Determining the Integrity of Applications and Operating Systems using Remote and Local Attesters

by

Raghunathan Srinivasan

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved March 2011 by the
Graduate Supervisory Committee:

Partha Dasgupta, Chair
Charles Colbourn
Aviral Shrivastava
Dijiang Huang
Prashant Dewan

ARIZONA STATE UNIVERSITY

May 2011

©2011 Raghunathan Srinivasan
All Rights Reserved

ABSTRACT

This research describes software based remote attestation schemes for obtaining the integrity of an executing user application and the Operating System (OS) text section of an untrusted client platform. A trusted external entity issues a challenge to the client platform. The challenge is executable code which the client must execute, and the code generates results which are sent to the external entity. These results provide the external entity an assurance as to whether the client application and the OS are in pristine condition.

This work also presents a technique where it can be verified that the application which was attested, did not get replaced by a different application after completion of the attestation. The implementation of these three techniques was achieved entirely in software and is backward compatible with legacy machines on the Intel x86 architecture.

This research also presents two approaches to incorporating software based "root of trust" using Virtual Machine Monitors (VMMs). The first approach determines the integrity of an executing Guest OS from the Host OS using Linux Kernel-based Virtual Machine (KVM) and qemu emulation software. The second approach implements a small VMM called MIvmm that can be utilized as a trusted codebase to build security applications such as those implemented in this research. MIvmm was conceptualized and implemented without using any existing codebase; its minimal size allows it to be trustworthy. Both the VMM approaches leverage processor support for virtualization in the Intel x86 architecture.

DEDICATION

To my late parents, I dedicate my second, just as I was their second

ACKNOWLEDGEMENTS

This has been an incredible journey. I started out in 2005 on the back of multiple setbacks in life. I am finishing 2011 with a Doctorate and much more. There are many people to whom I owe my sincerest gratitude for having reached this far. It has to start with my late parents for giving me a good direction in life. The next person I need to thank is my sister, Sathya, without whose nagging in 2005, I would not have attempted graduate studies. Next, I owe thanks to my other relatives who supported me through the toughest times a person can go through.

I would like to thank my advisor Dr. Partha Dasgupta without whose guidance none of this would be possible. I would like to thank Dr. Charles Colbourn, Dr. Dijiang Huang, Dr. Aviral Shrivastava, and Dr. Prashant Dewan for serving on my committee and giving me useful insight into my research. I would like to thank my innumerable lab mates who helped me out at various times. I would like to thank my close friends Tushar Gohad, Dr. Satyajayant Misra, and Dr. Pavel Ghosh for their thoughts on various issues during my PhD. I would like to thank Dr. Guoliang Xue and Dr. Matthew Pittinsky for guiding me on many issues. Other notable mentions are to Dr. Amiya Bhattacharya, Ranal Fernando, and Harie Srinivasa for helping me to complete this work. I would also like to thank all my room mates and friends during the last 6 years for having shared many truly delightful and funny memories.

My last ‘thanks’ is a special one reserved for my long term girlfriend, now fiancée and soon to be wife, Jessica. I have experienced remarkable upsurge in my fortunes ever since I met her. It started with getting an internship, resulted in many publications and has culminated in me successfully defending my Doctorate.

TABLE OF CONTENTS

Page

TABLE OF CONTENTS iv

LIST OF TABLES vii

LIST OF FIGURES viii

CHAPTER 1

1 INTRODUCTION 1

2 RELATED WORK 8

2.1 Problems with building secure systems 8

2.2 Hardware based integrity measurement schemes 8

2.3 Existing commodiy VMMs 9

2.4 Virtualization based integrity measurement 11

2.5 Software based integrity measurement schemes 12

2.6 Attacks against software based schemes and counter arguments 13

2.7 Program analysis and code obfuscation 14

2.8 Hardware extensions for virtualization on the Intel platform 15

2.9 Hypervisors Utilizing Extensions for Virtualization 16

3 THREAT MODEL AND ASSUMPTIONS 19

3.1 Threat model and assumptions for user application attestation 19

3.2 Threat model and assumptions for kernel attestation 20

3.3 Threat model and assumptions for guest OS attestation using KVM . . . 21

3.4 Threat model and assumptions for minimal VMM creation 21

4 DESIGN OF INTEGRITY MEASUREMENT CODE 22

4.1 Changing execution flow and locations of variables on stack 22

4.2 Inserting dummy instructions 23

4.3 Changing instructions during execution 23

4.4 Implementation 24

Chapter Page

Changing execution flow and locations of variables on the stack 24

Obfuscating instructions executed 25

5 REMOTE ATTESTATION OF USER APPLICATION $\mathscr{P}$ 26

5.1 Implementation 27

Injection of Code on $\mathscr{P}$ 27

Communication with Trent 28

Determining Machine Identifiers 29

Determining MD5 and Arithmetic Checksum 30

Determining Process Identifiers 31

5.2 Results 32

6 VERIFIED CODE EXECUTION 35

6.1 Stack allocation in Intel architecture 35

6.2 Executing $\mathscr{F}_0$ after $\mathscr{F}_1$ without executing a RET 37

6.3 Implementation 37

7 KERNEL ATTESTATION 40

7.1 Implementation 42

Identifying locations to measure in kernel 42

Communication with Trent$'$ 43

Fixing call instructions 43

Disabling interrupts 45

7.2 Results 45

8 ATTESTATION OF A GUEST OS FROM A HOST OS 47

8.1 Implementation 51

Starting a clone 51

Starting a TCP server inside clone 52

Reading memory contents of the guest OS 53

Results 54

Chapter Page

9 BUILDING A SECURE MINIMAL TRUSTED CODE BLOCK VMM . . . 56

9.1 Overview of dynamic launch model 58

9.2 Design of System 59

9.3 Implementation 62

Initial Processor Checks 62

Allocating memory for the VMM components 63

Loading State values into VMCS 64

Launching MIvmm 67

Continued execution of MIvmm 67

Lines of Code 68

10 CONCLUSION 69

REFERENCES 71

BIOGRAPHICAL SKETCH 76

LIST OF TABLES

Table Page

5.1 Average code generation time at server end 33

5.2 Time to compute measurements 33

7.1 Execution times for various components 45

8.1 Execution times for components of kvm-qemu setup 54

LIST OF FIGURES

Figure Page

1.1 Overview of Remote Attestation 5

1.2 Overview of verified code execution 6

4.1 Snippet from the checksum code 23

5.1 Detailed steps in Remote Attestation process 27

5.2 *send* routine through *socketcall* in ASM 29

5.3 Contents of /proc/net/tcp file 32

6.1 Sample C routine and its disassembly 36

6.2 Change of flow of execution 37

6.3 Tail portion of $\mathscr{F}_1$ 38

6.4 Fixing Jump target 38

7.1 user application initiates attestation request 40

7.2 user application sends attestation code to kernel space 41

7.3 kernel returns integrity measurements to user land 41

7.4 Verification of kernel integrity by trusted server 41

7.5 Fixing locations of call instruction 44

8.1 Overview of kvm-qemu interface 49

8.2 Overview of qemu clone operation 50

9.1 Overview of dynamic launch 59

9.2 System Design 60

9.3 Structure of allocated stack area for host 64

9.4 Lines of code of each component in the VMM 68

Chapter 1

INTRODUCTION

A consumer computing platform can be compromised by malicious code in many different ways. Preventing compromises requires safe coding, developing secure Operating Systems (OS), and developing secure kernel modules. Fault densities in OS kernels can range from 2 - 75 per 1000 Lines of Code (LOC) [Ostrand and Weyuker, 2002]. OS kernels are often supplemented by many device drivers or kernel modules which have higher error rates [Chou et al., 2001]. Buffer overflow is a common vulnerability that exists in many pieces of application software. This may allow malware to compromise systems [Iyer et al., 2010].

All copies of an application are identical; this gives an attacker (Mallory) the opportunity to analyze the presence and locations of vulnerabilities in the application, and develop means to exploit these flaws. Operating Systems offer little or no fault isolation; this can lead to a malware rapidly obtaining control of a computing platform [Wang and Dasgupta, 2008]. It has been mathematically proven that perfect detection of unknown viruses is equivalent to solving the Halting program [Cohen, 1993]. Smart malware can render detection schemes ineffective; this is due to the fact that traditional detection mechanisms operate off application binaries which can be disabled or patched to escape detection [Srinivasan and Dasgupta, 2007]. Consequentially a user (Alice) has to request integrity measurement of the platform from an external agent, or an entity that operates beyond the bounds of the operating system.

Remote attestation is a set of protocols that use a trusted service to probe the memory of a client computer to determine whether at least one application has been tampered with or not. Primarily used for DRM, these techniques can be extended to determine whether the integrity of the entire system has been compromised. Remote attestation has been implemented using hardware devices, virtual machine monitors

(VMM), and software based techniques. The Trusted Platform Module (TPM) chip has been used extensively to build hardware based solutions for remote attestation. In most cases, some integrity measurement values are stored in the Platform Configuration Registers (PCR) of the TPM. Anytime an integrity measurement is to be taken on the consumer platform, a private key stored in the PCR is used to sign the integrity values read from the system software [Stumpf et al., 2006], [Sailer et al., 2004], [Goldman et al., 2006]. A parallel hardware integrity measurement may use a secure co-processor that can be placed on the PCI slot of the client platform [Wang and Dasgupta, 2008], [Petroni Jr et al., 2004]. The co-processor contains an independent software stack which can read all memory locations on the client platform to determine whether any compromise has taken place. Virtualization schemes involve a special software layer known as the *hypervisor* or a VMM taking integrity measurements over a *guest* operating system [Garfinkel et al., 2003], [Sahita et al., 2007].

Software based solutions for Remote Attestation vary in their implementation technique. Most methods involve taking a mathematical or a cryptographic checksum over a section of the program in question ($\mathscr{P}$) , and reporting the results of verification to a trusted external server (Trent) [Seshadri et al., 2005], [Kennell and Jamieson, 2003]. TEAS [Garay and Huelsbergen, 2006] proves mathematically that it is difficult for an attacker to forge integrity results obtained on a client platform, provided the integrity measurement code changes for every attestation instance, however, an implementation framework is not provided in the work.

To provide a trusted computing environment to an end user, this dissertation provides four frameworks. The first two schemes obtain the integrity of an OS and a user application without the use of hardware support and without any virtualization support. The protocol involves the untrusted client platform communicating with a trusted external server Trent. Trent issues challenges which are executable code to the

OS or the user application depending on the entity being verified. For obtaining the integrity measurement of the OS Text section, the attestation service provider Trent$'$ provides executable code ($\mathscr{C}_{kernel}$) to the client OS (OS_{Alice}). OS_{Alice} receives the code into a kernel module and executes the code. It is assumed that OS_{Alice} has means such as Digital Signatures to verify that $\mathscr{C}_{kernel}$ did originate from Trent$'$. The challenge measures the integrity of the OS text section, System Call Table and Interrupt Descriptor Table (IDT).

It may be argued that once the OS is attested, it can be used to attest the application rendering the second scheme redundant. However, many applications execute on a client platform, and each gets updated frequently. If the OS performs integrity measurement on each binary, for security requirements the definitions should reside somewhere in the kernel. These definitions will have to be updated in the protected area frequently as each software gets updated. This can be considered a major overhead in the system. Instead of this, the simpler solution is to have an external agent such as the application vendor, or a network administrator provide integrity measurement for the applications as the definitions need to be updated only at one location. After the attestation is completed for the OS areas, the attestation proceeds with the second scheme which measures the integrity of a particular client application. The OS provides system call interface which is extensively used by the user application scheme, this way the OS serves as a root of trust for the application attestation scheme. For the sake of explanation, this work explains the user application prior to presenting the OS attestation scheme. Hence during the discussion of the user application it is assumed that the OS is pristine, although in practice the kernel attestation should precede the user application attestation.

For attesting the user application ($\mathscr{P}$), the trusted authority (Trent) issues a challenge to $\mathscr{P}$. The response provided by $\mathscr{P}$ allows Trent to determine whether its integrity is compromised. The challenge should have inherent characteristics that

prevent Mallory from forging any section of the results generated. A software protocol allows Mallory to perform various attacks. If the challenge is not different for every attestation instance, Mallory can construct a replay of a response from a previous instance of the attestation. If the challenge is not complex, Mallory can compute the response without executing the requested challenge and send the results to Trent. In addition, Mallory may bounce the challenge to another machine which contains a clean copy of the program to obtain results of the challenge. Mallory may also execute the challenge in a sandbox to determine its results.

To mitigate these situations, Trent generates a new instance of attestation code $\mathscr{C}$, which is sent to Alice for execution. $\mathscr{C}$ is binary code which is injected by the application $\mathscr{P}$ on itself. $\mathscr{C}$ does not require the system library support as it executes any required system call by executing software interrupts. This prevents any user level malware from tampering with the results of integrity measurements. The kernel should not be compromised for this process to work.

Since Alice injects $\mathscr{C}$, it has to be verified that $\mathscr{C}$ was indeed generated by Trent. To determine this Alice can setup an SSL connection to Trent and receive $\mathscr{C}$ during the SSL connection. Trent can be authenticated using a certificate based scheme while the rest of the communication can be encrypted using a session key.

Injection of code on a client machine (M_{Alice}) to obtain integrity measurements is an important aspect of the solution provided in this work. This reduces the window of opportunity that Mallory may have to analyze the measurement operations being performed on M_{Alice}. The operations performed by $\mathscr{C}$ in each attestation instance are changed to prevent Mallory from performing a *replay* attack. There are many operations performed during attestation that make determining the response difficult for Mallory without executing $\mathscr{C}$. In addition, $\mathscr{C}$ measures some machine and process identifiers which are determined through the system interrupt interface to make forging of results difficult. $\mathscr{C}$ has inherent programming constraints which ensure that

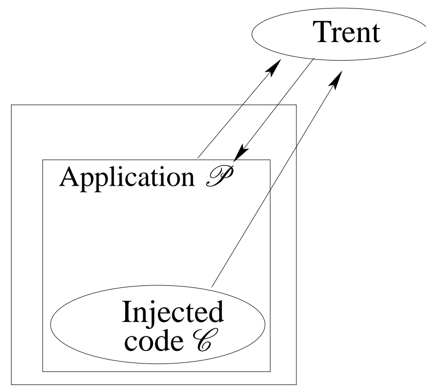

Figure 1.1: Overview of Remote Attestation

if $\mathscr{C}$ executes, it sends the results back to Trent.

Fig. 1.1 provides an overview of remote attestation. Trent is a trusted entity who has knowledge of the structure of a clean copy of the process ($\mathscr{P}$) to be verified. Trent has to be a trusted server, as Alice executes code received from Trent. Trent provides executable code ($\mathscr{C}$) to Alice which is injected on $\mathscr{P}$. $\mathscr{C}$ takes overlapping MD5 hashes and overlapping arithmetic checksums on sub-regions of $\mathscr{P}$ and returns the results to Trent. This prototype determines the integrity of a binary executing at (M_{Alice}). This protocol is robust against user mode viruses that can modify system libraries, but not against rootkits. The remote attestation implemented as part of this work is more robust and works under harder constraints more difficult than those implemented in previous works *Pioneer* [Seshadri et al., 2005], *Genuinity*, [Kennell and Jamieson, 2003].

It is possible that once *Remote Attestation* is completed, Mallory may replace the attested binary ($\mathscr{P}$) with a corrupted version ($\mathscr{P}'$). Alice would have no knowledge of such a change as long as $\mathscr{P}'$ performs all the functionalities of $\mathscr{P}$. To prevent Mallory from achieving such attacks, a framework for *verified code execution* is also presented in this work. This involves server making some changes to the code section of the client program $\mathscr{P}$ after the remote attestation is performed. Trent uses $\mathscr{C}$ to change a function call in $\mathscr{P}$ to call a new function $\mathscr{F}_1$ instead of calling $\mathscr{F}_0$.

