SANDIA REPORT
SAND2010-0516
Unlimited Release
Printed February 2010

Final Report for the Security-enabled Programmable Switch for Protection of Distributed Internetworked Computers LDRD

Jamie VanRandwyk, Timothy J. Toole, Nancy A. Durgin, Perry J. Robertson, Lyndon G. Pierson, Brent D. Kucera, Philip L. Campbell

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865)576-8401
Facsimile: (865)576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.doe.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800)553-6847
Facsimile: (703)605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2010-0516
Unlimited Release
Printed February 2010

Final Report and Documentation for the Security Enabled Programmable Switch for Protection of Distributed Internetworked Computers LDRD

Jamie Van Randwyk
Timothy J. Toole
Computer and Network Security

Nancy A. Durgin
Embedded Systems Engineering

Perry J. Robertson
RF and Opto Microsystems

Lyndon G. Pierson
Philip Campbell
Brent D. Kucera
Networked Systems Survivability & Assurance

Abstract

An increasing number of corporate security policies make it desirable to push security closer to the desktop. It is not practical or feasible to place security and monitoring software on all computing devices (e.g. printers, personal digital assistants, copy machines, legacy hardware). We have begun to prototype a hardware and software architecture that will enforce security policies by pushing security functions closer to the end user, whether in the office or home, without interfering with users' desktop environments. We are developing a specialized programmable Ethernet network switch to achieve this. Embodied in this device is the ability to detect and mitigate network attacks that would otherwise disable or compromise the end user's computing nodes. We call this device a “Secure Programmable Switch” (SPS). The SPS is designed with the ability to be securely reprogrammed in real time to counter rapidly evolving threats such as fast moving worms, etc. This ability to remotely update the functionality of the SPS protection device is cryptographically protected from subversion.

With this concept, the user cannot turn off or fail to update virus scanning and personal firewall filtering in the SPS device as he/she could if implemented on the end host. The SPS concept also provides protection to simple/dumb devices such as printers, scanners, legacy hardware, etc.

This report also describes the development of a cryptographically protected processor and its internal architecture in which the SPS device is implemented. This processor executes code correctly even if an adversary holds the processor. The processor guarantees both the integrity and the confidentiality of the code: the adversary cannot determine the sequence of instructions, nor can the adversary change the instruction sequence in a goal-oriented way.

Acknowledgments

The authors would like to acknowledge Student Interns Jason Hamlet, Ben Hamlet, and Paul Cotton for their contributions to this work. These students did a fine job completing and debugging certain pieces of the software and hardware implementation, and performed an initial “black-hat” assessment of the protection system.

Table of Contents

1 Conventions and Definitions....9
2 Protection of Distributed Internetworked Computers....9
2.1 Introduction....9
2.2 Design Approach....10
2.3 Cryptographic Assurance of Execution Correctness....11
2.4 SPS Design....12
2.4.1 Secure Software/Hardware Programmability....12
2.4.2 Attack Detection/Mitigation....12
2.4.3 Design Summary....13
3 Using MON to Detect and Mitigate attacks....13
3.1 Implementing MON on SPS....13
3.1.1 MON....14
3.1.2 Real-time O/S: eCos or MicroC/OS-II?....14
3.1.3 PCAP – Packet Capture Library....15
4 Implementing a Software Vulnerability (for demonstrating resistance to subversion)....16
4.1 eCos HTTP Monitor....16
4.2 "Buffer Overrun"....17
5 Cryptographically Assured Processor....18
5.1 Cryptographic Assurance Processor Architecture....19
5.1.1 Overview....19
5.1.2 Protected Volume....20
5.2 Concept of Operations – Updating Code operating on the Secure Processor....21
5.3 Cryptographic Considerations....21
5.3.1 Overview of Cryptographic Processor....21
6 Results....23
7 Summary and Conclusion....23
Appendix A: Cryptographically Assured Processor System Operation....25
1 System Operation....25
1.1 Operation of the Target Processor....25
1.2 Operation of the Pre-Processor....26
1.3 Processor Interface Logic....27
1.4 Software....27
1.4.1 Target Code....27
1.4.2 Compiling Code for Pre-Processor and Target Processor....28
1.4.3 Cryptographic Assurance Processor Memory Map....28
1.4.4 The Code Execution Process....29
1.5 Code Shrink-Wrapping....30
1.5.1 Shrink-Wrapper Overview....30
1.5.2 Procedure to Shrink-Wrap Files....34
1.5.3 Demonstration Hardware Connections....35
1.6 Summary of Cryptographic Processor System Operation....35
Appendix B. Build Notes....36
Appendix C. LDRD Data....38

This Page Intentionally Left Blank

List of Figures

Figure 1: Secure Programmable Switch Concept 11
Figure 2. Two common CPU architectures. 20
Figure 3. Cryptographic Assurance architecture. 20
Figure 4. Cryptographic Assurance Processor block diagram..... 22
Figure 5. Example SOPC Builder window for Target Processor. 25
Figure 6. Example SOPC window of the Pre-Processor. 27
Figure 7. CAPA memory map. 29
Figure 8. Demonstration hardware configuration (after Altera documentation). 35

List of Tables

Table 1. Interrupt Routines in Pre-Processor. 26
Table 2 Wrapped code layout 30
Table 3: Cryptographic Execution Assurance Header 32
Table 4 "As Built" Shrink-Wrapped Program 34

1 Conventions and Definitions

There are several conventions used in this document. The main body of the document is printed in 12 pt. Times New Roman. Computer program listings (i.e. C code) are given in `10 pt. Courier New` and are generally indented. Filenames are printed in *12 pt. Arial italics*. The names of files are generally limited to the root part of the name, leaving off the version number. The convention is to have a meaningful root name, such as *cap_main* that would be followed by the version number before the file extension (i.e. *cap_main5.c* would be found in the text as *cap_main.c*).

There are several definitions used in this document.

- The word *byte* refers to 8-bits.
- A *half-word* is two bytes.
- A *word* is four bytes.

2 Protection of Distributed Internetworked Computers

Current methods of enforcing security policy depend on security patches, anti-virus protections, and configuration control, all updated in the end user's computer with ever increasing frequency. This research is producing a method of hardening the corporate computer infrastructure by prototyping a mixed hardware and software architecture that will enforce policies by pushing distributed security functions closer to the end user's computer, but without reconfiguring the end user's computer itself, and without relying on the correct configuration of the end user's computer. Previous research has developed highly secure network components [2][3][4][5][6][7]. Because it is impractical to replace our entire infrastructure with secure, trusted components, this paper investigates how to improve the security of a heterogeneous infrastructure (Distributed Internetworked Computers) composed of both trusted and untrusted components.

2.1 Introduction

In current practice, network security functions, including virus scanning and personal firewall filtering, are pushed onto the end hosts. These functions are susceptible to failure because of being turned off by the user or not being updated as often as required. Automating the update of security configuration on the end user platform also introduces a new vulnerability in the form of the powerful automatic update mechanism itself (if subverted).

Also, security functions are available on computing devices to varying degrees (e.g. printers typically do not have built-in virus protections). We need a way to incorporate functions such as these, with centralized control to keep protections up to date, while taking the less trusted end user platform out of the management process. We prototyped an architecture which

combines both hardware and software elements that will enforce security policies by pushing security functions closer to the end user, whether in the office or home, without interfering with users' desktop environments. A central component of this architecture is a specialized programmable Ethernet network switch which is hardened against subversion. Embodied in this device is the ability to detect and mitigate network attacks that would otherwise disable or compromise the end user's computing nodes. We call this device a "Secure Programmable Switch" (SPS, shown in Figure 1). The SPS is also designed with the ability to be securely reprogrammed in real time to counter rapidly evolving threats such as fast moving worms, email viruses, etc. The remote update function of the SPS protection device is cryptographically protected from subversion. With this concept, the user cannot turn off or fail to update virus scanning and personal firewall filtering in the SPS device as he/she could if implemented on the end host. The SPS concept also provides protection to simple/dumb devices such as printers, scanners, legacy hardware, etc. The switch is designed to be programmable by an authorized party wishing to push security functions to end users – government organizations, large corporations, ISPs, or any other network managing entity. That entity will be able to cryptographically restrict programmability to authorized sources. We are investigating many security functions including the abilities to authenticate and authorize devices to the network, monitor for cyber attacks (both external and internal including denial of service (DOS) attacks), provide firewall and virus blocking capabilities, record audit logs, and other administrator-programmable functions. We leveraged existing work on Cryptographic Assurance of Execution Correctness (CAEC) [5][6][7] in programmable logic devices to allow easily secured and authorized upgrades of the switch code. We explored some of the issues regarding the integration of existing state-of-the-art software security mechanisms into programmable hardware to provide a robust and efficient security switch. Our design also addresses the physical security concerns of such a device.

2.2 Design Approach

Current network security monitoring and intrusion detection methods are highly centralized. We believe that by distributing some parts of these functions out to the network edges we gain higher granularity of monitoring. Such a high granularity of information may not always be warranted, however a security team monitoring a network using our device would then have that option at their disposal. Data reduction techniques will facilitate this monitoring capability. Using this data reduction, one can implement a mechanism to allow real-time drill-down or zoom-out logging for network traffic. Devices similar to our proposed programmable switch exist but without the combination of capabilities proposed here. Consumer switches are now marketed with router, firewall and many other built-in features, but they lack support for monitoring and inspection of traffic in higher speed networks (1+ Gb/s) and lack a centralized, trusted mechanism for software distribution. Other vendors have introduced hardware and software that inspect network traffic at speeds up to 2.4 Gb/s and higher. To our knowledge, these devices require placement as a network gateway and therefore lack the ability to inspect traffic at a more granular scale on the edges of the network. Other researchers have pioneered development of reconfigurable logic that can search for string matches in real-time network traffic between 1 and 10 Gb/s [9][10][11] Some of these techniques are approaching the ability to search the entire "SNORT Intrusion

Detection Match list" (currently about 5,000 character strings). Each individual technique suffers from some feature trade-offs (such as ability to fit large numbers of "fixed matches" in a single programmable logic device but having the inability to handle flexible variable length string matches, for example). Extension of the prototype components described here would enable the compilation of efficient virus detection search engines and the secure downloading of these engines into the programmable security switch. This will enable increased continuous protection and the flexibility to respond swiftly to new threats.

