AFRL-SR-AR-TR-09-0080

REPORT DOCUMENTATION PAGE

OMB No. 0704-0188

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ORGANIZATION.

1. REPORT DATE (DD-MM-YYYY)	2. REPORT TYPE	3. DATES COVERED (From - To)
02-26-2009	Final Performance Report	March 1, 2006 - Nov.30,2008

4. TITLE AND SUBTITLE
Evolvable Approaches to Software Verification and Validation

5a. CONTRACT NUMBER

5b. GRANT NUMBER
FA9550-06-10152

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Narahari, Bhagirath
Simha, Rahul
Choudhary, Alok

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
The George Washington University
2121 I St, NW, 6th Floor
Washington, DC 20052

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Office of Scientific Research/NL
875 N. Randolph St Room 3112
Arlington VA 22203
Dr Robert Herklotz

10. SPONSOR/MONITOR'S ACRONYM(S)
AFOSR

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

Distribution A: Approved for Public Release

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This project considered software protection in embedded systems built using encrypted execution platforms where instructions and data are stored in encrypted form in memory. The objective of this project was to develop architectural solutions to address physical attacks on such encrypted platforms when a sophisticated attacker has captured the device. The attacks are based on exploiting structure in the application code and data, which can be uncovered by direct manipulation of hardware. An integrated hardware-software approach was taken to design a secure system to protect against such attacks. The architecture utilizes a secure on-chip hardware component, in the form of a Field-Programmable Gate Array, as the main protection mechanism. The reconfigurable logic in hardware, when combined with the ability of the compiler to instrument the code, was used in powerful ways to strengthen the security of computing platforms. Several techniques, in architecture, compiler and security, were proposed and designed. Simulations and prototyping experiments showed that this approach is feasible, easy to implement and on average adds low performance overheads. The research involved faculty and graduate students, and partly supported two doctoral theses.

15. SUBJECT TERMS
security, hardware architecture, embedded systems, software security, tamper-resistance, encrypted execution,compiler

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Bhagirath Narahari
			SAR	19	19b. TELEPHONE NUMBER (Include area code) (202) 994 3326

Final Performance Report

Evolvable Approaches to Software Verification and Validation

AFOSR Contract Number FA9550-06-1-0152

Principal Investigator:
Bhagirath Narahari
Department of Computer Science
The George Washington University
2121 I St. NW, Suite 601
Washington, DC 20052
narahari@gwu.edu
(202)-994-3326

Cover Sheet: Form SF 298 attached.

1. Executive Summary.

The objective of this project was to develop integrated hardware-software approaches to address attacks on encrypted execution and data. These are attacks on highly-encrypted systems from a resourceful adversary who has physical access to the system and may not need to decrypt encrypted software. The attacks are based on exploiting structure - in encrypted instruction streams and data–that can be uncovered by direct manipulation of hardware in a well-equipped laboratory.

To achieve the goal, research was conducted to integrate and advance current techniques in compilers, hardware architectures, and security to develop novel techniques to protect against physical attacks on encrypted embedded systems. Specific objectives included

- The investigation of a threat model under physical capture of system
- The design of hardware architectures, to provide secure systems, using commercial off-the-shelf reconfigurable logic technology without requiring new processor designs
- The development of integrated compiler-hardware techniques to protect application code and data against physical attacks
- The study of security and performance tradeoffs and the development of adaptive software and hardware techniques
- The provision of cycle accurate simulation infrastructure to evaluate secure hardware techniques.

The innovation in the approach was in exploiting the power of integrated software-hardware methods. The hardware side of the innovation comes from using reconfigurable logic to implement security techniques in hardware. The reconfigurable logic in hardware, when combined with the ability of the compiler to instrument the code, can be used in powerful ways to strengthen the security of computing platforms.

Several techniques were proposed and designed to address these objectives. The research involved faculty and graduate students at the doctoral level, and partly supported two doctoral dissertations.

2. Status of Effort.

The research proposed involved a number of research areas: (1) protection against attacks on application code and application data; (2) use of reconfigurable logic to design secure hardware; (3) integration of hardware-compiler-security techniques; (4) study of security and performance tradeoffs; and (5) development of cycle accurate compiler and simulator infrastructure. The study of the threat model, representing an understanding of the types of attacks, provided us with an insight into the kinds of security techniques needed for our system.

An integrated hardware-software approach was taken to develop a system to combat the types of attacks considered, i.e., attacks on encrypted execution and data platforms (EED attacks). A system architecture (called CODESSEAL) was designed and a number of techniques were designed and tested for protection against attacks on application code and data. The hardware side of the innovation comes from using a secure reconfigurable logic hardware component in the form of Field Programmable Gate Arrays (FPGAs), a technology that has proven enormously successful in *performance* enhancement but that had not been exploited for *security*. We showed that the dynamic reconfigurability of FPGAs, when combined with the ability of a compiler to instrument encrypted code, can be used in powerful ways to strengthen security of computing platforms. The software side of this project's innovation claims can be found in the compiler algorithms developed to generate encrypted code, along with special instructions to the FPGA targeted towards EED attacks.

A number of different techniques were designed and evaluated to protect against code and data attacks. The algorithms required for each protection technique were designed and implemented, the compiler was modified, and a cycle accurate simulation infrastructure was implemented to measure the performance overhead of the proposed system. The low performance overheads validated our approach.