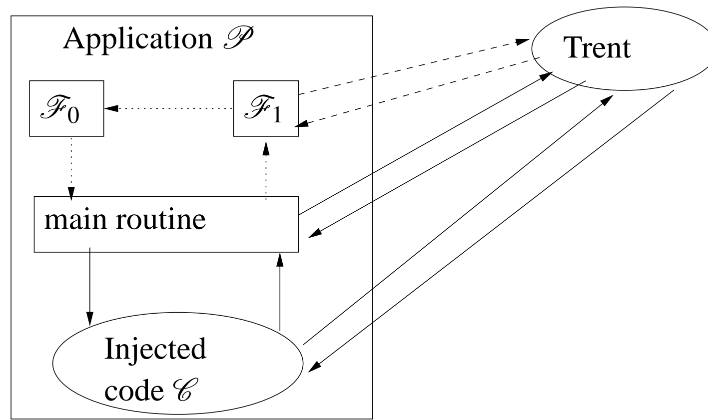

Figure 1.2: Overview of verified code execution

When the changed section of $\mathscr{P}$ executes it communicates back to Trent. This communication tells Trent that the attested program indeed completed execution. All changes made by Trent to the attested program are non-persistent and remain in-core to prevent Mallory from analyzing the changes made to $\mathscr{P}$. In addition this keeps the binary image of $\mathscr{P}$ unmodified. Fig. 1.2 shows the overview of the verified code execution process.

To remove the dependency on remote agents, this dissertation presents a scheme where an external agent residing on the same physical machine as the client OS. Any agent residing within the bounds of a corrupted OS is susceptible to getting subverted. Hence a local attester has to reside outside the client OS. To incorporate this threat model, this dissertation implements a virtualization based scheme where the integrity of a guest OS is measured by a Host OS using the Linux KVM interface. Depending on the threat model, the Host OS can communicate the results to the user sitting on the guest OS using a separate channel or pass the results back to the guest. The communication of results is not implemented as part of this framework.

Virtual machine monitors (VMM) are not completely secure; numerous vulnerabilities are known to exist in Xen 3, VMware Workstation 6, and VMware ESX Server 3 [Secunia, a], [Secunia, b], [Secunia, c], and [Wojtczuk, 2008a]. To address

this aspect, this dissertation also presents a framework for implementing a small VMM (MIvmm) on which security audits can be performed easily. A small code base is more manageable, and can be used as a trusted code base on which various applications can be built. The VMM presented in this work was implemented in under 4000 Lines of Code (LOC). This is due to the fact that the VMM supports only the minimum necessary features to support virtualization for one guest operating system.

The rest of this dissertation is organized as follows: Chapter 2 presents related work for all the implemented modules in this dissertation. Chapter 3 provides the threat model for each of the implementations. Chapter 4 presents the guidelines for generating attestation code; chapter 5 presents the remote attestation framework for obtaining the integrity of a user application. Chapter 6 presents the verified code execution component; chapter 7 presents the kernel attestation scheme. Chapter 8 presents the Linux KVM based attestation scheme; chapter 9 presents the small VMM, finally chapter 10 concludes this work.

Chapter 2

RELATED WORK

2.1 Problems with building secure systems

Many early security works featured on building secure kernels. A secure kernel implements basic security mechanisms to control the system resources, prevent intrusions, and provide verification of components [Ames Jr. et al., 1983], [Wika and Knight, 1994], [McCauley and Brongowski, 1979]. If the kernel is completely secure and trusted then the security of the rest of the software can be built around it. However, reliable and secure operating systems did not exist in the past [Tanenbaum et al., 2006], and with recent operating systems running into millions of lines of code with many rich features, it is unlikely that a secure operating system will exist in the future. Fault density is found to be in the range of 2 to 75 bugs in every 1000 lines of operating systems code [Ostrand and Weyuker, 2002]. Device drivers are known to have higher error rates than operating systems [Chou et al., 2001]. Due to these issues, relying on a commodity operating system kernel to provide protection and integrity measurements is not feasible.

2.2 Hardware based integrity measurement schemes

Some hardware based schemes that determine the integrity of a client platform operate off the TPM chip provided by the Trusted Computing Group [Stumpf et al., 2006], [Sailer et al., 2004], [Goldman et al., 2006]. These schemes may involve the kernel or an application executing on the client obtaining memory reads, and providing it to the TPM. The TPM signs the values with its private key and may forward it to an external agent for verification. The TPM may also be capable of providing secure bootstrap, but subsequent deployment of malware may go undetected based on the implementation of the protocol. TPM based solutions have the stigma of Digital Rights Management (DRM), may be difficult to reprogram and are not ideally suited

for mass deployment.

Integrity Measurement Architecture (IMA) [Sailer, 2008] is a software based integrity measurement scheme that utilizes the underlying TPM on the platform to measure the integrity of applications that are loaded on the client machine. IMA maintains a list of integrity values of all possible applications in the system. When an executable, library, or kernel module is loaded, IMA performs an integrity check prior to executing it. IMA measures values while the system is being loaded, however, it does not provide means to determine whether any program that is in execution is tampered in memory after it was loaded from the secondary storage.

Co-processor schemes that are installed on the PCI slot of the PC have been used to measure the integrity of the kernel as mentioned in section 2.1. One scheme [Wang and Dasgupta, 2008] computes the integrity of the kernel at installation time and stores this value for future comparisons. The core of the system lies in a co-processor (SecCore) that performs integrity measurement of a kernel module during system boot. The kernel interrupt service routine (SecISR) performs integrity checks on a kernel checker and a user application checker. The kernel checker proceeds with attesting the entire kernel .TEXT section and modules. The system determines that during installation for the machine used for building the prototype, the .TEXT section began at virtual address 0xC0100000 which corresponded to the physical address 0x00100000, and begin measurements at this address. The Copilot [Petroni Jr et al., 2004] is a hardware coprocessor that constantly monitors the host kernel integrity. It cannot handle dynamic kernel modules and user-level applications and it does not have a mechanism for a kernel patch.

2.3 Existing commodiy VMMs

A virtual machine monitor (VMM) adds a layer of software to emulate computer hardware such that one hardware platform can be partitioned into multiple logical

platforms. The VAX [Karger et al., 1991] security kernel was a VMM based security solution for the VAX processor. It creates isolated virtual processors each capable of running an operating system. The VAX architecture did not have provisions to support a VMM and hence certain changes were required to implement the security kernel. The security kernel is a layered architecture and isolates one layer from another completely. The kernel applies discretionary and mandatory access controls to each VM. VMM based detection schemes are used to detect the presence of malware and rootkits. The VAX processor is not manufactured anymore and this VMM is outdated as a result of it.

Xen is a virtual machine monitor that allows concurrent execution of several guest operating systems on one hardware platform. The first guest operating system is a trusted OS known as its domain 0. The domain 0 guest boots automatically with the hypervisor (VMM) and receives special privileges and direct access to all hardware on the system. Domain 0 can be used to manage other untrusted guest machines (domain U). The Xen hypervisor is a large system that comprises of nearly 150,000 lines of code [Weblink, p], which is coupled with a trusted domain 0 OS. This leads to a bulky solution, increasing the possibility of vulnerabilities in the implemented VMM.

Kernel-based Virtual Machine (KVM) is a Linux kernel based virtualization technology [Weblink, h]. Each virtual machine in KVM is a Linux process and it interacts with a driver known as the KVM driver. All hardware access for the virtual machine is handled by the KVM driver using a character device /dev/kvm, and through a modified qemu process. All code handling and exception handling is delegated to one kernel module. In the Linux 2.6.33 kernel the KVM device driver for Intel VT–x can be measured to be around 4000 lines of C code [Weblink, j]. However KVM also has other components such as an emulator, interrupt controller, memory management, and page table management which increase the overall size of KVM. It should be noted that these features are necessary once real applications are built on a

VMM. However, for obtaining a code block without vulnerabilities, these features can be eliminated and added later when required by each application. This also gives an application the freedom to choose how to implement the above features. As KVM is coupled with the Linux kernel, the security features provided by KVM are affected by the security features a standard Linux kernel provides.

VMware ESX server 3.x was known to have had 13 documented security vulnerabilities in 2009 [Secunia, a] - these attacks were documented as privilege escalation, security bypass, and exposure of sensitive information. VMware workstation 6 was reported to have had 5 new security advisories documented in 2009 [Secunia, b] - 1 of them was a privilege escalation based attack. Xen was documented to have had 5 new security advisories in 2008 [Secunia, c] which consisted of security bypass and DoS based attacks. These numbers show that commercial VMM solutions that have been in use extensively (may have also passed numerous audits) still have vulnerabilities present. The presence of newly discovered vulnerabilities over time makes it imperative to create a secure root of trust VMM.

2.4 Virtualization based integrity measurement

Terra uses a trusted virtual machine monitor (TVMM) and partitions the hardware platform into multiple virtual machines that are isolated from one another [Garfinkel et al., 2003]. Hardware dependent isolation and virtualization are used by Terra to isolate the TVMM from the other VMs. Using Terra, a scheme can be implemented where each class of application may be run on a different virtual machine. Terra is installed in one of the VMs (TVMM) and is not exposed to external applications like mail, gaming, and so on. The TVMM takes measurements on the VMs prior to loading them. Most traditional VMM based schemes are bulky and need significant resources on the platform to appear transparent to the end user, this holds true for Terra where the authors advocate multiple virtual machines.

VIS [Sahita et al., 2007] is a hardware assisted (Intel VT-x) virtualization scheme which determines the integrity of client programs that connect to a remote server. VIS contains an Integrity Measurement Module which reads the cryptographically signed reference measurement (manifest) of a client process. VIS requires that the pages of the client programs to be pinned in memory (not paged out). VIS restricts network access during the verification phase to prevent any malicious program from bypassing registration. VIS does not allow the client programs unrestricted access to network before the program has been verified.

2.5 Software based integrity measurement schemes

In *Pioneer* [Seshadri et al., 2005], the integrity measurement is done without the help of hardware modules or a VMM. The verification code for the application resides on the client machine. The verifier (server) sends a random number (nonce) as a challenge to the client machine. The response to the challenge determines if the verification code has been tampered or not. The verification code then performs attestation on some entity within the machine and transfers control to it. This forms a dynamic root of trust in the client machine. *Pioneer* assumes that the challenge cannot be redirected to another machine on a network, however, in many real world scenarios a malicious program may attempt to redirect challenges to another machine which has a clean copy of the attestation code. *Pioneer* incorporates the values of Program Counter and Data Pointer, in its checksum procedure; both the registers hold virtual memory addresses. An adversary can load another copy of the client code to be executed in a sandbox like environment and provide it the challenge. This way an adversary can obtain results of the computation that the challenge produces and return it to the verifier.

Genuinity [Kennell and Jamieson, 2003] implements a remote attestation system in which the client kernel initializes the attestation for a program. It receives

executable code and maps it into the execution environment as directed by the trusted authority. The executable code performs various checks on the client program, returns the results to a verified location in the kernel on the remote machine, which returns the results back to the server. The server checks if the results are in accordance with the checks performed, if so the client is verified. This protocol requires operating system (OS) support on the remote machine for many operations including loading the attestation code into the correct area in memory, obtaining hardware values such as TLB. It also requires the client OS to disable interrupts in order to have confidence that the attestation code actually executed. However, if a client OS is corrupted then it may choose to not disable interrupts in which case various meta-information about the process incorporated into the checksum will not be correct. Another problem with this scheme is that the results are communicated to the server by the kernel and not the downloaded code. This may allow a malicious OS to analyze and modify certain values that the code computes.

In *TEAS* [Garay and Huelsbergen, 2006], the authors propose a remote attestation scheme in which the verifier generates program code to be executed by the client machine. Randomized code is incorporated in the attestation code to make analysis difficult for the attacker. The analysis provided by them proves that it is very unlikely that an attacker can clearly determine the actions performed by the verification code; however implementation is not described as part of TEAS and certain implementation details often determine the effectiveness of a particular solution.

2.6 Attacks against software based schemes and counter arguments

Genuinity has been shown to have weaknesses. Genuinity has been shown to fail against a range of attacks known as substitution attacks [Shankar et al., 2004]. The attack suggests placing attack code on the same physical page as the checksum code.

The attack code leaves the checksum code unmodified and writes itself to the zero-filled locations in the page. If the pseudo random traversal maps into the page on which the imposter code is present, the attack code redirects the challenge to return byte values from the original code page. Authors of Genuinity countered these findings by stating that the attack scenario does not take into account the time required to extract test cases from the network, analyze it, find appropriate places to hide code and finally produce code to forge the checksum operations [Kennell and Jamieson, 2004]. The attacks were specifically constructed against one instance of the checksum generation, and would require complex re engineering to succeed against all possible test cases. It is also suggested that Genuinity reads 32 bit words for performing a checksum and hence will be vulnerable if the attack is constructed to avoid the lower 32 bits of memory regions [Seshadri et al., 2004]. These two claims are countered by the authors of Genuinity [Kennell and Jamieson, 2004]. The authors state that Genuinity reads 32 bits at a time, and not the lower 32 bits of an address.

Other works have also been researched against check summing software [Wurster et al., 2005]. However, every attack scenario has its limitations and can be worked around. In this dissertation, remote attestation is implemented by downloading new (randomized and obfuscated) attestation code for every instance of the operation. This operation makes it difficult for the attacker to forge any results that are produced by the attestation code. To launch a successful attack, Mallory would have to perform an 'impromptu' analysis of the operations performed and report the forged results to Trent within a specific time frame. This is considered difficult to achieve.

2.7 Program analysis and code obfuscation

Program analysis requires disassembly of code and the control flow graph (CFG) generation. The linux tool 'objdump' is one of the simplest linear sweep tools that perform disassembly. It moves through the entire code once, disassembling each

instruction as and when encountered. This method suffers from a weakness that it misinterprets data embedded inside instructions hence carefully constructed branch statements induce errors [Schwarz et al., 2003]. Linear sweep is also susceptible to insertion of dummy instructions and self modifying code. Recursive Traversal involves decoding executable code at the target of a branch before analyzing the next executable code in the current location. This technique can also be defeated by opaque predicates [Collberg et al., 1998], where one target of a branch contains complex instructions which never execute [Linn and Debray, 2003].

CFG generation involves identifying blocks of code such that they have one entry point and only one branch instruction with target addresses. Once blocks are identified, branch targets are identified to create a CFG. Compiler optimization techniques such as executing instructions in the delay slot of a branch cause issues to the CFG and require iterative procedures to generate an accurate CFG. The execution time of these algorithms is non-linear (n^2) [Cooper et al., 2002].

2.8 Hardware extensions for virtualization on the Intel platform

Hardware virtualization is a recent development on the x86 platform which provides processor extensions to create a VMM. The processor contains several fields which can be filled during boot or after the native OS has loaded to move the native OS into guest mode. VMM implementations involve a hypervisor that manages one or more VMs by operating at the highest software privilege level (VMX-root mode in VT–x) [Intel Corporation, 2010]. The VMM is invoked on the occurrence of certain events which can be setup prior to executing the VMM. On the occurrence of these events the processor loads the state of the VMM stored in certain area of the memory (termed VMCS in Intel VT–x) and jumps to its entry point. The VMM operates in two modes VMX root mode and VMX non-root mode. The guest runs in non-root mode and the VMM itself runs in the root mode. A control transfer into the VMM (host) is called a

VMExit and a transfer to the VM (guest) is called a VMEntry. The exit and entries happen at certain instructions as specified in the architecture, or as set up by the system administrator. A VM can also explicitly perform a VMExit by executing a VMCall instruction which is similar to a hypercall. The VMM enforces isolation and other system policies. The VMM can manage the launch and shutdown of VMs, memory/device isolation, control register access, MSR access, interrupts and instruction virtualization. Intel VT–x allows a user to override certain sections of the guest operating system routines based on the threat model and usage.

