PRIVATE COMPUTING ON PUBLIC PLATFORMS

A Dissertation Presented to the Graduate Faculty of

Southern Methodist University

in

Partial Fulfillment of the Requirements

for the degree of

Doctor of Philosophy

with a

Major in Computer Engineering

by

Thomas Morris

B.S. Texas A&M University
M.S. Southern Methodist University

July 31, 2008
Private Computing on Public Platforms

Advisor: Professor Suku Nair

Doctor of Philosophy conferred July 31, 2008.


Private Computing on Public Platforms (PCPP) is a new security approach which enables applications to run securely and privately on third party systems. PCPP isolates applications to ensure that the application control flow and data remain unaltered, unmonitored, and unrecorded before, during, and after execution.

In this thesis we define PCPP by expanding the unaltered, unmonitored, and unrecorded requirement to develop a public computing threat model. Additionally, we develop a set of overall PCPP requirements and characteristics which include; PCPP must be a software only implementation, PCPP will require opt-in from 3rd party remote platforms, PCPP will offer the ability to opt-out, PCPP will validate remote platforms prior to use, PCPP will protect individual applications rather than entire systems, PCPP can protect legacy applications, PCPP must provide an encryption key protection mechanism, and PCPP must defend against all threats in the public computing threat model.

We further propose a PCPP architecture which uses a set of 5 PCPP building blocks, host assessment, executable guard, secure context switch, secure I/O, and encryption key protection. The host assessment evaluates 3rd party remote platforms to
ensure that their configuration matches the execution and security requirements of the PCPP application. The executable guard is a new binary executable format designed to protect the executable code while it is stored in non-volatile memory on the 3rd party remote platform and also offers a secure executable launch process. Secure context switch encrypts all PCPP application state when the PCPP application loses ownership of the host processor and decrypts the state when the PCPP application regains control of the host processor. With Secure I/O all file contents are always encrypted when stored in non-volatile memory. Secure I/O protects file access by encrypting all write data and decrypting all read data. The encryption key protection service safely stores PCPP encryption keys on the 3rd party remote platform during application execution using a modification to the Linux context switch routine which protects encryption keys while not in use and uses a set of integrity checks to confirm only the protected application may access the stored keys. We offer expanded definitions and discussions of each PCPP building block in the body of this thesis.

We have completed implementations of all the PCPP building blocks. We offer discussions of the implementations and results comparing the execution time of ordinary applications to that of applications running with PCPP building blocks in place. Additionally, we offer a second PCPP architecture which call demand encryption/decryption which offers improved speed and security.
# TABLE OF CONTENTS

LIST OF TABLES  
LIST OF FIGURES  
ACKNOWLEDGMENTS

Chapter

1. INTRODUCTION  
   1.1. PCPP Overview  
   1.2. Public Computing Threat Model  
   1.3. PCPP Requirements

2. PREVIOUS WORKS  
   2.1. Introduction  
   2.2. Trusted Computing  
   2.3. Security Enhanced Linux  
   2.4. Host Assessment  
   2.5. Executable Guard  
   2.6. Secure Context Switch  
   2.7. Secure I/O  
   2.8. PCPP Key Protection  
   2.9. Conclusion
3. HOST ASSESSMENT

3.1. Introduction

3.2. Go/No-Go Assessment

3.3. Bayesian Classification

3.4. Data collected from Remote Hosts

3.5. Synthetic Data used to test the classifier

3.6. Host Assessment Implementation

3.7. Classification Results

3.8. Host Classification Discussion

3.9. Conclusion

4. EXECUTABLE GUARD

4.1. Introduction

4.2. Privacy Protected ELF (PPELF)

4.3. Conclusion

5. SECURE CONTEXT SWITCH (SCS)

5.1. Introduction

5.2. Accessing Other Process’s Memory Contents

5.3. Secure Context Switch (SCS) Overview

5.4. Performance Analysis

5.5. SCS Run time Overhead

5.6. Experimental Evaluation

5.7. Implementation Specifics
<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Public Computing Threat Model</td>
<td>6</td>
</tr>
<tr>
<td>2. PCPP Requirements</td>
<td>8</td>
</tr>
<tr>
<td>3. NP-Hard Problem Replacement Parameters</td>
<td>29</td>
</tr>
<tr>
<td>4. NP-Hard Problem Replacement Requirements</td>
<td>29</td>
</tr>
<tr>
<td>5. Remote Host Go/No-Go Requirements</td>
<td>35</td>
</tr>
<tr>
<td>6. Attributes collected for Naïve Bayesian Classification</td>
<td>41</td>
</tr>
<tr>
<td>7. Naïve Bayesian Classification with Synthetic Data</td>
<td>43</td>
</tr>
<tr>
<td>8. Naïve Bayesian Classification With Synthetic Data</td>
<td>48</td>
</tr>
<tr>
<td>9. ELF Launch Times</td>
<td>57</td>
</tr>
<tr>
<td>10. PPELF Launch Times</td>
<td>57</td>
</tr>
<tr>
<td>11. Secure Context Switch Run Time Terms</td>
<td>66</td>
</tr>
<tr>
<td>12. Run time Overhead Constants</td>
<td>69</td>
</tr>
<tr>
<td>13. Modified x86_64 Page Header Contents</td>
<td>80</td>
</tr>
<tr>
<td>14. Demand Encryption/Decryption Run Time Terms</td>
<td>86</td>
</tr>
<tr>
<td>15. Virtual Address Translation Data</td>
<td>86</td>
</tr>
<tr>
<td>16. PCPP Key Cache Integrity Hash Contents</td>
<td>118</td>
</tr>
<tr>
<td>17. Initial Key Transfer</td>
<td>124</td>
</tr>
<tr>
<td>18. Key Protection Overhead Variable Definitions</td>
<td>132</td>
</tr>
</tbody>
</table>
20. SELinux Applied to the Public Computing Threat Model 141
22. Resistance to Public Computing Threats by Technology 143
23. PCPP Requirements Comparison 144
<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. PCPP System Overview</td>
<td>4</td>
</tr>
<tr>
<td>2. PCPP Use of Linux Key Retention Service</td>
<td>25</td>
</tr>
<tr>
<td>3. Key Hiding with NP-Hard Heuristics</td>
<td>26</td>
</tr>
<tr>
<td>4. Key Hidden in TSP Instance</td>
<td>28</td>
</tr>
<tr>
<td>5. Key Hiding with Reverse Hash Chains</td>
<td>31</td>
</tr>
<tr>
<td>6. getnewkey() Algorithm</td>
<td>32</td>
</tr>
<tr>
<td>7. ELF versus PPELF</td>
<td>54</td>
</tr>
<tr>
<td>8. PPELF Load Procedure</td>
<td>55</td>
</tr>
<tr>
<td>9. Secure Context Switch - Relinquishing the CPU</td>
<td>62</td>
</tr>
<tr>
<td>10. Secure Context Switch - Regaining the CPU</td>
<td>63</td>
</tr>
<tr>
<td>11. Secure Context Switch Overhead</td>
<td>64</td>
</tr>
<tr>
<td>12. Secure Context Switch Predicted Overhead</td>
<td>70</td>
</tr>
<tr>
<td>13. Measured Run Time SCS Vs. Unmodified Kernel</td>
<td>73</td>
</tr>
<tr>
<td>14. Demand Encryption Decryption Context Switch out</td>
<td>81</td>
</tr>
<tr>
<td>15. Demand Encryption/Decryption Context Switch in Behavior</td>
<td>82</td>
</tr>
<tr>
<td>16. Decryption on Page Fault</td>
<td>83</td>
</tr>
<tr>
<td>17. Demand Encryption/Decryption Overhead for Low Memory Bandwidth Processes</td>
<td>88</td>
</tr>
<tr>
<td>18. Demand Encryption/Decryption Overhead Medium Memory Bandwidth Processes</td>
<td>89</td>
</tr>
<tr>
<td>19. Demand Encryption/Decryption Overhead with 100mS Time Slice</td>
<td>90</td>
</tr>
<tr>
<td>Number</td>
<td>Section Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>20.</td>
<td>Demand vs. SCS vs. Unaltered OS Run Times</td>
</tr>
<tr>
<td>21.</td>
<td>Demand vs. SCS Run Time Overhead Relative to Unaltered OS</td>
</tr>
<tr>
<td>22.</td>
<td>Demand Encryption/Decryption Run Times for 10MB Heap</td>
</tr>
<tr>
<td>23.</td>
<td>Demand Encryption/Decryption Run Times 100M Bytes</td>
</tr>
<tr>
<td>24.</td>
<td>Linux File System Implementation</td>
</tr>
<tr>
<td>25.</td>
<td>PPELF Format with Added SIO Exclusion List</td>
</tr>
<tr>
<td>26.</td>
<td>Encryption Key Cache</td>
</tr>
<tr>
<td>27.</td>
<td>PCPP Key Protection Flow Chart</td>
</tr>
<tr>
<td>28.</td>
<td>Context Switch Overhead from Various Key Protection Configurations</td>
</tr>
<tr>
<td>29.</td>
<td>Predicted PCPP Key Protection Overhead</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

This thesis would not have been possible without the help of many generous individuals. First, I would like to thank Dr. Suku Nair for guiding me through my PhD process. Our weekly meetings were a constant source of encouragement and motivation. Second, thank you Texas Instruments for funding so much of my education and for giving me the time and flexibility to pursue my PhD. Third, thank you to my PhD committee for your valuable feedback and kind words.

Where would I be without my wife and kids? I couldn’t have done this without them. Tristan and Mathis always encouraged me to take a break. Thanks I needed that. My wife always told me reviewers would like my papers more if I would just drop her name in the text. Here goes: Sayuri.
Chapter 1

INTRODUCTION

A long standing goal in computer science has been to enable the remote use of otherwise idle computing platforms. The actual problem of connecting to machines and launching processes remotely has arguably been solved. For instance, the Globus Alliance [18] offers a grid framework which has been used to create many working grids, some with thousands of connected computers. Such grids have found wide acceptance in academia. Additionally, many specialized grids to support specific research projects also exist, such as SETI@home [16] [17]. With the problem of connecting to remote hosts and launching applications largely solved, why then have resource intensive commercial computing users not taken advantage of this technology? We argue the main impediment to the adoption of this technology commercially is the privacy risks associated with executing applications on machines out of the direct control of the corporations wishing to use the remote resource. Currently, no technology is in place which protects the privacy of applications as they run on 3rd party platforms from other processes running on the same machine.

Globus grids and the SETI@Home project offer security typical of most grids. Globus grids require all users and all remote platforms to have a PKI certificate. The PKI certificates are used to authenticate the two remotely connected parties to one another. After authentication a session encryption key is negotiated to encrypt the communication.
channel between the two parties. However, beyond the authentication and the encrypted
communication channel grid users must implicitly trust that other processes and users on
the remote platform will not invade the privacy of their applications. Such a trust based
system does not scale well to larger grids, since as grids grow the likelihood of malicious
users and/or malicious applications impregnating the grid increases. SETI@Home uses
redundancy to validate results returned from remote platforms. This allows the SETI
project to compare results returned from multiple remote platforms for the same data and
throw out results which do not match the measured norm. Relying on redundancy of
course does not protect the privacy of any applications and data sent to the public
platform. Redundancy also drastically decreases the efficiency of using remote hosts for
processing. For example, if redundancy requires 3 hosts for each data set processed
overall efficiency is reduced by two thirds. As the number of redundant hosts increase
the efficiency drops further.

The security mechanisms used with Globus grids and the SETI@Home example
are inadequate for protecting high value applications and data. Globus offers an
essentially trust based application security model leaving any application security
deployment to the remote system administrator. Alternatively, SETI@Home does not
attempt to protect the privacy of the SETI application or its data. In contrast PCPP
systems must actively secure applications and their data when using public platforms.

1.1. PCPP Overview

Private Computing on Public Platforms (PCPP) is an application security
technology designed to protect applications operating in public computing environments.
In this context we liken the security measures of PCPP to the security measures taken by a traveling dignitary. In the case of a traveling dignitary security is divided into two parts. First, an advance team travels to the destination to assess the state of security, including liaising with local security authority, at the destination. The advance team then recommends approval or disapproval of the visit based upon the level of assessed security threat to the dignitary. Second, after the advance team approves the visit, the dignitary travels to the destination. During the actual visit, the dignitary does not travel alone relying on the local security authorities for protection. Rather, the dignitary brings along a team of trusted cohorts who actively guard the dignitary while abroad. PCPP implements a security model similar to the traveling dignitary example. First, PCPP assesses the public platform configuration and measures the security risk associated with the public platform. If this risk is deemed acceptable, PCPP will launch a job to execute on the foreign host. During execution on the foreign host the PCPP application is protected by 4 active security building blocks. We call the initial assessment step host assessment, and the 4 active security building blocks are called the executable guard, secure context switch, secure I/O, PCPP key protection.

We define public computing as the use of computer resources available over a networked connection. In our public computing model these spare compute resources, or remote hosts, are out of the direct control of the user. Rather than trust the security apparatus available on the remote host PCPP adds an application specific protection layer, via the active security building blocks, which actively protects the application for the entire time it resides on the remote host.
Figure 1 shows an overview of the PCPP system. First, we show a local client and PCPP host. The local client is a computer platform belonging to the PCPP user. This user wishes to dispatch an application to run on a remote platform, which we call the PCPP host. Before a PCPP host is used it undergoes a host assessment. This assessment confirms the remote PCPP host’s ability to execute the PCPP application and classifies the host as a threat or non-threat. If a PCPP host is classified as a non-threat the application may be launched on the PCPP host. Before dispatching the PCPP application to the PCPP host, the application executable is encrypted with our executable guard building block and all data which is to travel with the application is also encrypted. The application and a secured encryption key bundle are then sent to the PCPP host. After dispatch, the local client monitors the secure channel and periodically provides updated encryption key packets.

Once on the PCPP host the protected PCPP application executes in a secure environment isolated by four active PCPP building blocks called the executable guard,
secure context switch, secure I/O, and encryption key protection. The executable guard is an encryption layer integrated into the executable binary format which protects the executable code while it is stored on the PCPP host’s non-volatile storage and which securely loads the executable into memory at launch time. While executing the secure context switch and secure I/O building blocks protect the application’s context and all open files used by the application. Secure context switch and secure I/O rely on the property that on a single processor platform the executing application is secure while it owns the CPU, but must be protected any time other processes own the CPU. Secure context switch uses an operating system architecture update to encrypt application memory during context switch before allowing other processes control of the CPU. A key Secure I/O requirement is that all file contents must be encrypted while stored on the PCPP remote host. Additionally, Secure I/O uses an operating system architecture update to encrypt file write data before the data goes to non-volatile memory and decrypt file read data returned from non-volatile memory. The encryption key protection module uses a key cache to store PCPP encryption keys such that they are protected from other applications running on the PCPP host.

1.2. Public Computing Threat Model

Our public computing threat model is derived directly from our definition of private computing which we define as the ability to protect an application such that its executable code, data, and control flow remain unaltered, unmonitored, and unrecorded before, during, and after execution.

Table 1 summarizes the public computing threat model.
Table 1: Public Computing Threat Model

<table>
<thead>
<tr>
<th>Public Computing Threat</th>
<th>Relative Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alter executable code or control flow</td>
<td>Before</td>
</tr>
<tr>
<td>Alter data</td>
<td>Before</td>
</tr>
<tr>
<td>Copy executable code</td>
<td>Before</td>
</tr>
<tr>
<td>Record data</td>
<td>Before</td>
</tr>
<tr>
<td>Monitor program control flow and data</td>
<td>Before</td>
</tr>
</tbody>
</table>

Alteration of executable code can lead to false results from the executable, denial of service, or exploitation of an executable for malicious purposes. Such alterations can occur while the application is stored on a computer’s hard disk drive, or other non-volatile storage, and they can occur during execution while the executable code resides in instruction memory. Alteration of executable code while it is stored in non-volatile memory encompasses the before and after time frames from table 1. These alterations may be simple changes to the executable or wholesale replacement of the executable. During execution other processes running on the same platform may attempt to overwrite the application’s instruction memory contents to alter the application’s control flow. Both cases must be blocked.

Much like altering executable code altering application data can lead to false results from the executable, or control flow changes which again lead to denial of service, or exploitation of an executable for malicious purposes. Like executable code application data must be protected while stored in non-volatile memory and during execution when stored in DRAM or in temporary files.

Malicious users may copy executable code to enable its unauthorized use on other platforms or other data. Malicious users may also copy executable code to take it elsewhere to mine it for secrets, such as embedded keys, data or details on how the code is designed. Copying executable code requires accessing the code either while it is
stored in non-volatile memory, before and after execution, or while it is stored in instruction memory during execution. Both mechanisms for copying executable code must be denied.

We define recording data as the ability to copy data from non-volatile storage before and after execution or from DRAM during application execution. Such recording of data may enable common theft patterns such as theft of social security, driver’s license, and credit card numbers, pin numbers and passwords, and email addresses. In commercial settings such data theft may target email contents, financial information, or other corporate secrets.

Finally, malicious users may attempt to monitor executable code or data without copying or altering contents monitor. This can occur before during and after application execution. For instance, a user may simply view the contents of stored email messages or display the contents of an executable while it is stored in non-volatile memory. As both data and executable code may contain secrets this must not be allowed. During execution users may attempt to view the contents of temporary files, registers, and DRAM. Again this may divulge secrets and must be blocked.

1.3. PCPP Requirements

Table 2 lists a set of requirements for PCPP. First, PCPP systems must defend against all threats listed in our public computing threat model, i.e. PCPP systems must ensure that application executable code, control flow and data remain unaltered, unmonitored, and unrecorded before, during, and after execution on the public platform. Second, since requiring hardware changes would severely limit the number of potential
host platforms, changes on the remote platform required to enable PCPP should be limited to software. Third, PCPP must be an opt-in and opt-out system. This means two things, one, no platform will be used without the platform owner’s prior permission, and two, since PCPP is an opt-in system we are able to download and install new software onto the public platform which enables PCPP security and privacy. Some owners may wish to opt-out. As such, all changes made for PCPP must be reversible.

<table>
<thead>
<tr>
<th>Requirement</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Prevent all threats from the public computing threat model</td>
<td></td>
</tr>
<tr>
<td>2 Software Only Implementation</td>
<td></td>
</tr>
<tr>
<td>3 Opt-in / Opt-out system</td>
<td></td>
</tr>
<tr>
<td>4 Platform Configuration Validation</td>
<td></td>
</tr>
<tr>
<td>5 Application specific protection</td>
<td></td>
</tr>
<tr>
<td>6 Legacy Software Support</td>
<td></td>
</tr>
<tr>
<td>7 Encryption Key Protection</td>
<td></td>
</tr>
</tbody>
</table>

Fourth, we must be able to validate the configuration of PCPP systems. Platform validation confirms that the platform has a hardware and software configuration compatible with the intended application and confirms that all PCPP required software changes are in place and executing properly. Fifth, a PCPP system should provide application specific protection. As such, only chosen applications are subject to its protections, and consequently only chosen applications are subject to its overhead. Sixth, PCPP should provide protection for legacy software. In other words, we do not wish to require a rewrite or recompile of an application before it can be protected. Finally, any encryption keys used on the public platform must be protected. This includes protecting any encryption keys used to secure the channel between the public platform and the local
client and protecting any encryption keys used on the public platform to isolate PCPP applications.

In the remainder of this thesis we offer a chapter discussing related works, followed by chapters discussing each PCPP building block. Next, we compare PCPP to two existing technologies Trusted Computing and SELinux. Finally, we offer a chapter on future works and conclusions.
Chapter 2

PREVIOUS WORKS

2.1. Introduction

In this chapter we discuss previous works with some relation to PCPP. We first discuss two larger scale technologies, Trusted Computing (TC) [8] and Security Enhanced Linux (SELinux) [6], both of which provide application security features. Neither TC nor SELinux were designed to specifically eliminate the threats from our public computing threat model, however, both contain rich application security feature sets which can be partially applied to this end.

Next we include individual sections covering related works for each of the 5 PCPP building blocks: host assessment, executable guard, secure context switch, secure I/O, and PCPP key protection.

2.2. Trusted Computing

The Trusted Computing Group (TCG)\(^1\) is an industry consortium [4] with over 170 members, including Intel, AMD, and Microsoft. The TCG develops specifications for the trusted computing hardware building blocks and software interfaces which primarily focus on protecting user data [4]. Trusted Computing (TC) proposes fundamental changes to the computer architecture via the Trusted Computing Platform

\(^1\) The Trusted Computing Group is formerly known as the Trusted Computing Platform Alliance.
Architecture (TCPA) specification, a TCG document, defines a hardware device which attaches to a platform’s motherboard called the Trusted Platform Module (TPM). TC implements five key security features in hardware and requires that these features be tamperproof, resistant to observation, and resistant to reverse engineering. These security features are secure boot, hardware based encryption functions, a platform specific encryption key, curtained memory, and sealed data. Intel Trusted Execution Technology (TXT)\textsuperscript{2}, AMD Presidio, and Microsoft Next Generation Secure Computing Base (NGSCB)\textsuperscript{3} are all proposed TC implementations based upon the TCPA and which use a TPM.

The remainder of this section first details five Trusted Computing primary features; secure boot, hardware based encryption functions, embedded encryption key, curtained memory, and sealed data. Next we discuss how these features can be used to address the threats in our public computing threat model.

2.2.1. Hardware Based Encryption Functions

All TCPA TPM devices are required to implement in hardware a random number generator, a SHA-1 hash generator, hardware to generate RSA asymmetric keys, and RSA asymmetric encryption and decryption functions. Additionally, hardware implementations of symmetric encryption functions are optionally allowed. Embedding these functions in hardware provides two primary advantages. First, the hardware implementations are specified to be tamperproof, resistant to observation of intermediate values, and resistant to reverse engineering. This makes them less prone to manipulation

\textsuperscript{2} Intel Trusted Execution Technology is formerly known as LaGrande
\textsuperscript{3} Microsoft’s Next Generation Secure Computing Base is formerly known as Palladium.
and or observation than their software equivalents. The second advantage of implementing these functions in hardware is speed. Because of the throughput advantages of hardware based encryption and hash functions, TPM based platforms can be expected to support more extensive usage of these algorithms without sacrificing significant amounts of performance.

2.2.2. Embedded Encryption Keys

In addition to hardware implemented encryption functions the TCPA TPM contains a unique 2048-bit RSA symmetric encryption key, referred to in the TCPA documentation as the Endorsement Key (EK). EK is broken into two parts a public key portion called PUBEK and a private key portion called PRIVEK. PRIVEK is completely private to the TPM device and may never be read out of the TPM. Additionally, like all TPM hardware EK is specified to tamperproof and resistant to reverse engineering. EK may only be used for two purposes. First, EK may be used to establish a shared secret between the platform owner and the TPM. This shared secret is stored in the TPM and used to authenticate the platform owner before allowing certain security critical functions such as disabling the TPM. Second, EK may by used to derive Attestation Identity Keys (AIK). EK must be burned-in to the TPM during TPM manufacturing or at least prior to delivery of the protected computer to its final customer.

AIK are asymmetric RSA key pairs which are aliases of EK. These AIK are used to sign data. Such signatures are unique to the signing platform since the AIK is an alias of EK which is unique to one platform.
In addition to EK and AIK the TPM can create and store encryption keys for general usage. These general use encryption keys can be read out of the TPM for transmission to other applications or parties or stored and protected inside the TPM.

2.2.3. Secure Boot

Secure boot is a methodology used to verify the platform configuration during the boot process. Secure boot uses multiple Platform Configurations Registers (PCR), embedded on the Trusted Computing Platform (TPM) device, to verify platforms hardware and software configuration. The boot code is broken into multiple executable code segments. Before a segment may call the next segment in sequence it must first hash the next segment and store the resulting hash to a secure boot PCR. The first code segment, a small segment of the BIOS, hashes itself.

The boot code also reads identifying hardware configuration information and writes this to a second PCR.

The TPM gates the platform’s hardware reset signal. When the secure boot PCR contents match previously stored known good values the platform is allowed to exit reset. If the secure boot PCRs never contain the correct values the platform will never exit reset.

The TPM contains 24 PCR. Some PCR are dedicated to secure boot, while others are left for general use. PCR are linked to a hardware implementation of a SHA-1 hash function. When a new value $v$ is written to a PCR, $v$ is concatenated with the register’s existing contents $\eta_i$, and then the concatenated string is hashed with the SHA-1 algorithm creating $\eta_{i+1}$. 
\[ \eta_{t+1} = SHA1(\eta_t \parallel v) \]

This process can be repeated indefinitely, continuously writing new \( v \) to the PCR. The final value, \( \eta \), in the PCR uniquely represents the chain of values \((v_0,v_1,v_2,...)\) written to the PCR. \( \eta \) is not commutative, meaning the different orders of \((v_0,v_1,v_2,...)\) give different \( \eta \). Additionally, because of the one way nature of the hash function knowledge of \( \eta \) cannot be used to learn \((v_0,v_1,v_2,...)\).

2.2.4. Curtained Memory

The fourth feature common among Trusted Computing proposals is curtained memory. Curtained memory is not required by the TCPA specification. However, Intel TXT, and Microsoft NGSCB specify its use. However, since curtained memory is not covered by the TCG, the body intended to create uniform trusted computing specifications, Intel’s and Microsoft’s curtained memory proposals differ.

The intent of curtained memory is to physically isolate memory pages so that only authorized processes may access the protected pages. The granularity of this protection varies. Intel TXT [9] is able to isolate memory pages to a specific process, while Microsoft NGSCB [1] breaks the platform’s memory into 5 localities, 1 page each. These localities are dedicated as follows 0; normal application, 1; trusted application, 2; trusted OS, 3; auxiliary, and 4; trusted chip set. The auxiliary and TPM localities are only accessible by the TPM.

Both technologies specify that curtained memory must be inaccessible to DMA engines and other bus masters, meaning only the CPU may access curtained memory.
With Intel TXT multiple pages of curtained memory can be assigned to a single process, whereas, with Microsoft NGSCB only 5 static curtained memory pages exist on the entire platform meaning only certain code segments, not whole programs, can execute from curtained memory.

2.2.5. Sealed Storage

Combined use of Platform Configuration Registers (PCR) and the TPM embedded encryption capabilities enables sealed storage. Sealed storage uses a result stored in one or more PCR to generate an encryption key which is then used to encrypt, aka seal, data. Decrypting sealed data requires the same PCR to contain the same value(s) held at encryption time. The encryption key is only accessible to the TPM.

The sealing process is used in two places. First, the non-volatile storage (HDD) on the platform is sealed with the secure boot PCR contents. In this case the encryption key is derived from the hardware and software contents of the platform including the platform’s endorsement key (EK). This seal permanently ties the contents of the machines HDD to that specific hardware/software configuration. This prevents two types of data theft attacks; booting a different OS to bypass access controls, and removing a hard disk drive and placing it in another machine.

The second use of sealed storage enables application data privacy. In this case an application may load one or more PCR with some combination of values, likely including a hash of the application itself. Application data is then sealed based upon these PCR contents. After sealing the PCR contents are cleared. The only way to decrypt the sealed application data is to load the PCR with same set of values used prior to encryption.
2.3. Security Enhanced Linux

The standard access control system used by the Linux operating system is a discretionary access control (DAC) system which provides four roles; user, group, rest of world, and super user, and protects two types of objects; files and directories. Files and directories are each assigned a set of read, write, and execute permissions for each of three roles owner, group, and rest of world. Super users are allowed access to all files and may change file permissions unconditionally. When an application is executed the application inherits the user’s userid and groupid which allows applications to act as the user. The limited number of roles forces many applications to run with super user privileges in order to access protected objects. However, executing with super user privilege not only provides access to the needed object, but, actually gives unlimited access to the entire system. Hackers seek out and exploit vulnerabilities in programs which run with super user privilege because of this unlimited access property. For example, the sendmail application needs privilege to access the ports required to send email. Since, there is only one privileged role in Linux, portions of sendmail run as super user. Overtime many sendmail vulnerabilities have been found and exploited to give rogue programs super user privilege which in turn has allowed these rogue programs unlimited system access. We don’t mean to pick on sendmail alone, it is just an example. Many applications have suffered similar fates.

Discretionary access control systems with similar behavior can be found in the Unix operating system and the Windows operating systems. Security Enhanced Linux (SELinux) [6] [7] was developed by the United States National Security Agency as an example of a mandatory access control system which significantly improves the
capabilities and effectiveness of the access control system over DAS implementations. SELinux offers three main features to achieve this access control improvement. First, SELinux vastly increases the number of types of objects which can be controlled. Second, SELinux uses the Flask architecture to separate policy declarations from policy enforcement. Second, SELinux adds a flexible policy language which allows for both fine grain and general access controls.

