RADAR: COMPILER AND ARCHITECTURE SUPPORTED INTRUSION PREVENTION, DETECTION, ANALYSIS AND RECOVERY

A Thesis
Presented to
The Academic Faculty

by

Tao Zhang

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
December 2006

RADAR: COMPILER AND ARCHITECTURE SUPPORTED INTRUSION PREVENTION, DETECTION, ANALYSIS AND RECOVERY

Approved by:

Dr. Santosh Pande, Advisor
College of Computing
Georgia Institute of Technology

Dr. Karsten Schwan
College of Computing
Georgia Institute of Technology

Dr. Mustaque Ahamad
College of Computing
Georgia Institute of Technology

Dr. Jun Yang
School of Electrical and Computer Engineering
University of Pittsburgh

Dr. Wenke Lee
College of Computing
Georgia Institute of Technology

Date Approved: [Aug 18, 2006]

ACKNOWLEDGEMENTS

First and foremost, I thank my parents. Victims of Culture Revolution in China, they were deprived of the opportunities of higher education and spent their prime time in a remote and unknown village. They put their hope on me and did everything within their power to support my growth. Mom, Dad: I hope what I achieved so far makes you proud.

Many thanks to Santosh Pande. As my advisor, he gave me guidance, wisdom and encouragement throughout my Ph.D. study. Without his help, there is no way for me to finish this broad thesis topic. He is also a good friend in life and always supported me whenever he can.

Special thanks to Xiaotong Zhuang. It is fortunate to have him as my most important colleague. He is very smart and it is always enlightening to discuss research problems with him. He contributed a lot to my thesis work.

Thanks to my committee members for providing critical comments to my dissertation and helping me make it a quality one. In addition to my advisor, the committee consisted of Mustaque Ahamad, Wenke Lee, Karsten Schwan and Jun Yang. Thanks are also due to other professors and students who helped me along the way.

Finally, I thank all my friends in and out of Atlanta who made my life in the past several years a memorable experience.

Tao Zhang, August 2006.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Computer technology has changed our life fundamentally and computer systems have become indispensable for the proper functionality of our society. On the other hand, the rapid increase of the society's dependence on computer systems brings great interest to break in and attack those systems. Computer software determines the functionality of computer systems and is the primary target of attacks. With today's pervasive presence of computer systems and network connections, securing critical software from attacks has become an extremely important problem and has never been as challenging.

Critical software faces both software-based attacks and hardware-based attacks. Software-based attacks may come from other malicious software. For example, attacking software may read/write victim software's address space through flaws in process isolation. Software-based attacks may also break software by exploiting all kinds of vulnerabilities in the victim software, such as notorious buffer overflow vulnerabilities. Hardware-based attacks break software by utilizing specialized hardware and attacking the system on which the software is running, such as snooping system buses during the execution of the software. Software attacks are more commonly known. However, combating hardware attacks has been an extremely important problem in secure embedded systems domain, such as smart cards, and is becoming more and more relevant in general purpose computing domains.

In this dissertation, we propose RADAR – compileR and micro-Architecture supported intrusion prevention, Detection, Analysis and Recovery. RADAR is an infrastructure to help prevent, detect and even recover from attacks to critical software.

Instead of being a purely software-based approach or a purely hardware-based approach as in previous approaches, our approach emphasizes collaborations between compiler and micro-architecture to avoid the problems of purely software or hardware based approaches. Our infrastructure is based on micro-architecture level support and has its security rooted in hardware. At the same time, we call for compiler assist whenever it is necessary, such as to obtain expected software behavior, or whenever it is helpful to reduce the complexity of the micro-architecture support. With both micro-architecture and compiler support, our infrastructure can defend against both software and hardware attacks with superb security strength but reasonable hardware and performance cost.

We believe that a purely software-based approach is not able to meet the security challenges faced by critical software. First, a purely software-based approach can be easily reverse-engineered and then cracked. In addition, a purely software-based approach cannot defend against hardware attacks thus is not applicable in situations where hardware attacks are real threats. More importantly, security operations implemented in software are much more expensive than the hardware implemented version. The potential performance penalty greatly limits the security strength that a software-based approach can achieve. Overall, a purely software-based approach may not be able to achieve a satisfying security guarantee for critical software.