2.9 Hypervisors Utilizing Extensions for Virtualization

BitVisor [Shinagawa et al., 2009] implements a hypervisor that utilizes the drivers of the guest operating system to minimize its code size. The hypervisor implements a set of drivers to mediate all access to devices. BitVisor implements shadow DMA descriptors to control data transferred through DMA between guest OS and devices. BitVisor mediates data transferred between the guest OS and devices by intercepting data I/Os. BitVisor inspects and manipulates the content of data to implement security functionalities such as encryption or intrusion detection. The hypervisor implements parapass-through drivers for each of the device to be monitored. BitVisor also implements instruction emulators to handle mode transitions between real mode and protected mode of the Intel x86 based CPU. Due to these additions in it, the size of BitVisor is estimated as being close to 20KLOC, the size of each para-pass through drivers is an addition to this code size.

SecVisor [Seshadri et al., 2007] is a hypervisor that virtualizes Memory Management Unit and the IO Memory Management Unit of the CPU to allow hardware protections to be set over kernel memory. SecVisor checks all modifications to MMU and IOMMU state to protect protected code from DMA writes. SecVisor depends on user supplied policy to approve code that can be executed by the kernel.

Its security relies on a user inputs of trusted code, it can be noted that smart malware may attempt to fake user responses and inputs to send messages to the SecVisor TCB to modify its trusted code lists.

MAVMM [Nguyen et al., 2009] builds a minimal VMM that can extract information such as execution traces, memory dumps, system calls, disk accesses, and network interactions from programs running on the guest OS. MAVMM protects the hypervisor memory from being tampered by the guest using nested paging and protects from external DMA writes by using the IOMMU features of the virtualization extensions of the processor. MAVMM single steps through guest applications to determine instructions executed. MAVMM does not utilize the guest OS drivers to extract data; instead it uses a serial port to extract data with the help of BIOS. MAVMM has the ability to selectively monitor some processes and ignore other ones. To achieve this, a user level application specifies the names of the processes to be tracked. The hypervisor section of MAVMM is written in nearly 4KLOC. In comparison MIvmm implemented in this work has the core of the code to be around 2.5 KLOC. This is because MIvmm does not implement any security features, it merely offers a system which is small in code size and offers the capability to build various applications on top of it.

Bluepill [Rutkowska, 2006] is a rootkit that utilizes hardware extensions for virtualization provided by Intel VT –x and AMD –V to move the native operating system into a shell monitored by it. Bluepill identifies a paged out driver to be written on and writes binary code on the paged out code of the device driver. When the driver is loaded back to memory the injected code executes. The injected code turns on the hardware virtualization feature and forces the Vista Operating System to migrate to the guest environment. Bluepill does not survive system reboot. Vitriol [Zovi, 2006] is also a hardware assisted virtualization rootkit which executes on a platform having Intel VT–x which works in a similar fashion to Bluepill. Both rootkits implement only

the bare features necessary to implement a hypervisor using the hardware extensions on the x86 architecture.

Chapter 3

THREAT MODEL AND ASSUMPTIONS

This chapter provides the threat model and assumptions that will be used in each of the implemented works. Since each work is slightly different in its aim and scope, each work has a separate threat model and assumptions.

3.1 Threat model and assumptions for user application attestation

It is assumed that Mallory may have installed a backdoor at M_{Alice} which can inform Mallory that an attestation process has been initiated. The backdoor may divert the challenge to another machine inside Mallory's control which can provide the response for the challenge. The backdoor can also use dis-assembly tools to determine the operations performed by the challenge. In addition the backdoor may attempt to execute the challenge inside a sandbox to determine the results of the response.

Since Trent is a trusted server, it is assumed that Alice will execute the code provided by Trent. Trent may be the vendor of the binary or a commercial provider of remote attestation service for many binaries. It is also assumed that Alice has a digital signature scheme, which can identify that the executable code was generated by Trent. Because attestation code determines the IP address of the client which serves as its machine identifier, it is assumed that Alice is not executing the programs behind a NAT. This assumption is made as $\mathscr{C}$ takes measurements on M_{Alice} to determine if it is the same machine that contacted Trent. If $\mathscr{M}_{Alice}$ is behind a NAT the connections would appear to be coming from a router and not the machine, while $\mathscr{C}$ would respond with the machine IP. It can also be noted that in case Trent is a network administrator, the NAT does not come into play at all as the attester would be inside the NAT. In a home computing scenario, often there is only one computer on the network, so the case of another machine masquerading with the same IP can be ignored. Hence, the IP

measurement check can be done away with. Also many routers allow one machine inside the NAT to be placed outside it. This option can be used temporarily during communication between Trent and Alice.

$\mathscr{C}$ executes OS calls by using software interrupts. Due to this, it is assumed that the OS on M_{Alice} is not compromised by a *rootkit*. The presence of a *rootkit* would require the use of a VMM or a hardware based checker to determine integrity. Also so note that there are many software interrupts in the Linux operating system, due to which it can be assumed that a user level malware will find it difficult to intercept the operations of software interrupts.

It is assumed that $\mathscr{P}$ is not self-modifying code. Any integrity measurement technique cannot obtain measurements on self-modifying code because the state of the code section changes with time and execution. Moreover, on computing platforms based on the Intel x86 architecture, the code section is 'write protected' by default, which reduces the scenario of self modifying code existing in common applications. It is also assumed that Mallory may attempt to change the application $\mathscr{P}$ after it has been attested by Trent. To prevent this scenario, the first scheme is extended to determine whether the attested binary continued execution or was replaced by an attacker.

3.2 Threat model and assumptions for kernel attestation

For the kernel attestation part, this work assumes that the kernel is compromised; system call tables may be corrupted, and a malware may have changed the interrupt descriptors. Runtime code injection is performed on a kernel module to measure the integrity of the kernel. It is assumed that Alice has means such as digital certificates to determine that the code being injected is generated by a trusted server. It is also assumed that the trusted server is the OS vendor or a corporate network administrator with knowledge of the OS mappings for the client.

3.3 Threat model and assumptions for guest OS attestation using KVM

For the virtualization based OS kernel attestation, this work uses Linux-KVM and qemu emulator. It is assumed that the implementation of the qemu and kvm interface is secure. This means that it is assumed that no malware can exploit any bugs in the interface to exploit the Host OS. It is assumed that the Guest OS may be completely corrupted, but the Host OS is clean. The external server receives a connection request from the guest OS and requests the integrity measurements of the guest OS by communicating to the Host OS. It is assumed that the trusted entity knows the IP address of the Host machine for the guest OS in question. It is assumed that the Host OS runs on an Intel x86 based machine which has virtualization extensions VT-x built in its hardware. This assumption is made for KVM support. It is assumed that the Host has means such as digital signatures to verify Trent.

3.4 Threat model and assumptions for minimal VMM creation

For the minimal VMM creation section, it is assumed that the platform on which the VMM executes will have Intel-VT capabilities. It is assumed that the native OS is clean prior to launch of the VMM. Although this seems restrictive, it is required only as long as the VMM is implemented as a minimal feature VMM. If the ability to take integrity measurements on the native OS is incorporated in the VMM, then this assumption is not required. It is assumed that the processor will behave correctly and trap the execution of sensitive instructions into the VMM.

Chapter 4

DESIGN OF INTEGRITY MEASUREMENT CODE

Trent is a trusted server that provides integrity measurement code $\mathscr{C}$ to Alice. Alice injects the code on the user application $\mathscr{P}$. $\mathscr{P}$ transfers control to $\mathscr{C}$ and allows it to report measurements to Trent. Trent must prevent Mallory from analyzing the operations performed by $\mathscr{C}$. To achieve this, Trent can utilize a combination of obfuscation techniques.

Trent also maintains a time threshold (T) by which the response from M_{Alice} is expected. If $\mathscr{C}$ does not respond in a stipulated period of time (allowing for network delays), Trent will know that something went wrong at M_{Alice}. This includes denial of service based attacks where Trent will inform Alice that $\mathscr{C}$ is not communicating back.

Fig. 4.1 shows a sample snippet of the $\mathscr{C}$ mathematical checksum code. The send function used in the checksum snippet is implemented using inline ASM. It is evident that in order to forge any results, Mallory must determine the value of checksum2 being returned to Trent. This requires that Mallory identify all the instructions modifying checksum2 and the locations on stack that it uses for computation. To prevent Mallory from analyzing the injected code, certain obfuscations are placed in $\mathscr{C}$ as discussed below:

4.1 Changing execution flow and locations of variables on stack

To prevent Mallory from using knowledge about a previous instance of $\mathscr{C}$ in the current test, Trent changes the checksum operations performed by selecting mathematical operations on memory blocks from a pool of possible operations and also changes the order of the instructions. The results of these operations are stored

```
{
   ....
   x = <random value>
   a = 0;
   while (a<400) {
      checksum 1 += Mem[a];
      if ((a  % 55) == 0) {
         checksum2 += checksum1/x;
      }
      a++;
   }
   send checksum2;
   ....
}
```

Figure 4.1: Snippet from the checksum code

temporarily in the stack. Trent changes the pointers on the stack for all the local variables inside $\mathscr{C}$ for every instance. These steps prevent Mallory from successfully launching an attack similar to those used for HD-DVD key stealing [Weblink, e],[Weblink, f].

4.2 Inserting dummy instructions

Program Analysis is a non linear operation as discussed in section 2.7. An increase in the number of instructions that Mallory has to analyze decreases the time window available to forge the results of these operations. Trent inserts instructions that never execute and also inserts operations that are performed on M_{Alice} but not included as part of the results sent back to Trent. These additions to the code make it difficult for Mallory to correctly analyze $\mathscr{C}$ within a reasonable period of time.

4.3 Changing instructions during execution

Mallory may perform static analysis on the executable code $\mathscr{C}$ sent by Trent. A good disassembler can provide significant information on the instructions being executed, and allow Mallory to determine when system calls are made and when function calls

are made. In addition, it may also allow Mallory to see the area of code which reads memory recursively. If these tools do not have access to the code to be executed before it actually executes, then Mallory cannot determine the operations performed by $\mathscr{C}$. Trent removes some instructions in $\mathscr{C}$ while sending the code to M_{Alice} and places code inside $\mathscr{C}$ with data offsets, such that during execution, this section in $\mathscr{C}$ changes the modified instructions to the correct values. Therefore, without executing $\mathscr{C}$, it is difficult for Mallory to determine the exact contents of $\mathscr{C}$.

4.4 Implementation

Changing execution flow and locations of variables on the stack

Changing execution flow and locations on stack prevents the program analysis on $\mathscr{C}$. The source code of $\mathscr{C}$ was divided into four blocks which are independent of each other. Trent assigns randomly generated sequence numbers to the four blocks and places them accordingly inside $\mathscr{C}$ source code.

The checksum block is randomized by creating a pool of mathematical operations that can be performed on every memory location and selecting one operation from the pool of operations for each memory slot. The pool of operations is created by replacing the mathematical operation with other mathematical operations on the same location.

Once the mathematical operations are selected in the $\mathscr{C}$ source code, Trent changes the sub-regions for the checksum code and the MD5 calling procedure. This is done by replacing the numbers defining the sub-regions. $\mathscr{C}$ has sub-regions defined in its un-compiled code. To randomize the sub-regions, a pre-processor is executed on the un-compiled $\mathscr{C}$ so that it changes the numbers defining the sub-regions. The numbers are generated so that the sub-regions randomly overlap.

$\mathscr{C}$ allocates space on the local stack to store computational values. Instead of using fixed locations on the stack, Trent replaces all variables inside $\mathscr{C}$ with pointers

to locations on the stack. To allocate space on the stack Trent declares a large array of type 'char' of size N, which has enough space to hold contents of all the other variables simultaneously. Trent executes a pre-processor which assigns locations to the pointers. The pre-processor maintains a counter starting at 0 and ending at N-1. It randomly picks a pointer to assign a location and assigns it the value on the counter and increments the counter using the size of the corresponding variable in question. This continues until all the pointers are assigned a location on the stack. Trent compiles $\mathscr{C}$ source code to produce the executable $\mathscr{C}$ by placing these obfuscations.

Obfuscating instructions executed

Mallory cannot obtain a control flow graph (CFG) or perform program analysis on the executable code of $\mathscr{C}$ if the instruction being executed by $\mathscr{C}$ cannot be determined. Trent changes the instructions inside the executable code so that they cause analysis tools to produce incorrect results. $\mathscr{C}$ contains a section ($\mathscr{C}_{restore}$) which changes these modified instructions back to their original contents when it executes. $\mathscr{C}_{restore}$ contains the offset from the current location and the value to be placed inside the offset. Trent places information to correct the modified instructions inside $\mathscr{C}_{restore}$. $\mathscr{C}_{restore}$ is executed prior to executing other instructions inside $\mathscr{C}$ and $\mathscr{C}_{restore}$ corrects the values inside the modified instructions.

Chapter 5

REMOTE ATTESTATION OF USER APPLICATION $\mathscr{P}$

If Alice could download the entire copy of $\mathscr{P}$ every time the program had to be executed then Remote Attestation would not be required. However, since $\mathscr{P}$ is an installed application, Alice must have customized certain profile options, saved some data which will be cumbersome to create every time.

Alice uses $\mathscr{P}$ to contact Trent for a service, Trent returns to $\mathscr{P}$: a challenge which is executable code ($\mathscr{C}$). $\mathscr{P}$ must inject $\mathscr{C}$ in its virtual memory and execute it at a location specified by Trent. $\mathscr{C}$ computes integrity measurements and communicates the integrity measurement value M_1 directly to Trent. Trent has a local copy of $\mathscr{P}$ on which the same sets of tests are executed as issued to the client to produce an integrity measurement value M_0. Trent compares M_1 and M_0; if the two values are the same then Alice is informed that $\mathscr{P}$ has not been tampered. Trent wants to be certain that $\mathscr{C}$ took its measurements on $\mathscr{P}$ residing inside M_{Alice}. To provide this guarantee, $\mathscr{C}$ executes some more tests on M_{Alice} and returns their results to Trent. These checks ensure that $\mathscr{C}$ was not bounced to another machine, and that it was not executed in a sandbox environment inside a dummy process $\mathscr{P}_{dummy}$ within M_{Alice}.

There are many ways in which Mallory may tamper with the execution of $\mathscr{C}$. Mallory may substitute values of M_1 being sent to Trent, such that the evidence of modification of $\mathscr{P}$ is not discovered by Trent. It is also possible that Mallory may have loaded a clean copy of $\mathscr{P}$ inside a sandbox, execute $\mathscr{C}$ within it, and provide the results back to Trent. Mallory may redirect the challenge to another machine on the network in order to compute the integrity measurements and send the responses back to Trent. Without addressing these issues, it is not possible for Trent to correctly determine whether the measurements accurately reflect the state of $\mathscr{P}$ on M_{Alice}. If Trent can determine that $\mathscr{C}$ executed on M_{Alice}, $\mathscr{C}$ was not executed in a sandbox, and

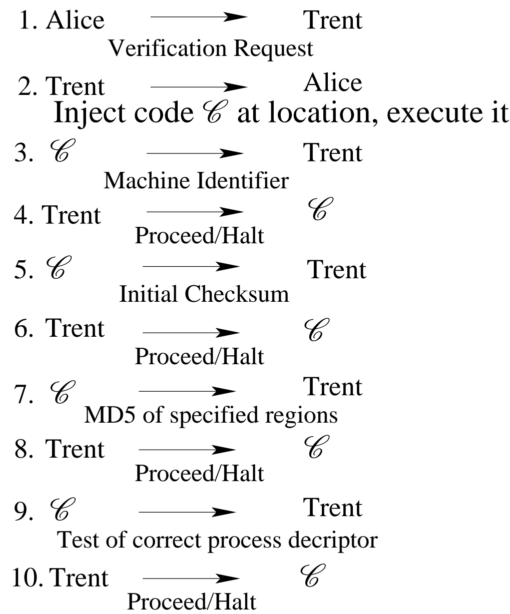

Figure 5.1: Detailed steps in Remote Attestation process

Trent can produce code whose results are difficult to guess, then the results can indicate the correct state of $\mathscr{P}$. Achieving these guarantees also requires that $\mathscr{C}$ provides Trent with a machine identifier and a process identifier of M_{Alice}.

