Hardware Architectures for Software Security

Joshua N. Edmison

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

Dr. Mark T. Jones, Chair
Dr. Lynn Abbott
Dr. Peter M. Athanas
Dr. Ezra Brown
Dr. Thomas L. Martin
Dr. Cameron D. Patterson

June 29, 2006
Bradley Department of Electrical and Computer Engineering
Blacksburg, Virginia

Keywords: Security, architecture, FPGA, configurable, graph theory, information flow
Copyright 2005 ©, Joshua N. Edmison
Hardware Architectures for Software Security

Joshua N. Edmison

(ABSTRACT)

The need for hardware-based software protection stems primarily from the increasing value of software coupled with the inability to trust software that utilizes or manages shared resources. By correctly utilizing security functions in hardware, trust can be removed from software. Existing hardware-based software protection solutions generally suffer from utilization of trusted software, lack of implementation, and/or extreme measures such as processor redesign. In contrast, the research outlined in this document proposes that substantial, hardware-based software protection can be achieved, without trusting software or redesigning the processor, by augmenting existing processors with security management hardware placed outside of the processor boundary. Benefits of this approach include the ability to add security features to nearly any processor, update security features without redesigning the processor, and provide maximum transparency to the software development and distribution processes. The major contributions of this research include the augmentation methodology, design principles, and a graph-based method for analyzing hardware-based security systems.
Acknowledgements

The work presented in this thesis could not have been accomplished without the support and guidance from a variety of individuals. For those who are not mentioned specifically, thank you.

I would like to thank my advisor Dr. Mark Jones for his support, confidence, and enlightenment over the last five years. His advice on academics, career choices, and general life issues has been indispensable.

My thanks go out to Dr. Peter Athanas and Dr. Cameron Patterson for their many hours of discussion, advice, guidance, and suggestions. Furthermore, I would like to thank them for serving on my committee.

I would to thank Dr. Tom Martin, Dr. Ezra Brown, and Dr. Lynn Abbott for taking the time to serve on my committee.

My gratitude goes out to all of my colleagues in the Configurable Computing Lab. Thanks for all of the help and support. Special thanks goes out to current and past members of the Secsoft Team for their phenomenal effort: Eric Lorden, Anthony Mahar, and Ben Muzal.

To all of my friends in Blacksburg, thanks for helping me retain my sanity.

I would like to thank my parents, Jeff and Rhonda Edmison, and sister, Bridgette, for their endless support, love and interest throughout my academic career. I certainly would not have been successful without them. They are a continued source of inspiration.

Finally, I can’t thank my wife Gina enough for her love, support, and sacrifices throughout my entire academic career. Thanks for putting up with my crazy schedule and listening when I needed it most.
Contents

1 Introduction 1
  1.1 Motivation ........................................... 1
  1.2 Contributions ........................................... 2
    1.2.1 Analysis ........................................... 4
  1.3 Document Organization .................................... 4

2 Literature Review 5
  2.1 Software Protection ....................................... 5
    2.1.1 Instruction Protection ................................. 5
    2.1.2 Data Protection ...................................... 6
    2.1.3 Execution Flow Protection ............................ 6
    2.1.4 Software Protection Summary ........................ 7
  2.2 Security Architectures ..................................... 7
    2.2.1 Instruction & Data Protection ......................... 8
      Hardware-Only & Hardware-Assisted Solutions .............. 8
      (Dedicated) Secure Coprocessors .......................... 11
      Software Solutions ...................................... 11
    2.2.2 Execution Flow Protection .......................... 12
      Hardware Solutions ...................................... 12
      Software Solutions ...................................... 14
  2.3 Formal Security Analysis .................................. 15
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3.1 Security Models</td>
<td>16</td>
</tr>
<tr>
<td>3 Information-Flow Hardware Security Analysis</td>
<td>18</td>
</tr>
<tr>
<td>3.1 Formal Methods and Security Analysis</td>
<td>18</td>
</tr>
<tr>
<td>3.1.1 Theorem Proving</td>
<td>19</td>
</tr>
<tr>
<td>3.1.2 Model Checking</td>
<td>20</td>
</tr>
<tr>
<td>3.2 Hardware - Software Interaction and Information Flow Models</td>
<td>20</td>
</tr>
<tr>
<td>3.3 Hardware Security Analysis and Information-Flow Graphs</td>
<td>22</td>
</tr>
<tr>
<td>3.4 Analysis Process &amp; Examples</td>
<td>27</td>
</tr>
<tr>
<td>3.4.1 XOM</td>
<td>27</td>
</tr>
<tr>
<td>XOM Model</td>
<td>28</td>
</tr>
<tr>
<td>XOM Analysis</td>
<td>29</td>
</tr>
<tr>
<td>3.4.2 AEGIS</td>
<td>30</td>
</tr>
<tr>
<td>AEGIS Model</td>
<td>30</td>
</tr>
<tr>
<td>AEGIS Analysis</td>
<td>31</td>
</tr>
<tr>
<td>3.5 Summary</td>
<td>35</td>
</tr>
<tr>
<td>4 The Secure Software Platform</td>
<td>36</td>
</tr>
<tr>
<td>4.1 Architecture Overview</td>
<td>36</td>
</tr>
<tr>
<td>4.1.1 Security Model &amp; Assumptions</td>
<td>36</td>
</tr>
<tr>
<td>4.1.2 Assumptions</td>
<td>38</td>
</tr>
<tr>
<td>4.2 Architecture Design Preliminaries</td>
<td>38</td>
</tr>
<tr>
<td>4.2.1 Design Goals</td>
<td>38</td>
</tr>
<tr>
<td>4.2.2 Design Questions and Challenges</td>
<td>39</td>
</tr>
<tr>
<td>4.3 Instruction Protection</td>
<td>41</td>
</tr>
<tr>
<td>4.3.1 Encryption Management Unit</td>
<td>41</td>
</tr>
<tr>
<td>4.3.2 Secure Key Management Unit</td>
<td>43</td>
</tr>
<tr>
<td>4.3.3 Secure Executables</td>
<td>43</td>
</tr>
<tr>
<td>4.3.4 Operating System Support</td>
<td>44</td>
</tr>
</tbody>
</table>
4.3.5 Instruction Protection Operational Summary ........................................ 45
4.3.6 Encryption Algorithm and Mode of Operation Selection ......................... 47
4.3.7 Instruction Integrity .............................................................................. 50
   Galois Countermode Operation .................................................................. 52
4.4 Data Protection ......................................................................................... 54
   4.4.1 Read-only/Static Data ........................................................................ 54
   4.4.2 Associating Instructions with its Data ................................................. 55
      Shared/Dynamic Library Support ............................................................. 56
      Consideration of Internal Processor State .............................................. 57
   4.4.3 IEMU Modifications for Data Protection ........................................... 59
   4.4.4 SKU Modifications ............................................................................ 59
   4.4.5 Secure Executable Modifications ....................................................... 59
   4.4.6 Operating System Modifications ....................................................... 59
   4.4.7 Data-side Protection Operational Summary ....................................... 60
   4.4.8 Encryption Algorithms and Modes of Operation .............................. 60
   4.4.9 Data Integrity ..................................................................................... 60
      CRC for Data Integrity Checking ............................................................. 62
   4.4.10 Operating System Interaction & I/O ................................................ 63
   4.4.11 Architecture Summary ..................................................................... 65
5 Implementation, Analysis, & Results ............................................................ 70
   5.1 Implementation ...................................................................................... 70
      5.1.1 Prototype Platform .............................................................. 70
         The PowerPC 405 Processor ............................................................ 71
      5.1.2 Encryption Management Units .................................................... 72
      5.1.3 Secure Key Management Unit ...................................................... 72
      5.1.4 Secure Executable Postprocessing Utility ..................................... 72
      5.1.5 Operating System Kernel Modifications ...................................... 73
   5.2 Security Analysis ................................................................................. 73
List of Figures

3.1 Example information flow graph for a standard computing system architecture .......................... 23
3.2 The multigraph resulting from applying a set of functions at an vertex to an incoming edge ................................................................. 24
3.3 The multipath (graph) resulting from the ordered application of functions set to incoming edges at each vertex along a path ........................................... 25
3.4 High-level XOM architecture ......................................................................................... 28
3.5 Information flow diagram for the XOM architecture evaluated under Rule 3.10 ........................... 31
3.6 Information flow diagram for the XOM architecture evaluated under more restrictive Rule 3.7 that doesn’t allow any unencrypted flows. This graph highlights the unintended consequence for instructions to be downgraded to the shared compartment due to the use of a single compartment key. 32
3.7 Information flow diagram for the AEGIS architecture evaluated using Rule 3.9 without software trust (security kernel) ................................................. 33
3.8 Information flow diagram for the AEGIS architecture evaluated using Rule 3.9 under the assumption of software trust (security kernel) ............................. 34
3.9 Information flow diagram for the AEGIS architecture under the assumption of software trust (security kernel) and using a single key for cryptographic operations ................................................................. 35
4.1 Top level view of the SSP design approach ........................................................................ 37
4.2 Hardware subcomponents of the security management unit (SMU) for instruction protection ......................................................................................... 41
4.3 Internal structure of the instruction-side encryption management unit ................................. 42
4.4 Credential identifiers mapping to credential values/information ........................................ 43
4.5 Secure executable object layout ...................................................................................... 45
4.6 Depiction of the operating system interaction with the SKU. Since the SKMH is encrypted and verified by the SKU hardware, the operating system cannot alter the contents of the SKMH in a manner that adversely affects the key generation process. Also, the channel between the SKU and the user token is assumed to be secure. The result of the SKU operation are cleartext keys (created from the credentials on the token) that are used in SSP operation.

4.7 Depiction of the page table additions to support keyslot information.

4.8 Process flow for the setup and execution of an instruction protected application on the SSP.

4.9 Process flow for the setup and execution of an instruction protected application on the SSP (continued from previous page).

4.10 AES counter mode operation.

4.11 This figure outlines the protocol for using a cryptographic hash for integrity checking on the SSP for a read-only scenario. Since read-only data is known a priori the hash values can be computed prior to execution.

4.12 In this depiction of Galois counter mode, note the feedback of the encrypted counter mode output to the Galois multiplier along with additional data to be authenticated. The resulting tag is a cryptographically secure hash that can be stored off chip and and retrieved for comparison to a newly calculated hash of the off-chip data.

4.13 Hardware subcomponents of the security management unit (SMU) for instruction protection with integrity checking (shown with GCM style checking).

4.14 Hardware subcomponents of the security management unit (SMU) for instruction and data protection.

4.15 The tag method for associating code and data.

4.16 Depiction of key set differentiation through tagging.

4.17 The data side EMU (DEMU) internals.

4.18 The data side process flow.

4.19 Hardware subcomponents of the security management unit (SMU) for combined instruction and data protection with integrity verification.

4.20 Tampering by an adversary off-chip results in an erroneous decryption.

4.21 Protocol for using a cryptographic hash for integrity checking on the SSP in a read/write scenario (data protection).
4.22 Protocol for using a CRC for integrity checking on the SSP in a read/write scenario ........................................ 65
4.24 CRC calculation latency .................................................. 67
4.25 Memory requirements for storing CRC values for an entire thirty-two bit memory space on-chip (Note: Memory requirements reduced dramatically if protection is limited to a certain portion of physical memory (this requires kernel mods and limits the total number of protected process)) .......... 68
4.26 A high-level view of the SSP operation that details a typical usage scenario .................................................. 69

5.1 This figure depicts the implementation of the SSP platform. Multiple PLB busses in combination with PLB-PLB bridges allow the shoehorning of the IEMU and DEMU into the memory hierarchy. The SKU interacts with the operating system and token via an OPB bus. ...................... 71
5.2 (a) Graph analysis results for a single context SSP systems that includes the icache_debug instruction. Note the undesired downgrade path (dashed), caused by the ability to move unencrypted instructions to registers from the instruction cache. (b) Graph analysis results, showing no violations, for an SSP system that incorporates instruction filtering to remove the affects of the icache_debug instruction .................................................. 74
5.3 Graph analysis results for multi-context SSP systems that include both instruction and data protection. The lack of violations indicates that the SSP architecture is secure under the stated assumptions .......................... 76
5.4 Attack difficulty in time for the CRC integrity scheme when using larger CRC bit widths ........................................................................ 78
5.5 Attack difficulty in time, for the CRC integrity scheme for smaller CRC bit widths (Note that the units of x axis are in hours, not minutes as in Figure 5.4) Also note that a specific halt time was not used. It is necessary that the particular countermeasure deployed limits the attack times to those shown on the left axis .................................................. 78
5.6 SSP Costs ........................................................................... 80
5.7 Simulated benchmark instruction and data cache miss statistics for the PPC405 processor ........................................................................ 85
5.8 Simulated benchmark instruction and data cache miss statistics for the PPC405 processor ........................................................................ 87
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Increase in size of the compiled kernel’s executable section for different kernel configurations</td>
<td>81</td>
</tr>
<tr>
<td>5.2</td>
<td>Increase of the compiled kernel’s data section for different kernel configurations</td>
<td>81</td>
</tr>
<tr>
<td>5.3</td>
<td>Executable file size increases for each of the benchmarks after post-processing using the SELF generation utility</td>
<td>82</td>
</tr>
<tr>
<td>5.4</td>
<td>Hardware logic consumption numbers for each of the major SSP components.</td>
<td>83</td>
</tr>
<tr>
<td>5.5</td>
<td>List of variables for the analytical SSP performance discussion</td>
<td>83</td>
</tr>
<tr>
<td>5.6</td>
<td>Predicted slowdowns for the benchmark applications using the derived analytical models for both the IEMU and DEMU SSP systems</td>
<td>86</td>
</tr>
<tr>
<td>5.7</td>
<td>Student’s t-test values for each of the comparison cases across all benchmarks</td>
<td>90</td>
</tr>
<tr>
<td>5.8</td>
<td>Student’s t-test values for each of the comparison cases across all benchmarks (lengthened benchmarks)</td>
<td>91</td>
</tr>
<tr>
<td>5.9</td>
<td>User time mean and variance for the fifteen runs of the original benchmarks</td>
<td>92</td>
</tr>
<tr>
<td>5.10</td>
<td>System time mean and variance for the fifteen runs of the original benchmarks</td>
<td>93</td>
</tr>
<tr>
<td>5.11</td>
<td>User time mean and variance for the fifteen runs of the lengthened benchmarks</td>
<td>94</td>
</tr>
<tr>
<td>5.12</td>
<td>System time mean and variance for the fifteen runs of the lengthened benchmarks</td>
<td>95</td>
</tr>
<tr>
<td>5.13</td>
<td>Absolute (percent from base) and relative (percent from previous) for the unmodified benchmarks. Note that the majority of the slowdown results from the kernel additions. Also, instruction and data cache flushing on context switches only contributes 7.5% slowdown.</td>
<td>96</td>
</tr>
<tr>
<td>5.14</td>
<td>Absolute (percent from base) and relative (percent from previous) for the lengthened benchmarks. Note that the majority of the slowdown results from the kernel additions. Also, instruction and data cache flushing on context switches is smaller than the original benchmark case.</td>
<td>97</td>
</tr>
</tbody>
</table>
6.1 Comparison of assumptions between the existing TCPs XOM and AEGIS with the SSP architecture .......... 101
Chapter 1

Introduction

1.1 Motivation

Several factors contribute to the growing necessity for hardware-based software security methods. The increased reliance upon and usage of computing systems across a broad range of domains and applications results in the inclusion of information within software that may be sensitive, valuable, and/or proprietary. Additionally, the growing complexity of applications and the corresponding execution platforms results in greater time and resource expenditure during the development phase of the application life-cycle; increasing the value of the software, the data it produces, and the speed at which data is produced [3].

While the added value due to resource expenditure and the existence of sensitive information are alone sufficient reasons to protect software, consideration of modern computing system design solidifies the case for hardware-based software protection. Modern computing systems have traditionally been designed explicitly with resource sharing and performance as the primary considerations. Because resource sharing is managed exclusively by the operating system, even slight exploitation or malicious modification to the operating system can allow an entity (application) to steal information from or about other entities (applications) utilizing shared resources. Considering the extreme complexity of the modern operating system coupled with the inherent flexibility of software, the probability of error or circumvention further increases, eliminating the ability to trust the operating system or any other software with security functionality. As a result, hardware mechanisms should be utilized to provide reliable software protection.

The concept of adding security features to computing systems via hardware is certainly not new (as detailed in Chapter [2]). The usefulness and adoption of hardware-based security functions is and has been hampered, among other reasons, by the attention given to performance over security during the design process, concerns over digital rights, concerns over privacy, and concerns regarding the inability to change, add, or remove hardware security
features. [4].

Existing solutions in the problem domain suffer from a variety of drawbacks that include impracticality in the form of processor redesign, assumptions such as trusting software, a lack of implementation beyond simulation, and failure to consider the effects on the overall usage model. It is important to note that because no system is completely secure from all attacks, each system must be designed with a certain set of security requirements and associated assumptions.

The research outlined in this document demonstrates that software security features can be enabled at a reasonable cost and without trusting software, by externally augmenting conventional computing devices with additional hardware. Also resulting from this research were a generic set of design principles in the form of processor characteristics needed to support the external augmentation methodology and a graph-based model for analyzing security properties of hardware-based security systems. The feasibility of the architecture is demonstrated by the implementation.

1.2 Contributions

The major contributions stemming from this research are as follows:

- A system was designed that can provide hardware-based software security without redesigning the processor and without trusting an operating system. The architecture, presented in Chapter 4, resides beyond the processor boundary, results in reasonable performance impact, and does not require significant changes to the software development flow.

- A set of processor interfaces required for the external augmentation approach to hardware-based software security were defined in Chapter 5. Future processors can utilize the set of interface requirements to create a Security Management Interface enabling “plug and play” security. By using a generic interface for security functions, verification effort need not be wasted on each processor-specific security implementation. Instead, verification efforts can be focused on the added security hardware. The ability to mix and match processors with security hardware may also increase the breadth and capabilities of security devices.

- A set of processor feature requirements for the external augmentation approach were defined in Chapter 5. The feature set can be used to evaluate existing processors for compatibility with the external augmentation methodology. Also, processor designers can incorporate (or not incorporate in the case of undesirable features) the feature set into future processors to enable the use of the external augmentation methodology.
• The cryptographic separation of instruction and data datapaths is proposed and is a feature of the SSP architecture as discussed in Chapter 4. The cryptographic separation of datapaths, in addition to creating a second cryptographic problem for an attacker, prevents the manipulation of instructions (reading or writing) by the data-side processor interface. More specifically, instructions cannot be retrieved as data and data cannot be executed (a common vector for attack in modern systems).

• A new method of integrity checking that reduces latency, allows flexible protection levels, and reduces possible attacks was proposed in Chapter 4. The new integrity verification mechanism, based on CRC, leverages the unique situation created by a trusted computing platform to remove the need for a cryptographic hash algorithm given sufficient on-chip storage space. Additionally, the mechanism provides inherent freshness assurance by operating on the non-transformed data. Finally, a security analysis of the CRC-based method performed in Chapter 5 shows that the CRC can provide substantial security at a low cost.

• The flexibility of secure applications was improved, as a direct result of the SSP design outlined in Chapter 5, through per-page rather than per-application key assignment. Using per-page keys, different portions of an application can have different types of cryptographic protection facilitating security differentiation. As a result, per-page keys enables cryptographic separation of datapaths. Finally, the use of per-page keys supports the closed-source software distribution model, allowing object files to be independently protected.

• An alternative and easy applicable graph-based model for use in analyzing hardware-based security systems, discussed and formalized in Chapter 3, was designed and used to verify the security of a specific instance of the SSP architecture in Chapter 5. The graph model exhibits sufficient flexibility to apply to a variety of architectures as demonstrated by its application to two existing hardware-based security architectures, namely XOM [5] and AEGIS [6].

• Demonstrated by an implementation (discussed in Chapter 5), the augmentation approach to hardware-based software protection was shown to be practically realizable and well understood. A working implementation indicates that the SSP architecture is immediately usable. Additionally, the measurement and benchmark results outlined in Chapter 5 explain and account for each of the spatial and temporal costs of the SSP. Also, the result support the assertion that the performance and overhead of the system are reasonable.

• The SSP architecture implementation was evaluated under a real, extant multitasking operating system. Enabled by the physical implementation, evaluation under a real operating system allowed for a much more accurate assessment of overall system performance as well as the operating system’s impact upon performance. The benchmark
results in Chapter 5 indicate that even in a support role, the operating system contributes significantly to the overall impact upon performance. Previous efforts, that used only simulation for characterization neglected this important aspect of system characterization.

1.2.1 Analysis

Analysis of system security is performed using a graph-based system model that applies information-flow concepts [7] [8] [9] to hardware security analysis. Using this technique, it is possible to quickly evaluate design decisions and explore architectural requirements for different types of protection. While not meant to be the primary focus of this work, the analysis method is a significant contribution to the domain. In addition to the security analysis, a generic design approach for using the augmentation methodology is developed and outlined.

1.3 Document Organization

Chapter 1 presents the motivation, approach, and contributions of this research. Chapter 2 discusses background information and currently related efforts within the problem domain, focusing on the areas of software protection, security architectures, and security analysis. The graph-based security analysis method is presented in Chapter 3. Design challenges, research goals, and the (Secure Software Platform)(SSP) architecture are presented in Chapter 4. Chapter 5 presents the details of the SSP implementation and a system security analysis along with benchmarking results. Finally, Chapter 6 summarizes the research performed, details the contributions, and presents future work.
Chapter 2

Literature Review

The design and analysis of a secure system requires an understanding of what aspect of the system is being protected and the types of attacks typically directed at the system. Section 2.1 outlines the problem of software protection, several different examples of attacks on software, and defines categories of software protection. Current and past secure architecture efforts and their corresponding features are presented in Section 2.2. Finally, Section 2.3 provides an overview of prominent formal methods and formal security models along with examples of their application to secure system verification.

