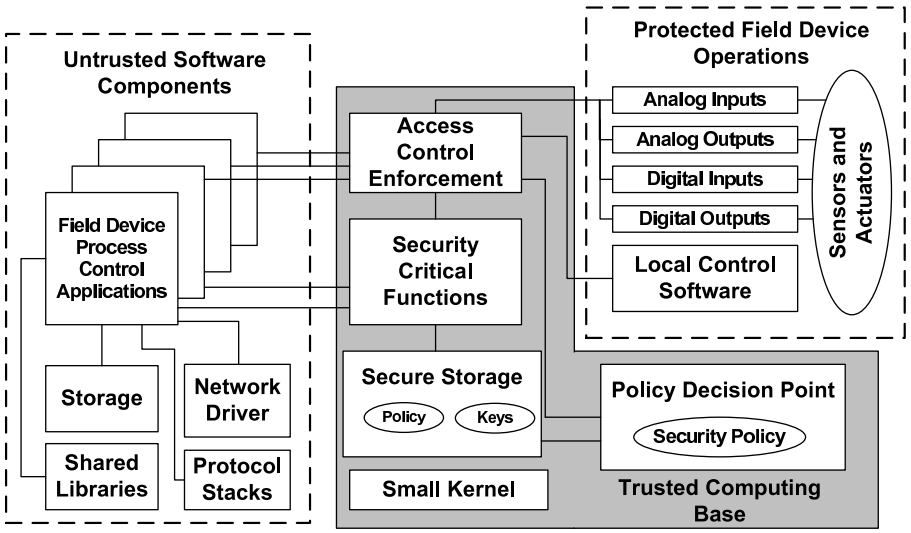

Figure 3. Security-hardened field device with a reduced kernel.

degradation as opposed to complete failure. Our approach also isolates security-critical data and code (e.g., cryptographic keys and cryptographic operations) in a separate compartment where they are protected from unauthorized access by a compromised network component. Field device integrity can be further enhanced by using a device-wide security policy enforcement component placed between the critical code and network accessible application code.

The partitioning is provided by a microkernel that supplies primitives for creating and enforcing the isolated compartments. Note that it is necessary to extract and isolate security-related code from control system software. For example, in a field device that supports security-enhanced DNP3 [7], the code that performs authentication and message integrity checking for DNP3 messages must be removed from the main body of code and placed in its own protected compartment. The isolation of security-related code has the added benefit of reducing the complexity of control applications, especially when security-related code has to be added or upgraded.

The architecture supports the enforcement of a security policy. The policy may include high-level specifications (e.g., RTU role-based access control [7]) or low-level device-specific requirements. Figure 3 presents a security-hardened field device with a reduced kernel architecture.

It is important to ensure that the security architecture does not impact field device performance, especially because industrial control systems have a very low tolerance for delay and jitter. IPC overhead (i.e., the time taken for IPC operations) can significantly affect field device performance because IPC is used extensively to implement communications between field device application code and protected operations. Device performance is also negatively impacted

by the inclusion of security-related operations (e.g., access control) that are performed before control messages are processed.

4. Prototype Development and Testing

This section describes the development of the prototype and the experimental results related to IPC performance.

4.1 OKL4 Microkernel

The OKL4 [12] implementation of the L4 microkernel was used for prototype development. OKL4 (from Open Kernel Labs) is based on the Pistachio-embedded microkernel developed by National ICT Australia (NICTA). The kernel supports the L4 version 2 API and is written in C++. It supports ARM, x86 and MIPS processors, and is targeted for embedded systems. OKL4 is released under a BSD license, although commercial licensing is also available.

The L4 microkernel provides three abstractions: address spaces, threads and IPC. Data (except from hardware registers) accessed by a L4 thread is contained in the thread's address space. An L4 address space is a partial mapping from virtual memory to physical memory. L4 threads are the basic unit of execution in L4; they share data by mapping parts of their address spaces to other address spaces. Each thread has a unique identifier (UID) and a register set that includes an instruction pointer and a stack pointer. Threads communicate with each other using IPC primitives provided by the L4 kernel; L4 IPC is synchronous and unbuffered. L4 supports the following basic IPC primitives:

- `receive()`: Wait for a message from a specific thread
- `reply_wait()`: Send a reply message to a client thread and wait for the next request
- `send()`: Send a message to a thread
- `wait()`: Wait for a message from a thread

An L4 system is composed of address spaces populated by threads that execute code in their address spaces. Each thread, identified by a thread UID, operates on data stored in its address space. The L4 kernel ensures that threads do not execute instructions in other address spaces or access data residing in other address spaces.