2.3.1. SELinux Objects

SELinux originally defined seven object classes which may be assigned protection [6]. These seven protected object classes are processes, open files, file systems, pipes and files, directories, sockets, and network interfaces. Each object class has a set of grantable permissions pertinent to that object class. This means for instance the available permission fields for processes are different from the available permissions fields for open files. The popularity of SELinux has lead to healthy growth in the number of object classes, [19] lists 30 object classes.

To provide an example of the descriptive powers of object class permissions we provide the permission fields available for processes and open files. Processes have 16 permission fields: execute, transition, entry point, sigkill, sigstop, sigchild, signal, fork, ptrace, getsched, setsched, getsession, getpgid, setpgid, getcap, and setcap. Open files have 5 permission fields: create, getattr, setattr, inherit, receive. Each of these permissions can be granted or excluded from a given type in system’s policy declaration. The permission names are generally descriptive, see [6] for more information, and represent permission to act upon the controlled object.
2.3.2. SELinux Flask Architecture

SELinux separates policy declaration from policy enforcement. Each controlled object in a system is assigned a security identification number (SID). The SID is used to look up associated policy labels which contain the actual permissions for the object. Object SID’s change over time as the policy configuration types associated with the object change. When an object begins to act upon another object the SELinux security server steps in and governs that access based upon the permissions defined in policy configuration. The creators of SELinux refer this separation of policy declaration and policy enforcement as the Flask architecture, a named derived from its original use with the Flask operating system.

2.3.3. SELinux Policy

SELinux policy defines a set of identities, a set of roles, and a set of types. Permissions are declared by allowing certain actions between pairs of types, thereby defining how objects may act upon one another and who may use objects. Each user on a system has one identity. A user’s identity is generally assigned one role, though multiple roles are allowed. Each role is associated with a list of types. Each protected object is associated with one or more types in the policy declaration.

Exactly how the identity, role, and type constructs work together to enforce access control can be confusing. Each user is given a unique identity in the SELinux policy configuration and each identity is associated with a set of roles. When a user logs into a system his or her login shell process inherits the user’s default role and the policy type of the user’s shell process transitions to a type associated with that role and that process.
Later as the user executes programs the SELinux security server verifies permissions before allowing or denying the requested action. This is done by examining the permissions assigned to the controlled object for the user’s process type.

The policy configuration may specify that certain actions cause an object’s type to change. For example, if Alice is an ordinary user and Bob is a system administrator their process types may transition differently when running the same application. Alice may transition from `default_alice_t` to `app_alice_t`. While Bob may transition from `default_bob_t` to `app_bob_t`. Type `app_alice_t` might allow Alice only read access to a file in a directory, while the type `app_bob_t` might allow Bob to read, write, create, and erase to files in the same directory. Such type transitions can be defined for any of the supported object and all objects can have multiple types associated with them.

Users may also have more than one role. For instance, Bob, the system administrator may have two roles, one for ordinary use, and a second role which is associated with the extra privileges he needs as a system administrator. Bob may default to the lesser privilege role and then increase his privilege by transitioning to a new role, which generally, requires reentry of the user’s password.

SELinux policy also supports default types and roles. This for example allows all files on a system not otherwise listed in the SELinux policy to have a default type. That default type can then be setup with default permissions for that object. However, SELinux policy derives it strength through specificity. In other words, simply allowing all objects in a system to take a default type would provide only limited benefits. Instead, objects with extra access control needs are defined in the SELinux policy with specific types and permissions associated with them. With such specificity policy can be
developed which zeroes in on an objects needs. For example, if a mail program, sendmail, needs to access port 25 to send mail a policy can be created giving that program access to port 25 specifically. If that program is the only program in the system which needs to access this port, then it should be the only program in the SELinux policy configuration associated with port 25. Since, SELinux prohibits actions not expressly allowed in the policy configuration; all other applications will not be allowed to access port 25. Such specificity limits potential damage when programs are exploited. This is true because applications are limited to actions defined by SELinux policy. For example, if sendmail is the only application specifically allowed to access port 25, exploits of other processes will not be able to access port 25. Port 25 is only vulnerable if sendmail is vulnerable.

In addition to the type based and role based protections mentioned above SELinux optionally allows multi-level security (MLS) access control. MLS allows multiple levels of security clearance to be defined with increasing authority to access objects. This is similar to government clearance levels for documents such as; secret, top secret, eyes only, etc. MLS systems are generally setup in such a manner that higher clearance levels allow wider access and lower clearance levels allow less access. Such MLS systems become trust based and differ from the SELinux normal of requiring specific policy to control specific actions. In an MLS system users with high privilege have broad access to the system.
2.4. Host Assessment

There are many commercial and freeware tools available to scan computers for security policy adherence and vulnerabilities. Virus scanners and spyware scanners are two examples. Virus scanners search system files using a pattern matching algorithm to detect known signatures of viruses. Spyware scanners review common storage locations for spyware programs which may be inadvertently downloaded onto user’s computers.

I.T. departments use more sophisticated tools to scan for policy adherence and vulnerabilities. Such tools include the CIS Next Generation Scoring Tool [24] and NMAP [25]. The CIS Next Generation Scoring Tool is used by system administrators to collect security related information via a single Java executable then create security benchmark scores. Such benchmark scores can be used to attest to the overall security of a group of computers or to judge the security configuration of a given machine. These tools typically review configuration information and are less concerned about the immediate security state of each individual machine. The CIS Next Generation Scoring Tool is merely an example. There are many tools like used by I.T. departments and government agencies. Other such tools include the SANS Institute S.C.O.R.E. [26], IBM AppScan [30], the Microsoft Baseline Security Analyzer and [31].

NMAP is a freeware security scanner which is popularly used by both the security community and the hacker community. NMAP is concerned about the security state of individual machines, at least as far open ports on the machine are concerned. NMAP can be used to scan one or many computers across a local intranet or across the internet. NMAP scans all known ports looking for open ports. If a port is open it uses pattern
recognition to attempt to learn the operating system of the scanned machine and the version of the server running on the open port.

With the exception of the virus and spy ware scanners the scanners discussed above are generally not used by the average computer user.

2.5. Executable Guard

There are many programs called packers which are used to pack executable code into an encrypted and obfuscated bundle. These packers generally wrap the executable with a second program which decrypts and or de-obfuscates the executable as it executes. These programs are often developed by and used by hackers to attempt to stop their executables from being scanned, reverse engineered, and traced. Shiva [32] is one such system which uses both encryption and obfuscation to protect ELF executables. Shiva embeds a decryptor and the encrypted executable code into a single executable file. The decryptor decrypts and de-obfuscates small blocks of code as they are needed at run time. The Shiva authors note that large programs suffer run time delays. UPX [33] is also an executable packer. UPX makes no claims about providing executable privacy; rather it is intended as an executable compression tool. We mention it here because UPX decompresses ELF programs into memory rather than decompressing onto disk then executing the decompressed program. UPX actually decompresses the program into a memory image then calls exec() to launch the decompressed code. This is not the same mechanism used to decrypt PPELF but both avoid transforming the executable only to write it back in plaintext form to nonvolatile storage.
File system encryption [34] encrypts data written to a file system or volume. Such systems do not address executable encryption directly, but, since all data on the file system is encrypted on writes to non-volatile storage and decrypted only on reads to non-volatile storage there is some overlap with PCPP. The main difference between file system encryption and PPELF is that file system encryption is a system level approach rather than application level. In PCPP we prefer to isolate the individual application. File system encryption does not provide such isolation, since all users of the file system can access its encrypted contents.

2.6. Secure Context Switch

We find no related work which isolates application memory by encrypting application memory pages while they are not in use. The closest thing to this is the executable code packers mentioned in section 2.5. These packers sometimes use encryption, sometimes compression to mask the contents of the executable until they are loaded into memory, sometimes leaving the contents encrypted or compressed until just before use. However, these packers work on executable code and do not attempt to protect application memory while the program executes.

One proposed technology [35] we found in our literature search is the idea of creating encrypted executables which execute in encrypted form and create output data which is also encrypted. The remote system would return the encrypted output for decryption on the application owner’s system. This work seems to have been a proposal which was never followed with an implementation.
2.7. Secure I/O

As mentioned in section 2.5, file system encryption [34] encrypts the data written to a file system or volume. In such systems, which are readily available on Linux and Windows operating systems data is encrypted as part of the write process and decrypted when read from the file system. If there is more than one file system mounted on a platform and not all of the file systems are encrypted then a user may decrypt a file by copying it from the encrypted file system to a non-encrypted file system. This is not true for Secure I/O, since the encryption is at the application level. The simple act of moving a Secure I/O protected file from one file system to another will not decrypt the file.

Furthermore as with the PPELF discussion in section 2.5, Secure I/O provides application level isolation, while file system encryption provides only file system wide isolation which means any program with access to the file system can access any data stored on the file system.

2.8. PCPP Key Protection

2.8.1. Linux Key Retention Service

The Linux Key Retention Service (LKRS) [14] allows applications to store and retrieve key material from key rings, which are structures which point to linked lists of keys. The LKRS is divided into facilities for managing kernel keys and facilities for managing user keys. Because PCPP applications run as user applications they have access to LKRS user key rings. Both key types, kernel and user, are stored on three default key rings; the session key ring, the process key ring, and the thread key ring.
The left half of Figure 2 shows the LKRS default session key ring. Within LKRS a session is a set of processes and threads. Processes and threads which are launched from a session inherit access to the session key ring. A process can disconnect itself from one session key ring and create a new session key ring for itself which is then private to that process and all processes and threads subsequently spawned by the process. The right half of Figure 2 shows a separate PCPP session key ring and its associated process and thread key rings which are disconnected from the default session key ring.

Process key rings are available to a single process and all threads which belong to that process. Process key rings cannot be read by other processes or threads which belong to the same LKRS session. Thread key rings are accessible only by the thread they belong to. Thread key rings cannot be accessed by the parent process (or any other process) or by other processes or threads which share the same process. Note from Figure 2 that each thread contains a private thread key ring, $TK_1$ or $TK_2$, and also shows inherited access to a process key ring and session key ring.

In previous work [10] we used the LKRS to store PCPP encryption keys. In that work a PCPP session was forked off and separated from the users default session as

![Diagram of PCPP Use of Linux Key Retention Service]

**Figure 2: PCPP Use of Linux Key Retention Service**
shown in Figure 2. Because the PCPP session is disjoint from all other LKRS session key rings the PCPP session and its child processes and threads should enjoy exclusive access to the session key ring $SK_{pcpp}$, the process key ring $PK_{pcpp}$, and any thread key rings. We show in chapter 8 that this intended privacy can easily be violated by users with privileged access.

As defined above, PCPP should be able to use an LKRS thread key ring to safely store its encryption keys. However, we have successfully implemented an exploit which allows one user on a Linux host to access the LKRS stored keys of another user on the same machine.

2.8.2. NP-Hard Encryption Key Hiding

NP-Hard key hiding [21] method uses a randomly chosen NP-Hard problem ($p$), a heuristic to solve the problem ($h$), and a problem instance ($i$). The master key $k_M$ is hidden in the solution, as derived by $h$, of the problem instance $i$. Figure 3 shows the overall NP-Hard key hiding process. The local client stores a set of NP-Hard problems

---

**Figure 3: Key Hiding with NP-Hard Heuristics**

<table>
<thead>
<tr>
<th>NP-Hard Problem</th>
<th>Available Heuristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traveling Salesman</td>
<td>1,2,3</td>
</tr>
<tr>
<td>Problem B</td>
<td>1,2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node</th>
<th>Node</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>$w_{ab}$</td>
</tr>
<tr>
<td>a</td>
<td>c</td>
<td>$w_{ac}$</td>
</tr>
<tr>
<td>b</td>
<td>c</td>
<td>$w_{bc}$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

1. Choose Problem ($p$)
2. Choose Heuristic ($h$)
3. Create Instance ($i$)
4. Compile Heuristic/Instance as $khm_{phi}$
5. Send $khm_{phi}$ to Host

Route | Key Val |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>$k_1$</td>
</tr>
<tr>
<td>a</td>
<td>$k_2$</td>
</tr>
<tr>
<td>c</td>
<td>$k_3$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
and multiple heuristics each problem. Before sending $k_M$ to the PCPP host, the local client randomly chooses a NP-Hard problem, $p$, a heuristic for that problem, $h$, and creates a random instance, $i$, of problem $p$. A local version of the $h$ is executed to solve $i$. This allows the initial $k_M$ to be embedded in the compiled heuristic which will be sent to the PCPP host. A solution look-up table (complete with embedded $k_M$), $h$, and $i$ are compiled as a single program and converted to a PPELF executable, to become $khm_{\text{phi}}$ (key hiding module). Examination of $khm_{\text{phi}}$ will not reveal $k_M$, $h$, or $i$. Choosing a random problem and heuristic will cause the size of $khm_{\text{phi}}$ to vary for heuristic, problem pair. Additionally, embedding the look-up table and $k_M$ in $khm_{\text{phi}}$ will also lead to varying sizes for $khm_{\text{phi}}$ since the problem instances will vary in size based upon the problem chosen and some directed random variation in problem instance size. Converting $khm_{\text{phi}}$ to PPELF hides the details of the problem, heuristic, instance, and initial from examination on the PCPP host. The PPELF encryption and varied contents will also lead to different hash signatures for $khm_{\text{phi}}$. Finally, once available $khm_{\text{phi}}$, is transported to the PCPP host.

On the PCPP host the $khm_{\text{phi}}$ is used to store and fetch $k_M$. As mentioned above $k_M$ is changed for each context switch pair.

Figure 4 shows an example of a key hidden in a TSP problem instance. In the figure the solution to the problem, i.e. the shortest route through each node, is shown with grey dashed arrows. The key is broken into parts, labeled $k0$, $k1$, ..., $k7$, and the parts are stored in a look-up table with the node name as the index. Since the key parts are stored as unencrypted plaintext the look-up table will need to have many more entries than just those used to store the key parts. Each of the other entries should store a random number.
As such, the overall entropy of the entire table should remain relatively constant making it difficult to distinguish the key material from the random numbers. Only use of the heuristic program khm\(_\text{phi}\) will fetch the correct key parts in the correct order.

Hiding the key parts in a TSP solution is relatively straight forward. Other NP-Hard problems can and should also be used with this method. Each NP-Hard problem will need to be examined and mapped to a mechanism for storing the key based upon its solution.

We mention above that \(k_M\) can only be fetched using khm\(_\text{phi}\) however if an adversary is able to learn the problem instance \(i\) and problem \(p\) a separate heuristic might be used to learn \(k_M\). Because of this we introduce a replacement algorithm which replaces \(p, h,\) and \(i\) periodically.

![Figure 4: Key Hidden in TSP Instance](image)

Table 3 shows a list of four problem replacement parameters and their respective definitions. Table 4 shows a set of four required properties which use the parameters from Table 3 to describe the periodic replacement of \(p, h,\) and \(i\). Adherence to the properties in Table 4 should make it computationally infeasible to fetch the key without using the khm\(_\text{phi}\) module.
Table 3: NP-Hard Problem Replacement Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_r$</td>
<td>khm replacement period</td>
</tr>
<tr>
<td>$t_c$</td>
<td>context switch period</td>
</tr>
<tr>
<td>$t_i$</td>
<td>time to solve instance $i$ without use of heuristic $h$</td>
</tr>
<tr>
<td>$t_h$</td>
<td>time to solve instance $i$ with use of heuristic $h$</td>
</tr>
</tbody>
</table>

The first property from Table 4, $t_r > t_c$, states that the replacement period for khm$_{phi}$ should be greater than the context switch period on the PCPP host. Replacement of khm$_{phi}$ will require network communication between the local client and the PCPP host which is quite slow relative to time required to complete a context switch. In order to feasible we cannot expect to get a new khm$_{phi}$ module for every context switch pair. Rather we must diffuse the extra communication time for replacing khm$_{phi}$ across many context switches. The second property states that the time to solve problem instance $i$ with khm$_{phi}$ must be much less than the context switch period. SCS and SIO already increase the context switch delay. We want to minimize any additional delay related to fetching the master key, $k_M$.

Table 4: NP-Hard Problem Replacement Requirements

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_r &gt; t_c$</td>
<td>Replacement period much greater than context switch period</td>
</tr>
<tr>
<td>$t_h &lt;&lt; t_c$</td>
<td>Time to solve with heuristic much less than context switch period, i.e. minimize overhead</td>
</tr>
<tr>
<td>$t_i &gt;&gt; t_h$</td>
<td>Time to solve with heuristic much less than time to solve without heuristic</td>
</tr>
<tr>
<td>$t_r &lt;&lt; t_i$</td>
<td>Replace faster than problem can be solved without heuristic</td>
</tr>
</tbody>
</table>

The third property from Table 4 states that the time to solve the problem with khm$_{phi}$ must be much less than the time required to solve the problem without a heuristic. The idea here is that without access to the heuristic an adversary would not be able to fetch the key in a timely manner because solving the problem without a heuristic is NP-Hard. This property fails to account for the fact that the problem may be solved with
another heuristic, perhaps even faster. This is one of the open issues with this method. If we were to use this method with PCPP we must overcome this issue. One possible way to overcome this issue is to embed errors in \( \text{khm}_{\phi} \). This would potentially stop the use of other heuristics which solve the same problem. The final property from Table 4 states that we must replace \( \text{khm}_{\phi} \) faster than it can be solved with the use of the heuristic embedded in \( \text{khm}_{\phi} \). Again, if we assume that no other heuristic can be used to solve the problem faster then we should be able to replace the problem instance faster than an adversary can successfully solve the problem.

Finally, successful use of the NP-Hard key hiding method requires the assumption that an adversary cannot execute the same module which PCPP uses to fetch the key, in this case, the assumption is that an adversary cannot execute \( \text{khm}_{\phi} \). Since, the reverse hash chain method requires the same assumption we discuss this issue after detailing the reverse hash chain key hiding method.

2.8.3. Reverse Hash Chain Encryption Key Hiding

With the reverse hash chain key hiding method [22] the terms of an \( n \)-length hash chain like that shown in equation 2 are used as set of master keys. The first master key used is the \( n-1 \) term of \( h_{c} \), the second master key is the \( n-2 \) term, and so on. The entire hash chain is not stored on the PCPP host, rather only the \( n^{th} \) term and the initialization vector, \( h_{c0} \), are stored on the PCPP host. When \( k_{M} \) is fetched the terms of \( h_{c} \) are calculated until the term \( h_{cn} \) is matched, \( h_{cn-1} \) is returned as \( k_{M} \). \( k_{M} \) cannot be learned from \( h_{cn} \) because of the one way nature of hash functions. The only way to learn \( k_{M} \) is by calculating the terms of \( h_{c} \) until \( h_{cn} \) is matched.
\[ f(x) = \text{hash}(x) \]

\[ hc = \{hc_0, f(hc_0), f(f(hc_0)), f(f(f(hc_0))), ... \} \]

Figure 5 shows the process for creating a hash chain on the local client and communicating that hash chain to the PCPP host. First, a hash function, \( f \), and an initialization vector, \( hc_0 \), are chosen. The hash function is randomly chosen from multiple options to decrease the chances of an adversary on the PCPP host learning the hash algorithm. The initialization vector is a random number. Next, \( f \) and \( hc_0 \) are used to generate the \( n \)-term hash chain, \( hc \), where is \( n \) is a suitable length. We will define the term *suitable length* in future work. The hash chain, \( hc, f, \) and \( hc_0 \) are all compiled on the local client into a hash chain module, \( hcm \), and \( hcm \) is converted to a PPELF executable. Finally, \( hcm \) is sent to the PCPP host for use in fetching master keys.

On the PCPP host each time the key cache is to be encrypted with a new master key, \( k_M \), the \textit{getnewkey()} procedure in \( hcm \) is called to return a key. Figure 6 shows an outline of the \textit{getnewkey()} procedure. In a nutshell \textit{getnewkey()} calculates the terms of \( hc \) until \( hc_n \) is matched. When \( hc_n \) is matched \( hc_{n-2} \) is returned as the new key, the size of

---

**Figure 5: Key Hiding with Reverse Hash Chains**

1. Choose hash function \( f \)
2. Choose \( hc_0 \)
3. Create \( n \) term hash chain \( hc \)
4. Compile \( f, hc_0, hc_0 \)
5. Convert to PPELF program, \( hcm \)
6. Send \( hcm \) to host
$hc$ is decremented by one, and $hc_{n-1}$ becomes the new endpoint.

When the PCPP hosts fetches $k_M$ for decryption of the key cache a procedure similar to the $getnewkey()$ procedure is used. The only differences between $getnewkey()$ and this second procedure are that the length of hash chain is not decreased in this case and $x_{n-1}$ is returned as $k_M$.

1. $x = hc_0$
2. $x_{n-1} = x$
3. $x = f(x_{n-1})$
4. if ($x != h_n$) goto step 2
5. $h_n = x_{n-1}$
6. if ($h_n = f(h_0)$) get new hash chain
7. return $x_{n-2}$

**Figure 6: getnewkey() Algorithm**

Since $hc$ has $n$ terms it can be used $n-1$ times before replacement, $hc_0$ should not be used as a key. Replacement requires the local host to create a new hash chain and to compile the new $hc$, $hc_0$, and $hc_n$ as a new $hcm$.

2.8.4. Common Problems

Both NP-Hard encryption key hiding and reverse hash chain encryption key hiding rely on the assumption that the compiled implementation used on the PCPP host is not executable by adversaries. This is because if an adversary can execute the key fetch algorithm he or she can fetch the master key in a timely manner and then use it to decrypt the key cache and consequently begin breaking down all other PCPP protections.

Essentially, by encrypting the key cache with a master key we have narrowed our protection problem to the protection of single master key. We hide the master key with one of the two key hiding mechanisms, however, in their current form this simply creates
a new secret which much be protected, shifting our protection burden yet again from the master key to the respective key fetch module and its contents.

For the NP-Hard key hiding method an adversary can learn the master key in a timely manner if he or she either learns the heuristic, $h$, and the problem instance, $i$, or if he or she can execute $khm_{\phi_i}$. For the reverse hash chain key hiding method an adversary can learn the master key in a timely manner if he or she learns the initialization vector $hc_0$, the hash method, and the $n^{th}$ term $hc_n$. Alternatively, if an adversary can execute the reverse hash chain key fetch module, $hcm$, he or she can also learn the key in a timely manner. So, for the NP-Hard method the new secrets are $h$, $i$, and $khm_{\phi_i}$. For the reverse hash chain method the new secrets are $hc_0$, the identity of the hash function, $f$, the $n^{th}$ term $hc_n$, and $hcm$. In order to use either of the proposed key hiding methods we must either find a way to protect these new secrets. If we cannot create an effective protection of the new secrets associated with the key hiding method we will search for some third key hiding methods.

2.9. Conclusion

In our review of previous works we have found no works which directly overlap with the contributions of PCPP. We find other approaches to the application isolation problem and other approaches similar to individual PCPP building blocks. We find no previous works which combine to meet all the PCPP requirements from chapter 1.
3.1. Introduction

Before execution on a remote host can be allowed the potential remote host must be validated to meet three general criteria. First, the remote host must provide the needed resources, correct hardware, operating system, etc. Second, the remote host must be have downloaded and installed all PCPP required operating system patches and those patches must be intact and unaltered. Finally, the security state of the prospective remote host must be such that it is deemed a non-threatening environment on which to execute PCPP protected applications.

The PCPP host assessment is broken into two parts, the go/no-go scan, the Bayesian classification, which combine to meet the three criteria mentioned above. The go/no-go scan uses a quick external scan combined with data gathered by a PCPP scan process running on the remote host to confirm the system meets the minimum execution requirements for our PCPP application, confirm certain minimum security mechanisms are in place, and to confirm that the operating system and all PCPP patches are in place and unmodified. The Bayesian classification module uses a much larger collection of data gathered from a PCPP scan process running on the remote host and similar collected data from previous PCPP runs on other remotes hosts to classify the prospective host as a threat or non-threat.
3.2. Go/No-Go Assessment

The first step in the PCPP remote assessment is verification of a set of go/no-go properties pertaining to the host. These properties are a minimum set of system capabilities and security requirements.

Table 5: Remote Host Go/No-Go Requirements

<table>
<thead>
<tr>
<th>Requirement Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Virus detection present</td>
</tr>
<tr>
<td>2</td>
<td>Virus signatures up to date</td>
</tr>
<tr>
<td>3</td>
<td>Last virus scan &lt;= 14 days</td>
</tr>
<tr>
<td>4</td>
<td>Intrusion detection present</td>
</tr>
<tr>
<td>5</td>
<td>Software firewall present</td>
</tr>
<tr>
<td>6</td>
<td>Spyware detection present</td>
</tr>
<tr>
<td>7</td>
<td>Spyware signatures up to date</td>
</tr>
<tr>
<td>8</td>
<td>Last spyware scan &lt;= 14 days</td>
</tr>
<tr>
<td>9</td>
<td>Root kit detection present</td>
</tr>
<tr>
<td>10</td>
<td>Root kit signatures up to date</td>
</tr>
<tr>
<td>11</td>
<td>Last root kit scan &lt;= 14 days</td>
</tr>
<tr>
<td>12</td>
<td>Open critical ports</td>
</tr>
<tr>
<td>13</td>
<td>OS Version</td>
</tr>
<tr>
<td>14</td>
<td>OS security patches up to date</td>
</tr>
<tr>
<td>15</td>
<td>External port scan data must match data gathered via internal host scan</td>
</tr>
<tr>
<td>16</td>
<td>Prior host classification as non-threat</td>
</tr>
<tr>
<td>17</td>
<td>PCPP Critical Host Patches and Software Unaltered</td>
</tr>
</tbody>
</table>

Table 5 lists the minimum set of requirements for the no/go-scan. For the items related to virus detection, intrusion detection, software firewall, spyware detection, root kit detection, and OS versions, an approved list of products will exist. The remote assessment scanners will support only these approved versions. If a remote host is using software unapproved by the PCPP client for one or more of the go/no-go requirements the requirement will not be satisfied.

The remote host will also be scanned externally to generate a list of open ports, software versions for services running on the open ports, and OS version information. The data gathered externally must match data provided to the on-host PCPP scanner.
Additionally, PCPP clients keep a database of information collected about previous PCPP jobs. If the prospective host has been used for a previous job it must have been classified as a non-threat.

Part of the go-no-go assessment must be to confirm that the remote host has not altered or encumbered the execution of any PCPP critical code. This critical code includes the operating system of the remote host including any PCPP required patches made to the operating system, the host assessment code itself, and any PCPP server code used to communicate between the PCPP local client and remote host. This sort of software configuration verification is a difficult problem in itself. In chapter 2 we discussed Trusted Computing as a previous work. Trusted Computing uses a complex root of trust mechanism and it’s Platform Configuration Registers (PCR), section 2.2.3, to create a single hash result which can be used to prove the validity of a systems configuration. The PCPP host assessment does not have such powerful hardware mechanisms at its disposal. The PCPP host assessment can use a hash implementation and a similar layered hash approach to validate a system configuration, but, the PCPP host assessment version is not rooted in a hardware chain of trust. In an attempt to overcome this deficiency the PCPP host assessment uses the Bayesian classification to attempt to recognize commonalities among host which tend to lie to the PCPP clients by altering some part of the PCPP critical software or attacking PCPP jobs as they run. We use Bayesian classification to predict whether the new potential host is a threat or non-threat.
3.3. Bayesian Classification

The second phase of the PCPP remote host assessment employs a Naïve Bayesian classifier [23] to predict whether the remote host is a threat or non-threat based upon historical PCPP experience. We measure the value of a set of 32 attributes primarily related to the security configuration of the remote host for each PCPP job run. We postulate that new PCPP hosts attack the PCPP application when the new host has many attribute measurements in common with historical hosts which attacked previous PCPP applications. Therefore, we want new hosts, with a number of attributes in common with previously known malicious hosts, to be classified as threats. Furthermore, we postulate that new PCPP hosts will not attack PCPP applications when the new host has many attribute measurements in common with historical hosts which did not attack previous PCPP applications. We want new hosts with much in common with previously known non-malicious hosts to be classified as non-threats.