2.1 Software Protection

Before existing software protection solutions and security analysis methods can be discussed aspects of software security must be reviewed. Software can be logically decomposed into two categories: executable code and data. The functionality of the software is completed by the flow of execution transferring between portions of executable code where operations are performed on or using data. Using these categories software protection can be classified into three major areas: 1. instruction protection; 2. data protection; 3. execution flow protection.

2.1.1 Instruction Protection

Instruction protection attempts to prevent executable instructions from being known (confidentiality) or to ensure that executable instructions fetched from memory are unmodified (integrity). Extending instruction protection further is the concept of preventing an attacker from inferring the function of a protected block of executable instructions and/or how that function is being performed. To demonstrate the importance of this subtlety, as an example,
suppose a block of executable instructions are protected for both confidentiality and integrity using standard cryptographic techniques. Also assume that the executable code cannot be directly read or modified and is being attacked by an individual named Mallory. Using any combination of analysis techniques (power, execution time, monitoring memory/disk access patterns), Mallory determines that the block of code is performing a filtering operation. Suppose after extended analysis and probing (modifying input data in memory), Mallory is able to determine that the filter operation being performed in the block of code actually consists of two filters the first being lowpass with a 3dB point at 1500Hz and the second a bandpass filter with a center frequency of 520 Hz.

Mallory has, without read or write access to the executable instructions, determined details about the application with enough information to possibly recreate the functionality of the code block. Note that the block of executable instructions, which in themselves have value since they are capable of directly interacting with the hardware to provide the desired functionality, are still protected. However, the higher-level functionality and even detailed internal workings may be obtainable.

2.1.2 Data Protection

Data protection, like instruction protection, attempts to prevent data from being known (confidentiality) or to ensure that data retrieved from memory is unmodified from the intended version (integrity). Data protection, however, differs fundamentally from instruction protection because data, unlike instructions, must be read and processed by instructions. As a result, data protection requires that data and the code that uses the data, must be associated to one another. Unlike instructions, data does not exhibit implicit functionality that can be inferred. Inferences concerning the contents of protected data rely upon understanding the functionality of the instructions that operate on the data. Because data in modern computing systems is highly structured and exhibits well defined representations, observation of data access patterns or using knowledge of how certain data structures are stored may enable other types of inference concerning data contents or instructions that operate on the data [10] [11] [12]. Data protection is further complicated by the need to share data between applications and export the results of an application for interpretation or use elsewhere.

2.1.3 Execution Flow Protection

Execution flow protection aims to prevent the flow of the application execution from being observed (confidentiality), being maliciously altered (integrity), and/or from being used without authorization (initial flow). An example of execution flow confidentiality is preventing an attacker from obtaining an algorithm or application functionality in the form of flow between functions or other code blocks of interest. Examples of execution flow protection
integrity include enforcing the execution flow such that an undesired piece of code cannot be executed and preventing the intended functionality of the application from being altered. Finally, initial flow protection attempts to prevent other malicious applications from utilizing functionality provided in a protected application or one of its subcomponents by forcing execution flow to an entry point.

### 2.1.4 Software Protection Summary

Given the software protection categories, the properties of a secure software application can be summarized as follows:

1. Instructions can not be directly stolen, inferred, or modified;
2. Data can not be directly stolen, inferred, or modified;
3. Algorithms and methods can not be stolen or inferred;
4. Execution flow can not be tampered with or monitored;
5. The application and all of its subcomponents can only be used by authorized entities;
6. Authorized users cannot be denied usage of or access to an application or its subcomponents.

Nearly all direct attacks against software, despite variations in methodology and sophistication, will exhibit intentions that fall under one of the previously stated protection properties. While it is generally accepted that no system can completely protect against all classes of threats, the properties listed above provide a general upper bound on protection goals for all systems that attempt to protect software.

### 2.2 Security Architectures

The following sections present an overview of architectures for securing software. Discussions of design issues associated with software protection architectures can found in [13] [14]. These design issues are extended in [15], which discusses similar issues arising from the combination of software security and multiprocessors.

The purpose, protection, and design assumptions of security architectures ranges from custom designed processors to small jump tables. Because many of the architectures that protect instructions also attempt to protect data, one classification of architectures will encompass both instruction and data protection. The second classification will include methods for protection execution flow. Although the work presented in this document focuses on
hardware methods of software protection, both hardware and software approaches for each classification must be evaluated and understood. Where appropriate, the specific type of protection provided (confidentiality, integrity, etc.) is noted.

2.2.1 Instruction & Data Protection

Hardware-Only & Hardware-Assisted Solutions

The concept of using hardware to protect software was publicly introduced originally by Robert Best who patented the idea of a crypto-microprocessor in the early 70's [16] [17]. The crypto-microprocessor was a microprocessor outfitted with encryption hardware that could decrypt encrypted memory for a single application using a single key. This concept was again revisited in [18] and was realized as a commercial product in the Maxim DS5002FP [19] secure microcontroller which could access 128kB of encrypted random access memory (RAM). Several years later Herzberg et. al. [20] examined aspects of software distribution when using a protected processor. The result of the work performed by Herzberg et. al. was a proposed protocol for distributing software that could utilize different types of cryptosystems and different types of distribution chains. The Abyss [21] architecture extended the work by Herzberg et. al. by only requiring that a secret be kept on the processor.

Abyss was an architecture that utilized a processor with a physical secret and introduced an authorization method called a token process. The token process acted as a single use device that could be accessed once and then could not be used again. This allowed for a vendor or other entity to authorize, for example, a single install which was represented in the Abyss architecture by a right-to-execute file. The right-to-execute file contained various pieces information needed to execute the application such as decryption keys. Without this file, the application could not be executed and effectively separated an application from the ability to execute it. Supervisor software, such as an operating system (OS), was trusted to disallow or authorize execution using the right to execute file. Abyss was one of the first proposals to use a secure coprocessor for software protection and to present the concept of partitioned computation [22].

The use of cryptoprocessors in a multitasking environment was first presented by Markus Kuhn who introduced the TrustNo 1 cryptoprocessor [23]. The TrustNo 1 architecture was an entire redesign of a processor to support secure operation. The processor design attempted to provide process compartmentalization by providing secure context switches, secure saves of the processor state, and tagged caches with key IDs to prevent unauthorized reuse of data in caches. TrustNo 1 did not support two applications with the same compartment key or the sharing of code or data. The TrustNo1 architecture was not implemented or evaluated.

In the same theme of TrustNo 1, the eXecute Only Memory (XOM) [24] [5] [25] [26] architecture relies on an entire processor redesign to provide software security. XOM, much like Kuhn’s TrustNo 1, provided complete compartmentalization of applications using both
confidentiality and integrity for data and instructions. XOM also utilized tagged registers and tagged cachelines to prevent data/instructions from propagating between applications and, unlike TrustNo 1, provided support for shared address space. Support for shared address space was provided by XOMOS which was responsible for implementing XOM-specific instructions and moving information across compartments. XOM did however require that shared and dynamic libraries be left unsecured unless they were statically linked into the executable. A reduced version of XOM was formally verified using the Murphi model checker [24]. XOM was also evaluated using SimOS [27] and was shown to impact performance less than 5%, although these numbers did not account for the fact that keys are stored off-chip and must be retrieved from memory. Yang et. al [28] attempted to improve the performance numbers presented by the XOM architecture by using one-time-pad (OTP) encryption/decryption scheme. Along similar lines is the work performed by Shi et. al. [29], that attempts to decrease the impact of strong encryption in the memory hierarchy using counter mode encryption. The performance, like that of XOM, was hampered by the storage of keys off-chip.

The next progression in using hardware for software security was AEGIS [30] [6] [31] [32]. Similar to previously discussed architectures, AEGIS provided software security by redesigning a standard processor architecture to include security features. AEGIS went beyond XOM and other architectures by providing integrity checking and context management in addition to data and code confidentiality. Much like XOM, AEGIS also tagged registers and caches and used the tags to provide compartmentalization. Hashed values of context information stored in memory along with the protected code were checked for integrity using hardware-assisted hash trees. The AEGIS architecture provided two modes of operation, tamper-evident and private tamper-resistant. The tamper-evident mode did not provide privacy for code or data but did provide tamper resistance for code, data, and process state. This form of process state protection provided the ability to protect the integrity of application flow once the application had begun execution. AEGIS also provided some tamper resistance in the form of a process specific secret derived from properties of the chip. Unlike XOM, AEGIS relied upon a micro-kernel architecture to provide security functionality. The AEGIS architecture was simulated using the SimpleScalar [33] simulation framework that indicated performance degradations in the range of 20% to 30%. AEGIS was not evaluated in hardware.

In an effort to remove some of the burden of secure processing from the processor architecture, other avenues of using hardware for software protection have been explored including the use of secure kernel/micro-kernels, small modifications to processor subcomponents, or dedicated secure coprocessors.

Dyad [34] [35] was one of the first architecture to rely on a secure micro-kernel. Dyad examined the necessary protocols and procedures for networked hosts utilizing secure coprocessors. Specific applications of interest were audit trails, electronic transactions, and copy protection. The secure coprocessor was responsible for providing a secure compartment for execution and security functionality not available in the OS. The secure coprocessor was not
responsible for enforcing security policies because the OS was booted securely and verified using the coprocessor. Keys and other secret material were assumed to be stored exclusively on the coprocessor’s internal, non-volatile memory. The Dyad architecture was neither simulated nor implemented.

The Cerium [36] architecture created security by utilizing a tamper resistant CPU in coordination with a micro-kernel. The micro-kernel, much like the secure kernel used in Dyad, was responsible for providing process separation and was specifically responsible for address space separation, protection during cache-evicts, cryptographic processes (authentication, encryption/decryption, etc), and certificate creation/signing. The Cerium architecture was neither simulated or implemented. Similarly, the ChipLock [37] architecture relied on a secure micro-kernel, in addition to processor modifications, to provide integrity and confidentiality for both instructions and data. Unlike Cerium however, ChipLock placed security functions in hardware and not in the micro-kernel.

Another security architecture proposed by Gilmont et. al [38] [39] modified traditional, architectural memory management structures, included a hardware cipher unit, a permanent memory for storing keys, appended TLB, appended page registers, and a L1 cache interface to allow for prefetching. This architecture moves security farther outside of the processor but still resides within the processor boundary. Gilmont’s MMU security architecture fails to address the protection of data and ignores issues related to operation in a multitasking environment such as shared libraries and different security levels. The architecture was evaluated using simulation and showed overhead costs between 2.5% and 6%.

An important aspect of hardware-based security solutions that modify or redesign processor internals is the effect of adding the additional security hardware to the processor. The additional hardware consumes logic resources that would otherwise be used for cache or memory which are essential to performance. Also, the additional security hardware might also impact timing and other requirements. Together the loss of logic resources and possible timing impact could severely hinder performance. These critical system issues have not been addressed in the design and characterization of existing processor redesign software security solutions.

Moving further outside of the processor, the secure execution protection framework (SPEF) developed by [40] [41] attempts to prevent execution of unauthorized code by creating a processor unique version of the executable via several transformation methods. Hardware between memory and the processor transforms the executable back to the original version. The processor is required to contain a unique processor identification secret.

Other recent efforts at placing security features at the chip boundary include the SafeOps architecture by Zambreno et. al. [42] [43] and CODESEAL [44] developed by Gelbart et. al. The SafeOps architecture attempted to protect software integrity using a combined compiler and FPGA approach. The compiler examined the executable for sequences of register usage that were used to check for integrity and consistency. Additionally, the compiler could embed several non-essential instructions to add to the register usage patterns and perform
opcode obfuscation. At execution time, instructions were pushed through the FPGA hardware which compared the register sequence to the register sequence established in the static analysis. If discrepancies were detected the FPGA hardware halted the processor. SafeOps was evaluated using SimpleScalar [33] which showed average performance degradation to be around 20%. The CODESEAL architecture extended the SafeOps architecture to include instruction decryption and hashing of blocks of information. This architecture was also validated using SimpleScalar and demonstrated performance degradation between 4% and 42%, depending on the type of protections incorporated and the benchmark applications.

(Dedicated) Secure Coprocessors

The final category of hardware solutions to software security is secure coprocessors. Secure coprocessors are generically defined as a computing device contained within a tamper-proof box that utilizes secure protocols to communicate and exchange information with the outside world. One of the first individuals to examine challenges facing widespread utilization of secure coprocessors was Bennett Yee [45] who explored applications of distributed machines that contain secure coprocessors. Yee utilized properties of the secure coprocessors to perform a variety of tasks and enable applications including secure booting, copy protection, audit trails, and electronic commerce. The Dyad architecture [34] [35] discussed earlier was created in conjunction with Yee’s work. Similar work was performed in [46] which explicitly explored software licensing schemes using secure coprocessors in the form of smartcards. Secure coprocessors made their way onto the commercial domain with the IBM 4785 secure coprocessor which underwent several certifications [47] [48].

Software Solutions

Moving away from hardware-based software protection solutions one finds software methods for protecting software. While using software to secure software seems paradoxical, many of the proposed methods either rely on a tamper resistant platform, a secure kernel, or merely aim to provide support to other hardware methods. The first of this class of architectures is Terra.

Terra [49] introduced the idea of trusted virtual machine monitors that compartmentalize applications or sets of applications using the virtual machine concept. Terra relied on the existence of a tamper resistant platform.

Obfuscation methods that attempt to complicate program analysis have also been explored. Wang et. al. [50] used a compiler implementation that produced obfuscated programs making static analysis of the program difficult. The obfuscated software is unobfuscated by the operating system by a runtime interpreter.

Duvarney et. al. [51] presented the idea of extending traditional executable formats, par-
particularly the Executable Linking Format (ELF) [52], to include security information. Their particular application appended standard ELFs with information that was used to perform runtime monitoring, static monitoring, and program transformation. This allowed security information to be added post compile if necessary.

Another platform under development in the commercial sector under the names of the Next Generation Computing Base (NGSCB) [53], Trusted Computing Platform Alliance, and Palladium utilizes a combination of hardware and software techniques to provide protection from software-based attacks. The NGSCB is not designed to prevent physical attacks on the local machine. NGSCB utilizes a small security coprocessor, called a trusted platform module, that is attached to the mother board to store and limit access to sensitive information such keys, certificates, etc. In cooperation with a secure portion of an operating system known as the nexus, the small security coprocessor assist in compartmentalizing the sensitive application from other system processes including the non-nexus portion of the operating system.

### 2.2.2 Execution Flow Protection

The following subsections outline varying solutions to execution flow integrity, execution flow confidentiality, and execution access in both hardware and software varieties.

The work performed in Goldreich et. al. [12] marked one of the first efforts that recognized theoretically that information leakage occurs in the form of execution flow information. The authors introduced the concept of obliviousness which states that a machine is oblivious if a sequence of instruction accesses are equivalent for any two inputs with the same running time. They show that the slowdown of an oblivious machine would at best be \( t \times \log_2 n \) where \( n \) is the number of memory locations. While it was a sufficient model for computing systems at the time of publication, Goldreich’s analysis is not necessarily applicable to modern computing system due to the addition of caches and operating systems which may inadvertently assist or hinder obliviousness.

### Hardware Solutions

The first class of hardware solutions for protecting execution flow aims to provide execution flow confidentiality.

Zhuang et. al. [54] utilized a fairly simple obfuscation mechanism, called a shuffle buffer, to complicate memory and address access monitoring as a method for gaining information about an executing application. The shuffle buffer is a circular memory of recently accessed memory blocks. When a memory block was accessed it replaced a block in the shuffle buffer and the information being replaced in the buffer was written back to memory. This process attempted to create a “random” memory access pattern. The tracking of memory blocks
throughout memory as a method for breaking the obfuscation method were not addressed.

Another similar architecture by Zhuang et. al. named HIDE [10] also examined the protection of instruction access sequences, likely obtained (leaked) by monitoring memory or disk accesses, using hardware. The HIDE architecture attempted to make instruction accesses independent of program execution using multilevel prefetch buffers in addition to a hot function set. As hot functions were identified and added to the set, the entire function block was prefetched and stored in the prefetch buffers. As the functions are requested, the prefetch buffers were accessed to retrieve the appropriate portions of the function. This required that the size of the function be added to the executable description such that the prefetch buffer hardware would know the appropriate amount of memory to allocate. Also the prefetch controller was required to encrypt and decrypt memory for writing blocks back that have been encrypted with a different key such that blocks cannot be tracked as they are moved within memory. Evaluation of the HIDE architecture method was performed using the SimpleScalar [33] simulation platform and the SPEC benchmarks [55]. Performance degradation was shown to be approximately 38%.

The next class of hardware solutions for protecting execution flow attempted to provide execution flow integrity. Generally speaking the approaches to this problem attempt to prevent code injection attacks or maintain signatures of the execution flow that can be compared to valid signatures to indicate errant flow.

Barrantes et. al. [56] sought to prevent code injection attacks (that could potentially alter execution flow) by implementing a form of program obfuscation called instruction set randomization. They proposed the production of machine-specific executable using a pseudorandom sequence loaded at runtime that is XORed with instructions by an interpreter that resides in the processes virtual address space. The method was evaluated using an emulator and demonstrated a 5% performance degradation.

Another approach for preventing invalid code injection was dynamic information flow tracking [57] that utilized the operating system to mark sources of information flow as possibly malevolent. The classification of information was used by a custom processor, that included architectural support at the TLB and registers, to tag information from these possibly malevolent sources. Tags propagated to data produced if the data consumed contained tags. The SimpleScalar [33] was used to evaluate the architecture which demonstrated modest performance degradation of around 5% and an approximate storage overhead of about 5%. The architecture does, however, require a custom processor and was not implemented.

The next two hardware architectures, basic block signatures [58] and runtime execution monitoring [59], used signature methods for verifying execution flow. Basic block signature verification attempted to prevent execution of unauthorized code by verifying runtime block signatures calculated by hardware with signatures created during the install process. The signatures were created during the install process using secret coefficients and the relative address of the basic block. Because this method verified the integrity of instruction sequences, access to the instruction pipeline hardware was necessary. The method was evaluated using
traces from the SPEC [55] benchmarks for an Alpha processor. Analysis of the method compared the number of misses in the basic block signature table to the number of instruction cache misses. The results indicated that if the number of sets in the basic block signature are sufficiently large, basic block signature misses are relatively small compared to the number of cache misses and should not contribute greatly to system performance degradation. Specific performance degradation numbers were not cited.

Runtime execution monitoring protected execution flow integrity by comparing pre-computed hashes of basic program blocks after they are loaded into L1 caches. If the hashes calculated on instructions and data passing through to the fetch and load store units did not match the original hashes an exception was raised. This method required the addition of hash compute and comparison hardware to the processor (behind caches) as well as a hash pointer instruction to the instruction set architecture (ISA). The concept was evaluated using the SimpleScalar simulation framework and SPEC2000 benchmarks. The additional hash information increased application size between 40\% and 100\% of the original. Performance reduction ranged from 6.4\% to 40\% depending on hash block size, cache sizes, and memory bandwidth consumed retrieving hashes from memory.

Software Solutions

Software solutions to execution flow are far more diverse than the hardware solutions above. One of the first and most basic methods for providing some level of execution flow integrity was the protection of the runtime stack from malicious modification as demonstrated in Stackghost [60]. Stackghost thwarted return address modification by XORing the return address with a unique “cookie” before placing it onto stack and XORing the value again when it is popped off the stack. Any modification to the XORed value will result in a completely errant return value. Similarly, the work performed in [61] utilized custom hardware added to the processor and slight OS modification to protect the runtime stack. Nearly a completely opposite approach to software-based execution flow integrity was Cyclone [62]. Cyclone was a “safe” version of C that attempted to eliminate common programming errors that cause violation execution flow integrity such as buffer overflows, and dangling pointers.

The approach taken by Ko et al. [63] towards execution flow integrity protection used high-level security specifications to scan execution traces for security violations. The actions of all processes on a host were merged into a single trace which is analyzed by a monitor application. Using a specification manager, this method was extended to operate in a distributed environment where traces containing multiple processes are analyzed on several hosts. The concept of parallel environment grammars is introduced as a method to enable real time parsing of the higher level security specifications in the monitoring application. Similarly, gray-box execution flow graph anamoly detection [64], performed dynamic analysis of programs in memory and searched for control flow transfer abberations.

Another unique approach to software execution flow protection is program shepherd-
ing [65] which utilized a dynamic optimizing interpreter and a security policy to prevent spurious execution flow transfers. This was done by forcing well defined function entry and exit points and verifying the source and destination of each control flow instruction such as a branch. Because all execution flow operations were monitored, any sandboxing instructions (wrappers around portions of code that perform some check) could not be bypassed. Performance was improved by caching blocks of code that have been examined using the security policy. Program shepherding was evaluated using the SPEC2000 benchmarks and indicated an average performance degradation around 16%. This method did not require modification to the executables or the operating system. In a similar fashion, the SafeOps architecture discussed previously in Section 2.2.1 provided some level of execution flow integrity by utilizing register transfer sequences as a flow enforcement mechanism.