One of the project objectives was to study security and performance tradeoffs and to develop evolvable adaptive protection schemes. We developed a model to apply different security schemes to different portions of the code based on a risk analysis of the application code. Based on experimental and theoretical analysis, the various security techniques provide different levels of protection against various structural (i.e., code)

attacks while incurring different amounts of performance overhead and architectural complexity. The key architectural concept of our approach–the use of reconfigurable logic, in the form of FPGA technology, to provide hardware solutions for software security–was also applied to solve additional security problems such as buffer overflow and network intrusion detection.

Our analysis of EED attacks provided a deeper understanding of the vulnerabilities of EED systems and therefore the need to protect against such attacks. In contrast to past approaches, we have taken an integrated software-hardware co-design approach thereby enabling us to develop protection mechanisms at the code producer level (the compiler) and the code consumer level (the processor). Our solutions presented novel techniques to the problem of physical attacks on encrypted systems *without re-designing the processor*. By utilizing FPGA technology for security, our research opened up new directions for providing adaptive security protocols based on application needs thereby introducing the problem of tradeoff of security and performance.

3. Summary of Achievements.

We made progress on several fronts in the project. Below we itemize these achievements by topic and summarize the main results obtained. More detailed results are available in the papers listed in the publications section, and available at http://www.seas.gwu.edu/~narahari/afosr/ along with the software developed as part of the research.

1. Threat Models

An investigation was conducted of the types of attacks that are possible on an encrypted execution and data (EED) platform when the adversary has captured the device (or has physical access to the device) and has the ability to snoop on the system bus and inject their code or data into the processor. Two broad types of attacks are possible: attacks on structural integrity and attacks on the data. For each type of attack, how an attacker could disrupt the execution was studied. These EED attacks can be briefly summarized as follows:

- *Code injection/Execution disruption* attacks where an attacker tries to modify or replace a portion of the application code.
- *Instruction replay* attacks, where the attacker tries to reissue a block of encrypted instructions from the application code.
- *Control flow attacks*, where the attacker tries to disrupt the correct control flow of the program by injecting application code blocks that are different from what were requested by the program.
- *Data injection/modification* attacks, wherein the attacker attempts to inject their data to alter or observe the application code behavior.
- *Data substitution* attacks, wherein the attacker tries to substitute a requested data block with another data block produced by the application including stale data.

The domain of embedded systems and applications includes avionics, communications equipment, unmanned vehicles or devices, sensors and electronic control systems. In a typical attack of the kind addressed in this research, a device is captured and probed in a sophisticated laboratory. Such an attack can result in loss of data (such as control settings, coordinates, and cryptographic keys), loss of intellectual property and the ceding of informational advantage. In addition, such an attack can be taken further to disrupt legitimate operation of the device. For example, a communications or control device could be replaced with a tampered version that could present an active and disruptive threat during operation. In all these cases, it is the combination of vulnerabilities in hardware–usually embedded processors–and software that forms the basis of these attacks.

2. Architecture and Hardware-Software Framework.

Much of the prior work in the field of encrypted execution was either susceptible to the physical attacks addressed in this research, i.e., EED attacks, or provided solutions that required re-design of the processor core. Requiring a re-design of the processor incurs a high cost and requires a buy-in from the chip manufacturers. The goal of this project was to provide architectural solutions which did not require processor redesign. This project combines compiler and architecture techniques to provide a solution that can be built using commercial off the shelf technology (COTS). A system architecture – the Compiler Development Suite for Secure Applications (CODESSEAL) – was designed and its performance was evaluated using a cycle accurate processor simulator (see publications [2,4,7,10]).

The approach, i.e., the CODESSEAL system, augments the back-end of the compiler to instrument each *code block* of the executable code (and data) with security-related labels that are then examined by a secure hardware component that sits between memory and the processor. The hardware platform we targeted is a standard processor coupled with an on-chip reconfigurable fabric – this enables us to leverage commercially available Field Programmable Gate Array (FPGA) platforms such as those from the Xilinx Virtex II Pro family. Our main technique works as follows. First, the back-end of the compiler module instruments the executable code by inserting integrity checking labels into each code block. Second, the secure hardware component implemented in the FPGA logic, which we call the Guard, intercepts cache block read and write requests from the memory controller. An overview of our architecture is shown in Fig. 1. The Guard processes each encrypted code block, using the inserted labels to conduct authorization and integrity checking to detect and prevent memory spoofing attacks, and passes on the decrypted code block to the processor's cache. We assume that both compilation and FPGA configuration occur in a safe location and that the FPGA cannot be manipulated by the attacker once configured. Figure 1 shows the overview of our system – Figure 1(a) shows the infrastructure and Figure 1(b) shows the hardware architecture. The overall algorithm for process with the roles played by the compiler and the architecture are illustrated in Figure 2.