We use a Naïve Bayesian classifier and a training database containing the historical attribute measurements and the posterior classification from all prior PCPP jobs to predict if a prospective host is a threat or non-threat. If a host is predicted to be a non-threat the pending PCPP job may be launched on the host. If the host is classified as a threat the pending job will not be launched on the host.

The Naïve Bayesian classifier classifies new hosts by applying equation 3 to compute the probability that tuple \( t_i \) is a member of class \( C_j \). Tuple \( t_i \) is the 32 attribute tuple collected from the prospective host. We compute probabilities for each class \( C_j \) in \( C \) (equation 2). The new host is assigned to the class with the higher probability from equation 3.
The three terms of equation 3 each provide separate levers in predicting the class of the new tuple.

First, the probability of tuple $t_i$ given class $C_j$, defined in equation 4, is a product of the rates of class membership for all of the attributes measurements found in the tuple. When many attribute measurements have high rates of membership in class $C_j$, the probability that tuple $t_i$ is a member of class $C_j$ increases. Conversely, when few attribute measurements have high rates of membership in class $C_j$ the probability that tuple $t_i$ is a member of class $C_j$ decreases.

Second, the probability of class $C_j$, defined in equation 6, is actually the portion of the historical data which belongs to each class. When the historical data has a high rate of a given class, the probability of the new tuple being a member of that class increases and when a given class is rare, the probability of the new tuple being a member of that class decreases.

Finally, the denominator of equation 3 is the probability of the tuple itself. We note that the denominator is independent of class. It is the same for both classes and does not impact the comparison of $P(\text{threat} \mid t_i)$ and $P(\text{non-threat} \mid t_i)$. As such, it can be left out of the calculations and the numerators alone can be computed and compared to provide our classification.

The math used in our implementation of the Naïve Bayesian classifier is discussed below. Equations 4-9 are derived from discussion in [23].

$$C \in (\text{threat}, \text{non}\text{-threat})$$

$$P(C_j \mid t_i) = \frac{P(t_i \mid C_j)P(C_j)}{P(t_i)}$$
When classifying a tuple, we compute the probabilities (equation 5) of each class \( C_j \) in \( C \) (equation 4) given the tuple \( t_i \). We classify tuple \( t_i \) into the class with the higher probability.

\[
P(t_i|C_j) = \prod_{k=1}^{32} P(t_{ik}|C_j)
\]

The probability of tuple \( t_i \) given class \( C_j \) (equation 6) is the product of individual conditional probabilities (equation 7) for each attribute in the tuple. The individual probabilities from equation 7 are computed by counting the occurrences of the attribute values for the given class \( C_j \). In practice, during our training step we create a table of probabilities from equation 6 for each attribute value for each class. We then use this probability table in the classification step to look-up the probabilities needed for equation 5.

\[
P(t_{ik}|C_j) = \frac{count(t_{ik}|C_j)}{count(C_j)}
\]

The probability of threat or non-threat (equation 8) is simply a count of all incidences of the given class in the training database divided by the total number of tuples in the training database. We compute these probability values during training as they do not change when classifying a new tuple.

\[
P(C_j) = \frac{count(C_j)}{count(threat) + count(non\text{-}threat)}
\]

One of the primary advantages of the Naïve Bayesian classifier is its simplicity. There is no complex difference equation required, there is no need for the measurements to take a numeric form and the math needed to compute the probabilities necessary for Bayesian classification are simple. For each attribute we simply define sub-classes and
then for computing the intermediate probabilities the majority of the work is spent counting occurrences of each attribute for each class.

Naïve Bayesian uses historical data gathered from previous PCPP jobs to measure the similarity of the prospective host to the hosts in the training set. Each attribute in the classification is treated independently, yet, Naïve Bayesian measures the probability of class membership as product of probabilities from each attribute and therefore capable of noticing patterns of similarity in the data which humans and other classifiers such as decision trees cannot observe.

For the exercises in this paper we chose to use just two classes, threat and non-threat. It is possible to use the same Naïve Bayesian classification with more than two classes. For instance, it may be desirable to break the threat class into multiple levels of risk which could allow certain PCPP jobs with lower value to be run on low risk hosts which must be avoided by higher valued PCPP jobs.

3.4. Data collected from Remote Hosts

When a new PCPP job is launched the PCPP client will gather a new remote assessment tuple from the prospective host. After a PCPP job completes the host’s remote assessment tuple and the result of the job are added to the classification training data. The result of the job, whether the active security monitor detected violations or not, is recorded as threat or non-threat, respectively.

In order to minimize PCPP job startup latency, we expect that the host will have precompiled the data required for Bayesian classification. This data will come from
periodic scans of the host which are stored as tuples for use in the PCPP host assessment.

Because the scan is run periodically it may be scheduled as a low priority task.

### Table 6: Attributes collected for Naïve Bayesian Classification

<table>
<thead>
<tr>
<th>Item</th>
<th>Bayesian Ranges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virus protection present</td>
<td>TRUE, FALSE</td>
</tr>
<tr>
<td>Virus protection brand</td>
<td>A,B,C,D,OTHER</td>
</tr>
<tr>
<td>Virus Signatures up to date</td>
<td>TRUE, FALSE</td>
</tr>
<tr>
<td>Virus days since last complete scan</td>
<td>GOOD, OKAY, BAD, TERRIBLE</td>
</tr>
<tr>
<td>Intrusion detection present</td>
<td>TRUE, FALSE</td>
</tr>
<tr>
<td>Intrusion detection brand</td>
<td>A,B,C,D,OTHER</td>
</tr>
<tr>
<td>Software firewall present</td>
<td>TRUE, FALSE</td>
</tr>
<tr>
<td>Software firewall brand</td>
<td>A,B,C,D,OTHER</td>
</tr>
<tr>
<td>Spy ware detection present</td>
<td>TRUE, FALSE</td>
</tr>
<tr>
<td>Spy ware signatures up to date</td>
<td>TRUE, FALSE</td>
</tr>
<tr>
<td>Spy ware days since last complete scan</td>
<td>GOOD, OKAY, BAD, TERRIBLE</td>
</tr>
<tr>
<td>Spy ware scanner brand</td>
<td>A,B,C,D,OTHER</td>
</tr>
<tr>
<td>Root kit scan present</td>
<td>TRUE, FALSE</td>
</tr>
<tr>
<td>Root kit scan brand</td>
<td>A,B,C,D,OTHER</td>
</tr>
<tr>
<td>Root kit signatures up to date</td>
<td>TRUE, FALSE</td>
</tr>
<tr>
<td>Root kit days since last scan</td>
<td>GOOD, OKAY, BAD, TERRIBLE</td>
</tr>
<tr>
<td>Open critical ports</td>
<td>TRUE, FALSE</td>
</tr>
<tr>
<td>Port server software up to date</td>
<td>TRUE, FALSE</td>
</tr>
<tr>
<td>OS patches up to date</td>
<td>TRUE, FALSE</td>
</tr>
<tr>
<td>OS automatic updates</td>
<td>TRUE, FALSE</td>
</tr>
<tr>
<td>OS version</td>
<td>XP_HOME, XP_PRO, WIN2K, OTHER</td>
</tr>
<tr>
<td>File system</td>
<td>NTFS, OTHER</td>
</tr>
<tr>
<td>IP region</td>
<td>USA, EUROPE, EASTEUILGRPE, ASIA, OTHER</td>
</tr>
<tr>
<td>Known bad MAC address</td>
<td>TRUE, FALSE</td>
</tr>
<tr>
<td>Password complexity</td>
<td>TRUE, FALSE</td>
</tr>
<tr>
<td>Expired passwords</td>
<td>TRUE, FALSE</td>
</tr>
<tr>
<td>Auto login disabled</td>
<td>TRUE, FALSE</td>
</tr>
<tr>
<td>Guest accounts disabled</td>
<td>TRUE, FALSE</td>
</tr>
<tr>
<td>Anonymous login disabled</td>
<td>TRUE, FALSE</td>
</tr>
<tr>
<td>Number of administrator accounts</td>
<td>TRUE, FALSE</td>
</tr>
<tr>
<td>Active X security level</td>
<td>TRUE, FALSE</td>
</tr>
<tr>
<td>Java Security level</td>
<td>TRUE, FALSE</td>
</tr>
<tr>
<td>MS Office macros</td>
<td>TRUE, FALSE</td>
</tr>
</tbody>
</table>

It is possible for the host to provide false answers to PCPP assessment inquiries.

In this event a prospective host may fool the PCPP assessment into providing a non-threat classification when a threat was the correct classification. In such cases presumably the
host would attack the PCPP job and this would be detected by the PCPP active security monitors. This would trigger an immediate shutdown of the PCPP job in progress, a classification of this host as a threat in the PCPP client training database, and seed the Bayesian classifier to classify similar hosts as threats in the future.

Table 6 shows all of the attributes and their legal values which we used in the Bayesian classification. The set of attributes collected was derived from a survey of information gathered by various vulnerability scanners available as freeware and commercially [24][25][26].

The cardinality of allowed values for most attributes was kept small intentionally. This limits the possibility of missing data in the training set and limits the number of counting steps required during training. Limiting allowed values also limits the granularity of the classifier. However, if it is found that more specificity is needed the possible value set can be expanded or single attributes can be replaced by multiple, more specific attributes. An example of limiting the legal values is found for the case of attributes which measure the number of days since the last (virus, root kit, spy ware) scan. We group these values in to the categories of GOOD, OKAY, BAD, and TERRIBLE. If we had left this attribute to be an integer from 0 to $N$, we would be required to compute $N$ conditional probabilities for that attribute in the training step. This would be extra work for little extra classification value.

The actual attributes listed in Table 6 are less important than the concept of using a Naïve Bayesian classifier in this context. It is relatively easy to scan a host for many different types of attributes. If better indicator attributes are found, they can easily be used in place of or in addition to those listed in Table 6.
Data from prior PCPP jobs may be shared amongst PCPP clients. This would allow the training data to become sufficiently large for accurate classification sooner. Sharing remote assessment tuples from prior PCPP jobs will also disseminate information about new threats faster and potentially limit the impact from these threats. The remote assessment tuples from prior PCPP jobs may be shared via a central server or peer to peer exchange. Tuples should be shared only if their validity can be guaranteed. Without this guarantee shared data could be used to maliciously manipulate the results of the classifier.

3.5. Synthetic Data used to test the classifier

In order to judge the effectiveness of the Naïve Bayesian classifier we created synthetic constrained random data sets to both train our classifier and test our classifier. In all we created seven constrained random data sets. These data sets are listed in Table 7.

<table>
<thead>
<tr>
<th>Data Set Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1k</td>
<td>1K constrained random tuples with patterned threats and 1% random threats</td>
</tr>
<tr>
<td>10k</td>
<td>10K constrained random tuples with patterned threats and 1% random threats</td>
</tr>
<tr>
<td>100k</td>
<td>100K constrained random tuples with patterned threats and 1% random threats</td>
</tr>
<tr>
<td>1k_nr</td>
<td>1K constrained random tuples with patterned threats</td>
</tr>
<tr>
<td>10k_nr</td>
<td>10K constrained random tuples with patterned threats</td>
</tr>
<tr>
<td>100k_nr</td>
<td>100K constrained random tuples with patterned threats</td>
</tr>
<tr>
<td>Random</td>
<td>100K constrained random tuples with 50% random threats</td>
</tr>
</tbody>
</table>

In the preceding table, we use the term constrained random to mean, first, that the attribute values for each tuple were generally chosen at random while only being allowed to take legal values for each attribute as listed in Table 6. Second, some attributes where constrained to take a certain value for a certain percentage of the time and or situationally. For instance, the virus protection present attribute on the system was
constrained to be TRUE 95% of the time. Further, if the virus protection present attribute was FALSE the virus brand attribute was forced to OTHER, the virus signatures up to date attribute was forced to FALSE, and the virus days since last complete scan attribute was forced to TERRIBLE.

There are three basic types of data listed in Table 7 which differ by how the classification as threat or non-threat was generated. Except for the data called Random, the data set names contain 1K, 10K, or 100K which indicate the number of tuples in the data set. The Random data set contains 100K tuples.

The six data sets, 1k, 10k, 100k, 1k_nr, 10k_nr, and 100k_nr all contain a set of hidden patterns. The hidden patterns are designed to show that the Naïve Bayesian classifier will generally detect these patterns and flag prospective hosts as threats. These patterns were generated in the data as additional constraints. First, anytime the bad MAC address attribute was TRUE, the tuple was classified as a THREAT. The bad MAC address attribute was allowed to be TRUE 0.5% of the time. Second, when the Virus protection brand attribute was B, the virus days since last complete scan attribute was BAD, and the OS patches up to date attribute was FALSE the class was forced to THREAT. Finally, when root kit protection present attribute was FALSE and the open critical ports attribute was TRUE the class was forced to THREAT.

In addition to the embedded patterns the three sets called 1k, 10k, and 100k contain an additional 1% of tuples which were classified as threat regardless of the value of the attributes in the tuple. The three sets called 1k_nr, 10k_nr, and 100k_nr do not contain patterns randomly classified as threat.
The *Random* data set contains no embedded patterns, rather 50% of the tuples are randomly classified as threats regardless of the value of the attributes and 50% are randomly classified as non-threats regardless of the value of the attributes.

All of the data mentioned above was created using Cadence Specman Elite [27], commercial software used to automate test benches used for digital integrated circuit verification. Cadence Specman Elite executes code written in the ‘e’ language [27] which contains a random constraint solver which we used here to create our synthetic data. The constraint solver has three features we used for this project. First, we used the ‘e’ language to specify legal values for each of the attributes in our synthetic data tuple. Second, we specified the distribution of attribute values for certain attributes in our tuple. Third, we specified relationships between the attributes which sometimes override the random behavior (our embedded patterns). We then used the constraint solver to randomly choose tuples which met our specifications.

3.6. Host Assessment Implementation

We implemented our classifier in Perl. Our implementation is broken into two parts. First, we train our classifier with all of the tuples gathered from previous PCPP jobs, or in our case, randomly created synthetic tuples, and the resulting classification of those jobs. Second, our implementation classifies individual tuples from prospective hosts for new PCPP jobs.

In the training step we build a table of conditional probabilities \( P(t_{ik}|C_j) \) and compute the probabilities of each class \( P(C_j) \). These tables are built with a single pass through the data. The probability tables are stored in hashes to speed up search in the
classification step. Computing the probability tables ahead of time allows significant speed up in computation of the final classification of a new prospective host.

For classification we simply look-up the $P(t_{ik}|C_j)$ probabilities for each attribute measurement in our tuple to generate $P(t_i|C_j)$. We then look up the class probability $P(C_j)$. Finally, we compute the probability of class $C_j$ given the tuple $t_i$, for each class in $C$, in our case threat and non-threat. We classify the tuple as a member of the class with the higher probability.

Occasionally in classification we encounter tuples with data values for attributes which require special treatment. We found two such cases.

First, we may find an attribute value in classification which was never sampled in the training data. In this case the conditional probability, $P(t_{ik}|C_j)$ is zero for both the threat and non-threat classes. If these intermediate zero values are kept the resulting probability of class membership $P(C_j|t_i)$ for both classes will be zero. In this case we ignore this attribute when computing the probability of class membership $P(C_j|t_i)$.

The second case which requires special treatment occurs when within the same tuple one or more attribute values always imply a threat classification while simultaneously one or more attribute values always imply a non-threat classification. Here, we mean that the attribute value was always associated with one class and never occurs with the other class. When we have separate attribute values that which always imply classification in opposite directions, the resulting posterior probabilities again become zero. We handle this case by dropping the attribute which implies non-threat and keeping the attribute which implies threat. This sways the final classification towards threat which for our application is the more conservative choice.
3.7. Classification Results

We ran our Naïve Bayesian implementation with the synthetic data a total of twelve times. For each run we trained the classifier with one data set and then classified all the tuples from a second data set.

The classification of each tuple in the test data was checked against the actual class of the tuple. We report the percentage of classifications which are correct, percentage of classifications which are incorrect, percentage of false negatives, and percentage of false positives in Table 8.

All of the first nine runs are trained with data which contains embedded patterns as we discussed earlier. We notice that when there is a pattern in the data, the Naïve Bayesian classification does a very good job of finding similar tuples in the test phase and classifying these correctly.

When the data set name contains the post fix ‘\_nr’ this means the data does not contain any random classification. In other words the class in these data sets always depends on the three patterns embedded in the data. When the data set name does not contain the ‘\_nr’ post fix the training data contains approximately one percent of the tuples randomly assigned to the threat class regardless of the presence of the aforementioned patterns. This can be seen in the results. For runs in which the test data contained this 1% random assignment to the threat class, the percentage of incorrectly classified tuples also approaches 1%.

In the final three runs the tuples in the training data were classified completely randomly. Since we have 2 classes, threat and non-threat, the split between classes was approximately 50% threat and 50% non-threat. The results from these runs show that the
classifier found no trends in the data and consequently incorrectly classified the data approximately 50% of the time. This illustrates that when the data has no trends, the Naïve Bayesian algorithm cannot accurately classify tuples.

We expect in real world application, there will generally be trends in the data similar to the ones we placed in the synthetic data. We know that certain attribute values lead to vulnerabilities which can be exploited.

Generally, a system which works to quickly remove these vulnerabilities will be a more secure system. Conversely, a system which does not fix vulnerabilities in a timely manner will be less secure. When these observations hold true, the Naïve Bayesian will do a good job classifying hosts. There will always be cases of new vulnerabilities which look like the random threats we introduced in our synthetic data. Generally, new vulnerabilities will tend to be classified as non-threats at first, then as more PCPP jobs are encountered where the vulnerability is exploited the trend will become apparent to the classifier and these tuples will be classified as threats.

<table>
<thead>
<tr>
<th>Training Data</th>
<th>Test Data</th>
<th>% Correct</th>
<th>% Incorrect</th>
<th>% False Negative (Fatally Wrong)</th>
<th>% False Positive (Missed Opportunity)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10k</td>
<td>1k</td>
<td>99.2</td>
<td>0.8</td>
<td>0.8</td>
<td>0.0</td>
</tr>
<tr>
<td>100k</td>
<td>10k</td>
<td>98.9</td>
<td>1.1</td>
<td>0.9</td>
<td>0.2</td>
</tr>
<tr>
<td>10k_nr</td>
<td>1k_nr</td>
<td>99.3</td>
<td>0.7</td>
<td>0.2</td>
<td>0.5</td>
</tr>
<tr>
<td>100k_nr</td>
<td>10k_nr</td>
<td>99.6</td>
<td>0.4</td>
<td>0.1</td>
<td>0.3</td>
</tr>
<tr>
<td>10k_nr</td>
<td>1k</td>
<td>99.2</td>
<td>0.8</td>
<td>0.8</td>
<td>0.0</td>
</tr>
<tr>
<td>100k_nr</td>
<td>10k</td>
<td>98.9</td>
<td>1.1</td>
<td>0.9</td>
<td>0.2</td>
</tr>
<tr>
<td>1k</td>
<td>10k</td>
<td>98.9</td>
<td>1.1</td>
<td>0.9</td>
<td>0.2</td>
</tr>
<tr>
<td>1k_nr</td>
<td>10k_nr</td>
<td>99.9</td>
<td>0.1</td>
<td>0.1</td>
<td>0.0</td>
</tr>
<tr>
<td>10k</td>
<td>1k_nr</td>
<td>99.3</td>
<td>0.7</td>
<td>0.2</td>
<td>0.5</td>
</tr>
<tr>
<td>random 100k</td>
<td>10k_nr</td>
<td>49.9</td>
<td>50.1</td>
<td>1.1</td>
<td>49.0</td>
</tr>
<tr>
<td>random 100k</td>
<td>1k_nr</td>
<td>54.9</td>
<td>45.1</td>
<td>0.8</td>
<td>44.3</td>
</tr>
<tr>
<td>random 100k</td>
<td>random 100k</td>
<td>51.0</td>
<td>49.0</td>
<td>23.5</td>
<td>25.5</td>
</tr>
</tbody>
</table>

Table 8: Naïve Bayesian Classification With Synthetic Data
A Dell branded Linux workstation was used to perform all of the computations for this paper. The workstation contains 2 Intel Pentium 4 2.8 GHz processors with 512 KB cache each. The workstation contains 4GB shared memory. The Linux kernel version is 2.4.21-37.ELsmp.

We measured the time spent in training and the time spent classifying the test data for each run. The average training time per tuple was $148 \mu S$. The average time spent in classification per tuple was $422 \mu S$. The time spent in training per tuple was about $1/3$ of the time spent in classification per tuple. However, generally we will train with thousands of tuples, while we always classify only one tuple at a time. This means that the majority of time will actually be spent in training. This is acceptable because the training can be done on the PCPP client as a background task periodically as new tuples are collected. The classification step must always be done before a new PCPP job is launched on a remote host. As such, the classification step adds delay to the PCPP job completion time and therefore should be as fast as possible.

3.8. Host Classification Discussion

Since the Bayesian classifier relies on trends in past experience to classify new prospective public platforms new attack trends may be missed the first few times they are encountered. In this case PCPP will black list any attacking machines and learn from the experiences recognizing any trends related to the new attacks.

It is possible that the public platform may have been altered in such way that tuples returned from the host classification step contain falsified information. We do 3 things to counter such an attack. First, a portion of the host assessment is performed via
an external port scan. We confirm during the go/no-go step that the external port scan findings match the internal open port scan. Second, like other threats we hope the Bayesian classifier will recognize trends in the lying behavior, in other words, if one platform lies, it will go undetected, but if the same exploits are used to cause many platforms to lie, we expect the Bayesian classifier to notice the trend. Finally, like any other misclassification, we rely on the other PCPP building blocks to actively protect the application.

It is possible to poison the Bayesian classifier training data base for a Trojan horse type attack. This could be done by implementing an exploit on multiple PCPP public platforms but barring attacks allowing for many successful runs. Overtime, the many successful runs would build a trend of success in the training database causing the Bayesian classifier to classify the Trojan horse platforms as non-threats. Eventually the Trojan horse attack could be sprung to catch unsuspecting PCPP applications. Like the other cases of misclassification we rely on the other PCPP building blocks to actively protect the application. Also, after the Trojan horse attack was sprung it would begin building a new trend in the training data base allowing the Bayesian classifier to begin to recognize this trend and begin to avoid other Trojan horse machines.

3.9. Conclusion

In this chapter we offer an approach for assessing PCPP remote hosts to determine if the host is an acceptable candidate for running PCPP applications. Our solution breaks the problem into two parts. First, we scan the remote host to ensure that a set of go/no-go requirements are met. Second, we use historical data collected from
previous PCPP jobs to train a Naïve Bayesian classifier. We then demonstrated the use of this classifier to classify prospective hosts as threats or non-threats.

The host assessment must include scans to confirm that PCPP critical code, residing on the remote host, i.e. operating system patches, internal scanning code, and server software, has not been in any way modified or restricted. This means the host assessment must as part of the go/no-go checks review hashes of all PCPP critical code to confirm that code has not been altered. Because the host assessment data collection is not rooted in a hardware chain of trust, the host assessment relies on the Bayesian classifier to predict whether a potential host is a threat or non-threat based upon past experience. Furthermore, the host assessment must rely on the Executable Guard, Secure Context Switch, Secure I/O, and PCPP Key Protection building blocks to ensure that the host assessment runs securely and privately.
Chapter 4

EXECUTABLE GUARD

4.1. Introduction

The executable guard is secure method for storing binary executables on a remote host and for loading the stored executables into memory. In practice an executable guard is an encrypted extension of an existing binary format for a given operating system and accompanying binary launch code which loads the executable into memory for execution while avoiding leaking private application code during this critical step. Rather, than decrypting the program to a separate plaintext copy prior to execution and then count on access control to thwart any foe, we load the encrypted program into the memory locations specified by the original program binary format and then decrypt. This allows Executable Guard programs never to be stored as a plaintext files in non-volatile memory (hard disk drive, flash drives, etc.) and only exist as plaintext in volatile memory. This property makes securing executable privacy much easier, since after execution volatile memory regions used by the program can simply be over written with zeroes or random data leaving no trace of the plaintext. The encrypted Executable Guard version of the program stored in non-volatile memory may be safely deleted after execution with a standard delete command. Although, the deleted program may be recovered by an attacker, the recovered program will be in its encrypted form and is therefore useless without its associated encryption key.
An executable can be created for any operating system. However, the nature of the executable guard requires changes to the portions of the operating system which load executables into memory immediately prior to execution. We implemented a version of an executable guard called privacy protected ELF (PPELF) for the Linux operating system. We chose to implement PPELF for the Linux operating system because Linux is an open source operating system which makes PPELF implementation feasible.

4.2. Privacy Protected ELF (PPELF)

PPELF is a variant of the ELF [37] binary executable format. ELF binary executables contain 3 types of headers and then code and data segments which make up the actual program. The first header in an ELF file is the ELF header. The ELF header contains a magic number identifying the file as an ELF program, description of the size and location of the program and section header lists, a pointer to the entry point of the program, and information describing the machine architecture the program is compiled to support. The second header type in an ELF file is the program header which is an array of headers. Each program header provides a pointer to a file location for a different segment of the program used during the program load process. Key among these segments is a segment identifying the interpreter, a separate program, to be used to load shared libraries, and pointers to the code and data segments which are to be loaded into memory for execution. The third header type is the ELF section headers which are also an array of headers. The section headers are used during the linking process to map to different sections of the file. Finally, the ELF program contains program and data
segments which make up the actual instructions used by the program and any data embedded in the program.

Figure 7 shows a cross section of an ELF program beside a cross section of a corresponding PPELF program. PPELF programs are encrypted versions of their ELF counterparts. During conversion from ELF to PPELF the ELF header is changed in one way. The magic number is changed from 0x177 | ‘ELF’ to 0x177 | ‘PCP’. The 0x177 is an unprintable character which denotes the program is binary. Changing the final three bytes of the magic number to ‘PCP’ causes the PPELF binary format handler to be used to load the program.

All of the contents of the ELF program after the ELF header are encrypted. We use 128-bit AES encryption with counter mode with a pseudo random initialization vector. The use of counter mode encryption allows random access to the file for decryption on a byte granularity. The initialization vector is stored at the end of the PPELF program.
Figure 8 shows the steps required to load a PPELF program, \textit{app}, into memory prior to execution. We call the program file \textit{app.exe} to delineate the program file from the executed process, \textit{app}. Figure 8 repeats the same set of images, with variations, 4 times corresponding to 4 times steps in the PPELF load procedure. We call these time steps frames. Each frame contains a box to note the active process, a box listing waiting processes, a hard disk drive (HDD) showing an encrypted \textit{app.exe} executable file, and DRAM with multiple memory blocks labeled with their
corresponding process. The first frame shows the `exec()` system call active. `exec()` calls
the binary loader routines from all registered binary formats until one binary loader
returns successfully. Because each binary format starts with a unique magic number, all
binary loaders will fail except our binfmt_pcpp binary loader. Like ELF programs
PPELF programs are loaded in two steps. First the program headers from `app.exe` are
used to load the code and data segments into memory. Second, an interpreter, specified
by the in the program headers, is loaded and executed to load any required shared
libraries. The second frame shows the binfmt_pcpp binary loader mapping the contents
of `app.exe` to memory pages owned by `app` process. Next, the binfmt_pcpp binary loader
loads the specified interpreter, called `interp` in Figure 8, to complete the load process.
Since, the program headers are used to which segments from `app.exe` should be loaded
and to specify the interpreter; it is read in and decrypted by the binfmt_pcpp binary
loader. After use, the decrypted program headers in DRAM are overwritten with zeroes
and the memory returned to the operating system. The binfmt_pcpp binary loader does
not decrypt the code and data segments of the program. Decryption of the code and data
segments is left for the interpreter. The third time step in Figure 8 shows the interpreter
running. Since the binfmt_pcpp binary loader mapped the code and data segments
`app.exe` to the virtual addresses specified by its program headers, the interpreter can
decrypt these segments in situ, meaning no other memory is allocated to hold
intermediate encryption results, leaving nothing to clean-up. After decryption, the
interpreter loads any required shared libraries and then jumps to the `app` start point. The
final frame shows `app`, with decrypted code and data segments in DRAM, as the active
process.
Since, the program is loaded into memory at the virtual addresses specified by the program headers before decryption and then decrypted in place; there are never copies of the decrypted program anywhere else in non-volatile or volatile memory. This allows one very important security feature, which is the fact that the PPELF program is never stored as plaintext in non-volatile memory. This is shown in all four frames of Figure 8.