The final method of providing execution flow protection are forms of obfuscation such as the Trusted flow [66] architecture and a process called watermarking [67]. TrustedFlow protected execution flow by embedding secret signatures in the original code. As the code executed these signatures along with other trace information are sent to another machine and are authenticated. Watermarking by Collberg et. al. [67] embedded a secret message into a cover message which are extracted and used to verify associated information. An evaluation tool for various obfuscation techniques can be found in [68]. Despite efforts in the area, some researchers assert that obfuscation cannot provide complete software protection [69]. With an understanding of existing hardware and software security architectures, methods for analyzing the security features of such systems can be discussed.

2.3 Formal Security Analysis

The goal of formal systems is to verify the operation of a subsystem, system, or interaction between a collection of systems using formal methods. Formal methods have generally fallen into two categories: theorem proving and model checking [70]. Several generally accepted rules-of-thumb relating to the use and application of formal methods can be found in [71]. In the security domain, formal methods have been utilized to support claims concerning security. One of the initial uses of formal methods in the security domain is cryptographic protocol analysis [72] [73], that uncovered an attack on the Needham-Schroeder exchange protocol [74]. Other variations of formal protocol analysis [75] [76] [77] examine abstract state machines and team automata as methods for examining cryptographic protocols. Lowe et. al. [78] made one of the first attempts at automating the protocol analysis process by creating a compiler called Casper which would generate process algebra descriptions automatically from a more abstract description. The analysis was completed by processing the algebras with a standard model checker.

Butler et. al. [79] examined the use of formal methods for validating and modeling trust. Another area of interest is secure systems. With the arrival of the secure coprocessor as discussed in Section 2.2.1, much effort was focused on formal verification of these systems
such as in [80]. A practical example of formal security analysis of a secure coprocessor, specifically the Infineon SLE88, can be found in [81]. The analysis utilized a combination of model checking and theorem proving, called interacting state machines, that was created and applied to the Infineon SLE88 [82] [83] [84] [85].

Recently, several custom architectures, discussed in detail in Section 2.2, have been proposed as methods to provide security functionality. These systems, like other hardware and/or software [86] are subject to functional verification and are also subject to security verification. The security features of the XOM architecture [24] [87] were analyzed and verified using model checking on a scaled-down model and was one of the few examples of applying formal methods to security hardware. The use of formal methods to verify secure systems is discussed in more detail in Chapter 3.

### 2.3.1 Security Models

Another component of secure systems are security models. Security models provide an abstract view of how information flows between entities and can generally be applied to nearly any system.

The work of Bell-Lapadula [88], Biba [89], and Clark-Wilson [90] comprise the foundation upon which almost all other work in the area of formal security models is based. Bell and Lapadula were the first to rigorously analyze multi-level security in computing systems using mathematics. From their analysis, Bell-Lapadula was able to provide a basic set of rules that must be enforced to maintain multi-level security, namely the simple-security property (ss-property) and the *-property. The ss-property states that a subject of a particular level of classification can only read information at an equal or lesser classification level (read-up). The *-property states that a subject can only write information at an equal or higher classification level (write-down). While the particulars of the system structure are more rigorously defined by Bell-Lapadula, the security of the system relies on these basic properties.

The Bell-Lapadula model was designed for data confidentiality and did not consider data integrity. Biba [89] attempted to address this issue by creating a model quite similar to Bell-Lapadula for data integrity. Biba’s integrity model relies on two rules to provide data integrity. The first rule prohibits the writing of information from a lower classification to a higher classification. The second rule prevents the reading of data from a lower classification by a subject at a higher classification. The second rule prevents the reading of data from a lower classification by a subject at a higher classification.

Clark and Wilson [90] extended the work of Biba to utilize separation of duties. The Clark-Wilson integrity model is based on transactions that transition a system from one consistent state to the next. To achieve this, the Clark-Wilson model decomposed information into two categories: constrained and unconstrained. Constrained data, unlike unconstrained data, had to adhere to a specific set of rules. Integrity verification procedures tested data for compliance to the associated rules. If successful, transformation procedures were applied to
complete the transaction and update the system to the next state. In addition to modeling information flow in commercial settings which are generally more concerned about information integrity over information confidentiality, Clark-Wilson also differentiated between certification and enforcement.
Chapter 3

Information-Flow Hardware Security Analysis

This chapter presents the concept and demonstration of a novel graph-based modeling method, that applies concepts from information-flow analysis, to hardware-based secure computing systems. Section 3.1 describes the process and difficulties associated with existing formal methods. Examples of theorem proving and model checking are also provided. Section 3.2 discusses the creation of information flows, their relationship with security hardware, and desirable properties of a system model for use in security analysis. Finally, Section 3.3 outlines the proposed method and demonstrates its application on two existing architectures XOM [5] and AEGIS [6].

3.1 Formal Methods and Security Analysis

An important aspect of designing a secure system is the ability to support claims of security. Similar to functional system verification, security verification can occur at several different levels ranging from an abstract specification to a full implementation. Currently, most security verification such as [24] [84] [73] is performed using formal and informal methods developed for verifying system functionality. Independent of the type of analysis method utilized, the process of verifying security features of a system typically requires three steps [91].

1. Specify the property to be verified: The property or properties that require verification must be established and precisely defined.

2. Describe the system: Effective system analysis requires that a system be modeled at the appropriate level of abstraction.
Select an analysis method: An analysis method must be appropriately selected given the system description the properties to be verified.

(a) If required by the selected analysis method, the property specifications must be converted to a method-specific description.

(b) If required by the selected analysis method, the system descriptions must be converted to a method-required representation.

The first step, selection of the property to be verified, usually results in a property specified in English such as “Bob must never obtain Alice’s private key”. After the system is described and the analysis method selected, the English specification is transformed using knowledge of the system description and the semantics provided by the analysis method. The second and arguably most difficult step, describing the system, requires that the system be represented at the appropriate level of abstraction. A complex description with too little abstraction will render the analysis impossible or overly complex. In contrast, an overly abstracted design may negate the security verification process by removing features necessary for the analysis. The system description step is further complicated by the requirements of the selected analysis method. If the analysis method requires a particular system description format such as a finite state machine (FSM), the abstracted model created in step two must be converted, regenerated, or described natively. The difficulty of the analysis itself given the chosen method depends heavily upon the property selected for verification and the description of the system. Different properties may, but not necessarily, require different combinations of system descriptions and analysis methods. Traditionally, formal methods such as those discussed in Section 2.3 have been utilized to verify both general system operation and security properties of systems ranging from protocol exchanges to operating system access policies. The two formal methods primarily used in security analysis are automated theorem proving and model checking.

3.1.1 Theorem Proving

Theorem proving attempts to reason (prove or disprove) about a system and its properties which are both described in mathematical logic [92]. Assuming proper system description and property selection, the results of theorem proving can allow for very concrete claims about the system’s functionality [92]. Although theorem proving can be quite powerful, it is not without its difficulties. Theorem proving requires careful abstraction of the system into a form that can be described in mathematical logic [93]. While claims resulting from theorem proving are very concrete, over abstraction can eliminate too much detail rendering the claims invalid. Also, modifying the model to represent a system modification may be quite difficult and/or time consuming. Once the logical system description is generated, the process of performing the proof is usually semi-automated as discussed in Section 2.3. The level of automation depends heavily on the type of logic used and requires a varying amount
of user interaction. In the security domain, theorem proving has been primarily applied to cryptographic protocol exchanges. An example of theorem proving applied to an AND gate is presented in Appendix A [94] [95].

### 3.1.2 Model Checking

Model checking, in contrast to theorem proving, attempts to verify functionality and/or security properties of a system through exhaustive system state inspection and requires that the system be described as one or more finite automata [96]. Since most computing devices can be described as state machines, model checking maps well to the domain of digital design. Unfortunately, the usefulness of model checking can be diminished by the state-space explosion phenomenon resulting from an overly detailed system model. Several different methods such as randomized state searching and symbolic model checking have been devised to increase the number of state that can be successfully searched. Model checking has been primarily used to verify functional operation of various systems and perform circuit equivalence tests. In the security domain, model checking has seen very limited use and was recently applied to a reduced version XOM secure processor [24] to uncover a replay attack. An example of model checking applied to basic mutual exclusion between two processes is presented in Appendix B [97] [98].

As mentioned earlier in Section 2.3 recent work has centered around the creation of a hybrid method that generates logic-based system descriptions from finite state machine representations of the system. The generated logical system descriptions and logically defined security properties are then processed using an automated theorem prover. This method was successfully applied to portions of the Infineon SLE88 secure coprocessor [81].

The application of theorem proving and model checking to security analysis, particularly in the area of security hardware, has seen limited success. Although theorem proving and model checking are general methods that can be applied to nearly any system, the major difficulty with their use is the process of developing appropriate system models. As a result, the key to successfully analyzing security properties of hardware-based security systems lies in developing a better method of describing and modeling such systems.

### 3.2 Hardware - Software Interaction and Information Flow Models

Security in computing systems revolves almost entirely around the problem of restricting or preventing the flow of information between entities. Security models discussed in Section 2.3.1 such as Bell-Lapadula, attempt to formalize information flow by defining generic rules for preventing unwanted information flow within a constrained scenario. In the software
domain, the concepts of information flow [7] are used to examine the flow of information between variables primarily to find control-flow dependencies that reveal information about the contents of specific variables [8] [9]. Information flow analysis of software, particularly Java, has seen some success in JFlow [99]. The success of software information flow analysis coupled with the fact that hardware and software interact to create information flow hints at the promise of applying similar concepts to the hardware domain. In order to effectively apply information flow concepts to the hardware domain, specifically hardware-based secure computing platforms, the creation of information flows via hardware-software interaction must be understood.

Modern computing systems can generally be decomposed into hardware and software components. Beyond the initial decomposition, software is further divided into functionally distinct elements, executable instructions and data. Instructions are responsible for creating and orchestrating data and control flow. The stream of instructions is generally initialized and maintained by hardware, however software is tasked with the responsibility of differentiating instruction flows by applying context, usually via an operating system. By using cryptographic and other techniques, hardware-based security architectures attempt to effectively eliminate or minimize information flow between hardware components, software components, and other external entities. Desirable properties for a security-based hardware system model include the following

1. **Ease of application**: A useful system model for security analysis should be easy to develop from an implementation or specification.

2. **Capture architectural details**: A useful system model for security analysis should be capable of capturing necessary architectural details for the particular type of analysis in question.

3. **Reasonable state space**: A smaller state space is necessary to reduce computational complexity if state space searching analysis methods such as model checking are to be used.

4. **Analysis method compatibility**: A useful model should allow analysis using a variety of techniques.

In an effort to meet these needs, it is proposed that hardware-based security architectures can be generically modeled and effectively analyzed using information flows.
3.3 Hardware Security Analysis and Information-Flow Graphs

The fundamental concept behind applying information flow to the analysis of security hardware is the representation of the system as a directed graph \( D(V, E) \) where information flows are denoted by the set of edges \( E \) and hardware entities are denoted by the set of vertices \( V \). In this system model, vertices of the graph represent portions of the hardware that store or manipulate information flows. Edges in the graph, which represent information flows, are created by the interaction of software in the form of instructions with the hardware elements.

Using a standard model for the current computing paradigm of a processor with caches, register storage elements, and memory, a graph depicting basic information flows can be constructed as shown in Figure 3.1. As depicted in Figure 3.1, a load operation on a standard processor causes information to flow from memory (PROC11), through the data cache (DC1), and into the appropriate register (R1). To model a complete system, each instruction in the processor’s instruction set architecture (ISA) must be translated into flow operations. Because hardware is by definition static and a finite set of instructions exist that interoperate with hardware to create information flows, it is possible to create an exhaustive graph \( R \) that represents the set of all possible information flows through the system given a particular ISA.

While any one of the previously discussed analysis methods could be applied to this system model if appropriate, some features can be directly analyzed using far more simplistic techniques. As mentioned earlier, an important aspect of security modeling is the ability to determine if information flows from one entity to another. The simple property of information flow (path) between two nodes \( i \) and \( j \) in a graph-based system model is generalized as a sequence of vertices \( \{v_0, v_1, \ldots, v_k\} \) with edges \( E = \{e_{0,1}, e_{1,2}, \ldots, e_{k-1,k}\} \) \[100\].

Using the exhaustive graph \( R \), the problem of checking Rule 3.1 (path existence) can be generalized as the transitive closure problem from graph theory \[101\]. The transitive closure calculation creates the closure graph \( K(V, E') \) where \( \forall u, v \in V \), \( (u, v) \in E' \) iff a directed path exists between \( u \) and \( v \in R \). The transitive closure operation is computationally feasible when performed using the Floyd-Warshall algorithm which has a computational complexity of \( O(n^3) \) where \( n \) is the number of vertices \[102\].

\[ \exists P_{i,j} \in R \tag{3.1} \]

While it is important to include and analyze absolute flow violations, it is also necessary

\[ \footnote{Note: While information flows could be assigned various attributes and/or weights, information flows should not be confused with network flows. All references to a flow or flows concern information flows not network flows.} \]

\[ \footnote{Not all edges are shown for the sake of brevity.} \]
Figure 3.1: Example information flow graph for a standard computing system architecture
Figure 3.2: The multigraph resulting from applying a set of functions at an vertex to an incoming edge

to consider that the properties of a flow may change due to manipulation by hardware. Given a path \( P \in R \) let the function \( F_\ast : P \rightarrow S \) where \( S \) is a set of edges. Also, for a vertex \( q \) and set of edges \( S \), let \( F_q \) be the set of available functions at \( q \) such that \( \forall f \in F_q, f : S \rightarrow S' \). Also, \( F_q \) is not necessarily applicable to all permutations of adjacent edges.

For a path \( P \) with vertices \( V = \{v_0, v_1 \ldots v_{i-1}, v_i\} \) for \( 0 \leq i \leq k \) with \( k = |V| \), and an initial value \( W \)

\[
F_\ast(P) = F_k \circ F_{k-1} \cdots \circ F_1(W). \tag{3.2}
\]

\( P \) need not necessarily be simple and can have repeated vertices and edges. However, cycles in \( P \) cannot be infinitely traversed. All distinct paths in \( R \) are searched.

Given two function sets \( F \) and \( G \) respectively, the composition \( F \circ G \) is defined as

\[
F \circ G = \{f_0(g_0) \cup f_0(g_1) \cdots \cup f_0(g_i) \cdots
\cup f_1(g_0) \cup f_1(g_1) \cdots \cup f_i(g_j) : f \in F, g \in G, k = |G|, n = |F|, 0 \leq i \leq k, 0 \leq j \leq n\} \tag{3.3}
\]

Let \( f^{-1} \) be the inverse of function \( f \) such that the composition \( f^{-1} \circ f \) yields

\[
f^{-1}(f) = I. \tag{3.4}
\]

Let \( f_\emptyset \) be the null function such that

\[
f_\emptyset(I) = I \tag{3.5}
\]

The affect of applying of \( F_\ast \) to a single vertex path is shown in Figure 3.2. Similarly, the affect of applying \( F_\ast \) to a path with multiple vertices is shown in Figure 3.3.

Applying context to the model, let the available functions be encryption and decryption functions \( t \) and \( d \), respectively, such that

\[
d = t^{-1}(). \tag{3.6}
\]
In using the encryption and decryption functions, it may be possible for instruction operations to create a flow of information (path) that results in the unauthorized decryption of information to plaintext.

Let $P$ be the path under scrutiny. The path $P$ is a violation if

$$\exists I \in F_s(P)$$

(3.7)

The violation rule shown in Equation 3.7 captures all paths that result in unencrypted information. In analyzing actual systems, this rule is too restrictive since some systems elements may be authorized to contain or process unencrypted flow. To include the concept of authorization in the model, a logical attribute (or label) is added to each node to denote access to unencrypted flow. Given a vertex $i$, let the $\varphi(i) = true$ denote that the vertex $i$ is authorized to access or process unencrypted information.

The violation rule must be modified correspondingly such that the path $P_{i,j}$ is a violation if:

$$(\exists I \in F_s(P_{i,j})) \land (\neg \varphi(j))$$

(3.8)

Given the design of computing systems for time-multiplexing of resources, the context of the system operation must also be considered when examining information flows for violations. Typically, system context in modern multitasking computing platforms is managed by an operating system, however the addition of certain hardware features may allow for hardware differentiated contexts. Within a particular hardware operating context, certain hardware operations may be enabled or inhibited. An important distinction when discussing hardware and software contexts is that the operating system contexts are not necessarily equivalent to hardware contexts. If software (i.e. the operating system) is allowed to control the hardware context, then the two contexts will be equivalent, however, it is also possible for the hardware to derive its operating context given some other state or property of the system. The ability to maintain context separation is an important feature for secure computing platforms and can be analyzed by extending the system model.

Because hardware is static and the ISA interacting with the hardware is fixed, multiple hardware contexts are modeled by adding duplicate graphs of the single-context exhaustive reachability graph $R$. Consider a supergraph $C(V_c, E_c)$, where each vertex $t \in V$ contains
a subgraph $R$, the previously defined exhaustive reachability graph. Each edge $w \in E_c$ connects the corresponding nodes $u_1$ and $u_2$ between subgraph $R_1$ and $R_2$. An edge between corresponding nodes in two separate contexts indicates that the information residing at the node (information flow state) is unmodified in the event of a context switch. System elements are marked as context stable if information at a node is unmodified in the event of a context switch. If an element is context stable, edges between corresponding nodes in each context are added to the initial graph. Let the function $\tau(J)$ denote the context stability of the architectural feature/subfeature $J$ such that if $\tau(J) = true$, $J$ is context stable. Also, let the function $c(v)$ indicate the context of the vertex and the function $C_{num}(C)$ denote the number of context subgraphs in $C$. Operations on the graph defined by the ISA may remove these features.

An example of an architecture feature that is usually context stable is a cache. Because modern systems are designed for resource sharing, caches generally maintain state across context switches. As a result, information from one context can flow to another context via the caches on a context switch. If a cache flush were performed on a context switch, then edges connecting the caches in the different contexts would be removed. The addition of context information to the model allows further refinement of the violation definition given a path $P$ with source and destination vertices $i$ and $j$ as

$$\exists I \in F_{*}(P_{i,j}) \land ((\neg \varphi(j) \lor (c(i) \neq c(j))))$$ \hspace{1cm} (3.9)

Let $W$ be the set of allowable edge values. The violation rule shown in Equation 3.9 can be further generalized as

$$((F_{*}(P_{i,j}) \not\subseteq W) \land ((\neg \varphi(j) \lor (c(i) \neq c(j))))$$ \hspace{1cm} (3.10)

Another important aspect of analyzing information flows is the differentiation of explicit and implicit flows. Explicit information flows result from the direct flow of information between two nodes such as that caused by a read operation. In contrast, implicit flows result when flows are modified and/or combined by some operation such as an add. Adding this feature to the existing model allows flows to be characterized as a weak or strong violation. For an edge $e$ let the function $w(e) = true$ if the edge is implicit. A violating path $P$ is said to be a strong violation if

$$\forall e \in P, w(e) \neq true$$ \hspace{1cm} (3.11)

A weak violation indicates that information is leaked rather than completely divulged and may be inferred if enough information is available from other parts of the system.
3.4 Analysis Process & Examples

To demonstrate the utility of the model developed in this chapter, two existing architectures XOM and AEGIS are modeled and analyzed. In addition to demonstrating the application of the model, the analysis reveals a possible attacks on both XOM and AEGIS. It should be noted that while possible attacks were found, the attacks may have been ruled out by usage assumptions or scope of XOM or AEGIS. Analysis of an architecture using the presented model requires the following steps:

1. Convert the system under test into graph vertices
   (a) Specify primary architectural features and attributes
   (b) Specify subfeatures and their memberships in the primary architecture features
      i. Specify context stability node attribute
      ii. Specify plaintext allowed node attribute
      iii. Specify other attributes of the subfeatures such as initial encryption values, key sets, etc.

2. Define each instruction in the ISA as a operation on the graph (add/remove flows, etc.)

3. Specify the instructions to “execute”

4. Analyze resulting graph for violations

3.4.1 XOM

The XOM [5] architecture provides security features in the form of application compartments by extending an existing processor design to include register tags, cache tags, symmetric cryptography, and an extended instruction set. A simplified depiction of the XOM architecture is shown in Figure 3.4. In an effort to explore the performance reduction caused by the added security features the XOM architecture was simulated in [24] using SimOS [27] with the MIPS R10000 [103] processor architecture. The creators of XOM were one of the first to apply formal methods to security analysis of hardware devices [24]. In applying a state model, Suh et. al. discovered a replay attack that affected their integrity checking methodology.

The application of the information flow model to the XOM architecture (shown in Figure 3.4) begins by defining the architecture as a set of vertices. In the instruction-based computing paradigm most architecture models will, at a minimum, consist of memory, caches, and registers. The addition of hardware-based security capabilities requires further representation. XOM compartments are represented in the model as a hardware context. In this instance of the model, three contexts will be used to represent two protected applications
and the XOM shared compartment. The memory feature contains three subfeatures corresponding to the physical memory spaces of each of the three XOM compartments. The nodes of each subfeature are marked as containing a corresponding initial flow $F_i$ where $i$ denotes the subfeature index. The L1 instruction and data caches each contain a single subfeature and are marked as not context stable. Due to the cache tagging mechanism employed in the XOM architecture, information cannot flow between two contexts via the instruction and data caches. Because the same tagging mechanism is employed in the unified L2 cache, the L2 cache portion of the architecture is an extension of the first level and was not included in the model. Since the registers also employ tagging, they too are marked as NOT context stable. The location of the symmetric encryption unit outside of the caches indicates the plaintext is allowed in the L1/L2 caches, and registers. The symmetric encryption unit was specified with two keys for each compartment, the compartment encryption key and the shared compartment key.