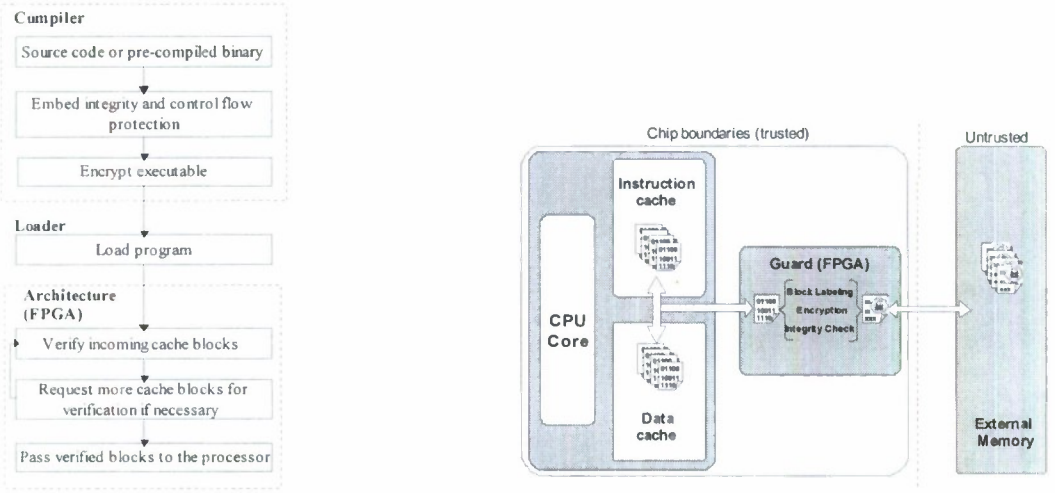

Figure 1: (a) Framework and (b) Conceptual Architecture

Compile-time processing

*C1.*Break code *in* cache blocks (cb)
*C2.*Label each block cb_i
*C3.*Compute each cache block signature **sign_compile(cb_i+cb_{label})**
*C4.*Embed signature in object file
*C5.*Encrypt (AES)

Run-time Validation(loader)

*L1.*Set the program context (**key**)
L2. Set program start address (a_{start})

Run-time Validation (FPGA guard)

*H1.*Monitor memory read/writes from the cache controller
*H2.*Capture requested address **a_i=address(cb_i)**
*H3.*Fetch and decrypt cache block cb_i
*H4.*Fetch stored signature of cache block cb_i, **x_i=sign_compile(cb_i)**
*H5.*Compute the signature of the fetched block cb_i, **y_i=sign_runtime(cb_i+ a_i + a_{start})**
*H6.*If **(y_i == x_i)** then **valid (send to cache)** else **HALT**

Figure 2: The Process: Compiler and Architecture(FPGA) Roles

The contribution is the approach itself: by relying on the secure hardware component, i.e, the Guard, our approach can accelerate the execution of encrypted programs in a secure environment with low overheads and without requiring new processor designs. One particularly attractive feature of our approach is that a single piece of information (the signature encapsulated in the label) is used to detect all three types of memory spoofing attacks. A second advantage is that the labels are easily inserted post-compilation and, therefore, our approach can be applied to legacy binaries. A third advantage arises from the use of FPGAs: we showed (in [10]) how a basic EED platform can be implemented using FPGA hardware, leaving the standard processor

components unmodified, and how the FPGA can be used to optimize the computations involved in decryption and integrity-checking. Furthermore, because the FPGA is reprogrammable, encryption algorithms can be changed post-deployment and because FPGAs are widely used, chip manufacturers are increasing resistance to physical attacks. One caveat of our approach is that the Guard module requires knowledge of the cache block size and the address where the program is loaded, because address offsets are part of the labels. Typically in embedded systems this information is known prior to deployment. However, our approach may require a special secure loader for complex servers or for desktop computers.

Implementing the CODESSEAL concept required the design of a suite of techniques to implement the system. The suite of techniques provided were based on three parameters: (i) the technique used to generate the signature and embed the label into the code block; (ii) the granularity at which the code block is defined; and (iii) location where the signature is stored in the case of data blocks. For each parameter, different techniques were developed with varying security strengths and performance overhead. These are summarized briefly in what follows next. As noted earlier, a particularly attractive feature of the approach is that a single piece of information in the label can detect all types of attacks. This information is the integrity signature. This signature essentially embeds the program control flow into the binaries and thus can prevent, and detect, code injection and changes to the program control flow that are forced by the attacker. Similarly, the signature embedded into the data blocks can prevent data injection and data substitution attacks. The method used to generate the signatures, which are embedded as labels in the code block, gives rise to different security techniques. We designed two different methods to compute the code block signatures – (a) a SHA-1 signature and (b) a 32-bit CRC. The SHA-1 technique results in higher security strength but incurs higher performance overhead. Next, the granularity of the "*code block*" in the application code can be defined in different ways with different impact on performance and portability. Data blocks are always defined to be cache blocks and thus there is no decision to be made about granularity of data blocks. We considered two granularities for code block: (a) the cache block and (b) a basic block of code (segment of code between two branch points). These techniques are presented in detail in publications [2,10] and [4] respectively. By operating at the basic block granularity we can deal with program control flow properties that are intrinsic to the application and independent of the architecture specific parameters such as cache block size, and size of cache. This also requires that our compiler optimizations work at a level removed from the architecture details. In contrast, the cache block granularity implies architecture dependence, and working at a lower level in the compiler, but simpler compiler modifications and better performance. The basic block provides more portability but incurs larger performance overhead. For the third parameter, the signatures can be stored inside the code block or in a separate section of memory. Storing the signature separately results in lower performance but requires more complexity in the cache controller hardware (inside the FPGA). A summary of the performance is provided in Tables 1 and 2 in Appendix A.

