INTEGRITY VERIFICATION OF APPLICATIONS ON RADIUM ARCHITECTURE

Mohan Krishna Tarigopula

Thesis Prepared for the Degree of

MASTER OF SCIENCE

UNIVERSITY OF NORTH TEXAS

August 2015

APPROVED:

Bill Buckles, Major Professor
Hassan Takabi, Committee Member
Mark Thompson, Committee Member
Barrett Bryant, Chair of the Department of Computer Science and Engineering
Costas Tsatsoulis, Dean of the College of Engineering and Interim Dean of Toulouse Graduate School

Tarigopula, Mohan Krishna. *Integrity Verification of Applications on RADIUM Architecture.* Master of Science (Computer Science), August 2015, 44 pp., 4 tables, 7 figures, bibliography, 40 titles.

Trusted computing capability has become ubiquitous these days, and it is being widely deployed into consumer devices as well as enterprise platforms. As the number of threats is increasing at an exponential rate, it is becoming a daunting task to secure the systems against them. In this context, the software integrity measurement at runtime with the support of trusted platforms can be a better security strategy.

Trusted computing devices like TPM secure the evidence of a breach or an attack. These devices remain tamper proof if the hardware platform is physically secured. This type of trusted security is crucial for forensic analysis in the aftermath of a breach.

The advantages of trusted platforms can be further leveraged if they can be used wisely. RADIUM (Race-free on-demand Integrity Measurement Architecture) is one such architecture, which is built on the strength of TPM. RADIUM provides an asynchronous root of trust to overcome the TOC (Time of check to time of use) condition of DRTM (Dynamic root of trust measurement). Even though the underlying architecture is trusted, attacks can still compromise applications during runtime by exploiting their vulnerabilities.

I propose an application-level integrity measurement solution that fits into RADIUM, to expand the trusted computing capability to the application layer. This is based on the concept of program invariants that can be used to learn the correct behavior of an application. I used Daikon, a tool to obtain dynamic likely invariants, and developed a method of observing these properties at runtime to verify the integrity. The integrity measurement component was implemented as a Python module on top of Volatility, a virtual machine introspection tool. My approach is a first step towards integrity attestation, using hypervisor-based introspection on RADIUM and a proof of concept of application-level measurement capability.

Copyright 2015

by

Mohan Krishna Tarigopula

ACKNOWLEDGMENTS

I would like to thank my family for their endeavor in supporting my education. I would also like to thank my friends and relatives for their constant encouragement who believed in me as much as I myself do.

I would like to express my gratitude and thanks to Dr. Mahadevan Gomathisankaran for accepting to advise me on my Master's Thesis and course work. His continuous guidance and support has put me on track and so I was able to accomplish this task. The moral encouragement and technical expertise I have received from him kept me pushing through challenging situations.

I would also like to thank my thesis committee Dr. Bill Buckles, Dr. Hassan Takabi and Dr. Mark Thompson for their guidance and comments on my work which helped me in completion of my thesis.

I wish to express my sincere thanks to Dr. Krishna Kavi for his supervision and suggestions on my thesis in the absence of my major professor.

I also take this opportunity to thank my research teammates: Tawfiq Shah, Srujan Kotikela and Patrick Kamongi for sparing their valuable time in helping me with some critical issues.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS iii

LIST OF TABLES vi

LIST OF FIGURES vii

CHAPTER 1 INTRODUCTION 1

1.1. Motivation 1

1.2. Application Behavior 1

1.3. Integrity Measurement 2

1.4. My Contributions 4

1.5. Thesis Organization 4

CHAPTER 2 RUNTIME INTEGRITY VERIFICATION 5

2.1. Application Integrity 5

2.2. Assumptions and Threat Model 8

2.3. Daikon and Application Invariants 9

2.4. Training Phase 11

2.5. Radium Architecture 14

2.5.1. Background 14

2.5.2. ARTM 16

2.6. Runtime Integrity Verification 18

2.7. Integrity Protection and Remote Attestation 21

2.8. Shortcomings and Limitations 21

2.9. Chapter Summary 22

CHAPTER 3 IMPLIMENTATION 24

3.1. RADIUM Setup 24

3.2. Invariant Extraction 25

3.3. Debug Testing with Daikon 27
3.4. Integrity Measurement Component 28
3.5. Security and Efficiency Analysis 29
CHAPTER 4 RELATED WORK 32
4.1. Remote Attestation to Dynamic System Properties: Towards Providing Complete System Integrity Evidence 32
4.2. Automatic Security Debugging Using Program Structural Constraints 33
4.3. DIDUCE 35
4.4. Purify 35
4.5. Copilot 35
4.6. Runtime Detection of Heap Based Overflows 36
4.7. Flicker 36
4.8. TrustVisor 37
4.9. OSLO 37
CHAPTER 5 CONCLUSION AND FUTURE WORK 39
5.1. Conclusion 39
5.2. Future Work 39
REFERENCES 41

LIST OF TABLES

Page

Table 2.1. Daikon Invariants 12

Table 2.2. Instances and Invariant Count for Prozilla and Ghttpd 13

Table 2.3. Function Prologue 18

Table 3.1. Results: Performance and Efficiency 30

LIST OF FIGURES

		Page
Figure 2.1.	Stack Overflow	5
Figure 2.2.	ELF Memory Layout	7
Figure 2.3.	Threat Model	8
Figure 2.4.	SRTM	15
Figure 2.5.	RADIUM Architecture	17
Figure 2.6.	TPM Attestation Protocol	22
Figure 3.1.	Integrity Measurement Prototype	26

CHAPTER 1

INTRODUCTION

1.1. Motivation

The integrity of software plays a key role in minimizing risk from threats of various kinds. Financial and privacy threats are major concerns of individuals as well as organizations. Trusted computing based integrity verification platforms can address these concerns. Trusted platform module (TPM) [16] developed by Trusted Computing Group offer more security than software-only solutions by adding a layer of trust to the platform. In dynamic and static trusted architectures, the core root of trust is necessary, on which trustworthiness of the rest of the components are dependent. TPM has been used as the root of trust in these systems. On these systems a series of trust measurement events are performed, starting from the core root of trust to the application layer.

Typically, trusted platforms only provide an environment for measuring trusted behavior of applications, but they themselves do not do the application attestation[1]. The application layer is more vulnerable to attacks than kernel or hardware layers. A US-CERT report [37] implies almost 80 % of the security incidents are the result of vulnerabilities compromising software integrity. My work primarily deals with application level integrity management.

1.2. Application Behavior

Every application has certain properties that are needed to remain valid at certain program points during runtime, and those properties are considered to be invariants. Identifying these properties can be useful in understanding application behavior. There are many different kinds of tools and methods to determine invariants for applications depending upon the type of application. I used Daikon, [24], [14] an invariant extraction tool, to learn about the correct behavior of an application. Daikon can obtain invariants from applications written in programming languages such as C, C++, Java, Perl, etc. It can also be employed to

[1] Providing evidence about a system or one of its components

debug and verify the integrity of C applications, as these are the majority on a typical Linux platform. A program has to be run through a front end that instruments it and produces a trace file of a list of variables and their values. Daikon takes the trace files as input and runs it through its machine learning like algorithms to produce invariants. A training phase is needed to run applications in various use cases. Various sets of invariants are obtained in this process. Not all of the invariants produced by the Daikon are useful for security debugging. Therefore, the obtained invariants have to be consolidated into a required list depending upon the purpose for which they are being used. Usage of more invariants produces fine-grained results but incurs considerable performance overhead. The second phase is the measurement process of observing the application for these properties at runtime. Any anomaly shall give a hint of a bug or an attack.

The invariants can be broadly classified into two categories: Data invariants [20] and Structural invariants [21]. Data invariants primarily consist of properties of variables or logical relationship among variables that must be satisfied during runtime of the program. Some types are constant invariants, equality (or inequality) invariants, invariants among elements of an array, etc. Structural invariants are properties of an application based on program rules. For example, in a stack frame of a function the return address has to point to the next instruction in a code section of memory, the frame pointer should not change during the function execution, etc. These properties must remain unchanged during the execution of any application.

1.3. Integrity Measurement

Observing structural invariants at runtime is a non-trivial task as the properties are in transient state. Doing this from outside the context of operating system is even more complex. One existing approach is to run the applications inside debuggers or processor emulators like VALGRIND. However, this is intrusive and inefficient. Moreover, this is a host-based approach. If host OS is compromised, the trust of integrity measurements will be invalid. To overcome this problem I propose introduction of canary variables manually, which can be linked to structural constraints. I have instrumented applications with canary

variables and obtained data invariants for their values. The canary (an analogy to canaries used in coal mines) variables are placed inside every function such that any inconsistencies in the invariant nature of structural constraints will be reflected in data invariants of the canary variables. In the training phase, I extracted invariants for a sample application in normal operation and while it is being exploited. It is evident that the invariant values for canary is not same in both cases. I would like call my method as Runtime Integrity Verification on RADIUM or RIVeR, in short.

RADIUM [22] consists of a modified XEN as a bare metal hypervisor on top of physical hardware with a TPM chip. It has a measuring service Virtual Machine (VM). Upon the request from an external entity, to determine the trusted status of a VM (termed as measured VM or target VM) or specific application inside a VM, the measuring service performs integrity check or rootkit detection. Rootkit detection mechanism was demonstrated in [22]. RADIUM architecture makes sure that measuring service is in a trusted state and its communication to TPM and from TPM to outside challenger is secure. The integrity measurement component will be initiated by the measuring service and will perform a runtime check of an application. The process is explained in detail in the following chapters. The integrity measurement component is given the invariant data of the application collected during the training phase on a clean machine. The integrity measurement component compares the runtime values against their trusted invariant values and decides whether the application is compromised or not.