Thus, we believe that it is time to call for micro-architecture level support for software security. With hardware support for cryptographic operations, such as encryption, decryption and hashing, our infrastructure can achieve strong process isolation to prevent attacks from other processes and to prevent certain types of hardware attacks, such as using specialized hardware to read/tamper the system data bus traffic or

the external memory system directly to evade process isolation mechanism completely. Moreover, we show that an unprotected system address bus leaks critical control flow information of the protected software but has never been carefully addressed previously. The information leakage could facilitate an attack and bring significant damage to both code and data confidentiality. To enhance intrusion prevention capability of our infrastructure further, we present a scheme with both innovative hardware modification and extensive compiler support to eliminate most of the information leakage on system address bus.

However, no security system is bullet-proof and is able to prevent all attacks. Although our hardware infrastructure is able to achieve strong process isolation and to prevent common hardware attacks, it does not prevent attacks exploiting software vulnerabilities such as buffer overflow attacks. In general, we have to assume that certain attacks will get through our intrusion prevention mechanisms. To protect software from those attacks, we build a second line of defense consisted of intrusion detection and intrusion recovery mechanisms.

Our intrusion detection mechanisms are based on anomaly detection. In other words, we try to detect anomalous program behavior caused by attacks based on expected program behavior. Anomaly detection does not target to specific attacks and is able to detect novel or unknown attacks. In this dissertation, we propose three anomaly detection schemes. The first one is a training based scheme to detect anomalous dynamic program paths. The scheme monitors the software execution at a very fine granularity and is able to detect attacks with high precision and neglectable performance overhead. The major concern of the scheme is false positives. So we propose the second scheme based on

compiler branch correlation analysis to detect infeasible paths without any false positives. But the detection capability is not as good. Finally, we present a third scheme based on compiler data flow analysis to detect tampering to critical software data. The third scheme is able to achieve both zero false positives and strong detection strength, but the performance overhead is much higher. We demonstrate the effectiveness of our anomaly detection schemes thus the great potential of what compiler and micro-architecture can do for software security.

After an intrusion is detected, most previous approaches simply shut down the attacked software to avoid any further damage. However, the ability to recover from an attack is very important for systems providing critical services. Thus, intrusion recoverability is an important goal of our infrastructure. To provide intrusion recoverability, two major tasks have to be done. First, we need to analyze the attack to identify exactly when and how the attack happens so that we can identify the tampered system state and gather useful information for later forensic analysis. Second, the tampered system state has to be recovered through certain mechanisms. We focus on memory state in this dissertation, since most attacks break into a system by memory tampering. Intrusion analysis is a difficult problem since there could be an arbitrarily large interval between the tampering to the system state and the detection of the tampering. We propose two schemes for intrusion analysis. The execution logging based scheme incurs little performance overhead but has higher demand for storage and memory bandwidth. The external input points tagging based scheme is much more space and memory bandwidth efficient, but leads to significant performance degradation. After intrusion analysis is done and tampered memory state is identified, tampered memory

state can be easily recovered through memory updates logging or memory state checkpointing.

In summary, our RADAR infrastructure aims to protect critical software from both hardware attacks and software attacks with strong security guarantee. It integrates the abilities of intrusion prevention, intrusion detection, intrusion analysis and intrusion recovery, which are the major aspects of protecting software from malicious attacks. It emphasizes collaborations between compiler and micro-architecture to achieve those abilities. With micro-architecture level support and compiler assist, our RADAR infrastructure can achieve strong security guarantee for critical software with reasonable hardware cost and performance degradation.

1 INTRODUCTION

In this dissertation, we propose an infrastructure called RADAR to help prevent, detect and even recover from attacks to critical software. Instead of being a pure software-based approach or a pure hardware-based approach, our approach emphasizes collaborations between compiler and micro-architecture to avoid the problems of pure software or hardware based approaches. Our infrastructure is based on micro-architecture level support and has its security rooted in hardware. At the same time, we call for compiler assist whenever it is necessary, such as to obtain expected software behavior, or whenever it is helpful to reduce the complexity of the micro-architecture support. With both micro-architecture and compiler support, our infrastructure can defend against both software and hardware attacks with superb security strength but reasonable hardware and performance cost.