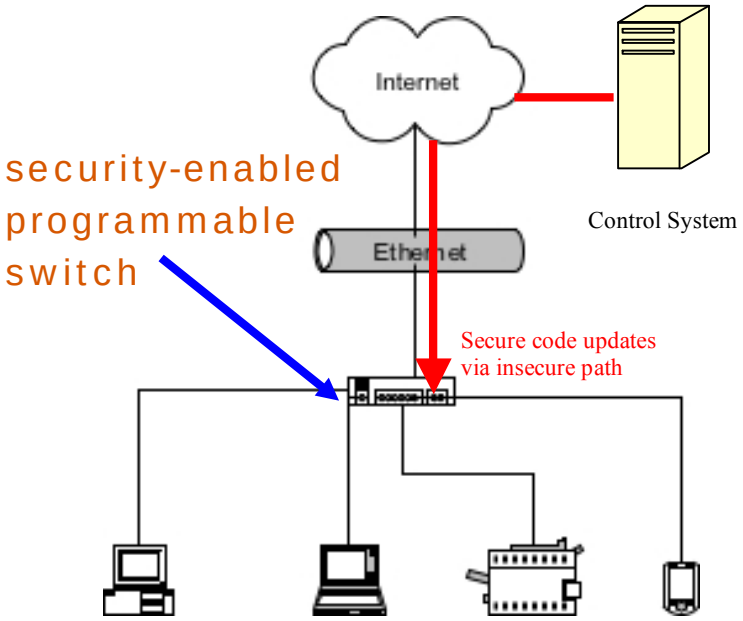

Figure 1: Secure Programmable Switch Concept

2.3 Cryptographic Assurance of Execution Correctness

Current computing architectures are “inherently insecure” because they are designed to execute any arbitrary sequence of instructions without regard to execution correctness. As a result they are subject to subversion by malicious code. We have developed a method of faithfully executing instruction sequences called “Cryptographic Assurance of Execution Correctness” (CAEC). This method protects instruction sequences from subversion during code distribution and storage [2][5]. Our method cryptographically seals the program instructions at a certification facility, then decrypts and authenticates each instruction within the executing Central Processing Unit (CPU), thereby providing a mechanism for detecting and responding to the insertion of malicious code or the unauthorized modification of previously certified code. By decrypting and authenticating the instructions within the CPU itself, the dependence on a “trusted software loader” is eliminated. This also reduces the “Trusted Volume” (i.e., the computing machinery, including communication lines, within which data is assumed to be physically protected from an adversary) by shrinking the control boundaries using cryptography. A design approach for the physical security (tamper-resistance) of the reconfigurable logic to be loaded into the prototype platform was

developed. (The implementation of chip-level tamper resistance is beyond the scope of this work.) The authors have implemented CAEC using a configurable soft-core processor [5][6][7] in a Field Programmable Gate Array (FPGA) logic device. For this project the "Cryptographic Assurance of Execution Correctness" techniques were redesigned for the chosen hardware and embedded software platform that was chosen for the prototyping of the distributed security functions. This method provides a type of software protection that begins when the software leaves the control of the developer and ends within the trusted volume of a target processor [2][5]. That is, CAEC provides program integrity, even while the program is in execution.

2.4 SPS Design

We have built a prototype FE engine in hardware using the latest generation of FPGAs. This specific implementation was developed using Altera's NIOS II configurable soft-core embedded processor. The NIOS II processor offered many advantages to this project, including the ability to compile two Altera NIOS II CPUs into a single Stratix 2S60 FPGA device. This design also allows a combination of fast, specialized hardware processing for high throughput as well as data processing by a general purpose CPU.

2.4.1 Secure Software/Hardware Programmability

Faithful Execution protects software through a cryptographic process. The application of this protection, which we call the "shrink wrap" process, protects both code confidentiality and code integrity by encoding and packaging the software at the instruction level. Individual instructions or sequential chains of instructions are encrypted to maintain confidentiality. Additionally, the encoding process includes redundant instruction information so as to allow authentication of each machine instruction and the execution order of the instructions. This work builds on the referenced previous work in Faithful Execution to allow easy yet cryptographically protected and authorized remote upgrades of the switch hardware as well as software. Protection of hardware upgrades is accomplished by encrypting the check-summed hardware download, and decrypting the hardware download within the FPGA [1].

2.4.2 Attack Detection/Mitigation

A Denial of Service (DOS) attack was selected for study [12], and detection and mitigation techniques were designed. A small network test bed in which to demonstrate successful detection and mitigation of such attacks was designed and constructed. Various codes suitable for the detection and mitigation of this initial attack scenario were examined. This project chose to build on an in-house tool called *netload* that gathers packet statistics on a given network link, providing statistics on a per TCP port basis. This gives an idea of typical use on the host side of the monitoring point and will provide a basis for identifying significant new worm/virus threats and helps detect anomalies in host network behavior. The SPS provides a platform for initiation of network countermeasures. Network

countermeasures may include traffic rate limiting as well as artificially generated deceptive responses to network stimuli, appearing as if authentic responses from real network hosts. This type of network deception has proven operationally to add another layer of protection to networks [13]. Mechanisms in the SPS permit variable granularity of data collection.

A Sandia-developed network flow monitoring system ("*MON*") implemented in the SPS provides a capability in the switch for passing flow information to a management node. This design allows the amount and type of data for examination to be alterable in real time. This feature will essentially allow an analyst to drill down or zoom out in any part of the network to view and/or record desired traffic.

The prototype device helps to address the problem of fast-moving attacks via computer networks. Fast and secure reconfiguration of such a device will aid in mitigating these fast attacks. These components will facilitate in monitoring for DoS attacks and mitigation of Internet worms. We have tested the completed work in an isolated test bed.

2.4.3 Design Summary

Our approach to fortifying the security of a heterogeneous infrastructure involves the development of a programmable network switch to place in-line with computing devices in an office or enclave setting. This device, hardened against malicious code by using cryptographically protected processor techniques initially described in a previous paper [2], provides many security functions. Some of the functions required are secure re-programmability and/or secure administration by authorized parties 1) without the end user giving up control of his/her computer and 2) without relying on the timely installation of patches by the end user. This approach should provide deployment of more scalable security protection by reducing the required number of computer security personnel to assure proper installation of patches and updates. This specialized "trusted" hardware can be distributed strategically throughout a network and made to mitigate fast moving cyber attacks, to mitigate the insider threat, and to enhance data collection and reduction during a fast attack. This approach is expected to allow finer-grained control of security functions and enable cooperative network monitoring, resulting in rapid detection and mitigation of new, undetected threats.

3 Using MON to Detect and Mitigate attacks

3.1 Implementing MON on SPS

"*MON*" is a network traffic monitoring tool written by Jim Hutchins (8965) and used by computer security at Sandia/CA for intrusion detection purposes. *MON* captures session information about TCP/IP traffic on the network and logs information about different types of sessions, such as TCP, ICMP, UDP sessions (address, port, time, duration, amount of data,

which TCP flags were set in that session, etc.). It also can log additional information about certain types of application protocols, such as HTTP (web traffic) or FTP (file transfer). The production version of *MON* logs this information to a set of files, which are examined by an automated tool or human expert, and also archived. Note that *MON* simply monitors the traffic and logs information about it – it does not take any specific actions based on anomalous traffic, but it can be combined with other tools in order to take actions (raise alarms, block intruders, etc.).

Currently, *MON* is deployed at the border of the network (at a capture point located near the corporate firewall). We would also like to distribute *MON* internally within a network, in order to monitor internal network traffic. The SPS is an example of an ideal platform for deploying *MON* (or other intrusion detection tools) at the switches in the internal network. In addition, the core functionality of *MON* is also used in the program "*NetState*", which is another intrusion detection tool that is used to monitor the state of hosts on a network. The *NetState* sniffer is another tool that would be very useful to deploy internally on a switched network, so by successfully porting *MON* to SPS, we also demonstrate the viability of porting *NetState*.

3.1.1 MON

MON is a fairly simple program. It captures Ethernet traffic from a network interface, parses the traffic, and keeps session information in memory. By choosing to log the session information to a serial channel, instead of saving it to disk, we eliminate any need for disk access by *MON*, so it is a totally memory-bound program. In a real production environment, each distributed version of *MON* would need to log its output to some central point, possibly on a separate (logically or physically segmented) intrusion detection monitoring network, but that is beyond the scope of this research. The amount of memory needed is proportional to the amount of traffic (specifically, the number of different simultaneously active sessions that *MON* has to keep track of). If *MON* is monitoring traffic seen by a local switch, then the number of sessions (and thus the amount of memory needed) is fairly small, which makes the program suited for a small embedded application such as SPS.

MON is a program that is easily portable to any Unix-like system (Linux, BSD, etc.). Besides the basic standard I/O facilities (*stdio.h*), it also needs timer interrupt support, and a method to get Ethernet packets from the operating system without disrupting normal network traffic by other applications. The method used to gain access to the Ethernet packets is the "*pcap*" library, which uses the Berkeley packet filter (BPF) library to capture packets from the operating system's TCP/IP stack.

3.1.2 Real-time O/S: eCos or MicroC/OS-II?

Since *MON* requires quite a bit of system support, the most expeditious way to port it to an embedded system like the SPS was to port it to run under one of the several embedded real-time operating systems that are available for the NIOS II processor. Two such operating systems, both freely available for the NIOS II processor, are MicroC/OS-II (which comes with the Altera Stratix II development kit) and eCos (http://ecos.sourceware.org/).