Getting access to virtual machine memory is needed for integrity monitoring. I used LibVMI [23], a memory introspection library for XEN to accomplish this task. I also used Volatility [39], a memory forensics suite for further analysis of memory access for violations. The memory image of the application is probed for invariants with the measurement component and compared with the invariant values obtained during the training phase. If the values are different, it indicates that the application has a bug or an attacked has happened. As the integrity verification is initiated and performed on Measuring VM, which is in trusted state, the test results are also trusted. Even if the application and measured VM are attacked

by a malware or rootkit, they cannot hide their presence from memory at runtime. Memory introspection is a commonly used technique to find rootkits hiding from operation system and other security mechanisms that work at both system and user level.

Once the application integrity is verified, the measurement component identifies the trusted state as good or compromised. If it is good, it saves corresponding value to TPM. If the application is compromised, the trusted state becomes "compromised." Further, the constraint that is violated is also saved in TPM along with its state. Once Trusted State is written to TPM, the measurement component communicates the same details to external challenge via the secure protocol.

1.4. My Contributions

(1) Application data invariants to observer structural invariants:

As hypervisor level measurement of structural constraints is not an efficient process, I used data invariants of stack local variables to learn application behavior, specially the stack. These invariants were later used for detection of runtime integrity violations of application stack.

(2) Integrity verification of application at hypervisor level:

While many integrity verification mechanisms work at operating system level, I propose a solution that works at hypervisor level. This makes the integrity verification a secure and trusted operation. I implemented my solution on RADIUM architecture (Srujan, Tawfiq et.al.), and have extended RADIUM's dynamic measurement capabilities to the application level.

1.5. Thesis Organization

The rest of this thesis document is organized into following chapters: Chapter 2 discusses relevant background concepts and presents adversary model, data and structural invariants, program instrumentation, integrity verification and attestation protocol, and limitations. Chapter 3 explains prototype implementation. Chapter 4 is on related work. Conclusion of this work, and future work are discussed in chapter 5.

CHAPTER 2

RUNTIME INTEGRITY VERIFICATION

2.1. Application Integrity

Integrity of an application [10] can be defined as the assurance that an application is not modified or accessed in an unauthorized way. Integrity is closely tied with trustworthiness of an application and any integrity violations lead to the breach of trust. What makes an application untrusted and how do we identify it? The goal of my research is to try to answer these questions. A software application is a piece of software code written in a programming language. Integrity violation of application can be a result of modification of the application code or its objects in a static or dynamic way. Static attacks include an attacker changing the application code residing on secondary storage (e.g. hard drives) in order to gain unauthorized access to data.

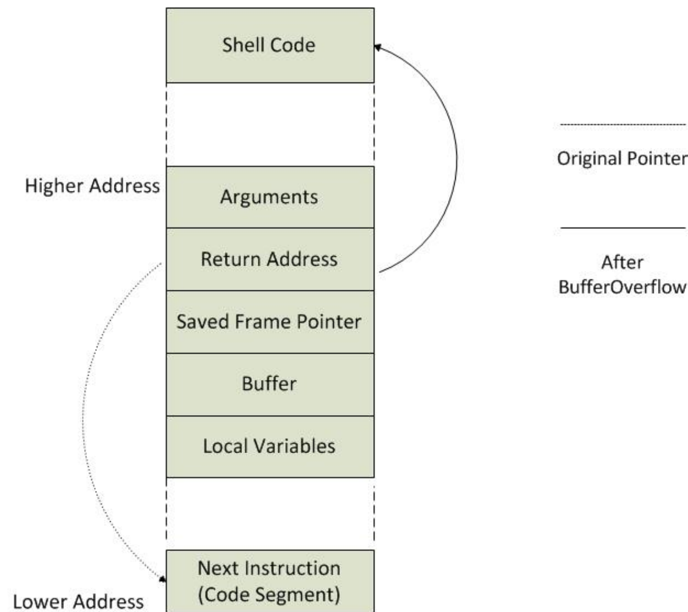

Figure 2.1. Stack Overflow

Modifying memory objects of applications at runtime is a typical example of a dynamic integrity attack. An integrity violation may also lead to the breach of Confidentiality and Availability [10]. Usually, attackers exploit vulnerabilities in applications and perform

runtime attacks to modify the application behavior. This may give privileges to run arbitrary code on a machine or make the system dysfunctional and may lead to access to system resources and user data. Buffer overflow, format string attack, and double free attack are some of the well-known attacks. These memory corruption attacks exploit respective vulnerabilities caused by lack of input data validation. A poorly designed application may use certain library functions, which does not validate input before writing it to memory. Subsequently command or instruction is copied into memory and executed, giving adversary control over application. Return address overwriting is a popular way of executing shell code. Typically, the return address is overwritten with some other value and an arbitrary code is placed in the memory location pointed by the new return value. The result is an application integrity violation. Figure 2.1 shows a buffer overflow attack on the stack. The stack grows from a higher address to lower address. Local variables declared as character arrays are used to write excess content overwriting return address, which lies next to variables on the stack.

Address Space Layout Randomization (ASLR) [35] and StackGuard [13] are some of the existing mechanisms to prevent attacks on memory objects. These have become standard security practices in major Linux distributions like Ubuntu, Red Hat, Debian, etc. In address randomization, various sections of program like stack, heap, executable code, etc. are laid out randomly on memory, making it difficult for an attacker to calculate possible address location for memory objects such as return address or saved frame pointer. If address randomization is not enforced, the stack allocation for an application is usually the next available location in memory and it is easy to compute within few trials. StackGuard places canary word on stack next to return address, to detect overwriting of return address by a buffer overflow exploit. Runtime memory layout for the Linux application's Executable and Linkable Format (ELF) [12] program is shown in figure 2.2.

The code section stores machine level instructions of an application. The stack is for storage of variables of functions, arguments passed, and other corresponding memory objects like return address, saved frame pointer, etc. The heap section is for dynamic memory allocation by "malloc" and other such functions. Exec Shield [28] is an executable space

protection mechanism, which prevents the execution of an arbitrary code place on writable sections of memory such as the stack and heap. This cannot prevent from overwriting the memory, but prevents an executable code written to the stack or heap. There are limitations

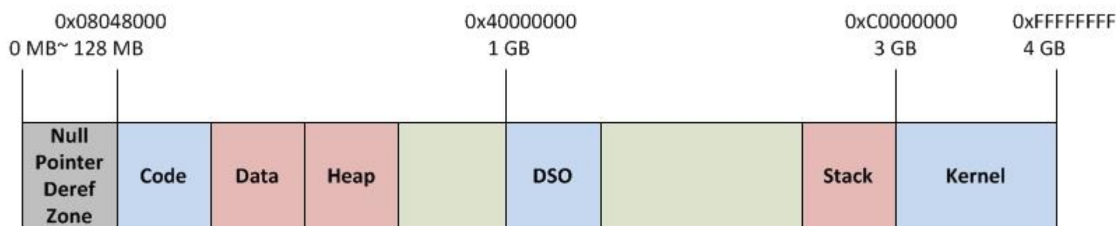

FIGURE 2.2. ELF Memory Layout

to the above-mentioned protective mechanisms. For example, an intelligent attacker can still be successful, although with reduced probability, in calculating the memory address of objects on a stack frame. Practically, it is impossible to secure against all attacks. Some surveys like [40] have shown that above-mentioned protective mechanisms could not prevent all types of attacks. Developers tend to use non-safe functions because of the flexibility they provide. The vast majority of open-source software is developed as a collaborated effort of large groups of programmers. In this context, it is not possible to eliminate bugs in applications even with the alternative safe library functions (for example strncpy in place of strcpy). In addition, many libraries and applications were developed a long time ago, in which the core part of the code is unchanged. New vulnerabilities have been discovered in this software. Recently discovered Heartbleed [5] vulnerability in SSL library is an example. So, securing integrity violation evidence is critical after an attack happens for forensic analysis. Trusted computing is a plausible solution for this software integrity problem.

Trusted platforms depend upon root of trust to achieve trustworthiness. TPM provides this root of Trust for RADIUM at the hardware level. At the time of system boot, the trust is propagated to the application level through a chain of trusted state measurements at BIOS/Firmware level, Kernel level, and application level. This ensures that the integrity measurement process itself is protected and trusted.

2.2. Assumptions and Threat Model

The Virtual machine (VM) on which applications run and use for business activities are called measured VM or target VM, and the VM, which performs attestation at runtime, is measuring VM or measured service. Both the measured (target) VM and the measuring service run on a Xen-based hypervisor and the whole architecture along with hardware is termed as RADIUM. The RADIUM server on which the attestation is performed is assumed equipped with a TPM. At the time of machine startup, Core Root of Trust Measurement (CRTM) is used for trusted boot to ensure that the hypervisor is booted in trusted state. BIOS is where CRTM hardware measurement starts and the chain of measurements propagates from hardware to measuring service. RADIUM uses Asynchronous Root of Trust for dynamic measurements of hypervisor and measuring service each time it receives an attestation request. OS of the target VM can still be vulnerable to runtime attacks. Static integrity of an application is measured before launching it using cryptographic hash functions. Besides using for integrity evidence protection, TPM is also used for the secure attestation protocol for communication between challenger and measuring service. Application Invariants have to be extracted beforehand on clean and verified virtual machine with a configuration identical to that of target VM.