The combined compiler/FPGA technique can secure embedded systems that are susceptible to a class of local attacks even under the protection of a fully cryptographic runtime environment. Designed as a general mechanism for protecting both instructions and data, our system achieves a high level of security, and utilizes cache boundaries to greatly simplify the integrity checking process. The choice of FPGA hardware as a security-enabling mechanism greatly simplifies the required architectural modifications to a conventional CPU. Our simulations and prototyping experiments have shown that this approach is feasible, easy to implement, and on average adds low performance overhead to an existing cryptographic platform (see Tables 1,2 in Appendix A). As concerns regarding security have spread from the networking and software processing domains to the reconfigurable computing community, we see a continuing need to evaluate the potential of reconfigurable devices (such as FPGAs) as a trusted component in computing systems.

3 Security and Performance Tradeoffs: Region Based Security Levels.

One of our objectives was to study security and performance tradeoffs and to develop evolvable adaptive protection schemes. Our different code, and data, protection schemes resulted in different performance overheads and security strengths. The reconfigurable logic enables different security techniques to be implemented in hardware, and these different security techniques could be invoked by the software depending on what security strength was desired by a particular software module. We developed a model to apply different security schemes to different portions of the code based on a rudimentary risk analysis of the application code. We explored security driven code profiling to provide a proof of concept compiler-architecture environment that allowed us to tradeoff the security and performance.

We proposed and designed Region-Based Security (RBS), a compiler driven approach that combines risk-analysis and selective protection to help reduce the overhead in encrypted execution platforms. The key idea is that each block of code may be assessed for its vulnerability, following which protection is applied selectively. For example, variable declarations and mathematical operations are reportedly not as susceptible as control or data instructions. Hence, it makes sense to accord these more susceptible instructions a higher degree of protection. We contrasted this new approach with the standard encrypted-execution approach of protecting the entire executable. Implementation of the concept required addition of a risk analysis module to the compiler. This module is responsible for assessing the risk inherent in each block of code. The focus was not on developing novel risk analysis techniques; we assumed that any existing risk-analysis module can be used. Static analysis of source code is performed to find the vulnerabilities and assign risk levels to code segments – as a proof of concept, three levels were assigned: high, low and neutral. The object code augmented with the risk levels annotated into the code is then passed to the CODESSEAL system for assigning regions with risks. The CODESSEAL architecture then invokes the corresponding security algorithm, implemented on the reconfigurable logic in the hardware, during run-time.

Experimental results for the RBS technique, a summary of the results is shown in Table 4 in Appendix A, for a small set of benchmarks, demonstrate that the execution overhead is reduced considerably using this approach (see publication [3]). The results showed that RBS was an effective, and because it is complementary to the actual security mechanisms, can be implemented as an independent compiler module. An RBS optimization also leads to lower power consumption and memory usage, both valuable resources in embedded systems.

4 Compiler-Hardware Technique to counter Buffer Overflow Attacks

Buffer overflow attacks are widely accepted as one of the greatest threats to the security of networked systems. Mobile embedded devices that move between different networks and are capable of Internet connectivity present a unique challenge due to resource constraints. In [11], the architecture and compiler techniques developed were examined closely to develop a general solution to buffer overflow attacks. The hardware-software co-design technique, which draws on our techniques, was applied to combat the buffer overflow attacks. The technique does not require a redesign of the processor core, and has no impact on software development cost and low performance overhead. Code modification is done in the compiler tool-chain, therefore the programmer is not burdened by the extra validation mechanisms. Instrumentation directly on binary images was proposed in some cases, allowing for legacy code and libraries to be seamlessly retrofitted with the new security techniques. The special hardware modules do not require redesign of the processor core or changes in the instruction set architecture (ISA); memory-mapped instructions provided by the compiler/linker are used instead. The implementation of the hardware module can be accomplished by a number of contrasting methods, such as the use of a pre-synthesized soft core, a gateway chip that interfaces the CPU with the system bus, or a hard core processor with FPGA (field programmable gate array) fabric.

As a natural extension to our CODESSEAL architecture, the existing processor was augmented using the added hardware (FPGA logic) as a "Guard". The Guard monitors all communications between the CPU and main memory. Every instruction fetched by the CPU on an instruction cache (I-cache) miss goes through the Guard, and all instructions fetched by the CPU from main memory are available to the Guard. There is no processor modification; the main core does not need to be aware of the additional verification, which takes place outside the core boundary. A simplified view of the functionality implemented to protect the return address is:

- Upon a function call, the return address is stored in the Guard's hardware stack.
- Upon a function's return, the Guard inserts the correct return address back to the processor.

To provide these two steps, the Guard must be aware of both function calls and returns. To accomplish this, the compiler needs modification. Before every function CALL, the compiler inserts a memory-mapped "store to the Guard" instruction (*push-into-guard*) to push the function's return address to the Guard's hardware stack.

Similarly, another instruction (*notify_guard*) is added just before every function RETURN to trigger the Guard verification.