The key factors in selecting an operating system were: 1) ease of porting *MON* to the operating system, 2) stability of the TCP/IP stack, particularly the threading support. Though both MicroC/OS-II and eCos have support for the basic functions needed by *MON*, after some initial investigation we selected eCos for use in the SPS project, mainly because it provided a very unix-like library interface, and it appeared that the TCP/IP implementation was more stable than the lightweight IP stack (lwip) implemented in MicroC/OS-II. Also we had evidence that the threading support in the Nios II port of MicroC/OS-II was very poor.

For this project we used version 5.0 of the port of eCos for Nios (*nios2ecos50.exe* available from http://forum.niosforum.com/forum/). Version 5.1 is available as of the writing of this document, but we did not attempt to use it in this project.

3.1.3 PCAP – Packet Capture Library

Most of the port of *MON* to eCos is straightforward, because of eCos' unix-compatible library interface. The most difficult aspect was porting the packet capture functionality – that is, the capability to write an application that can capture raw Ethernet traffic in promiscuous mode. The canonical application used to implement this function in unix systems is the *tcpdump* program (http://www.tcpdump.org/), which uses the *PCAP* packet capture library, which in turn relies on the Berkeley Packet Filter library (*BPF*).

PCAP and *BPF* have been ported to many operating systems, but unfortunately eCos is not yet among them. The eCos code does support some basic hooks for implementing *BP*F, but the actual implementation of *BPF* has not been done (or at least is not publicly available).

The difficult part of implementing *BPF* is the functionality that grabs packets from the low-level Ethernet driver, bypasses the TCP/IP stack, and passes the raw packets to an application (in this case *MON*) for processing. Ideally this packet capture should be implemented so that it does not interfere with normal operation of the TCP/IP stack. We wanted to be able to run other applications on the SPS alongside *MON*, such as a web server or other programs that uses TCP/IP sockets.

Due to time constraints, we did not attempt to do a complete proper port of the *PCAP* and *BPF* libraries to eCos for Nios. A correct port would involve implementing *BPF* as a device driver that allows any application to open and read the "device" to receive data. That is the normal way of passing data between the operating system ("kernel") and an application. However, writing a correct device driver would likely have taken several weeks to get properly working and debugged, whereas a "hacked" port of just enough *BPF* functionality to allow the raw Ethernet packets to be passed to *MON* took only about a day.

The "hacked" port takes advantage of the fact that there is not a true "kernel mode" in the eCos operating system. Unlike with a true unix-os, there is no memory protection between "kernel mode" and "user mode", so it is possible to pass data directly from the kernel to an application, just in the same way it can be passed between user mode applications. So, a simple message queue was implemented using the eCos provided "mailbox" synchronization primitive. The queue passes Ethernet packets between the low-level Ethernet driver thread

(running in the "kernel") and *MON*. The packets are hooked at the same place they would be for a true *BPF* implementation (using the `bpf_mtap` function call). Since this simply taps into the device data stream and makes an extra copy of each packet, it doesn't interfere with normal operation of the TCP/IP stack.

The main limitation of the hacked implementation of *BPF* is that only one application can tap into the network data stream at a time. That means we cannot have more than one application running that examines raw Ethernet traffic. Eventually we may desire to have more than one such application, so it may become necessary to do a proper port of *BPF* to implement this cleanly. Also, the behavior of the *BPF* "hack" in the presence of multiple Ethernet device interfaces is currently unknown.

In addition, we need the Ethernet device to run in "promiscuous" mode, so that it captures all traffic seen on the Ethernet wire, and not just packets addressed to its own IP address. This was implemented by modifying the device initialization code to force the promiscuous mode to always be on. That was the most expeditious way to implement it, due to the lack of a conveniently exported functional interface to enable this feature properly. This could be fairly easily corrected (to make promiscuous mode selectable by the application) in a future revision of the software. Note that a complete implementation of the *PCAP* library includes a functional interface to enable/disable promiscuous mode, so that would be the cleanest way to implement it properly in the future.

4 Implementing a Software Vulnerability (for demonstrating resistance to subversion)

A goal of this project was to demonstrate the implementation of "secure intrusion detection" functionality on the SPS. To that end, we wanted to develop a demonstration of how a security flaw (such as a buffer overflow vulnerability) could be exploited in the non-SPS system, but not in an SPS-based system. This means we wanted to embed some kind of vulnerability in the software, and then be able to exploit it.

In order for *MON* to run securely, both *MON* and the operating system (eCos) need to be secure. Since there is no memory protection implemented in the Nios II processor, any security vulnerability on any software running on the system can be potentially exploited to subvert the behavior of the intrusion detection software. That means the security flaw we exploit for demonstration purposes does not have to be present in *MON* itself.

4.1 eCos HTTP Monitor

One of the optional features of eCos is an implementation of an HTTP-based web server "monitor" program. This monitor allows the current system status (threads, memory, network statistics) to be displayed remotely over the Ethernet connection by using a web browser. For ease of demonstration, we chose to implement our "software vulnerability" in the HTTP Monitor program, rather than in *MON*. This way we can invoke the vulnerability remotely, just by typing an address into a web browser. In particular, we implemented a new

HTML page (located at *http://<ip address>/monitor/hack.html*) that executed code that "exploits" the software vulnerability.

4.2 "Buffer Overrun"

Rather than implement a complete exploit, we chose instead to implement code that executes a necessary component of a "buffer overrun" attack – specifically the portion of the attack that modifies a data buffer and then executes it.

A typical buffer overrun exploits a vulnerability that allows the attacker to insert his own data into an internal buffer of the program, in order to corrupt the stack and then execute his own code (which is typically part of the data he sends). A true buffer overrun attack generally must be tailored very specifically to a given executable image, since the offsets and addresses necessary to execute the attack can change each time the program is modified and recompiled. Also there may be tricks involved in formatting the data and actually sending it to an exploitable buffer in the program.

Rather than get bogged down in the details of implementing a complete buffer overrun, we just coded some hooks into the HTTP Monitor program that would execute when the *hack.html* page was invoked by the web browser. The code of the "hack" sets up a data buffer with the correct machine opcodes that execute a function call. The address of the function to be called is determined by the code at run time, which makes the code easily relocatable/recompilable, but would not be possible in a true buffer overrun attack. The actual contents of the function to be called are arbitrary. For demo purposes we simply display a message indicating that the code was executed. In a true attack scenario, we would execute code that does something malicious that (for example) prevents *MON* from executing correctly or subverts other security features of the switch.

In a non-SPS system, when *hack.html* is invoked, a message is printed at the web browser indicating the hack was successful. In the SPS system, since the processor is attempting to execute data as code, this will be detected and the system will halt (with a diagnostic error message). Arguably if a software vulnerability exists in a program, and an exploit is run against this program, you would prefer that the program halt execution (giving you a chance to find and fix your security vulnerability), rather than that the exploit be successful.

This code was tested in a non-cryptographically assured processor system and the vulnerability introduced behaved as described above. The protected behavior described above was not fully demonstrated in a cryptographically assured processor due to problems properly integrating and shrinkwrapping certain libraries that use system timer functions. (The cryptographically assured processor used in this experiment operated without hardware acceleration of the cryptographic unwrapping function, and operated too slowly to properly execute the system timer functions.) The cryptographically assured processor was shown to detect instruction integrity violations in simpler test codes that did not involve these timer functions.

5 Cryptographically Assured Processor

Today's general-purpose processors, based on the Von Neumann Architecture, allow the execution of any arbitrary sequence of instructions. While this has led to wide spread use, it also represents a major vulnerability as malicious code can easily be substituted and executed (as in a software virus). Also, programs that attempt to protect data from disclosure are fraught with difficulty since the keys and instructions used by the program to encrypt its data can be inspected and reverse-engineered.

We developed a Cryptographic Assurance Processor Architecture (CAPA), built on a technique called Faithful Execution (FE) that can execute code correctly even if the adversary owns the processor. The processor guarantees both the integrity and the confidentiality of the code: the adversary cannot determine the sequence of instructions, nor can the adversary change the instruction sequence in a goal-oriented way.

Faithful Execution protects instruction sequences from corruption or subversion during code distribution and storage by cryptographically sealing the instructions and at execution time, decrypting and authenticating instructions within the trusted volume of the executing computer. We have implemented FE by cryptographically “shrink-wrapping” executable code in a trusted verification facility where the correctness of the code has been determined. The shrink-wrap process is performed using a special compiler. At run-time, within the protected volume of the computer system, the processor removes the protection and confirms the instruction and sequence integrity. This method protection over a large portion of the software life cycle from the time it is shrink-wrapped in the trusted facility through the distribution, loading, and storage phases, up to the point where the instructions or data are accessed by the target CPU [2]. More details on the FE concept can be found in earlier works [3], [4], [5]. A detailed description of an implementation of cryptographic assurance of execution correctness using the Nios I soft core processor is found in [6]. This implementation was extended and adapted to operate on the Nios II soft core processor for application of these techniques to the Secure Programmable Router. In the following discussion, only the basic concept and those elements that have changed since the NIOS I implementation are documented. Please refer to [6] for a more complete discussion of the design tradeoffs considered in the context of the Nios I soft core processor.

Our “Cryptographic Assurance of Execution Correctness” approach differs from previous attempts in two ways. First, the decryption of instruction sequences is performed within the CPU chip hardware itself, thereby eliminating the need for “trusted loader software” to decrypt the executable and load plaintext code into memory. Since the trusted loader software is absent, it cannot be subverted to load a malicious code in place of the intended one. Second, the instructions and data are cryptographically protected even while in memory, waiting to be fetched by the CPU for execution. This protects against the possibility of modification by malicious code after load time, and protects secret variables that may be embedded in the code against disclosure. In addition to providing privacy, our method provides “sequence integrity” and “instruction integrity. The approach can be tailored to provide transparency (integrity only) and non-transparency (privacy and integrity) and for use with exportable algorithms. Methods of "bootstrapping" the key generation and

management to achieve transparency in an environment of mutual suspicion were also investigated [7].