**XOM Model**

Let $X(V, E)$ be the graph model for the XOM architecture with subgraphs $M, I, D, R, S \in X$ denoting the memory, instruction cache, data cache, registers, and symmetric encryption primary features, respectively. Also let $k_i$ and $b_i$ denote the allowable key set in the $i$th context for the encryption and decryption functions, respectively. Finally, let the key value zerod, denote the null key (i.e no encryption or decryption applied).
The model of the XOM architecture shown in Table 3.4.1 is an example of how a designer can apply the modeling method discussed in this chapter to a security architecture. Two important aspects of creating an architecture model are choosing the values of the $\varphi()$ and $\tau()$ functions. In doing so, it is necessary to keep the function of hardware in mind. For example, memory ($M1I$, $M1D$, $M2I$, $M2D$, $M3I$, $M3D$) is by definition stable across contexts. In the XOM architecture, both the instruction side and data side caches ($IC$ and $DC$) contain additional tags that denote the XOM compartment associated with a particular cache line. In the case of a (hardware) context switch to another compartment, data from either cache cannot be used in another compartment unless the compartment tags match. In the mathematical model, the inability for data to flow between compartments via the tagged caches is represented by marking the instruction and data cache as context instable ($\tau(IC), \tau(DC) = false$). If the caches were standard caches without the XOM tags, the caches would have been marked as context stable. In setting the plaintext allowed attribute, the designer can mark the portions of the architectures are allowed to process/contact plaintext information such that the architecture can be analyzed using Rule 3.10. In the case of the XOM architecture, plaintext information can only reside in the shared compartment and the symmetric cryptographic unit (represented as $M3$ and $S1$ respectively) where $\varphi(M3I), \varphi(M3D), \varphi(S1) = true$. The three contexts that are modeled represent two protected XOM compartments and the shared XOM compartment.

### XOM Analysis

The ISA for the MIPS R10000 processor [103] along with the ISA extension defined by XOM were modeled and applied to the graph. For the sake of brevity and clarity, the graph includes only the necessary set of instructions:

1. **load**: A load operation creates flows from $M \rightarrow DC \rightarrow R$. 

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$M1I \in M$</td>
</tr>
<tr>
<td>2</td>
<td>$M1D \in M$</td>
</tr>
<tr>
<td>3</td>
<td>$M2I \in M$</td>
</tr>
<tr>
<td>4</td>
<td>$M2D \in M$</td>
</tr>
<tr>
<td>5</td>
<td>$M3I \in M$</td>
</tr>
<tr>
<td>6</td>
<td>$M3D \in M$</td>
</tr>
<tr>
<td>7</td>
<td>$IC \in IC$</td>
</tr>
<tr>
<td>8</td>
<td>$DC \in DC$</td>
</tr>
<tr>
<td>9</td>
<td>$</td>
</tr>
</tbody>
</table>

### Table 3.4.1 Model of the XOM architecture

The model of the XOM architecture shown in Table 3.4.1 is an example of how a designer can apply the modeling method discussed in this chapter to a security architecture. Two important aspects of creating an architecture model are choosing the values of the $\varphi()$ and $\tau()$ functions. In doing so, it is necessary to keep the function of hardware in mind. For example, memory ($M1I$, $M1D$, $M2I$, $M2D$, $M3I$, $M3D$) is by definition stable across contexts. In the XOM architecture, both the instruction side and data side caches ($IC$ and $DC$) contain additional tags that denote the XOM compartment associated with a particular cache line. In the case of a (hardware) context switch to another compartment, data from either cache cannot be used in another compartment unless the compartment tags match. In the mathematical model, the inability for data to flow between compartments via the tagged caches is represented by marking the instruction and data cache as context instable ($\tau(IC), \tau(DC) = false$). If the caches were standard caches without the XOM tags, the caches would have been marked as context stable. In setting the plaintext allowed attribute, the designer can mark the portions of the architectures are allowed to process/contact plaintext information such that the architecture can be analyzed using Rule 3.10. In the case of the XOM architecture, plaintext information can only reside in the shared compartment and the symmetric cryptographic unit (represented as $M3$ and $S1$ respectively) where $\varphi(M3I), \varphi(M3D), \varphi(S1) = true$. The three contexts that are modeled represent two protected XOM compartments and the shared XOM compartment.
2. instruction fetch: A fetch operation creates flows from $M \rightarrow IC \rightarrow R$.

3. store: A store operation creates flows from $R \rightarrow DC \rightarrow M$.

4. move\_to\_shared: The move\_to\_shared XOM operation allows the use of the shared compartment key in the encryption function.

5. move\_from\_shared: The move\_to\_shared XOM operation allows the use of the shared compartment key in the decryption function.

The result of the ISA application and analysis is shown in Figure 3.5. Given the lack of violations (denoted by red dashed lines) in Figure 3.5, the XOM architecture is secure under Rule 3.10. If the model is analyzed using the more restrictive Rule 3.7, the types of possible violations can be examined as shown in Figure 3.6. The violations highlighted in red show that both instruction fetch paths and the data load paths can downgraded to the shared compartment. While the XOM architecture may assume a correctly written program, it is possible for both instructions and data to flow to an unauthorized and unencrypted state. This violation results from the ability to share data via the availability of the move\_to\_shared and move\_from\_shared instructions. The possible downgrade of instructions however, is a consequence resulting from the use of a single compartment key for both instructions and data. Also, the use of a single key for both instructions and data allows data to be executed. The attack on instruction confidentiality was not discovered by the formal analysis performed in [24] due to the focus on integrity checking scheme and construction using a system state model. If the shared compartment or the move\_to\_shared instruction is removed, the XOM analysis using the model indicates that the architecture is secure.

3.4.2 AEGIS

The AEGIS architecture is similar to the XOM architecture in overall structure and use of symmetric encryption beyond the L2 cache boundary. Unlike XOM, AEGIS relies on a software-based security kernel to provide certain security features in lieu of hardware-based security support such as cache/register tags found in XOM. The AEGIS architecture also differs from XOM in that the OpenRisc [104] processor was used as a base processor for the design. Other AEGIS features include an integrity verification method using hash trees that removes the replay attack found in XOM, and the generation of platform-specific keys using physical unclonable functions (PUFs).

AEGIS Model

The AEGIS model is shown Table 3.4.2. In this particular model, the software is not trusted. As a result, the caches and registers are marked as context stable. Also, we assume for now that AEGIS is capable of differentiating contexts and uses different keys for each context.
AEGIS Analysis

The graph shown in Figure 3.7 indicates violations under the condition that software trust is not allowed. The violations result from the use of a security kernel for clearing/saving register contents and for preventing unauthorized access to cache contents via a virtual memory manager (VMM). If the security kernel is assumed to be error free and/or not
Figure 3.6: Information flow diagram for the XOM architecture evaluated under more restrictive Rule 3.7 that doesn’t allow any unencrypted flows. This graph highlights the unintended consequence for instructions to be downgraded to the shared compartment due to the use of a single compartment key.
Figure 3.7: Information flow diagram for the AEGIS architecture evaluated using Rule 3.9 without software trust (security kernel)

susceptible to malicious modification, the violations in Figure 3.7 cannot occur. Figure 3.7 indicates the possibility of improper information flow for the given architectural structure, all available instructions, and configuration of security functions.

While the model presented in this chapter is primarily for examining software trust and confidentiality properties of hardware-based architectures, the model can also be used to analyze systems that trust software. If several parameters of the AEGIS model in Table 3.4.2 are modified to reflect software trust, the AEGIS architecture can be analyzed for security under its intended assumptions of trusting a security kernel. Assuming the AEGIS security kernel is implemented correctly, the contents of registers are appropriately saved/cleared between contexts and the access to data residing in caches is controlled by the virtual memory manager. The additional protection features provided by the security kernel are represented in the modified model by changing the context stability attribute for the registers and caches. Figure 3.8 shows that under the assumption of a working security kernel, the AEGIS architecture is secure.

Another aspect of the AEGIS architecture that can be modeled is the use of a single key for cryptographic operations. Modifying the model once again such that each context uses the same key and performing the analysis yields the graph shown in Figure 3.9. This graph shows that under the assumption of trusting the security kernel, the AEGIS architecture is
Figure 3.8: Information flow diagram for the AEGIS architecture evaluated using Rule 3.9 under the assumption of software trust (security kernel)
secure even when a single key is used.

The final aspect of the AEGIS system that can be modeled is the ability to downgrade data from a security level $K$ to a security level $L$ where $K > L$. To model the declassification method used by AEGIS, the key set definitions can be changed to the use only the null key 0.

Further analysis shows that similar to XOM, AEGIS instructions can be downgraded and exposed due to the use of a single key for both instructions and data as shown in Figure 3.7. The downgrade of instructions can occur if the application is erroneous, is written maliciously, or if the integrity checking features are circumvented.

### 3.5 Summary

Graph-based modeling of information flows as described in this chapter provides a generic method for representing hardware-based security architectures, addressing one of the major problems in the security analysis of such systems. The model is easily applicable and enables security analysis. Using the model, the confidentiality and software trust characteristics of specific hardware-based security systems was analyzed.
Chapter 4

The Secure Software Platform

This chapter describes in detail the hardware-based software protection architecture presented in this document. Section 4.1 provides an overview of architecture and its characteristics. Section 4.2 presents a high-level discussion of design goals and details the important questions addressed by this research. Sections 4.3 and 4.4 describe the architectural details and design decisions associated with each protection feature.

4.1 Architecture Overview

The Secure Software Platform differs in several ways from existing software protection solutions, such as those presented in Chapter 2. First, unlike the processor redesign and software-based methods of earlier architectures, the SSP adds security features to standard microprocessors by externally augmenting the processor with custom hardware placed between the processor boundary and external interfaces as shown in Figure 4.1. Second, the design of the SSP strives to remove trust from all software, namely the operating system and applications. Third, the SSP provides greater operational flexibility without sacrificing security by supporting multiple keys per application and shared libraries. Finally, the SSP has little impact upon the existing software development process. The SSP achieves its operational and security goals through a variety of software protection features including instruction confidentiality, data confidentiality, instruction integrity, and data integrity.

4.1.1 Security Model & Assumptions

Before the design of the architecture is discussed in detail, the system security model and associated assumptions must be addressed. The primary security goal of the work described in this document is to prevent unauthorized distribution and unauthorized operation of
software. The previous attempts at software protection discussed in Chapter 2 provide not only a range of possible protection features, but also describe many of the attacks against software. During the design of the SSP, the attacks considered span both software and hardware attacks including, but are not necessarily limited to, the following:

1. Software Attacks
   
   Improper coding: Poor coding practices can lead to attacks such as buffer overflows.
   
   Malicious coding: A malicious entity at the development stage may insert code to perform a malicious function.
   
   Malicious applications: Applications may attempt to gain information about other applications residing on the same platform.
   
   Malicious operating system: An operating system might be compromised or modified to aid an attacker.

2. Hardware Attacks

   Interface Probing: An attacker can physically monitor external address and data busses.
   
   Power Attacks: An attacker may monitor power usage in an attempt to determine internal system state.
   
   Timing Attacks: An attacker can modify external clocks or observes timing aspects of the system to extract information.
   
   Memory modification: An attacker may modify external memory maliciously to alter or manipulate system behavior for analysis purposes.
4.1.2 Assumptions

Another important aspect of designing secure systems is clearly defining assumptions associated with security claims. The assumptions listed in this section are applied throughout the design and implementation process.

The first assumption of the SSP is that the processor and custom security hardware reside in a tamper-resistant package and are considered to be trusted entities. Tamper resistance implies that it is infeasible to physically disassemble the package to access plaintext signals or observe values on internal signals from outside of the package. By trusting the processor, it is assumed that the processor operates exactly as specified.

The second assumption is that all cryptographic algorithms (encryption, hash, digital signature) are cryptographically secure such that brute force and/or differential cryptanalysis are infeasible. This assumption does not obviate the need for various cryptographic parameters such as seeds, initialization vectors (IV), and keys to be appropriately applied and cryptographic algorithms to be correctly implemented.

The third assumption is that related problems such as key management/distribution and usage model development, although considered during the design, are correctly implemented.

Finally, it is assumed that all software, including the operating system, can provide support functionality but is not relied upon to execute security operations or provide truthful information.

4.2 Architecture Design Preliminaries

4.2.1 Design Goals

A viable, hardware-based solution to software protection must exhibit several important characteristics if it is to overcome the substantial resistance to change in the computing community. The complexity of modern computing systems, which range in capability from standalone embedded applications to a cluster of multiprocessor machines requires that hardware-based security mechanisms exhibit sufficient flexibility and extensibility to accommodate a wide variety of computing systems and applications. The following design goals were established with the aforementioned requirement in mind:

1. Absolute Design Goals

   *Achieve security*: The SSP should meet and/or exceed the stated security goals up to the limitations of the methodology (if they exist).

   *Protect secrets*: All cryptographic secrets such as keys should only exist within the trusted hardware and should never exit the security hardware in plaintext form.
Eliminate software trust: All software, including the operating system and protected applications, are considered untrusted, but can provide support functionality.

Maintain reasonable performance/overhead impact: The increased execution time and memory (storage and dynamic) overhead of added security features should be at an acceptable level.

2. First Tier Design Goals:

Reduce development chain impact: Steps in the software development chain such as source code authoring, compiling, and linking should require minor modification, if any, to support the security hardware of the SSP.

Processor-agnostic design: The design should not be tailored or designed around a specific architecture. It should be applicable to nearly any architecture in the current computing paradigm with little, if any, modification.

Analyzable security: If possible, the design of the architecture should be analyzable using a known or developed method to support system security claims.

3. Second Tier Design Goals:

Maintain reasonable architectural requirements: The design of the SSP should be such that the level of observation and level of interface required of the processor are within acceptable levels.

Legacy support: The SSP design should support regular, non-protected applications in addition to protected applications.

Scalability and extensibility (for other platforms): The design should, if possible, be scalable and extensible for use in other types of computing platforms or modes such as co-processor, cluster, multiprocessor, and embedded.

4.2.2 Design Questions and Challenges

This section presents the major questions and challenges that guide the research presented in this document. Additional questions related to each major question are also included.

1. Is it possible to add security features via external augmentation of processors with security management hardware? Because the security hardware is outside of the processor boundary does the limited access to internal processor information such as registers, caches, interrupts, virtual address, and the instruction pipeline, place a fundamental limit on the level or type of security augmentation can provide? Must the processor be redesigned or software trusted to obtain certain features? Are these limitations or lack thereof typical across most types and styles of processors?
2. What level of observability and/or access is required to the processor to add the various security features? If it is discovered that the amount of external observability and interface limits the security features that can be provided via augmentation, what is the minimal level of access to the internals of the processor that is required for each type of protection? If the amount of external observability and interface don’t dictate access needs, how far can the interface/observability be minimized? Can greater observability and better interface capabilities simplify the implementation of the security hardware or greatly improve performance? Can the assumptions or constraints be modified to vary observability and interface requirements? Do these requirements change with the class, types, style, or size of the processor?

3. How can the security claims of the hardware-based augmentation methodology be verified? Can existing formal methods be applied to the proposed system? How are the custom hardware and the system assumptions incorporated into the analysis? If existing methods are not sufficient, what is needed from a useful model and can a custom model be developed?

4. What are the performance effects and/or overhead associated with each type of added protection? Can the tradeoffs between performance, specifically the amount of time required to perform a task, and security be characterized? Can the tradeoffs between overhead, in the form of memory usage, and security be characterized? How do the working assumptions affect performance and overhead? How much of the performance variation is application specific? What set of metrics should be used to quantify these relationships?

5. What is the necessary or desired impact on the software development and deployment processes? Is access to source code needed or can compiled executable be postprocessed? Does this process change for different levels and or types of protection? Can this assumption/criteria be used to simply the design of the security hardware or greatly improve performance? Does modification or involvement in this flow place trust in software?

6. Is there loss in functionality? By adding security features using the augmentation method, is there any functionality? If so, is this functionality severely crippling? Would there be any gains from creating a custom processor with internal features like Aegis or XOM? How much improvement is there? In other words, does the result justify the cost/difficulty? Is there a gain in functionality that redesign cannot provide?

7. What types of protection are necessary to achieve the stated security goal? Must instructions be protected? Must data be protected? Must execution flow be protected? To what level (ie. integrity, confidentiality, etc) must these features be implemented? Do dependencies exist between them?
4.3 Instruction Protection

The first step towards achieving the goals of the SSP design is to protect the confidentiality of instructions by applying strong encryption to executable portions of an application. Instruction confidentiality has several benefits beyond thwarting direct observation. Combined with the inherent functionality of instructions, confidentiality provides a mechanism for access control and also complicates various attacks. The instruction-side protection architecture shown in Figure 4.2 consists of two primary components, the instruction encryption unit (IEMU) and the secure key management unit (SKU). The IEMU is responsible for identifying protected instructions as they are requested by the instruction cache and removing cryptographic protection using the appropriate key-algorithm combination. Key generation and key table population are performed by the SKU. The following sections discuss the details of each architectural component.

4.3.1 Encryption Management Unit

The instruction-side encryption management unit (IEMU) is the key component of the SSP instruction protection mechanism and is responsible for removing strong encryption from instructions flowing into the processor as shown in Figure 4.3. As instructions are fetched by the processor, the EMU must determine if the current request requires a cryptographic operation. If a cryptographic operation is deemed necessary, the EMU selects the appropriate
key from the working key table and decrypts the instructions upon arrival from main memory.

Because the IEMU resides beyond the processor boundary, incoming instruction fetches refer only to physical addresses. As a result, physical addresses must be mapped to protection information to determine if a cryptographic operation is necessary. In nearly every modern processor, memory management units are responsible for updating the cached memory mappings stored in hardware, usually in the form of translation lookaside buffers (TLBs) or block address translation tables (BATs). TLBs are cache-like structures that are commonly used in modern processors to increase the speed of resolving virtual-physical address mapping and can be either hardware or software controlled [105]. Block address translation units are similar to TLBs, but operate on portions of memory much larger than a single page. To maintain the single-page mapping between a physical page and its protection attributes, a hardware structure in the IEMU resembling an expanded version of a translation lookaside buffer (TLB), called the TLB mirror, is used. The TLB mirror contains the same physical address content of the processor’s TLB with additional information pertaining to the cryptographic status of each memory location as shown in Figure 4.3. More specifically, the TLB mirror contains the keyslot number associated with the key used to protect the contents of the the page in question. When memory access is requested, the address is compared to those residing in the TLB mirror table. If a matching address is located, the corresponding keyslot is used to index into the key table. A keyslot value of zero indicates that a cryptographic operation is not necessary and the instruction information is passed to the processor unmodified. Regardless of the the type of TLB control (software or hardware), the IEMU hardware must have access to MMU miss/hit and virtual-physical mapping information.
4.3.2 Secure Key Management Unit

The secure key management unit (SKU) is responsible for the generation of keys and management of their existence in hardware [106]. While many different key management methods are applicable, the SSP architecture uses a hierarchical, credential-based key management scheme [107]. Each credential identifier is associated with its corresponding credential as shown in Figure 4.4. A key is defined by a set of credential identifiers. The final working key used for encryption and decryption operations is generated by combining (usually cryptographically) the set of credentials defined by the credential identifiers as depicted in Figure 4.4. Upon receiving a key generation request, the SKU must retrieve the credential using the credential identifiers from the appropriate external source (smart card, IButton [108], network, etc.). After retrieving the credential information, the SKU generates the working key(s), places the working key(s) into a key table, and returns the location of the key(s) to the requester. Without the appropriate credential identifiers or the correct credential information, the working key(s) cannot be generated and access to protected information is cryptographically denied. It is assumed that the SKU is implemented such that the external interface does not result in secret information being inappropriately divulged.

4.3.3 Secure Executables

To create applications that contain confidentiality-protected instructions and can be used on the SSP, the executable portions of the application must be encrypted prior to execution. Additionally, the executable object must be augmented to contain information relating to the protection. The key management scheme [107] utilized by the SSP allows for the creation of secure objects. Similar to secure objects, standard executables are transformed into secure
executables.

A secure executable shown in Figure 4.5 is created by appending several additional data sections to the original executable similar to [51]. The first necessary extension, the secure key management header (SKMH), specifies the credentials necessary to create the working keys. Details of the SKMH structure, creation, and use can be found in [106]. Prior to the execution, the protected header is passed to the SKU which removes the header protection, extracts the key identifier list, and extracts the credential identifiers. Using the credential identifiers, the SKU can retrieve the associated credential data as described in the key list and generate the working keys. The second required addition to the executable, the key mapping, describes how the executable is protected using the working keys. More specifically, the key mapping associates a portion of an application with an application-relative key number. Each application-relative key number maps to a list of credentials, contained in the SKMH, needed to generate that particular key as shown in Figure 4.5. The mapping of each page to a application-relative key number enables the use of multiple keys for different portions of the executable at a page level granularity.

Important to the creation of a secure executable is the ability to check the correctness of the finalized, encrypted executable. Given a standard executable $X$, the secure elf generation process $S()$ can be applied $S(X)$ to create the secured executable. Comparing $S(X)$ to $X$ provides the ability to identify all plaintext portions of the executable. The correctness of the secure executable can be checked by applying the inverse creation process, $Y = S'(S(X))$. If $Y$ and $X$ match, the executable is encrypted properly with the correct key and all portions of the application that are meant to be encrypted are encrypted properly. It is important that the $S()$ and $S'$ be implemented independently to avoid pollution of either process with systematic errors. Finally, the secure secure executable process is assumed to operate correctly and is a trusted portion of the development chain.