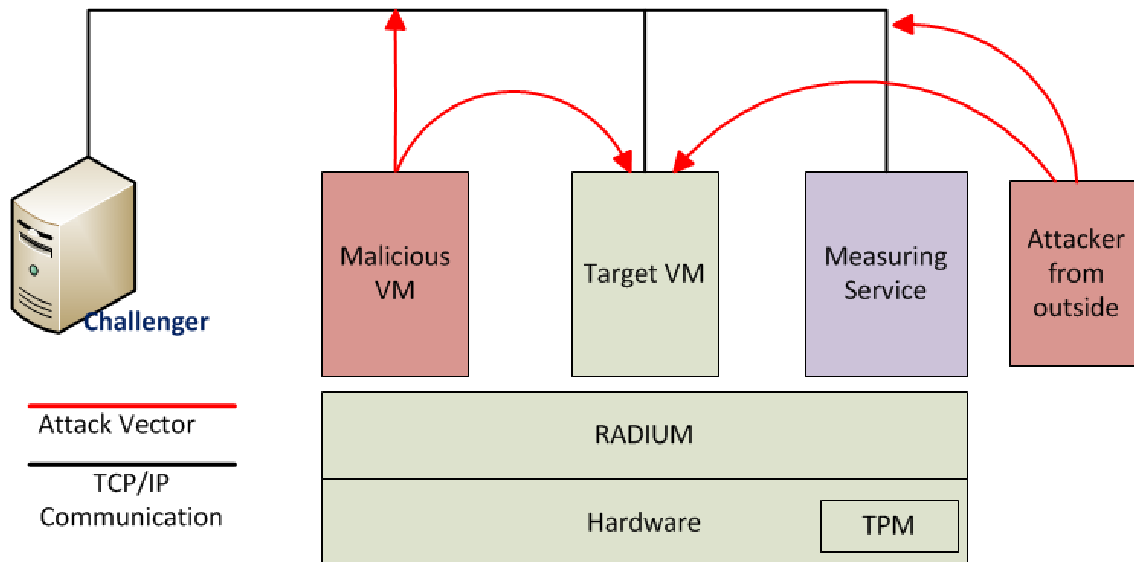

FIGURE 2.3. Threat Model

The adversary is assumed to have no physical access to RADIUM, so the attacker cannot perform any hardware-based attacks. I also assumed that the attacker does not have access to the user level privileges to measured VM but can launch remote attacks on a networked application. With a well-designed buffer-overflow attack on a vulnerable application, the adversary can gain root level privileges on the target VM. Even though the VMs are partitioned and isolated, the adversary may gain access to other VMs and launch network based attacks on target VM. The adversary may try to subvert the communication between a challenger and measured service. The attacker may also perform off-line attack on target VM or application by modifying binaries. Attacks on RADIUM are not in the scope of this research. They are discussed in detail in [22].

2.3. Daikon and Application Invariants

Daikon developers defined invariant as "a property that holds at a certain point or points in a program" [14]. In general, invariant is a condition that has to be satisfied during program execution. The context of execution influences invariant definition at a program point. Based on the source of invariants they are calcified into two types: 1. Data Invariants and 2. Structural Invariants [20]. Data invariants are properties of individual variables or (logical or mathematical) relationships among a group of variables. Examples are constant, original, equality/inequality invariants. Structural invariants are rules that have to be true during execution. The definitions of the rules are generic in nature and they depend upon programming language and execution environment. For example, return address of the stack should always point to a code section of memory and frame pointer of the stack should not change during function execution. Though the invariants reflect program behaviors at specific times, not all of them are documented and even developers may not be aware of their existence.

The invariants are of special interest when it comes to security debugging, vulnerability identification, integrity verification, and understanding program control flow. If one or more invariants of a program are not holding true then these invariants are said to be violated and this indicates an anomaly in application behavior. An attack could be the rea-

son behind this violation. This implies the invariants can be used for integrity verification. Nevertheless, the challenge is to define all the set of invariants the program can produce. In addition, invariant conditions may vary or new invariants may be produced for different execution instances of the same application. Some invariants may not occur for certain executions. Not all invariants of a program can be useful for debugging or integrity verification as many of them are trivial or cannot affect the core nature of the application, and they will be useless. So selecting right invariants is key for debugging or measurement

Besides the dynamic invariants, one can also define static invariants for applications, which should remain unaltered when program is off-line. An example can be the binary and configuration files of an application. These reside on magnetic or solid state disks in a digital format defined by the file system of the drives. The integrity of these files can be a simple measure for static invariant nature of the system. A cryptographic hash value of an executable binary file that was calculated when it is in a known trusted state can hold this static invariant nature. By comparing the current value with trusted value, static integrity can be measured. Static invariant measurements complement dynamic measurements.

There are many tools and methods to extract invariants for applications. For my research, I used Daikon Invariant detector because of its ability to get invariants for applications written in C and C++. It is one of the first tools developed for this purpose. Daikon uses an algorithm similar to machine learning to obtain likely invariants over an execution instance of a program. Daikon needs a front end for C and C++ applications while it does not need any for Java based programs. I used the recommended tool, Kvasir for this purpose. Kvasir is a script which runs on top of a processor emulator called Valgrind [29]. Valgrind runs applications inside its emulated software processor. Valgrind can interrupt a program while it is executing and can read its variables and memory objects. Kvasir uses this feature of Valgrind and produces a trace file with details of variables at function exits and entries. Sample trace values are shown in Table 2.1

The trace values are written to a file with an extension of ".dtrace". These values are fed to Daikon, which then run the variable data in its machine-learning algorithm and

produce actual invariants. The trace values can be directly given to Daikon without the intermediate step of writing it to a file. The dynamic invariants of an application can be classified into two types: Data invariants and Structural invariants [20]. Program data invariants deal with properties and relation between the variables of an application while structural invariants are constraints that have to hold good on meta-data i.e. memory objects of data: stack, heap, etc. Daikon can only extract data invariants while many popular attacks violate structural invariants [21]. Buffer overflow on stack and heap are some of these. In order to obtain meaningful constraints from applications for these meta-properties I used the classical method of instrumenting application at selected program points. I placed canary variables inside functions as local variables such that they can be linked with structural properties. In this research, data invariants of instrumented canary values were used for detection of selected vulnerabilities: buffer overflow and format string vulnerabilities. The data invariants are connected to the return address constraint. A detailed list of structural constraints can be found in [21]. Table 2.2 gives a list of invariants from Daikon corresponding to "dtrace" values of variables. These were extracted from Prozilla, a download accelerator. The "dtrace" definitions are properties of invariants defined by Kvasir. Corresponding variable values at entry and exit of the function message() are given in column two and three, respectively. The last column gives some of the observed invariants of the variables.

Some more invariant relation between variables canary and connections are listed in Chapter 3. The list of invariant definitions given in Daikon manual [14] is not exhaustive and new definitions can be added to the list. Daikon was originally developed for extracting invariants for Java application. It has been extended to get invariants for programs that are written in C, C++, Perl, etc. with the help of various front-ends, which produces trace files. Daikon produces invariants taking these trace files as input data.

2.4. Training Phase

As discussed in the previous section, the invariants are execution specific. Therefore, to obtain all possible invariants, the applications have to be executed in all possible use cases. The training phase involves the process of identifying these use cases and configuration

Table 2.1. Daikon Invariants

Dtrace Definitions	Dtrace (at message() function entry)	Dtrace (at message () function exit)	Daikon Invariant
variable ::canary var-kind variable rep-type int dec-type int	1000	1000	::canary == 1000 (entry) ::canary == 1000 (exit) ::canary == orig(::canary) (exit)
variable ::connections var-kind variable rep-type hashcode dec-type connect*	0x0 1	0x0 1	::connections[].http_sock elements <::canary (exit)
variable ::rt var-kind variable rep-type hashcode dec-type runtime	::rt 0x8064b80 1	::rt 0x8064b80 1	::rt has only one value (entry) ::rt == orig(::rt) (exit)
variable ::rt.num_connections var-kind field num_connections enclosing-var ::rt rep-type int dec-type int	::rt.num_connections 4 1	::rt.num_connections 4 1	::rt.num_connections == 4 (entry),::rt.ftps_mirror_req_n <orig(::canary) (exit)

options and obtaining invariants for them. The more use cases the training phase covers the complete the invariants obtained. I have manually identified frequently used cases and configuration options and used a script to produce invariants. These test cases are vulnerable to attacks at run time.

To obtain trace values, the application needed to be compiled with the dwarf-2 debugging format enabled. Usually, application binaries used in a production environment are not compiled with debugging format enabled as it creates overhead, so access to source code is required to use the application with Daikon. I used Prozilla [36], a tool for downloading files from web and FTP servers and Ghttpd [30], a lightweight web server daemon to obtain

invariants for the applications to do bug detection and integrity analysis, while the applications are exploited with shell code to gain root privileges. These applications have known vulnerabilities [2] [3] [1], which I exploited by using publicly available exploit programs from [8], [7] and [32]. Invariants were collected when the application was running under normal execution and when they were being exploited. I compared the results to confirm attacks on the applications. Various use cases and possible executions corresponding to these use cases

TABLE 2.2. Instances and Invariant Count for Prozilla and Ghttpd

Application	Use case	Instance Count	Vulnerability	No. of Invariants
Prozilla 1.3.7	HTTP file download	6	Stack overflow	18
	FTP file download	4	Format String	8
Ghttpd 1.4	Web Page access	4	Buffer overflow	12

are tabulated in Table 2.2. The use case details are determined from their documentation. I collected invariants for these use cases with some possible configurations and situations. These cases are: limited bandwidth, single thread, different port and interrupted download. Number of invariants obtained for the tested vulnerabilities are also presented in Table 2.2. Daikon can be customized to produce invariants for specific function calls, variables, and types of invariants. A complete list of options can be found in the Daikon user manual [14]. Constant invariant and equality invariant are of interest to me, which were used for detection of the simulated attacks. I also selected vulnerable functions (ex: message() in Prozilla) and produced limited number of invariants, so their analysis and runtime verification would be less tedious and efficient.

Structural constraints are programming rule-based in nature, so data invariants obtained by Daikon are not helpful in detecting attacks that violate structural constraints. The canary values were introduced in functions as local variables and assigned to them with global canary value. The global canary has to satisfy "constant" invariant and local canaries have to satisfy "equality" invariant. I have analyzed the source code by debugging it

and identified the functions, which has known vulnerabilities. The process of placing new statements into code is known as instrumentation. In general, the instrumented statements do not interfere with application behavior or their control flow. It is mainly used to debug the application for logical flaws and for understanding the control flow of the application code. The performance overhead created by the instrumented statements would be negligible. StackGuard instruments applications with canaries at compile time to detect stack corruption at runtime. In this research, I demonstrated a way of using RADIUM's trusted measuring service for runtime integrity verification. I discussed detailed merits and demerits of my work at the end of this chapter.