The performance of the technique was evaluated through simulations, using the same simulation infrastructure used for the other techniques (see Appendix A for details on the simulation infrastructure). Table 3, in Appendix A, summarizes the experimental results including the performance overhead. Because the protection technique operates on every function call, the performance penalty is highly dependent on the number of function calls for each program.

The buffer overflow protection technique achieves a number of advantages over prior solutions:

- It requires no modification of the processor or the underlying ISA, thus requiring minimal system changes. Processor re-manufacturing and the re-instrumentation of existing systems with new hardware can be very costly and typically requires both hardware and operating system updates and changes.
- On average the technique achieved a 7% performance penalty on the benchmarks that were tested.
- The system may require a minimal change to the operating system (as a small additional module for the context switching), but the secure memory storage for deeply nested stacks is handled by the Guard's encryption logic, so no additional protection of memory management is necessary from the operating system.
- Legacy software or libraries may be instrumented as binaries, so re-compilation may not be necessary. This gives the opportunity to use legacy libraries in a secure manner, even if the source code is not available.
- Programmers can use external libraries in a secure manner, without worrying about potential vulnerabilities to buffer overflows.
- Targeted protection of potentially vulnerable pieces of code may be performed, thus reducing the overall performance penalty.

5. FPGA Architecture Design

The key architectural concept in our approach was the use of reconfigurable logic, in the form of Field Programmable Gate Array (FPGA) technology, to provide hardware solutions for software security. In addition to exploring the design of the FPGA logic to support the security techniques (discussed in [2,4,10]), complementary uses of FPGA architectures for other secure software applications were also explored. In particular two architectural solutions were devised and tested – (i) processor-memory bus encryption and (ii) network intrusion detection. In [8], a processor-memory bus encryption technique for embedded systems that requires no changes to applications or hardware was proposed and evaluated. This technique exploits cache locking or scratchpad memory, features present in many embedded processors, permitting the operating system (OS) virtual memory subsystem to automatically encrypt data belonging to protected processes as it is written to off-chip memory. In [5,6], FPGA-based architectures for anomaly detection in network transmissions were designed.

Results show that this architecture correctly classifies attacks with detection rates exceeding 99% and false alarms rates as low as 1.95%. Further details on these solutions are provided below.

Network Intrusion Detection Architectures: Network Intrusion Detection Systems (NIDSs) monitor network traffic for suspicious activity and alert the system or network administrator. NIDSs can be classified into two types: *signature detection* and *anomaly* or *outlier detection*. Signature detection, or misuse detection, searches for well-known patterns of attacks and intrusions by scanning for pre-classified signatures in TCP/IP packets. On the other hand, anomaly detection is used to capture behavior that deviates from the norm and thus can be identified as malicious activity. Since such connections are described by large set of dimensions, processing these huge amounts of network data becomes extremely slow. Moreover, with the onset of Gigabit networks, current generation networking components for NIDS will soon be insufficient for numerous reasons; most notably because existing methods cannot support high performance demands.

Field Programmable Gate Arrays (FPGAs) are an attractive medium to handle both high throughput and adaptability to the dynamic nature of intrusion detection. In this work, we designed an FPGA-based architecture for anomaly detection in network transmissions (see publications [5,6]). We first developed a Feature Extraction Module (FEM) which aims at summarizing network information to be used at a later stage. Our FPGA implementation showed that we can achieve significant performance improvements compared to existing software and ASIC implementations. Then, we went one step further and demonstrated the use of Principal Component Analysis (PCA) as an outlier detection method for NIDSs. PCA is appealing since it effectively reduces the dimensionality of the data and therefore reduces the computational cost of analyzing new data. In our experiments even though each connection record has 41 features, we showed that PCA can effectively achieve over 99.9% detection rate with only 7 principal components. Results also showed that our architecture gives false alarms rates as low as 1.95% for KDD 1999 cup data sets.

We implemented our design on a Xilinx Virtex-II Pro FPGA platform, taking advantage of extensive pipelining and hardware parallelism. Overall, our architectures for FEM and PCA outlier analysis achieve up to 21.25 Gbps and 24.72 Gbps of core throughput, respectively, clocking at a frequency of 96.56 MHz. Hence we show that an FPGA implementation of NIDS not only gives ample flexibility but also satisfy the needs of Gigabit connections.

5. Personnel Supported

At The George Washington University:

- Professor Bhagi Narahari. Supported in part during summer months.

- Professor Rahul Simha. Supported in part during summer months.
- Dr. Olga Gelbart, graduate (doctoral) student, graduated 2008.
- Mr. Eugen Leontie, graduate (doctoral) student.

At Northwestern University:

- Professor Alok Choudhary. Supported in part during summer months.
- Prof. Joseph Zambreno, former doctoral student, graduated 2006, currently on the faculty at Iowa State University.
- Abhisek Das, Graduate student.
- D. Nguyen. Graduate student.