### 4.3.4 Operating System Support

The role of the operating system in the SSP architecture is to provide support functionality in the form of information management. The first task of the operating system is to pass protected portions of secured executable objects, namely the SKMH, to the SKU hardware for use in key generation as shown in Figure 4.6. The operating system detects a protected executable by observing a variety of characteristics including filename, additional flags in the executable headers, or merely the existence of security-related headers such as those discussed in Section 4.3.3. The second task performed by the operating system is the completion of the mapping between physical pages and working keys. While the mapping of virtual resources to physical resources is a standard feature of modern, multitasking operating systems, the additional mapping between physical pages and key information required for IEMU operation must be added. During the key generation process, the operating system must update the key mapping data structure extracted from the secure executable with the key slot values.
Figure 4.5: Secure executable object layout

returned from the SKU as shown in Figure 4.7. As virtual pages are mapped to physical pages during page faults, the page fault handler must search the key map and update the appropriate data structures describing physical memory with key slot information. If the operating system fails to perform these operations or maliciously modifies information such that it is untruthful, the system will fail to operate. The failure will result from decrypting incoming instructions with the wrong key or failing to decrypt the information entirely. It is important to note that that the operating system handles only pointers to key locations (keyslot values), not the keys themselves. The level of indirection provided by keyslot values coupled with the read-only nature of instructions removes the need to trust the operating system.

4.3.5 Instruction Protection Operational Summary

In and effort to tie the responsibilities of each of the individual system components to the application execution process, the entire execution flow for an instruction protected (confidentiality) application is shown in Figure 4.8.
Figure 4.6: Depiction of the operating system interaction with the SKU. Since the SKMH is encrypted and verified by the SKU hardware, the operating system cannot alter the contents of the SKMH in a manner that adversely affects the key generation process. Also, the channel between the SKU and the user token is assumed to be secure. The result of the SKU operation are cleartext keys (created from the credentials on the token) that are used in SSP operation.
4.3.6 Encryption Algorithm and Mode of Operation Selection

Recent efforts in trusted computing platforms (TCPs) such as those mentioned previously have adopted the Advanced Encryption Standard (AES) [109]. AES, in addition to being a National Institute of Standards (NIST) standard [109], is among the fastest available algorithms, requires a relatively small amount of hardware area, exhibits a high degree of parallelism, accepts a wide range of block sizes, supports many modes of operation, and is well accepted by the security community. As a result, it was selected as the encryption algorithm for the SSP. It is important to note that the SSP is not dependent upon the AES algorithm itself and could use any encryption algorithm that exhibits similar properties and the required modes of operation.

The mode of operation selected for instruction protection is counter mode [110], shown in Figure 4.10. In addition to exhibiting very low latency as compared to its block mode counterparts, a feature that is critical for the latency-sensitive path between cache and main memory, counter mode is necessary for supporting the existence of data in executable sections. Data typically resides in the executable segments in shared/dynamic libraries and is used to support long jumps by position-independent code across the memory space. Countermode allows the ciphertext version of the data to be replaced with the plaintext without adversely affecting the decryption the encrypted block. In block mode, the switching
Figure 4.8: Process flow for the setup and execution of an instruction protected application on the SSP
Figure 4.9: Process flow for the setup and execution of an instruction protected application on the SSP (continued from previous page)
of plaintext with ciphertext would cause a decryption error. Given the increased risk of instruction tampering [14] when using counter mode encryption methods, the use of integrity verification must also be considered. Addition comments related to the selection of the instruction-side mode of operation can be found in Appendix C.

4.3.7 Instruction Integrity

Integrity protection for instructions implies that the contents of an instruction can be verified prior to execution with a high amount of certainty. Although commonly confused, encryption does not provide integrity [111]. Cryptographic integrity is generally provided by cryptographic hash-functions such as those discussed in Appendix D. In using integrity verification mechanisms, it is important to consider the consequences of composing encryption and integrity [112] and to maintain a separation of goals by using separate keys for encryption and integrity checking algorithms [110]. Within the context of TCPs, the role of integrity checking is to reduce controllability. A hash is calculated for the information in question and is stored off-chip with the encrypted data. When the information is requested, both the hash and data are retrieved and the hash is recalculated. If the retrieved and calculated hashes are equal, the data is valid. Typical usage of hash functions for verifying instruction (read-only) integrity in TCPs is shown in Figure 4.11. The current set of accepted cryptographic keyed hash algorithms, MD5 [113] and SHA1 [113], exhibit high latency (≈ 160 cycles per 512 bit message) and have large hash digests (≥ 160 bits), greatly impacting execution times and storage overhead [5] [6]. As a result, it is necessary to use an alternative to standard cryptographic algorithms, namely Galois Counter mode (GCM) [114], an integrity-aware encryption method.
Figure 4.11: This figure outlines the protocol for using a cryptographic hash for integrity checking on the SSP for a read-only scenario. Since read-only data is known \textit{a priori} the hash values can be computed prior to execution.
Figure 4.12: In this depiction of Galois counter mode, note the feedback of the encrypted counter mode output to the Galois multiplier along with additional data to be authenticated. The resulting tag is a cryptographically secure hash that can be stored off chip and retrieved for comparison to a newly calculated hash of the off-chip data.

Galois Countermode Operation

The GCM mode of operation, shown in Figure 4.12 encrypts a block of a message using the standard counter mode operation and produces the authentication data using a Galois field multiplication over the binary field $GF(2^{128})$. The resulting $n$-bit authentication data provides $n - k$ bits of authentication security for messages that are $2^k$ bits long [115] and can be truncated if necessary. GCM also provides the ability to authenticate additional data. Hardware implementations of GCM [116] have shown that the authentication tag calculation can be performed in as little as a single cycle and that the hardware consumed by the Galois field multiplier is relatively small ($\sim 78$ slices). The storage overhead $O$, of GCM mode is $O = \frac{1}{n}$ where $n$ is the number of 128 bit blocks in the message. Many thirty-two bit architectures with instruction caches have cachelines of 128 bits in size. Assuming a maximum of two cachelines per memory access, the storage overhead ranges between 1 and $\frac{1}{2}$ of the original code size. The low-latency and reduced overhead as compared to MD5/SHA1 make GCM suitable for using in the IEMU. The location of the GCM integrity checking unit in the SSP architecture is shown in Figure 4.13.
Figure 4.13: Hardware subcomponents of the security management unit (SMU) for instruction protection with integrity checking (shown with GCM style checking)
4.4 Data Protection

Data differs significantly from its executable counterpart in that:

1. Data can be both read or written.
2. Data lacks the inherent functionality exhibited by executable instructions.

Due to the additional write capability exhibited by data, the data-side security hardware, specifically the data encryption management unit (DEMU), must also apply protection. The placement of the DEMU in the SSP architecture is shown in Figure 4.4. In the protection of instructions, the type of protection is designated by a key identifier. The key identifier indicates the correct key to use in the removal of cryptographic protection from instructions. This method is failsafe in the sense that if an error occurs or if the identifier is maliciously tampered with, the decryption will fail, resulting in invalid information being passed to the processor for execution. The application of cryptographic protection requires the additional knowledge of how to protect the information. If the same protection method used for instructions were used in the application of cryptographic protection, an error or malicious tampering with the key identifier would cause the inappropriate application of protection.

The lack of inherent functionality must also be considered in the design of the data-side protection mechanisms. In the case of protected instructions, if an application inadvertently or maliciously directs its flow of execution to a protected instruction space and the key is available in the hardware key tables, the encrypted instructions will be decrypted and executed. The redirection has a direct impact on the operation on the original. In the case of data, the lack of inherent functionality allows instructions to switch between data streams without affecting the operation of the executing instructions. As a result, if a method similar to that used for instructions were used to protect data an application would be able to read any data whose key resides in the hardware.

4.4.1 Read-only/Static Data

A special case of data protection is read-only data. Read-only data is similar to instructions in that it is protected prior to execution and only requires the removal of cryptographic protection. At the cost of additional hardware, read-only data can be protected differently than read/write data using the counter mode or Galois counter mode methods applied to instructions. Cryptographic separation of read/write and read-only data has the benefit of providing an additional cryptographic problem to a would-be attacker. Because read-only data commonly contains constants, coefficients, or other types of sensitive information, separation of read/write and read-only data is a desirable and optional feature.
The fundamental issue behind the protecting data is that executable code and its corresponding data are not associated. Associating instructions with their corresponding data implicitly provides information necessary for both the removal and application of cryptographic data protection.

### 4.4.2 Associating Instructions with its Data

In existing resource sharing systems, the operating system is trusted to enforce process separation both temporally (contexts) and spatially (virtual address space). Associating code with its data is made much more difficult by the inability to trust the operating system. Associating code with its data without trusting the operating system requires that the hardware to have the ability to associate executable code with the physical pages it accesses.

To achieve this association, the hardware must determine which subset of code is executing. Operating systems separate application code and data using a process identifier (PID), however the PID cannot be used as identifier due to the inability to trust the operating system. Luckily, the IEMU hardware provides another mechanism for detecting the execution of protected instructions and identifying what instructions are executing: the instruction
 decryption keys. Keys do not uniquely identify applications because applications may have multiple keys. To differentiate between applications that may include similar keys in their key sets, each set of application keys is tagged with a tag value $T_N$. This tag is used to identify that a particular application is currently executing and provides a many-to-one mapping between application instruction keys and the corresponding data key set as shown in Figure 4.15. Tags also provide separation even in the event of overlapping keys in the key set as shown in Figure 4.16.

**Shared/Dynamic Library Support**

Generally, an application is composed of a top-level executable and several shared libraries [106], shared libraries must also be able to access protected data. Because the association between protected instructions and data relies upon the set membership of instruction decryption keys, all libraries of a data-protected application must also include executable instruction protection. A library without instruction protection would be associated with a tag value of zero which would not allow the library to access data in a protected application.

If only the single tag discussed in Section 4.4.2 was used it would be necessary to use the same data key for both static data and dynamic data. If an additional tag is added as shown in Figure 4.15 to indicate the static key, the need to compile libraries uniquely for each application is removed. Using the tag association method, the internals of the DEMU
Consideration of Internal Processor State

Because the SSP hardware is located beyond the processor boundary, the state of the instruction cache is also a concern when determining what instructions are executing. To be certain that only executable code decrypted from keys tagged with $T_{current}$ is executing, the instruction cache must be invalidated prior to execution of code protected using keys from the set $T_{next}$. By examining the tag values for the two most recent code fetch requests the hardware can perform an instruction cache flush whenever a change in tag value is detected. Although cache invalidation/flushing seems like a drastic operation, results shown in Chapter 5 indicate that cache invalidation on context switches as a small impact on execution time. When checking tag values four possible cases exist. Given two non-zero tag values $A$ and $B$, the possible cases are:

1. $T_{next} = A$, $T_{current} = A$: The system is executing in the same hardware context and does not require a flush/invalidation.
2. $T_{next} = A$, $T_{current} = 0$: The system is transitioning from an unprotected context to a protected context and requires a flush/invalidation.
3. $T_{\text{next}} = 0, T_{\text{current}} = A$: The system is transitioning from a protected context to an unprotected context and requires a flush/invalidation.

4. $T_{\text{next}} = B, T_{\text{current}} = A$: The system is transitioning between two different protected contexts and requires a flush/validation.

Case 1 represents continued processing of the current protected application. Cases 2 and 3 represent switching to and switching from a protected process. Regardless of the cause of a zero-to-nonzero or nonzero-to-zero tag transition, (OS context switch or an attacker) the hardware must perform cache flushes/invalidations to remove instructions and data. If flushing and/or invalidation is not performed, information from the protected application may be leaked or known data could be used taint the protected application. If a non-encrypted OS is used, a nonzero-to-nonzero tag transition, denoted by Case 4, indicates an attack in progress or an erroneous operating system. If the SSP was modified to support an encrypted operating system, Case 4 could no longer be used as an attack indicator. If the fourth case occurs, the DEMU must instruct the IEMU to stop processing instructions, signal an error, and/or perform countermeasures, if available.

The protection of data is also hampered by the inability to protect the contents of registers on interrupts and context switches. It is possible for data to leak via general purpose registers along with other application state registers. Interrupts and context switches also provide opportunities to modify the application state including the execution point (i.e. program counter). Given the relatively large time slices ($\sim 10\text{ms}$) and fast execution capability of
modern processors (400Mhz $\rightarrow \sim 4$ million instructions in 10ms) the availability of these opportunities is small. When combined with a reasonably complex application, the lack of instruction information resulting from instruction confidentiality protection and tampering countermeasures such as integrity, the effects of information leakage through registers can be reduced significantly.

4.4.3 IEMU Modifications for Data Protection

To support data protection, the instruction side key table must be modified to include the tag identifiers and any necessary flags (such as the downgrade flag). Also, signals corresponding to the most recently used tag and flags must be included in the communication between the data and instruction side.

4.4.4 SKU Modifications

To support data protection, the SKU must be modified to support key generation requests that specify multiple keys. Upon finishing the key generation and populating the key tables, the SKU must also return the range of key slots used. Finally, the SKU must tag the generated keys with a unique (relative to all other existing entries) key set identifier.

4.4.5 Secure Executable Modifications

To support data protection any read-only data that is protected must be encrypted and placed in the appropriate section. Also, data sections that may contain protected data must be properly aligned in the secure executable such that protected and unprotected information do not reside in the same page.

4.4.6 Operating System Modifications

To support data protection, the operating system must perform three crucial functions. First, the operating system must appropriately mark memory pages as protected, read-only, and prevent the data pages of a protected application from being swapped. The protected flag allows plaintext data to be read by a data protected application and is only used during the reading process. The read-only flag allows selection of the appropriate key (static/dynamic) in the DEMU hardware. The swapping of data protected pages is not allowed if the physical address is used in the integrity checking process. If pages are swapped, it is likely that the new virtual-to-physical mapping will be different causing the integrity check to fail. As a result, the same virtual-physical mapping must be maintained through the execution lifetime
of a data protected process. Second, the operating system must ensure that the beginning of the stack/heap is page aligned so that unprotected input data passed into the application on the stack is not corrupted. Finally, if the data cache is software controlled, the operating system must ensure that the data cache is configured as write-through. If the operating system fails to perform these necessary functions, the system will fail to function properly. It is important to note that failure (by the operating system) to perform required tasks will not result in a breach of instruction or data security.

4.4.7 Data-side Protection Operational Summary

In and effort to tie the responsibilities of each of the individual DEMU system components to the application execution process, the entire execution flow for a data-protected (confidentiality only) application is shown in Figure 4.18.

4.4.8 Encryption Algorithms and Modes of Operation

The protection of data confidentiality using cryptographic algorithms is not as straightforward as using the same counter mode operation utilized by instruction protection. Counter mode operation requires the generation of unique key-counter pairs for each block of encrypted information. Similar to instructions this requirement can be satisfied for read-only data, as its values are static and known a priori. As for read/write data, multiple (likely differing) values would be written to the same physical location using the same key/counter pair, violating the conditions of counter mode use.

In lieu of counter mode and despite additional calculation latency, CBC block mode encryption is used to handle the data protection encryption mode. CBC block mode encryption removes the need to produce unique key-counter pairs at the cost of higher latency. While memory access patterns prevent widespread use of CBC across the entire memory space, CBC can be used across a small number (one or two) of blocks. The initial IV is provided using method used for counters in the IEMU counter mode unit. One additional consideration when using block mode encryption is the need to perform cryptographic operations on blocksize portions of data. Since the data cache will fill or flush a line at a time, this requirement is easily satisfied. If the cache operates on data subsets smaller than the encryption algorithm blocksize, extra memory access may be necessary to retrieve the additional data need to perform the block encryption or decryption.

4.4.9 Data Integrity

In the case of the SSP, integrity checking is used to verify the contents of information stored off-chip in main memory. Instead of sending a message to another party, the SSP
Figure 4.18: The data side process flow
Figure 4.19: Hardware subcomponents of the security management unit (SMU) for combined instruction and data protection with integrity verification

essentially sends a message to itself as shown in Figure 4.21. Taking advantage of the unique circumstance of data protection on the SSP, the characteristics of block mode encryption, and common error detection codes, a new low-latency method of checking data integrity is possible.

Suppose the SSP stores a block mode-encrypted portion of data \( C \) to external memory where \( C = E(A) \) and \( A \) is the original data value. Meanwhile an attacker tampers with the value \( C \) in external memory (as shown in Figure 4.20) such that the value in memory is now a tampered value of \( C \) while will be denoted as \( C_* \). When the tampered block, \( C_* \) is fetched from memory, the decrypted plaintext \( P \) is calculated, \( P = D(C_*) \). Given the nature of the block mode encryption and decryption functions \( E \) and \( D \), \( P \neq A \). Because the incorrect value \( P \) can be viewed as a version of \( A \) with random errors, an error detection code such as a cyclic redundancy check (CRC) [110] can be used to inspect the integrity of the original data.

**CRC for Data Integrity Checking**

The CRC algorithm allows random errors resulting from the decryption of a tampered block to be reliably detected with very low latency. Due to the relatively small size of a CRC as compared to a cryptographic hash digest, the CRC has the potential to be stored on-
chip. On-chip storage not only avoids costly off-chip communication with memory, but also avoids attacks directed at cryptographic hash functions and issues surrounding generic composition [112]. Because the CRC operates on the original data and not the encrypted data, there is no need to include additional data such as a counter to ensure data freshness. The amount of resources consumed in time and space depend upon the number of CRC bits used and the size of target message. Figures 4.24 and 4.23 show the number of slices consumed on a Xilinx Virtex-II Pro FPGA [2] for varying CRC core configurations and the CRC latency as a function of blocksize, respectively. The amount of logic consumed by even the most aggressive configuration is less than 1000 FPGA slices. Examining Figures 4.24 and 4.23 shows that the selection of a moderately sized CRC data width and memory blocksize have a large impact upon latency and memory storage. Even for large message sizes, a moderate CRC data width results in reasonable latency.

4.4.10 Operating System Interaction & I/O

One issue complicated by the use of confidentiality protected data is system input/output (I/O). In most systems, all I/O at the physical layer is assumed to be in plaintext. In a system such as the SSP that supports encrypted data, the possibility exists for inputs such as standard I/O streams, network, mouse, and keyboard to be encrypted. The situation is further complicated by the possibility of mixing both encrypted and plaintext I/O sources.
Figure 4.21: Protocol for using a cryptographic hash for integrity checking on the SSP in a read/write scenario (data protection)
To successfully use the different types of I/O without greatly complicating the standard programming model, the operating system must properly indicate if protection is used and maintain appropriate page alignment to comply with the page granularity requirements of the SSP.

To properly indicate protection attributes of different data streams, the operating system must know that a particular I/O source is protected or the application must provide some information about the incoming data stream. In the first case, new versions of standard I/O streams such as encrypted input and encrypted output could denote the use of encrypted I/O. In the second case, the application could pass an additional parameter to system calls that access I/O (such as open()) to instruct the operating system to mark corresponding pages as protected. While unencrypted output is almost always desired in a secure application, plaintext input might also be undesirable as it provides a pathway for an attacker to inject known information.

### 4.4.11 Architecture Summary

Typical usage of the SSP architecture is shown in Figure 4.26. Source code is compiled using a standard compile. Using information from an authority whose role is to specify
Figure 4.23: FPGA resource consumption for a single CRC core [1] on a Xilinx XV2P30-6 [2] using Xilinx XST 7.1
Figure 4.24: CRC calculation latency

the type and level of security applied to the compiled executables, each executable is post-processed to create a secured application. Additionally, the security authority must place the corresponding credentials on a secure token. To execute the protected application, the user then inserts the token into the SSP and requests execution of the protected application. Removing the token from the platform removes execution authorization.
Figure 4.25: Memory requirements for storing CRC values for an entire thirty-two bit memory space on-chip (Note: Memory requirements reduced dramatically if protection is limited to a certain portion of physical memory (this requires kernel mods and limits the total number of protected process)
Figure 4.26: A high-level view of the SSP operation that details a typical usage scenario
Chapter 5

Implementation, Analysis, & Results

In this chapter, Section 5.1 discusses the implementation of the SSP. Section 5.2 presents a security analysis of the SSP and Section 5.3 outlines performance and resource consumption results. Finally, Section 5.4 outlines a set of design principles in the form of processor interface requirements and processor features needed to support the processor augmentation methodology.

5.1 Implementation

While many previous trusted computing platforms such as XOM [5] and AEGIS [6] were simulated, implementation of the SSP was both useful and necessary for several reasons. First, an implementation provides the ability to characterize the effects of operating system modifications. Second, in the process of implementing the design, several design hurdles that would likely not be discovered using simulation alone were uncovered. Third, the resulting implementation enabled measurement of required logic and memory overhead. Finally, a working prototype platform supports the assertion that the SSP is immediately realizable and allows the system to be fully characterized. Due to FPGA resource limitations and the reliance upon application-specific security requirements, instruction and data integrity verification mechanisms were not implemented.

5.1.1 Prototype Platform

The Xilinx ML310 [117] development board was selected as the implementation platform due to its wide range of features including a Xilinx Virtex-2 Pro (V2P) FPGA (specifically the XC2VP30) [2], 256 MB of memory, 10/100 Ethernet, and high-speed I/O. The close coupling of hardcore PowerPC 405 processor cores (PPC405) [118] with configurable logic
Figure 5.1: This figure depicts the implementation of the SSP platform. Multiple PLB busses in combination with PLB-PLB bridges allow the shoehorning of the IEMU and DEMU into the memory hierarchy. The SKU interacts with the operating system and token via an OPB bus.

found in the V2P family of FPGAs, combined with other features of the ML310 enabled the implementation of the SSP.