### Table 9: ELF Launch Times

<table>
<thead>
<tr>
<th>Application</th>
<th>Size [MB]</th>
<th>ELF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Binfmt [uS]</td>
<td>Interp [uS]</td>
</tr>
<tr>
<td>Helloworld</td>
<td>0.007</td>
<td>27</td>
</tr>
<tr>
<td>Vim</td>
<td>1.2</td>
<td>27</td>
</tr>
<tr>
<td>Firefox</td>
<td>13.1</td>
<td>28</td>
</tr>
</tbody>
</table>

### Table 10: PPELF Launch Times

<table>
<thead>
<tr>
<th>Application</th>
<th>Size [MB]</th>
<th>PPELF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Binfmt [uS]</td>
<td>Interp [uS]</td>
</tr>
<tr>
<td>Helloworld</td>
<td>0.007</td>
<td>42</td>
</tr>
<tr>
<td>Vim</td>
<td>1.2</td>
<td>44</td>
</tr>
<tr>
<td>Firefox</td>
<td>13.1</td>
<td>41</td>
</tr>
</tbody>
</table>

Table 9 and 10 show measured launch times for three PPELF applications in two steps. This data illustrates several points. First, the times marked binfmt is the amount of time spent in the binary format handler. These times are slower for PPELF than ELF. However, the increased delay is static is static across the three test programs. This is the result only decrypting the program headers in binfmt_pcpp. The program headers are small at 32 bytes each. There are multiple program headers per executable with 7 program headers in the helloworld program, 8 in Firefox, and 9 in Vim. This makes the...
The total number of bytes for decryption in the binary format handler vary from 256 bytes to 288 bytes.

The times marked interp are the time spent in the interpreter for each of the three programs. These times dominate the total launch time both for ELF and PPELF programs. For the ELF case much of the time spent in the interpreter is spent loading share libraries from disk and relocating non-reentrant shared library code in memory. For PPELF the time spent in the interpreter is dominated by the decryption of the PT_LOAD segments. The PPELF interp times increase as the executable file size increases, although not quite linearly as explained below by the Vim example.

Linux uses demand paging when mapping files to memory, including, executables. This means that the entire executable is initially mapped to memory by assigning page table entries and virtual addresses, but, the actual contents of the file are not copied from the hard disk drive to memory until the pages are actually accessed by the processor. Since, PPELF decrypts all of the contents of the program in the interpreter all the pages must be read into memory from disk before decryption. On the other hand, in the ELF cases the time in interp includes copying only accessed pages from hard disk drive to memory. This is particularly apparent with Vim. Since Vim does very little on start-up, merely clearing the screen and printing a few characters before waiting for input it accesses comparatively few memory pages. As such there is a more pronounced difference between the ELF Vim launch time and the PPELF Vim launch time.

With the exception of Firefox, the large increase in launch time is still small enough to make it unnoticeable to a human user. For Firefox, the same is actually true because on our slow system although the Firefox window boundary appears quickly, the
total time for all of the graphics to load and for Firefox to become usable takes about 6 seconds when measured on a stop watch. When using the PPELF version of Firefox there is a 1 second increase, but we argue going from 6 to 7 seconds on a slow system is still largely unnoticeable to the user. We also note here that these numbers were gathered on an AMD K7 processor running at 900 MHz. On a modern processor the relative delays may stay the same, but the noticeable impact to the user would diminish.

4.3. Conclusion

The Executable Guard ensures the privacy of binary executables stored on a remote host and ensures the secure and private loading of said executables into memory at program launch time. In this chapter we defined the Executable Guard and then described our Linux based Executable Guard implementation, Privacy Protected ELF (PPELF).

PPELF is an encrypted binary executable format for the Linux operating system based upon the ELF binary executable format. PPELF programs are encrypted ELF program which are decrypted by the ELF interpreter after being mapped from disk to dynamic memory. Since PPELF programs are decrypted in place after being mapped into memory PPELF programs are never stored in a decrypted form on non-volatile memory.

PPELF programs do incur a nominal delay in launch time which increases with the size of the program, but there is no increase in run time of the program after it is loaded into memory.
Chapter 5

SECURE CONTEXT SWITCH (SCS)

5.1. Introduction

Secure Context Switch (SCS) is one of the five building blocks used to solve the PCPP problem. SCS protects the memory pages which store a process’s executable code, data segment, stack, and heap during execution. SCS provides this protection by isolating these memory pages using applied encryption techniques. Specifically, SCS encrypts the aforementioned memory pages immediately before a process relinquishes the CPU during context switch. Immediately before a PCPP protected process regains control of the CPU the encrypted memory pages are decrypted. Our current SCS implementation requires a single processor architecture. On a single processor platform attackers cannot read, write, or alter the PCPP application’s memory when the PCPP application itself is running. When the PCPP application relinquishes the CPU an attacker may attempt to read memory locations owned by the protected application, however, these pages will appear encrypted to the attacker.

5.2. Accessing Other Process’s Memory Contents

PCPP needs to isolate a protected application’s memory from other applications running on a platform. With most operating systems non-privileged users may not access memory which is in use by other users. However, privileged users generally may access
any memory in the system. PCPP needs to isolate its memory contents from even privileged users.

We built a program and an accompanying kernel module for use on Linux platforms to demonstrate the ability of one user to access the memory of another. First, we built a kernel module which scanned the task structures (a structure used by the Linux to hold a thread’s context) of all tasks currently active in the system looking for a certain signature. This signature can be any attribute the task structure contains. In our case we looked for a Boolean indicating the task was a PCPP task. Once a desired task structure was found our kernel module simply followed a pointer which points to a linked list of all contiguous memory areas assigned to the task we were snooping on. Once these memory locations are found accessing them requires pointing the page fault handler at the victim processes memory map and then accessing it like any other memory pointer. Alternatively, one can convert the virtual address to physical address manually to bypass setting up the page fault handler. Bypassing the page fault handler also bypasses any access permissions set for the page to be accessed. We built this first portion of our memory snooper as a kernel module so that this code would run with kernel privilege. We then added a mechanism to allow calling this code from any user process.

We built a second program which runs as an ordinary user program which calls the kernel module to snoop on PCPP processes. We called the snooper module repeatedly in a loop so that we could run until we found a PCPP process in memory to attack. When the snooper successfully found a PCPP process it simply printed a portion of that process’s memory to the screen as hexadecimal code.

To use our snooper we required root privilege only to install the kernel module.
Once this module was installed we were able to access the memory of a process running in Alice’s account, with the snooper client running in Bob’s account.

5.3. Secure Context Switch (SCS) Overview

SCS protects volatile memory assigned to a process by interceding in the context switch procedure and encrypting application volatile memory contents during context switch out and decrypting the same memory contents during context switch in. Specifically, SCS decrypts the process’s stack, CPU registers, heap, and instruction memory. When run on a single processor platform the SCS effectively denies other active processes the ability to access the protected processes volatile memory.

At the end of any PCPP time slice or any call to the kernel’s `schedule()` routine by the PCPP application the Linux scheduler will cause the PCPP application to relinquish the CPU in order to allow some other unrelated waiting process to gain control of the CPU. The scheduler, calls the `context_switch()` routine to switch the current memory context.
map and CPU register contents before relinquishing CPU control to the next process. SCS intercedes in the context switch process and encrypts the memory pages holding the PCPP process’s stack, CPU register contents, heap pages, and instructions before jumping to the next process. When a waiting PCPP process regains control of the CPU, SCS, again within the context switch routine, decrypts the PCPP process context and then allows the context switch to proceed.

Figure 9 and Figure 10 demonstrate the SCS process pictorially. Figure 9 shows a PCPP protected process *app* relinquishing the CPU. In the first frame, the earliest in time, of Figure 9, *app* owns the CPU and its memory contents are completely decrypted. The second frame shows the transfer of CPU ownership. Here the operating system’s context switch routine is executing. The memory pages belonging to app are half grayed to demonstrate that they are being encrypted as part of the context switch. By the 3rd frame process P2, a non-PCPP protected process owns the CPU. In this frame the all memory contents of *app* are encrypted.
Figure 10 shows a non-PCPP protected process, P2, relinquishing the CPU and the PCPP protected process, app, gaining control of the CPU. In the first frame, the earliest in time, of Figure 10, P2 owns the CPU while app waits on the ready to run queue. While waiting app’s memory contents are completely encrypted. The second frame shows the transfer of CPU ownership. Here the operating system’s context switch routine is executing. The memory pages belonging to app are half grayed to demonstrate that they are being decrypted as part of the context switch. By the 3rd frame app owns the CPU, and as such all of its memory contents are decrypted.

5.4. Performance Analysis

Figure 11 shows how SCS adds to a processes overall run time. The top timing diagram in Figure 11 shows a process running on an unaltered kernel. For the unaltered timing diagram the context switch period is set at 10mS and the context switch overhead (x) is constant at 10μS. The figure does not show context switch period to context switch

![Figure 11: Secure Context Switch Overhead](image-url)
overhead ratio to scale. The bottom timing diagram represents a kernel running with SCS. Since SCS must encrypt and decrypt memory pages belonging to the protected process at each context switch coincident to a PCPP process, the context switch overhead increases by $\beta$. In the figure we set $\beta = x$. The total run time delay for a process protected by SCS is equal to the number of times the protected process context switches in (takes ownership of the processor) and out (relinquishes the processor), which we call $n$, times $\beta$. In the figure the protected process context switches out 2 times and context switches in 2 times, so $n$ is 4 in this case. As such, the total run time increase for the PCPP application attributable to SCS is $4\beta$. For the case in the diagram, although the context switch overhead doubled due to the use of SCS, the total impact to the protected processes run time is a fraction of the run time for the same process running without SCS.

$\beta$ increases linearly with the amount of memory which must be encrypted or decrypted during a context switch. Our current SCS implementation encrypts/decrypts all of the pages which store the PCPP processes executable code, stack, and heap. We do not encrypt pages which store shared libraries and we do not encrypt pages which store memory mapped files.

5.5. SCS Run time Overhead

We have developed a set of equations to predict run time of SCS protected applications, $t_{scs}$, and non-SCS protected applications, $t_{norm}$. We use these two run time results to compute the overhead related to using SCS protection, $t_{scs}/t_{norm}$. It is important to note that $t_{scs}/t_{norm}$ does not take into account or attempt to estimate the number of times slices that a process will wait after being context switched out. As such, neither $t_{scs}$ nor
\( t_{\text{norm}} \) correctly predict the exact program run time as standalone equations. However, we assume the wait time is the same for both cases and it therefore factors out when computing the run time overhead, \( t_{\text{scs}}/t_{\text{norm}} \).

**Table 11: Secure Context Switch Run Time Terms**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_{\text{scs}} )</td>
<td>Run time with SCS protection</td>
</tr>
<tr>
<td>( t_{\text{norm}} )</td>
<td>run time without SCS protection</td>
</tr>
<tr>
<td>( e )</td>
<td>number of complete times slices needed to complete execution</td>
</tr>
<tr>
<td>( T )</td>
<td>period of one time slice</td>
</tr>
<tr>
<td>( \eta )</td>
<td>adjusted number of times slices after adding demand encryption/decryption overhead</td>
</tr>
<tr>
<td>( t_{\text{cs}} )</td>
<td>Context switch time without SCS protection</td>
</tr>
<tr>
<td>( t_{\text{in}} )</td>
<td>time for protected application to context switch in</td>
</tr>
<tr>
<td>( t_{\text{out}} )</td>
<td>time for protected application to context switch out</td>
</tr>
<tr>
<td>( t_{\text{decrypt}} = t_{\text{encrypt}} )</td>
<td>time to encrypt/decrypt one page of memory</td>
</tr>
<tr>
<td>( \rho )</td>
<td>average number of pages of memory accessed per time slice</td>
</tr>
</tbody>
</table>

Equations 10 - 12 below predict the run time for SCS protected processes. Table 11 provides definitions for the terms used in equations 10 - 12.

\[
t_{\text{scs}} = eT + \eta_{\text{out}} + \eta_{\text{in}}
\]

Equation 10 provides an approximation of the run time required to run an application with SCS enabled. Equation 10 starts by representing the amount of time a program needs to run in terms of context switch time slices, or epochs, \( e \), times the period of 1 epoch, \( T \). As such \( eT \) is the amount of run time the program would require if it were running without any interruption, i.e. in an environment free of context switches. We represent this basic run time as \( eT \) because \( e \) provides a minimum number of \( T \) length time slices required to run the program. All other terms in equation 10 are overhead associated with context switching.
The second term of equation 10, $\eta_{out}$, adds the context switch time for $\eta$ context switch outs, i.e. relinquishing the CPU. The third term of equation 10, $\eta_{in}$, adds the context switch time for $\eta$ context switch ins, i.e. regaining control of the CPU. In the second and third terms $\eta$ is the adjusted number of times slices used by the application. We adjust $\eta$ to account for the time needed for an ordinary context switch, for the time needed to encrypt all decrypted pages during the context switch out process, and for the extra time needed to decrypt the same pages during context switch in.

$$t_{out} = t_{in} = t_{cs} + \rho_{\text{encrypt}}$$  \hfill 11

Equation 11 shows the relationship between $t_{in}$ and $t_{out}$. For our SCS implementation $t_{in}$ and $t_{out}$ are equal because on each context switch in we decrypt all pages assigned to the protected application and on context switch out we encrypt all pages assigned to the protected application. Since we use the same AES encryption routine for encryption and decryption, encrypting $\rho$ pages takes the same time as decrypting $\rho$ pages.

$$\eta = \left( \frac{2t_{cs} + 2e\rho t_{decrypt}}{T} \right) + e$$  \hfill 12

Equation 12 estimates the adjusted number of time slices a SCS protected process will require. Equation 12 adds the minimum number of uninterrupted time slices, $e$, to the number of extra time slices resulting from the extra time required for basic context switching and extra time required for encrypting and decrypting protected pages during the context switch. The first term of equation 12 calculates the extra times slices required for context switching and encryption/decryption by first summing the time for 2 ordinary context switches and 2 sets of page encryptions/decryptions and then dividing by the
period of one time slice, T. We use 2 context switch times and 2 encryption/decryption
times because context switch come in pairs, one at the beginning of the time slice and one
at the end of the time slice.

\[ t_{\text{norm}} = eT + \eta_{\text{out}} + \eta_{\text{in}} \]  

Equation 13, which computes \( t_{\text{norm}} \), looks just like equation 10. The only different
is the definitions of \( \eta \), for the number of adjusted context switches, and the definitions for
\( t_{\text{in}} \) and \( t_{\text{out}} \).

\[ \eta = \left( \frac{2t_{\text{cs}}}{T} \right) + e \]  

Equation 14 shows the definition of \( \eta \) for the non-SCS protected case. Since,
without SCS protection there is no encryption/decryption of memory pages this term is
removed relative to equation 12 which defines \( \eta \) for the SCS protected case.

\[ t_{\text{out}} = t_{\text{in}} = t_{\text{cs}} \]  

Equation 15 defines \( t_{\text{in}} \) and \( t_{\text{out}} \). Again since there is no encryption/decryption
required for the non-SCS protected case the encryption time term is removed relative to
equation 11 which defines \( t_{\text{in}} \) and \( t_{\text{out}} \) for the SCS protected case.

\[ \text{scs\_overhead} = \frac{t_{\text{cs}}}{t_{\text{norm}}} \]  

Equation 16 defines the overhead for SCS protected applications when compared
to the same application running without SCS protection.

To use equation 16 to predict the SCS run time overhead we need to know, \( t_{\text{encrypt}} \),
the time required to decrypt a 1 memory page for the x86_64 Linux architecture which
uses 4096 byte pages. We also need to know the time to context switch, $t_{cs}$, when SCS is not used.

<table>
<thead>
<tr>
<th>Table 12: Run time Overhead Constants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Item</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>$t_{encrypt}$</td>
</tr>
<tr>
<td>$t_{cs}$</td>
</tr>
</tbody>
</table>

We measured both $t_{cs}$ and $t_{encrypt}$ on our PCPP development platform. For $t_{encrypt}$ we timed a loop which encrypted a single page 200000 times. We then took this total time and divided by 200000 to get the time per page. Because the loop included so many encryptions the final result will include some time for context switching. We left this overhead in our number to be pessimistic. Our development platform is an AMD Athlon 6000+ processor (3GHz clock frequency). We were unable to find comparative AES encryption throughput values for the exact same processor, however, we were able to cross check our numbers against numbers taken by a third party [38] on an AMD Athlon 3000+ (with 2.25 GHz clock frequency) and an Intel Pentium 3 running at 3.2 GHz. Converting our results to the same units yields similar performance to the results from the Athlon and Pentium processors cited above.

To learn $t_{cs}$ we used the LMBench bench mark suite [36]. LMBench measures context switch by ping-ponging between multiple processes a known number of times and then dividing the total run time for this ping-pong routine by the number of ping-pongs. LMBench repeats this process with 2, then 8, and then 16 processes all running concurrently. The scheduler used in Linux 2.6 is supposed to have order 1 complexity, meaning it runs in constant time and is independent of the size of the process or the
number of processes waiting to run. LMBench reports that the context switch time increases as the number of waiting processes increases and as the size of the processes increases. The authors of LMBench claim this is a result of cache latency times which cannot be removed from the measurements. Because the results from LMBench varied we chose to use the fastest reported context switch time, which is 1 $\mu$s. We used the fastest time because this is a worst case for highlighting SCS overhead, since with faster context switching the SCS overhead will have more pronounced impact on overall run time.

We used equations 10-16 and the values from table 12 to plot the expected SCS overhead for programs with 100, 200, and 300 pages of memory. These results are shown in Figure 12. The results from Figure 12 predict that as a protected process uses more memory the overhead associated with SCS protecting that memory increases. For
smaller processes this overhead is bearable, but as process memory use grows the SCS overhead grows rapidly.

We can compare these predicted results to the experimental results from Figure 13 by converting the number of pages to bytes and then comparing to the results of Figure 12 to the top most curve of Figure 13 which shows the ration of SCS run time to non-SCS run time. For the 100 pages case, which is 409600 bytes, the predicted overhead converges to approximately 37%. Looking at Figure 13 we see a similar value in the 400000 byte range of the curve. For the 200 pages case, which is 819200 bytes, the predicted overhead converges to approximately 90%. Looking at Figure 13 we see a similar value in the 800000 byte range of the curve. For the 300 pages case, which is 819200 bytes, the predicted overhead converges to approximately 155%. Looking at Figure 13 we see a similar value in the 1600000 byte range of the curve.

All the predicted results from Figure 12 were computed assuming, T, the context switch time slice period is 20 mS. If we increase T the predicted overhead resulting from SCS protection decreases. If decrease T the predicted overhead resulting from SCS protection increases. Remember that T represents the average time slice length a program experiences when executing rather than a target times slice length used to configure the operating system’s scheduler at operating system compile time. The T used to calculate SCS overhead is affected not only by the target time slice length, but also by how often the program relinquishes control of the CPU for other reasons including such reasons as to allow I/O to complete or to execute system calls which require a context switch.
The run time overhead values derived from equation 16 are theoretical. As it is very difficult to control the number of context switches and the length of uninterrupted run time between context switches it is very difficult to use equation 16 to predict actual run time results. Equation 16 is provided to demonstrate how run time results will vary as the number of pages encrypted and decrypted per context switch period changes. Actual results will of course also depend on the number of context switches and the length of uninterrupted run time between context switches.

5.6. Experimental Evaluation

We collected empirical data to compare the run time of a SCS protected application to that of a non-protected application. All data was gathered running a program called *matrix* which takes \( m \) and \( n \) size parameters from the command line and then dynamically allocated memory to store the \( m \times n \) matrix. Finally, *matrix* fills the matrix by writing an index value to each location of the matrix. We chose *matrix* as our test subject because it easily demonstrates the affect of increasing allocated memory size on an application’s run time when using SCS protection.
When running to collect data we always ran with square matrices, i.e. $m = n$. We ran matrix in a nested loop. The outer loop incremented the matrix size from 10 to 1000 by 10. The inner loop ran matrix 1000 times for each matrix size. In total we ran matrix 1 million times with SCS encryption enabled and 1 million times with SCS disabled for comparison. Each run was timed individually and the result was recorded in a file.

After execution we processed the run time data to prepare for plotting. First, we trimmed the run time outliers from the data. We trimmed the 2 smallest run times and 2 largest run times. We performed this trimming step because we found that some matrix size run time averages were skewed by individual runs which took orders of magnitude longer to execute than their peers. We conjecture that this occurs when a process is subjected to more than the ordinary number of context switches. This can occur if the machine becomes busy with another task. While we intended for the host to be lightly loaded while running the matrix program some interruptions still occurred. Next we

![Figure 13: Measured Run Time SCS Vs. Unmodified Kernel](image-url)
averaged the remaining 996 samples for each matrix size. Figure 13 offers a run time comparison between the SCS protected matrix application and the non-SCS version. Figure 13 shows 3 curves. The bottom two curves, drawn with solid lines, show the average run time for both the SCS and non-SCS runs plotted against the total memory size of the dynamically allocated matrix. The dynamically allocated matrix uses 4 bytes per location; there are \( m \times n \) locations. The higher of the two run time curves plots the SCS run times and the lower curve shows the run times for the non-SCS runs. Both curves show increasing run time as the size of the matrix increases. This is the expected behavior since matrix is an order-n complexity application. The separation between the SCS curve and the non-SCS curve, the difference in run time, also increases as the matrix grows larger. This occurs because the SCS overhead increases with the size of processes memory image while the context switch time for non-SCS applications is constant. The increasing separation between the run times is further illustrated by the top most curve of Figure 13, drawn with a dashed line, which shows the SCS run time divided by the non-SCS run time plotted against the total memory size of the dynamically allocated matrix. The Y-axis for this curve is labeled on the right hand side of the graph. For the collected data set the SCS runs range between 0.9 and 2.5 times slower than their non SCS counter parts. This ratio increases as the applications allocated memory size increases.

We propose a method, in chapter 6, for dramatically improving SCS performance.

5.7. Implementation Specifics

Implementing SCS requires patches to the Linux operating system. These patches occur in two primary locations. First, PCPP uses a custom binary format handler. This
binary format handler loads PPELF executables into memory for execution. PPELF is an executable guard implementation for the Linux operating system, see chapter 4 for a detailed description. In addition to loading the PPELF application the PPELF binary format handler fills certain needed data structures in the new processes task structure including setting a Boolean value to indicate the process is a PCPP process, creating an AES instance for the process, creating a key cache for the process, and finally inserting the executable’s encryption key into the key cache.

In addition to the new code in the binary format handler SCS requires changes to the Linux kernel scheduler in the routine which implements the context switch. Here we insert code to encrypt the memory pages of outgoing PCPP processes and decrypt the memory pages of incoming PCPP processes. When an outgoing PCPP process is encountered, we learn this by checking for the Boolean set by the PCPP binary format handler, SCS loops through all of the memory pages assigned to the protected process encrypting pages which map the executable code and data segments, the process’s heap, and the process’s stack. These memory pages are available via the application’s task structure.

Since the context switch routine runs with interrupts disabled there are several challenges when implementing this portion of SCS. First, the Linux kernel contains a large number of API which are available for use in kernel programs. However, many of these API are not available when interrupts are off since they may require the active process to sleep. Since we are working during the context switch allowing our process to sleep would hang the operating system as there would be no way to switch to another process. Furthermore, sleeping would conflict with the security requirements of SCS,
since the whole intent of SCS is to encrypt/decrypt the protected processes context before another process gains control of the CPU. We were impacted in two primary areas by the lack of access to existing kernel API and processes. First, the Linux Kernel contains a built-in AES implementation. We were not able to use this AES implementation, and were therefore forced to add our own. Second, we were not able to use the Linux page fault system to convert virtual addresses to physical addresses. Instead we implemented our own virtual to physical address mapping API.

We used a counter mode implementation of AES to enable block-wise random access. The actual AES encryption and decryption routines were a modified copy of the AES implementation available in the Linux kernel. We copied and modified the AES routine to enable its use within the context switch routine. Most of changes were minor, simply removing calls to the Linux `might_sleep`. One change was significant, we were unable to use the hand optimized assembly routines which are available for our architecture, x86_64, and instead used a C based implementation. Our AES key length was 128-bits for our experiments, though 192-bit and 256-bit key lengths are supported.

As we mentioned in section 3B we wrote a memory snooper to demonstrate Bob’s ability to access the contents of Alice’s processes. We used this snooper to monitor the contents of the PCPP processes memory with SCS enabled. This allowed us to confirm that the protected application’s memory contents were properly encrypted when the process was sleeping.

We implemented SCS as a series of patches to version 2.6.20.6 [28] of the Linux operating system. We performed our experiments on a 64-bit AMD platform running at 3 GHz with 2 GB of DDR2 DRAM running at 800 MHz. During our experiments the one
of two CPUs was disabled to emulate the performance on a single processor platform.

5.8. Conclusion

Secure Context Switch (SCS) encrypts a process’s context, i.e. its executable code, data segments, and heap, when the process relinquishes control of the processor and decrypts the context when a process regains control of the processor. By placing this encryption/decryption step inside the operating system’s context switch routine the context is isolated from any other process running on the same platform. We illustrated that SCS does result in a performance penalty which we quantified with mathematically predicted results and empirical results from our working SCS implementation which we realized as a patch to the Linux operating system. With the implementation discussed in this chapter SCS overhead limits the practical amount of memory SCS can protect to less than 4 MB.

Chapter 6 introduces Demand Encryption/Decryption, a method for significantly reducing SCS overhead while maintaining the same level of security by only decrypting memory pages when they are accessed and thereby limiting the re-encryption work required in the context switch to those pages which were decrypted due to access.

Secure Context Switch offers a novel method for isolating application memory from other processes running on the same execution platform. SCS reduces this isolation problem down to one of protecting a set of encryption keys. Said another way, the SCS keys must be protected for SCS to be effective. Chapter 8 of this thesis details our PCPP
key protection methodology which further reduces the problem to one of protecting a single key and then offers a solution to protecting a key on an open white box platform via a set of operating system enhancements.
6.1. Introduction

The Secure Context Switch (SCS) building block described in chapter 5 decrypts memory pages belonging to a protected application immediately before the application gains control of the CPU regardless of whether those pages are to be accessed. Additionally, SCS encrypts all memory pages belonging to the protected application when the application relinquishes control of the CPU. We have shown in chapter 5 that this leads to increasing run time overhead as protected applications use more memory.

The run time overhead associated with SCS caused us to look for a more efficient methodology for encryption and decryption of PCPP protected application memory pages. Demand encryption/decryption is a solution to this problem which offers a significant speed up over our previous SCS implementation by dramatically decreasing the number of pages which must be encrypted and decrypted.

In addition to speeding up SCS demand encryption/decryption simplifies implementation of the Executable Guard (chapter 4) building block. The PPELF implementation discussed in chapter 4 required modification of Linux GNU LIBC interpreter to decrypt the PPELF contents after they had been loaded into memory. With demand encryption/decryption the PPELF contents can be left encrypted after loading into memory. Any pages which must be read by the GLIBC interpreter will be decrypted
on demand as they are accessed by the interpreter.

6.2. Demand Encryption/Decryption Design

With demand encryption/decryption protected memory pages are decrypted only as they are accessed. For any given time slice, a page that is not accessed is not decrypted and any page that is accessed is decrypted immediately before the first access to that page. Pages are still encrypted in the scheduler’s context switch routine, however; only pages which were accessed and consequently decrypted in the previous time slice must be encrypted in the context switch routine.

Each page in memory has a 64-bit header which stores information about the state of the memory page. This header contains fields which indicate if the page is present in memory, dirty (needs to be written to disk), read/write permissions, and other information needed by the paging system. For the x86_64 architecture on version 2.6.20.2 [28] of the Linux kernel 11 of the 64 bits in the page header are in use. The remaining 53 bits are reserved for future use. We add a bit called plaintext to the page header information. The plaintext bit indicates that the page is currently decrypted when set and indicates that the page is currently encrypted when cleared. Table 13 shows the modified contents of the x86_64 page header with the plaintext bit added as bit 9.