Trent can retain a sense of certainty that the results are genuine by producing code that makes it difficult for Mallory to pre-compute results. Once these factors are satisfied, Trent can determine whether $\mathscr{P}$ on M_{Alice} has been tampered. Fig. 5.1 shows the detailed steps in performing Remote Attestation.

5.1 Implementation

Injection of Code on $\mathscr{P}$

The attestation code $\mathscr{C}$ is injected by $\mathscr{P}$ on itself. This allows $\mathscr{C}$ to execute within the process space of $\mathscr{P}$. This way $\mathscr{C}$ can use all descriptors of $\mathscr{P}$ on M_{Alice} without creating new descriptors. The advantage of this is that $\mathscr{C}$ cannot be executed in a sandbox easily and $\mathscr{C}$ can also determine whether more than one set of descriptors are present for $\mathscr{P}$. At the client side $\mathscr{P}$ makes a connection request to Trent. Trent responds by providing the size of attestation routine $\mathscr{C}$ followed by the actual

executable code to determine the integrity of $\mathscr{P}$. Trent also sends the information on the location inside $\mathscr{P}$ where $\mathscr{C}$ should be placed. $\mathscr{P}$ receives the code and prepares the area for injection by executing the library utility *mprotect* [Weblink, k] on the area. Once injection is complete, $\mathscr{P}$ creates a function pointer which points to the address of the location and calls $\mathscr{C}$ using the pointer.

Communication with Trent

The attestation routine does not have any calls to system libraries. This is because libraries may get compromised by an attacker to return incorrect results. In addition, the references to libraries are present at different location in every machine. It is easier to generate interrupts to execute the required functionality instead of placing the correct references to the libraries in C. Moreover, a call to a system library may expose the functionality of the code to Mallory. Execution of libraries for communication is achieved by executing the software interrupt with the interrupt number for the OS call *socketcall*.

Communication to Trent is achieved by using the socket connection that $\mathscr{P}$ created for an attestation request. All messages are sent to Trent using the *socketcall* [Weblink, i] system call. ASM code for a network send using *socketcall* is shown in Fig. 5.2. The routine allocates space on the stack for the parameter, followed by placing the parameters on the stack. The system call number for *socketcall* is 102, which is moved into the A register. The call number for a send in *socketcall* is 9, this value is moved to the B register, then the location of the parameters are moved to the C register and the system call is executed using the interrupt instruction (INT 80). Once the interrupt returns the stack is restored to the original value and the result is obtained in the A register. The functions provided inside *socketcall* is present in the Linux source code in the file $< include/linux/net.h >$.

```
_asm_ (
        "sub  $16,%%esp\n"
        "movl   %%ebx,(%%esp)\n"
        "movl   %%ecx,4(%%esp)\n"
        "movl   %%edx,8(%%esp)\n"
        "movl   $0,12(%%esp)\n"
        "movl   $102,%%eax\n"
        "movl   $9,%%ebx\n"
        "movl   %%esp,%%ecx\n"
        "int    $0x80\n"
        "add    $16,%%esp\n"
        : "=a" (res)
        :"b" (send_sock), "c" (p_MD5Buf), "d" (len)
    );
```

Figure 5.2: *send* routine through *socketcall* in ASM

Determining Machine Identifiers

To determine that $\mathscr{C}$ is not re-directed to another machine, Trent obtains the machine identifier on which $\mathscr{C}$ executes. Trent had received the request for attestation from Alice, hence has access to the IP address of the machine from which the request came. $\mathscr{C}$ obtains the IP address of the platform on which it is executing and communicates the result to Trent. Trent compares the two values to determine if the platform in which $\mathscr{C}$ is obtaining results is the same as the platform from which the initial attestation request came. It can be argued that IP addresses are dynamic; however there is little possibility that any machine will change its IP address in the small time window between Alice requesting a challenge - to measurements being provided by $\mathscr{C}$ to Trent. M_{Alice} is not behind a NAT; hence Trent observes the IP address of M_{Alice} and $\mathscr{C}$ reports the same address. It can be argued that Mallory may have redirected the challenge to another machine ($M_{Mallory}$), and changed the address of the network interface on $M_{Mallory}$ to match that of M_{Alice}. But as M_{Alice} is not behind a NAT it would be difficult for Mallory to provide the address to another machine on an

external network and achieve successful communication.

$\mathscr{C}$ determines the IP address of M_{Alice} using system interrupts. The interrupt ensures that the address present on the network interface is correctly reported to Trent. This involves loading the stack with the correct operands for the system call, placing the system call number in the EAX register and loading the other parameters in registers EBX, ECX, EDX and executing the interrupt instruction. Reading the IP address involves creating a socket [Weblink, l] on the network interface and obtaining the address from the socket by using another system call *ioctl* [Weblink, g]. The obtained address is in the form of an integer which is converted to the standard A.B.C.D representation.

Determining MD5 and Arithmetic Checksum

To determine whether the code section of $\mathscr{P}$ has been tampered, $\mathscr{C}$ computes an MD5 hash on the code section of $\mathscr{P}$. It is possible that since the code section of the binary is available, Mallory may compute the MD5 hash of every possible boundary region prior to Trent sending a challenge. To prevent this attack, Trent defines sub-regions in the binary and overlaps on the sub-regions before measuring the MD5 hash of the overlapping regions. Overlapping checksums ensure that if by accident the sub-regions are defined identically in two different versions of $\mathscr{C}$, the overlap provides a second set of randomization and ensures that the results of computation produced by $\mathscr{C}$ are different. This also ensures that some random sections of $\mathscr{P}$ are present more than once in the checksum to make it more difficult for Mallory to hide any modifications to such regions.

To increase the complexity of the attestation procedure, Trent changes the MD5 measurement to a two phase protocol. MD5 code cannot be randomized. The only changes that can be made are to the overlapping sub-regions. To prevent possible attacks on this protocol, Trent also obtains an arithmetic checksum of the code section

of $\mathscr{P}$. The checksum is taken on overlapping sub-regions as described above. The sub-regions defined for the arithmetic checksum are different from the sub-regions defined for obtaining the MD5 hash.

The sub-regions on the MD5 hash are defined by Trent in the source code of the attestation routine using constants. Prior to compilation, Trent runs a pre-processor which generates random numbers to change these constants. The checksum operations are randomized by creating a basic arithmetic operation for a memory location and modifying the basic arithmetic operation to create alternate operations. This provides a pool of operations that can be performed on each memory location. During code generation, one operation is randomly selected for each memory location and placed in the attestation routine. This changes the arithmetic operations performed for every attestation request. The results of these operations are stored temporarily on the stack. Trent changes the pointers on the stack for all the local variables inside $\mathscr{C}$ for every instance. These steps prevent Mallory from successfully launching an attack similar to those used for HD-DVD key stealing [Weblink, e], [Weblink, f]. Trent places dummy instructions that never execute and inserts some operations that are performed on M_{Alice}, but not included as part of the results sent back to Trent. Trent also places a time limit (T) within which the response for these computations must be received. The addition of these operations is aimed to make analysis of operations within the time frame difficult for Mallory.

Determining Process Identifiers

To determine that the attestation routine was not bounced to execute inside a second copy of $\mathscr{P}$, Trent obtains the state of the machine by comparing the open descriptors on M_{Alice} against a known state of a clean machine. Trent knows that in a clean machine there must be only one set of file descriptors used by $\mathscr{P}$. If there are multiple copies of the descriptors used by $\mathscr{P}$, then an error is reported to Trent. $\mathscr{C}$ identifies

sl	local_address	rem_address	st	tx_queue rx_queue	tr tm->when	retrnsmt	uid	timeout	inode
0:	0100007F:1F40	00000000:0000	0A	00000000:00000000	00:00000000	00000000	0	0	5456
1:	00000000:C3A9	00000000:0000	0A	00000000:00000000	00:00000000	00000000	0	0	4533
2:	00000000:006F	00000000:0000	0A	00000000:00000000	00:00000000	00000000	0	0	4473
3:	0100007F:0277	00000000:0000	0A	00000000:00000000	00:00000000	00000000	0	0	5690
4:	0100007F:0019	00000000:0000	0A	00000000:00000000	00:00000000	00000000	0	0	5358
5:	0100007F:743A	00000000:0000	0A	00000000:00000000	00:00000000	00000000	0	0	5411

Figure 5.3: Contents of /proc/net/tcp file

descriptors that match the known descriptors used by $\mathscr{P}$ and determines the process using these descriptors in the system. If the process using these descriptors are the same as the process inside which $\mathscr{C}$ executes, then an OK state is sent to Trent.

$\mathscr{C}$ obtains the pid of the process ($\mathscr{P}_0$) under which it is executing using the system interrupt for *getpid* [Weblink, c]. It locates all the remote connections established to Trent from M_{Alice}. This is done by reading the contents of the '/proc/net/tcp' file. The file has a structure shown in Fig. 5.3. This file has some more fields that are omitted from the figure. Once all the connections are identified, $\mathscr{C}$ utilizes the *inode* of each of the socket descriptor to locate any process using it. This is done by scanning the '/proc/$< pid >$/fd' folder for all the running processes on M_{Alice}. In the situation that $\mathscr{P}$ is not corrupted, there should be only one process id ($\mathscr{P}_0$) using the identified inode. If $\mathscr{C}$ encounters more than one such process, then it sends an error message back to Trent.

5.2 Results

The remote attestation scheme was implemented on Ubuntu 8.04 (Linux 32 bit) operating system using the gcc compiler; the application $\mathscr{P}$ and attestation code $\mathscr{C}$ were written in the C language. The time threshold (T) is an important parameter in this implementation. The value of T must take into account network delays. Network delays between cities in IP networks are of the order of a few milliseconds [Weblink, d]. Measuring the overall time required for one instance of Remote Attestation and adding a few seconds to the execution time can suffice for the value of T. The

Table 5.1: Average code generation time at server end

Machine	Test generation time (ms)	Compilation time (ms)	Total time (ms)
Pentium 4	12.3	320	332
Quad Core	5.2	100	105

Table 5.2: Time to compute measurements

Machine	Server side execution time (ms)	Client side execution time (ms)
Pentium 4	0.6	22
Quad Core	0.4	16

performance of the system was measured by executing the integrity checks on the source code for VLC media player interface [Weblink, n]. Some sections of the program were removed for compilation purposes. The performance of the system was measured on two pairs of systems. One pair of machines were legacy machines executing on an Intel Pentium 4 processor with 1 GB of ram, and the second pair of machines were Intel Core 2 Quad machine with 3 GB of ram. The tests measured were: the time taken to generate code including compile time, time taken by the server to do a local integrity check on a clean copy of the binary and time taken by the client to perform the integrity measurement and send a response back to the server. The time taken for compiling the freshly generated code is reported in Table 5.1. As expected, the Pentium 4 machine has slightly lower performance than a platform with 4 Intel Core 2 processors.

The integrity measurement code $\mathscr{C}$ was executed locally on the server, and also sent to the client for injection and execution. The time taken on the server to execute is the time the code will take to generate integrity measurement on the client, because both machines were kept with the same configuration in each case. These times are reported in Table 5.2. As the code takes only in the order of milliseconds to execute on the client platform, the value for T can be set in the order of a few seconds to allow for network delays.

It can be observed from Table 5.1 that it takes an order of a few hundred

milliseconds for the server to generate code, while from Table 5.2 it can be observed that the integrity measurement is very light-weight and returns results in the order of a few milliseconds. As a result, the code generation process can be viewed as a huge overhead. However, the server need not generate new code for every instance of a client connection. It can generate the measurement code periodically every second and ship out the same integrity measurement code to all clients connecting within that second. This can alleviate the workload on the server.

Chapter 6

VERIFIED CODE EXECUTION

Once Remote Attestation determines the integrity of a program, the server begins communication and sharing of sensitive data to the client program. However, Mallory, the attacker, may choose to wait till the attestation process is completed and then substitute the client program $\mathscr{P}$ with a corrupted program $\mathscr{P}_c$. To prevent Mallory from doing this, Trent has to obtain some guarantee that the process that was attested earlier is the same process performing the rest of the communication. Trent cannot make any persistent changes to the binary as Mallory would detect these changes under the current threat model. Trent has to change the flow of execution from normal in the client process such that the sequence of events reported will allow Trent to determine whether the attested process is executing.

As discussed before, Trent knows the layout of the program $\mathscr{P}$. At the end of Remote Attestation, Trent sends a new group of messages to $\mathscr{C}$. The message contains some code executable code $\mathscr{F}_1$ that Trent instructs $\mathscr{C}$ to place at a particular location in $\mathscr{P}$. Trent also instructs $\mathscr{C}$ to modify a future function call F0 in $\mathscr{P}$ such that instead of calling $\mathscr{F}_0$, $\mathscr{P}$ calls $\mathscr{F}_1$. $\mathscr{F}_1$ communicates back to Trent and this way Trent knows that the copy of $\mathscr{P}$ which was attested in the previous step is still executing. At the end of its execution, $\mathscr{F}_1$ undoes all its stack operation and jumps to the address where $\mathscr{F}_0$ is located. If $\mathscr{F}_1$ executes a return instruction then control would move back to $\mathscr{P}$ and some functionality of $\mathscr{P}$ would be lost. It cannot execute a function call to $\mathscr{F}_0$ as this may cause loss of some parameters passed by $\mathscr{P}$.

6.1 Stack allocation in Intel architecture

In the Intel x86 implementation of Linux, stack is defined during compilation time and allocated only during runtime. Every function allocates the required stack at the start

```
int C1()
{
  char mesg_string[20];
  strcpy(mesg_string, "Hello World");
  printf("\n %s", mesg_string);
  return 0;
}
  8048514: 55       push %ebp
  8048515: 80 e5    mov %esp, %ebp
  8048517: 80 ec 38 sub $0x38, %esp
  .......
  .......
  8048569: c9       leave
  804856a: c3       ret
```

Figure 6.1: Sample C routine and its disassembly

of its execution by subtracting required amount of bytes for local memory from the stack pointer using the SUB instruction.

Fig. 6.1 shows a sample routine and its disassembly. As seen in the code snippet, the code first saves the base pointer on the stack; this action saves the frame for the previous function. The execution then moves the current stack pointer (which points to the location beyond the last push), into the base pointer; this is done as most of the addressing inside a routine is performed relative to the current base pointer. The execution then subtracts some memory from the current stack pointer to allocate memory for the local variables. The rest of the instructions constitute the program functionality. The last two instructions are *leave* and *ret*. The leave instruction reverses all net stack operations performed by the function and it pops the value stored in stack for the base pointer. The ret instruction resumes execution at the return address stored on the stack.