4.2 Hardware Platform

Gumstix [3] was selected as the hardware development platform. It provides a range of embeddable computers powered by ARM XScale processors and has been used in a number of commercial devices, indicating the platform may provide a path for possible commercialization. The Connex 400 was chosen for

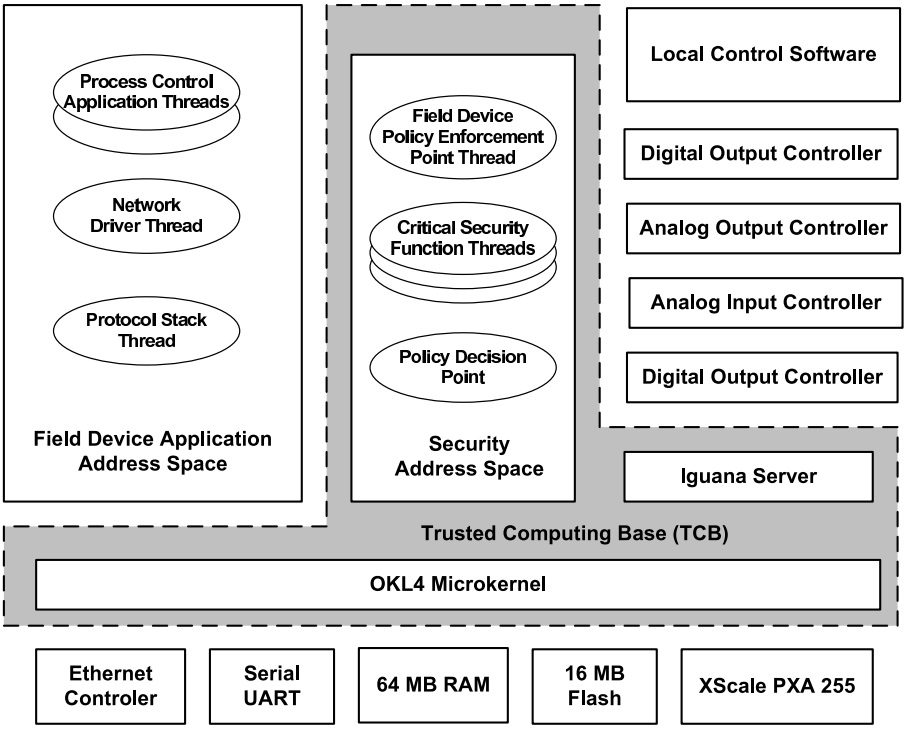

Figure 4. OKL4-based security-hardened field device.

the development. It has an XScale PXA 255 (32-bit) processor running at 400 MHz with 64 MB RAM and 16 MB of flash memory. In addition to the Gumstix motherboard, the development platform includes the Netstix and Console ST daughter boards. Netstix provides an Ethernet controller that can be used as a network interface for the field device. The console ST board provides two UART serial interfaces, one of which serves as a console interface.

4.3 System Development

The approach outlined in Section 3 was applied to the development of a prototype security-hardened field device using the OKL4 microkernel and the XScale PXA 255 processor. Figure 4 provides a high-level view of the development platform implementation. Protected field device components are implemented as "servers" as in Iguana [11], a transparent, lightweight interface to the L4 kernel included with OKL4. Each server is assigned its own address space where it is protected from other system components by the L4 kernel. The security functions and policy enforcement component are both part of a single security layer address space. This address space provides an interface (via IPC) to user applications and is located in the TCB. User applications

may execute unprivileged instructions on the processor, but are limited by the security architecture from executing privileged instructions or accessing memory outside their address spaces. All privileged actions (e.g., reads) and, in particular, updates of device points are accessed through the security layer server.

The L4 IPC provides the path along which the field device servers, security layer and field device applications exchange information and cooperate. IPC overhead is of particular concern, especially with regard to calls from field device applications to protected operations via the security layer. As discussed below, IPC overhead was evaluated by implementing one of the servers shown in Figure 4, a limited field device security layer and a simple test application program.

4.4 IPC Performance

Since our approach makes extensive use of the microkernel's IPC primitive, IPC overhead must be low enough to ensure that field device performance does not affect DCS operations. To evaluate the IPC overhead associated with protected field device operations, a security layer server and data server were implemented using Iguana's IDL, and a field device application was written. The data server is designated as the point server because it eventually provides access to analog and digital I/O for the field device. Under the security architecture, only the point server has access to the memory locations associated with connected I/O equipment. Analog I/O is not currently implemented in the prototype, so the analog input value was stored in a persistent variable.