This team has previously implemented FE using a Java Virtual Machine (JVM) prototype. This software implementation, in which a protection engine was inserted between the JVM and the data store, was successfully demonstrated in September 2002. Details of the software prototype can be found in [4] and [5].

This project used a prototype execution platform consisting of a Programmable Logic Device (PLD) developed by Altera combined with a soft-core processor called Nios developed by Altera for use in their PLDs. The prior implementation [6], based on the Nios I processor, was augmented to operate using the Nios II processor and to shrink-wrap and un-shrink-wrap more complex programs. The Nios I processor is a 32-bit RISC processor having fixed sized 16-bit op-codes. The Nios II processor is a 32-bit RISC processor with more powerful instruction set organized as fixed sized 32-bit op-codes. The processor core and bus interconnect switch are completely configurable using Altera's System On a Programmable Chip (SOPC) Builder. This greatly sped up the prototype development by providing a flexible, re-configurable platform to try out various architectures variations.

5.1 Cryptographic Assurance Processor Architecture

5.1.1 Overview

How does the CAPA differ from the traditional Von Neumann computer architecture? Let's look at the Von Neumann architecture first. In the Von Neumann architecture, memory holds both instructions and data. A central processing unit (CPU) fetches instructions from memory and executes them, often performing operations on data. Having memory separate from the CPU makes the computer programmable. Registers in the CPU are used to hold operational information. Commonly, we find a program counter (PC), an instruction register (IR) and multiple general-purpose registers in the CPU. When the instruction memory and the data memory are contained in separate physical memory spaces, the architecture is referred to as a Harvard architecture. A comparison of the two architectures is shown in Figure 2.

When the two architectures are compared, it can be easily seen that while the Harvard architecture does not permit self-modifying code, it can permit simultaneous memory fetches. One advantage of this is the greater memory bandwidth.

In the CAPA, shown in Figure 3, a Pre-Processor serves as the memory for the Target Processor's CPU. In a way, this is similar to the Harvard Architecture in that the Pre-Processor can enforce data and instruction separation rules preventing modification of a program by itself. However, the Pre-Processor delivers instructions and accepts data through a single common interface.

There are two processors, one to run the target application called the Target Processor (TP) and one that fetches, decodes and delivers instructions and data to the TP called the Pre-Processor (PP). The Pre-Processor and the Target Processor are both constructed from Altera Nios Processors. The processors contain 32-bit CPUs with 32 bit registers. The two processors have separate buses and are interconnected via glue logic that takes care of latching and decoding instructions and data.

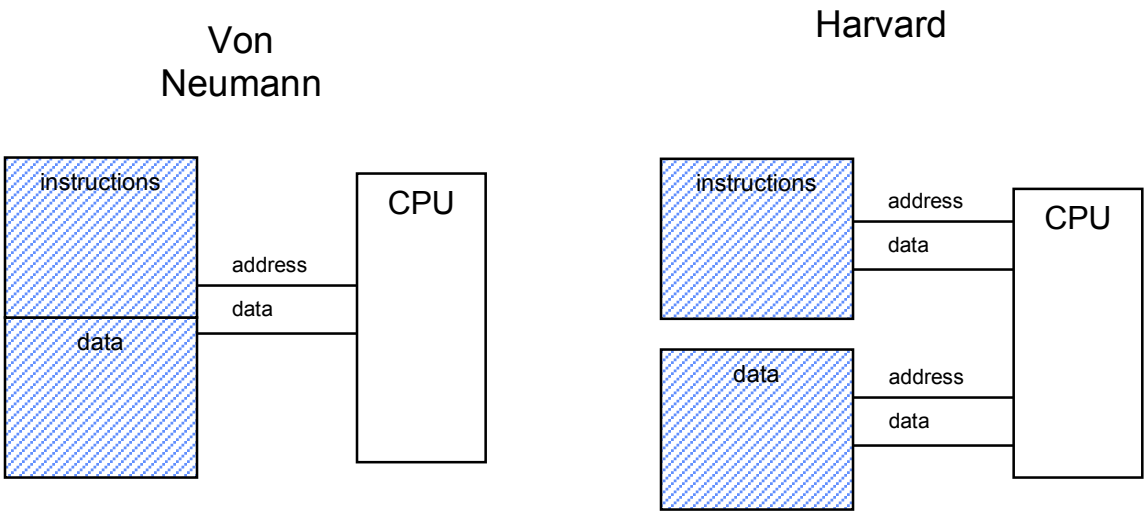

Figure 2. Two common CPU architectures.

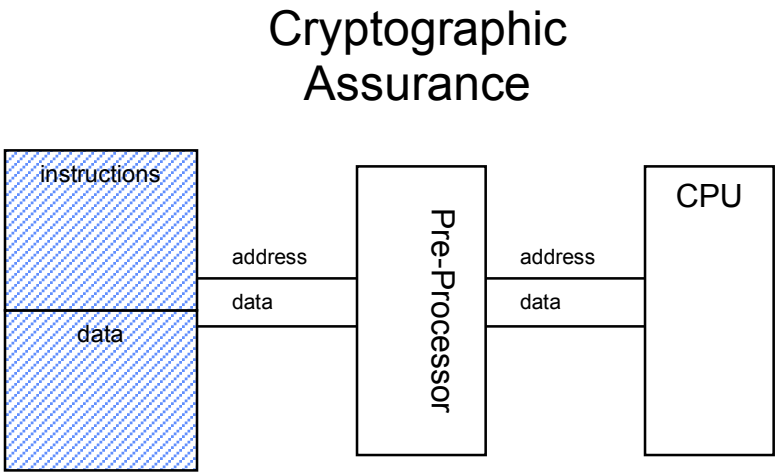

Figure 3. Cryptographic Assurance architecture.

5.1.2 Protected Volume

The required protected volume is minimized by shrinking the decryption/authentication process and implementing it within the volume of the CPU chip. This eliminates the need to

provide physical protection at the equipment or circuit module level, and allows application of chip-level anti-tamper techniques to the physical protection of this volume. The application of chip level anti-tamper techniques is beyond the scope of this paper.

5.2 Concept of Operations – Updating Code operating on the Secure Processor.

The design of the Altera Stratix II programmable logic devices allow a 128-bit AES decryption key to be programmed into individual Stratix II devices, enabling decryption of an encrypted hardware configuration file at load time.

The same key is used by the Quartus II design software to generate an encrypted configuration file stored in an external memory or configuration device. At configuration load time, the Stratix II device uses the pre-stored key to decrypt the configuration file before checking error checksums and installing the configuration file in the SRAM that controls the operation of the FPGA chip. While this capability is primarily designed to protect royalty income for intellectual property (IP) vendors (since only with the proper decryption key can one properly load and operate the IP configuration file), it can also be used to prevent unauthorized modification of the hardware configured to operate in the FPGA. Attempts to configure the Stratix II device with an unencrypted donfiguration file or a configuration file encrypted with the wronk key result in configuration failure. Therefore, tampering of the design file can be detected.

This implementation is FIPS-197 certified. The decryption key is stored securely inside the FPGA. Many security techniques have been implemented to provide secure key storage within the chip. In addition, readback of any configuration file, regardless of encrypted or unencrypted, is not permitted in Stratix II or Stratix II GX FPGAs, adding another layer of security.

5.3 Cryptographic Considerations

A prior work [5][6] examined various cryptographic modes of operation for privacy and integrity of an instruction stream as it is being fetched by a processor. Tradeoffs between speed, required memory space, and security were analyzed. Based on this earlier work, a lightweight yet non-trivial cryptographic algorithm that incorporated fetch address locations to provide sequence authentication was implemented. The basic cryptographic algorithm is a place-holder and can be replaced with other algorithms of a robustness suitable to the intended application.

5.3.1 Overview of Cryptographic Processor

The Cryptographic Processor in total consists of a Target Processor and an instruction/data Pre-Processor interconnected by glue logic. The block diagram is shown in Figure 4.

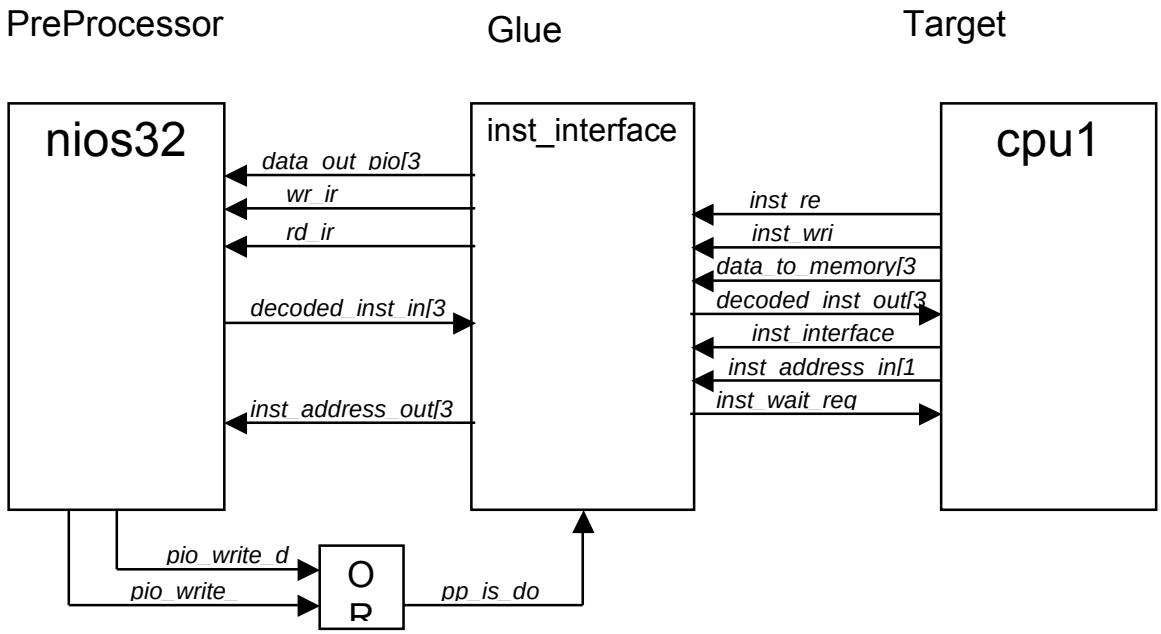

Figure 4. Cryptographic Assurance Processor block diagram.