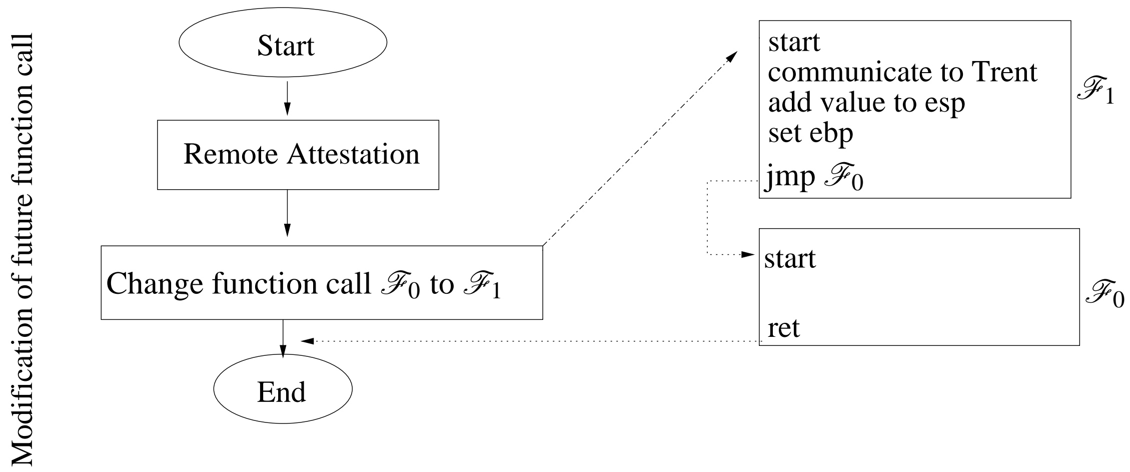

Figure 6.2: Change of flow of execution

6.2 Executing $\mathscr{F}_0$ after $\mathscr{F}_1$ without executing a RET

$\mathscr{F}_1$ allocates some stack for its local memory in a fashion similar to above. However, instead of letting it return to $\mathscr{P}$, a jump instruction will be executed after the stack operations are reversed. $\mathscr{F}_1$ can either move the value of the frame pointer into the stack pointer to reverse stack allocation (which is equivalent of performing an addition on the stack pointer), pop the current stack value into the base pointer, and then jump to the start of $\mathscr{F}_0$, or execute the leave instruction and jump to $\mathscr{F}_0$. This allows $\mathscr{F}_0$ to receive all parameters passed on the stack by $\mathscr{P}$ and resume normal execution. When $\mathscr{F}_0$ executes a return instruction, the control moves back to $\mathscr{P}$. Fig. 6.2 shows the represents this process diagrammatically.

6.3 Implementation

As part of sending messages to $\mathscr{C}$, Trent provides the code of $\mathscr{F}_1$, the location where $\mathscr{F}_1$ should be placed and a particular address inside $\mathscr{P}$ which corresponds to a function call to $\mathscr{F}_0$. $\mathscr{C}$ places the code $\mathscr{F}_1$ at the specified location and changes the target of the call instruction inside $\mathscr{P}$ to point to $\mathscr{F}_1$. When $\mathscr{F}_1$ executes, it communicates to Trent and informs Trent that it executed. It can be noted here that $\mathscr{F}_1$

can use the existing connection to Trent or open a new connection. Since Trent generated $\mathscr{F}_1$, Trent can place a random secret inside $\mathscr{F}_1$ which gets communicated to Trent. On receiving the secret Trent knows that $\mathscr{F}_1$ executed.

```
__asm__  ("mov %ebp, %esp \n"
          "pop       %ebp \n"
          "jmp       0x8048bff \n");
```

Figure 6.3: Tail portion of $\mathscr{F}_1$

The tail portion of $\mathscr{F}_1$ is provided with code similar in functionality as shown in fig. 6.3. $\mathscr{F}_1$ clears its stack by moving the base pointer into the stack pointer. It then pops the base pointer value of the previous routine into EBP. Then finally a jump is performed to $\mathscr{F}_0$. As $\mathscr{F}_1$ is compiled as a standalone function, the gcc complier generates an incorrect target address for the Jump instruction. This is fixed by the

```
int fix_address(void){
      int length_ofF1= 0xbd;
      int location_F0 = 0x08048bff;
      int location_F1 = 0x08048c92;
      int offset_of_jmp_in_F1 = 0xa3;
      int eip_offset_for_jmp = 5;
      ....
      T_address = location_F0 -
                  (location_F1 +
                    offset_of_jmp_in_F1+
                    eip_offset_for_jmp );
      Write_back[0xa4]  =  T_address & 0x000000FF;
      Write_back [0xa5] = (T_address & 0x0000FF00) >>8;
      Write_back [0xa6] = (T_address & 0x00FF0000) >>16;
      Write_back [0xa7] = (T_address & 0xFF000000) >>24;
      ....
}
```

Figure 6.4: Fixing Jump target

server side program by correcting the target of the jump instruction as seen in fig. 6.4. The code calculates the actual 4 byte address for the JMP instruction and then writes it back in the binary in the little-endian format.

Chapter 7

KERNEL ATTESTATION

To measure the integrity of the kernel we implement a scheme which is similar to the user application attestation scheme. Trent$'$ is a trusted server who provides code ($\mathscr{C}_{kernel}$) to M_{Alice}. It is assumed that Alice has means such as digital signature verification scheme to determine whether $\mathscr{C}_{kernel}$ was sent by Trent$'$. Alice receives $\mathscr{C}_{kernel}$ using a user level application $\mathscr{P}_{user}$, verifies that it was sent by Trent$'$ and places it in the kernel of the OS executing on M_{Alice}. $\mathscr{C}_{kernel}$ is then executed and obtains integrity measurements (H_{kernel}) on the OS Text section, system call table, and the interrupt descriptors table. $\mathscr{C}_{kernel}$ passes these results to P_{user}, which returns these results to Trent$'$. If required $\mathscr{C}_{kernel}$ can encrypt the integrity measurement results using a onetime pad or a simple substitution cipher, however, as the test case generated is different in every instance this is not a required operation. Trent$'$ also provides a kernel module $\mathscr{P}_{kernel}$ that provides *ioctl* calls to $\mathscr{P}_{user}$. As seen in figure 7.1, $\mathscr{P}_{user}$ receives $\mathscr{C}_{kernel}$ from Trent$'$. In figure 7.2, $\mathscr{P}_{user}$ forwards the code to $\mathscr{P}_{kernel}$.

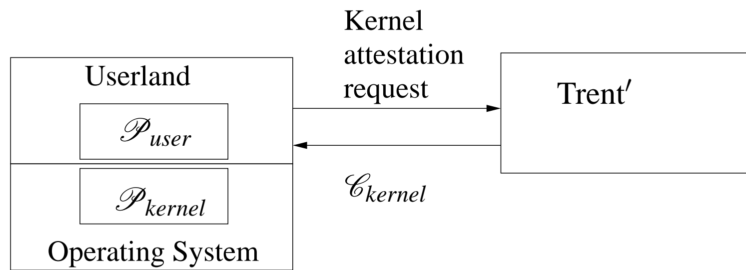

Figure 7.1: user application initiates attestation request

$\mathscr{P}_{kernel}$ places the received code in its code section at a location specified by Trent$'$ and executes it. $\mathscr{C}_{kernel}$ obtains an arithmetic and MD5 checksum on the specified regions of the kernel on M_{Alice} and returns the results to $\mathscr{P}_{user}$ as seen in figure 7.3. $\mathscr{P}_{user}$ then forwards the results to Trent$'$ who determines whether the measurements obtained from the OS on M_{Alice} match with existing computations as

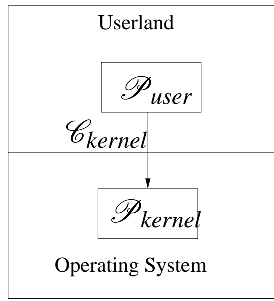

Figure 7.2: user application sends attestation code to kernel space

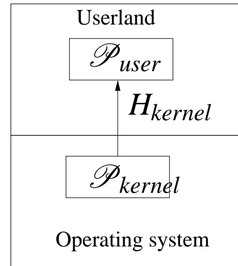

Figure 7.3: kernel returns integrity measurements to user land

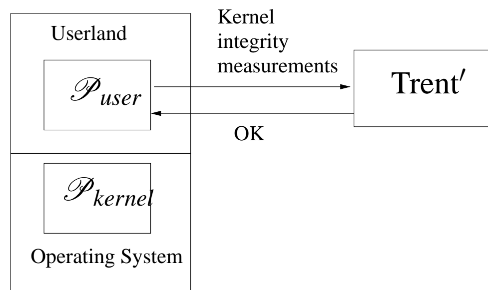

Figure 7.4: Verification of kernel integrity by trusted server

shown in figure 7.4. Since Trent′ is an OS vendor or a corporate network administrator, it can be assumed that Trent′ has local access to a pristine copy of the kernel executing on M_{Alice} to obtain expected integrity measurement values generated by $\mathscr{C}_{kernel}$. Although this seems like Trent′ would need infinite memory requirements to keep track of every client, most OS installations are identical as they are off the shelf. In addition, if Trent is a system administrator for a number of machines on a corporate network, Trent′ would have knowledge of the OS on every client machine.

7.1 Implementation

The kernel attestation was implemented on an x86 based 32 bit Ubuntu 8.04 machine executing with 2.6.24-28-generic kernel. In Linux the exact identical copy of the kernel is mapped to every process in the system. For the application attestation described in previous chapters, the support of OS is required in the form of system calls and interrupts. The system calls and interrupts are stored inside the high memory (above 3 GB) of the process space in a 32 bit Linux OS. The high memory constitutes kernel memory. We need to ensure that these sections of the OS are clean while providing integrity measurements of a process. The following section descibes the integrity measurement of the OS text section, system call table and the interrupt descriptor table.

Identifying locations to measure in kernel

The /boot/System.map-2.6.24-28-generic file on the client platform was used to locate the symbols to be used for kernel measurement. The kernel text section was located at virtual address 0xC0100000 and the end of kernel text section was located to be at 0xC03219CA which corresponded to the symbol '_etext'. The system call table was located at 0xC0326520, the next symbol in the maps file was located at 0xC0326B3C, a difference of 1564 bytes. The 'arch/x86/include/asm/unistd_32.h' file for the kernel build showed the number of system calls to be 337. Since M_{Alice} was a 32 bit system, the space required for the address mappings would be 1348 bytes. We took integrity measurements from 0xC0326520 - 0xC0326B3B. The Interrupt descriptor table was located at 0xC0410000 and the next symbol was located at 0xC0410800, which gives the IDT a size of 2048 bytes. A fully populated IDT should be 256 entries of 8 bytes each which gives a 2KB sized IDT, this is consistent with the System.maps file on the client machine.

Communication with Trent′

The trusted server Trent′ communicates to a user level application $\mathscr{P}_{user}$. $\mathscr{P}_{user}$ can be assumed to be an application provided by Trent′. Trent′ also provides a kernel module ($\mathscr{P}_{kernel}$) to the client platform which is installed as a device driver for a character device. $\mathscr{P}_{user}$ communicates to a kernel module $\mathscr{P}_{kernel}$ using the *ioctl* interface provided by a character device 'remote_attestation_device' which is created using a command 'mknod /dev/remote_attestation_device c 100 0'. The last two numbers in the command provide the MAJOR_NUM and MINOR_NUM for the device.

$\mathscr{P}_{user}$ receives the code from the trusted authority and opens the char device. $\mathscr{P}_{user}$ then executes an *ioctl* which allows the kernel module to receive the executable code. As in the user application attestation case Trent′ does not send the MD5 code for every attestation instance. The trusted authority sends a driver code which populates a data array and provides it to the MD5 code which stays resident on $\mathscr{P}_{kernel}$. To prevent Mallory from exploiting this aspect the trusted authority also provides an arithmetic checksum computation routine which is downloaded for every attestation instance. This provides a degree of extra unpredictability to the results generated by the integrity measurement code.

Fixing call instructions

Kernel modules can be relocated during compile time. This means that Trent′ would not know where the MD5 code got relocated during installation of the module. In order to execute the MD5 code, Trent′ requests the location of MD5 function in the kernel module from the client end. After obtaining the address, Trent′ generates the executable code ($\mathscr{C}_{kernel}$) which has numerous calls to the MD5 code. At generation, the call address may not match the actual function address at the client end. Once $\mathscr{C}_{kernel}$ is generated, the call instructions are identified in the code and the correct target

```
call_target = -( (address_injected_driver +
                   call_locations[0] +
                   length_ofcall
                   )
                   - address_mdstring  );
code_in_file[jump_locations[0] +1 ] = call_target;
```

Figure 7.5: Fixing locations of call instruction

address is patched on the call instruction. Once this patching is done, Trent′ sends the code to the client end. The call address calculation is done as shown in fig. 7.5.

$\mathscr{C}_{kernel}$ is loaded in a char array code_in_file. The location where $\mathscr{C}_{kernel}$ address to be injected is determined by Trent′ by selecting a location from a number of 'nop' locations in the module, this address is termed as address_injected_driver in the above code snippet. The call location in the generated executable code is determined by scanning the code for the presence of the call instruction. The target of the call instruction is a 4 byte value in the x86 architecture. Finally, the address of mdstring (which is the location of MD5 code) is obtained from the client machine as described above. The second statement changes the code array by placing the correct target address. This procedure is repeated for all the call instructions in the generated code. It must be noted that $\mathscr{C}_{kernel}$ calls only the MD5 code and no other function. If obfuscation is required, Trent′ can place some function calls that do not have any bearing on the final result. These calls can be executed by evaluating an 'if statement'. Trent′ can construct several if statements such that they never evaluate to true. It can be noted that even if the client does not communicate the address of the MD5 code, $\mathscr{P}_{kernel}$ can be designed such that the MD5 driver provided by the trusted authority and the MD5 code reside on the same page. This means that the higher 20 bits of the address of the MD5 code and the downloaded code will be the same and only the lower 12 bits would be different. This allows the Trent′ to determine where $\mathscr{C}_{kernel}$ will reside on the client machine and automatically calculate the target address for the

MD5 code. This is possible because the C compiler produces lower 12 bits of function addresses while creating a kernel module and allows the higher 20 bits to be populated during module insertion.

Once the code is injected, Trent$'$ issues a message to the user application requesting the kernel integrity measurements. $\mathscr{P}_{user}$ executes another *ioctl* which causes the $\mathscr{P}_{kernel}$ to execute the injected code. $\mathscr{C}_{kernel}$ reads various memory locations in the kernel and passes the data to the MD5 code. The MD5 code returns the MD5 checksum value to $\mathscr{C}_{kernel}$ which in turn returns the value to the *ioctl* handler in the $\mathscr{P}_{kernel}$. $\mathscr{P}_{kernel}$ then passes the MD5 and arithmetic checksum computations back to $\mathscr{P}_{user}$ which forwards the results to the Trent$'$.

Disabling interrupts

If required, the disable interrupt instruction (CLI) can be issued by $\mathscr{C}_{kernel}$ to prevent any other process from obtaining hold of the processor. It must be noted that in multi processor systems disable interrupt instruction may not prevent a second processor from smashing kernel integrity measurement values. However, as the test cases are different for every attestation instance, Mallory cannot use any prior knowledge to smash the integrity measurement values.

7.2 Results

Table 7.1: Execution times for various components

Time (ms)	Pentium 4	Core 2 Quad
Fixing call instructions of $\mathscr{C}_{kernel}$	0.45	0.2
Execution of $\mathscr{C}_{kernel}$	175	54.3
Network delay	21	15

Table 7.1 provides cumulative results for various operations on an Intel Pentium 4 with 1 GB of RAM and an Intel Core 2 Quad machine that had 3 GB of RAM. As expected, the Pentium 4 machine has slightly lower performance than a

platform with 4 Intel Core 2 processors. The kernel attestation scheme takes longer to execute than the times required for application attestation in chapter 5. This is because the size of the application is small compared to the size of the OS text section, system call tables, and the IDT. The network delay shown in the table is for each send/receive operation occurring between the client and server machines. Hence if the two machines perform 10 sends/receive, the network delay value will be greater than other components. The times shown do not present the time required to generate the challenge. This is because the generation involves issuing a ‘make’ command which takes variable time. Code generation can be seen as a major overhead for the server for each attestation due to issuing a make command. To alleviate the load on the server code generation can occur for a number of test cases beforehand, and stored in persistent medium. During attestation any one of them can be used randomly.