The primary goal was to determine only the IPC overhead of a call. The security layer server has access to all the field device servers, enabling it to enforce access control for these resources. It also creates address spaces for field device applications and maps them to needed resources. Thus, the security layer maintains control of the resources available to field device applications. Field device application threads are started by the security layer, which waits for an IPC request from a field device application.

Figure 5 shows an example case where the policy enforcement point thread receives a request to read an analog input. If the request is allowed, then the operation is performed and the result passed back to the user level thread that made the call. This involves several IPC operations as shown in Figure 5. Initially, the field device policy enforcement point thread and the field device point service call `IPC_Wait` (1) and (2) (`IPC_Wait` is a blocking IPC that waits for an incoming IPC message). IPC activity is initiated by the application thread `IPC_Send` to the policy enforcement point thread (3). When the send succeeds, the field device application thread calls `IPC_Receive` to wait for a response IPC (4). Next, the policy enforcement point security layer issues an `IPC_Send` to the appropriate field device server thread (5). The policy enforcement point thread then calls `IPC_Receive` to wait for a response from the server (6). The server responds with an `IPC_Send` to the policy enforcement point thread (7). Finally, the policy enforcement point thread calls `IPC_Send`

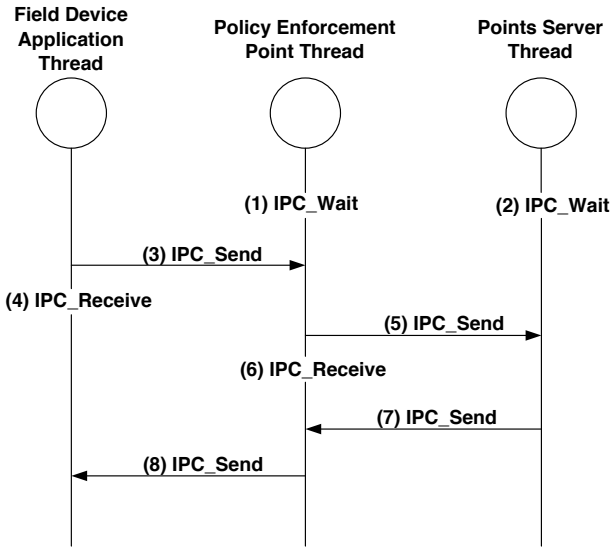

Figure 5. IPC operations involved in policy enforcement.

to return the response to the field device application thread that initiated the IPC sequence (8).

A code fragment from the test application is shown in Figure 6. The test loop iterates 300 times. Each loop instance records the start and finish times of a loop instance using the `timer_current_time()` call provided by Iguana. Iguana's time tick is one microsecond and the `timer_current_time()` returns the current tick count. Subtracting the final time from the start time gives the number of microseconds elapsed between (3) and (7). To ensure that protected operation calls are indeed reaching the point server and are being correctly returned, different values were written to and read from the point server. The observed results of the reads and writes were used to confirm that the field device application thread was retrieving values from the point server. No policy was enforced to ensure that only the IPC overhead was measured; the elapsed time between (3) and (8) represents the total IPC overhead of a protected call.

The elapsed time reported by the code fragment in Figure 6 also includes the overhead of the `timer_current_time()` call. A separate test program without the L4_IPC calls was used, leaving just the two calls to `timer_current_time()`. This program obtained a measure of the timer overhead, which was determined to be 59.63 μs. The time was rounded down to 59 μs when calculating the actual IPC overhead so that the rounding error is added to the IPC overhead.

The test program was run a total of four times on the development platform. The first value reported for each run was more than 1,000 μs, but the remaining sample times were closely grouped around 123 μs. The higher elapsed time for the first measurement of each run is very likely because the kernel performs a

```
define READ_ANALOG_INPUT_1 0x01
for (i = 0; i < 300; i++)
{
  L4_MsgClear(&msg);
  L4_Set_MsgLabel(&msg,READ_ANALOG_INPUT_1);
  L4_MsgLoad(&msg);
  stime = timer_current_time();
  tag = L4_Send(thread_l4tid(listener));
  assert(L4_IpcSucceeded(tag));
  L4_MsgClear(&msg);
  tag = L4_Receive(thread_l4tid(listener));
  ftime = timer_current_time();
  val = L4_Label(tag);
  printf("test app read_analog_input_1 call took  \%"
  PRIu64 " milliseconds, or \%" PRIu64 " microseconds\n",
  ((ftime - stime)/1000ULL), (ftime - stime) );
}
```

Figure 6. Test application code fragment.

one-time initialization for thread IPCs. Consequently, the first recorded time was dropped from the performance calculations.