<table>
<thead>
<tr>
<th>Table 13: Modified x86_64 Page Header Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
</tr>
<tr>
<td>NX</td>
</tr>
</tbody>
</table>
In order to protect memory pages when other applications own the processor all protected memory pages must be encrypted before relinquishing the CPU. When the protected process relinquishes control of the CPU it will check the header of each page assigned to the protected application to find pages with a set plaintext bit, indicating the page is currently plaintext. These plaintext pages are encrypted during the context switch process. Figure 14 illustrates this concept. On the left hand side of the diagram we see a system memory diagram with 3 pages, labeled App. Page 1, 2, 3, which are owned by the protected application App. In this case, immediately before the application relinquishes control of the CPU, labeled as context switch out, App. Page 1 and App. Page 3 are plaintext and App. Page 3 is encrypted. During the context switch out process App. Page 1 and App. Page 3 are encrypted, and consequently the left hand side of the diagram shows all three pages encrypted after the context switch process is complete. At this point all pages belonging to the protected application are secure and it is safe to relinquish the processor to another process.

**Figure 14: Demand Encryption Decryption Context Switch out**

In order to protect memory pages when other applications own the processor all protected memory pages must be encrypted before relinquishing the CPU. When the protected process relinquishes control of the CPU it will check the header of each page assigned to the protected application to find pages with a set plaintext bit, indicating the page is currently plaintext. These plaintext pages are encrypted during the context switch process. Figure 14 illustrates this concept. On the left hand side of the diagram we see a system memory diagram with 3 pages, labeled App. Page 1, 2, 3, which are owned by the protected application App. In this case, immediately before the application relinquishes control of the CPU, labeled as context switch out, App. Page 1 and App. Page 3 are plaintext and App. Page 3 is encrypted. During the context switch out process App. Page 1 and App. Page 3 are encrypted, and consequently the left hand side of the diagram shows all three pages encrypted after the context switch process is complete. At this point all pages belonging to the protected application are secure and it is safe to relinquish the processor to another process.
When a process regains control of the CPU no pages are immediately decrypted. This is illustrated in Figure 15. Here we show the system memory contents with App. Page 1, 2, 3 all encrypted. Immediately, after the context switch process App. Page 1, 2, 3 all remain encrypted. With demand encryption/decryption pages are decrypted on demand when they are accessed for the first time after each context switch. Since, the TLB (translation look-aside buffer) is cleared during each context switch we know that every page read by the PCPP protected program after any context switch will cause a page fault. The page fault handler is altered to call the page decryption routine if the page’s plaintext bit is not set. Figure 16 illustrates the decryption process. In the figure, the protected application App. is executing instructions without regard to when the last context switch occurred or when the next context switch will occur. When an instruction accesses a memory area which has not been previously accessed in the current time slice the access will cause a page fault. This is the ordinary behavior regardless of whether

![Diagram](image-url)

**Figure 15: Demand Encryption/Decryption Context Switch in Behavior**
PCPP is protecting the application or not. The page fault occurs because after each context switch the TLB must be flushed to avoid virtual address confusion due the fact that most Linux processes have overlapping virtual addresses. When the page fault occurs the operating system translates the virtual address to a physical address and then checks if the page is in memory or not. Ordinarily, i.e. without demand encryption/decryption, once the page is in memory control returns to the program. In the demand/encryption case the plaintext bit is also checked. If the plaintext bit is set, the operating system may return to application execution. If the plaintext bit is cleared the page is first decrypted and then control is returned to the executing application.

There are two important page fault scenarios to consider. In the first scenario, the required page is present in memory and the page fault handler only needs to translate the virtual address to a physical address before returning control the program which generated the page fault. In the second scenario, the required page is not present in memory. In this case, the page fault handler must have the page loaded from disk before

**Figure 16: Decryption on Page Fault**
address translation can occur.

It is important to note that when a page fault occurs the operating system does not context switch away from the protected application, rather the page fault is handled by a function call to the page fault handler which runs without a context switch and then returns back to the code which generated the fault on completion. This is important because it means it is safe to leave the protected application’s context decrypted while the page fault exception is handled.

If the page fault handler finds that the page is not in memory a context switch will occur to allow time for the page to be copied from disk to memory. While the protected application sleeps, the page which caused the exception will be copied from disk to memory. PCPP files are always encrypted when stored on disk and the loaded page remains encrypted once in memory. At some point after the page is copied into memory another context switch will occur, giving CPU control back to the protected application. The first instruction run in this new time slice is the instruction which originally caused the page fault. This instruction will cause a second page fault. This time the page will be in memory and will be decrypted as described in the first scenario above. Please note, that for this example, when the operating system preempts the protected application and context switches it out of control of the CPU all of the protected applications pages, those with plaintext bit asserted, are encrypted as with any other context switch.

6.3. Predicting Demand Encryption/Decryption Overhead

Demand paging, translation look-aside buffers, and caching mechanisms take advantage of most processes tendency to operate from the same portions of code and
access the same areas of memory for relatively long periods of time and repeatedly. Both
demand paging and caching offer significant performance improvements to most
processes. The demand encryption/decryption methodology also benefits from this
locality property to provide significant speed-up over the SCS implementation discussed
in chapter 5.

\[ t_{ded} = eT + \eta'_{out} + \eta'_{in} + \eta \rho t_{decrypt} \]  

We developed equation 17 to predict the run time for a demand
encryption/decryption protected application. Equation 17 is very similar to equation 10
for predicting SCS run time, which was introduced in chapter 5. The only difference is
the added 4th term, \( \eta \rho t_{decrypt} \), which adds in the time it takes to decrypt accessed pages,
which now occurs in the page fault handler rather than in the context switch in routine.
The other terms of equation 10 do not change for demand encryption/decryption though
the values used for the terms do change. There are two primary changes. First, \( t_{out} \) no
longer equals \( t_{in} \). For SCS the number of pages decrypted during context switch in, \( \rho \),
matches the number of pages encrypted during context switch out. For demand
encryption/decryption the number of pages decrypted during context switch in is zero,
since decryption now takes place in the page fault handler (this decryption time moves to
the 4th term of equation 17). For demand encryption/decryption \( t_{in} \) is now equal to \( t_{cs} \), the
context switch time without protection.

\[ t_{in} = t_{cs} \]  

For demand encryption/decryption \( t_{out} \) is \( t_{cs} \) plus the time required to encrypt the
pages decrypted in the previous time slice.

\[ t_{out} = t_{cs} + \rho t_{encrypt} \]
Table 14: Demand Encryption/Decryption Run Time Terms

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_{\text{ded}}$</td>
<td>Run time with demand encryption/decryption protection</td>
</tr>
<tr>
<td>$t_{\text{norm}}$</td>
<td>Run time without demand encryption/decryption protection</td>
</tr>
<tr>
<td>$e$</td>
<td>number of complete times slices needed to complete execution</td>
</tr>
<tr>
<td>$T$</td>
<td>period of one time slice</td>
</tr>
<tr>
<td>$\eta$</td>
<td>adjusted number of times slices after adding demand encryption/decryption overhead</td>
</tr>
<tr>
<td>$t_{\text{in}}$</td>
<td>time for protected application to context switch in</td>
</tr>
<tr>
<td>$t_{\text{out}}$</td>
<td>time for protected application to context switch out</td>
</tr>
<tr>
<td>$t_{\text{decrypt}} = t_{\text{encrypt}}$</td>
<td>time to encrypt/decrypt one page of memory</td>
</tr>
<tr>
<td>$\rho$</td>
<td>average number of pages of memory accessed per time slice</td>
</tr>
<tr>
<td>$s$</td>
<td>file size in bytes</td>
</tr>
</tbody>
</table>

Table 14 provides definitions for the terms used in equations 17-20.

To properly apply equation 17 we must be able to predict the number of pages which will be decrypted in a given time slice. To do this we gathered data to estimate the significance of the locality property. Specifically, we patched version 2.6.20.2 of the Linux kernel to gather information about the number of times processes context switch and to gather the number of times processes page fault to translate a virtual address to a physical address.

Table 15: Virtual Address Translation Data

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average virtual address translations per time slice</td>
<td>32.3</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>280.2</td>
</tr>
<tr>
<td>Max virtual address translations in a single time slice</td>
<td>2258</td>
</tr>
</tbody>
</table>
Table 16 shows the average number of virtual address translations per time slice, the standard deviation for the number of address translation per time slices, and the maximum number of address translations per time slice. This data was collected by enabling the monitor function described above for a period of time on our test machine. During the collection period we ran many applications on the machine including starting and stopping internet browsers, word processors, spreadsheet programs, various GUI’s, and assorted command line programs. During the collection period a total of 1268 processes started and stopped, using 29266 time slices, and performing 945563 virtual to physical address translations. The virtual address to physical address translation count only includes translations performed by the operating system and does not include TLB hits. This number of address translations matches the number of times the demand encryption/decryption code would need to decrypt a memory page.

\[
\text{overhead} = \frac{t_{\text{ded}}}{t_{\text{norm}}}
\]

Figure 17 and Figure 18 show the calculated run time overhead for various process lengths, where overhead is the percent increase in run time of a program running with demand encryption/decryption protection compared to the same program running without protection, as defined by equation 20.

For both Figure 17 and Figure 18 we set the time slice period, T, to 10 mS. The different curves in the figures represent different values of \( \rho \) from equation 10, the average number of address translations per time slice which equals the number pages which will need decryption and encryption per time slice. Figure 17 plots \( t_{\text{ded}} \) overhead values for \( \rho \) equal to 1, 5, and 10 translations per time slice. Figure 18 plots \( t_{\text{ded}} \) overhead
for \( \rho \) equal to 10, 32, 50, and 100 translations per time slice. Both graphs plot \( t_{\text{ded}} \) overhead as the required run time, measured in time slices, increases.

The curves in Figure 17 and Figure 18 converge to a stable overhead value as the required run time increases. This happens because for very small run times the encryption and decryption overhead is very large relative to the required run time. As the run time increases the encryption and decryption time contributes relatively less to the overall run time.

Figure 17 shows the run time overhead for program with small memory bandwidth requirements or high locality. All three curves of Figure 17 converge to less than 10% overhead. The smallest of course would be the overhead for a process which averages just 1 translation per time slice. Averaging 1 translation per time slice is a
special case since that would require a program to run out of one page for its instructions and never access any pages for data. This is possible but unlikely. Ordinary programs have at least one page for instructions, one page for a data segment from the original program, one page for a heap, and one page for the program’s stack, or all told at least 4 pages. Of course having a minimum of 4 pages does not mean they must all be accessed in a given time slice. However, we feel the curve for averaging 5 page accesses per time slice represents a more likely minimum overhead. For 5 pages per time slice the overhead converges to approximately 3%.

Figure 18 shows run time overhead for programs with larger memory bandwidth requirements or less memory locality; we call this medium bandwidth to recognize that...
there are classes of programs which access many more pages of memory per time slice, such programs may not be good candidates for demand encryption/decryption protection, although, we know of no better alternative.

Figure 18 includes the curve showing the run time overhead for a program with 32 address translations per time slice. This curve represents the average number of translations per time slice we saw in our collected data set (see Table 16). For an average of just over 32 translations per time slice the run time overhead converges to just over 20 percent.

The run time overhead increases with the number of address translations per time

![Figure 19: Demand Encryption/Decryption Overhead with 100mS Time Slice](image-url)
slice. By 100 translations per time slice the run time overhead is close to 90%. As mentioned above the curves for both Figure 17 and Figure 18 were derived using an average time slice of 10 mS. In fact this number is quite small. The Linux operating system adjusts processes times slice allotments based upon process priority and upon how I/O bound the process is, with I/O bound processes being given larger priority to minimize the interruptions to the user. The minimum time slice value for a process in the 2.6.20.2 version of the Linux kernel is 5 mS. The default time slice is 200 mS and the maximum time slice is 800 mS. A time slice in the Linux operating system is different from what we used as a time slice in equation 10. In equation 10 a time slice represents an uninterrupted segment of time in which our protected process owns the CPU. It should be seen as an average value. With the equation 10 definition a process which relinquishes the processor to wait for I/O or other systems calls to complete has completed a time slice. In other words I/O bound processes will have small average time slice lengths, while CPU bound processes will have average time slices which approach the amount of time allotted from the operating system, which varies with priority.

Linux defines the time slice [29] as the amount of time a process is allowed to stay on the active run queue. Being on the active run queue does not mean running. A process may context switch in and out many times without using all of its allotted time and therefore stay on active run queue across many context switches. After this time slice expires the process is moved to a queue holding all processes with expired time slices. Once the active run queue is empty all processes from the expired run queue are moved back to the active queue.
Figure 19 shows the run time overhead for 4 values of \( \rho \), 32, 50, 100, and 323 with the average time slice period, \( T \), set at 100mS. With this much larger average time slice the run time overhead decreases considerably. For instance, for an average of 100 address translations per time slice the overhead drops from close to 90\% for \( T=10\text{mS} \) to approximately 6\% for \( T=100\text{mS} \). We also included a curve for 323 average address translations per time slice in Figure 19. This value represents the average number of address translations per time slices plus 1 standard deviation from our sampled data, from table 16. For 323 address translations per time slice the run time overhead converges to approximately 22\%. We did not show the same curve for 10 mS times slices because at this small average time slice value the run time overhead converges to slightly over 500\%, which would not fit on the chart.

The run time overhead values derived from equation 20 are theoretical. As it is very difficult to control the number of context switches and the length of uninterrupted run time between context switches it is very difficult to use equation 20 to predict actual run time results. Equation 20 is provided to demonstrate how run time results will vary as the number of pages encrypted and decrypted per context switch period changes. Actual results will of course also depend on the number of context switches and the length of uninterrupted run time between context switches.

6.4. Demand Encryption/Decryption Implementation Results

We implemented and evaluated Demand Encryption/Decryption on a computer running the Linux operating system version 2.6.20.2 with OS patches to implement the demand encryption/decryption functionality. This Linux computer was equipped with a
AMD Athlon 64 X2 6000+ CPU with 2GB of DDR-800 DRAM. All results discussed below were gathered with the OS set to use just one CPU. Evaluation results from this implementation showed considerable improvement, compared to the original SCS implementation discussed in chapter 5 in run time performance and in the total supported memory size.

Figure 20 shows three curves labeled Demand, SCS, and Unaltered OS. Each curve shows average run times for our matrix program described in chapter 5 as memory usage increases. The Demand curve shows run times using demand encryption/decryption, the SCS curve shows run times when using our original SCS architecture, and the Unaltered OS curve shows run times when using an unaltered Linux OS. The memory usage ranges from 400 bytes to 4 million bytes for all three cases. For this memory size range Demand encryption/decryption is always either as fast as or faster than the original SCS architecture. In fact this memory range represents all of the

![Figure 20: Demand vs. SCS vs. Unaltered OS Run Times](image)
supported memory size range for the original SCS architecture, while demand encryption/decryption is capable of supporting much larger memory usage as we will demonstrate below.

Figure 21 shows relative run times of demand encryption/decryption compared to the unaltered OS version and the original SCS compared to the unaltered OS version. In other words, figure 21 shows the average run time of each plotted size for demand encryption/decryption, or original SCS, divided by the average run time for the unaltered OS version. Again we have only plotted results for up to 4 million bytes memory usage since this is an effective maximum for the original SCS architecture. Figure 21 shows that for this small memory size range demand encryption/decryption is nearly always faster than the original SCS.

![Demand vs. SCS Run Time Overhead Relative to Unaltered OS](image)

**Figure 21: Demand vs. SCS Run Time Overhead Relative to Unaltered OS**
Figure 22 shows a scatter diagram of run times for a 100 runs of a program we call *array* which allocates an array of integers, in this case 25 million integers (each integer uses 4 bytes), then writes a value (the index number) in each array location, and finally re-loops through the array validating the array contents. Each run plotted allocated an array of 2.5 million integers, which since one integer uses 4 bytes on our host machine, equates to a 10 million byte heap. The primary difference from run to run which accounts for the variation in run times is the number of pages which were encrypted and decrypted during execution which is the value shown on the x-axis. Figure 22 demonstrates very clearly the linear relationship between program run time and the number of pages that are encrypted and decrypted during execution. The variation of the total number of pages encrypted and decrypted from run to run occurs because of the varied frequency of context switches and varied time relative to the start of execution of the context switches. As the number of context switches increases the number of encrypted pages and decrypted pages also increases. Additionally, context switches
which occur early in this program life cycle may occur before the entire heap is allocated and therefore have less impact on run time than a context switch which occurs after the entire heap has been allocated and filled with data.

Figure 23 is similar to Figure 22. In this case, we again ran the array program 100 times and plotted the run time versus the total number of encrypted and decrypted pages. This time the array program was set to allocate an array of 25 million integers, which consumes 100 million bytes in the heap. Both the run time and the total number of pages encrypted/decrypted appear to vary significantly from run to run. In fact, however, the standard deviation for the points plotted in Figure 23 is significantly smaller than the standard deviation from Figure 22. Again if we exclude the two outliers above 3.3S run time we see a linear relationship of run time to total number of encrypted and decrypted pages. Also, the total number of pages encrypted and decrypted across all the runs is

![Demand Encryption/Decryption Run-Times 100M Bytes](image)

**Figure 23: Demand Encryption/Decryption Run Times 100M Bytes**
very consistent. This consistency comes from the larger array size which leads to larger run time which in turn leads to a more consistent context switch rate.

With the original SCS implementation as memory use increased run time also increased. This trend continued until the time to encrypt and decrypt the protected programs heap, stack, code, and data segments every context switch became too burdensome. With the original SCS architecture the protected program and the entire operating system began to hang when total memory usage reached around 6-8 million bytes. One major advantage of the demand encryption/decryption architecture is that much larger memory usage is supported. The memory sizes used for both Figure 22 and Figure 23 are 3X and 25X larger than the upper limit seen for the original SCS. These sizes do not represent a maximum memory size for the demand encryption architecture. There is no upper limit on memory usage for the demand encryption architecture, instead there is an upper limit on the number of pages which can be encrypted in a given context switch. A practical limit on the number of pages encrypted in a single context switch would be around 1000-2000 thousand pages. This upper limit occurs because the OS will miss servicing timer interrupts if too many pages are encrypted when interrupts are being ignored. If the upper limit is surpassed a program may still run if the context switch routine was further altered to service the timer interrupts from within the context switch routine.

In fact, we saw problems when validating both the original SCS architecture and the demand encryption/decryption architecture with timer interrupts not being serviced properly during PCPP protected program execution. We found that Linux OS keeps soft time rather than relying on a hardware real time clock when the OS is running. The OS
updates its software clock each timer tick. If timer interrupts are missed which occurs when a larger numbers of pages require encryption, the software clock becomes inaccurate because of an assumption that a maximum of only 1 tick will ever be missed when correcting the soft clock. This lead to timer inaccuracies when we initially measured SCS and demand encryption/decryption run times. We worked around this problem by reading our host’s hardware timer directly when measuring run times.

6.5. Conclusion

Demand encryption/decryption is a much faster memory isolation technique than that used in the implementation of Secure Context Switch from chapter 5 while offering the same level of security. It is difficult to predict the exact run time overhead for any given process. However we have shown data which suggests that demand encryption/decryption is a feasible memory isolation technique across many types of processes especially processes which tend to be more CPU bound.

Demand encryption/decryption provides a significant performance boost to the Secure Context Switch (chapter 5) PCPP building block making the overall run time overhead for PCPP much smaller.
Chapter 7

SECURE I/O (SIO)

7.1. Introduction

Secure I/O (SIO) is a PCPP building block which protects files and file access. All files are stored in an encrypted form. Both traditional files stored in non-volatile memory and file mappings which map to ordinary volatile memory are protected by Secure I/O by patching the operating system to decrypt reads and encrypt writes at a generic layer in the read and write implementations, such that all reads and writes are protected regardless of the underlying file system. Finally, files may be excluded from protection by adding file names to a SIO exclusion list.

SIO protects 2 types of file access. First, files may be created or accessed using the POSIX open, create, read, write, and close commands. Second, files may be accessed using the mmap command which maps file contents to virtual memory.

Linux implements a virtual file system (VFS) layer which provides access to many file types via a single interface. For the POSIX read and write commands we add patches to the VFS layer read and write implementations to encrypt data being written to a file and to decrypt data read from files. For memory mapped files, we declare the virtual memory pages assigned to the memory mapped file as encrypted by setting the cryptext bit in each page’s page table entry. This causes the page to be decrypted before
a read access. For memory mapped file writes we intercept write back data and encrypt it just before it is written to disk.

As with the executable files protected with the executable guard, described in chapter 4, SIO files are never stored as plaintext. Permanent encryption of the files stored on the remote host combined with decryption on read access and encryption on read access protect files before, during and after execution.

7.2. Secure I/O (SIO) Overview

The intent of SIO is to protect all file system accesses made via the POSIX open, create, read, write, and close commands and file system accesses made to memory mapped files. We require existing files to be encrypted before use on the remote host. Therefore we always decrypt read data before returning it to the protected application. For writes we always encrypt write data before writing it to the external file system. File systems may map to non-volatile memory, such as hard disk drives, may map to virtual memory, or may map to other custom types of interfaces. SIO treats all file systems as equal regardless of the ultimate type of storage. As such, all data written to a file system is encrypted and all data read from a file system is decrypted regardless of the type of storage.

The goal of SIO is to protect accesses to all the file systems a PCPP application may use. Since it is difficult or impossible to predict which file systems an application will use before execution PCPP requires protection of all file systems available on the remote host. As such, SIO requires encryption and decryption patches for all file systems
implemented on the PCPP host. The SIO patches must exist on the remote host before it may be used for PCPP processing.

The required SIO patches may be implemented by indentifying all available file systems and then implementing decryption/encryption patches for each file system’s read/write implementations respectively. Alternatively, SIO decryption and encryption may be added at a layer which covers most or all of the file systems on the PCPP host, limiting the number of necessary patches. Implementing the decryption and encryption patches at a universal layer makes supporting new file systems easier since no new patches are required when a new file system is created.

For most operating systems the POSIX file I/O commands are implemented as system calls. The two commands which require patches to support SIO are read and write. SIO must encrypt all write data and decrypt all read data. Additionally, reads and writes to memory mapped files must be protected.

For file reads data in the file is assumed to be encrypted, unless the file is marked as excluded from SIO protection. Decryption of the read data should take place as close to the application layer as possible. The POSIX read command takes a file descriptor, a pointer to a buffer where the read data should be deposited, and a size parameter. Data being read may be copied from the file system into intermediate buffers to facilitate DMA transfers. Because these intermediate buffers belong to the operating system data they are accessible by privileged users of the machine. Since, PCPP data should be private from all other users SIO should never decrypt data until it is stored in a location owned by the PCPP application. As such decryption of the read data should only take place after the read data has been copied into the buffer provided to the read system call.
As with reads, write data should only be stored as plaintext in memory owned by the PCPP application. The POSIX write command takes a file descriptor, a pointer to a buffer where the write data resides, and a size parameter. SIO encrypts write data before it is copied from memory owned by the application to memory owned by the operating system. Care must be taken on writes to encrypt the data being written to the file without adversely affecting program functionality. For instance, if the application attempts to access data written to a file by directly accessing the data buffer provided to the write command, without first reading the data back from the file system, the data should not appear corrupted or encrypted. Writes may be implemented in two ways: as buffered writes and as direct writes.

For buffered writes, write data is copied from the user provided buffer into a buffer owned by the operating system. Once data is in the operating system owned buffer the write is queued to wait to be physically copied to the underlying file system. For buffered writes, data must be encrypted before it is copied into any memory location not owned by the protected application. For direct writes, the write data is copied from the user provided buffer directly to the file system. In this case, the write data must be encrypted before this copy step. SIO transforms direct writes into buffered writes by allocating a new buffer to hold the encrypted write data separately from the buffer provided by the application.

SIO also supports protection of memory mapped files. The memory mapped case also requires patches to decrypt file contents for read accesses and to encrypt data before writing to a file. When files are memory mapped the file contents are copied into virtual memory owned by the application and a start address pointing to a copy of the file in
memory is returned to the application. The file contents may be copied immediately into virtual memory or the file contents may be copied into memory on demand (when accessed). SIO must decrypt the file contents immediately after they are copied into the applications virtual memory. Decryption should only occur once the file contents reside in virtual memory owned by the PCPP application. If intermediate buffers are used during the mapping process these buffers should never contain decrypted file contents. Memory contents which are populated immediately upon execution of the memory map command should be decrypted immediately after the memory population process completes. Memory pages which are populated on demand should be setup to be decrypted on demand by setting the pages cryptext bit during the memory mapping process. If a pages cryptext bit is set, that page will be decrypted on access, see chapter 6 for details.

Memory mapped files may be declared as SHARED or PRIVATE which determines the write behavior. When the memory contents associated with a SHARED file are altered the underlying file will also be updated. When the memory contents associated with a PRIVATE file are altered the underlying file is not updated. SIO must encrypt data belonging to SHARED files before it is written to the file. As with the POSIX write case, SIO cannot encrypt the virtual memory page which holds the memory mapped file contents when a write is detected. This may lead to corrupted file accesses after writes. Instead, we must intercept the data when it is being written back to the file system and encrypt it as it is copied to file system buffers.
7.3. SIO Implementation Details

We implemented SIO for the Linux operation system. Figure 24 shows a block diagram of the Linux file system implementation. Each of the file I/O commands, including the POSIX file I/O commands open, create, read, write, and close, and the mmap command are implemented as system call. Each system call has a corresponding implementation in the system call layer. The system call layer implementations call corresponding implementations in the Linux Virtual File System (VFS). The Linux VFS provides a façade for each of the file I/O commands which interfaces to the many different supported file systems. Each file system includes a set of call back functions associated with the needed file operations: open, create, read, write, close, and mmap. When the VFS is called with a given file descriptor the VFS implementation fetches a file structure which contains pointers to the file systems call back functions. The VFS

![Figure 24: Linux File System Implementation](image-url)
implementation then calls the appropriate function to complete the command. The Linux VFS is a convenient location to make SIO changes. Changes made at or above the Linux VFS layer will affect all file systems. Changes made below the Linux VFS layer will not affect all file systems.

As of this writing our SIO implementations provide a proof of concept but are not complete. For the POSIX read command we made changes in the read system call implementation which is above the Linux VFS layer. We inserted a patch which intercepted and decrypted the returned read immediately before returning from the read system call. Since we added the decryption step above the Linux VFS reads from all file systems will be affected.

For the POSIX write command we made changes in the generic buffered write code which sits below the Linux Virtual File System (VFS) layer. For our SIO write implementation we encrypted the write data immediately after it was copied from the protected applications buffer to a VFS buffer used to post the write until it actually completes. Adding our patch at this layer removed the need to make a copy of the buffer and hold this SIO owned buffer until it could be further copied to a VFS buffer or until the write completed in the direct write case. We made this trade off at the cost of not protecting the write data for all file systems. For completeness, further work is needed to patch all other paths to other file systems or to move the write data encryption to a higher layer which will secure all file systems.

For the mmap command we again made changes in the system call layer. Since these changes are above the VFS all file systems are protected. Immediately after the lower level mmap implementation completes and returns the starting virtual address of
the mapped file we update the page table entries for all pages mapping the file to set the page’s *cryptext* bits. Setting the *cryptext* bits causes the page fault system, see demand decryption/encryption in chapter 6 for more details, to decrypt the page the first time it is read. If a file is declared as SHARED, then writes to the in-memory copy of the file result in the same changes being written back to the underlying file. As of this writing we have not implemented support for memory mapped files which require write back.

The above implementation descriptions are adequate to support POSIX read and write as well as memory mapping of files flagged as PRIVATE for the EXT3 [43] file system. Since EXT3 is the most popular file system used for hard disk drive management it serves as a good proof of concept for SIO. To support all file systems the POSIX write patch should be moved to a higher layer and write back capability should be supported memory mapped files with the SHARED flag.

SIO uses a file exclusion list to learn file names which should be excluded from

**PPELF Program**

![Diagram of PPELF Program with SIO Exclusion List](image)

*Figure 25: PPELF Format with Added SIO Exclusion List*
SIO protection. We added this feature for two reasons. First, SIO slows down run time as more data must be encrypted/decrypted for each read/write. By not encrypting all files some of the SIO overhead can be avoided. Second, every file contains different data and has different security needs. Some files contain security sensitive data and therefore must be encrypted, while other files are less sensitive or not sensitive at all and may be left as plaintext.

We implemented the SIO file selectivity as a list of files which should not be encrypted. In other words, any file not on the list is encrypted. We use a list of excluded files, rather than a list of included files, because we prefer to err on the side of caution. If a user forgets to list a file name in the SIO selectivity list it is encrypted. If we had chosen an inclusion based list, i.e. only encrypting listed files, it would be easy to imagine users accidentally leaving filenames out of the list and consequently leaving the forgotten files as plaintext. Using the exclusion based list also makes it easy to handle situations when a program creates a file on the fly, especially, files whose names are chosen at run time.