Chapter 8

ATTESTATION OF A GUEST OS FROM A HOST OS

This chapter presents a scheme to obtain the integrity measurement of an client OS by utilizing an external trusted server Trent and virtualization on the client machine. The client OS in question is a guest OS. The guest OS executes on top of a Host OS which communications to Trent and obtains the integrity measurements on the guest OS. The virtualization scheme used in this work is the Linux KVM interface. KVM is used in combination with the qemu software to provide virtualization. KVM provides many OS level calls through the ioctl interface. The integrity measurements were obtained in this work using existing functions provided by KVM and adding ioctl calls.

A Host OS or a Virtual Machine Monitor (VMM) provides a useful interface to execute multiple OSs on the same physical platform. Each guest OS or a virtual machine (VM) is a standalone operating environment that is independent of other virtual machines. Each VM is dependent on the underlying VMM to interface to the hardware on the platform. Virtualization has become quite popular with the advent of multi core platforms. This allows utilization of hardware resources as most executions do not saturate the CPU, this way the resources on one platform can be shared among multiple users.

Apart from being useful for sharing of resources, virtualization in itself offers some security features. Virtualization is intended to keep each guest environment completely isolated from other guest environments. Virtualization offers memory isolation, code isolation, disk isolation, and separate time chunks on the physical hardware on the platform. In the best case scenario, one OS can be installed on the platform as a Host/VMM followed with the install of an emulation/virtualization environment, and install guest OSs on top of the Host/VMM. Multiple OSs can be installed on the machine; each OS can be limited to doing certain tasks. For example,

watching streaming videos can be limited to one operating system; checking mail can be limited to another operating system. Using bank applications, credit cards, and other financial transactions can be limited to another operating system. This way the use case scenario for each operating system can be limited, which in turn limits possible overflow attacks and phishing that may occur. Multiple virtual machines are relatively quick to start and execute with the advent of multi core platforms, and this entire process can be seen as a light overhead for achieving security. A base snapshot of a particular virtual machine can be stored and the working copy of the operating system can be purged regularly to replace it with the base copy. This way a pristine working copy of the OS is available regularly. This serves to remove any infections that may have occurred over a period of time. However, this last step may be highly cumbersome.

A domain 0 virtual machine (like XEN), which has limited user interaction, and no outside world interaction can also be used. Such VMMs can monitor all other resident virtual machines and alert the user to any changes in the guest environment. This concept is utilized already to build virtual machine monitors like Terra [Garfinkel et al., 2003].

Security features offered by virtualization are heavily dependent on the underlying VMM that allows virtualization to occur. If the VMM layer itself is buggy, then isolation and security features cannot be implemented perfectly [Weblink, m]. Xen is a commonly used VMM layer and is known to run into many thousands of lines of code. Isolating and discovering bugs in such a large volume of code is difficult. Xen consists of a Domain 0 which is a trusted root environment which has access to all other guest VM. Every other guest VM is an unprivileged Domain U. Dom 0 can access all contents of Dom U. As long as Dom 0 stays secure, it can detect any malicious activity in other domains. However, Xen is also known to be vulnerable to buffer overflow, DMA write, and other attacks which concern buggy device drivers.

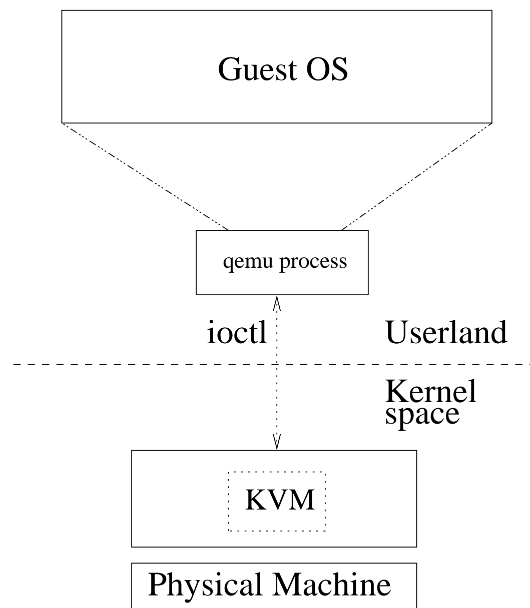

Figure 8.1: Overview of kvm-qemu interface

It was shown in black hat that attackers can get root access to the Dom 0 on a machine, introduce a buggy driver, and overwrite portions of Xen code using DMA [Wojtczuk, 2008b]. However, the crucial assumption is that attackers can get root access to the account. If the system administrator places stringent security measures such that attackers find it difficult to get root access to Dom 0, then this attack will be difficult to execute. Nevertheless virtualization still offers an important security benefit, which is strong isolation of execution environments [McDermott, 2007].

KVM utilizes the hardware assisted virtualization features present in the x86 architecture to offer a light weight virtualization on computing platforms. KVM is installed as a kernel module and the emulation/virtualization of the guest OSs is performed by a software called *qemu*. The qemu software is launched as a process, and the guest OS is loaded inside the process. It is assumed that the guest OS cannot escape the execution environment and any malware that may have infected the guest cannot infect the Host OS.

Figure 8.1 depicts the overall kvm-qemu interface. The Guest OS is loaded

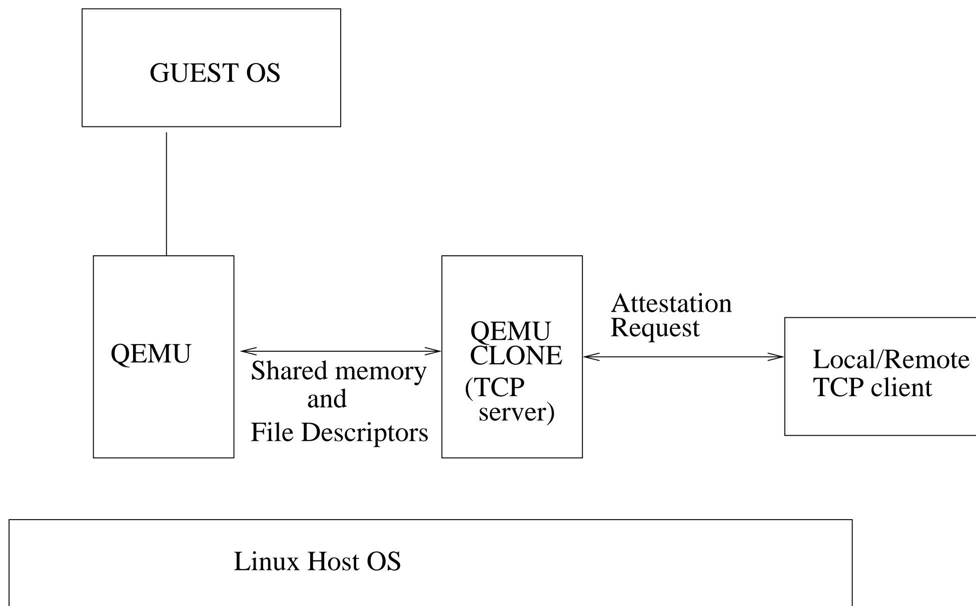

Figure 8.2: Overview of qemu clone operation

entirely inside the qemu software process. All of guest physical memory is actually virtual memory of the qemu process. The qemu software communicates with kvm module using ioctl. kvm provides the required accelerators for qemu.

To measure the integrity of a guest OS, the qemu software was modified to launch a ‘clone’ along with the initial process execution. Figure 8.2 depicts the procedure. The qemu-clone is a TCP server that shares the memory of the qemu process. The clone waits for a signal from Trent (TCP client). The signal includes the sections of the guest memory that Trent needs to verify. On receiving the signal, the clone executes an ioctl which transfers execution to the kernel module. The kernel module obtains the memory areas requested and reads the contents back to the qemu-clone process. The qemu-clone process takes an MD5 on the memory contents and returns the MD5 values to Trent. If Trent finds correct MD5 values, then the remote attestation is completed.

8.1 Implementation

This work was implemented on a 32 bit Ubuntu 10.04 OS executing the linux 2.6.32.28 kernel. The qemu software version used was 0.13.0. The Host OS executed on an Intel Core 2 quad machine with 3 GB RAM. Since KVM utilizes hardware assisted virtualization features, this system could not be implemented on legacy machines.

Starting a clone

The clone system call [Weblink, a] creates a new process just like the fork call [Weblink, b]. However, clone also allows the two processes to share context such as file descriptors, global ariables. It essentially implements threads that share concurrent memory space. The child process requires a stack which is allocated on the parent's heap region. Certain flags determine which memory contents are shared between the two processes. The most pertinent flags to this implementation are the CLONE_VM and the CLONE_FILES flags. CLONE_VM allows the calling process and the child processes to run in the same memory space. Memory writes performed by the calling process or by the child process are also visible in the other process. CLONE_FILES allows the two processes to share file descriptors. A clone is launched by executing the code below.

```
clone(child_function, child_stack + CHILD_STACK_SIZE,
                             CLONE_VM | CLONE_FILES, NULL);
```

The parameters are explained as follows - child_function is the function to be executed as a clone. The child_stack is allocated using a malloc call prior to executing the clone call with a size of 0x4000. In the Intel architecture, the stack moves down and heap grows up, hence CHILD_STACK_SIZE (of value 0x4000) is used to move

the stack pointer up for the clone process. The next argument allows the clone to use the main process' files and memory.

Starting a TCP server inside clone

The clone created inside the qemu process was used to implement a TCP server. The TCP server waits for an incoming connection on port 2000. Once a TCP client makes an incoming connection the server receives various parameters and executes the ioctl described in section 8.1. Since the clone stack was allocated as 0x4000 bytes, data buffer variables were allocated on the heap to avoid the possibility of exceeding the allocated stack.

The clone creates the server executing the following code

```
listenSocket = socket(AF_INET, SOCK_STREAM, 0)
serverAddress.sin_family = AF_INET
serverAddress.sin_addr.s_addr = htonl (INADDR_ANY)
serverAddress.sin_port = htons (listenPort)
bind (listenSocket, (struct sockaddr *) &serverAddress,
                                 sizeof (serverAddress)
listen (listenSocket, 5);
connectSocket = accept(listenSocket,
                       (struct sockaddr *) &clientAddress,
                                   &clientAddressLength)
```

The last line of code shown in the snippet is kept inside a while loop. This is done so that once an attestation instance is completed, the server breaks the client connection and waits for a new connection to be made. Four pairs of send and receive are required for each attestation instance. The first pair does a 'hello handshake'. The

second pair receives the guest physical address from which the measurements have to be taken. The third pair receives the number of bytes starting from the provided address that need to be measured, the last pair sends the MD5 on the requested memory region back to the TCP client.

Reading memory contents of the guest OS

The KVM module provides a function '*kvm_read_guest*' which provides direct access into the physical memory of the executing guest OS. The parameters it requires are the file descriptor for the guest OS, the guest physical address, length of data to be read and a char pointer to read the values into. The file descriptor is automatically filled when the execution enters the kernel module through the ioctl interface of '/dev/kvm'. It determines the guest frame where the memory is located and calls kvm_read_guest_page which does the page table walk using 'gfn_to_hva' to find what is the Host virtual address that corresponds to the guest physical address. Once the host virtual address is found, the memory contents are copied into the destination pointer provided.

The 'kvm_read_guest' functionality is used to implement a new ioctl 'KVM_GUEST_INTEGRITY'. The ioctl populates the provided parameters of guest address, length, and memory pointer into an instance of 'struct kvm_userspace_memory_region'.

Fresh memory is allocated in the kernel to copy the guest OS data. The user space pointer is not directly used to avoid errors which may occur in case the '*kvm_read_guest*' call does not return correctly. The parameters passed by the userland are then used to call kvm_read_guest function. Once '*kvm_read_guest*' returns with a valid value, the memory contents are copied to the user space pointer using 'copy_to_user'. After a successful copy the kernel memory allocated for executing the ioctl is freed by executing *kfree*.

Table 8.1: Execution times for components of kvm-qemu setup

Time (micro seconds)	Core 2 Quad
Execution of ioctl	0.5
Execution of MD5 inside qemu for 100 bytes of data	.5
Network Round Trip on same machine	30
Network Round Trip on Gigabit Ethernet	150
Network Round Trip through fast ethernet switch	260

Table 8.1 provides the results for operations performed during the attestation of a guest OS from the Host OS using the kvm interface. The time required to execute the *ioctl* which extracts the memory from the guest space and delivers it to the qemu clone was found to be less than a microsecond. Similarly taking an MD5 on the requested memory was found to be less than a microsecond. The data is not large, most requests were kept down to the order of 100 bytes, hence the small turnaround time for the attestation.

The network round trip time for one ‘send and receive pair’ was found to be an average of 30 microseconds over 20 tries while using the 127.0.0.1 interface. As seen in section 8.1 there were 4 pairs or send and receive required for each attestation instance. Hence a total of approximately 120 microseconds would be required for one attestation instance if the user initiates the attestation request from the Host OS or on the same physical machine.

Network round trip time was measured for a client and server process executing on two machines which were connected through Gigabit ethernet ports. The time required for a send and receive pair was found to be averaged as 150 micro seconds over 20 tries.

Network round trip time was measured for a client and server process when one machine was connected through a fast ethernet switch while the other machine was connected to a Gigabit port. The time required for a send and receive pair was

found to be averaged as 260 micro seconds over 20 tries. This represents a remote attestation scenario when the user typically has a slower network connection compared to a trusted server Trent who would be connected to a Gigabit speed network.

Chapter 9

BUILDING A SECURE MINIMAL TRUSTED CODE BLOCK VMM

This chapter presents a hardware assisted VMM called MIvmm for the x86 computing platform implemented under 4000 lines of code (LOC) that can be used to build secure applications. Attackers may patch operating system routines and system utilities to hide network sessions, processes, and open ports. Due to the reasons described above malicious logic can have as much power as the OS itself. Determining the integrity of a platform requires that we use systems that are secure enough to provide an indication of an attack to the user. It is known to be difficult to build a secure operating system [Tanenbaum et al., 2006]; hence it is difficult to build a secure root of trust while using a commercial OS as its base.

The use of hardware or a VMM based root of trust offers a crucial tool to system administrators while determining the integrity of systems. A VMM provides a root of trust and a minimal Trusted Computing Base (TCB) to prevent many escalation based attacks from taking place. However, the robustness of any root of trust mechanism built using a VMM depends on the security of the underlying VMM.

Traditional VMMs are known to be bulky; VMware ESX server is known to run into 200K lines of code [Weblink, o] while the latest version of Xen which utilizes hardware extensions for virtualization has nearly 150K lines of code [Weblink, p]. It is estimated that software modules that are around 2000 LOC have nearly 40 faults while modules with 4000 LOC may have as much as 60 faults [Fenton and Ohlsson, 2002], with this observation it can be assumed that the larger the code base, the higher are chances of vulnerabilities to exist in the module, which in this case is a VMM. It is difficult to perform code audits on such large systems to determine whether they are completely secure or not. Numerous vulnerabilities are known to exist in Xen 3, VMware Workstation 6, and VMware ESX Server 3 [Secunia, a], [Secunia, b],

[Secunia, c], and [Wojtczuk, 2008a]. These vulnerabilities allow attackers to break the VMM sandbox environment and take control of the hypervisor/host OS. Due to this, such bulky VMMs are not desired to provide a minimal trusted computing base.