.

The Target Processor makes a request to the Pre-Processor for an instruction or data and supplies the address. The Pre-Processor fetches the instruction (or data) and decodes it delivering the decoded data to the Target Processor. The glue logic is used to handle the timing and to condition some of the control lines for proper operation.

Appendix A contains specifics regarding the Cryptographically Assured Processor System Operation.

6 Results

The intrusion detection code called *MON* was successfully ported to the eCOS embedded real-time operating system for Nios II (including the insertion of a vulnerability for the purpose of demonstrating its detection. This vulnerability to the HTTP Monitor was implemented and the Nios II implementation (without the Cryptographic Processor) was shown to be susceptible to subversion. The Cryptographic Processor implementation of this code was not fully demonstrated due to the difficulties described below.

Major components of the Secure Programmable Switch system were prototyped. Previously developed concepts in the cryptographic assurance of execution correctness were applied to the chosen embedded hardware platform. An Altera NIOS-II "soft core" CPU was compiled into an FPGA along with the cryptographic means to decrypt and authenticate the protected instruction stream.

Methods for encrypting the compiled hardware to protect the physical security of the the reconfigurable logic to be loaded into the prototype platform were also examined. This encryption of the FPGA hardware configuration is separate from the cryptographic software protection described above.

The ability to shrinkwrap a program intended to be protected by encrypting and signing the instruction sequence and its data, heap and stack areas was developed. The ability to "unwrap" such a protected program by decrypting and authenticating the instruction sequence within the CPU was then prototyped in an FPGA. This resulted in the ability to protect not only simple programs, but also more complex programs that manipulated the stack and the heap as well as simple static data structures. After application of this technique to protect programs that extensively used interrupt processing, we found that the protected instruction fetch rate was insufficient to perform both the interrupt processing and the main program processing. This necessitated the acceleration of the protected instruction fetch process with specialized hardware, including the decryption and authentication of the instruction stream.

This re-focusing of effort to deal with the cryptographic overhead of the slow prototype prompted the project to re-design the implementation of the secure switching function, and the development of hardware acceleration of the cryptographic functions.

While this change prevented full exploration of the use of the prototyped SPS in a network testbed environment before the end of the project, the hardware acceleration will enable the prototyping of the protection of larger, more complex and wider variety of programs as they execute.

7 Summary and Conclusion

An increasing number of corporate security policies make it desirable to push security closer to the desktop. It is not practical or feasible to place full security and monitoring software on

all computing devices (e.g. printers, personal digital assistants, copy machines, legacy hardware). We have begun to prototype a hardware and software architecture that will enforce security policies by pushing security functions closer to the end user, whether in the office or home, without interfering with users' desktop environments. We designed a specialized programmable Ethernet network switch to achieve this. Embodied in this device is the ability to detect and mitigate network attacks that would otherwise disable or compromise the end user's computing nodes. We call this device a "Secure Programmable Switch" (SPS). The SPS is designed with the ability to be securely reprogrammed in real time to counter rapidly evolving threats such as fast moving worms, etc. This ability to remotely update the functionality of the SPS protection device is cryptographically protected from subversion.

With this concept, the user cannot turn off or fail to update virus scanning and personal firewall filtering in the SPS device as he/she could if implemented on the end host. The SPS concept also provides protection to simple/dumb devices such as printers, scanners, legacy hardware, etc.

This work investigated appropriate security functions to be provided by the Secure Programmable Switch. The detection and mitigation of Distributed Denial of Service (DDOS) attacks was chosen for extensive study. A small network test bed in which to demonstrate successful detection and mitigation of such attacks was designed and built. The code for the detection and mitigation of this initial attack scenario was developed, based on an intrusion detection system developed at Sandia called "*MON*". These components were exercised in the isolated network testbed.

When fully deployed, this development of a Secure Programmable Switch (SPS) distributed network protection device will allow finer-grained control of security functions and enable cooperative network monitoring resulting in rapid detection of new, undetected threats. This device will enable monitoring for insider and outsider threats, necessary components of a robust security architecture. Because this device would sit "in-line" with a user's network connection and presumably be required for network access, end-user devices without built-in security capabilities, such as printers or PDAs (personal digital assistants), would be protected from malicious traffic.

While this project demonstrated the basic feasibility of such a Secure Programmable Switch, the enhancements to the underlying protection technology now enable protection of large, complex embedded programs from subversion, thereby leveraging the security of other high assurance applications of interest to the HS and DSA/IO program areas.

Appendix A: Cryptographically Assured Processor System Operation

1 System Operation

1.1 Operation of the Target Processor

The Target Processor is configured using Altera's SOPC Builder application.

The Target Processor runs the main application code. The Pre-processor and the Target Processor have a common address space from 0x1000000-0x2000000 that is used to pass instructions and data between the two processors. The target processor has on-chip boot ROM and RAM space to run protected local instructions and for debug during development. The Target Processor begins operation by fetching the first application address which is the beginning of a header inserted ahead of the application code by the shrink-wrap process.

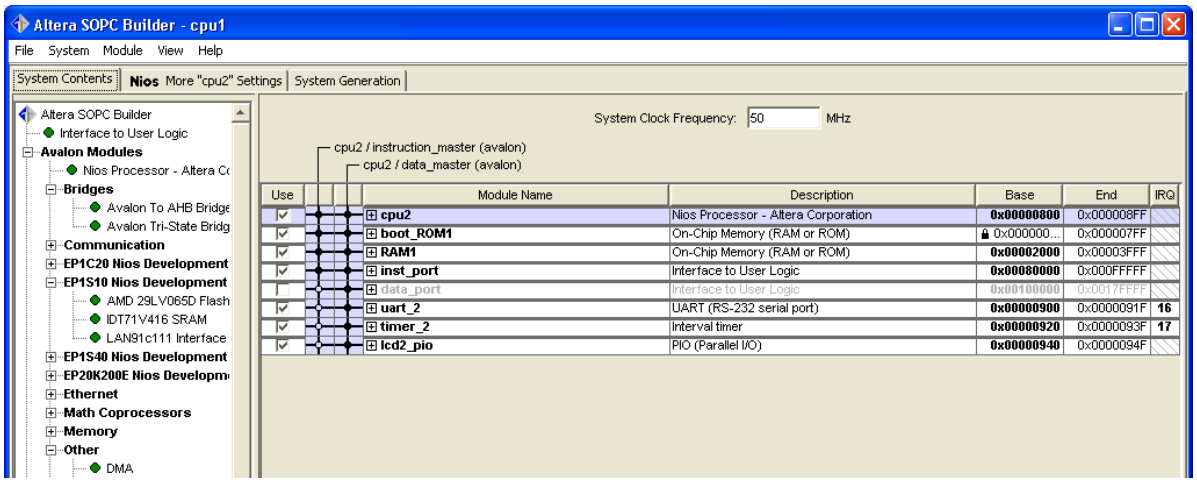

Figure 5. Example SOPC Builder window for Target Processor.

The Target Processor is configured using Altera's SOPC Builder application. The application window for the Target Processor is shown in Figure 6. The Target Processor is an instantiation of a Nios II soft core processor together with other bus components on an instantiation of Altera's Avalon Bus.

1.2 Operation of the Pre-Processor

The preprocessor holds the application program and fetches instructions for the Target Processor. There is a main program *preproc.c* that controls this process. This program initializes the interrupt processing routines and timers and then starts the kernel. The kernel in our prototype is a while loop that accumulates fetch performance information.

There are four primary interrupt routines used in the Pre-Processor as shown in Table 1.

Table 1. Interrupt Routines in Pre-Processor.

Routine	Description
tproc()	Timer
button_push()	Detect button pushes
inst_fetch()	Get & decrypt instruction from memory at address
data_read()	Read and decrypt data in memory at address
data_write()	Encrypt and Store data in memory at address

Tproc() is used to handle a timer that is set to interrupt the processor once every second. As an example, this routine flashes the green LEDs on the development board.

Button_push() is used to detect a button press. Currently, nothing is done.

Inst_fetch() is used to get requested instructions and immediate operands and sends them to the TP.

Data_read() is used to retrieve data to be fetched by the Target Processor.

Data_write() is used to store data coming from the Target Processor.