The PowerPC 405 Processor

The PPC405 processor core is a thirty-two bit RISC processor that performs single-cycle (excluding multiplication and division), in-order execution. The PPC405 also includes a software-controlled memory management unit (MMU) with split instruction and data caches, each being 16KB in size. The PPC405 operates at 400Mhz and is capable of executing a complete Linux [119] operating system.
5.1.2 Encryption Management Units

The data and instruction encryption management units (EMU) were implemented using a combination of the VHDL and Verilog hardware-description languages. To achieve the necessary mappings (many-to-one, one-to-many, many-to-many) in the hardware implementations of the EMUs, several levels of content addressable memories (CAMs) were required [120]. Cryptographic functionality for the IEMU was provided by an open-source AES implementation modified for the specific needs of the SSP [121]. Due to FPGA resource limitations including logic and block RAMs an XOR block with added delay was used to emulate the AES blockmode operation of the DEMU. The resulting IEMU and DEMU implementations were each interfaced to the instruction and data caches of the PPC405 using the IBM Processor Local Bus (PLB) [122] standard, via PLB-PLB bridges. The IEMU and DEMU TLB mirror and key table structures were configured to hold sixty-four instruction key slots. Given the use of Xilinx IP Cores (PLB IPIF) and multiple levels of CAMs, several (∼6) additional cycles of latency were added to memory transactions (∼22 cycles) beyond those added by the SSP security features (∼2-10 cycles).

5.1.3 Secure Key Management Unit

Because of the complexity of the chosen key management scheme and to reduce development time, the SKU was implemented [106] using the Xilinx MicroBlaze [123] softcore microprocessor operating at 100Mhz. The user token that provides the credentials to the SKU for the key generation process was implemented using an iButton [108]. Communication between the SKU and the operating system for the purpose of passing secure headers and key generation status was established via an On-Chip Peripheral Bus [124], memory mapped I/O, and a device driver kernel module.

5.1.4 Secure Executable Postprocessing Utility

A post-processing utility was created that enables the generation of secure executable objects. The utility accepts as input a standard ELF [52] and a secure key management header provided by a security authority [106], and outputs a secure ELF executable object as discussed in Section 4.3.3. In processing an ELF object, the utility is able to recognize objects (typically shared or dynamically linked libraries) that contain data in the executable segment and apply encryption appropriately; enabling the use of protected shared libraries. The utility is able to apply a variety of cryptographic algorithms.

To support data-protection, a custom linker script that appropriately aligns data segments for protection was also created for use in the linking process. Additionally, the utility added additional entries to the key mapping list to hold the address boundaries for the stack and heap, respectively.
5.1.5 Operating System Kernel Modifications

The operating system used in the implementation was MontaVista Linux [119] version 2.4.20. Required modifications to the operating included the following:

1. *struct_page memory structure:* The *struct_page* data structures that describe the state of physical memory pages were modified to include key slot and counter information.

2. *task_struct memory structure:* The *task_struct* that describes the status and state of a process was modified to include protection status information.

3. *ELF loader:* The ELF binary application loader was modified to recognize secure applications, pass the appropriate information to the SKU for the key generation process, and update necessary data structures with results from the key generation process.

4. *TLB miss interrupt handler:* The interrupt handler was modified to pass mapping information to the EMUs concerning physical pages, keyslot locations, and flags.

5.2 Security Analysis

5.2.1 SSP

Using the implementation configuration that includes the PPC405 processor, a security analysis of the SSP architecture can be performed using the method discussed in Chapter 3. The single context model for instruction protected used for the SSP is shown in Figure 5.2.1. The results of the analysis shown in Figures 5.2a and 5.2b, indicate that the instruction cache debug (*icache_debug*) instruction of the PPC405’s ISA can allow protected instruction to be downgraded by moving them to the registers and out as plaintext through the unprotected data side.1 Because the *icache_debug* instruction is not essential to general processor operation, an instruction filtering unit can be added to the IEMU that scans decrypted instruction blocks and replaces the *icache_debug* instruction with the *nop* instruction. The filtering operation can be performed on a decrypted block of instructions in a single cycle, adding a small amount of latency to instruction-side datapath.

---

1Some edges are omitted for the sake of clarity. Also, the labels within each node denote type, membership, contents, and node number. For example, *P1I* is the instruction memory for process one.
Figure 5.2: (a) Graph analysis results for a single context SSP systems that includes the *icache_debug* instruction. Note the undesired downgrade path (dashed), caused by the ability to move unencrypted instructions to registers from the instruction cache. (b) Graph analysis results, showing no violations, for an SSP system that incorporates instruction filtering to remove the affects of the *icache_debug* instruction.
Extending the SSP model in Figure 5.2.1 to include data protection (DEMU) and performing the analysis yields Figure 5.3. The analysis indicates that the SSP architecture successfully removes trust from software for both instruction and data protection due to the lack of violations. This version of the SSP incorporates instruction filtering, cache flushing (instruction and data) and also includes register cleaning. In combination with the instruction filtering, the cryptographic separation of the instruction and data datapaths prevents instruction information from being manipulated by dataside operations. The cryptographic separation of datapaths distinguishes the SSP from other trusted computing platforms such as XOM and AEGIS. While instruction side cache flushing prevents execution of instruction outside of the current hardware context, data cache flushing prevents data from leaking to other hardware contexts. Finally, register cleaning prior to context switching prevents information leakage across context between registers. An extended discussion concerning the downgrading of information is found in Appendix F.
Figure 5.3: Graph analysis results for multi-context SSP systems that include both instruction and data protection. The lack of violations indicates that the SSP architecture is secure under the stated assumptions.

Table 5.2.1: Model of the SSP with both the IEMU and DEMU for multiple contexts

5.2.2 Security Analysis of the CRC Integrity Verification Scheme

The CRC-based integrity checking scheme achieves reduced latency and flexibility, at the cost of reduced security. The security provided (i.e. the probability of detecting tampering) when using CRC depends on the following parameters
1. **CRC Width**: The CRC width of $n$ bits determines overall detection probability $P_{\text{detect}} = 1 - \frac{1}{2^n}$.

2. **CRC Polynomial**: The CRC polynomial determines the type of errors (single-bit, multi-bit, burst, etc.) that are detectable.

3. **Encryption Algorithm Characteristics**: The types of errors introduced by tampering with encrypted values will affect detection capability.

4. **Memory Block Size**: The size of the message input to the CRC affect the amount of time required to complete the CRC calculation.

Suppose an adversary is armed with the operational knowledge of the proposed CRC-based scheme and can perform an attack every $t$ seconds. Assume an attack consists of tampering with a value and observing the effect of the tampered value. Also, assume the attacker has a method of knowing if the CRC value of the tampered decrypted data collides with the correct CRC value stored on chip. For an $n$-bit CRC the amount of days for an attacker to find a collision, $A_t$ is expressed as

$$A_t = \frac{2^{\frac{n}{2}} \cdot t}{86400}.$$  \hfill (5.1)

The $2^{\frac{n}{2}}$ term in Equation 5.1 results from consideration of the Birthday Attack [110]. The Birthday Attack, resulting from the famous Birthday Paradox [110], states that for an $n$-bit hash function, only $2^{\frac{n}{2}}$ hashes must be calculated before a collision is found, not $2^n$. Figure 5.4 shows the number of days needed to find a collision as a function of the number of seconds per attack $t$ and the CRC width $n$ using Equation 5.1.

If the use of smaller CRC values such as eight or sixteen bits is desired, sufficient countermeasures may be required to achieve security requirements. An example of a countermeasure would be to halt hardware operation for a certain period of time if an integrity verification failed. Figure 5.5 details the effects of a timeout countermeasure when using shorter CRC values. The attack time estimates in Figures 5.4 and 5.5 are realistic because an attack must be focused on a single platform and cannot be parallelized unless multiple identical units are available for attack.

### 5.3 SSP Cost Analysis Results

The results in this section outline the contribution of each SSP feature (where applicable) to the total spatial and temporal costs and also indicate that the costs of using the SSP are reasonable. In interpreting the results, it is important to remember that they are specific to the particular prototype implementation described in Section 5.1 and is not optimized or tuned for any particular metric such as performance or resource consumption.
Figure 5.4: Attack difficulty in time for the CRC integrity scheme when using larger CRC bit widths.

Figure 5.5: Attack difficulty in time, for the CRC integrity scheme for smaller CRC bit widths (Note that the units of x axis are in hours, not minutes as in Figure 5.4.) Also note that a specific halt time was not used. It is necessary that the particular countermeasure deployed limits the attack times to those shown on the left axis.
5.3.1 SSP Architecture Cost Breakdown

The security features added to a standard microprocessor by the SSP architecture come at the cost of space and time. Figure 5.6 decomposes the costs various system hardware and software features within the spatial and temporal cost categories.

5.3.2 Spatial Costs

The spatial costs of the SSP include additional memory storage space and extra hardware logic. The additional memory storage stems from additions to the operating system need to support the SSP hardware and the logic area consumed by the SSP-specific hardware.

Kernel Additions

Table 5.1 compares the size of the executable portion of the kernel for the base configuration to each of the associated kernel configurations listed in Section 5.1.5. The total added size of the combined kernel additions is less than 1/5th of 1%. The majority (70%) of the additional size is contributed by the ELF loader modifications. In a similar manner, Table 5.2 compares the size of the data portion of the compiled kernel configurations. The sum total of the additions to the data segment of the compiled kernel image are also less then 1/5th of 1%; contributed entirely by the ELF loader modifications. The additional space stems from the combination of additional constant definitions and error message strings. The page table extensions, though small, contributes not to that static kernel space but to the space allocated for kernel data structures at runtime. Given the small size of the struct page at approximately eight unsigned integers, the addition of four integers approximately doubles the size of each struct page.

SELF Additions

Creating a SELF object also results in additional memory storage overhead. It is important to note that the encryption of instruction and read-only data itself does not add to the portion of the executable that is loaded into memory for execution. The increased space requirement results from the addition of the key map data structure and the SKMH. The number of entries in the key map is directly proportional to the combined number of executable pages and read-only data pages. The size of the SKMH [106] is large enough to support the desired number of keys and does not change as a function of executable size. Table 5.3 shows the

\[\text{\textsuperscript{2}}\] The four integers represent the key slot value and the four thirty-two bit portions of the 128 bit counter values.

\[\text{\textsuperscript{3}}\] One exception to this statement is the need to pad the last read-only data block to comply with block encryption requirements.
<table>
<thead>
<tr>
<th>Spatial Costs</th>
<th>Hardware</th>
<th>Temporal Costs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software</td>
<td>Hardware</td>
<td>Software</td>
</tr>
<tr>
<td>Executable</td>
<td>IEMU Logic</td>
<td>Process Setup</td>
</tr>
<tr>
<td>Additions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Page Table</td>
<td>DEMU Logic</td>
<td>Page Fault Handler</td>
</tr>
<tr>
<td>Extensions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Page Fault Handler</td>
<td>SKU Logic</td>
<td>TLB Miss Interrupt</td>
</tr>
<tr>
<td>Extensions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TLB Miss Interrupt</td>
<td>IIVU Logic</td>
<td></td>
</tr>
<tr>
<td>Additions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Executable</td>
<td>DIVU Logic</td>
<td></td>
</tr>
<tr>
<td>Loader(kernel)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.6: SSP Costs
### Table 5.1: Increase in size of the compiled kernel’s executable section for different kernel configurations

<table>
<thead>
<tr>
<th>Kernel Configuration</th>
<th>Executable Sec. Size (bytes)</th>
<th>Pct. Increase (from Base)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>1253000</td>
<td>—</td>
</tr>
<tr>
<td>Exec. Loader</td>
<td>1254480</td>
<td>0.1181</td>
</tr>
<tr>
<td>TLB Interrupt</td>
<td>1253384</td>
<td>0.0306</td>
</tr>
<tr>
<td>Page Fault</td>
<td>1253260</td>
<td>0.0208</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>0.1695</strong></td>
</tr>
<tr>
<td><strong>SSP</strong></td>
<td>1255112</td>
<td><strong>0.1686</strong></td>
</tr>
</tbody>
</table>

### Table 5.2: Increase of the compiled kernel’s data section for different kernel configurations

<table>
<thead>
<tr>
<th>Kernel Configuration</th>
<th>Data Sec. Size (bytes)</th>
<th>Pct. Increase (from Base)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>117384</td>
<td>—</td>
</tr>
<tr>
<td>Exec. Loader</td>
<td>117552</td>
<td>0.1431</td>
</tr>
<tr>
<td>TLB Interrupt</td>
<td>117384</td>
<td>0.0000</td>
</tr>
<tr>
<td>Page Fault</td>
<td>117384</td>
<td>0.0000</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>—</td>
<td><strong>0.1431</strong></td>
</tr>
<tr>
<td><strong>SSP</strong></td>
<td>117552</td>
<td><strong>0.1431</strong></td>
</tr>
</tbody>
</table>

Table 5.1: Increase in size of the compiled kernel’s executable section for different kernel configurations

Table 5.2: Increase of the compiled kernel’s data section for different kernel configurations
executable file size increases resulting from SELF post-processing. With the exception of the JPEG benchmarks executables, the SELF additions were less than $\frac{1}{10}$th of 1% larger than the original. The discrepancy results from the much smaller original size of the JPEG benchmark executables.

<table>
<thead>
<tr>
<th>Size (bytes)</th>
<th>Pct. Diff from Original</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Original</strong></td>
<td><strong>SELF</strong></td>
</tr>
<tr>
<td>FFT</td>
<td>4970209</td>
</tr>
<tr>
<td>qsort</td>
<td>4559112</td>
</tr>
<tr>
<td>Rijndael</td>
<td>4624440</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>4584403</td>
</tr>
<tr>
<td>GSM (toast)</td>
<td>4819165</td>
</tr>
<tr>
<td>GSM(untoast)</td>
<td>4819165</td>
</tr>
<tr>
<td>JPEG (cjpeg)</td>
<td>335820</td>
</tr>
<tr>
<td>JPEG (djpeg)</td>
<td>339795</td>
</tr>
<tr>
<td>Stringssearch</td>
<td>4544331</td>
</tr>
<tr>
<td>STREAM</td>
<td>4643557</td>
</tr>
<tr>
<td><strong>Mean</strong></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.3: Executable file size increases for each of the benchmarks after post-processing using the SELF generation utility

**SSP Logic**

Table 5.4 details the resource usage in FPGA slices and FPGA block RAMs (BRAMs) of each major SSP component and the entire synthesized SSP design. The discrepancy between the total SSP design (82%) and the sum of the component usage (32%) is explained by the use of several cores needed to interface with I/O on the ML310. Specifically, the extra cores included a PCI core (~30%), an $I^2C$ core, a UART core, and debug (Chipscope) cores. Gate counts for each module can be approximated using an appropriate conversion factor from slices to gates for the particular device in use. This calculation must also include an assumption concerning the usage ratio of slice logic for logic functions and memory which has an effect on the number of required gates. Typical approximations assume that 80% of a slice is used for logic and 20% for memory.\[4\]

---

4 Gates per slice estimates vary widely (~ 15-100 gates/slice) [125]
<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$</td>
<td>Application performance in total execution time</td>
</tr>
<tr>
<td>$MA$</td>
<td>Average memory access time</td>
</tr>
<tr>
<td>$MA_{inst}$</td>
<td>Memory accesses resulting from instruction fetches</td>
</tr>
<tr>
<td>$MA_{data}$</td>
<td>Memory access resulting from data fetches</td>
</tr>
<tr>
<td>$MA_{total}$</td>
<td>Total memory accesses</td>
</tr>
<tr>
<td>$DC_{miss}$</td>
<td>Average L1 data cache miss rate</td>
</tr>
<tr>
<td>$DC_{hit}$</td>
<td>L1 data cache hit time in cycles</td>
</tr>
<tr>
<td>$DC_{penalty}$</td>
<td>Average L1 data cache miss penalty in cycles</td>
</tr>
<tr>
<td>$IC_{miss}$</td>
<td>Average L1 instruction cache miss rate</td>
</tr>
<tr>
<td>$IC_{hit}$</td>
<td>L1 instruction cache hit time in cycles</td>
</tr>
<tr>
<td>$IC_{penalty}$</td>
<td>Average L1 instruction cache miss penalty in cycles</td>
</tr>
</tbody>
</table>

Table 5.5: List of variables for the analytical SSP performance discussion

### 5.3.3 Temporal Costs

**Analytical Analysis**

For a processor that performs in-order execution, processor performance in terms of total execution time is directly correlated to the average memory access time [105]. Using the known set of processor and system features the analytical equations for average memory access time can be derived. Consider the variables in Table 5.5.

Assume that the processor in question has split L1 data and instruction caches. Using the variables from Table 5.5, the average memory access time for a processor with a split cache configuration is expressed as [105]

$$MA = \frac{MA_{inst}}{MA_{total}} \ast (IC_{hit} + IC_{miss} \ast IC_{penalty}) + \frac{MA_{data}}{MA_{total}} \ast (DC_{hit} + DC_{miss} \ast DC_{penalty}).$$

(5.2)

In adding the protection of the SSP, the average memory access time (penalty) is increased by $E_{iemu}$ and $E_{demu}$ cycles as a result of the encryption and decryption processes of the IEMU and DEMU, respectively. It is important to note that due to the architectural reliance of...
the DEMU upon the IEMU, the DEMU cost cannot be calculated independently from the IEMU. Adding the cost of cryptographic operations to Equation 5.2 yields the performance average memory access time for a full SSP system $M_{ssp}$

$$M_{ssp} = \frac{M_{inst}}{M_{total}} \times (IC_{hit} + IC_{miss} \times (IC_{penalty} + E_{iemu})) + \frac{M_{data}}{M_{total}} \times (DC_{hit} + DC_{miss} \times (DC_{penalty} + E_{demu})).$$

(5.3)

Since average memory access time is approximately proportional to total execution time for an in-order processor, slowdown [105] relative to the base system, $S_{ssp}$ can be approximated by

$$S_{ssp} = \frac{M_{ssp}}{M_{A}} = 1 + \frac{M_{data}}{M_{total}} \times DC_{miss} \times E_{demu} + \frac{M_{inst}}{M_{total}} \times IC_{miss} \times E_{iemu}}{M_{A}}. \quad (5.4)$$

Slowdown of the DEMU relative to an IEMU-enabled SSP, $S_{demu/iemu}$ is

$$S_{demu/iemu} = 1 + \frac{M_{data}}{M_{total}} \times DC_{miss} \times E_{demu}}{M_{A_{iemu}}}. \quad (5.5)$$

where $M_{A_{iemu}} = M_{ssp}$ with $E_{demu} = 0$.

**Benchmark Selection**

To evaluate the cost of using the SSP architecture, several benchmarks were executed on the implementation described in Section 5.1. Several benchmarks suites including SPEC [55], EEMBC [126], MiBench [127], OpenBench [128], and LMBench [129] were evaluated for use in the benchmarking process. Ultimately, the MiBench and LMBench suites were selected for their frequent citation in related literature, broad range of applications, and free availability. Due to the large size of the combined suites a subset of the suites were selected for use in benchmarking the SSP. From LMBench, the microbenchmarks for context switch latency $lm\_lat$ and page fault latency $lm\_page$ were selected for investigating the effects of kernel additions needed for SSP operation. To assist in selecting an appropriate set of benchmarks that best represent the entire range of application characteristics, miss rates for the instruction cache, data cache, data-side TLB, and instruction-side TLB miss rates were measured for each benchmark using a SimpleScalar [33] configured to model the PPC405 processor. A comparison of cache and TLB miss rates obtained from the simulation model for each of the selected benchmarks is shown in Figures 5.7 and 5.8. Though one benchmark from each set was selected, an additional benchmark from the telecommunications group, namely GSM, was incorporated into the benchmark set to ensure proper coverage of the entire range of application characteristics. Given that the benchmarks in the MiBench suite
all exhibit low data cache miss rates (< 3%), an additional benchmark, STREAM, that exhibits a much higher data cache miss rate of ∼ 16% was added to the benchmark set. The overall set of selected benchmarks strive to cover the widest possible range of cache and TLB miss rates to ensure proper characterization of the SSP system.