We added the exclusion list to the tail of the PPELF file format. Figure 25 shows the exact location of the exclusion list within the PPELF file format. It is placed immediately below the tail of the original ELF file and above the AES initialization vector.

The SIO file exclusion list is read as part of the PPELF binary load process. The binary loader grabs the list from the file and then adds the contents of the exclusion list to the protected process’s task_struct in the pepp_struct, see Appendix A.4. The list in the
pcpp\_struct is a simple linked list. Each file is an element of the list and each element contains a next pointer. When the next pointer is NULL the list ends.

Currently, the SIO exclusion list requires exact file name matches to exclude a file. This limitation is a result of our implementation and not a property of the SIO concept. Future SIO implementations could improve upon the supported methods for specifying excluded files. We can foresee use of pattern matching and or use of file attributes to make exclusion decisions.

7.4. SIO Overhead

SIO protection does have a performance cost. The performance impact of SIO is independent of file size. Rather SIO run time overhead depends on the amount of file system traffic which must be encrypted and decrypted. The overhead also differs according how files are accessed.

Files accessed with the POSIX read and write commands will incur delay each time a read or write call is executed. Because we encrypt and decrypt at a layer as close to the application as possible all of the encryption and decryption delay will have a direct impact on the PCPP application. Each time a read is performed the returned read data must be decrypted.

\[
T_{\text{decr}} = S_d t_{\text{decrb}}
\]

Equation 21 defines the time to decrypt data as part of a read command. Since the read command takes as input the size of the read we can estimate the decryption run time impact as the size of the read, \(S_d\), multiplied by the time to decrypt one byte, \(t_{\text{decrb}}\).
This is a first order approximation. Smaller reads will be less efficient to decrypt and will have relative larger run time impact.

\[ t_{wencr} = S_d t_{encrb} \]

Equation 22 defines the time to encrypt data as part of a write command. Since the write command takes as input the size of the write we can estimate the encryption run time impact as the size of the write, \( S_d \), times the time to encrypt one byte, \( t_{encrb} \). This is a first order approximation. Smaller writes will be less efficient to encrypt and will have relatively larger run time impact.

The run time impact associated with protecting memory mapped files can be broken into two parts. First, when a page is memory mapped an SIO patch asserts the cryptext bit for all virtual memory pages assigned to map the file. This operation is miniscule compared to the time to decrypt data on read accesses and encrypt data on write accesses. The run time overhead for decrypting on read access is the same as for decrypting any other demand encryption/decryption page. This time is defined in chapter 6. The time for encrypting pages after write accesses will be similar to the read access times, however, since we have yet to implement this portion we do not have an exact estimate.

7.5. SIO Compared to Encrypted File Systems

SIO is different from encrypted file systems. Encrypted file systems [34], such as ECRYPTFS [42] in Linux provide file system wide protection. Encrypted file systems encrypt all data written to the file system and decrypt all data read from the file system with a single key regardless of which user or which application accesses the data.
Alternatively, SIO works at the application level. SIO files are encrypted with a key known only to the PCPP application. With ECRYPTFS and other encrypted file systems any application with permission to access the file system may access any file stored on that file system, subject to access permissions provided for that file. With SIO only the PCPP application may access the protected file.

With the Linux VFS architecture an application may read a file from the ECRYPTFS partition and then write the file contents to a separate file handle which resolves to an unencrypted file system. Such an action would result in an decrypted copy of the file stored on another partition. Since SIO encrypts all writes regardless of the target file system the act of copying a file from one file system to another would not create a decrypted copy of the file.

7.6. Conclusion

Secure I/O (SIO) is an effective method for protecting files which are used by protected PCPP applications. With SIO file contents are always encrypted while stored on disk, or other non-volatile storage. For the POSIX read command data is decrypted immediately before being returned to the application. For POSIX writes data is encrypted before being copied to the file system and before being copied into any operating system owned intermediate buffer. For memory mapped files all virtual memory pages belonging to the memory mapped file are declared as encrypted to enable demand encryption/decryption, as described in chapter 6, which ensures decryption before reading memory mapped file contents. Memory mapped write back data is encrypted by intercepting the write back data and encrypting it before it is copied to any
operating system owned buffer and or before it is copied to the file system itself. By placing all SIO operating system patches at generic points in the file system implementation SIO protects all file systems with a single set of patches. Permanent encryption of SIO files and protection of file accesses combine to protect files before, during and after execution.

Run time overhead resulting from the SIO protection discussed in this chapter grows as the volume of file writes and reads increases. Programs with more file I/O will see larger delays resulting from SIO, while programs with less file I/O will see smaller delays from SIO.
8.1. Introduction

PCPP intends to isolate applications not only from other ordinary users but also from privileged users. The PCPP architecture and concepts described in the chapters above are intended to be applicable to any operating system; however, to date we have restricted PCPP development to the Linux operating system. Up until now our PCPP implementations have used the Linux Key Retention Service (LKRS) [14] to store the encryption keys. However, as we will show in this chapter, LKRS does not isolate stored keys from other privileged users which may run applications on the same execution host. As such, LKRS does not meet the complete PCPP key protection requirements.

Consequently, we have developed a new key protection methodology which isolates PCPP encryption keys from all other processes running on the same platform, regardless of the privilege assigned to that process. We call this new methodology PCPP key protection. PCPP key protection uses a key cache to hold all of the keys used by the other PCPP building blocks. This key cache is tagged with integrity information during the operating system’s context switch routine, at the point immediately before a PCPP isolated process relinquishes the CPU to another process, and then encrypted with a master key, $k_m$. The master key is then securely stored until the PCPP isolated process regains the CPU, at which point, again during the operating systems context switch...
routine $k_m$ is retrieved to decrypt the isolated process’s key cache.

8.2. Linux Key Retention Service Exploit

LKRS attaches a thread key ring pointer to each thread’s task structure (a structure used by Linux to hold a thread’s context). By examining the LKRS implementation source code we were able to learn the steps required to dereference the thread key ring and traverse a set of pointers which led from the key ring to the stored key. With this knowledge we were able to create an exploit which allows access to LKRS key rings across Linux accounts. The exploit requires root privilege to install a server module. After this module is installed however a user can read the keys of any other user on the system. PCPP requires isolation of keys even from privileged users. As such, this exploit proves LKRS is inadequate for use with PCPP as a key protection system.

Our LKRS exploit had two parts, a client and a server. The server is a Linux kernel module which stays resident in memory and waits to be called by the client via the Linux proc interface. When called the server program loops through all available tasks structures in the system (i.e. the task structure for every task on the system). This is done using an available kernel mnemonic for_each_process. The server loop looks for threads with a valid (not NULL) thread key ring. When a thread key ring is found our kernel module traverses the pointer tree attached to the thread key ring and then prints the contents of each key attached to the thread key ring. Because the server portion of the exploit is a kernel module it bypasses all access controls and is able to access the thread key ring contents of any task in the system.
The client portion of our exploit uses the Linux proc interface to call the server and trigger it to search for active thread key rings. We separated the exploit into a client server configuration for two reasons. First, by calling the server from the client we can avoid putting the exploit code in an endless loop which would waste compute cycles and potentially call attention to the exploit. Second, the server needs to be a kernel module to bypass access controls and therefore requires root privilege to install. However, the client does not require root privilege and can be called from any user account.

We did not implement similar exploits which stole key ring contents from session and process key rings. The exploit code would be slightly different, but just as easily attacked.

Since LKRS relies on the Linux access control system to protect its key rings and PCPP requires isolation of keys even from privileged users LKRS does not provide adequate key protection for PCPP processes.

8.3. PCPP Key Protection Overview

PCPP key protection patches the Linux kernel scheduler’s context switch routine

![Encryption Key Cache](image)

*Figure 26: Encryption Key Cache*
to securely store a PCPP process’s encryption keys immediately before a PCPP process relinquishes the CPU and to retrieve the stored encryption keys immediately before a PCPP process regains ownership of the CPU.

Each building block, Executable Guard (PPELF), Secure Context Switch, and Secure I/O, will have at least one encryption key and likely more than one. We use a key cache to store all of the encryption keys. The key cache uses an index to look-up and store keys as shown in Figure 26. We use a master key, $k_m$, to encrypt the key cache.

Using the $k_m$ to encrypt the key cache reduces PCPP’s protection burden to a single point. Building a fortress around $k_m$ in turn protects all of the collected keys and all of the PCPP executable code and data. This fortress is a modified context switch routine which safely stores $k_m$ during the period the PCPP protected process does not own the CPU.

The master key $k_m$ is used every context switch by SCS and SIO. Context switches always come in pairs. If we assume the PCPP protected application is currently running and about to switch out, i.e. relinquishes control of the CPU, then during the first context switch SCS and SIO encrypt their respective data with keys stored in the key cache and then $k_m$ is used to encrypt the key cache. The second half of the context switch pair is when the PCPP protected application switches in, i.e. regains control of the CPU, SCS and SIO use keys from the key cache to decrypt their respective data. $k_m$ is used for exactly on context switch pair and then replaced. Each time the PCPP protected application switches out a new $k_m$ is chosen.

Figure 27 shows a flow chart of the modified Linux context switch routine. The white boxes represent the functionality of the context switch routine before modification.
and the grayed boxes represent the additional functionality added to protect PCPP encryption keys. The unaltered context switch routine performs two simple steps. First, a pointer to the current processes memory map is changed to point to the memory map of the incoming process. Second, a small block of assembly code is called to swap out the CPU registers from the old process replacing them with the register values belonging to the incoming process. The PCPP updates to the context switch routine introduce two new paths through the context switch routine, one to handle outgoing PCPP processes and one to handle incoming PCPP processes.

The context switch routine is passed two pointers, prev (previous task) and next (next task), when called, each is a pointer to a Linux task structure. The prev task
structure holds the context belonging to the process relinquishing the CPU and the next task structure holds the context of the process which is gaining control of the CPU. We modified the PCPP task structure to contain a variable called pcpp. The pcpp variable is a Boolean which defaults to FALSE. When a PCPP process is launched the executable guard sets the pcpp variable to TRUE.

When prev is a PCPP process the modified context switch routine first encrypts any memory pages belonging to prev which are not currently in an encrypted state (this step is actually part of the SCS PCPP building block which is described in detail in chapters 5). The encryption keys used to encrypt the context are held in a PCPP key cache which is attached to prev’s task structure. When a PCPP process owns the CPU the key cache is in a decrypted state. It is encrypted as the PCPP is context switched out. Before encrypting the key cache we first add an integrity hash value to the key cache. The key cache is then encrypted using a master key, \( k_m \). Finally, \( k_m \) is stored to await retrieval when the PCPP process regains control of the CPU.

When next is a PCPP process, the right hand path of the context switch routine is taken. First, \( k_m \) is retrieved from storage. Next, the key cache is decrypted. After the key cache is decrypted the integrity hash value, stored in the key cache when this PCPP process last relinquished the CPU, is validated. If the integrity hash value is incorrect a PCPP process shutdown commences. This shut down erases all PCPP files, overwrites all PCPP memory pages, and then kills the PCPP process. If the integrity hash value is correct the context switch will continue.

It is possible for both the previous thread and the next thread to be PCPP threads. In this case both branches of PCPP code in the context switch routine will be run, first the
prev.pcpp branch, then the next.pcpp branch. In this case, the two PCPP tasks would each have separate key caches and master keys. As such encrypting prev’s key cache and master key would not affect the retrieval of next’s master key or decryption of next’s key cache.

Table 16: PCPP Key Cache Integrity Hash Contents

<table>
<thead>
<tr>
<th>Index</th>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x_0)</td>
<td>pcppcs_start</td>
<td>Physical address of the first instruction in PCPP context switch code</td>
</tr>
<tr>
<td>(x_1)</td>
<td>pcppcs_end</td>
<td>Physical address of the last instruction in PCPP context switch code</td>
</tr>
<tr>
<td>(x_2)</td>
<td>pcppchselhash</td>
<td>Hash of the PCPP context switch code</td>
</tr>
<tr>
<td>(x_3)</td>
<td>keycache_loc</td>
<td>Physical address of the PCPP key cache</td>
</tr>
<tr>
<td>(x_4)</td>
<td>tgid</td>
<td>Process ID of the protected application</td>
</tr>
<tr>
<td>(x_5)</td>
<td>instrptr</td>
<td>Physical address of random location in PCPP process’s instructions</td>
</tr>
<tr>
<td>(x_6)</td>
<td>instrhash</td>
<td>Hash of random 32 bytes from instruction memory</td>
</tr>
</tbody>
</table>

The integrity hash is a single value generated from the items listed in Table 16. Starting with the first value, \(x_0\), we xor each value with the previous hash and then hash that intermediate result. The resulting hash will change if any of the items from Table 16 changes. Equation 24 shows the process for creating the integrity hash mathematically.

\[
i_{n+1} = h(x_n \oplus i_n)
\]

First, we store the starting and ending physical addresses of the PCPP context switch code (the grey boxes from Figure 27). These are used to confirm, after decrypting the key cache, when the PCPP process regains control of the CPU, that the PCPP context switch code has not moved. Since, this code is in the kernel address space it cannot be paged out and moved, it should stay in the same location permanently. The next integrity item is a hash of the PCPP context switch code. This item is used to confirm that the PCPP context switch code has not changed since the PCPP process last relinquished the

118
CPU. The next item is physical address of the key cache itself. The key cache physical address is used to confirm that the key cache has not moved since the PCPP process last relinquished the CPU. The next item is the process ID of the protected process. The process ID is used to confirm that the process decrypting the key cache matches the process which previously encrypted it. The next item is the address of a random location in the PCPP process’s instruction code. This address is the start location for hash of 32 random bytes from the PCPP process’s instruction code. This hash is used as a spot check to confirm the protected process’s instructions were not altered. If the integrity hash value found in the key cache after decryption does not match the hash computed at the time of decryption the integrity check fails and the PCPP process is safely killed.

We developed two similar methods for storing $k_m$. Neither stores $k_m$ directly, both store $k_m$ indirectly via knowledge of the 1$^{st}$ and last members of a hash chain. $k_m$ is the $(n-1)^{th}$ member of a $n$-length hash chain such as that shown in equation 1, where $n$ is the number of successive hashes computed to find $h_n(h_0)$. We store $h_0$ and $h_n$ as indicators for how to retrieve $k_m$. Retrieving $k_m$ requires computing the hash chain from the starting value $h_0$ until $h_n$ is reached and then setting $k_m = h_{n-1}$.

$$h_n(h_0) = h(h(...h(h_0)))$$

8.3.1. SHA1/MD5 Based Hash Chains

Our first approach to implementing the hash chains described by equation 24 uses SHA1 [39] and MD5 [40] implementations for $h$. To create a new hash chain we first choose $h_0$ using a random number generator. Which random number generator to use depends upon the target system. It is important to choose a random number generator
which meets cryptographic requirements. Our implementation was in Linux so we used the /dev/random [38] device to fetch a random number. Once we have \( h_0 \) we choose two more random numbers, \( r_1 \) and \( r_2 \). We use \( r_1 \) to set the length of the hash chain. The length of the hash chain must be greater than 16 and less than 128 links. We take the least significant 7 bits of \( r_1 \) to force the hash chain length to be less than 128. We then set it to 16 if the trimmed \( r_1 \) value is less than 16. We use \( r_2 \) to choose between the SHA1 and MD5 hash algorithms. If \( r_2 \) is odd we use the SHA1 algorithm, if \( r_2 \) is even we use the MD5 algorithm.

We store \( h_0, h_n, \) and the hash type in the PCPP processes task structure. The items are stored in the clear when the PCPP process relinquishes control of the CPU. When the PCPP task regains control of the CPU \( h_0, h_n, \) and the hash type are used to recreate the hash chain and derive \( k_m \).

Because SHA1 and MD5 are common hash algorithms with many readily available implementations is it conceivable that a foe may attempt to steal \( h_0, h_n, \) and the hash type to derive \( k_m \). Avoiding such an attack is difficult. Our second hash chain methodology minimizes this risk.

8.3.2. HMAC Based Hash Chains

Protecting the master key is a difficult problem. In order to retrieve the key after storage we must also store some information on how to retrieve it. However, when the PCPP process relinquishes the CPU a foe may access this same information and couple that with knowledge of our hash chain implementation to recreate the hash chain. It is not possible to store a secret we can use which absolutely cannot be used by a foe. The
SHA1/MD5 method stores $h_0$, $h_n$, and the hash type, which are then used to retrieve the key. We have shown in section 8.2 that privileged users can easily access the memory contents of other processes. This makes storing $h_0$, $h_n$, and the hash type in the clear dangerous. However, to hide these or encrypt them we would need a new secret and that secret would need to be stored. There is always a last secret which must be stored where a foe may conceivably find it. The SHA1/MD5 hash chains from section 8.3.1 are too easily thwarted. We developed to our HMAC [41] based chains to make it more difficult, though, we cannot make it impossible.

To make retrieving the master key more difficult for a foe we desire several properties. First, we prefer to choose from many hash algorithms, more than two is with the SHA1/MD5 case above. Second, we prefer to construct the hash implementations in such a way that any foe finds it much easier to try to use our hash implementations rather than build his own implementations and simply use his to create a copy of our hash chains.

Since, HMAC is a keyed hash we can use different keys to generate separate implementations. In fact, we build many separate HMAC functions with predefined keys embedded in the HMAC implementation executable code. Since, the HMAC algorithm implementations are distinct and built with different keys; they meet the requirement of many different hash choices.

The keys used for each HMAC implementation are defined as constants in their respective implementations and copied into memory as the first step when the HMAC implementation is called. Because the keys are constants and too large to copy with a single *mov* assembly instruction they are stored in parts in the executable code and copied
into registers when the HMAC implementation is called. After the HMAC is computed we overwrite these registers with zero. Storing the HMAC keys in parts as constants in the executable code, rather than in the processes task structure or some other more accessible location serves to make it more difficult for a foe to steal the HMAC keys, though definitely not impossible.

Ideally, the HMAC implementations could be further obfuscated to make even more difficult for a foe to find the keys in the HMAC executable code. Currently, the keys are stored as 4, 32-bit constants and loaded into registers with 4 mov calls. Finding them would be somewhat difficult, but certainly not impossible. If the code were obfuscated to diffuse the keys further into the HMAC implementation it would become more difficult for a foe to find the HMAC keys. This would serve to encourage a foe to attempt to run our HMAC implementation rather than try to steal the HMAC keys and run offline.

It is important that the HMAC implementations only be used for one PCPP application run. Re-use of the same HMAC implementations with the same embedded keys would give foes extra time to learn the HMAC keys. The HMAC implementations can be derived on the PCPP local client and sent to the PCPP remote host during application launch or they can be created on the remote host loaded into memory when the PCPP application launches.

Equation 25 describes the ordinary HMAC. Ordinary HMAC takes a single key and xor’s that key with separate ipad and opad values.

\[ hmac(x) = h(k \oplus opad, h(m, k \oplus ipad)) \]
We found it simpler to dimply use two separate keys rather than create one key and then XOR twice for use in the HMAC. Our HMAC is shown in equation 26.

\[
hmac(x) = h(k_1, h(m, k_2))
\]

As with the SHA1/MD5 hash chains from section 8.3.1 we first choose \( h_0 \) using a random number generator. We also use the random number generator to choose the length of the hash chain and to choose the underlying hash algorithm used to compute the HMAC this chain is derived from. The length of the hash chain must be greater than 16 and less than 128 links. Our current implementation chooses between MD5 and SHA1, though other hash algorithms would be acceptable as long as they meet HMAC requirements.

We store \( h_0 \), \( h_n \), and the hash type in the PCPP processes task structure. The hash type now indicates both the underlying hash algorithm and which HMAC implementation to use. The \( h_0 \), \( h_n \), and the hash type values are still stored in the clear when the PCPP process relinquishes control of the CPU. As mentioned above, \( k_1 \) and \( k_2 \) are embedded in the HMAC executable code. When the PCPP task regains control of the CPU \( h_0 \), \( h_n \), and the hash type are used to call the appropriate HMAC implementation to recreate the hash chain and derive \( k_n \).

8.4. Transferring Keys from Local Client to Remote Host

In addition to safe storage of encryption keys on the remote host, the initial master key and key cache must be securely transmitted to the remote host from the local client.

The PCPP host must run a server which receives the PCPP application executable, any files sent with the executable, a hash chain, and an encrypted key cache from the
local client. Once the server receives the complete launch packet it launches the application on the remote host. This PCPP server must use SSL to encrypt the connection between the local client and the remote host. It must also be protected by all the PCPP building blocks like any other PCPP application to ensure the PCPP servers executable and data remain unaltered, unmonitored, and unrecorded.

Table 17: Initial Key Transfer

<table>
<thead>
<tr>
<th>Index</th>
<th>Who</th>
<th>Step</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Remote Host</td>
<td>Send HMAC keys over SSL secure channel</td>
</tr>
<tr>
<td>2</td>
<td>Local Client</td>
<td>Create key cache: stores initial encryption keys for PPELF and any files sent with the executable</td>
</tr>
<tr>
<td>3</td>
<td>Local Client</td>
<td>Create initial master key ( k_m ), and hash chain</td>
</tr>
<tr>
<td>4</td>
<td>Local Client</td>
<td>Encrypt key cache</td>
</tr>
<tr>
<td>5</td>
<td>Local Client</td>
<td>Send key cache and hash chain to remote host</td>
</tr>
<tr>
<td>6</td>
<td>Remote Host</td>
<td>Store key cache and hash chain to server PCPP structure</td>
</tr>
<tr>
<td>7</td>
<td>Remote Host</td>
<td>Launch application</td>
</tr>
<tr>
<td>8</td>
<td>Remote Host</td>
<td>New process inherits key cache and hash chain from parent</td>
</tr>
</tbody>
</table>

Table 18 shows the steps required to securely send the initial master key and hash chain from the local client to the remote host. The executable and any files to be sent with the executable must be encrypted before being sent to the remote host. This process may take place immediately before sending the executable and files to the remote host or it may happen sometime in advance. Either way the encryption keys used for this process must be sent to the remote host with the executable and any other files. These encryption keys are placed in a key cache as described above and the key cache is encrypted with the initial master key \( k_m \). Immediately, before this encryption step a hash chain is built which is used both to derive and store \( k_m \). The encrypted key cache and the hash chain are then sent to the remote host. All communication between the local client and remote host is encrypted to prevent third party eavesdropping. Once on the remote
host the PCPP server stores in the hash chain and key cache in its own PCPP structure. The PCPP server then uses exec to launch the PCPP application. When the new PCPP process is launched it inherits a copy of its parent’s task structure. If the parent is a PCPP process, which in this case it is, the parent’s task structure will contain a PCPP structure. The parent PCPP structure contains a pointer to second PCPP structure, which is intended for copying to a child during launch. When a PCPP process’s task structure is copied the copying code first searches for a child PCPP structure. If a child PCPP structure is available this structure is copied and used as the new process’s PCPP structure. If the child PCPP structure does not exist, the parent’s PCPP structure is used. For the case when the PCPP server is launching PCPP applications there will always be a child PCPP structure. When PCPP applications launch children they likely will not have PCPP structures and will therefore inherit the PCPP structure, complete with key cache and key chains from the parent.

The first item in Table 17 is specific to the HMAC case. Here, the PCPP server must send a pair of HMAC keys to the local client which can be used to create the hash chain.

8.5. Defense against Attacks

We foresee three types of attacks against the PCPP key protection system. First, an attacker may attempt to jump midway into the key protection code in an attempt to bypass certain checks. Second, an attacker may attempt to learn the master key, $k_m$, by implementing his own key retrieval code or copying the PCPP key retrieval code to a separate location and modifying it. Finally, an attacker may attempt to copy a PCPP
process’s Linux task structure, the PCPP process’s PCPP structure, and all or part of its encrypted memory contents to build separate task structure which is then placed on the ready to run queue for decryption by the context switch routine. In the remainder of this section we describe how the PCPP key protection system defends against these attacks.

In the first attack case a foe may attempt to jump to an intermediate point in the PCPP context switch code expecting to find a function return which will pop the foe’s return address from the stack and return control to the attacking program. For example an attacker may wish to jump to the master key retrieval code expecting the master key to retrieve and then expecting a conveniently placed return to send control back to the attacking program. We stop such an attack by compiling all of the PCPP context switch code, shown in Figure 27, as inline code. This means all function calls are replaced by the compiler with a unique copy of the function placed in line with the calling code where the function call was previously. This stops the PCPP context switch code from calling functions using the call assembly instruction and then using a return mnemonic to jump back to the calling location. By doing this we avoid attackers jumping directly to the first instruction of a PCPP function and then using our own return call to jump out of the routine. By in-lining all PCPP context switch code the first encountered return instruction occurs at the end of the entire context switch routine. As such anytime an attacker jumps to any instruction in the PCPP context switch routine it must run through to the end of the PCPP context switch routine. Jumping into the context switch code without properly setting pointers to a previous task and a next task will cause the operating system to lock up, ceasing all activity on the machine until a reboot is performed. If an attacker manages to properly create pointers to a previous and next task
the result would still almost definitely be a locked up system. In-lining stops foe’s from jumping into the context switch routine expecting to return out before a context switch.

The second attack scenario involves an attacker either copying the PCPP master key retrieval code to a separate location or using his own hash implementation, with a pilfered hash chain initialization value, end value, and hash type, to retrieve the master key. Copying the key retrieval code and executing it in another location is difficult but possible. Since the code is in-lined there will be no symbols in the instruction code pointing to the starting address of the key retrieval code. Also, there will be no obvious way to know where the key retrieval code ends. If a foe finds the start and end of the key retrieval code he will then need to copy it and then add code to protect the integrity of his own codes registers by identifying all registers used in the copied code and pushing these to the stack at the beginning of the copied code and popping them from the stack when returning to his own code. If all of this is done the copied code could be used.

Instead of copying the key retrieval code to separate location an attacker may choose to only copy the hash chain initialization value, the hash chain end value, and the hash type value from the victim process’s PCPP structure. We make this very difficult by using HMAC hash chains with HMAC keys embedded in the HMAC implementations as constants and by using an HMAC algorithm based upon two keys rather than the standard HMAC which is based upon just one. To successfully use his own HMAC code first the attacker would need to implement an HMAC which mimics the PCPP HMAC behavior. This would not be overly difficult since the code could be copied from the PCPP installation. Next, the foe would need to steal the HMAC keys. Since these are embedded in the HMAC executable code which is in-lined in the rest of the PCPP
context switch code these key may be difficult to find. Furthermore, if the HMAC implementations were obfuscated to hide or diffuse the keys in the HMAC implementation finding these keys would be all the more difficult.

Both copying the key retrieval code and executing it elsewhere and copying just the inputs to the key retrieval code and deriving the master key from a separate hash implementation are possible attacks. However, both are considerably difficult. If an attacker does manage to retrieve $k_m$ he may then proceed to decrypt the key cache. In this case he will learn the values of the protected keys in the key cache. To limit the damage in case this does happen we choose a new encryption key each time an isolated page is encrypted in the demand encryption/decryption algorithm. By doing this we limit the useful life of keys stolen by an attacker. We also change $k_m$ each time a PCPP context switch relinquishes the CPU.

The last attack vector involves an attacker copying all or part of a PCPP process’s context to a separate process’s memory space and then attempting to use the PCPP context switch code to decrypt the copied PCPP memory but still switch CPU control to a program chosen by the attacker. If an attacker attempts to build a Linux task structure which points to a set of PCPP data pages but replaces the instruction memory pages with a separate program the context switch code will decrypt the PCPP data pages as the attacker desires. However, it will also attempt to decrypt the replaced instructions with a key from the key cache. Two things can happen to the instructions at this point. If the instructions were not encrypted, or encrypted with a different key than the one found in the key cache, the instructions would be mangled by the decryption step and the integrity check from Figure 27 would fail causing the PCPP shutdown process to run. If the
attacker managed to encrypt his replacement program pages with the same key used for the actual PCPP application then the replacement program would decrypt correctly. However, the integrity check would still fail because the hash of the random 32 bytes of instruction code will not match the hash pushed into the key cache when the PCPP application most recently relinquished the CPU.

8.6. PCPP Key Protection Performance Overhead

There is a run time performance overhead associated with using the PCPP Key protection system. This overhead is limited to increasing the time required to complete a context switch. There is no performance degradation outside the context switch routine. We measure overhead for two configurations of the PCPP key protection system. First, we measured the overhead for a system which stores the master key with SHA1 and MD5 hash chains. Second, we measured the overhead for a system which stored the master key with HMAC based hash chains.