The training shall be conducted in a trusted environment and the invariant data shall be preserved for later usage at runtime. For this research, I extracted invariants for both the applications on a clean Ubuntu machine with identical configuration of the VM used to do measurements.

2.5. Radium Architecture

2.5.1. Background

TPM is a tamper resistant integrated chip that can be used to extend cryptographic security setup for creating a dynamic trusted measurement architecture. Existing architectures are vulnerable to TOCTOU (Time of check to time of use) and denial of trusted service [22] attacks. To solve these problems, RADIUM architecture uses asynchronous launch of measuring service.

A TPM is mounted on the motherboard along with other chips, and it uses a Low Pin Count (LPC) bus to communicate with the microprocessor. The TPM can store trusted measurement values and cryptographic keys (which it generates) for secure communication with a challenger who requests an attestation. It stores these values in special registers called Platform Configuration Registers (PCRs), which are 160-bits in size. On x86 PC architecture, a TPM has 24 PCRs (0,1,...,23) and specific range of PCRs store measurements related to specific system components: PCRs 0-4 store BIOS state, PCRs 5-7 store boot loaders. PCRs 8-15 are for OS measurements and are called static PCRs. PCRs 17-22 are

dynamic in nature, which store runtime attestation measurements. TPM performs extend, binding, sealing, and quote operations. "extend" is the only write operation PCR performs. The new value is hashed along with existing value and stored. "binding" operation is used to encrypt the measurements with an asymmetric pair of keys (storage key) with the private part of the key permanently locked inside TPM. The sealing operation ties the platform state with the binding operation making the TPM trusted storage. Platform states can be verified with "quote" operation. The quote is used to attest status of the target (VM or application) to an external challenger. It provides the challenger with a signed measurement value stored in the PCRs.

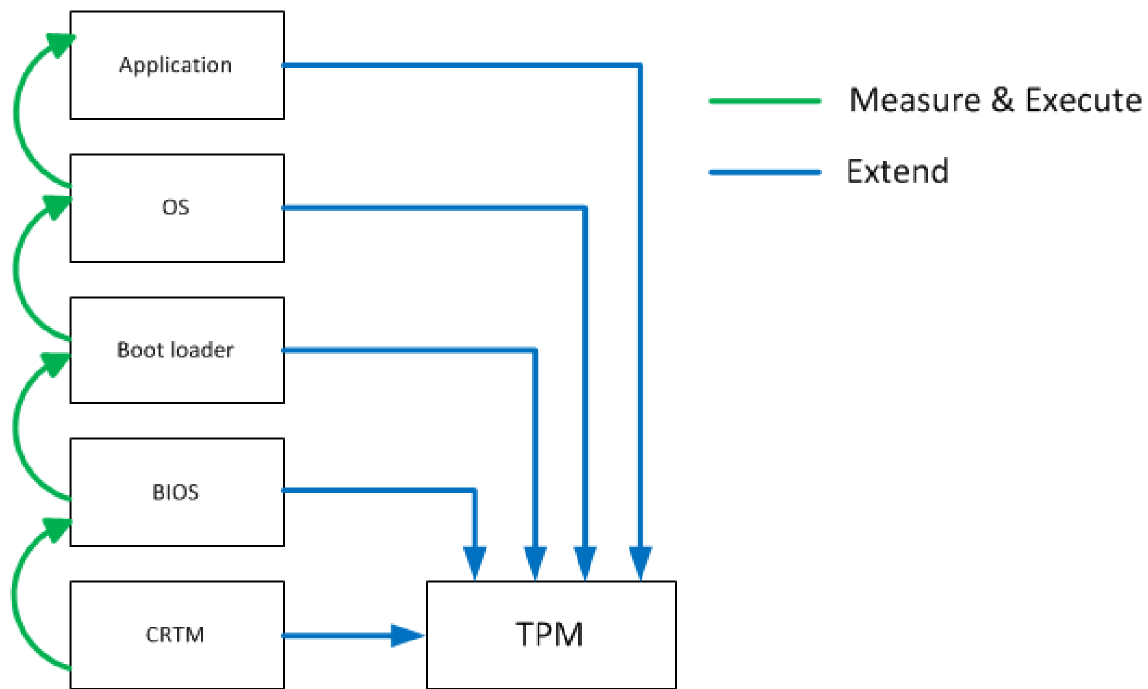

FIGURE 2.4. SRTM

Static Root of Trust Measurement (SRTM) and Dynamic Root of Trust Measurement (DRTM) are existing trusted architecture solutions that use TPM capabilities for integrity measurements. In both the solutions, the TPM serves as hardware CRTM, the starting point of trust. The measurements are done in a chain of events: CRTM-> BIOS-> OS Kernel-> Application. Each of the components represented in this chain measures the next component by computing a hash value and extending it to the TPM's static PCRs. In SRTM's chain,

trusted states can only be computed at boot time. So, every time an application has to be launched in trusted state, the system has to be rebooted to start measurements from CRTM.

In DRTM, a special setup instruction is used to start trust measurement. The setup instruction is the cue for the processor to suspend running processes and start Measured Launch environment (MLE), which is an isolated environment. Optionally, Authenticated Code Module (ACM) can be used as the intermediate measurement component between setup instruction setting and MLE. In that case, CRTM (TPM) measures ACM and ACM measures MLE before it will be started with protected virtualization. As MLE is isolated, it is secured from attacks and trusted. DRTM extends the measurements into the TPM starting with PCR 17. As the name suggests DRTM can be invoked dynamically at runtime along with boot time invocation. The measured boot is similar to SRTM but only more secure with virtualization protection.

AMD Secure Virtual Machine (SVM), Intel Trusted Execution Technology (TXT), Measured Boot, and Trusted hypervisor are some of the technologies that use SRTM and DRTM. SKINIT AND GETSEC are instructions used by AMD SVM and INTEL TXT respectively to launch DRTM environment. AMD SVM does not use any ACM while INTEL TXT uses ACM. Trusted Grub [6], Oslo [19], and tboot [11] are some examples of measure boot service. Terra [15] used cryptography to create isolated VMs, and it is a trusted hypervisor type of solution. Policy-based access control for hypervisors is another kind of trusted platform solution.

2.5.2. ARTM

DRTM, virtualization technology (VT), access control policy, and measuring service(s) are core components of RADIUM. RADIUM uses the on-demand measurement service, a novel method, to handle TOCTOU ATTACKS. Applications or measured VMs can be launched in a trusted environment any time through the Asynchronous Root of Trust for Measurement (ARTM). This is an asynchronous way of using MLE, which contrasts with the DRTM-only launch, in which the hypervisor and MLE have to be launched synchronously. Hypervisor is the root of trust in ARTM. As hypervisor's footprint is much smaller com-

pared to kernel of typical commodity operating system, this root of trust can be considered equivalent to hardware-based CRTM.

Type-1 hypervisor is the one, which runs on top of hardware that forms the trusted base for RADIUM. This is also called bare metal hypervisor. It creates an isolated environment for virtual machines to run on top of it. Hypervisor can run operating systems as whole, modified, or even applications alone. Modified operating systems are employed to achieve better functionality. Access Control Policy, a module of hypervisor, controls the VM and application communication in a secure fashion.

RADIUM can host both trusted and untrusted environments. To prevent TOCTOU attacks a trusted environment is put to use immediately after it is measured and launched. On the other hand, a VM launched without measurement is considered an untrusted environment. Hypervisor uses unique UUIDs to identify environments. The measuring service will be measured and launched whenever a VM or an application is needed to be measured. RADIUM supports multiple measuring services simultaneously. Measuring services may be used to look for the presence of a kernel rootkit or measure run-time integrity of an application. To perform any measuring task, the measuring service will be given necessary permissions through access control policy defined in ACP module in the kernel.

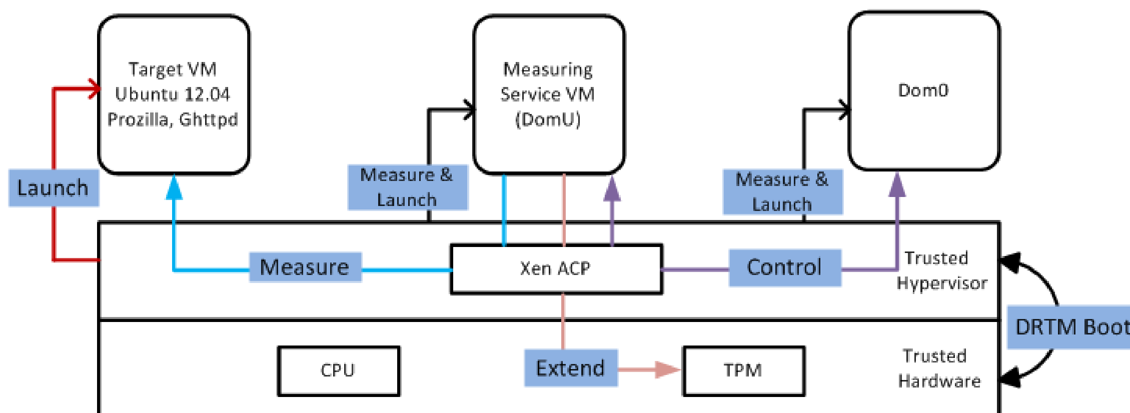

FIGURE 2.5. RADIUM Architecture

2.6. Runtime Integrity Verification

Monitoring applications during runtime is a challenge because, in the transient state the application memory objects undergo continuous change. The observation has to occur virtually in no time. Many times the attacks are also transient and may not last long or may not leave trails after they are done. That is why runtime monitoring has been a very difficult task to perform effectively. In section, 2.1 of this thesis, some traditional tools and methods to detection some runtime attacks were discussed. These are operating system and compiler level mechanisms. StackGuard and ProPolice are two examples. These tools use compile-time instrumentation to detect attacks on the stack. The stack can also be made non-executable [28] so that any malicious code placed on it are prevented from execution thus preventing the attacker from obtaining access to the system. Some other tools instrument applications dynamically at runtime for monitoring the stack.