Analytical Benchmark Performance Analysis

The predicted slow down for each benchmark can be calculated by combining the known properties of PPC405 processor used in the implementation, the benchmark simulation results, and the equations derived in Section 5.3.3. Using the characterization information from [120] of the IEMU implementation described in Section 5.1, the IEMU adds three additional cycles of latency to an instruction fetch operation. Additionally, the DDR memory on the ML310 incurs a 226ns average access penalty. Given that the implementation operates at 100Mhz, the IEMU operation adds an additional 30ns to each external memory operation. The ratio of instruction accesses to total memory access can be derived for each benchmark using the results from the simulation shown in Table 5.6. Table 5.6 shows the estimated slowdown for each of the selected benchmarks using Equation 5.4.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>I-Cache Acc.</th>
<th>D-Cache Acc.</th>
<th>I-Cache Misses</th>
<th>D-Cache Misses</th>
<th>IEMU</th>
<th>DEMU</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>856071769</td>
<td>197515901</td>
<td>25706193</td>
<td>753826</td>
<td>1.0032</td>
<td>1.0038</td>
</tr>
<tr>
<td>qsort</td>
<td>623881926</td>
<td>167711469</td>
<td>5251161</td>
<td>1332902</td>
<td>1.0009</td>
<td>1.0022</td>
</tr>
<tr>
<td>Rijndael</td>
<td>666824958</td>
<td>211118131</td>
<td>106178362</td>
<td>999873</td>
<td>1.0161</td>
<td>1.0170</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>256089396</td>
<td>89335449</td>
<td>110862</td>
<td>830418</td>
<td>1.0000</td>
<td>1.0020</td>
</tr>
<tr>
<td>GSM</td>
<td>1869477031</td>
<td>416703646</td>
<td>10259707</td>
<td>91378</td>
<td>1.0006</td>
<td>1.0006</td>
</tr>
<tr>
<td>JPEG</td>
<td>1011092224</td>
<td>28972223</td>
<td>20118</td>
<td>330027</td>
<td>1.0000</td>
<td>1.0003</td>
</tr>
<tr>
<td>Stringsearch</td>
<td>7252603</td>
<td>2244346</td>
<td>76656</td>
<td>1949</td>
<td>1.0011</td>
<td>1.0012</td>
</tr>
<tr>
<td>STREAM</td>
<td>776169120</td>
<td>432053777</td>
<td>7193</td>
<td>153500948</td>
<td>1.0000</td>
<td>1.0353</td>
</tr>
</tbody>
</table>

Table 5.6: Predicted slowdowns for the benchmark applications using the derived analytical models for both the IEMU and DEMU SSP systems
Figure 5.8: Simulated benchmark instruction and data cache miss statistics for the PPC405 processor

Benchmark Results

Using the context switch latency ($\text{lm\_con\_lat}$) and pagefault latency ($\text{lm\_pagefault\_lat}$) benchmarks from the LMBench suite, the context switch and pagefault latency characteristics of the SSP-modified kernel were compared to the unmodified base kernel. The results of the comparisons are shown in Figure 5.9 and Figure 5.10, respectively. The SSP kernel modifications approximately doubled the context switch latency from $\sim 80\mu$s to $\sim 160\mu$s. The SSP kernel additions have a similar affect upon pagefault latency, however the list-based implementation for searching the key mapping structure in the SSP kernel causes the pagefault latency to increase faster than the base kernel for large file sizes. A more efficient implementation would use a constant time searching structure such as a hash table.

Each of the selected benchmarks from the MiBench suite were executed and timed, using the Linux `time` command, on the implementation platform in five different scenarios.

1. **Base**: The base configuration does not include any SSP security-related hardware or SSP kernel modifications.

2. **SSP Kernel**: The SSP kernel configuration includes the SSP modified kernel but does

---

The files used for this benchmark were text files filled with a constant character.
not include any SSP security related hardware.

3. **SSP IEMU**: The SSP IEMU configurations includes the SSP modified kernel and an AES-enabled IEMU.

4. **SSP IEMU w/ I-Cache and D-Cache Flushing**: In addition to the features of the SSP IEMU configuration, the SSP modified kernel in this configuration forces cache invalidations/flushes on context switches between protected and unprotected applications.

5. **SSP DEMU**: The SSP DEMU configuration adds the DEMU (data-protection) to the SSP IEMU with cache flushing configuration.

Fifteen runs were executed for each benchmark within each of the scenarios. The motivation for choosing fifteen as the number of required runs stems from the results of a Student’s t-test with $\alpha = 0.05$ and $t_{crit} = 1.68$ shown in Tables 5.7 and 5.3.3. The resulting t-values indicate that, in general, the comparisons are significant. The majority of the values less than $t_{crit}$ result in comparing the SSP kernel to the SSP IEMU. Given the small amount of additional latency (3 cycles) and the measurement limits of the linux `time` command, small t-values are expected. Additionally, Table 5.9 and Table 5.10 show the mean and variance for each of the fifteen run sets across all benchmarks for the user (application) time and system (operating system) times, respectively. Similarly, Tables 5.11 and 5.12 show mean and variance statistics for the lengthened benchmarks.

The results of the benchmark runs, normalized to the base scenario to obtain slowdown, are shown in Figures 5.11 and 5.12. For each bar, the lower lighter half denotes the total system (kernel) time consumed by the benchmark. The remaining, darker, upper portion of the bar indicates the amount of user (application) time consumed during the benchmark run. Figure 5.11 shows the slowdown results for the original benchmarks whereas Figure 5.12 shows the slowdown results for lengthened versions of the benchmarks. Additionally, Tables 5.13 and 5.14 detail the average absolute (compared to the baseline) and relative (compared to the previous scenario) percent slowdown for the original and lengthened benchmarks, respectively. The lengthened benchmarks were necessary to ensure that variations caused by the use of a multitasking operating system were not causing errant results. On average, the SSP kernel modifications caused a 26.5% increase in overall run time. As calculated from the analytical model, the average runtime increase resulting from the addition of the IEMU was negligible as compared to the previous SSP kernel case. The additional cache flushing between context switches contributed an additional 7.5% to the total execution time beyond the IEMU-enabled configuration. Finally, as expected the addition of the DEMU hardware added a negligible amount of overhead as compared to the cache flushing case. As compared to the base system, instruction protection increased execution times by an average of 26.5%. As compared to the base, data protection increased execution times by an average of 55.6%.

---

6If cache sizes were much larger than the 16KB caches of the PPC405, the slowdown would likely be much worse. Exploration of the effects of flushing large caches is a possible area for future work.
The discrepancies between the analytical model and the benchmark results stems from the inclusion of an operating system which is not represented in the analytical model. As a general trend, the results in Figures 5.11 and 5.12 indicate that the slowdown of application when used on the SSP platform is closely related to the proportion of system time. More specifically, the greater the system time of an application, the greater the slowdown caused by the SSP hardware. A larger system time increases slowdown by incurring more frequent context switches. Context switches pollute caches and TLBs, resulting more external memory references and operating system operations.

5.4 Design Principles

Using the combined results of the security analysis and benchmarks, design principles related to the set of processor interfaces and processor features required to support the external addition of software security can be established. Using these design principles and feature requirements, future processors can be designed with an appropriate hardware-based Security Management Interface (SMI) and exclude features that sacrifice security. By using a standard security interface, designers of secure systems could have the added benefit of “plug-and-play” security.
<table>
<thead>
<tr>
<th></th>
<th>FFT</th>
<th>JPEG</th>
<th>RIJNDAEL</th>
<th>DIJSKTRA</th>
<th>GSM</th>
<th>JPEG</th>
<th>STR SRCH</th>
<th>STREAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Kernel - SSP Kernel</td>
<td>16.15</td>
<td>13.03</td>
<td>3.72</td>
<td>3.92</td>
<td>10.45</td>
<td>13.36</td>
<td>128.93</td>
<td>57.61</td>
</tr>
<tr>
<td>SSP Kernel - SSP IEMU</td>
<td>1.44</td>
<td>0.58</td>
<td>1.64</td>
<td>0.01</td>
<td>0.17</td>
<td>0.67</td>
<td>7.14</td>
<td>0.73</td>
</tr>
<tr>
<td>SSP IEMU - SSP IEMU w/ cache flush</td>
<td>9.09</td>
<td>3.52</td>
<td>3.52</td>
<td>2.73</td>
<td>9.34</td>
<td>6.51</td>
<td>1.13</td>
<td>26.70</td>
</tr>
<tr>
<td>SSP IEMU w/ cache flush - SSP DEMU</td>
<td>2.39</td>
<td>9.46</td>
<td>2.84</td>
<td>3.16</td>
<td>0.60</td>
<td>7.96</td>
<td>31.32</td>
<td>82.48</td>
</tr>
<tr>
<td>SSP DEMU - SSP DEMU w/ cache flush</td>
<td>12.31</td>
<td>5.40</td>
<td>2.55</td>
<td>0.45</td>
<td>15.06</td>
<td>7.23</td>
<td>0.87</td>
<td>18.17</td>
</tr>
</tbody>
</table>

Table 5.7: Student’s t-test values for each of the comparison cases across all benchmarks
<table>
<thead>
<tr>
<th>Comparison Case</th>
<th>FFT</th>
<th>JPEG</th>
<th>RIJNDAEL</th>
<th>DIJSKTRA</th>
<th>GSM</th>
<th>JPEG</th>
<th>STR SRCH</th>
<th>STREAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Kernel - SSP Kernel</td>
<td>94.62</td>
<td>121.71</td>
<td>14.01</td>
<td>106.98</td>
<td>87.10</td>
<td>1240.20</td>
<td>102.89</td>
<td>3.42</td>
</tr>
<tr>
<td>SSP Kernel - SSP IEMU</td>
<td>3.80</td>
<td>12.10</td>
<td>12.92</td>
<td>24.84</td>
<td>5.16</td>
<td>9.93</td>
<td>3.18</td>
<td>0.24</td>
</tr>
<tr>
<td>SSP IEMU - SSP IEMU w/ cache flush</td>
<td>69.34</td>
<td>30.24</td>
<td>2.13</td>
<td>14.26</td>
<td>50.80</td>
<td>204.26</td>
<td>6.68</td>
<td>1.54</td>
</tr>
<tr>
<td>SSP IEMU w/ cache flush - SSP DEMU</td>
<td>0.82</td>
<td>118.98</td>
<td>17.05</td>
<td>31.57</td>
<td>43.23</td>
<td>454.08</td>
<td>44.24</td>
<td>5.87</td>
</tr>
<tr>
<td>SSP DEMU - SSP DEMU w/cache flush</td>
<td>99.10</td>
<td>66.18</td>
<td>22.88</td>
<td>5.05</td>
<td>48.95</td>
<td>320.16</td>
<td>3.15</td>
<td>1.32</td>
</tr>
</tbody>
</table>

Table 5.8: Student’s t-test values for each of the comparison cases across all benchmarks (lengthened benchmarks)
<table>
<thead>
<tr>
<th></th>
<th>FFT</th>
<th>QSORT</th>
<th>RIJNDAEL</th>
<th>DIJKSTRA</th>
<th>GSM</th>
<th>JPEG</th>
<th>STR SCH</th>
<th>STREAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>6.73</td>
<td>1.08</td>
<td>7.31</td>
<td>2.24</td>
<td>8.61</td>
<td>1.05</td>
<td>2.70</td>
<td>35.59</td>
</tr>
<tr>
<td>SSP Kernel</td>
<td>6.85</td>
<td>1.15</td>
<td>7.64</td>
<td>2.37</td>
<td>8.83</td>
<td>1.19</td>
<td>4.34</td>
<td>36.39</td>
</tr>
<tr>
<td>SSP IEMU</td>
<td>6.86</td>
<td>1.13</td>
<td>6.83</td>
<td>2.40</td>
<td>8.91</td>
<td>1.18</td>
<td>4.34</td>
<td>36.42</td>
</tr>
<tr>
<td>SSP IEMU w/ Cache flush</td>
<td>7.04</td>
<td>1.17</td>
<td>7.47</td>
<td>2.68</td>
<td>9.04</td>
<td>1.17</td>
<td>4.37</td>
<td>37.36</td>
</tr>
<tr>
<td>SSP DEMU</td>
<td>6.94</td>
<td>1.40</td>
<td>8.87</td>
<td>3.00</td>
<td>9.03</td>
<td>1.68</td>
<td>5.04</td>
<td>40.94</td>
</tr>
<tr>
<td>SSP DEMU w/ Cache flush</td>
<td>7.17</td>
<td>1.48</td>
<td>8.78</td>
<td>3.02</td>
<td>9.34</td>
<td>1.72</td>
<td>5.06</td>
<td>39.74</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>FFT</th>
<th>QSORT</th>
<th>RIJNDAEL</th>
<th>DIJKSTRA</th>
<th>GSM</th>
<th>JPEG</th>
<th>STR SCH</th>
<th>STREAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>0.0007</td>
<td>0.0005</td>
<td>3.6181</td>
<td>0.0578</td>
<td>0.0075</td>
<td>0.0031</td>
<td>0.0111</td>
<td>0.0033</td>
</tr>
<tr>
<td>SSP Kernel</td>
<td>0.0011</td>
<td>0.0008</td>
<td>2.1963</td>
<td>0.0780</td>
<td>0.0104</td>
<td>0.0077</td>
<td>0.0190</td>
<td>0.0108</td>
</tr>
<tr>
<td>SSP IEMU</td>
<td>0.0032</td>
<td>0.0011</td>
<td>1.6223</td>
<td>0.1149</td>
<td>0.0079</td>
<td>0.0034</td>
<td>0.0141</td>
<td>0.0043</td>
</tr>
<tr>
<td>SSP IEMU w/ Cache flush</td>
<td>0.0009</td>
<td>0.0012</td>
<td>2.6507</td>
<td>0.1268</td>
<td>0.0037</td>
<td>0.0039</td>
<td>0.0221</td>
<td>0.0052</td>
</tr>
<tr>
<td>SSP DEMU</td>
<td>0.0023</td>
<td>0.0011</td>
<td>1.7683</td>
<td>0.2325</td>
<td>0.0059</td>
<td>0.0020</td>
<td>0.0197</td>
<td>0.0124</td>
</tr>
<tr>
<td>SSP DEMU w/ Cache flush</td>
<td>0.0025</td>
<td>0.0013</td>
<td>2.0046</td>
<td>0.3714</td>
<td>0.0032</td>
<td>0.0047</td>
<td>0.0570</td>
<td>0.0042</td>
</tr>
</tbody>
</table>

Table 5.9: User time mean and variance for the fifteen runs of the original benchmarks
### System Time Mean

<table>
<thead>
<tr>
<th></th>
<th>FFT</th>
<th>QSORT</th>
<th>RIJNDAEL</th>
<th>DIJKSTRA</th>
<th>GSM</th>
<th>JPEG</th>
<th>STR SCH</th>
<th>STREAM</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Base</strong></td>
<td>0.2950</td>
<td>0.1513</td>
<td>5.2175</td>
<td>0.1381</td>
<td>1.2081</td>
<td>0.7950</td>
<td>6.5875</td>
<td>0.6294</td>
</tr>
<tr>
<td><strong>SSP Kernel</strong></td>
<td>0.5327</td>
<td>0.3720</td>
<td>7.4887</td>
<td>0.3567</td>
<td>1.8040</td>
<td>1.2540</td>
<td>12.5090</td>
<td>1.6100</td>
</tr>
<tr>
<td><strong>SSP IEMU</strong></td>
<td>0.5480</td>
<td>0.3787</td>
<td>7.3680</td>
<td>0.3240</td>
<td>1.6980</td>
<td>1.2153</td>
<td>11.9390</td>
<td>1.6313</td>
</tr>
<tr>
<td><strong>SSP IEMU w/ Cache flush</strong></td>
<td>0.6200</td>
<td>0.4160</td>
<td>8.3980</td>
<td>0.3860</td>
<td>2.0480</td>
<td>1.4293</td>
<td>11.8210</td>
<td>1.6100</td>
</tr>
<tr>
<td><strong>SSP DEMU</strong></td>
<td>0.8200</td>
<td>0.5100</td>
<td>8.8120</td>
<td>0.4793</td>
<td>2.1580</td>
<td>1.5753</td>
<td>13.9060</td>
<td>2.0400</td>
</tr>
<tr>
<td><strong>SSP DEMU w/ Cache flush</strong></td>
<td>0.8700</td>
<td>0.5267</td>
<td>10.1387</td>
<td>0.5387</td>
<td>2.3947</td>
<td>1.7307</td>
<td>13.9640</td>
<td>1.9775</td>
</tr>
</tbody>
</table>

### System Time Variance

<table>
<thead>
<tr>
<th></th>
<th>FFT</th>
<th>QSORT</th>
<th>RIJNDAEL</th>
<th>DIJKSTRA</th>
<th>GSM</th>
<th>JPEG</th>
<th>STR SCH</th>
<th>STREAM</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Base</strong></td>
<td>0.0031</td>
<td>0.0034</td>
<td>0.2001</td>
<td>0.0024</td>
<td>0.0335</td>
<td>0.0113</td>
<td>0.0147</td>
<td>0.0031</td>
</tr>
<tr>
<td><strong>SSP Kernel</strong></td>
<td>0.0019</td>
<td>0.0012</td>
<td>0.3465</td>
<td>0.0062</td>
<td>0.1839</td>
<td>0.0313</td>
<td>0.0315</td>
<td>0.0250</td>
</tr>
<tr>
<td><strong>SSP IEMU</strong></td>
<td>0.0028</td>
<td>0.0023</td>
<td>0.0734</td>
<td>0.0038</td>
<td>0.0050</td>
<td>0.0031</td>
<td>0.0282</td>
<td>0.0046</td>
</tr>
<tr>
<td><strong>SSP IEMU w/ Cache flush</strong></td>
<td>0.0092</td>
<td>0.0083</td>
<td>0.5839</td>
<td>0.0048</td>
<td>0.2326</td>
<td>0.0506</td>
<td>0.0378</td>
<td>0.0124</td>
</tr>
<tr>
<td><strong>SSP DEMU</strong></td>
<td>0.0020</td>
<td>0.0013</td>
<td>0.0455</td>
<td>0.0033</td>
<td>0.0044</td>
<td>0.0032</td>
<td>0.0430</td>
<td>0.0261</td>
</tr>
<tr>
<td><strong>SSP DEMU w/ Cache flush</strong></td>
<td>0.0024</td>
<td>0.0015</td>
<td>0.1152</td>
<td>0.0042</td>
<td>0.0053</td>
<td>0.0051</td>
<td>0.0749</td>
<td>0.0070</td>
</tr>
</tbody>
</table>

Table 5.10: System time mean and variance for the fifteen runs of the original benchmarks
<table>
<thead>
<tr>
<th>User Time Mean</th>
<th>FFT</th>
<th>QSORT</th>
<th>RIJNDAEL</th>
<th>DIJKSTRA</th>
<th>GSM</th>
<th>JPEG</th>
<th>STR SCH</th>
<th>STREAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>61.36</td>
<td>19.87</td>
<td>23.46</td>
<td>21.97</td>
<td>51.55</td>
<td>43.18</td>
<td>10.49</td>
<td>1.77</td>
</tr>
<tr>
<td>SSP Kernel</td>
<td>62.67</td>
<td>20.82</td>
<td>22.70</td>
<td>30.88</td>
<td>52.88</td>
<td>66.47</td>
<td>14.06</td>
<td>1.82</td>
</tr>
<tr>
<td>SSP IEMU</td>
<td>62.79</td>
<td>21.14</td>
<td>29.89</td>
<td>28.29</td>
<td>52.45</td>
<td>65.35</td>
<td>14.02</td>
<td>1.82</td>
</tr>
<tr>
<td>SSP IEMU w/ Cache flush</td>
<td>64.46</td>
<td>21.63</td>
<td>26.99</td>
<td>29.98</td>
<td>53.66</td>
<td>66.67</td>
<td>14.46</td>
<td>1.86</td>
</tr>
<tr>
<td>SSP DEMU</td>
<td>64.1387</td>
<td>25.2593</td>
<td>31.3600</td>
<td>33.9953</td>
<td>54.5027</td>
<td>98.7127</td>
<td>18.7040</td>
<td>2.05</td>
</tr>
<tr>
<td>SSP DEMU w/ Cache flush</td>
<td>66.1273</td>
<td>26.6807</td>
<td>37.8287</td>
<td>35.2453</td>
<td>55.3993</td>
<td>101.9733</td>
<td>18.0160</td>
<td>1.99</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>User Time Variance</th>
<th>FFT</th>
<th>QSORT</th>
<th>RIJNDAEL</th>
<th>DIJKSTRA</th>
<th>GSM</th>
<th>JPEG</th>
<th>STR SCH</th>
<th>STREAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>0.03</td>
<td>0.02</td>
<td>14.00</td>
<td>0.05</td>
<td>0.52</td>
<td>0.26</td>
<td>0.13</td>
<td>0.0033</td>
</tr>
<tr>
<td>SSP Kernel</td>
<td>0.06</td>
<td>0.25</td>
<td>3.20</td>
<td>82.67</td>
<td>0.35</td>
<td>3.20</td>
<td>1.58</td>
<td>0.0108</td>
</tr>
<tr>
<td>SSP IEMU</td>
<td>0.04</td>
<td>0.32</td>
<td>50.51</td>
<td>71.26</td>
<td>0.30</td>
<td>2.27</td>
<td>2.19</td>
<td>0.0043</td>
</tr>
<tr>
<td>SSP IEMU w/ Cache flush</td>
<td>0.05</td>
<td>0.40</td>
<td>26.21</td>
<td>72.40</td>
<td>0.11</td>
<td>1.46</td>
<td>0.80</td>
<td>0.0052</td>
</tr>
<tr>
<td>SSP DEMU</td>
<td>0.50</td>
<td>0.39</td>
<td>11.19</td>
<td>66.26</td>
<td>0.32</td>
<td>17.91</td>
<td>5.02</td>
<td>0.0124</td>
</tr>
<tr>
<td>SSP DEMU w/ Cache flush</td>
<td>0.23</td>
<td>0.26</td>
<td>34.23</td>
<td>132.35</td>
<td>0.07</td>
<td>9.26</td>
<td>1.17</td>
<td>0.0042</td>
</tr>
</tbody>
</table>