6. Publications

Copies of the publications listed below are also available at http://www.seas.gwu.edu/~narahari/afosr/

1. Secure Execution with Components from Untrusted Foundries. R. Simha, B. Narahari, J.Zambreno, A. Choudhary. In *Proc. of Advanced Networking and Communications Hardware Workshop (ANCHOR 2006),* held in Conjunction with Int. Symposium on Computer Architecture (ISCA), May 2006, Boston.
2. "Compiler-FPGA Technique to Detect Memory Spoofing in Encrypted-Execution Platforms", E. Leontie, O. Gelbart, B. Narahari, *Proc. 6th Annual Security Conference*, April 2007, Las Vegas.
3. "Compiler directed region-based security for low overhead software protection", Vijay Kongubangaram, Olga Gelbart, Rahul Simha, Bhagi Narahari. In *Proc. 3rd IEEE International Symposium on Dependable, Autonomic and Secure Computing (IEEE-DASC 2007)*, Sept., 2007.
4. "A compiler-hardware approach to software protection for embedded systems", Olga Gelbart, Eugen Leontie, Bhagirath Narahari, Rahul Simha. In *International Journal of Computers and Electrical Engineering, March 2009.*
5. "An Efficient FPGA Implementation of Principal Component Analysis based Network Intrusion Detection System." A. Das, S. Misra, S. Joshi, J. Zambreno, G. Memik and A. Choudhary. In *Proc. of Design, Automation & Test in Europe (DATE),* Munich, Germany, March 2008.
6. "An FPGA-based Network Intrusion Detection Architecture.", A. Das , D. Nguyen, J. Zambreno, G. Memik, and A. Choudhary. *IEEE Transactions on Information Forensics and Security (TIFS)*, Volume 3, Issue 1, March 2008.
7. "Architectural Support for Securing Application Data in Embedded Systems", E. Leontie, O. Gelbart, B. Narahari, R. Simha. In *Proc. IEEE Conf. Electronics and Info Tech (EIT),* Iowa, May 2008.

8. "Operating system controlled processor-memory bus encryption". X. Chen, R.P. Dick, A. Choudhary. In *Proc. of Design, Automation & Test in Europe (DATE),* Munich, Germany, March 2008.
9. "Providing Secure Execution Environments with a last line of defense against Trojan circuit attacks". G. Bloom, B. Narahari, R. Simha, J. Zambreno. Submitted to Computers and Security Journal. Under review.
10. "Detecting memory spoofing in secure embedded systems using cache-aware FPGA guards." E. Leontie, O. Gelbart, B. Narahari, R. Simha, J. Zambreno. Under submission to ACM Transactions on Embedded Systems.
11. "A Compiler-hardware technique for protecting against buffer overflow attacks on embedded systems." E. Leontie, O. Gelbart, G. Bloom, B. Narahari, R. Simha. Under submission to IEEE Transactions on Information Forensics and Security.

Doctoral dissertations:

- J. Zambreno. Compiler and Architectural Approaches to Software Protection and Security. Ph.D. Thesis. Electrical and Computer Engineering. Northwestern University, June 2006.
- Olga Gelbart. Integrated Hardware/Software Approaches to Software Security for Embedded Systems, D.Sc., Computer Science, The George Washington University. May 18, 2008.

7. Interactions/Transitions

- Presentation of papers at various conferences.
- Alok Choudhary, "Compiler and Architectural Approaches to Software Protection and Security," talk at Intel Corp, July 17, 2006.
- Bhagi Narahari, J. Zambreno, "Protecting Critical Computing Systems: A Hardware/Software co-design approach", presented at Institute for Defense Analysis (IDA), March 2006.
- Bhagi Narahari, "Hardware/Software co-design approaches to software security", presented to Network Defense Seminar, George Washington University, October 2006.
- Bhagi Narahari, "Integrated Software-hardware approaches to Software Security", presented at Cyber Defense conference, Rome Air Force Labs, May 2007.
- Bhagi Narahari, "Hardware/Software co-design approaches to software security", presented at Intel Corp, R&D Labs, Bangalore, India, December 2006.
- Bhagi Narahari, "Hardware/Software co-design approaches to software security", presented at HP Labs, Bangalore, India, December 2006.
- Rahul Simha, participated in the Digital Identity Systems Workshop at Polytechnic University in Brooklyn, September 2007.
- Presentation by GWU group to visitors from Rome Labs in 2008.

- The PIs at The George Washington University have been exploring collaborations with Intelligent Automation Inc (IAI) and with an engineer from Thales Communication Inc (a provider of tactical communications equipment to the US military and government). Currently no technology transfer initiatives have been identified, but discussions will continue.

New Discoveries, inventions, or patent disclosures. None. We will explore the possibility of patenting the FPGA implementations of secure coprocessing.

Honors/Awards

- Eugen Leontie, a graduate student at The George Washington University, received the Department of Computer Science's Hekimian award in 2008 for his research in architecture support for software security.

Appendix A: Experimental Platform and Results.

We implemented and evaluated our techniques through extensive simulations using the cycle accurate Simplescalar processor simulator. Our compiler techniques were implemented by modifying the GCC compiler.

Simulation Infrastructure.

Every tool and software we have chosen to use is open-source. The project is developed on Red Hat and Fedora Core Linux systems. We use the GCC [128] 3.3 cross-compiler, configured to produce statically-linked ARM executables, as our software development environment and the SimpleScalar 3.0 tool to simulate an ARM processor augmented with an FPGA. The SimpleScalar tool was set up to run with or without branch prediction, with 32-byte cache line size, and with various protection methods described in Chapter 4 and implemented in the FPGA.