In the earlier discussion about Daikon, I mentioned that it produces invariants at function entries and exits. Therefore, the runtime analysis of invariants has to occur between the function call and function exit. When a function is called, a few instructions are executed for the function to setup. The instruction allocates memory for the stack and saves (previous) frame pointer to the stack. This is called a function prologue. A sample prologue instruction is given in Table 2.3. The "ebp" and the "esp" refer to CPU registers. They hold the values for base pointer and stack pointer, respectively. Stack is observed once the function prologue

Table 2.3. Function Prologue

Address	main program	Instruction
0x8048360	main+0	push ebp
0x8048362	main+1	mov ebp, esp
0x8048365	main+5	sub esp, 0x00000248
0x8048368	main+8	and esp, 0xfff00000
0x804836a	main+11	mov eax, 0x0
0x804836f	main+16	sub esp, eax

is set up. The subsequent instructions write data to stack, overwrite some objects, delete some of them, and so on and so forth. This process continues until the function is returned. Therefore, the changes made to stack has to be recorded and analyzed for violations. The application execution takes place in the target VM that is being measured. The measurement has to be done from the measuring service VM.

I propose to use Volatility [39], a memory introspection tool to do runtime introspection of the target VM memory for violations. Volatility has many plug-ins written in python to perform various introspection and forensic tasks. Memory introspection tasks can be performed dynamically on a live machine while it is running or statistically on a memory dump that was previously extracted. Originally, Volatility is a forensic tool used to analyze the memory dump. However, my work requires introspection to be performed on a live VM memory from outside the target VM. Volatility needs LibVMI, a virtual machine introspection library that provides access to the target VM memory. Some of the Volatility modules used in this research were: `linux_pslist (to print list of processes), linux_memmap` (to print memory map), `linux_proc_maps` (to print process memory map), and `map_dump` (to extract data from application memory dump. The goal of the experiment was to detect the attack or anomaly as soon as possible, before the transient state of application destroys attack evidence. Extraction of frequent memory snapshots and analyzing them on they fly at the speed of application execution is not possible. The application has to be paused to get the process' memory dump. This helps minimizing the TOCTOU condition resulting in better integrity verification. . There would be some time lapse from the time of attack to the time of detection. This work is based on trusted computing, whose main purpose is to find integrity verification in trusted fashion and does not offer protection against threats.

The introspection library LibVMI uses measuring service to access the target VM with the help of hypercalls. A hypercall is a system call equivalent of hypervisor. Measuring VM makes hypercalls for LibVMI to make memory page snapshots. Hypercalls' access is managed by the Access Control Policy. So, the memory access of target application through measuring service is secure.

I have developed a Python script for parsing the memory snapshots of the application for invariant data. The integrity measurement procedure was tested for its efficiency on Prozilla, Ghttpd, and Nullhttp which have known vulnerabilities, as discussed in section 2.4. As I mentioned earlier, all the functions were instrumented with canary values and the application was compiled. The invariants specific to the injected canary variables were obtained. The data was input to integrity monitoring component before the application was launched. Once I exploited the application with an input with shell code and then the monitoring component detected buffer overflow attack on a string buffer used in function message(). The next section outlines the procedure of sealing evidence in TPM and the attestation procedure. Below is a brief outline of parsing program algorithm:

```
invariantLlist = [input];
while (app running)
{
    currentFuncName = getCurrentFunction();
    entryList, exitLlist = funcList(message);
    //[framePointer, returnAddress, canary]
    if (functionEntry){
        if (stackCurrent == entryList)
            status = true;
        else
            status = false;
    }
    if (functionExit){
        if (stackCurrent == exitList)
            status = true;
        else
            status = false;
```

```
    }
}
TpmSeal ();
AttestChallenger ();
```

2.7. Integrity Protection and Remote Attestation

Once the evidence of integrity is collected by RIVeR, it will be saved to the TPM with an "extend" operation. An encryption key is used to seal data into the TPM. As the hypervisor is trusted, the encryption key generated by it will be immune from attacks. Hypervisor handles all communication with the TPM for measuring service. Measuring service sends the measurement details to hypervisor via callstack API. Hypervisor extends the measurement PCR 23 of the TPM. A challenger may request for attestation any time. The TPM replies with the latest available measurements.

The communication process of attestation has to be protected from network attacks. So, a TPM-assisted secure protocol is used for communication between TPM and challenger. TPM supports authorization protocols like OIAP, OSAP, and DSAP [16]. Challenger requests the TPM for measurements with "quote" operation, which is accessible to challenger through the TCP/IP network. The challenger sends a random nonce along with a "quote" request. Attester replies with nonce, PCR value signed with AIK. Basic structure of the TPM attestation is shown in figure 2.6. The AIKs are generated with an RSA scheme. TPM is known to be secure from various types of attacks like replay attack, masquerading, and tampering.

After receiving the reply to quote (measurements) from the RADIUM, challenger examines the integrity evidence. If the measurements show the target is trusted, challenger may continue accessing the trusted services on RADIUM.

2.8. Shortcomings and Limitations

A part of my project is dependent on Daikon. Therefore, some shortcomings of Daikon were also inherited into my work. The main limitation of Daikon is it lacks the capability of

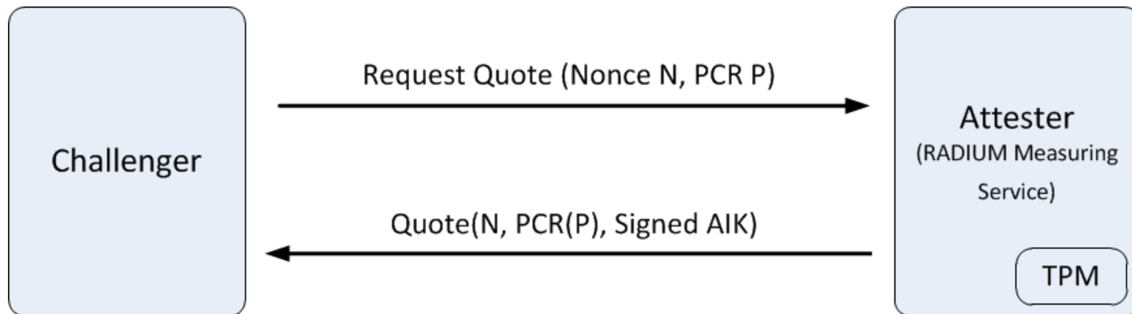

FIGURE 2.6. TPM Attestation Protocol

deducing structural constraints. Manually instrumenting application is not a perfect solution to catch buffer overflow, but using random canaries is closest to the best available solutions. Direct observation of all the structural constraints can be a more efficient solution.

Data invariants for instrumented code does not reflect all types of structural constraints. In addition, this method cannot infer invariants for complex data structures like linked lists. Another limitation is the incomplete set of dynamic properties. If an attack violates invariants that are not predefined, they cannot be detected. There is also the theoretical possibility of false positives caused by incorrect invariant definitions; although, I have not observed this in the attack tests I have conducted. Integrity measurement component's (RIVeR) primary limitation is performance. The application (or target VM) needed to be paused before taking a memory snapshot of its stack frames, which imposes some performance overhead and increases total runtime. As this is runtime measurement, some transient attacks may go unnoticed before they are caught.

2.9. Chapter Summary

In this chapter, I have presented methodologies for defining proper behavior of an application by extracting invariants. For observing structural constraints, I have instrumented the application used for testing. This work is aimed at proving integrity measurement capabilities of RADIUM at the application layer. The integrity measurement tool, RIVeR, is primarily built to fit into RADIUM. I made use of existing introspection technologies like LibVMI and Volatility to do the runtime measurements of application. The integrity evidence thus obtained is handed over to TPM by measuring service and further shared with

challenger from outside using a secure protocol. The scope and limitations of my research are also presented in this chapter.

CHAPTER 3

IMPLIMENTATION

Implementation of my project involves the following steps: 1. Setting up RADIUM with trusted environment. 2. Extracting invariant for application on a clean target VM on RADIUM. 3. Security debugging with the help of the invariants. 4. Building Integrity measurement component and testing its efficacy.

3.1. RADIUM Setup

RADIUM was implemented on Xen, a type-1 bare metal hypervisor. Xen 3.2 kernel is used for this purpose. Both measuring service and target VMs are using Ubuntu 12.04. The measuring service was implemented as DomU [1] , while Dom0 [2] was also using Ubuntu 12.04 OS. The hardware comprised of an Intel I5 processor (4th Generation) with VT-x, VT-d, and TXT enabled and 4096 Megabytes of RAM.

The setup of RADIUM follows a series of steps in order to put various components in place. Xen Security Module (XSM) based Access Control Policy is used by RADIUM. The Xen kernel has to be compiled with customized ACP before installation. Trusted boot, the launch of measuring service, target-VM measurement, and runtime attestation are high-level steps of the measurement process. During Trusted boot, when the physical machine is started, BIOS performs POST, and then control will be handed over to boot loader (tboot) [11] is used in RADIUM), which then invokes GESTSEC [SENTER], a special instruction of DRTM. It launches Intel SINIT ACM and TXT hardware. Next, the hypervisor (Xen) is measured and launched. The DRTM measurements are stored in PCRs 17 and 18 of TPM this command:

```
tpm_sealdata -z -I secret.key -o ./secret.blob -p17 -p18
```

[1]DomU- Unprivileged domain, with no access to hardware

[2]Dom0- Domain 0 or control domain, which is started first and manages DomUs

An encryption key was generated in Dom0, used for "seal" operation, and was sealed by the above command. The key was used for the first and subsequent launches of the measuring service. The unseal operation has to be used to obtain the key from TPM.

```
tpm_unsealdata -I ./secret.blob -o secret.key -z
```

The success of the unseal operation is based on the state of the platform. If the state is untrusted, it is not possible to unseal the key and launch the measuring service. The measurement of target application will be stored in PCR 19 of TPM. As a physical TPM was used in RADIUM, only one measuring service could be launched at a time. A virtual TPM [9] can be used to overcome this limitation, which allows multiple simultaneous measurements using single physical TPM. Figure 3.1 shows prototype implementation of the integrity measurement.