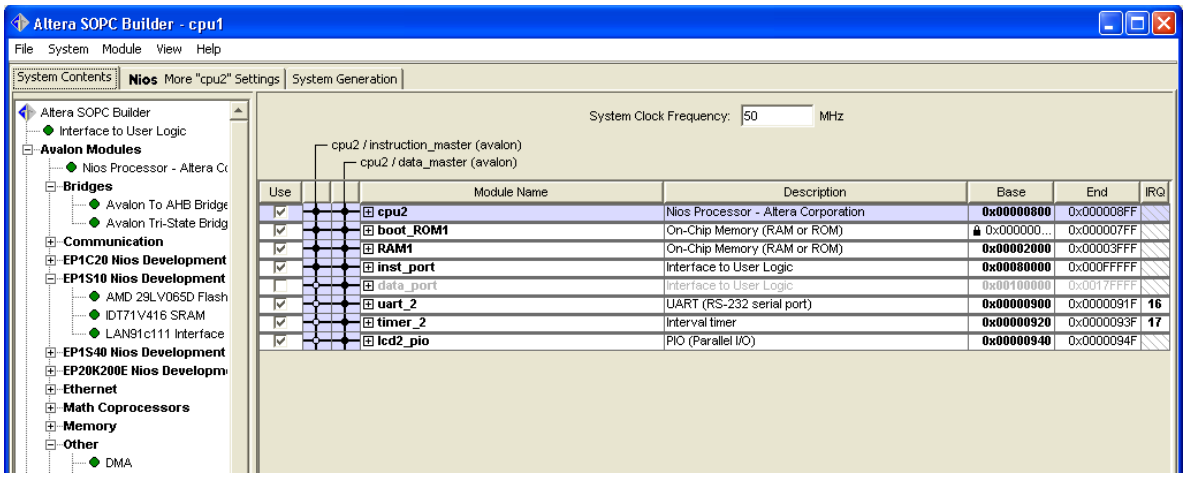

Figure 6. Example SOPC window of the Pre-Processor.

1.3 Processor Interface Logic

Between the two processors is a block of logic, written in VHDL, that makes the pre-processor appear as if were standard memory to the Target Processor. The preprocessor fetches, decodes, and delivers the requested memory contents to the Target Processor. The interface logic also takes data from the TP and presents it to the Pre-Processor.

The operation is as follows. The preprocessor CPU core performs an instruction fetch in the memory space serviced by the processor interface logic. The Avalon Bus first translates the request and writes it to the instruction port. The processor interface logic next raises a wait request line to the target processor and likewise raises an instruction request line to the preprocessor. The preprocessor, seeing the request line high, responds to the interrupt. The interrupt service routine first raises a line that tells the logic that the instruction is being fetched (this also clears the interrupt request) and it reads a 32-bit parallel address port and fetches the actual data from memory connected to the preprocessor's Avalon bus. The fetched data is decrypted and latched out a second 32-bit parallel data port. The preprocessor next lowers the line telling the processor interface logic that valid data is present on the instruction port. The target processor lowers the wait request line going to the target processor. The Avalon bus lastly finishes the memory fetch cycle.

This memory cycle is repeated for each instruction or data fetch/store.

1.4 Software

1.4.1 Target Code

There are two pieces of code executed by the Target Processor. The first is a ROM-resident program called the GERMS monitor. The second is the protected application code itself, as unwrapped and fetched through the Pre-Processor. The Target Processor's reset address begins execution of the GERMS monitor which processes simple memory read/write and go (begin execution at a particular address). The GERMS monitor instructions are retrieved directly from protected, on-chip ROM-resident locations that are not mediated by the Pre-Processor. The only function of the GERMS monitor in the Target Processor is to facilitate debugging and to initiate the fetch of the first address of the shrink-wrapped header. Later versions of the target processor may simply reset to begin fetching the first address of the header, thereby eliminating the use of the GERMS monitor.

1.4.2 Compiling Code for Pre-Processor and Target Processor

Nios I versions of the cryptographic assurance processor used a utility called "nios-build" to cross-compile a C program to be executed on a Nios I soft core processor compiled and configured into an Altera FPGA. While this utility can still be used, the preferred method for cross-compilation of Nios II programs is to use an Altera Nios Integrated Development Environment (IDE) that provides integration of coding, compiling, software load, and debug functions. The main portions of code to be compiled are the *preproc.c* which is executed by the Pre-Processor, and the *application* to be shrinkwrapped. The application to be shrinkwrapped must be compiled for the Target Processor's environment but must go through additional processing steps before loading into the Pre-Processor's memory (for later delivery to the Target Processor). In order to provide better configuration management (specifically to avoid the potential for incorporating outdated or improper code libraries), we ceased using the "nios-build" tool and compiled both the Pre-Processor code and the Target Processor code via the IDE.

1.4.3 Cryptographic Assurance Processor Memory Map

The CAPA uses the memory map shown in Figure 7. CAPA memory map.. The Pre-Processor and the Target Processor each have their own separate memory spaces. The common memory addressed at 0x10080000 is used to pass instructions and data between the two CPUs.

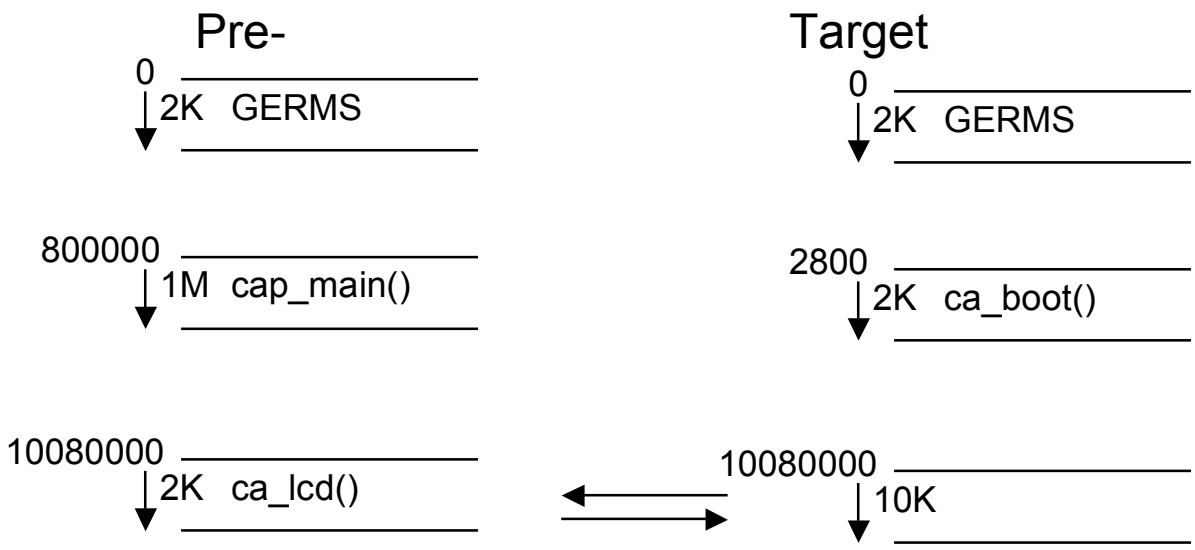

Figure 7. CAPA memory map.

1.4.4 The Code Execution Process

Compiled software can be loaded into the configured hardware by using either the *nios-2-download* utility or by using the Nios II IDE interface. Either of these methods provides for download of the compiled code into the configured hardware via a specialized USB adapter to a JTAG interface (Joint Test Action Group, an IEEE standard for boundary scan technology for communicating serial data into and out of integrated circuits). Embedded and compiled into the FPGA hardware is a JTAG slave interface that acts as a memory loader as well as a debugger and a serial communications device for the pre-processor.

First, the shrinkwrapped code for the Target Processor is loaded into the Pre-Processor's SDRAM memory space via *nios-2-download* prior to the load of the *preproc.c* code that executes in the Pre-Processor.

Subsequently, the IDE download causes this memory loader to populate the SRAM with the *preproc.c* code that executes in the Pre-Processor. During this process, a "system id" embedded in the hardware at hardware configuration time is checked against a "system id and timestamp" compiled into the load file. If no match is found, warning messages indicate a mismatch between the compiled software and the hardware target into which it is being loaded, and the load operation does not complete. (This mechanism is disabled for the prior load of the shrink-wrapped program by *nios-2-download* switch options and/or by removing the system id from the Target Processor configuration.) In a production system, this subsequent load of preproc.c may be replaced with a pre-placed ROM-based memory (generated at hardware compile time and loaded at hardware configuration time) containing the *pre-proc.c* executable.

Reset of the Pre-Processor begins its execution of *preproc.c*, and reset of the Target Processor (or the GERMS command "g1000000" entered on the Target Processor's UART, if so configured) causes the Target Processor to perform its the first fetch of the shrinkwrapped header.

The *preproc.c* code executing in the Pre-Processor services the first instruction fetch interrupt from the Target Processor, and upon finding its cache of header data un-initialized, proceeds to process the information in the shrinkwrapped header before finally delivering the first fetch to the Target Processor (which is a JMP past the rest of the header data, to the start of the shrinkwrapped code segment). Subsequent fetches cause decryption and delivery of instruction or data without further header processing.

1.5 Code Shrink-Wrapping

1.5.1 Shrink-Wrapper Overview

FE protects software through a cryptographic process. The application of this protection, or the "Shrink-Wrap" process, protects both code confidentiality and code integrity by encoding and packaging the software at the instruction level. Individual instructions or sequential chains of instructions are encrypted to maintain confidentiality. Additionally, the encoding process includes redundant instruction information so as to authenticate each machine instruction and the execution order of the instructions.

After a trusted facility develops and certifies as correct a piece of software, it encodes the software for execution on the FE hardware. The encoding process repackages the software into six segments, as shown in Table 2 Wrapped code layout. The segments include a software header, the encrypted software code, a heap area, the authentication data, the initialization vector data, and the preprocessor instructions. The Initialization Data field and the Preprocessor Instructions field are necessary only with stateful encryption and are not implemented in the current version.

Table 2 Wrapped code layout

Header
Code
Heap
Authentication Data
Initialization Data (IV)
Preprocessor Instructions (PP)

Implementation of FE permits several variables in its design. Both stateless and stateful instruction encryption is possible as well as the use of several encryption algorithms. Additionally, it may be desired that different parts of a piece of software be protected differently. In particular, the handling of a software's code segment and data segment may be different. The shrink-wrap header defines

these variables to the preprocessor as well as the memory size and the relative memory locations of the encoded software segments.