Table 5.11: User time mean and variance for the fifteen runs of the lengthened benchmarks
<table>
<thead>
<tr>
<th>System Time Mean</th>
<th>FFT</th>
<th>QSORT</th>
<th>RIJNDAEL</th>
<th>DIJKSTRA</th>
<th>GSM</th>
<th>JPEG</th>
<th>STR SCH</th>
<th>STREAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>1.32</td>
<td>0.89</td>
<td>21.07</td>
<td>0.13</td>
<td>6.56</td>
<td>29.59</td>
<td>0.69</td>
<td>0.05</td>
</tr>
<tr>
<td>SSP Kernel</td>
<td>2.10</td>
<td>2.65</td>
<td>31.60</td>
<td>0.69</td>
<td>12.25</td>
<td>61.34</td>
<td>3.16</td>
<td>0.10</td>
</tr>
<tr>
<td>SSP IEMU</td>
<td>2.05</td>
<td>2.52</td>
<td>31.76</td>
<td>0.68</td>
<td>11.59</td>
<td>61.76</td>
<td>3.45</td>
<td>0.11</td>
</tr>
<tr>
<td>SSP IEMU w/ Cache flush</td>
<td>2.34</td>
<td>2.69</td>
<td>35.66</td>
<td>0.76</td>
<td>12.99</td>
<td>67.08</td>
<td>3.51</td>
<td>0.13</td>
</tr>
<tr>
<td>SSP DEMU</td>
<td>2.62</td>
<td>3.16</td>
<td>42.22</td>
<td>0.91</td>
<td>19.56</td>
<td>72.46</td>
<td>3.15</td>
<td>0.23</td>
</tr>
<tr>
<td>SSP DEMU w/ Cache flush</td>
<td>2.98</td>
<td>2.91</td>
<td>46.84</td>
<td>0.55</td>
<td>20.45</td>
<td>77.58</td>
<td>3.55</td>
<td>0.19</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>System Time Variance</th>
<th>FFT</th>
<th>QSORT</th>
<th>RIJNDAEL</th>
<th>DIJKSTRA</th>
<th>GSM</th>
<th>JPEG</th>
<th>STR SCH</th>
<th>STREAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>0.0039</td>
<td>0.0067</td>
<td>0.5410</td>
<td>0.0007</td>
<td>0.0533</td>
<td>0.3049</td>
<td>0.0199</td>
<td>0.003</td>
</tr>
<tr>
<td>SSP Kernel</td>
<td>0.0030</td>
<td>0.0263</td>
<td>0.4652</td>
<td>0.0041</td>
<td>1.1505</td>
<td>1.1048</td>
<td>1.5433</td>
<td>0.01</td>
</tr>
<tr>
<td>SSP IEMU</td>
<td>0.0063</td>
<td>0.0379</td>
<td>4.2860</td>
<td>0.0011</td>
<td>3.6555</td>
<td>0.3110</td>
<td>1.9547</td>
<td>0.004</td>
</tr>
<tr>
<td>SSP IEMU w/ Cache flush</td>
<td>0.0200</td>
<td>0.0265</td>
<td>3.4066</td>
<td>0.0018</td>
<td>3.9079</td>
<td>0.3799</td>
<td>0.7766</td>
<td>0.005</td>
</tr>
<tr>
<td>SSP DEMU</td>
<td>0.0220</td>
<td>0.1104</td>
<td>14.0383</td>
<td>0.0053</td>
<td>0.1809</td>
<td>1.5488</td>
<td>2.5425</td>
<td>0.01</td>
</tr>
<tr>
<td>SSP DEMU w/ Cache flush</td>
<td>0.0128</td>
<td>0.0831</td>
<td>0.9995</td>
<td>0.0099</td>
<td>0.1199</td>
<td>0.4743</td>
<td>1.0518</td>
<td>0.004</td>
</tr>
</tbody>
</table>

Table 5.12: System time mean and variance for the fifteen runs of the lengthened benchmarks
Figure 5.10: Page fault latency as a function of filesize

Table 5.13: Absolute (percent from base) and relative (percent from previous) for the unmodified benchmarks. Note that the majority of the slowdown results from the kernel additions. Also, instruction and data cache flushing on context switches only contributes 7.5% slowdown.
Table 5.14: Absolute (percent from base) and relative (percent from previous) for the lengthened benchmarks. Note that the majority of the slowdown results from the kernel additions. Also, instruction and data cache flushing on context switches is smaller than the original benchmark case.
Figure 5.12: Slowdowns for each of the selected benchmark for different system configurations (lengthened benchmarks)
5.4.1 Processor Interface Requirements for the Augmentation Methodology

1. *Harvard Architecture*: A target processor must have separate instruction and data paths to cryptographic separation or instructions and data.

2. *MMU Interface*: A target processor must have an MMU interface that provides direct access to the memory translation information.

3. *Cache Control*: The MMU of a target processor must provide an interface for hardware-initiated cache flushing and/or invalidation.

4. *Register Interface*: If register leakage is a concern, a target processor must have a hardware-controlled interface to monitor internal register state.

5.4.2 Processor Feature Requirements

1. *ISA Requirements*: An essential instruction cannot move information from the instruction cache(s) to registers.
Chapter 6

Conclusion

6.1 Summary

The primary goal and accomplishment of this research is enabling software security features, without trusting software, by externally augmenting conventional computing devices with additional hardware. This goal was achieved without a large impact upon performance and without significant changes to the software development flow. Also resulting from this research were a generic set of design principles in the form of processor characteristics needed to support the external augmentation methodology and a graph-based model for analyzing security properties of hardware-based security systems. The feasibility of the architecture was demonstrated by the implementation. Table 6.1 compares the assumption set and corresponding implications of the SSP architectures with those of existing trusted computing platforms. The SSP differs from previous TCP architectures in that its assumption set is much smaller, resulting in improved security and in some scenarios increased flexibility. The three areas of contribution outlined in Chapter 1 can be decomposed into more specific contributions as listed below.

6.2 Contributions Revisited

The major contributions stemming from this research are as follows:

- A system was designed that can provide hardware-based software security without redesigning the processor and without trusting an operating system. The architecture, presented in Chapter 4, resides beyond the processor boundary, results in reasonable performance impact, and does not require significant changes to the software development flow.
### Assumption vs. Implication

<table>
<thead>
<tr>
<th>Assumption</th>
<th>Implication</th>
<th>XOM</th>
<th>AEGIS</th>
<th>SSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating system can be trusted</td>
<td>The operating system could be erroneous or malicious</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Applications can be trusted</td>
<td>Erroneous or malicious may perform an inadvertent downgrade</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Same key for all applications</td>
<td>OS must differentiate applications</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Single, hardware key for all applications</td>
<td>Single point of failure</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Application-level key granularity</td>
<td>Removes the possibility of shared libraries</td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Table 6.1: Comparison of assumptions between the existing TCPs XOM and AEGIS with the SSP architecture
• A set of processor interfaces required for the external augmentation approach to hardware-based software security were defined in Chapter 5. Future processors can utilize the set of interface requirements to create a Security Management Interface enabling “plug and play” security. By using a generic interface for security functions, verification effort need not be wasted on each processor-specific security implementation. Instead, verification efforts can be focused on the added security hardware. The ability to mix and match processors with security hardware may also increase the breadth and capabilities of security devices.

• A set of processor feature requirements for the external augmentation approach were defined in Chapter 5. The feature set can be used to evaluate existing processors for compatibility with the external augmentation methodology. Also, processor designers can incorporate (or not incorporate in the case of undesirable features) the feature set into future processors to enable the use of the external augmentation methodology.

• The cryptographic separation of instruction and data datapaths is proposed and is a feature of the SSP architecture as discussed in Chapter 4. The cryptographic separation of datapaths, in addition to creating a second cryptographic problem for an attacker, prevents the manipulation of instructions (reading or writing) by the data-side processor interface. More specifically, instructions cannot be retrieved as data and data cannot be executed (a common vector for attack in modern systems).

• A new method of integrity checking that reduces latency, allows flexible protection levels, and reduces possible attacks was proposed in Chapter 4. The new integrity verification mechanism, based on CRC, leverages the unique situation created by a trusted computing platform to remove the need for a cryptographic hash algorithm given sufficient on-chip storage space. Additionally, the mechanism provides inherent freshness assurance by operating on the non-transformed data. Finally, a security analysis of the CRC-based method performed in Chapter 5 shows that the CRC can provide substantial security at a low cost.

• The flexibility of secure applications was improved, as a direct result of the SSP design outlined in Chapter 5 through per-page rather than per-application key assignment. Using per-page keys, different portions of an application can have different types of cryptographic protection facilitating security differentiation. As a result, per-page keys enables cryptographic separation of datapaths. Finally, the use of per-page keys supports the closed-source software distribution model, allowing object files to be independently protected.

• An alternative and easy applicable graph-based model for use in analyzing hardware-based security systems, discussed and formalized in Chapter 3, was designed and used to verify the security of a specific instance of the SSP architecture in Chapter 5. The graph model exhibits sufficient flexibility to apply to a variety of architectures as
demonstrated by its application to two existing hardware-based security architectures, namely XOM [5] and AEGIS [6].

- Demonstrated by an implementation (discussed in Chapter 5), the augmentation approach to hardware-based software protection was shown to be practically realizable and well understood. A working implementation indicates that the SSP architecture is immediately usable. Additionally, the measurement and benchmark results outlined in Chapter 5 explain and account for each of the spatial and temporal costs of the SSP. Also, the result support the assertion that the performance and overhead of the system are reasonable.

- The SSP architecture implementation was evaluated under a real, extant multitasking operating system. Enabled by the physical implementation, evaluation under a real operating system allowed for a much more accurate assessment of overall system performance as well as the operating system’s impact upon performance. The benchmark results in Chapter 5 indicate that even in a support role, the operating system contributes significantly to the overall impact upon performance. Previous efforts, that used only simulation for characterization neglected this important aspect of system characterization.

### 6.3 Future Work

Together the design principles, architectural foundation, and implementation established by this work provide ample opportunity for future exploration. The first foreseeable area for future work is creating a standard for specifying and handling encrypted I/O streams, including mixed (encrypted and plaintext) I/O. To enable the use of multiple types of I/O, the operating system must be able to cooperate with the SSP hardware to identify and appropriately map the I/O into memory. The second area of exploration is the addition of higher-level caches to the hierarchy. In addition to mitigating the effects of latency added to the memory hierarchy by the cryptographic operations, certain cache configurations may also reduce the observability of addresses on the memory bus similar to [10]. A third avenue is extending the SSP architecture for use in the secure multiprocessing and secure cluster computing domains. Finally, adapting the SSP to allow for an encrypted and/or integrity verified operating system has the potential to further complicate attacks on the SSP system.
Bibliography


[23] M. Kuhn, “The trustno 1 cryptoprocessor concept.”


C. D. Patterson, 2005, private communication.


Appendix A

Theorem Proving Example: An AND gate

The following example uses the theorem proving analysis method to verify the functional operation of an AND gate. This specific example was taken from lecture notes for a formal verification course at Concordia University taught by Dr. Sofiene Tahar [95]. The concepts were originally formulated and presented by Mike Gordon [94] and are reproduced here to clarify the concepts outlined in this document. It should be noted that this example is NOT the original work of this author. Proof related statements are presented in boldface.

AND Specification:

\[ \text{AND}_\text{SPEC}(i_1, i_2, out) := out = i_1 \land i_2 \]

Primitive Specification:

\[ \text{NAND}_\text{SPEC}(i_1, i_2, out) := out = \neg(i_1 \land i_2) \]
\[ \text{NOT}_\text{SPEC}(i, out) := out = \neg i \land i_2 \]

AND Implementation:

\[ \text{AND}_\text{IMPL}(i_1, i_2, out) := out = \exists x.\text{NAND}(i_1, i_2, x) \land \text{NOT}(x, out) \]

Desired proof:

\[ \forall i_1, i_2, out.\text{AND}_\text{IMPL}(i_1, i_2, out) \implies \text{AND}_\text{SPEC}(i_1, i_2, out) \]

Proof:

\[ \text{AND}_\text{IMP}(i_1, i_2, out) \{\text{from above circuit diagram}\} \]
\[ \vdash \exists x.\text{NAND}(i_1, i_2, out) \land \text{NOT}(x, out) \{\text{by def of AND impl}\} \]
\[ \vdash \text{NAND}(i_1, i_2, out) \land \text{NOT}(x, out) \{\text{strip off } \exists x\} \]
\[ \vdash \text{NAND}(i_1, i_2, out) \{\text{left conjunct of line 3}\} \]
\[ \vdash x = \neg(i_1 \land i_2) \text{ \{by def of NAND\}} \]
\[ \vdash \text{NOT}(x, out) \text{ \{right conjunct of line 3\}} \]
\[ \vdash out = \neg x \text{ \{by def of NOT\}} \]
\[ \vdash out = \neg(\neg(i_1 \land i_2)) \text{ \{substitution , line 5 into 7\}} \]
\[ \vdash out = (i_1, i_2) \text{ \{simplify, \neg t=t\}} \]
\[ \vdash \text{AND}(i_1, i_2, out) \text{ \{by def of AND spec\}} \]
\[ \vdash \text{AND.IMPL}(i_1, i_2, out) \implies \text{AND.SPEC}(i_1, i_2, out) \]
Appendix B

Model Checking Example: Mutual Exclusion

The following example uses the model checking analysis method to verify the functional operation of mutual exclusion between two processes. This example was originally presented as a tutorial on the Carnegie Mellon University (CMU) [97] and is reproduced here to clarify the concepts outlined in this document. It should be noted that this example is NOT the original work of this author. The example is formulated in the Symbolic Model Verifier (SMV) [98] language developed at CMU. SMV code is shown in boldface.

The module main is the top level instantiation of two processes which can exhibit the states shown in s0 and s1, share a mutex variable turn, and are initialized with the appropriate values for the read variable.

MODULE main
VAR
s0: {noncritical, trying, critical, ready};
s1: {noncritical, trying, critical, ready};
turn: boolean;
pr0: process prc(s0, s1, turn, 0);
pr1: process prc(s1, s0, turn, 1);

The ASSIGN statement initializes the value of turn while the FAIRNESS keyword provides the constraint that neither process can stay in the critical state forever.

ASSIGN init(turn) := 0;

FAIRNESS !(s0 = critical)
FAIRNESS !(s1 = critical)
This module essentially defines the state machine for each of the processes instantiated above.

\[
\text{MODULE prc(state0, state1, turn, turn0)}
\]

\[
\text{ASSIGN}
\]

\[
\text{init(state0) := noncritical;}
\]

\[
\text{next(state0) :=}
\]

\[
\text{case}
\]

\[
\begin{align*}
\text{(state0 = noncritical) : \{trying,noncritical\};} \\
\text{(state0 = trying) \& ((state1 = noncritical) \mid (state1 = trying) \mid (state1 = ready)): ready;} \\
\text{(state0 = ready): critical;} \\
\text{(state0 = trying) \& (state1 = trying) \& (turn = turn0): critical;} \\
\text{(state0 = critical) : \{critical,noncritical\};}
\end{align*}
\]

\[
1: \text{state0;}
\]

\[
\text{esac;}
\]

\[
\text{next(turn) :=}
\]

\[
\text{case}
\]

\[
\begin{align*}
\text{turn = turn0 \& state0 = critical: !turn;}
\end{align*}
\]

\[
1: \text{turn;}
\]

\[
\text{esac;}
\]

\[
\text{FAIRNESS running}
\]

The SPEC statements define the property to be verified by the model checking utility using temporal logic. The A and G identifiers at the beginning of the statement are standard temporal logic operators indicating that for ”All sequences” it is ”Globally” true that both process cannot be in the critical state at the state time (i.e.mutual exclusion)

\[
\text{SPEC AG((s0 = critical) \rightarrow ! (s1 = critical))}
\]

\[
\text{SPEC AG((s1 = critical) \rightarrow ! (s0 = critical))}
\]

This set of SPEC statements specifies another property about the system, namely that over all sequences (the Aidentifierr from above) that at some point in the Future given that a state is trying to enter its critical section that it will eventually be allowed enter the critical section.

\[
\text{SPEC AG((s0 = trying) \rightarrow AF(s0 = critical))}
\]

\[
\text{SPEC AG((s1 = trying) \rightarrow AF(s1 = critical))}
\]

After executing the model in the SMV environment, it is shown that the model presented above is not sufficient to guarantee mutual exclusion. As with most model checking utilities
a counter example is provided. The output is as follows:

- specification $\text{AG} (s0 = \text{critical} \rightarrow !s1 = \text{critical})$ is false
- as demonstrated by the following execution sequence

state 1.1:
s0 = noncritical
s1 = noncritical
turn = 0
state 1.2:
[executing process pr1]
state 1.3:
[executing process pr1]
s1 = trying
state 1.4:
[executing process pr0]
s1 = ready
state 1.5:
[executing process pr0]
s0 = trying
state 1.6:
[executing process pr1]
s0 = ready
state 1.7:
[executing critical]
s1 = critical
state 1.8:
s0 = critical
Appendix C

Instruction-side Algorithm Selection (Additional comments)

In contrast to countermode, blockmode operation shown in Figure C.1 introduces a larger latency (~10 cycles) but does not exhibit bitwise linearity. While tampering with specific bits in a blockmode encrypted block is more difficult, blockmode encryption does not remove the need for integrity protection as detailed in Section 4.3.7. Blockmode encryption enables additional variations on algorithm operation such as chaining block cipher (CBC) which uses the result of a previous encryption as the initialization vector (IV) of the next encryption [110]. CBC style encryption imparts the requirement of sequential decryption but is more secure than the electronic codebook (ECB) mode of operation. Although execution on a modern processor is in general, sequential, typical control flow constructs of instruction set architectures (ISA) such as jumps and branches cause execution to deviate from the completely sequential model. As a result, encryption modes such as chaining block cipher (CBC) that force serial calculation by requiring information from previous operations are not compatible with the instruction-protection environment.

![Diagram of AES blockmode operation](image)

Figure C.1: AES blockmode operation
Appendix D

Cryptographic Hash Algorithms

Cryptographic hash algorithms typically used for integrity verification such as MD5 [110] and SHA1 [110] exhibit extremely high per-message latencies (∼160 cycles per 512 bit message) and large hash digests (∼160 bits) [30]. The combination of additional memory access latency and increased memory bandwidth overhead associated with retrieving large hash digests have a significant impact on performance. Additionally, many cryptographic hashing methods require the use of a key. Because the use of the same key for encryption and hash calculation is considered poor security practice [110], additional keys are required when using cryptographic hash algorithms. Furthermore, constraints on generic composition, requiring that encryption be performed prior to the hash calculation (encrypt-then-MAC) [112], limits the ability to parallelize the encryption and integrity checking operations. Several methods have been proposed in an effort to overcome the large latency of cryptographic hash algorithms such as Log Hash (LHash) trees [30] and hash caches [6].
Appendix E

Integrity-Aware Encryption and Galois Countermode use in the IEMU

Integrity aware encryption is a branch of cryptographic research that attempts to combine the encryption and integrity checking (hashing) operations with little performance impact beyond standard encryption. Recent efforts in the area aim to extend the operation modes of standard encryption algorithms, such as AES. Several efforts include the Integrity Aware Parallelizable Mode (IAPM) [130], Offset Codebook Mode (OCB) [131], EAX [132], Counter with CBC-MAC mode (CCM) [133], and Carter-Wegman Countermode (CWC) [134]. Despite the promising concept, these algorithms exhibit several drawbacks. First, the use of nearly all of the mentioned authenticated encryption modes of operations is hindered by licensing fees and patent restrictions. Second, the algorithms that do not have license issues (and even for several that do) require multiple encryption passes resulting in very high latency. Finally, only a subset of the algorithms support the ability to authenticate additional data beyond the message itself. Generally, the algorithms that do support additional data tend to focus on accommodating the structure of network transmissions. There is however one particular mode of operation, Galois Counter Mode (GCM) [114], that has the correct mixture of features to be a viable solution for low-latency integrity verification of instructions.

E.1 GCM in the IEMU

Because instructions are fetched as cachelines it is likely that, that at most, only a few (∼2-4) 128 bit blocks would be processed per GCM operation. The resulting hash security of GCM falls between the acceptable range of 126-122 bits. Using the additional data feature of the authentication mechanism, information such as a page relative address could be used to prevent the replay of arbitrary blocks of instruction data. Additionally, the requirement of IV information fits easily into the current instruction side implementation and would be
supported similarly to the counters used in standard countermode operation. In the case of multiple cycles for the Galois field multiplication, GCM mode authentication still allows for at least an order of magnitude faster calculation than even fully-pipelined SHA1 or MD5 hashes.
Appendix F

Downgrading Information

For particular types of operating system cooperation or for data sharing, the ability to downgrade data from a secure state to a less secure state may be useful and/or required. To achieve downgrade on the SSP system, some portion of the application must be trusted to provide information pertaining to the desired protection level. Trusting the application is necessary because information does not exist describing the protection method, instead the hardware defaults to the protection level defined by the dynamic data key. As a result, the software may have the option of overriding this default. Two distinct types of downgrade are possible, explicit and implicit.

An implicit downgrade is achieved by adding a flag to the key sets indicating downgrade capability. Any executable code encrypted with a key flagged with the downgrade indicator may select a secondary (usually passthrough) transformation mechanism to apply to data. Because all data within a page must be protected using the same key, the resulting data must be written to a different portion of memory reserved for the downgrade operation. Interaction with the DEMU would occur in a manner similar to the explicit downgrade request.

In an explicit downgrade, the application sends a portion of data to the SSP hardware via a mailbox style interface and requests the downgrade. The hardware then decrypts the information, applies the new level of protection, and returns the downgraded information to the application. To ensure that the correct application is making the request, only the static and dynamic data keys available in the SSP hardware context at the time of the request can be used to decrypt the data in the mailbox. As an additional protection measure, the mailbox interface can be locked by a single request until the request is complete.
Appendix G

Vita

Joshua Nathaniel Edmison was born on January 15, 1981 in Greenville, North Carolina. He earned his Bachelor of Science, Master of Science, and Doctor of Philosophy degrees in Computer Engineering from Virginia Tech in 2002, 2004, and 2006, respectively. He currently resides in Maryland with his wife, Gina, and conducts research at BBN Technologies.