The measuring environment provided by RADIUM has the advantage of better understanding of the measured VM. A variety of measurements can be performed on top of RADIUM. Kernel level measurements were demonstrated on [22] that outlines a methodology for detection of kernel rootkit on target VM. My work is application level measurement, in which I assessed the integrity of application at runtime. It complements the measurement performed in [22]. Before attesting application at runtime, the trusted behavior of the application has to be defined. I used Daikon, an invariant extraction tool to define this trusted nature of applications at normal executions.

3.2. Invariant Extraction

Daikon is developed in Java, so it needs a Java execution environment to create invariants. Kvasir is the front-end tool for C and C++ applications that is used to extract trace values. Kvasir is built as a skin on top of Valgrind, a processor emulator used for debugging. Valgrind dynamically instruments applications at runtime to print all values of variables. Kvasir constructs a trace file with the help of Valgrind output. Valgrind has various modules for various tasks. Kvasir uses the "memcheck" module for profiling of applications.

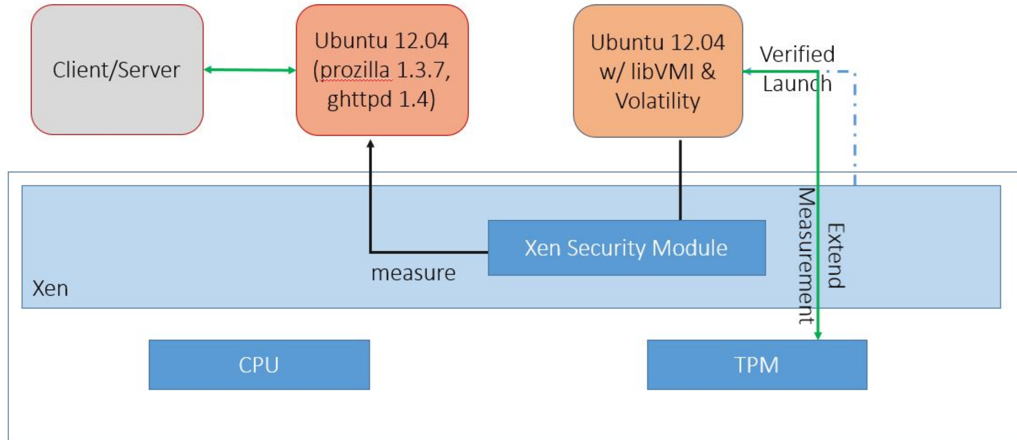

FIGURE 3.1. Integrity Measurement Prototype

For Kvasir and Daikon to work on an application, it has to be compiled with the dwarf-2 debugging format in the GCC (Gnu C Compiler) [4]. GCC version 4.6.3 was used for compiling all the applications. The following command was used to extract trace values:

```
kvasir-dtrace --ppt-list-file=input.ppts \
--dtrace-file=proz.dtrace ./proz localhost:80/index.html
```

The term "proz" in the above command represents the ELF binary of the application Prozilla. The application has to be executed on command line in normal fashion and must be preceded by a "kvasir-dtrace" command along with required command line options. Once trace values are obtained, the next step is to run the trace file through Daikon to get actual invariants.

```
java daikon.Daikon --config_option \
daikon.derive.Derivation.disable_derived_variables=true \
proz.dtrace 2&>1 | tee new.inv
```

Once the above command is executed, Daikon runs the trace values through its machine learning algorithms and writes likely invariants to a file. A set of such invariants were obtained for various use cases (Table 2.2) of Prozilla and Ghttpd.

3.3. Debug Testing with Daikon

texttDaikon can also be used for debugging and vulnerability testing of applications. I tried to find the function with buffer overflow vulnerability on Prozilla using Daikon. The procedures of Prozilla were instrumented with a canary value. I created a global value, called it "canary," and assigned it a random value (1000 in my example). A local variable called "local_can" was also created and assigned the same value as that of global "canary". This introduction of canaries produced direct invariants: equality and original. This is printed as [::canary == orig(::canary)]. This invariant has to hold true at function exit. This invariant occurs in a few functions; message() is one of them, which has the actual vulnerability in library function "fprintf", which doesn't check bounds while writing data. Some invariants were also derived based on the relation of the canary with other variables in the program. There were many more "invariant relationships" among other variables as well, but they are not all in the scope of the vulnerable sections of code. Below is a list of canary invariants, defined in Daikon's output format. Connections[] is a data structure that holds information about the total number of connections made by Prozilla.

```
::connections[].remote\_bytes\_received elements < ::canary
::connections[].hs.contlen elements < ::canary
::connections[].hs.accept\_ranges elements < ::canary
::connections[].hs.statcode elements < ::canary
::connections[].try\_attempts elements < ::canary
```

Exploit code was compiled and run on a server, which acts like a web or FTP server, accepting file download requests from Prozilla. While it tried to download a file from this malicious server, the server exploited the vulnerability on message() procedure, overwrote the return and other elements on function stack, wrote a shell code stack, executed it, and gained root to shell. Prozilla had been run inside Kvasir while it was being exploited, and trace values and invariants for the same were obtained. As expected, the invariant file, missing invariants for canary values as the function, never returned properly. It gave the clue of violation

occurred in the function. Further investigation with GDB (GNU Debugger) confirmed the same.

3.4. Integrity Measurement Component

Integrity measurement at runtime is a two-step process: memory acquisition and parsing. Memory acquisition is the critical step and it is nontrivial. It should not interfere with application execution. If this is not carefully done, the acquired memory will be corrupted, making the whole effort a futile exercise. I setup RADIUM server with LibVMI, the memory introspection library for Xen and KVM. It provides API for reading from and writing to a virtual machine's memory. I used volatility on top of LibVMI to run the integrity analysis. Volatility takes the memory dump of a system as input and performs various introspection operations on it. With LibVMI API, Volatility can run the analysis of a live VM's memory. Volatility needs to have a created profile of the OS kernel memory. The profile is a zip file of kernel data structures and debug symbols, used to identify critical data in the physical memory and parse it.

As an initial step, I tried to observe integrity for stack constraint violations, specifically return address constraint. In chapter 2, I explained how I instrumented applications and obtained data invariants connected to the return address constraint. I used a simple method of obtaining a memory snap shot of the process at function entry and right before return. Using a parsing script, the values on the stack (local variables, arguments, frame pointer, return address, etc.) were extrapolated and verified against invariant data (variable data from training). Before starting the target-VM, the invariant data was input to the measurement component from a file. texttttLinux_pslist command was used to find the PID of the application (Prozilla). Linux_proc_maps and linux_mem_maps commands were executed on the VM to obtain process memory map. From memory map the address range of the stack of application was identified and extracted that part of the memory as a raw dump. These commands were run as a part of parsing script. A list of commands in exact syntax is listed below.

```
$sudo python vol.py --profile LinuxUbuntu1204x86 linux_pslist -l
vmi://<Vm name>  | grep proz
$sudo python vol.py --profile LinuxUbuntu1204x86 linux_proc_maps -l
vmi://<Vm name> -p <PID>
```

To obtain memory, the process has to be halted to avoid corruption of memory dump. This would induce some performance overhead, but it is necessary for accurate measurement. I converted the binary format of memory into readable format, using hexdump [27] and gnu strings [34]. I then parsed it for invariants (values of canary variables). Constraint violations were observed when the stack was corrupted by simulated attacks. The parser was unable to find canary values and return addresses, which were overwritten by the malicious buffer code from the exploit. In this implementation, I only tested one application at a time, so the evidence sealing to TPM can be as simple as saving a "state" value to TPM's PCR 17 and 18 using the commands given in section 3.1.

3.5. Security and Efficiency Analysis

Integrity measurements were taken during function execution so that any modification of memory objects during the execution phase could be caught. Measuring service running RIVeR would detect an attack on measured application and secure the evidence in TPM. Even if an attacker is successful in executing a shell code and gaining access to root privileges on a measured VM, it will not be possible to access or manipulate the evidence. These attacks could include network-based attacks from compromised VMs on RADIUM or from an outside adversary trying to exploit vulnerabilities in OS or applications. InterVM attacks are prohibited through the access control mechanism of Xen Security Module. The communication protocol between measured service and challenger is also secure against replay and spoofing attacks as random nonces and TPM generated attestation keys are used. Attack protection on the hypervisor and measuring service and static attestation of target VM are discussed in detail in [22].

Performance overhead due to integrity measurement was calculated by taking the

TABLE 3.1. Results: Performance and Efficiency

Prozilla 1.3.7						
Vulnerability	CVE-ID	Invariant/constraint	Use Cases	Attacked Simulated	Attacks detected	Performance overhead
Remote Stack overflow	CVE:2004-1120	Return address	Limited bandwidth	Yes	Yes	80%
			Interrupted download	Yes	Yes	102%
			Single thread	Yes	Yes	64%
			Different Port	Yes	Yes	110%
Format string	CVE:2005-0523	Data invariant	Limited bandwidth	Yes	Yes	60%
			Interrupted download	Yes	Yes	60%
			Single thread	Yes	Yes	50%
			Different Port	Yes	Yes	77%
Result handling buffer overflow	CVE:2005-2961	Data invariant	Limited bandwidth	Yes	Yes	65%
			Interrupted download	Yes	Yes	60%
			Single thread	Yes	Yes	54%
			Different Port	Yes	Yes	90%
Ghttpd 1.4						
Get request overflow	CVE:2001-0820	Return address	Web Page Access	Yes	Yes	27%
			File Not found	No	-	12%
			Virtual Hosts enabled	Yes	Yes	86%
Deamon Overflow	CVE:2002-1904	Caller-Calee	Web Page Access	Yes	Yes	33%
			File Not Found	No	-	12%
			Virtual Hosts enabled	Yes	Yes	56%
Nullhttp 0.5						
Get request overflow	CVE:2001-0820	Return address	Web Page Access	Yes	Yes	27%
			File Not found	No	-	12%
			Virtual Hosts enabled	Yes	Yes	86%
Deamon Overflow	CVE:2002-1904	Caller-Calee	Web Page Access	Yes	Yes	33%
			File Not Found	No	-	12%
			Virtual Hosts enabled	Yes	Yes	56%

average speed of application over five executions for downloading different file sizes. Running time was measured with and without the integrity measurement. Integrity verification has an average runtime overhead range of 50%-110%. There was some time lapse between the moment attacks happened and the moment it was detected, as parsing of memory map is being performed while the application continues execution. This is done to reduce run time.