Figure 28 shows the context switch run time overhead for 4 PCPP key protection configurations. For all of the measurements we built a small routine which solely performed the key protection steps. The steps when context switching out include: create new master key, create integrity value, add integrity values to key cache, and encrypt the key cache. The steps when context switching in include: retrieve the master key, decrypt key cache, calculate integrity values, and validate the integrity values in the key cache. To accurately measure the time to perform these steps we built a loop which perform each step in sequence. We then timed this loop while it executed 100,000 times. The time for one loop iteration was the total measured time over 100,000. The numbers are
the time for 1 context switch. Since the key retrieval operations and key storage operations are close to mirrors of one another we divide the time for one iteration by 2 to get the overhead for 1 context switch.

The first pair of results in left most position of Figure 28 show the run time overhead when all PCPP context switch code is compiled with the inline attribute set and with the use of a hash of all of the PCPP context switch code in the key cache to confirm the context switch code did not change since key cache encryption. The overhead for this configuration is large for both the MD5/SHA1 version and the HMAC version. This overhead is dominated by the time to compute the hash of the in-lined PCPP context switch code. Analysis of this code shows that the in-lined version of the measure code listed above was over 41K bytes for the MD5/SHA1 version and the size of the HMAC

![Graph](image)

Figure 28: Context Switch Overhead from Various Key Protection Configurations
version was over 54K bytes. The second set of bars in Figure 28 show the impact of removing the hash of the PCPP context switch code. With this hash removed the overhead drops from 219uS to 22uS for the MD5/SHA1 case and from 340uS to 44uS for the HMAC case.

The next two data sets in Figure 28 show the impact of compiling the PCPP context switch code without the in-line attribute. For both cases the context switch overhead is similar to, but slightly higher than, the overhead for the case which used in-lined code but skipped the hash check of the context switch code. We expect these versions to be slightly slower than the in-lined version since the overhead of dealing with function calls is added into these results.

We conclude from this that in-lining the context switch code causes the code to grow significantly in size. This by itself would generally be acceptable since most hosts would have plenty of memory to accommodate a larger context switch routine. However, the larger in-lined code size does lead to slow hash times for validating the PCPP context switch code. We have explained the necessity of this validation step above and feel it is a required step to ensure the security of the PCPP key protection system.

We derived a set of equations to predict overall performance impact of using the PCPP key protection system. These equations are similar to those used in chapters 5, 6, and 7 to predict the run time overhead due to Secure Context Switch, Demand Encryption/Decryption, and Secure I/O protection respectively.

$$ t_{kprot} = eT + \eta_{out} + \eta_{in} $$

Equation 27 provides an approximation of the run time required to run an application with SCS enabled. Equation 27 starts by representing the amount of time a
program needs to run in terms of context switch time slices, or epochs, $e$, times the period of 1 epoch, $T$. As such $eT$ is the amount of run time the program would require if it were running without any interruption, i.e. in an environment free of context switches. We represent this basic run time as $eT$ because $e$ provides a minimum number of $T$ length time slices required to run the program. All other terms in equation 27 are overhead associated with context switching.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_{kprot}$</td>
<td>Run time with PCPP key protection</td>
</tr>
<tr>
<td>$t_{norm}$</td>
<td>run time without PCPP key protection</td>
</tr>
<tr>
<td>$e$</td>
<td>number of complete times slices needed to complete execution</td>
</tr>
<tr>
<td>$T$</td>
<td>period of one time slice</td>
</tr>
<tr>
<td>$\eta$</td>
<td>adjusted number of times slices after adding PCPP key protection overhead</td>
</tr>
<tr>
<td>$t_{cs}$</td>
<td>Context switch time without PCPP key protection</td>
</tr>
<tr>
<td>$t_{in}$</td>
<td>time for protected application to context switch in</td>
</tr>
<tr>
<td>$t_{out}$</td>
<td>time for protected application to context switch out</td>
</tr>
<tr>
<td>$t_{key}$</td>
<td>time to retrieve/store master key, encrypt/decrypt key cache, and perform integrity checks</td>
</tr>
</tbody>
</table>

The second term of equation 27, $\eta_{out}$, adds the context switch time for $\eta$ context switch out(s), i.e. relinquishing the CPU. The third term of equation 27, $\eta_{in}$, adds the context switch time for $\eta$ context switch in(s), i.e. regaining control of the CPU. In the second and third terms $\eta$ is the adjusted number of times slices used by the application. We adjust $\eta$ to account for the time needed for an ordinary context switch, extra time during context switch out to generate a new master key hash chain, calculate and store integrity information, and encrypt the key cache, and extra time during context switch in
to retrieve the master key, calculate and check integrity information, and decrypt the key cache.

\[ t_{out} = t_{in} = t_{cs} + t_{key} \]  

Equation 28 shows the relationship between \( t_{in} \) and \( t_{out} \). For our PCPP key protection implementation \( t_{in} \) and \( t_{out} \) are set to equal one another because the code for context switching in mirrors the code for context switching out.

\[ \eta = \left( \frac{2et_{cs} + 2et_{key}}{T} \right) + e \]  

Equation 29 estimates the adjusted number of time slices a PCPP key protected process will require. Equation 29 adds the minimum number of uninterrupted time slices, \( e \), to the number of extra time slices resulting from the extra time required for basic context switching and extra time required for encrypting and decrypting protected pages during the context switch. The first term of equation 29 calculates the extra times slices required for context switching by first summing the time for 2 ordinary context switches and 2 master key retrievals/stores and then dividing by the period of one time slice, \( T \). We use 2 context switch times and 2 master key retrieval/store times because context switches come in pairs, one at the beginning of the time slice and one at the end of the time slice.

\[ t_{norm} = eT + \eta t_{out} + \eta t_{in} \]  

Equation 30, which computes \( t_{norm} \), looks just like equation 27. The only difference is the definitions of \( \eta \), for the number of adjusted context switches, and the definitions for \( t_{in} \) and \( t_{out} \).

\[ t_{in} = t_{out} = t_{cs} \]
Equation 31 defines $t_{in}$ and $t_{out}$. Since there is no master key retrieval/storage, no integrity checks, and no key cache encryption/decryption required, the $t_{key}$ term is removed relative to equation 28.

$$\eta = \left( \frac{2t_{cs}}{T} \right) + e$$

Equation 32 shows the definition of $\eta$ for the non-PCPP key protection case. Since, without PCPP key protection there is no master key retrieval/storage, no integrity checks, and no key cache encryption/decryption this term is removed relative to equation 29.

$$kprot\_overhead = \frac{t_{kprot}}{t_{norm}}$$

Equation 33 defines the overhead for PCPP key protected applications when compared to the same application running without PCPP key protection.

PCPP key protected applications when compared to the same application running without PCPP key protection.
We used equations 27-33 to plot the predicted overhead associated with using PCPP key protection. Figure 29 shows the predicted run time overhead for both the MD5/SHA1 and HMAC implementations with in-lined code and all integrity checks in place. Both implementations have predicted run time overhead of less than 5%. While this overhead is significant, we believe it is acceptable for the improved key protection gained by using the PCPP key protection system.

8.7. Conclusion

We have shown that the Linux Key Retention Service (LKRS) does not meet PCPP encryption key protection requirements because it relies on the Linux access control system to protect its key rings. This allows users with root privileges to install a kernel module which can then be used by Alice to access Bob’s keys stored on a LKRS
key ring. Since, PCPP isolates processes from even root privileged users LKRS key protection is insufficient.

In this chapter we presented a modified Linux context switch routine which encrypts a PCPP key cache and with a master key, $k_m$, when a PCPP process relinquishes control of the CPU. After encryption of the key cache $k_m$ is securely stored. When the PCPP process regains control of the CPU, $k_m$ is retrieved and then the PCPP key cache is decrypted. Before allowing a PCPP process to resume ownership of the CPU integrity information stored in the key cache when the PCPP process relinquished ownership of the CPU is validated. $k_m$ is not stored as plaintext on the host platform, rather $k_m$ is the n-1$^\text{th}$ member of a hash chain in which the root of the hash chain and the n$^\text{th}$ element of the hash chain are stored. The hash algorithm is chosen on the local client randomly and sent to the host platform during application launch.

While our PCPP key protection system is still vulnerable to other privileged processes stealing $k_m$ when the PCPP process is not running this is made difficult by the use of the HMAC hash chains and the integrity information validation. Also, coupling the PCPP key protection mechanism with our Secure Context Switch technology limits the useful life $k_m$. We believe our PCPP key protection methodology is a considerable improvement over the Linux Key Retention Service.

Overall, we believe that our PCPP key protection system is a significant improvement over other software based key protection systems.
Chapter 9

COMPARATIVE ANALYSIS

9.1. Introduction

In chapter 1 we defined private computing as define as the ability to protect an application such that its executable code, data, and control flow remain unaltered, unmonitored, and unrecorded before, during, and after execution. We also offered a list of PCPP requirements formalized in Chapter 2.

In this chapter we offer a comparison of PCPP to two related technologies which were described in more detail in chapter 2. First, we will discuss out Trusted Computing may be applied to the PCPP problem. Second, we will discuss how SELinux may be applied to the PCPP problem. Finally, we compare the three technologies directly.

9.2. Trusted Computing Applied to PCPP Problem

The five Trusted Computing (TC) features described in the previous works chapter secure boot, hardware based encryption functions, a platform specific encryption key, curtained memory, and sealed storage can be combined to create formidable application privacy advances. However, the most prominent use of the TPM, Microsoft’s Bitlocker [15], does not provide application privacy as its primary goal, rather, Bitlocker aims to protect data stored on computer hard disk drives after computers, generally laptops, are lost or stolen. BitLocker uses the secure boot PCR to seal to enable full
volume encryption of the platform’s non-volatile memory (HDD). While such encryption protects against access after a platform comes into the hands of a malicious third party it does not protect data from other applications running on the platform in normal operating conditions. In this case, since the data on the hard drive is sealed only to the secure boot platform configuration, all applications running on the platform have access to the data stored on the disk. Bitlocker’s use of TPM features is adequate for its stated goal, but does not protect against the threats from our public computing threat model.

Table 19: Trusted Computing Applied to the Public Computing Threat Model

<table>
<thead>
<tr>
<th>Altered Code</th>
<th>Altered Data</th>
<th>Copied Code</th>
<th>Recorded Data</th>
<th>Monitored Code</th>
<th>Monitored Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>B D A B D A B D A B D</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

B = Before Execution, D = During Execution, A = After Execution

Table 19 shows our assessment of how the five primary Trusted Computing security features might be used to allay the threats in our public computing threat model.

TC protects data in all cases. Before and after execution data may be protected by sealing it to the application which has permission to access it. During execution application data may be protected by storing it in curtained memory. We assume software based access controls may have vulnerabilities or may be overridden by a privileged user. The type of curtained memory implementation is important to how well the application memory is protected from other concurrent processes. The Intel TXT curtained memory is preferred since the entire process can be stored in curtained memory isolated only to that process.
Executable code may be protected from alteration, copying, and monitoring during execution. However before and after execution, i.e. while the executable is in non-volatile memory (stored on HDD) it is vulnerable to alteration, copying, and monitoring. All contents of the HDD will be sealed to the secure boot platform configuration. This includes any applications. However, this feature is designed to stop unauthorized users from accessing data on a lost or stolen platform; it does not stop other processes running in normal conditions from accessing HDD contents, including the application in need of protection.

In summary, we show in table 20 that it is not possible to protect against all of the threats in our public computing threat model using the Trusted Computing features discussed in this section. Trusted Computing does not provide sufficient protection of executable before and after execution. Application data can be sealed to limit access. This process is left to the application allowing applications to choose what contents are written to a PCR before sealing and unsealing application data. Relegating the job of sealing data to the application offers the most flexibility, but, also requires that the application be cognizant of Trusted Computing to take advantage of this feature. This in turn limits the protections available to legacy software.

9.3. SELinux Applied to PCPP Problem

SELinux policy can be written to protect the execution code and data of privacy sensitive applications. For instance, policy can be written to give access to data stored on a platform to only a single type which is associated with a single application. This can be used to lock data to a single application. Since, applications are files, types can be
associated with these files to limit access to only one specific role, i.e. limit access to one user. Both of the above can be used to specifically limit access to data and executable code while it is stored on a platform, which meets the before and after requirements for our public computing model. We must add here that while access to specific files can be restricted the files themselves are not encrypted. This means that if due to some policy error the files can be accessed they are readable and alterable by an attacker.

Developing policy to protect executable code and data during execution is more difficult. SELinux policy can be developed to restrict access to temporary files. There are also permissions associated with a process which allow SELinux to restrict access to ptrace, an application used to trace a program’s stack and track its use of system calls. There are also process permissions which allow or deny configuring the stack and/or heap as executable. These exist to protect against buffer overflow attacks. Beyond this we find no permissions to restrict the use of the mmap and mprotect system calls which can be used to change the read, write, and execute permissions of user memory. As such, it appears that if a kernel module can be inserted which changes access permissions of another process’s memory, then that memory can be accessed. This may be difficult as insmod, the application which installs kernel modules, should be specifically controlled by the platform’s policy, but definitely possible since insmod would be available to privileged users, such as system administrators.

Table 20 shows SELinux’s ability to defeat the threats from our public computing threat model. As stated above policy can be written to protect application and data files while stored on the protected platform. This leads to checks for the before and after columns for data and executable code. Since we found no method for restricting memory
permission changes, beyond restriction of executable flags, we left the columns for executable code and data unchecked.

Table 20: SELinux Applied to the Public Computing Threat Model

<table>
<thead>
<tr>
<th>Altered Code</th>
<th>Altered Data</th>
<th>Copied Code</th>
<th>Recorded Data</th>
<th>Monitored Code</th>
<th>Monitored Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>D</td>
<td>A</td>
<td>B</td>
<td>D</td>
<td>A</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

B = Before Execution, D = During Execution, A = After Execution

SELinux policy is large and complex. There exist many ways to do the same thing. This makes validation of SELinux policy difficult [20]. Because of this difficulty in validating SELinux policy and because of the lack of encryption used to protect data and executable code as a secondary line of defense we colored the checks in grey rather than black to indicate a weakened ability to defend against the threats.

Early distributions of SELinux default policy suffered from incompatibilities between SELinux policy and many Linux applications. These incompatibilities effectively denied service from many applications and proved difficult for ordinary users to diagnose and fix. Consequently, overtime the Linux community has moved from a broadly strict default policy to default policies which specifically protect known critical objects, but allows all other objects to default back to permissions which match that of the Linux DAS access control system and avoid incompatibilities.
9.4. PPELF, SCS, SIO Impact on Public Computing Threats

Table 21 shows the contribution of PPELF, Secure Context Switch, and Secure I/O toward defeating the public computing threat model from chapter 1. PPELF protects executable code before and after execution. Executable programs do contain some data, and this data is also protected by PPELF, however, we do place checks in the data columns for PPELF, since the bulk of the data protection comes from Secure Context Switch (SCS), and Secure I/O (SIO). SCS protects both executable code and data during execution. SCS specifically protects only data stored in volatile memory and does not protect data associated with open files. SIO protects data files before, during and after execution. SIO protects file stored in non-volatile memory at all times, and protects the memory mapped portions of files during execution.

**Table 21: Application of Active Security Building Blocks to Public Computing Threats**

<table>
<thead>
<tr>
<th>Active Security Building Block</th>
<th>Altered Code</th>
<th>Altered Data</th>
<th>Copied Code</th>
<th>Recorded Data</th>
<th>Monitored Code</th>
<th>Monitored Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPELF</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Secure Context Switch</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Secure I/O</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Total</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

B = Before Execution, D = During Execution, A = After Execution

When combined PPELF, SCS, and SIO protect against all the threats from the public computing threat model. However, since all three use encryption as their primary tool in affecting this protection, the ultimate security of the system rests on PCPP’s ability to protect encryption keys. On one hand this good because we have boiled the
privacy problem down to a single item, the encryption key, which needs protection. On the other hand protecting encryption keys in an open environment is a difficult problem.

9.5. Comparison of PCPP, Trusted Computing, and SELinux

Table 22 shows a comparison of Trusted Computing, SELinux, and PCPP’s ability to defend against the threats from our public computing threat model.

### Table 22: Resistance to Public Computing Threats by Technology

<table>
<thead>
<tr>
<th>Technology</th>
<th>Altered Code</th>
<th>Altered Data</th>
<th>Copied Code</th>
<th>Recorded Data</th>
<th>Monitored Code</th>
<th>Monitored Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>B</td>
<td>D</td>
<td>A</td>
<td>B</td>
<td>D</td>
<td>A</td>
</tr>
<tr>
<td>Trusted Computing</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SELinux</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>PCPP</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

B = Before Execution, D = During Execution, A = After Execution

SELinux does not provide protection of volatile memory while applications execute. SELinux also uses complex policy declarations to declare its access controls. The complexity of this policy makes it difficult to validate a platform's policy and thereby difficult to rely on SELinux for application privacy.

Trusted Computing is not capable of defending against all of the threats in the public computing threat model. Trusted Computing does not bar alteration, copying, or monitoring of executable code while it is stored on the hard disk, i.e. before and after execution. Trusted Computing offers executable code some indirect protection from alteration while it is stored on the hard disk drive. First, encrypting the entire hard disk drive, as with the Bitlocker application, offers some protection to executable code, but does not stop authorized platform users from accessing and altering executables. Second,
since sealing data to an application often requires writing the hash of the application to a PCR, the same hash must be written to the PCR before unsealing the data. As such, if an executable is altered while stored on the platform’s hard disk the altered application will not be able to access any data sealed prior to the alteration. This of course does not stop alteration of the executable; but it does stop an altered application from accessing preexisting sealed data.

Table 23: PCPP Requirements Comparison

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Trusted Computing</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>SELinux</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>PCPP</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 23 compares Trusted Computing, SELinux, and PCPP based upon the PCPP requirements listed in Chapter 2. As mentioned above, both Trusted Computing and SELinux do not defend against all the threats in our public computing threat model. SELinux and PCPP are software only implementations, meaning that older platforms could be upgraded to support PCPP; alternatively Trusted Computing requires a hardware device to be mounted on the platforms motherboard, which would make upgrading older platforms difficult. All three technologies support an opt-in/opt-out model. Trusted Computing and PCPP offer methods for validating a public platform’s hardware and software configuration. Trusted Computing uses its Platform Configuration Registers to validate the configuration, while PCPP uses its remote assessment for this purpose. SELinux policy is very difficult to validate. This is the case because SELinux policy is
so flexible and complex. All three technologies provide application specific protection. Trusted Computing allows application data to be sealed to applications. SE Linux allows for policy specifically written for certain applications. PCPP allows protection of individual applications and their data. SE Linux and PCPP offer protection to legacy software, while, Trusted Computing does not, since Trusted Computing requires applications to be updated in order to support the sealed data feature. All three technologies offer encryption key protections. Trusted Computing allows encryption keys to be created and stored in tamperproof hardware. SE Linux policy can be written to protect encryption keys, in fact, SE Linux policy has been written to protect keys stored using the Linux Key Retention Service, the service currently used by PCPP.

9.6. Conclusion

Both Trusted Computing and SE Linux while not intended to meet PCPP requirements offer a fair amount of overlap with PCPP capabilities. However, neither Trusted Computing nor SE Linux provide complete coverage of the PCPP requirements.
Chapter 10

CONCLUSIONS AND FUTURE DIRECTIONS

Unlike the primarily trust based security models offered by existing distributed computing models such as Globus grids and SETI@Home, PCPP offers an active risk mitigating security approach. PCPP ensures that application control flows and data remain unaltered, unmonitored, and unrecorded before, during, and after execution and as such enables applications to run securely and privately on third party systems.

In this thesis we have defined the PCPP problem including deriving a public computing threat model and listing a set of PCPP requirements needed to achieve a secure and private distributed computing solution.

We detailed a PCPP implementation of 5 building blocks, host assessment, executable guard, secure context switch, secure I/O, and an encryption key protection.

The host assessment assures that a potential remote execution host meets three criteria. First, the remote host must provide the needed resources, correct hardware, operating system, etc. Second, the remote host must be have downloaded and installed all PCPP required operating system patches and those patches must be intact and unaltered. Finally, the security state of the prospective remote host must be such that it is deemed a non-threatening environment on which to execute PCPP protected applications.

The PCPP host assessment is broken into two parts, the go/no-go scan, the Bayesian classification, which combine to meet the three criteria mentioned above. The
go/no-go scan uses a quick external scan combined with data gathered by a PCPP scan process running on the remote host to confirm the system meets the minimum execution requirements for our PCPP application, confirm certain minimum security mechanisms are in place, and to confirm that the operating system and all PCPP patches are in place and unmodified. The Bayesian classification module uses a much larger collection of data gathered from a PCPP scan process running on the remote host and similar collected data from previous PCPP runs on other remotes hosts to classify the prospective host as a threat or non-threat.

As future work we would like to improve the host assessment in three ways. First, currently it possible to poison the data collected and used to classify the remote host as a threat or non-threat. We would like to look for data collection methods which have a higher probability of collecting truthful information about the remote host. This may include methods for ensuring data collection methods run unaltered on the remote host or methods for gathering more useful data from external scans, and or methods for gathering information about the remote host from third parties. Second, we wish to survey more classification methods. The Bayesian classifier may allow third parties to intentionally flood the training data with tuples designed to influence a particular classification outcome. For instance, a foe may allow a PCPP client to use his machine many times to fill the training database with tuples indicating that his machines configuration is a non-threat with the intention of attacking at a later time. We wish to find a classifier which is less susceptible to this type of manipulation. Third, Trusted Computing uses a hardware based chain of trust model to prove that a machine is running with a specific hardware and software configuration. We would like to explore the applicability of such a system.
to the PCPP model. We are interested in both the chain of trusts applicability when the chain of trust is built from a software only solution, thus adhering to current PCPP requirements, and when hardware is used to build the chain of trust, thus removing the PCPP requirement for a software only solution.

The executable guard is a secure method for storing binary executables on a remote host and for loading the stored executables into memory. In practice, an executable guard is an encrypted extension of an existing binary format for a given operating system and accompanying binary launch code which loads the executable into memory for execution while avoiding leaking private application code during this critical step. Rather, than decrypting the program to a separate plaintext copy prior to execution and then count on access control to thwart any foe, we load the encrypted program into the memory locations specified by the original program binary format and then decrypt. This allows Executable Guard programs never to be stored as a plaintext files in non-volatile memory (hard disk drive, flash drives, etc.) and only exist as plaintext in volatile memory. This property makes securing executable privacy much easier, since after execution volatile memory regions used by the program can simply be over written with zeroes or random data leaving no trace of the plaintext. The encrypted Executable Guard version of the program stored in non-volatile memory may be safely deleted after execution with a standard delete command. Although, the deleted program may be recovered by an attacker, the recovered program will be in its encrypted form and is therefore useless without its associated encryption key.

For this thesis we implemented an Executable Guard for the Linux operating system which we called PPELF. As future work, we would like to explore implementing
executable guards for other operating systems including UNIX and the Windows operating system.

Secure Context Switch protects the memory pages which store a process’s executable code, data segment, stack, and heap during execution. SCS provides this protection by isolating these memory pages using applied encryption techniques. Specifically, SCS encrypts the protected memory pages immediately before a process relinquishes the CPU during context switch. For our first SCS implementation all encrypted memory pages are decrypted immediately before the PCPP protected process regains control of the CPU. With the demand encryption/decryption method protected pages are decrypted only when accessed, which significantly reduces the number of pages which must be decrypted and then re-encrypted for a given time slice providing a significant speed up as a result.

Our current SCS implementation requires a single processor architecture. On a single processor platform attackers cannot read, write, or alter the PCPP application’s memory when the PCPP application itself is running. When the PCPP application relinquishes the CPU an attacker may attempt to read memory locations owned by the protected application, however, these pages will appear encrypted to the attacker.

As future work we would like to explore the use of SCS as is in multi-processor environments in order to quantify the increased risk due to the potential that a foe may run in parallel on a different CPU and attempt to access PCPP protected memory contents while these memory contents are decrypted. We hope the relative risk increase will be small when demand decryption/encryption is used since only a subset of the PCPP applications pages will be decrypted at any one time.
We would also like to research use of hardware architectures designed to support application isolation. Such hardware architectures may include architectures which move the SCS encryption/decryption into hardware or may include methods for reliably isolating applications such that encryption and decryption are no longer required.

Secure I/O (SIO) protects files owned by the PCPP protected application while the files are stored on the remote host. SIO requires all files to be stored in encrypted form. SIO then uses an operating system architecture update to decrypt file read data before it is returned to the application and encrypt file write data before it is sent to the file system. Permanent encryption of the files stored in non-volatile memory and protection of file accesses combine to protect files before, during and after execution. SIO uses an exclusion list to designate files which do not use SIO protection.

As future work we would like to investigate more powerful schemes for creating file exclusion lists. Currently, files must be designated by name, we would like to explore methods for using regular expressions and or conventions to determine which files to protect and which not to protect.

Further, we would like to explore the use of SIO as is in multi-processor environments in order to quantify the increased risk due to the potential that a foe may run in parallel on a different CPU and attempt to intercept SIO protected file accesses.

Also, any future hardware architectures designed to support application isolation would apply to the SIO building block.

We showed in this thesis that the Linux Key Retention Service (LKRS) does not meet the PCPP encryption key protection requirements. We built and demonstrated an exploit for this thesis which attacks LKRS stored keys across user accounts.
Consequently, we developed the PCPP key protection methodology which isolates PCPP encryption keys from all other processes running on the same platform, regardless of the privilege assigned to that process. PCPP key protection uses a key cache to hold all of the keys used by the other PCPP building blocks. This key cache is tagged with integrity information during the operating system’s context switch routine, at the point immediately before a PCPP isolated process relinquishes the CPU to another process, and then encrypted with a master key, $k_m$. The master key is then securely stored until the PCPP isolated process regains the CPU, at which point, again during the operating systems context switch routine $k_m$ is retrieved to decrypt the isolated process’s key cache.

As future work we would like to explore hardware based key creation and storage methodologies. Such methodologies would include use of the Trusted Computing TPM to create and store keys as well as new architectures designed specifically to meet PCPP needs.

Together the 5 PCPP building blocks protect the application against all of the threats in our public computing threat model and meet all other PCPP requirements defined in chapter 1 of this thesis.

As future work we would like to apply to PCPP to various problem areas. First, we would like to integrate PCPP into the Globus Toolkit. Once integrated into the Globus toolkit, we would like to install PCPP on a live Globus server where we could study PCPP performance impact for a broad range of applications. Second, we would like to apply PCPP to certain specific problems in which application isolation would be
beneficial. For instance we would like to build a PCPP protected web server and a PCPP protected ad hoc wireless networking server.

Finally, In addition to defining PCPP and its implementation we offered a detailed discussion (chapter 2) of two application security related works SELinux, and Trusted Computing. We show how both of these technologies might be applied to the PCPP problem and offer a technology comparison (chapter 9) which shows PCPP’s overall superiority to both technologies as applied to the PCPP problem.
Appendix A

IMPLEMENTATION DETAILS

In this appendix we offer details on our PCPP implementation. We implemented the PCPP building blocks by patching and augmenting the Linux operating system kernel. We used version 2.6.20.2 as the starting point for our implementation.

We do not show all of the code we wrote to implement PCPP just code snippets we feel offer value to the reader.

A.1. Linux exec System Call

```c
for (fmt = formats ; fmt ; fmt = fmt->next) {
    int (*fn)(struct linux_binprm *, struct pt_regs *) = fmt->load_binary;
    if (!fn)
        continue;
    if (!try_module_get(fmt->module))
        continue;
    read_unlock(&binfmt_lock);
    retval = fn(bprm, regs);
    if (retval >= 0) {
        put_binfmt(fmt);
        allow_write_access(bprm->file);
        if (bprm->file)
            fput(bprm->file);
        bprm->file = NULL;
        current->did_exec = 1;
        proc_exec_connector(current);
        return retval;
    }
    read_lock(&binfmt_lock);
    retval = -ENOENT;
    put_binfmt(fmt);
}
```
if (retval != -ENOEXEC || bprm->mm == NULL)
break;
if (!bprm->file) {
    read_unlock(&binfmt_lock);
    return retval;
}

The above for loop is taken from the Linux OS implementation of the exec system call, the system call used by most programs to launch other programs. This loop loops through all known binary formats attempting to launch the program sent to the exec system call. If a binary format successfully launches the program it will return a non-negative value the execution was successful and system call returns, otherwise the next available binary format handler is called. Finally, if no binary format handler successfully launches the program an error is returned.