The shrink-wrap process uses a python script (targetbuild.py) to prepare the input files and parameters. The python script starts with the "*.elf" file ("Executable and Linkable Format")from the output of the gcc code cross-compiler, and uses the utilities *nios2-elf-objcopy* to convert the ".elf" file to a ".srec" file (**Motorola S-records** (**SREC**) are a form of simple ASCII encoding for binary data).

The shrink-wrap software (*wrapper.c*), starts with the "*.srec" file and produces a file called "*wrapped.srec*" that must be subsequently moved and/or renamed to the appropriate directory from which it will be loaded. The Wrapper converts the srec input records to binary and appends the Header data to the front of the binary. It then encrypts the binary code. Finally, the Wrapper appends an authenticated copy of the code to the end of the file and outputs these pieces in srec format. For the prototype, encryption and authentication transformations follow the methods described for encryption modes B0, B1, B2, B3, B4, B5,.C1or C2 as defined in [6].

What follows is a description of the Header format. The header is composed of two sections: a First Preamble and one or more Second Preamble(s). The shrink-wrap process allows for multiple sections of the binary code to be encrypted with different keys and methods. However, for the SPS prototype, we used only a single encryption segment. Hence, only one Second Preamble is ever used.

(32 bit word) index		example contents					Description	Comment
1	06	13	00	00			Passback Instruction (1)	jump over header
2	00	30	00	30			Passback Instruction (2)	(PC + 0x46 = 0x50 past header)
3	01	FF	02	FE			Preamble (1)	arbitrarily chosen pattern used to identify memory area as containing a valid header
4	03	FD	04	FC			Preamble (2)	
5	01	00	00	00			Preamble Version	
6	01	00	00	00			# encryption segments	set to 1 for initial version of wrapper
7							length of preamble	Total length of all preambles combined
8	01	02	03	04			Preamble checksum (1)	
9	05	06	07	08			Preamble checksum (2)	
10	00	00	00	00			Program ID	Unique ID for each and every encrypted program; maps to key list ID
11	50	00	00	00			Program Offset	# bytes from start of the file where executable code starts
12	00	00	01	70			Program Segment length	Length in bytes of program (executable) code
13	00	00	01	70			Program Integrity Length	length in bytes of the Program's Integrity segment
14	01	40	00	00			Data Segment Base	Base address of Data Space; needed for split bus
15							Data Segment end	preprocessor needs to know end of data space to set up stack
16	00	00	00	10			Data Segment Length	
17	00	00	00	10			Data Integrity Length	
18	00	00	00	00			IV Segment Length	Length in bytes of IV segment (currently not used)
19	00	00	00	00			PP Instruction Segment Length	Length in bytes of PreProcessor Instruction Segment (Currently not used)
20	00						Crypto Mode	select B or C series encryption; 0=B0; 6=B6;7=C1;12=C6

Table 3: Cryptographic Execution Assurance Header

One should note that in the header description there is no mention of encryption keys or encryption method. The index into this information is the "program_id" in the Second Preamble. A separate key file or data structure links the N-tuple of (program_id, encryption method, encryption keys). Key management is maintained by control of this data. Both the Wrapper and the UnWrapper make use of the key file. For the prototype, we compiled the key file into the code for both the Wrapper and the UnWrapper; however, in an applied application, this data would require further protection.

Every time a Target CPU application runs, several events occur. The application starts by requesting its first CPU instruction from memory through the Pre-processor. The Pre-processor places the Target CPU in a wait state and interrogates the application's header segment to determine cryptographic context and retrieves the correct keys from a secure memory location within the Pre-processor. After the first instruction request, the cryptographic context for that occurrence of the application is established and is not repeated for following instruction requests from the application occurrence. The Pre-processor then retrieves the instruction and performs the instruction decryption and authentication. Providing that the instruction is authentic, the Pre-processor passes the decrypted instruction on to the Target and removes the wait state. The Target then executes the instruction and increments its program counter to request the next instruction through the Pre-processor. Should the authentication check fail, the Pre-processor withholds passing the instruction on to the Target, and performs an exception handling process.

In the B series of encryption, the IV and PP sections are not used in general. However, to overcome the issue of separating the heap and the integrity data, the integrity data is placed in two segments. The preprocessor does not use the first segment. Rather it is set aside for use as heap space. The shrink-wrapped code is built as follows in Table 3: Cryptographic Execution Assurance Header to overcome this issue.

Table 4 "As Built" Shrink-Wrapped Program

Pre-amble or Header
Encrypted Program Code
Place Holder for the Heap
Encrypted Integrity Code (used for integrity check)

1.5.2 Procedure to Shrink-Wrap Files

To shrinkwrap a new code for the target processor, the application is first compiled using the Nios II IDE (after pointing the IDE to the Target Processor's hardware environment and associated system library). The application is required to be linked at an address just above the header. This is most easily done by pre-compiling the application to start at the header location, then editing the resulting linker generation script to move the application up by exactly the space required for the header (50 hex bytes). A second compilation specifying the modified linker script will result in proper location of code and variables for input to the shrink-wrap process.

For example, in the generated linker script, find and comment out the line that specifies the location and extent of the memory region containing the target processor's instruction port (inst_port memory region):

```
/* inst_port_UNUSED : ORIGIN = 0x01000020, LENGTH = 4186080 */

And replace it with a similar line with the origin increased by hex 50 and
length decreased by the same amount (decimal 80):

inst_port_UNUSED : ORIGIN = 0x01000050, LENGTH = 4186000

The resultant linker script is renamed (generated.x, for example) and
specified to guide the linkage of the subsequent build of the application
program.
```

Once the object file is located to compensate for the length of the header file, the "*.elf" file is then input to the python script called *targetbuild.py*, along with parameters communicating the start and ending address (of the header and instruction space), the start and end address of the data space, the size of the header, and the desired encryption/authentication mode.

A specific example of a sequence of commands to shrink-wrap an application are given in theAppendix B. Build Notes.

1.5.3 Demonstration Hardware Connections

The firmware design for this demonstration was compiled using Quartus II 5.0. The software was compiled using the `nios-build` command as described in Appendix A: Cryptographically Assured Processor System Operation. Two computers are used for this demonstration. One is connected via its serial port to the *Console Serial Port* and the other is connected to the *Debug Serial Port*. The Console Port is connected to the Pre-Processor. The Debug Port is connected to the Target Processor.

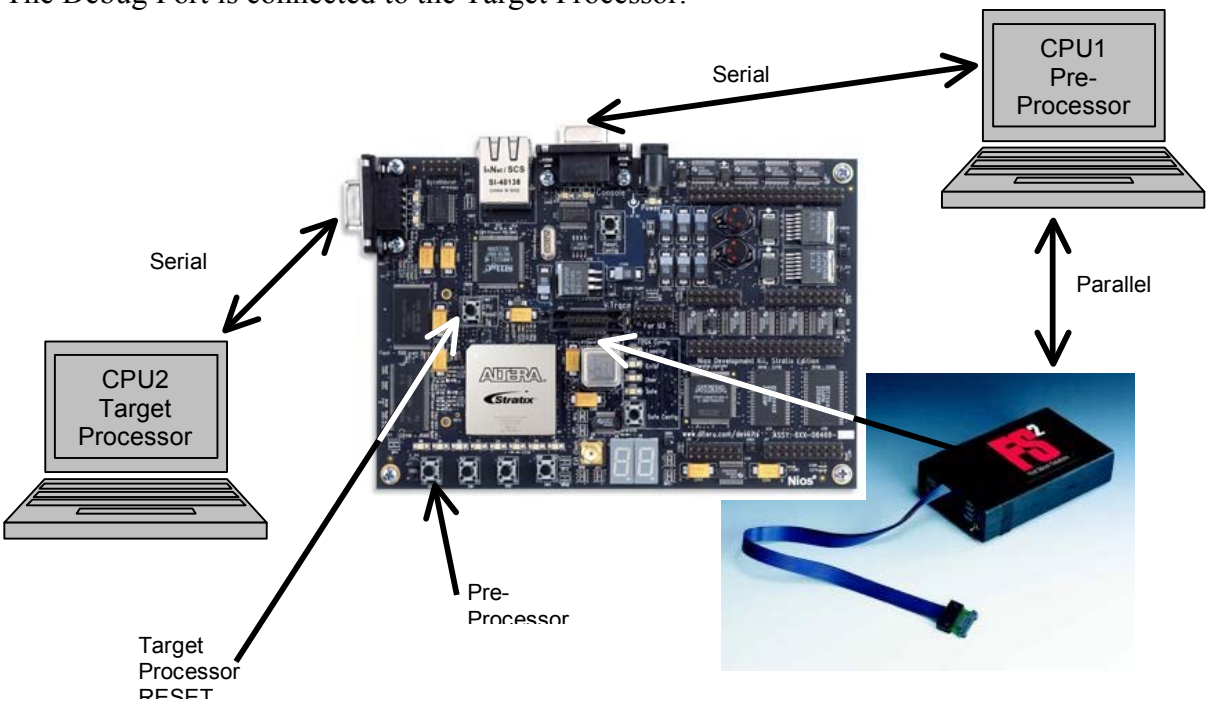

Figure 8. Demonstration hardware configuration (after Altera documentation).

1.6 Summary of Cryptographic Processor System Operation

A Cryptographic Processor has been developed using two Altera embedded Nios processors. One processor runs the decoded application code while the other fetches and decodes the instructions. Both processors were compiled and demonstrated using an Altera Stratix FPGA.

We implemented all shrink-wrap methods described in SAND2004-6478 "Stateless and Stateful Implementation of Faithful Execution" [8], however, only Methods 1 and 2 produced a sufficiently fast fetch rate to cope with system timer interrupts when processed without hardware acceleration of the cryptographic overhead.