Intel VT–x [Intel Corporation, 2010] and AMD–V [Advanced Micro Devices, 2010] are recent hardware extensions for the x86 platform that provide virtualization support in the processor. These hardware extensions allow for the creation of a minimal secure TCB which can be utilized to build complex security software stacks. MIvmm implements only the minimum necessary features to support virtualization for a single guest operating system. This allows the entire VMM to be implemented in under 4KLOC. The core of the VMM code comprising the launch of the VMM and handling of VM exits is completed in around 2KLOC. This small code size can allow security audits of the code and formal proofs. It also allows vulnerabilities such as buffer overflow to be minimized and identified easily. To keep in line with the design aspects certain features which are normally part of a VMM were removed from MIvmm. MIvmm does not support multiple VMs. This eliminates the need for device virtualization and scheduling of VMs. MIvmm does not need to handle RESET vector of the CPU. This is due to the fact that in the current implementation of the VMM, interactions with the BIOS are removed to reduce the size of the code base. MIvmm does not virtualize interrupts. A standard VMM that supports multiple VMs will have to implement these features. However, to build a secure code base, these features are currently stripped out of MIvmm.

If used in conjunction with the Intel Trusted Execution Technology (TXT), MIvmm can prevent rootkits like the bluepill [Rutkowska, 2006] from infecting the system. Such rootkits utilize the hardware extensions of the platform, the hypervisor can be tuned to disallow attempt by any program to launch another hypervisor when MIvmm is already executing. In addition, these rootkits execute instructions such as ‘CPUID’, ‘VMXON’, and ‘VMLAUNCH’. These instructions are sensitive

instructions and the processor traps into the Host (hypervisor) for the execution of these instructions.

MIvmm was implemented on the Intel *x*86_64 architecture. To reduce the number of lines of code in the hypervisor, MIvmm is launched after the native OS boots up thereby bypassing real mode emulation of VT. MIvmm is launched by loading specific state values in the Virtual Machine Control Structure (VMCS) and executing a series of instructions. Most of the state values are copied directly from the native OS. The VMCS is also filled with certain conditions known as exit conditions on which the processor traps the execution of the guest operating system and executes the VMM which can determine whether to allow the event, or disallow the event. Once launched, MIvmm performs routine exits from the guest OS which are handled by the host (VMM).

During exit handling the parameters received from the guest operating system are validated. This is achieved by selecting a range of allowed values that the guest registers may contain while executing the sensitive instruction. If the exit contains allowed values, then the instruction is emulated in the VMM and the resulting values are stored back in the guest registers prior to resuming the guest. If the values are determined to be invalid, the guest operating system is resumed at the next instruction. It may be noted that since MIvmm is installed by the OS as a device driver, a compromised OS can modify the VMM during launch. However both Intel and AMD provide TXT and SVM technologies that have the ability to measure the integrity of a VMM prior to loading it. As a result any wrongdoing by the OS can be clearly identified.

9.1 Overview of dynamic launch model

We utilized the Intel VT–x hardware extensions to create a dynamic launch

```
Power on/Machine boot
Load the OS
Insert device driver
Perform compatibility checks
Copy OS state to guest components in the VMCS
Setup host state components in VMCS
Launch VMM, move the OS into guest environment
Handle VMexits, continue executing until VMM is turned off
```

Figure 9.1: Overview of dynamic launch

VMM (MIvmm) that provides a root of trust in the hypervisor layer. Dynamic launch involves allowing the native operating system to boot up completely and then porting it into the guest environment. Fig. 9.1 shows the steps to be followed for dynamic launch of a VMM. Once the native OS boots, a device driver containing all the VMM code is installed on the OS. The device driver can be controlled by a ring 3 application to perform the steps or it can execute all the required steps on its own as part of its module entry depending on the threat model. The device driver checks for machine compatibility for executing instructions that require the presence of Intel VT–x on the platform. The driver copies each of the required guest state components from the native OS into a control area known as the Virtual Machine Control Structure (VMCS), sets up the VMM (host) state area in the VMCS and executes the instructions to launch the VMM. Once launched the VMM executes in the background and executes when the CPU traps certain events on the guest OS.

9.2 Design of System

The design goal of MIvmm was to create a root of trust mechanism on the platform while keeping the lines of code in the implementation as small as possible. To create a secure root of trust it is imperative that the number of bugs in the root be almost negligible. As discussed in section 2.1, the fault rates have a density of 2 –75 every 1000 lines of code in OSs. Due to this a design requirement for MIvmm was that it

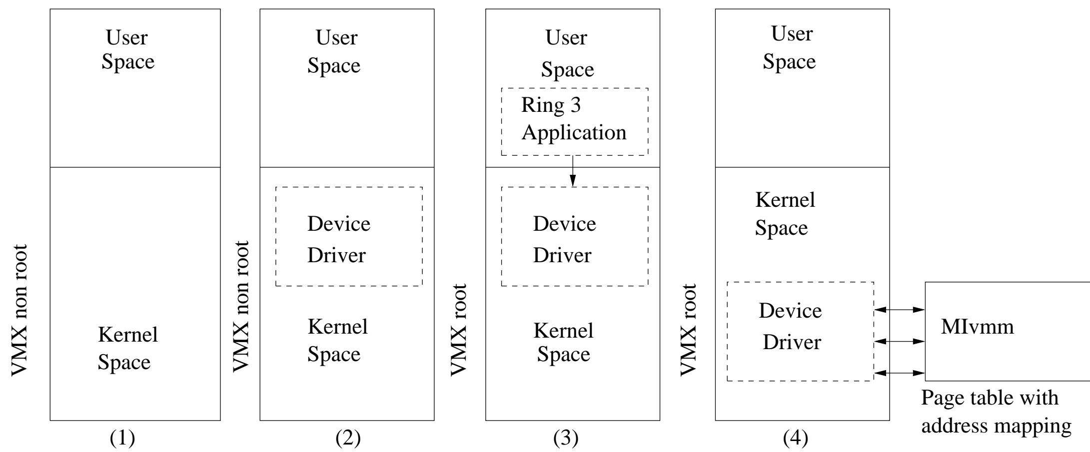

Figure 9.2: System Design

should be implemented in under 10,000 –15,000 lines of code. The small code base also eliminates any possible vulnerability that may creep into the code apart from faults. This also allows removal of unnecessary features from the VMM. Overall a smaller code base provides an efficient and secure solution to the providing a root of trust. MIvmm leverages Intel Virtualization Technology or VT-x to overlay memory protections from the hypervisor onto software running in a VM, hence it serves as a root of trust to an untrusted system.

The VMM acquires control of the hardware on the machine including the CPU and monitors specified events on the system. Only one guest OS was implemented executing on top of the VMM, this enabled removal of features such as device virtualization and memory isolation from the VMM. The dynamic launch model was chosen to allow the native OS to handle all device drivers and bootstrap. These reductions enable creating a VMM with minimal features as specified by the Intel VT–x technology.

Fig. 9.2 provides an overview of the system design of MIvmm. The VMM is loaded after a successful boot of the native OS. The VMM is started as a device driver as seen by the dotted box. A ring 3 application issues a series of commands using the ioctl interface to start the VMM. On a successful execution of the VMLAUNCH

instruction the guest OS completely migrates to the guest mode and a thin layer VMM executes underneath. The exit conditions are stored in the VMCS region, on encountering these conditions the processor loads the host state configuration into memory and executes it. The processor resumes the guest OS after the exit condition is handled, the transfer of execution control to the guest OS is called a VMResume/VMEntry. VMExits and VMResumes occur routinely till the VMM is turned off or the machine is powered down/rebooted.

MIvmm does not survive a system reboot. This model allows the reduction of the code from the VMM. The VMM only needs to implement code for managing its memory resources and storing certain state area. All other code such as user interaction, device management is left to the guest operating system. Although this can be seen as a drawback, a smaller VMM allows bug free implementation of a secure code base. If needed while building other applications, these features can be incorporated in MIvmm. This also mitigates memory leaks, buffer overflow and other attack scenarios.

Once inserted, the device driver performs certain checks, allocates memory for VMXON region and executes VMXON. This puts the operating system in VMX root mode. It then allocates memory for the host (VMM) code, stack and various other required memory regions. This is followed by loading state values into the VMCS and finally executing VMLAUNCH instruction which moves the OS into guest mode. Once the OS is completely moved to the guest mode, it executes on the platform hardware. As discussed before, the execution of certain specified conditions in the VMCS cause the processor to trap the execution of the guest and load the host state (VMM) components for execution.

9.3 Implementation

The VMM was implemented on Linux Fedora 11 64 bit Operating system on the Intel x86_64 architecture on the Intel Core i7 930 processor. The VMM was written in C, inline assembly and assembly, it was compiled using the GNU Compiler Collection (gcc). The implementation is logically separated into performing initial checks, allocating required memory, loading values into VMCS, launching VMM, and continued execution of VMM. Preliminary operations involve installing the driver, allocating required memory, and starting VMX operations. Loading the values in VMCS involves reading nearly 100 state values from the guest register and storing them in the VMCS. Launch of VMM involves porting the native OS into the guest mode and executing the VMM in the background. This occurs if the VMLAUNCH instruction executes without errors. Continued execution involves handling and returning from VM-exits.

The VMM is written as part of a device driver. After the native OS boots up completely, the device driver is installed using an *insmod* command. Once installed the driver stays dormant till a Ring 3 application issues a series of ioctl commands to launch the VMM. Preliminary checks are performed by the driver on receiving one ioctl command from the Ring 3 application. The second ioctl command allocates the required memory for the VMM. The third ioctl command starts VMX operations, loads values into VMCS and launches the VMM. It must be noted that the control does not return to the Ring 3 application from the time VMXON is executed till a successful VMLAUNCH occurs.

Initial Processor Checks

The driver executes CPUID with RAX =1, the resulting value in bit 5 of the RCX register determines whether the processor supports VMX operations. The driver

checks the state of the IA32_Feature_control MSR. This is a safety feature in the architecture which prevents malicious programs from entering VMX operations. Bit 0 and 2 of this MSR must be set to start VMX operations in normal operating mode. This MSR can be changed only through the BIOS and not from the operating system. VMX operations are enabled by setting the CR4.VMXE (bit 13), CR0.PE (bit 1), CR0.NE (bit 5), and CR0.PG (bit 31). The driver then finds the size of the VMXON, VMCS regions and the Processor Revision ID by reading the IA32_VMX_BASIC MSR. The first 32 bits in the MSR contain the revision identifier and bits 32-44 contain the size of the VMXON region and the VMCS region. This size is reported as a value between 0 and 4096. On the machine used to develop the VMM this register reported the size as 1024. However there is an architectural restriction that both these memory regions must be aligned on a 4K address (lower 12 bits of the address must be 0).

Allocating memory for the VMM components

The VMM requires memory for VMXON region, VMCS region, host stack region, and exit conditions for MSR bitmaps. Each of the above VMM regions needs to be in physically contiguous memory; hence the device driver uses kmalloc with the GFP_KERNEL option instead of the vmalloc call to allocate N KB of memory for each region.

For VMXON and VMCS regions it was difficult to allocate a 1K region as specified by the IA32_VMX_BASIC MSR using kmalloc and obtain memory which was address aligned on the 4K boundary. Due to this, an allocation of 4K memory region was chosen for both VMXON and VMCS regions. On both memory regions (VMXON and VMCS) the processor revision ID read from the IA32_VMX_BASIC MSR. 4 pages (4 * 4096 bytes) of memory were allocated for the host stack. The page with the higher address was reserved for the guest state registers. The host stack can

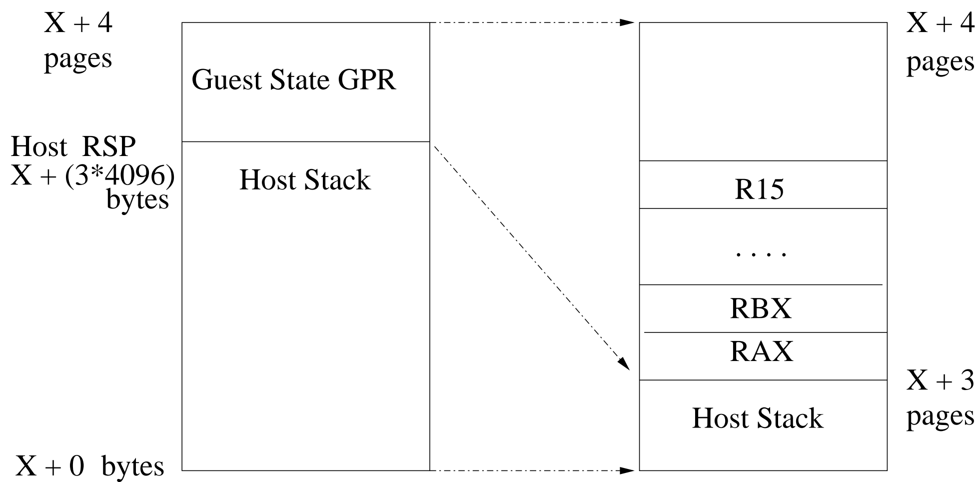

Figure 9.3: Structure of allocated stack area for host

be seen in Fig. 9.3 On a VM-exit, the host entry routine (written in assembly to remove compiler additives) saves each of the 15 GPRs in the system on the highest page using MOV instructions. This way the host stack pointer remains unchanged, and the Host RSP can be used as a frame pointer to the guest state registers while handling the exit. The driver also allocates memory bitmap vector for each possible MSR read and write. This is done as MSR reads and writes cause a VM-exit. A bitmap vector can be created in the allocated memory which indicates to the processor to cause a VM-exit on the specified reads and writes in the vector.

Loading State values into VMCS

Once the processor revision ID is written on the VMXON region, we execute VMXON instruction. This puts the processor state in VMX root mode and allows us to write values into the VMCS using the VMWRITE instruction. After this the processor revision ID is written into the VMCS region and VMPTRLD is executed with the physical address of the VMCS region. This makes the VMCS region as the current VMCS region in the processor state.

After the VMCS pointer is loaded as the current VMCS, the guest state values are loaded into the VMCS by executing a series of VMWRITEs. These values are:

1. Read the values of CS, SS, DS, ES, FS, GS, LDTR, TR selectors, GDTR, IDTR, GS, FS base, control registers CR0, CR3, and CR4 and wrote these values in the guest state area of the VMCS. We also determine values of CS, SS, DS, ES, FS, GS, LDTR, TR segment limits and access rights. The base values of all the segments other than IDTR are determined from the global descriptor table. The location of the GDT is found by executing the SGDT instruction. The base address of the interrupt descriptor table is determined by executing the SIDT instruction. The segment limit and access rights are determined by reading the appropriate entry in the GDT.

2. Read the native operating system RSP and assigned it in the VMCS. Determined the instruction where the guest would 'wake-up and assigned it to the guest RIP.

3. Read the values present in the following MSRs and stored them in the guest state VMCS. IA32_DEBUGCTL, IA32_SYSENTER_CS, IA32_EFER, IA32_SYSENTER_ESP, IA32_SYSENTER_EIP, IA32_PAT, IA32_PERF_GLOBAL_CTRL.

4. We also determined some non-register state information for the guest VMCS. These are activity state and VMCS link pointer. The activity state is determined by reading the IA32_VMX_MISC and the link pointer is statically assigned FFFFFFFF_FFFFFFFFH.

5. Read and stored values of control registers CR0, CR3, and CR4 in the host state area of the VMCS. Allocated a 4 page memory area for the host stack and assign the address to host RSP. Created a host entry point function and assigned it to the host RIP.

6. Read the selector fields for CS, SS, DS, ES, FS, GS, and TR segments and stored it in the host state VMCS. Base address fields for FS, GS, TR, GDTR, and IDTR. The base address for FS and GS base are determined from the

respective MSRs. The other base values are determined by reading the selector offset into the GDT just like in the case of the guest state area.