The PPELF binary format handler is added to the available formats list with a call to a Linux kernel API register_binfmt.

A.2. PPELF Binary Format Handler: binfmt_pcpp

Below are code snippets from the PCPP binary format handler which is part of the PPELF implementation of the executable guard. We built the PPELF binary format handler by taking the ELF binary format handler and adding code snippets to customize if for PPELF. The two formats are very similar to one another with PPELF being mostly just an encrypted version of its ELF cousin.

The PCPP binary format handler is a Linux kernel module. It is compiled separately and loaded into the kernel after boot using the insmod command.
### Linux Binfmt Structures

Linux contains multiple binary format handlers to load executable code compiled in various binary formats. The two most popular formats are *a.out* and *ELF*, with ELF being the more modern and more common of the two. Each binary format handler has an associate `linux_binfmt` structure. The above code snippet is from the PPELF `linux_binfmt` structure. The `linux_binfmt` structure provides call back functions to handle certain events. The most important of these to PCPP is the `load_binary` call back function which is the function called to load the binary into memory and actually launch the program.

```c
if (memcmp(loc->elf_ex.e_ident, PCPPMAG, SPCPPMAG) != 0)
goto out;
```

The above snippet is taken from the PPELF `load_binary` call back function. One of the first checks in any binary format handler to check the program's magic number. Each binary format has a different magic number which is the first four bytes of the binary executable. The first byte of the magic number must be a non printable character. If the first byte is a printable character the exec system call will assume the file is script and attempt to execute the file with what ever text it finds on the first line of the file. If the first byte is non-printable the exec call attempts to execute the program with available binary format handlers via the loop discussed in Appendix A.1.

The PPELF magic number is the number 0x7f followed by the characters 'PCP'.

```c
//get key
encrkey = request_key(&key_type_user, KEYDESC, NULL);
if (IS_ERR(encrkey)) {
```

155
printk(KERN_INFO "Error: Can not find key.\n");
retval = PTR_ERR(encrkey);
goto out_free_ph;
}
rcu_read_lock();
keypayload = rcu_dereference(encrkey->payload.data);
rcu_read_unlock();
key = keypayload->data;

The above code segment reads a key from the Linux Key Retention Service (LKRS) session key ring used by the PPELF. Currently, PCPP processes are still run in a separate PCPP session which is used to store the encryption key which was used to encrypt the binary executable.

//get master_iv
master_iv = kmalloc(sizeof(u8)*BLOCK_LEN,GFP_KERNEL);
ivoffset = vfs_llseek(bprm->file, 0, SEEK_END);
ivoffset = vfs_llseek(bprm->file, ivoffset - BLOCK_LEN, SEEK_SET);
retval = kernel_read(bprm->file, ivoffset, (char *)master_iv, BLOCK_LEN);
if (retval != BLOCK_LEN) {
    if (retval >= 0)
        retval = -EIO;
    goto out_free_ph;
}

Next we read the master_iv which was used to encrypt the PPELF executable from the end of the executable file. The master_iv is stored at the tail of PPELF executable as plain text.

Once the encryption key and master_iv are available they are both written into the PCPP key cache where they are safely stored for the life of the program. Safe storage of the executable encryption key before the program launches is an open problem.
iv = kmalloc(sizeof(u8)*BLOCK_LEN,GFP_KERNEL);
memcpy(iv, master_iv, BLOCK_LEN);
addiv(iv, loc->elf_ex.e_poff / BLOCK_LEN);
desc = setup_desc(key, GFP_USER);
retval = cnt_encrypt((void *)elf_phdata, size, *desc, iv, loc->elf_ex.e_poff % BLOCK_LEN);

if(retval != size) {
    //encryption error
    goto out_free_ph;
} //end if

The above snippet sets up a local initialization vector (iv). Because we use
counter mode AES encryption the actual iv used to decrypt any block of code is the
master_iv plus the byte offset of that block into the executable file. With the iv set we
then call cnt_encrypt to decrypt the ELF program header which gives pointers to the rest
of the logical blocks in the program. Note the name cnt_encrypt. Counter mode
encryption as with AES is symmetric, so we can use the same routine to encrypt and
decrypt. We encrypt the program header here because it is about to be accessed by the
existing binary format handler code.

Finally, if there is an encryption error we goto out_free_ph to safely shutdown,
destroy the encryption key and overwrite the area in memory containing the decrypted
program header..

memcpy(iv, master_iv, BLOCK_LEN);
addiv(iv, elf_ppnt->p_offset / BLOCK_LEN);
desc = setup_desc(key, GFP_USER);
retval = cnt_encrypt((void *)elf_interpreter, elf_ppnt->p_filesz, *desc, iv,
elf_ppnt->p_offset % BLOCK_LEN);

Next, if the executable program header indicates the presence of a ELF interpreter
we must decrypt the code segment containing the interpreter location. The binary format
handler will call this executable to complete the program load process, specifically to load any shared libraries. We also leave the majority of decryption of the executable to the interpreter since this leaves the executable code encrypted as long as possible.

If an interpreter is not specified the binary format handler must load the code segments as specified by the ELF program header. In this case the PPELF binary format handler will also need to decrypt these code segments. We added code to handle this case which is not shown in this appendix. It looks very similar to both of the above calls to the `cnt_encrypt` routine.

With demand encryption/decryption both the encryption steps above and the decryption in the GLIBC interpreter become unnecessary. In the demand encryption/decryption case the decryption will occur in the page fault exception handler immediately before the code segments are actually accessed.

```c
    current->pcpp = setup_pcpp_struct(master_iv, key);
```

The above line sets up the PCPP key cache with executable's encryption key and `master_iv`. `current` is a Linux kernel structure used as a container for Linux processes context. It contain information such as pointers to all open files associated with the process, all memory used by the process, the process ID and much more information. We added a structure to `current` to hold necessary PCPP information. That structure is shown below in Appendix A.4.
A.3. GLIBC Interpreter – rtld.c

We mentioned in Appendix A.2 that the majority of executable decryption is left to be completed in the ELF interpreter. Most programs use the GLIBC ELF interpreter for this step. We updated the GLIBC interpreter to handle PPELF executable decryption.

We changed just 3 files in the GLIBC source tree Makeconfig, elf/Makefile, and elf/rtld.c. The first two are make files which we updated to include the AES API in the compilation of the rtld.c file. rtld.c contains the actual programmatic changes required to handle PPELF.

```c
ehdr = (void *)((ElfW(Addr)) phdr - sizeof(ElfW(Ehdr)));
if(memcmp(ehdr, PCPPMAG, SPCPPMAG) == 0) {
    pcpp = 1;
    binkey = malloc(KEY_LEN);
    retval = getkey(KEYDESC, binkey);
    if (retval) {
        freekey: for(i=0;i<KEY_LEN;i+=4) strncpy(binkey, JUNKSTR, 4);
        free(binkey);
        assert(0);
    } //end if

    master_iv = malloc(BLOCK_LEN);
    retval = getkey(IVDESC, master_iv);
    if (retval) {
        free(master_iv);
        goto freekey;
    } //end if

    aes_encrypt_key((unsigned char*)binkey, KEY_LEN, ctx);
    biniv = malloc(BLOCK_LEN);
    memcpy(biniv, master_iv, BLOCK_LEN);
    offset = sizeof(ElfW(Ehdr));
    size = sizeof(ElfW(Phdr))*phnum;
    __mprotect ((void *)ehdr, size+sizeof(ElfW(Ehdr)), PROT_READ | PROT_WRITE);
    addiv(biniv, offset/BLOCK_LEN);
    retval = cnt_encrypt((void *)phdr, size, binkey, KEY_LEN, biniv, offset%BLOCK_LEN, ctx);
```
We do this by checking the magic number in the ELF header. Finding the ELF header location is a bit of a trick. The binary format handler is called with a pointer to the ELF header location, but the GLIBC interpreter is not. For ordinary ELF files it is not needed. We know the ELF header is always loaded into memory and is stored immediately before the ELF program header. To find the ELF header we simply subtract the size of the ELF header from the ELF program header pointer. Once we have the ELF header location we can determine if the executable is a PPELF executable or an ordinary ELF executable by checking the ELF magic number. If the executable is PPELF we then fetch the encryption key and master_iv and then decrypt the ELF program header. Before we can decrypt the ELF program header we first change the memory permissions of the program header segment to allow writes. This must be done because the binary format handler set permissions for all executable code to read and execute only.

```c
if (pcpp) {
    encrstart = (void *)mapstart;
    size = ph->p_filesz + (ph->p_vaddr & (ph->p_align - 1));
    offset = ph->p_offset - (ph->p_vaddr & (ph->p_align - 1));

    if (offset <= (sizeof(ElfW(Ehdr)) + phnum*sizeof(ElfW(Phdr))) &&
        (ElfW(Addr))ehdr == mapstart) {
        //skip Phdr & Ehdr as they are already plaintext
        size -= ((ElfW(Addr))phdr + phnum*sizeof(ElfW(Phdr)) - mapstart);
        encrstart = (void *)((ElfW(Addr))phdr + phnum*sizeof(ElfW(Phdr)));
        offset = sizeof(ElfW(Ehdr)) + phnum*sizeof(ElfW(Phdr));
    } else if (offset <= sizeof(ElfW(Ehdr))) {
        //skip Ehdr as it is already plaintext
        size = sizeof(ElfW(Ehdr));
        encrstart = (void *)((ElfW(Addr))mapstart + sizeof(ElfW(Ehdr)));
        offset = sizeof(ElfW(Ehdr));
    } //end if

    prot = 0;
    if (ph->p_flags & PF_R)
        prot |= PROT_READ;
    if (ph->p_flags & PF_W)
```
prot |= PROT_WRITE;
if (ph->p_flags & PF_X)
    prot |= PROT_EXEC;
memcpy(biniv, master_iv, BLOCK_LEN);
adiv(biniv, offset/BLOCK_LEN);

if (!(ph->p_flags & PF_W))
    __mprotect ((void *)encrstart, size, prot | PROT_WRITE);
retval = cnt_encrypt(encrstart, size, binkey, KEY_LEN, biniv, offset%BLOCK_LEN, ctx);
if (!(ph->p_flags & PF_W))
    __mprotect ((void *)mapstart, ph->p_filesz + (ph->p_vaddr & (ph->p_align - 1)), prot);
assert(retval == (unsigned long int)size);
} //end if pcpp

Once the interpreter has the ELF program header it loops through the pointers to the various segments of the program and handles them based upon their type. We decrypt all PT_LOAD segments. These are all the segments of the executable file which are intended to be loaded into memory before execution.

The above routine is embedded in a case statement which handles the various ELF segment types. This code is in the “PT_LOAD” case.

There is an if statement near the top this code to make sure we do not decrypt the ELF header or the ELF program header. The ELF header is never encrypted and the ELF program header was previously decrypted so decrypting it again would corrupt it and lead to a segmentation fault.

Before we decrypt the segment we first look up the access permission settings for this segment in the program header. We then update the protections to include write permission to enable decryption. After decrypting the segment we again update the permissions to the value defined by the ELF program header.
As usual before encryption we must set the local initialization vector (iv) to the master_iv plus the byte offset, measured from the start of the executable file, of this segment.

A.4. Addition to the Linux task_struct: pcpp_struct

#define PCPPKEYCACHELEN 32
struct pcpp_struct {
    unsigned int keycache[PCPPKEYCACHELEN];
    void *tfm;
    struct sio_excl_list *exclude;
};

As mentioned in Appendix A.2 we added a PCPP structure to the Linux task_struct structure which is a container used to hold information about each process's context. The pcpp structure contains 3 items. The first is an array of integers which make up the key cache. Currently, the key cache is set to hold 32 integers. The second item is a pointer tfm which is pointer to the AES implementation used by the secure context switch for this process. The final item is the SIO exclusion list which is used to exclude files from encryption. Any file on this list is not encrypted during the context switch. Any file not on the list is encrypted. The SIO exclusion list is simple linked list. The list ends when the next pointer item from sio_excl_list structure is NULL.

A.5. Secure Context Switch Code

    static inline struct task_struct *
    context_switch(struct rq *rq, struct task_struct *prev,
        struct task_struct *next) {
        struct mm_struct *mm = next->mm;
        struct mm_struct *oldmm = prev->active_mm;
        if (!mm) {
            next->active_mm = oldmm;
            return prev;
        }
        return rq

162
switch_to(prev, next, prev);
return prev;
}

The above code snippet is the unaltered context switch routine from the Linux kernel version 2.6.20.2. The context switch procedure is very simple. First the active memory map is swapped with some special case handling for programs which do not have a memory map. Next, the registers and stack are swapped.

if (unlikely((prev->pcpp != NULL) || (next->pcpp != NULL)))
  pcppcs(prev, next);

We added the above two lines just after the variable declaration in the context_switch routine shown above. These lines check if either the previous or next process is a PCPP process. If it is the pcpp context switch code, pcppcs, is called. The pcppcs subroutine is uses the static inline attributes to force it compile as in line code (i.e. the code for the subroutine is placed in situ and no function call is used).

The pcppcs subroutine contains 2 primary blocks, one to encrypt outgoing contexts and one to decrypt incoming contexts.

if (likely(prev->pcpp->pcpp_job == PCPPVAL)) {
  if (likely(prev->mm != NULL)) {
    //get key
    key = (char *)prev->pcpp->kc->keys;
    set_key((struct pcpp_aes_ctx *)prev->pcpp->actx, key);

    for (vma = prev->mm->mmap; vma; vma = vma->vm_next) {
      atomic_inc(&oldmm->mm_count);
      enter_lazy_tlb(oldmm, next);
    } else
      switch_mm(oldmm, mm, next);
  if (!prev->mm) {
    prev->active_mm = NULL;
    WARN_ON(rq->prev_mm);
    rq->prev_mm = oldmm;
  }
  /* Here we just switch the register state and the stack. */
  switch_to(prev, next, prev);
  return prev;
}
if ((vma->vm_file == NULL) && (vma->vm_start < prev->mm->start_stack)) {
    page_encrypt(vma->vm_start & PAGE_MASK, PAGE_ALIGN(vma->vm_end), prev);
}
} //end for
page_encrypt(prev->mm->start_code & PAGE_MASK, PAGE_ALIGN(prev->mm->end_code), prev);
page_encrypt(prev->mm->start_data & PAGE_MASK, PAGE_ALIGN(prev->mm->end_data), prev);
newkm(prev->pcpp);
newiv(prev->pcpp);
addintegrity(prev, head, tail);
//location check
asm volatile("leaq locchk1,%0
 : " =r (rip));
asm volatile("locchk1: nop");
pcpp_loc_chk(prev);

// encrypt key cache
set_key((struct pcpp_aes_ctx *)prev->pcpp->actx, prev->pcpp->km);
generic_encrypt((void *)prev->pcpp->kc, sizeof(struct pcpp_keycache), prev->pcpp->keycacheiv, prev);
// erase local copy of master key
pcpp_memset(prev->pcpp->km, 0, KEY_LEN); // clear the master key location
actx = (struct pcpp_aes_ctx *)prev->pcpp->actx;
pcpp_memset((void *)actx->buf, 0, 120); // clear the key stored in the AES implementation
} //end if
} //end if

The above code handles secure context switch of outgoing processes. First, the code checks if the outgoing process, denoted as prev, is a PCPP protected process. If prev is a PCPP process we proceed. All of the memory regions which need encryption are pointed to by the prev->mm pointer. The code first confirms this is non-NULL. For PCPP jobs the prev->mm pointer can be NULL if there is a context switch very early in the PP ELF procedure. Generally, however, the prev->mm pointer is not NULL. Next we set a pointer to the key and make room for the IV which we will setup in the page_encrypt routine. The procedure set_key sets up the key for our AES routine. We must do this every context switch and overwrite that data with a zero key to stop other
users from stealing the key from the guts of our AES routine. Next we encrypt all pages which are not backed by a file (the heap). Then we encrypt the executable code segments and the data segments from the executable file. Next, we choose and store a new master key. We then add integrity information to the key cache before encrypting the key cache. Finally, we set all memory locations used to store the encryption keys to zero.

```c
if (likely(next->pcpp->pcpp_job == PCPPVAL)) {
    if (likely(next->mm != NULL)) {
        // Fetch master key
        getkm(next);
        // Decrypt key cache
        set_key((struct pcpp_aes_ctx *)next->pcpp->actx, next->pcpp->km);
        generic_encrypt((void *)next->pcpp->kc, sizeof(struct pcpp_keycache), next->pcpp->keycacheiv, next);
        // Location check
        asm volatile("leaq locchk2,%0:1n:0t" : =r : (rip));
        asm volatile("locchk2: nop");
        pcpp_loc_chk(next);
        chkintegrity(next, head, tail);
        // Get key
        key = (char *)next->pcpp->kc->keys;
        set_key((struct pcpp_aes_ctx *)next->pcpp->actx, key);
        for (vma = next->mm->mmap; vma; vma = vma->vm_next) {
            if ((vma->vm_file == NULL) && (vma->vm_start < next->mm->start_stack)) {
                page_encrypt(vma->vm_start & PAGE_MASK, PAGE_ALIGN(vma->vm_end), next);
            }
        }
        page_encrypt(next->mm->start_code & PAGE_MASK, PAGE_ALIGN(next->mm->end_code), next);
        page_encrypt(next->mm->start_data & PAGE_MASK, PAGE_ALIGN(next->mm->end_data), next);
        // Set key
        set_key((struct pcpp_aes_ctx *)next->pcpp->actx, next->pcpp->zerokey);
    } // End if
} // End if
```

The above snippet handles decryption of incoming PCPP jobs in the kernel's context switch routine. It is nearly identical to the encryption routine for handling outgoing PCPP jobs. This is possible because AES encryption and decryption are
symmetric and therefore we can use the same routines for encryption as decryption. The main difference is here we must fetch the master key to decrypt the key cache. We then validate the integrity information in the key cache before proceeding.

```c
#define page_encrypt(start, end, iv, task) 
   memcpy(iv, (void *)((task->pcpp->keycache+BLOCK_LEN), BLOCK_LEN); 
   for (addr = start; addr < end; addr += PAGE_SIZE ) { 
      startaddr = get_paddr(task->mm, addr); 
      if (startaddr != NULL) { 
         retval = cnt_encrypt(startaddr, PAGE_SIZE, (void *)iv, 0, (struct pcpp_aes_ctx *)task->pcpp->tfm); 
      } 
   }
```

The above snippet is a macro defining the page_encrypt function used in the context switch routines to encrypt and decrypt pages. We start by setting up the initialization vector (IV). We copy it from the key cache. We copy the IV to a separate location because the cnt_encrypt routine increments the IV and we don't want our master_iv updated. That is an inefficient piece of code, we could have used a directive to tell the compiler not change that pointer. Next, we get the physical address of the page to be encrypted/decrypted. We start from a virtual address, but the TLB and page fault handler are inaccessible while in the context switch routine because interrupts have been turned off. Therefore, we translate the virtual address to a physical address manually in the get_paddr routine. Next we encrypt/decrypt the page and move on.

A.6. PCPP AES Implementation

The Linux Kernel contains an AES implementation for use by various kernel subparts. However, we were not able to use this AES implementation for secure context switch because it will not run when interrupts are disabled. Because of this we copied the
kernel AES module and made small changes to allow it to run in the context switch routine. We do not include the AES code here because of its size.

We made two major changes. We moved our AES implementation into a header file to be included with the context switch routine so that it could be called directly rather than through the kernel crypto umbrella. By calling the AES code directly we bypass the checks which confirm that interrupts are not off. The interrupt check is there because the kernel version may context switch to load a crypto module into memory if it is not present when called. We set-up our AES code as part of the PPELF process to ensure that it is in memory and therefore ensure that there is no need to context switch.

The second change was to localize the code or in other words remove all dependence on outside code and modules. Comparing the kernel crypto/aes.c to our include/linux/pcpp.h one would notice a change from use of a kernel crypt “tfm” structure to the use of the pcpp_aes_ctx structure. This structure is part of the pcpp additions to the Linux task_struct structure which is the container used by the operating system to hold each processes context. PCPP processes each contain there own instance of a ctx structure which holds look-up tables used by the AES routine during encryption/decryption.

A.7. Linux Key Retention Service Attack

We developed a set of code which we used to prove that Linux Key Retention Service key rings used by one user can be stolen from another user's account. This code was built in a client server arrangement. The server is a Linux kernel module that waits to be called and then once called reviews the task_struct belonging to every process in the
system looking for non-NULL LKRS thread key rings. If a process contain a non-NULL thread keyring the server read and printed the contents of that key ring. Since the server portion is a kernel module it must be inserted into the kernel with the `insmod` command, an action which requires root permission.

Once the keyattack server is in place and listening a separate program which can run with user permissions signals the kernel module to attack.

```c
int searchforkeys(void) {
    struct task_struct *p;
    int retval = 0;
    struct key *threadkeyring, *foundkey;
    struct user_key_payload *upayload;
    struct keyring_list *keylist;
    int kix;

    rcu_read_lock();
    for_each_process(p) {
        if (p->thread_keyring != NULL) {
            threadkeyring = p->thread_keyring;
            printk("TMDEBUG: %d has populated thread keyring, desc = %s\n", p->tgid, threadkeyring->description);
            keylist = rcu_dereference(threadkeyring->payload.subscriptions);
            for (kix = 0; kix < keylist->nkeys; kix++) {
                foundkey = keylist->keys[kix];
                printk("Key description = %s. payload =\n", foundkey->description);
                upayload = rcu_dereference(foundkey->payload.data);
                printbytes(upayload->data, 16);
            }
            retval = p->tgid;
        }
    } //end for_each_process
    rcu_read_unlock();
    return retval;
}
```

The above snippet is a loop which reviews each active process on the system for `task_struct` containing a non-NULL thread_keyring. If the a populated thread keyring is
found the code goes on to loop through each key attached to the key ring printing its
description and payload contents. We simply printed the payload contents, however, we
easily could have placed the contents in a file setup with permissions for an adversary to
read.

A.8. Converting ELF to PPELF

We created a program called elf2pcpp which converts ELF programs to PPELF
programs. The program reads in an ELF program encrypts all contents below the ELF
header and then appends a IV to the tail of the PPELF version of the program.

```
biniv = malloc(BLOCK_LEN);
fillrand(biniv, BLOCK_LEN);
ivlen = BLOCK_LEN;

char *master_iv;
master_iv = malloc(BLOCK_LEN);
memcpy(master_iv, biniv, BLOCK_LEN);
```

First we create the initialization vector (iv) for the encryption algorithm. The iv is
simply and random number. We then store the iv for later writing to the tail of the
program in a variable called `master_iv`.

Not shown here for space reasons we open the ELF file with the ELFEXE file
handle and a PPELF version with the PCPPEXE file handle.

```
//load elf (just read in the file)
size = sizeof(Elf64_Ehdr);
elf_header = malloc(size);
//read in the ELF header
if(read(ELFEXE,(void *)elf_header, 0, 1, size))
  goto out_bad_io;
if (strcmp(elf_header->e_ident, ELFMAG) == 0) goto out_bad_ftype;
```
memcpy(elf_header->e_ident, PCPPMAG, SPCPPMAG);
if (rawrite(PCPPEXE, (void *) elf_header, 0, 1, size))
goto out_bad_io;

Above we read in the ELF header whose contents and size we know from an ELF include file which contains structure definitions for all the interesting information in an ELF file. After reading the ELF header we confirm that its magic number matches the ELF magic number, then we change the magic number to the PPELF magic number called PCPPMAG above. The PCPP magic number is 0x7f, followed by the string 'PCP'. Once the magic number is changed we write out the PPELF header to the PPELF file.

offset = sizeof(Elf64_Ehdr);
addiv(biniv, offset/BLOCK_LEN);
size = (unsigned int) fread(segment, 1, SEGMENT_SZ, ELFEXE);
while (size > 0) {
    retval = cnt_encrypt((void *) segment, size, binkey, KEY_LEN, biniv, offset % BLOCK_LEN);
    if(retval != size) {
        printf("Error: Encryption length incorrect intended size = %d, actual size = %d.\n", size, retval);
        goto out_bad_io;
    } //end if

if (rawrite(PCPPEXE, (void *) segment, offset, 1, size))
goto out_bad_io;
offset += size;
size = (unsigned int) fread(segment, 1, SEGMENT_SZ, ELFEXE);
memcpy(biniv, master_iv, BLOCK_LEN);
addiv(biniv, offset/BLOCK_LEN);
} //end while

Above we setup the iv by adding the byte offset in the file to the master_iv. Then we enter a loop which encrypts the remaining contents of the file in segments where SEGMENT_SZ is 1024 bytes. We encrypt in segments because some ELF files can be
rather large encrypting smaller chunks keeps elf2pcpp from consuming very large quantities of memory. Each segment that is encrypted is then written to the PPELF file.

```c
// write the master_iv to the end of the file
retval = (unsigned int) fwrite((void *) master_iv, 1, BLOCK_LEN, PCPPEXE);
if (retval != BLOCK_LEN) {
    printf("Error: Unable to write iv to file.\n");
goto out_bad_io;
} //end if
```

With the body of the ELF program encrypted we then write the `master_iv` to the tail of the file.

A.9. PCPP Key Protection Details

The code segments in Appendix A.5 contained calls to subroutines used for PCPP key protection. The PCPP key protection code contains code use to ensure that the pcppcs routine (shown in Appendix A.5) is not moved in physical memory. The PCPP key protection code also contains subroutines which choose a new master key, retrieve a master key, add integrity information to the key cache, and to verify integrity information from the key cache.

The first instructions of the pcppcs routine use assembly code to load the starting and ending address of the pcppcs routine.

```c
// this should be the first instruction of pcppcs
// it is used to learn the physical address of the pcppcs routine
asm volatile("leaq pcppcshead,%0\n\nt" : "=r" (head));
asm volatile("leaq pcppcstail,%0\n\nt" : "=r" (tail));
asm volatile("pcppcshead: nop");
```

The above inline assembly instructions load the address of pcppcshead and pcppcstail into the head and tail variables respectively. The "pcppcshead: nop" is placed
immediately before the body of the pcppcs code. A second "pcppcshead: nop" inline assembly instruction is added after the last instruction in the pcppcs routine. The load effective address instruction (leaq) is used to load these addresses into their respective variable. We chose this method for learning the start end locations of the pcppcs routine because the code is compile as in line code and therefore a function pointer was not an option.

```c
static inline void newkm(struct pcpp_struct *pcpp) {
    int chainlength, i;

    get_random_bytes(pcpp->chaininit, 20);
    chainlength = *(int *)pcpp->chaininit & 0x7f;
    if (unlikely(chainlength < 16))
        chainlength = 16;
    pcpp->hashtype = *(int *)pcpp->chaininit & 0x1;

#ifdef PCPP_USE_HMAC
    pcpp->hmac_index = (*(int *)pcpp->chaininit & 0x2) >> 1;
#endif

    PCPP_MAC(pcpp, pcpp->chaininit, pcpp->chainend);
    for(i=1;i<(chainlength-1);i++) {
        PCPP_MAC(pcpp, pcpp->chainend, pcpp->chainend);
    }
    __inline_memcpy(task->pcpp->km, pcpp->chainend, KEY_LEN);
    PCPP_MAC(pcpp, pcpp->chainend, pcpp->chainend);
}
```

The above routine, `newkm`, builds a new hash chain which is used to store the master master key. First, `newkm` the Linux kernel’s built in random number generator to choose a new initialization value for the hash chain. Next we choose a new hash chain length. Finally, we loop through creating hashes of the previous value in the chain. We stop at the $n-I^{th}$ iteration to copy the $n-I^{th}$ element into the master key location.
static inline void getkm(struct task_struct *task) {
    char link[20], nextlink[20];
    int i;

    i = 0;
    __inline_memcpy(link, task->pcpp->chaininit, 20);
    PCPP_MAC(task->pcpp, link, nextlink);
    while((pcpp_memcmp(task->pcpp->chainend,nextlink,20) != 0) && i<128) {
        __inline_memcpy(link, nextlink, 20);
        PCPP_MAC(task->pcpp, link, nextlink);
        i++;
    }
    __inline_memcpy(task->pcpp->km, link, KEY_LEN);
}  //end getkm

The above routine, getkm, takes the chaininit value from the PCPP structure and builds a hash chain in a loop by continually hashing the previous link in the hash chain. When the chainend is reached the routine copies the n-1th value into the master key variable.
BIBLIOGRAPHY