Using an open-source compiler enabled us to add our own modifications to it in order to embed software protection mechanisms at compile-time. GCC provides a Control Flow Graph (CFG) data structure for the currently-compiling program. Since the CFG breaks up the program into basic blocks, it provides us a way of embedding control flow information (basic block labels) as well as integrity protection information (a SHA-1 hash or a CRC in this case) at the beginning of each basic block at compile-time. Using the compiler, we can also encrypt our software, by applying an encryption algorithm (AES in our case) to each 128-bit sized block of instructions. We are using the compiler in a cross-compiler mode with static linking to produce static executables for the arm-linux target with embedded protection information. Static executables are used since they encapsulate the entire control flow and function call graphs in one file.

We also used an open-source simulator for the ARM processor: SimpleScalar. We are simulating an ARM 1020E 400 MHz processor. We have augmented the simulator code with an implementation of an FPGA, which is how the Guard is implemented in hardware. We are using the characteristics of a Xilinx Virtex-II Pro FPGA at 200 MHz as our FPGA implementation. Thus every FPGA computation cycle that does not overlap processor execution creates 2 processor penalty cycles. The FPGA is positioned between the L1 instruction and data caches and the main memory. The external bus and main memory are assumed to run at 100 MHz. On every instruction or data cache miss, the FPGA is called and an instruction block is fetched into the FPGA. The block is verified and only then is passed to the processor's appropriate cache. The FPGA contains all implementations of the instruction verification mechanisms as well as the implementation of encryption and hashing algorithms. It also contains its own on-board memory of up to 3MB.

We have implemented our instruction protection scheme using two suites of bench-marks: MiBench and DIS MiBench suite of benchmarks has been specifically designed for a wide range of embedded applications: from image processing to network and tree analysis algorithms. We have chosen several benchmarks from different

categories to show that our scheme works for a variety of applications. The DIS Suite is a data-intensive suite of benchmarks designed to contain a large number of memory accesses. We have chosen these benchmarks particularly to investigate how well they can function under the data protection mechanisms.

The penalty cycles were estimated for instruction fetches assuming a 32-byte or 64 byte cache block. We note that multiple cache blocks may need to be accessed by the FPGA pre-fetch logic in case of a basic block that spans across multiple cache blocks. Recent FPGA implementations of AES manage to achieve high throughput by pipelining the execution path and unrolling techniques. The AES and SHA-1 implementations chosen by our model is one that minimizes the operational latency since high throughput is not the target in this architecture.

The timing requirements for implementing these algorithms on a Xilinx FPGA were obtained from commercial implementations.The 10 FPGA cycles for AES encryption/decryption translates into 20 processor penalty cycles that are added to the cache miss penalty. The FPGA implementation of SHA-1 takes 82 FPGA cycles which translates to 164 processor cycles. In the case where CRC is used in place of SHA-1, the guard takes 2 cycles. The other Guard processing time requirements is 1 cycle for address validation. The space utilization by the fpga guard, to implement the various algorithms, is low compared to the amount of logic available on the chips that we considered. For example, the AES-decryption takes 284 slices out of 10K available slices on a Xilinx Virtex2Pro.

Experimental Results.

Table 1 summarizes the performance of the different schemes on a set of selected from the MI-Bench suite and the Data Intensive Systems (DIS) suite of benchmarks. The table shows the performance overhead, measured as the percentage increase in execution time (when compared to encrypted execution) of the benchmark, for each of our schemes. The first two columns (to the right of the benchmark) show the performance for the Cache Block granularity. The next four columns show the performance for the Basic Block granularity – using the CRC and SHA-1 signature schemes. For both granularities, we examined the two cases where signatures are stored (a) internally in the code block and (b) external to the code block in a separate portion of memory. For both the Cache block and Basic block schemes, storing signatures internally led to higher performance overhead. Comparing the last four columns, we observed that while SHA-1 provides the higher security strength than CRC, it incurs a much higher overhead due to both the time required to compute SHA-1 on the FPGA and the amount of space needed to store the 160-bit signature.

We implemented and tested the data protection schemes. Since data is fetched on cache misses, we only dealt with a cache block granularity for data protection. We designed three levels of data protection schemes: (a) only encryption of data, (b) encryption with data block labels, and (c) encryption, labels and signature. Only encryption does not prevent data injection and substitution attacks. To prevent data injection we require address labels and signatures, analogous to code protection schemes. We note that an attack where stale data is substituted is still possible under our current data protection

schemes, and our ongoing efforts are focusing on embedding a timestamp into the data. The performance overhead is once again measured as the percentage increase in execution time (when compared to encrypted execution) of the benchmark. Table 2 summarizes the performance overhead of our data protection schemes for some of the benchmarks. We observe that signatures incur larger overheads, and the overall performance overheads are within acceptable ranges.

For the simulation of the buffer overflow protection technique, we again used the SimpleScalar[14] simulation suite and the gcc cross-compiler for the ARM processor. Our processor contained one level of instruction and data caches, each 32KB in size, 32-way associative, with 32-byte cache lines. The simulator used was sim-outorder. The processor chosen was for an average embedded system: a 400MHz processor augmented with a 200MHz FPGA (for the Guard), and an external bus and memory running at 100MHz. We tested the simulation with benchmarks from MiBench [15] and Data Intensive Systems (DIS) [16]. Table 3 provides a summary of the experimental results; the baseline configuration corresponds to no protection.