7. Read the values in the following MSRs and stored them in the host state VMCS. IA32_SYSENTER_CS, IA32_EFER, IA32_SYSENTER_ESP, IA32_PAT, IA32_SYSENTER_EIP, IA32_PERF_GLOBAL_CTRL.

8. Determined the values of the following VM-execution control fields and stored them in the VMCS. Pin-Based VM execution controls, Primary and Secondary Processor-Based VM execution controls, MSR-Bitmap address. The Pin-Based VM execution controls are determined by reading the contents of IA32_VMX_PINBASED_CTLS and IA32_VMX_TRUE_PINBASED_CTLS. The Primary Processor-Based VM execution controls are determined by reading the contents of IA32_VMX_PROCBASED_CTLS and IA32_VMX_TRUE_PROCBASED_CTLS. The Secondary Processor-Based VM execution controls are determined by reading the contents of MSR IA32_VMX_PROCBASED_CTLS2. The MSR-Bitmap address is a 4 K memory area. Each MSR is represented by 2 bits; one for read access and another for write access. Each bit represents whether the VM should exit into the host area for the respective access. If the bit is set, then the access causes a VM-exit. We cleared all the bits in our implementation.

9. Determined the values of VM-Exit controls and VM-Entry controls. The exit controls are determined by reading the MSRs IA32_VMX_EXIT_CTLS and IA32_VMX_TRUE_EXIT_CTLS. The entry controls are determined by reading the following two MSRs: IA32_VMX_TRUE_ENTRY_CTLS and IA32_VMX_ENTRY_CTLS.

Launching MIvmm

Loading state values into the VMCS is followed by the execution of VMLAUNCH instruction. If successful the native operating system transitions into the guest mode and the VMM runs underneath. The processor performs sanity checks on the VMCS values. If the VMCS state values are incorrect the processor reports an error in RFLAGS. The corresponding error number is read from a VMCS component VM-Instruction Error. The errors encountered during the implementation of MIvmm were: incorrect VM-execution control fields, incorrect host-state fields, and incorrect guest state fields. The occurrence of the events in order can be explained as the processor first performs a check on the control fields in the VMCS, followed by checks host state fields. If the first two checks cause an error, then the processor aborts the VMX operation and returns to the native OS. If these two fields are successful the processor starts loading the guest state values while performing checks on the guest state fields. If the checks on the guest state fields report any error the processor performs a VM-exit and starts executing the Host.

Continued execution of MIvmm

Once launched, the VMM executes in the background and performs routine VM-exits on the execution of CPUID and other mandatory exits as specified by Intel VT-x. On a VM-exit the host entry routine saves each of the 16 general purpose registers using an instruction of the format: MOV REGISTER, X (% RSP) where X is the offset from the host RSP. The first register is stored at offset 0H; the second register is stored at offset 8H, and so on. Exit handling operations are performed on this stored frame and these values are restored from memory onto registers just prior to VMRESUME using MOV instructions. The guest OS also contains a VMCALL interface. The VMCALL interface allows the guest to voluntarily cede control to the VMM. The guest provides

```
Ring 3 application: 100 LOC
Driver routine including ioctl definitions: 150 LOC
Preliminaries checks: 200 LOC
Launching VMM: 1900 LOC
Exit handling: 50 LOC (C) + 150 LOC (Assembly)
Print routines for debugging: 1000 LOC
```

Figure 9.4: Lines of code of each component in the VMM

information to the host on which VMCALL is requested by passing certain values through the GPRs. The registers can be chosen by the programmer to implement the VMCALL interface.

Lines of Code

MIvmm was implemented in under 4000 lines of C code. It was comprised of the components as shown in Fig. 9.4. As can be seen from the numbers in Fig. 9.4, MIvmm implementation provides a very minimal trusted code base. If the print routines used for debugging are excluded, MIvmm can be quantified as smaller than 4KLOC. This enables elimination of vulnerabilities that occur due to implementing features that are not essential for executing a VMM.

Chapter 10

CONCLUSION

This research presents software based techniques to obtain the integrity of a user application, and OS kernels entirely in software. A trusted external entity provides Alice with generated code that when executed on the client side provides guarantee that the client side application is not compromised. This work also extends remote attestation by verifying whether the binary attested continued executing or was replaced by the attacker. The check involves placing new code in the in-core image of the binary and replacing a function call inside the binary to point to the new code. The execution of the new code provides an attesting server the guarantee that the binary was not replaced. A series of such changes made inside the attested binary reduce the opportunities that an attacker may have to hijack authenticated sessions by tampering other client end software.

This work presented a technique to obtain the integrity measurement of the OS text section, system call table and Interrupt descriptor table. These measurements are important as the remote attestation scheme for the user application requires the assistance of system calls and the interrupt interface to obtain its measurements. This scheme was implemented on Intel x86 architecture using Linux and its performance was measured.

This work presented a virtualization based technique to determine the integrity of a guest OS using Linux-KVM. A VMM based solution is more secure than the previous device driver based solution provided in the dissertation as it is considered difficult for a malware operating in the guest OS to affect the execution of the Host OS.

This research also presented the need to develop a secure thin VMM which can be used to build other security protocols. The VMM was built utilizing Intel VT-x

technology on Linux (Fedora 11) and supports one guest operating system as it is built for providing a trusted code base. The VMM is launched using the dynamic launch model, i.e., after the operating system boots up, and was implemented in under 4KLOC on the C language using GCC.

As future work the Remote Attestation scheme can be implemented with hardware assisted virtualization as every consumer x86 computing platform is currently manufactured with this ability. The native OS can be ported into the guest OS mode as described in the VMM implementation and remote attestation on the user application and the OS kernel can be performed. Once completed, the OS can be brought back to the native state.

REFERENCES

Advanced Micro Devices (2010, June). AMD64 architecture programmers manual volume 2: System programming.

Ames Jr., S., M. Gasser, and R. R. Schell (1983). Security kernel design and implementation: An introduction. *Computer 16*(7), 14–22.

Chou, A., J. Yang, B. Chelf, S. Hallem, and D. Engler (2001). An empirical study of operating systems errors. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pp. 73–88. ACM.

Cohen, F. (1993). Operating system protection through program evolution* 1. *Computers & Security 12*(6), 565–584.

Collberg, C., C. Thomborson, and D. Low (1998). Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 184–196. ACM.

Cooper, K., T. Harvey, and T. Waterman (2002). Building a control-flow graph from scheduled assembly code. Technical report, Dept. of Computer Science, Rice University.

Fenton, N. and N. Ohlsson (2002). Quantitative analysis of faults and failures in a complex software system. *Software Engineering, IEEE Transactions on 26*(8), 797–814.

Garay, J. and L. Huelsbergen (2006). Software integrity protection using timed executable agents. In *Proceedings of the 2006 ACM Symposium on Information, computer and communications security*, pp. 189–200. ACM New York, NY, USA.

Garfinkel, T., B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh (2003). Terra: A virtual machine-based platform for trusted computing. *ACM SIGOPS Operating Systems Review 37*(5), 206.

Goldman, K., R. Perez, and R. Sailer (2006). Linking remote attestation to secure tunnel endpoints. In *Proceedings of the first ACM workshop on Scalable trusted computing*, pp. 24. ACM.

Intel Corporation (2010, March). Intel 64 and IA-32 Architectures Software Developers Manual Volume 3B: System Programming Guide. Technical report.

Iyer, V., A. Kanitkar, P. Dasgupta, and R. Srinivasan (2010). Preventing overflow attacks by memory randomization. In *Proceedings of the 21st IEEE International Symposium on Software Reliability Engineering*, pp. 339–347. IEEE.

Karger, P., M. Zurko, D. W. Bonin, A. Mason, and C. Kahn (1991). A retrospective on the VAX VMM security kernel. *IEEE Trans. Software Eng. 17*(11), 1147–1165.

Kennell, R. and L. Jamieson (2003). Establishing the genuinity of remote computer systems. In *Proceedings of the 12th USENIX Security Symposium*, pp. 295–308.

Kennell, R. and L. Jamieson (2004). An analysis of proposed attacks against genuinity tests. Technical report, CERIAS Technical Report, Purdue University.

Linn, C. and S. Debray (2003). Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*, pp. 290–299. ACM.

McCauley, E. and P. Brongowski (1979). KSOS-The design of a secure operating system. In *afips*, pp. 345. IEEE Computer Society.

McDermott, J. (2007). Xenon: High assurance xen. Retrieved April 20, 2010: `http://www.xen.org/files/xensummit_4/XenSummitSpring07_McDermott.pdf`.

Nguyen, A. M., N. Schear, H. D. Jung, A. Godiyal, S. T. King, and H. D. Nguyen (2009). MAVMM: Lightweight and Purpose Built VMM for Malware Analysis. In *2009 Annual Computer Security Applications Conference*, pp. 441–450. IEEE.

Ostrand, T. and E. Weyuker (2002). The distribution of faults in a large industrial software system. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pp. 64. ACM.

Petroni Jr, N., T. Fraser, J. Molina, and W. Arbaugh (2004). Copilot-a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th conference on USENIX Security Symposium-Volume 13*, pp. 13. USENIX Association.

Rutkowska, J. (2006). Subverting vista kernel for fun and profit. In *Black Hat Briefings*.

Sahita, R., U. Savagaonkar, P. Dewan, and D. Durham (2007). Mitigating the lying-endpoint problem in virtualized network access frameworks. In A. Clemm, L. Granville, and R. Stadler (Eds.), *Managing Virtualization of Networks and Services*, Volume 4785 of *Lecture Notes in Computer Science*, pp. 135–146. Springer Berlin - Heidelberg.

Sailer, R. (2008). I.B.M. research - integrity measurement architecture. Retrieved November 3, 2010: `http://domino.research.ibm.com/comm/research_people.nsf/pages/sailer.ima.html`.

Sailer, R., X. Zhang, T. Jaeger, and L. Van Doorn (2004). Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the 13th USENIX Security Symposium*, pp. 223–238.

Schwarz, B., S. Debray, and G. Andrews (2003). Disassembly of executable code revisited. In *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, pp. 45–54. IEEE.

Secunia. Vulnerability report: Vmware esx server 3.x. Retrieved June 6, 2010: `http://secunia.com/advisories/product/10757/`.

Secunia. Vulnerability report: Vmware workstation 6.x. Retrieved June 6, 2010: `http://secunia.com/advisories/product/14321/`.

Secunia. Vulnerability report: Xen 3.x. Retrieved June 6, 2010: `http://secunia.com/advisories/product/15863`.

Seshadri, A., M. Luk, N. Qu, and A. Perrig (2007). SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pp. 350. ACM.

Seshadri, A., M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla (2005). Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. *ACM SIGOPS Operating Systems Review 39*(5), 1–16.

Seshadri, A., A. Perrig, L. Van Doorn, and P. Khosla (2004). Swatt: Software-based attestation for embedded devices. In *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*, pp. 272–282. IEEE.

Shankar, U., M. Chew, and J. Tygar (2004). Side effects are not sufficient to authenticate software. In *Proceedings of the 13th USENIX Security Symposium*, pp. 89–102.

Shinagawa, T., H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato (2009). BitVisor: a thin hypervisor for enforcing i/o device security. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pp. 121–130. ACM.

Srinivasan, R. and P. Dasgupta (2007). Towards more effective virus detectors. *Communications of the Computer Society of India 31*(5), 21–23.

Stumpf, F., O. Tafreschi, P. Röder, and C. Eckert (2006). A robust integrity reporting protocol for remote attestation. In *Second Workshop on Advances in Trusted Computing (WATC06 Fall)*. Citeseer.

Tanenbaum, A., J. Herder, and H. Bos (2006). Can we make operating systems reliable and secure? *Computers 39*(5), 44–51.

Wang, L. and P. Dasgupta (2008). Coprocessor-based hierarchical trust management for software integrity and digital identity protection. *Journal of Computer Security 16*(3), 311–339.

Weblink. clone(2) - linux system call man page. Retrieved February 15, 2011:`http://linux.die.net/man/2/clone`.

Weblink. fork(2) - linux system call man page. Retrieved February 15, 2011:http://linux.die.net/man/2/fork.

Weblink. getpid(2) - linux system call man page. Retrieved June 6, 2010: http://linux.die.net/man/2/getpid.

Weblink. Global ip network latency. Retrieved on January 17, 2010: http://ipnetwork.bgtmo.ip.att.net/pws/network_delay.html.

Weblink. Hackers discover hd dvd and blu-ray processing key - all hd titles now exposed. Retrieved on November 3, 2009: http://www.engadget.com/2007/02/13/hackers-discover-hd-dvd-and-blu-ray-processing-key-all-hd-t/.

Weblink. Hi-def dvd security is bypassed. Retrieved on November 3, 2009: http://news.bbc.co.uk/2/hi/technology/6301301.stm.

Weblink. ioctl(2) - linux system call man page. Retrieved June 6, 2010: http://www.manpagez.com/man/2/ioctl/.

Weblink. Kvm: Kernel-based virtualization driver. Retrieved June 6, 2010: http://www.linuxinsight.com/files/kvm_whitepaper.pdf.

Weblink. Linux/unix command - *socketcall*. Retrieved June 6, 2010: http://linux.about.com/library/cmd/blcmdl2_socketcall.htm.

Weblink. Lxr the linux cross reference, code for kvm. Retrieved October 6, 2010: http://lxr.linux.no/#linux+v2.6.33/arch/x86/kvm/vmx.c.

Weblink. mprotect(2) - linux system call man page. Retrieved June 6, 2010: http://linux.die.net/man/2/mprotect.

Weblink. socket(2) - linux system call man page. Retrieved June 6, 2010: http://linux.die.net/man/2/socket.

Weblink. Virtualization security. Retrieved April 21, 2010: http://kerneltrap.org/OpenBSD/Virtualization_Security.

Weblink. Vlc media player source code ftp repository. Retrieved on February 24 2010: http://download.videolan.org/pub/videolan/vlc/.

Weblink. Vmware esx server virtual infrastructure node evaluators guide. Retrieved June 6, 2010: http://www.vmware.com/pdf/esx_vin_eval.pdf.

Weblink. Xen documentation. Retrieved June 6, 2010: http://www.xen.org/products/xenhyp.html.

Wika, K. G. and J. Knight (1994). A safety kernel architecture, Technical Report No.CS-94-04. Technical report, Department of Computer Science, University of Virginia.

Wojtczuk, R. (2008a). Adventures with a certain xen vulnerability (in the pvfb backend).

Wojtczuk, R. (2008b). Subverting the Xen hypervisor. *BlackHat USA*.

Wurster, G., P. van Oorschot, and A. Somayaji (2005). A generic attack on checksumming-based software tamper resistance. In *Security and Privacy, 2005 IEEE Symposium on*, pp. 127–138. IEEE.

Zovi, D. (2006). Hardware virtualization-based rootkits. In *Black Hat USA*.

BIOGRAPHICAL SKETCH

Raghunathan Srinivasan was born in Patna, India. He received his schooling through ICSE (X^{th} grade) and CBSE (XIIth grade). He moved after XIIth grade to greener pastures of Chennai for receiving a Bachelor of Engineering Degree in Computer Science from Anna University, India. He joined Arizona State University, USA on the insistence of his sister to pursue a Masters Degree in Computer Science instead of starting work in the software industry. Not satisfied with obtaining only a MS in 2007, he continued on after finishing MS to obtain a PhD in Computer Science at ASU under Dr. Partha Dasgupta. During his PhD he gained useful expertise in operating systems, kernel programming, computer architecture, placing relocatable executable code in running processes, virtualization, and device driver programming.

Raghu has worked as an Intern at Intel Corporation in Oregon in 2009 and in Arizona in 2010. The internships helped him to gain useful tricks into kernel programming. Raghu has also briefly worked as an unpaid intern at Reliance Infocomm in Chennai, India. Raghu will be joining Intel Corporation at Chandler, Arizona after his marriage to Jessica.