Usage of function calls as measurement points also influenced the performance.

I was able simulate and detect two exploit attacks on Prozilla, Ghttpd, and Nullhttp: buffer overflow and format string vulnerabilities. So far, the procedure has been effective in detecting attacks. The exploits used to simulate attacks were obtained from www.exploit-db.com. Verifying an application from inside the host machine would be more efficient, but this would defeat the advantage of the trust brought in by asynchronous architectures like RADIUM. The results are shown in Table 3.1.

CHAPTER 4

RELATED WORK

In this chapter, some of the works related to this thesis is summarized. My work is inspired by [22] and closely related to [20]. Here, I have presented a summary and analysis of these works and mentioned other works containing concepts that overlap with my work. I intended to make my tool work in a very different environment (i.e. on a trusted hypervisor), while most of the related works I summarized are host-based solutions.

4.1. Remote Attestation to Dynamic System Properties: Towards Providing Complete System Integrity Evidence

Remote attestation is a mechanism used for providing integrity evidence of a platform to a remote entity. Currently, there are many static remote attestation mechanisms, but to detect runtime attacks, dynamic attestation is needed. The authors identified that some stack properties like saved frame pointer, return address, and stack pointer can be useful in dynamic attestation. The integrity of the system can be measured based on these dynamic properties. The attestation process must also be protected while measuring the dynamic properties for integrity.

ReDAS (Remote Dynamic Attestation System) [20] "performs application-level dynamic attestation by monitoring running applications and secures integrity violation using hardware support." The authors classified dynamic properties into two types: Structural and Global data properties. These properties are extracted from applications during a training phase. The integrity measurement component observes the application during the execution phase and verifies whether the dynamic properties are unmodified by comparing their state with values obtained during the training phase. The application is monitored during system call time. This gives a fine balance between performance and granularity. Thus, data invariants and stack structural constraints are measured for their integrity. ReDAS uses a TPM assisted trusted mechanism to secure integrity evidence. TPM's Platform Configuration Registers are a tamper-proof form of storage technology. According to TPM Specifications, "The

measurement values are encrypted with one-way hash function and are extended to PCR 8. (PCRs are numbered from 0-15)" [16]. In the dynamic attestation protocol, challenger sends the request along with a nonce. "The attestation service responds with a two-part message; the first one is current integrity evidence and the send one is TPM signature and nonce received from the requester. Nonce is protection against reply attacks while hashed value protects against duplication" [20].

The main shortcoming of ReDAS is that it uses a static chain of trust at the time of boot. Then, the kernel or integrity verification module can be compromised at runtime. RADIUM, which is a dynamic attestation platform, overcomes this shortcoming. The machine no longer needs to be restarted every time an attestation is performed. ReDAS also inherits limitations of Daikon. For example, it cannot infer useful invariants for complex structures like linked lists. Therefore, ReDAS cannot detect violations that are not covered by the defined set of invariants. Some violations may be missed due to the transient nature of applications (i.e. violations may erase the trace before they are captured).

4.2. Automatic Security Debugging Using Program Structural Constraints

Understanding and detecting security bugs in an application is a cumbersome process, as it needs manual tracing of the runtime behavior of applications. Manual debugging is not a good solution for detecting unknown vulnerabilities. Oftentimes the attacks last for very short lengths of time, which makes detection difficult. Program structural constraints make the debugging process more efficient. These constraints can be observed during the runtime of an application to find vulnerabilities. The authors used "static analysis methods to determine structural constraints and dynamic monitoring component to verify if the constraint is satisfied or not" [21]. The tool is called CBones.

On a Linux platform, an application's ELF [12] has a memory structure as follows: Code, Data, Heap, DSO, Stack, and Kernel (Figure 2.2). Of all these segments, stack, heap, and data are considered the most important areas in which to look for clues of correct execution. Therefore, the structural constraints are defined for these segments. Some of the constraints the authors have defined are: caller-callee constraint, return address constraint,

frame pointer constraint, saved register constraint, saved frame pointer constraint, stack Limit constraint (Stack), memory allocation/de-allocation request constraint, boundary tag constraint, heap boundary constraint, chunk boundary constraint, consecutive free chunks constraints (Heap), shared library function pointer constraint, constructor function pointer constraint, destructor function pointer constraint.

Choosing the point of time to observe the constraints is key for successful detection of vulnerabilities. This is the period when the program is in a transient state with respect to that constraint. For stack constraints, the time for observation is from function call to time when prologue is setup. This can be continued until the memory de-allocation. To find constraint violations, a two-step procedure is used. In step one, a coerce grained approached is used to find the function in which violation occurs. Then, in second step, the exact violated constraint is determined. CBones implementation has two components: constraint extractor and monitoring agent. Constraint extractor is essentially a Ruby script [1]. For a C program, it extracts constraint information such as name, address, activation record size, and saved registers count. The program has to be compiled with debugging flag (-g) in order to extract this information from binary. The debugging output, obtained in dwarf format, is parsed with a dwarf-parser written in Ruby. Objdump [2] is used to disassemble the binary and extraction call instructions for caller-callee constraint. Constraint extractor can only work on binaries complied without any optimizations.

Integrity monitoring agent was developed as skin to Valgrind [29], a software CPU-emulator program, which is popularly used as a debugging tool. Valgrind uses its own intermediate assembly language called Ucode. The constraints extracted are stored in the monitoring agent in internal data structures. Procedure, CallStack, and ChunkList are some of those internal data structures. A limitation of this monitoring agent is that it cannot verify a structural constraint while the program is in a transient state with respect to this constraint. The monitoring agent deems a situation a "safe point" when no constraint is

[1]Ruby, a programming language.

[2]A GNU binary utility used for disassembling to learn about object files.

violated. It intercepts function calls and returns by capturing "jump" and verifies stack and heap for violations. Monitoring agent also checks for integrity of memory writes by capturing responsible system calls.

CBones is mainly a debugging tool used to verify and test for vulnerability presence. For some applications, it has performance overhead of 15-50 times. High overhead is common for debugger. While using similar techniques for integrity violation, a trade-off between performance and efficiency is inevitable.

4.3. DIDUCE

DIDUCE [17] is a bug detection tool based on dynamic likely invariants. It instruments the program dynamically to hypothesize invariants. This tool is developed for program errors and to identify remote use-cases, which are otherwise hard to find. DIDUCE obtains and verifies the invariants at runtime to find anomalies. The method of invariant extraction is similar to Daikon, But DIDUCE was developed for Java programs and is more of a bug detection tool than an integrity measurement tool.

4.4. Purify

Purify [18], a quality assurance tool, specializes in bugs related to memory leaks and access errors. Object code is instrumented, with memory access checking instructions, to verify the legitimacy of every read and write operation on memory. Purify only runs on SPARC architecture, and, like other debugging tools, it slows down execution.

4.5. Copilot

Copilot [31] uses extended hardware support (PCI Card) to measure runtime kernel integrity. In Copilot, Memory is accessed dynamically and probed for the malware presence by computing hashes. This method is unlikely to be adopted in existing systems, as it adds additional hardware cost.

4.6. Runtime Detection of Heap Based Overflows

This work [33]is similar to Stack Shield [38] and StackGuard [13], but the solution is for detection of heap overflows, while [38] and [13] are protection mechanisms against memory corruption attacks on stack. This method is implemented as a library of C language and uses the idea of boundary tags for heap management. In C programming, heap is used for dynamic memory allocations at runtime, using library functions like malloc and calloc. Free memory blocks on heap are called "chunks," and each chunk header contains the size of current and previous chunks. "This information is called in-band information, or boundary tag information," [33] and is used for traversing through all chunks of heap. Heap overflow exploits target the pointer used to link the available chunks, and it controls values on chunks. Canary values are introduced into memory chunk structure to detect heap overflows at runtime. The canary is checksum of the chunk header seeded with a global variable, which is assigned a random variable at the startup of the program. This is similar to random canaries used in StackGuard. The heap management algorithms are modified accordingly to detect any overflow-based attacks that overwrite the canary values. When the chunk is de-allocated with "free" call, the canary value is calculated and verified. If any change to the value is detected, the process will be aborted. Though this method is effective in detecting heap-based overflows, this does not cover protection of stack and is a host-based approach, which inherits limitations of untrusted environment.

4.7. Flicker

Flicker [26] is an isolated environment for security-sensitive code execution. Trust is based on the extremely small size of the TCB (Trusted Computing base) code. Flicker tries to solve the security problems that arise with large sized operating systems, which has huge attack surface and major part of the OS code is executed in privileged mode. The vulnerabilities in code are the attack points for adversaries to gain privileged access. The idea is to separate the part of execution with high privileges from the rest of the execution. The isolated security-sensitive code, which only have to trust a minimal TCB makes Flicker a better-trusted platform. Before or after execution of privileged code in Flicker, it is not

accessible. Integrity of execution can be attested to a challenger from outside. Flicker makes use of TPM-based sealed storage capabilities for attestation of trusted state.

The downside is the applications have to be redesigned with sensitive code that has to be executed separate from rest of the application. This approach is not practical for all types of applications and Flicker has high performance overhead. As Flicker allows untrusted environments to execute, malicious code can launch DOS attacks.

4.8. TrustVisor

TrustVisor [25] is another attempt to use minimal TCB for creating isolated execution environments. The isolated, measured environment created by a TrustVisor, which is a special purpose hypervisor, protects selected parts of applications. In this model the trust is moved from OS or application to TrustVisor. It uses DRTM based trusted launch. The secure execution is protected against local adversaries who can run arbitrary code on legacy application and network based adversaries who can attack network traffic.