For evaluating the adaptive security scheme – the Region based security (RBS) scheme – we implemented the scheme in the gcc compiler. The RBS scheme was then tested on our simulation infrastructure. Table 4 summarizes some of the key results of our experiments. Two security schemes are considered – a SHA-1 signature scheme and a CRC scheme. While SHA-1 has higher security strength, it also incurs a larger penalty. The performance penalties incurred for each security scheme is shown in the table for various benchmarks. The RBS based adaptive security scheme incurs lower penalty in most of the benchmarks.

Table 1: Performance of Code Protection Techniques

BenchMarks (taken from MI Bench benchmark suite and Data Intensive Systems, DIS, suite)	Cache Block Labels		Basic Block Labels			
			CRC		SHA-1	
	Internal Storage (Labels Stored in the Cache Block)	External storage (Labels interleaved with cache blocks and stored separate from code block)	External storage (Labels stored separate from code block)	Internal storage (Labels stored inside code block)	External Storage (Signature, and label, stored separate from code block)	Internal storage (Signature, and labels, stored in code block)
Bitcount	7.55	0.06	0.02	9.74	0.18	47.40
Crc	8.02	0.04	0.02	17.02	0.15	99.15
Dijkstra	3.89	3.94	0.02	11.57	0.54	58.01
Fft_inv	5.23	5.26	0.08	14.08	3.51	47.54
Fft	5.95	2.87	0.08	13.35	1.25	39.16
Sha	9.03	0.16	0.07	5.24	0.48	22.10
Stringsearch	10.21	12.5	5.68	20.54	38.30	110.48
susan.corners	9.01	9.72	1.72	7.79	11.54	26.24
Susan.edges	8.6	5.23	0.94	5.69	0.08	18.40
susan.smoothing	8.28	0.17	0.04	7.79	0.30	24.59
Field	0.03	0.04	0.01	2.28	0.08	11.02
Pointer	0.04	0.03	0.01	6.74	0.12	33.72
Transitive	9.89	9.3	5.28	12.64	35.24	70.50
Update	18.53	16.7	9.24	23.11	62.21	114.28
Average	7.45	4.72	1.66	11.26	11	51.61

Benchmark	Encryption Only	Encryption & Labels	Encryption, Hashing, Labels
Bitcount	0.02	0.024	0.07
Crc	0.02	0.021	0.061
Djikstra	5.27	5.50	15.61
Fft	3.80	3.89	10.71
Sha	0.10	0.11	0.30
Stringsearch	9.74	10.22	29.63
Field	0.027	0.028	0.083
Transitivc	4.72	4.96	14.27
Update	9.70	10.18	29.39

Table 2: Percentage Increase in Execution Time using Data Protection.

Benchmark	baseline	Function Call Frequency	# Function calls	# Function Returns	Stack protection (Cycles)	%penalty
bitcount	1985181143	0.74	14625715	14625650	2159563042	8.78
crc	2661916962	3	79853258	79853236	3407158679	28
dijkstra	629872896	0.1	648251	648238	634215728	0.69
fft	634777292	1.02	6478257	6478223	685360839	7.97
fft(inv)	358609172	0.7	2494613	2494579	373084425	4.04
patricia	916560445	0.92	8446864	8446851	1020045067	11.29
sha	472226975	0.02	105630	105609	474143316	0.41
stringsearch	9012582	0.45	40799	40787	9248271	2.62
susan_smooth	1216801457	0.01	113712	113690	1217489061	0.06
susan_edge	174595349	0.06	97364	97345	175464647	0.5
susan_corners	65129481	0.02	13143	13111	65203728	0.11
field	1275431654	0.06	788032	788016	1283041545	0.6
pointer	409127443	0.53	2160907	2160895	426517463	4.25
tc	31323833	0.06	19480	19468	31528118	0.65
update	5359253	1.1	58806	58794	5908967	10.26

Table 3: Performance of Buffer Overflow Protection Technique

Benchmark	% performance penalty					
	Entire system (internal storage, SHA-1)	RBS-Based approach	Entire System (internal storage, CRC)	RBS-Based approach	Entire System (external storage)	RBS based approach
bitcount	47.3	22.46	9.46	4.11	0.10	0.09
crc	99.08	54.06	17.01	7.39	0.08	0.07
dijkstra	56.96	27.04	11.41	4.96	0.48	0.48
fft	41.2	23.52	12.42	5.40	1.23	1.16
patricia	1631.26	497.7	43.47	18.89	0.39	0.34
sha	21.83	4.28	5.20	2.26	0.27	0.24
stringsearch	89.13	34.42	16.86	7.33	20.05	17.56
susan	22.12	11.9	7.38	3.20	5.80	5.17
field	11.92	3.04	2.73	1.19	0.04	0.03
pointer	38.85	17.82	8.23	3.57	0.16	0.15
transitive	31.15	9.95	7.12	3.09	8.72	7.92
update	56.64	39.02	15.71	6.83	18.36	16.61

Table 4: Performance of Region-based Security Technique