We have shown how secure computing is possible by using cryptographic assurance of execution correctness, implemented through the concept of Faithful Execution. We have explained how Faithful Execution cryptographically seals a piece of software at a code testing and certification facility, keeps it secure by distributing it in the encrypted form, and decrypts and authenticates it as it is being executed. We then detailed the implementation of a hardware prototype using an FPGA and configurable soft-core processors, an implementation that effectively has a cryptographic wedge inserted between the CPU and memory. We also described the related shrink-wrap process to seal the certified code for use in a Faithful Execution environment.

We performed an initial "black hat" assessment of the hardware prototype to verify the security of the system and demonstrate proper operation. We also measured the instruction processing performance overhead incurred by several variations of the decryption and authentication processes. Next steps should explore applications that could benefit from Faithful Execution and that can tolerate and justify the instruction processing overhead of the cryptographic operations. Future improvements could include inserting stronger cryptographic algorithms, more flexible modes of authentication, and faster hardware implementations to improve performance.

Appendix B. Build Notes

```
Notes regarding setup of applications and windows to operate the
prototype:


Run Quartus II 5.0 Programmer

Set project:

C:\CAP\CAP_target_9_13_06\CAP_target_sdram\standard.qpf

Open programmer window  (tools.>>programmer)

Make sure program/configure is set

Check hw setup (usbblaster)

Start programming hardware  (if no errors, dual processor cryptograpically
assured hardware is loaded)


SDK window ( used to download the shrinkwrapped code)

Cd
/cygdrive/c/CAP/CAP_target_sdram_9_13_06/CAP_target_sdram/software/hello_w
orld_3/Debug/test/

Nios2-download –d 1 –I 1 wrapped.srec  (to download shrinkwrapped program)
```

Nr –t –p com1 *(to connect to the target processor serial port)*

Run NIOS II IDE

Workspace: C:\CAP\workspace

Right click on “preproc” >> run as >>nios 2 hardware

(when verified, preproc.c code is loaded into the preprocessor cpu and begins execution, ready for first fetch from target processor)

In an SDK window:

Nr –t –p com1 *(to connect to the target processor serial port)*

Press sw0 on prototype (reset target processor to start germs monitor)

G1000000 *(fetch instruction at 0x1000000 and begin execution in target processor)*

(at this point the preprocessor begins fetching instructions and data for execution by the target processor The first fetch causes the preprocessor.c to process the header at 1000000-1000050, then delivery of the instruction at 1000000 (which is a jump to 1000050))

To shrinkwrap a new code for the target processor:

Compile in the NIOS II IDE, then perform wrap operation in /Debug/ above the new directory “test” (placed below /Debug/).

In a CYGWIN window:

Cd /cygdrive/c/CAP/CAP_target_sdram_9_13_06/CAP_target_sdram/software/hello_world_3/Debug/test/

Python targetbuild.py <name_of_srec_file_to_be_shinkwrapped> 1000000 13fffff 1400000 17fffff 50 0

(the hex address parameters are mostly ignored in this version. The main parameter is the last one 0-5 encodes B0 –B5)

Appendix C. LDRD Data

The Sandia National Laboratories' Laboratory Directed Research and Development program under Project 79813 funded this effort, "Security Enabled Programmable Switch for Protection of Distributed Internetworked Computers". The project manager was John Howard; the principal investigator was Jamie VanRandwyk. Team members included Philip Campbell, Nancy Durgin, Tim Toole, Perry Robertson, Lyndon Pierson, and Brent Kucera.

Awards: N/A.

Publications:

Type: Refereed Publication, Presented and Published in Proceedings
Authors: Lyndon G. Pierson, Perry J. Robertson, Timothy J. Toole, Jamie Van Randwyk
Title: Protection of Distributed Internetworked Computers
Publication Name: 39th IEEE International Carnahan Conference on Security Technology, October, 2005, Las Palmas De Gran Canaria, Spain.

Type: Other Publication, Miscellaneous publications
Authors: Jamie VanRandwyk, Timothy J. Toole, Nancy A. Durgin, Lyndon G. Pierson, Perry J. Robertson, Philip L. Campbell, Brent Kucera
Title: Final Report: Secure Programmable Switch for Protection of Distributed Internetworked Computers LDRD
Location Published: Albuquerque NM, USA
Detail: SAND Report in process of publication, Report Number SAND 2010-0516

Patents (applied or issued): In preparation based upon prior Technical Advances SD-6192, SD-7051, SD-7052, and new material in SD-10424 developed and tested under this LDRD.

Technical Advances: SD-10424. This Technical Advance describes a method of assuring against introduction of malicious code in computer systems by cryptographically decrypting and authenticating the sequence of instructions inside the CPU chip itself.

Copyrights (for Software): None.

Employee Recruitment: N/A

Student Involvement: This project engaged summer student intern Jason Hamlet, Ben Hamlet, and Paul Cotton.

Non-LDRD Funding: None.

9 10 11 12 13

References

[1] ”Design Security in Stratix II and Stratix IIGX Devices”, http://www.altera.com/products/devices/stratix2/features/security/st2-security.html

[2] Lyndon G. Pierson, Philip L. Campbell, John M. Eldridge, Perry J. Robertson, Thomas D. Tarman, and Edward L. Witzke, Secure Computing Using Cryptographic Assurance of Execution Correctness, in *Proceedings, 2004 International Carnahan Conference on Security Technology*, held in Albuquerque, NM, October 11-14, 2004, IEEE, 2004.

[3] Thomas D. Tarman, Edward L. Witzke, Lyndon G. Pierson, and Philip L. Campbell, On the Use of Trusted Objects to Enforce Isolation Between Processes and Data, in *Proceedings, 2002 International Carnahan Conference on Security Technology*, held in Atlantic City, NJ, October 20-24, 2002, IEEE, 2002.

[4] Philip L. Campbell, Lyndon G. Pierson, and Thomas D. Tarman, *Prototyping Faithful Execution in a Java Virtual Machine*, SAND2003-2327, Sandia National Laboratories, Albuquerque, NM, September 2003.

[5] Philip L. Campbell, Lyndon G. Pierson, and Thomas D. Tarman, *Principles of Faithful Execution in the Implementation of Trusted Objects*, SAND2003-2328, Sandia National Laboratories, Albuquerque, NM, September 2003.

[6] Perry J. Robertson, Lyndon G. Pierson, Philip L. Campbell, John M. Eldridge, Edward L. Witzke, Final Report and Documentation for the Cryptographic Assurance Processor LDRD, SAND2005-0297, Sandia National Laboratories, Albuquerque, NM, January 2005. (limited distribution)

[7] Philip L. Campbell, Lyndon G. Pierson and Edward L. Witzke, “Trusted Objects,” *Conference Proceedings of the 2001 IEEE International Performance, Computing, and Communications Conference*, Phoenix, AZ, April 4-6, 2001.

[8] Phlip L. Campbell, Lyndon G. Pierson, John M. Eldridge, Edward L. Witzke, Perry J. Robertson, Stateless and Stateful Implementation of Faithful Execution, SAND2004-6478, Sandia National Laboratories, Albuquerque, NM, January 2005. (limited distribution)

[9] IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM '04, April 20-23, 2004, Napa CA: “Deep Packet Filter with Dedicated Logic and Read Only Memories” Y. Cho, W. Mangione-Smith, UCLA.

[10] IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM '04, April 20-23, 2004, Napa CA: “A Methodology for Synthesis of Efficient Intrusion Detection Systems on FPGAs” Z. Baker and V. Prasanna, USC.

[11] IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM '04, April 20-23, 2004, Napa CA: “Scalable Multi-Pattern Matching on High Speed Networks” C. Clark and D. Schimmel, Georgia Institute of Technology.

[12] W. J. Blackert, D. M. Gregg, A. K. Castner, E. M. Kyle, R. L. Hom, R. M. Jokerst. "Analyzing Interaction Between Distributed Denial of Service Attacks And Mitigation Technologies," *discex*, vol. 01, no. 1, p. 26, DARPA 2003.

[13] Van Randwyk, Thomas, Carathimas, McClelland- Bane. “Adaptive Network Countermeasures” SAND2003-8624. Oct 2003.

Distribution

1	MS	0484	B. W. Marshall, 8000
1	MS	0671	G. E. Rivord, 5610
1	MS	0671	M. J. Skroch, 5612
1	MS	0671	C. M. Villamarin, 5633
1	MS	0672	P. L. Campbell, 5616
1	MS	0672	J. D. Dillinger, 5616
1	MS	0672	R. L. Hutchinson, 5616
1	MS	0672	T. S. McDonald, 5614
5	MS	0672	L. G. Pierson, 5616
1	MS	0672	W. D. Neumann, 5614

1	MS	0672	R. C. Schroeppel, 5614
1	MS	0806	J. M. Eldridge, 4336
1	MS	0806	E. L. Witzke, 4336
1	MS	0806	L. Stans, 4336
1	MS	0874	D. W. Palmer, 1711
1	MS	0874	P. J. Robertson, 1711
1	MS	1206	N. A. Durgin, 5622
1	MS	1206	T. D. Tarman, 5622
1	MS	1206	J. V. Vonderheide, 5622
1	MS	9004	J. M. Hruby, 8100
1	MS	9011	E. B. Talbot, 8965
1	MS	9011	T. J. Toole, 8965
5	MS	9011	J. A. Van Randwyk, 8965
1	MS	9052	R. E. Stoltz, 9052
1	MS	9151	H. H. Hirano, 8960
1	MS	9151	L. M. Napolitano, 8900
1	MS	9159	H. R. Ammerlahn, 8962
2	MS	0899	Technical Library, 4536
2	MS	9018	Central Technical Files, 8944
1	MS	0161	Legal Intellectual Property, 11500
1	MS	0188	D. Chavez, LDRD Office, 1011