4.9. OSLO

Usage of TPM's on commodity and commercial systems is gaining ground; Microsoft's bit locker disk encryption system is an example. The authors evaluated the security of TPM and root of trust model architectures. Two primary goals of trusted computing were discussed in this work: remote attestation and tamper evident sealed storage.

Authors identified some bugs in trusted boot loaders and ways to exploit them. One of the bugs was that kernel image was not completely hashed, which breaks the continuity of chain in trusted boot loading process. Another bug was hashing and loading of kernel image was happening independently. In their first attack, authors found out a way to reset TPM without having to reboot the system. They achieved this by grounding the LRESET pin, which reset the register. Obviously to perform this attack, physical access to the platform is required. Cases of this attack include stolen devices as well as device owners resetting their own TPM to exploit Digital Rights Managements if they are using TPM. The second attack was exploiting the allowance of unverified BIOS as CRTM. The attack was demonstrated

on a laptop with TPM 1.2. The authors replaced the BIOS with a modified version where they removed MPTPMTransmit() from BIOS driver of TPM. As a result, was the machine lost its trust. These attacks were designed to show that the TPM implementation did not meet the trusted chain requirements. DRTM based OSLO [19](Open Secure Loader) was proposed as a solution to above-mentioned bugs and attacks.

OSLO's DRTM makes PCR 17 resettable any time through a secure process, thus avoiding the need to reboot the machine every time a measurement is needed. PCR 17 reset is different from whole TPM reset as a special bus cycle is used to do the reset. Because of this TPM reset cannot be faked as PCR 17 reset. The fact that BIOS and Bootloader are no longer part of CRTM trust chain also makes DRTM secure against BIOS and Bootloader attacks. Secure Loader (SL) is used to initialize CPU and SKINIT instruction to extend SL into PCR 17 of TPM. OSLO was built on top of AMD's DRTM technology and was written in C. Although OSLO was successful in reducing TCB footprint, it is vulnerable to TOCTOU attacks as discussed in Section 2.6 of this document. It is also worth mention that RADIUM is built on top of DRTM, an improvement of it.

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1. Conclusion

In this thesis, I have tried to solve the problem of runtime integrity of application on trusted platforms from hypervisor level. First, I presented a way to find a known correct behavior of application. The concept of program invariants was used to deduce properties of an application that ensure its integrity. I demonstrated a way of using Daikon invariants to verify security sensitive bugs in applications, and I developed a method for run time integrity of application using memory introspection tools. The invariant data from Daikon and manually formulated structural invariants based on program rules were compared for verification of application integrity. I developed a Python program for memory page parsing in order to determine attacks on applications. In the current method, function entries and exits were used as measuring points. In this process, verifying invariants induced some overhead. Runtime VM introspection support provided by LibVMI was used to enhance memory forensics capability of Volatility. On top of these two tools, the Python program that I developed was used to determine invariants on running applications. The framework was tested for efficiency on real world applications (Prozilla, Nullhttp, and Ghttpd). The goal of my research was to present a proof of concept for the runtime integrity measurement capabilities at application level on RADIUM, and I was successful in detecting runtime attacks. The current method does incur a high performance overhead, but that is not the focus of this work.

5.2. Future Work

To increase the performance and efficiency, different measurement points can be used. Obtaining and observing the invariants system calls reduces the performance overhead considerably compared to the current method of observing at function calls. This reduction in performance overhead can be achieved by modifying Daikon and Kvasir. As the dynamic properties used in the current model are not complete, more invariant properties are needed

to make the integrity verification efficient. For example, there are no invariant definitions useful in detecting attacks on heap or complex data structures. There is also room for improvement in the parsing algorithm needed to obtain fine-grained measurements.

Time consumed to pause the VM for memory snapshot extraction can be reduced by integrating integrity measurement capability into LibVMI, thus reducing performance overhead. This can be done as library in LibVMI or as a Volatility module. The invariant extraction process can be further improved by adding control flow behavior of application. This can be done by using debugging tools like Strace, GDB, or Valgrind and by porting the data into invariants. This approach can further be used in predicting vulnerable measuring points of application, in addition to using vulnerable areas as the measurement points.

REFERENCES

[1] *Cve-id: Cve-2001-0820, buffer overflows in gaztek ghttpd 1.4*, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2001-0820.

[2] *Cve-id: Cve-2004-1120, multiple bufferoverflows in prozilla 1.3.7*, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2004-1120.

[3] *Cve-id: Cve-2005-0523, format string vulnerability in prozilla 1.3.7*, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-0523.

[4] *Gcc, the gnu compiler collection*, GNU Project, https://gcc.gnu.org.

[5] *Heartbleed, vulnerability summary for cve-2014-0160*, https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0160.

[6] *Trusted grub*, https://www.sirrix.com/content/pages/trustedgrub_en.htm.

[7] Serkan Akpolat, *Prozilla - remote stack overflow exploit*, 2004, http://www.exploit-db.com/exploits/652/.

[8] Serkan Akpolatt, *Prozilla - remote format string exploit*, 2005, http://www.exploit-db.com/exploits/806/.

[9] Stefan Berger, Ramón Cáceres, Kenneth A. Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn, *vtpm: Virtualizing the trusted platform module*, Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15, USENIX-SS'06, 2006.

[10] Matt Bishop, *Introduction to computer security*, Addison-Wesley Professional, 2004.

[11] Joseph Cihula, *Trusted boot: Verifying the xen launch*, Xen Summit 7 (2007).

[12] Tool Interface Standard (TIS) Committee, 1995.

[13] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang, *Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks*, Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7, SSYM'98, 1998, pp. 5–5.

[14] Daikon, *The daikon invariant detector user manual*, 2015, http://plse.cs.washington.edu/daikon/download/doc/daikon.html.

[15] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh, *Terra: A virtual machine-based platform for trusted computing*, ACM SIGOPS Operating Systems Review, vol. 37, ACM, 2003, pp. 193–206.

[16] Trusted Computing Group, *Tpm specifications version 1.2*, http://www.trustedcomputinggroup.org/developers/trusted_platform_module/specifications.

[17] Sudheendra Hangal and Monica S. Lam, *Diduce: Tracking down software bugs using automatic anomaly detection*, Proceedings of the 24th International Conference on Software Engineering, ICSE '02, ACM, 2002, pp. 291–301.

[18] Reed Hastings and Bob Joyce, *Purify: Fast detection of memory leaks and access errors*, In Proc. of the Winter 1992 USENIX Conference, 1991, pp. 125–138.

[19] Bernhard Kauer, *Oslo: Improving the security of trusted computing*, Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium, SS'07, USENIX Association, 2007, pp. 16:1–16:9.

[20] Chongkyung Kil, E.C. Sezer, A.M. Azab, Peng Ning, and Xiaolan Zhang, *Remote attestation to dynamic system properties: Towards providing complete system integrity evidence*, Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP International Conference on, June 2009, pp. 115–124.

[21] Chongkyung Kil, E.C. Sezer, Peng Ning, and Xiaolan Zhang, *Automated security debugging using program structural constraints*, Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual, Dec 2007, pp. 453–462.

[22] Srujan Kotikela, Mahadevan Gomathisankaran, Gelareh Tabani, et al., *Radium: Race-free on-demand integrity measurement architecture*, (2015).

[23] libVMI, *libvmi*, http://libvmi.com/.

[24] P. J. Guo S. McCamant C.Pacheco M. S. Tschantz M. D. Ernst, J. H. Perkins and C. Xiao, *The daikon system for dynamic detection of likely invariants.*, Science of Computer Programming (2007), 34–45.

[25] Jonathan M McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor,

and Adrian Perrig, *Trustvisor: Efficient tcb reduction and attestation*, Security and Privacy (SP), 2010 IEEE Symposium on, IEEE, 2010, pp. 143–158.

[26] Jonathan M McCune, Bryan J Parno, Adrian Perrig, Michael K Reiter, and Hiroshi Isozaki, *Flicker: An execution infrastructure for tcb minimization*, ACM SIGOPS Operating Systems Review, vol. 42, ACM, 2008, pp. 315–328.

[27] David Mertz, *Hexdump. the gnu text utilities*, http://gnosis.cx/publish/programming/text_utils.html.

[28] I. Molnar, *Exec-shield*, http://people.redhat.com/mingo/exec-shield/.

[29] Nicholas Nethercote and Julian Seward, *Valgrind: A framework for heavyweight dynamic binary instrumentation*, SIGPLAN Not. 42 (2007), 89–100.

[30] Gareth Owen, *ghttpdsweb server*, http://gaztek.sourceforge.net/ghttpd/index.html.

[31] Nick L. Petroni, Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh, *Copilot - a coprocessor-based kernel runtime integrity monitor*, Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13, USENIX Association, 2004, pp. 13–13.

[32] Qitest1, *ghttpd 1.4 daemon buffer overflow vulnerability*, 2001, http://www.exploit-db.com/exploits/20929/.

[33] William Robertson, Christopher Kruegel, Darren Mutz, and Fredrik Valeur, *Run-time detection of heap-based overflows*, Proceedings of the 17th USENIX Conference on System Administration, USENIX Association, 2003, pp. 51–60.

[34] GNU Strings, *Gnu strings*, https://sourceware.org/binutils/docs/binutils/strings.html.

[35] PaX Team, *Pax address space layout randomization (aslr)*, http://pax.grsecurity.net/docs/aslr.txt.

[36] Totosugito, *Prozilla download manager*, 2013, https://github.com/totosugito/prozilla-2.0.4.

[37] US-CERT, *Home land security software assurance*, https://www.us-cert.gov/sites/default/files/publications/infosheet_SoftwareAssurance.pdf.

[38] Vendicator, 2001, http://www.angelfire.com/sk/stackshield/.

[39] Volatility, *Volatility*, http://code.google.com/p/volatility/.

[40] John Wilander and Mariam Kamkar, *A comparison of publicly available tools for static intrusion prevention*, 2002.