Table 1. IPC overhead for protected calls.

Description	**Value**
Average reported elapsed time	123.19 μs
Standard deviation	0.784908
95% confidence interval	0.002
Timer overhead	59 μs
Actual average IPC overhead for protected operation call	64.19 μs

A total of 500 samples were selected from the remaining times. The results are shown in Table 1. The mean value is 123.19 μs with a standard deviation of 0.785 and a 95% confidence interval of 0.002. This value includes the 59 μs of timer overhead. After subtracting the timer overhead, the actual IPC overhead for the entire sequence shown in Figure 6 is 64.19 μs. Since L4 IPC calls are synchronous and there are a total of four sends in the sequence, the observed IPC overhead is distributed across four IPC send-receive pairs. Assuming that the overhead is evenly distributed, a single IPC from one L4 thread to another takes an average of 16.05 μs. These times are significantly better than the 100 μs reported for first-generation microkernels and are low enough to encourage further prototype development.

5. Conclusions

Embedded operating systems used in field devices provide little, if any, security functionality. This exposes industrial control systems and the critical infrastructure assets they operate to a variety of cyber attacks. Creating security-hardened field devices with microkernels that isolate vital monitoring and control functions from untrusted applications is an attractive solution. This strategy also produces a small TCB, which reduces vulnerabilities and facilitates the application of formal methods. Unlike most microkernel-based implementations for field devices that have been plagued by poor IPC performance, the prototype constructed using the OKL4 microkernel running on a 400 MHz XScale PXA 255 microprocessor exhibits low IPC overhead (64.59 μs) for protected device calls. While the system is not yet complete and other performance issues remain to be considered, the low IPC overhead is encouraging enough to warrant the continued development of the prototype.

References

[1] J. Alves-Foss, C. Taylor and P. Oman, A multi-layered approach to security in high assurance systems, *Proceedings of the Thirty-Seventh Annual Hawaii International Conference on System Sciences*, pp. 302–311, 2004.

[2] B. Guffy and J. Graham, Evaluation of MILS and Reduced Kernel Security Concepts for SCADA Remote Terminal Units, Technical Report TR-ISRL-06-02, Intelligent Systems Research Laboratory, Department of Computer Engineering and Computer Science, University of Louisville, Louisville, Kentucky, 2006.

[3] Gumstix, Products, Portola Valley, California (www.gumstix.com/products.html).

[4] N. Hanebutte, P. Oman, M. Loosbrock, A. Holland, W. Harrison and J. Alves-Foss, Software mediators for transparent channel control in unbounded environments, *Proceedings of the Sixth Annual IEEE SMC Information Assurance Workshop*, pp. 201–206, 2005.

[5] H. Hartig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzynski, F. Mehnert and M. Peter, The Nizza secure-system architecture, *Proceedings of the International Conference on Collaborative Computing: Networking, Applications and Worksharing*, 2005.

[6] J. Hieb and J. Graham, Security-enhanced remote terminal units for SCADA networks, *Proceedings of Nineteenth ISCA International Conference on Computer Applications in Industry and Engineering*, pp. 271–276, 2006.

[7] J. Hieb, S. Patel and J. Graham, Security enhancements for distributed control systems, in *Critical Infrastructure Protection*, E. Goetz and S. Shenoi (Eds.), Springer, Boston, Massachusetts, pp. 133–146, 2007.

[8] V. Igure, S. Laughter and R. Williams, Security issues in SCADA networks, *Computers and Security*, vol. 25(7), pp. 498–506, 2006.

[9] J. Liedtke, On micro-kernel construction, *ACM SIGOPS Operating Systems Review*, vol. 29(5), pp. 237–250, 1995.

[10] A. Miller, Trends in process control systems security, *IEEE Security and Privacy*, vol. 3(5), pp. 57–60, 2005.

[11] National ICT Australia, Project Iguana, Eveleigh, Australia (ertos.nicta .com.au/software/kenge/iguana-project/latest).

[12] Open Kernel Labs, Products, Chicago, Illinois (www.ok-labs.com).

[13] L. Singaravelu, C. Pu, H. Hartig and C. Helmuth, Reducing TCB complexity for security-sensitive applications: Three case studies, *ACM SIGOPS Systems Review*, vol. 40(4), pp. 161–174, 2006.

[14] A. Tanenbaum, J. Herder and H. Bos, Can we make operating systems reliable and secure? *IEEE Computer*, vol. 39(5), pp. 44–51, 2006.