1.1 Motivation

Computer technology has changed our life fundamentally and computer systems have become indispensable for the proper functionality of our society. Numerous embedded systems form parts of critical systems in automobiles, aircrafts, satellites, robots, appliances, and medical devices to control their operations. At the same time, conventional computer systems are integral parts of critical infrastructures such as financial services, telecommunication, transportation, energy production and distribution networks. In the year 2000, 385 million microprocessors were produced for use in conventional computers, at the same time 6.4 billion microcontrollers were manufactured for use in embedded devices [51].

The increasing of the society's dependence on computer systems brings great interest from criminals and terrorists to break in and tamper those systems. Although the vast majority of computer security incidents are not reported to public, we can still get a sense of the severity of the problem by those publicly available ones. For example, on Apr. 12 2001, VISA lost 20 million dollars due to extortion by hacker organizations. On Aug. 26 2002, systems of Daewoo Securities were hacked and $21.7 million stock was illegally sold [112]. On Jun. 2005, CardSystems Solutions Inc, a third-party processor of payment-card data, was breached and as many as 40 million cards may have been exposed [71]. The impact of incidents occurred in military or national security related areas will be far more significant but those incidents most likely will not be publicly reported. Figure 1 shows the number of Internet incidents reported to CERT [15] in recent years. This graph merely shows the tip of the iceberg, but it can be easily seen that the situation gets worse quickly.

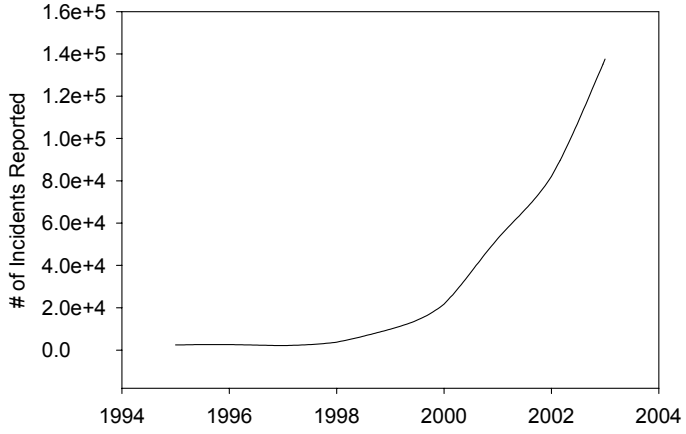


Figure 1. Number of incidents reported to CERT.

Koopman showed the potential scale of the computer security problem in the future using thermostat as an example[58]. A household thermostat controls heating and cooling of the house. Most thermostats have an embedded computer and many of them will have internet connection in the future to allow remote control from the house owner or the utility company. Tampering of one thermostat may not be able to cause significant damage, but what if millions of them are controlled by an attacker? The situation is very realistic since those thermostats most likely will execute the same software thus can be hacked in the same way. Moreover, they are likely to be under center control of the utility company, which could also be broken by the attacker. With millions of thermostats under control, the attacker can do serious damages. He can set the target temperature to an extreme point and cause a black out of a power grid. Or he can bump the temperature a little bit to increase energy consumption and inflate energy bills in a subtle way. Obviously, tampering to more complicated computer systems having more responsibilities could lead to much more severe damages.

In summary, with more and more pervasive presence of computer systems and network connections in the future, securing critical computer systems from attacks has become an extremely important problem and has never been as challenging. Computer software determines the functionality of computer systems and is the primary target of attacks. In this dissertation, we will focus on the problem of protecting critical software from attacks.

Critical software faces both software-based attacks and hardware-based attacks. Software-based attacks may come from other software. For example, attacking software may read/write victim software's address space through flaws in process isolation.

Software-based attacks may also break software by exploiting all kinds of vulnerabilities in the target software, such as notorious buffer overflows. Hardware attacks break software by utilizing specialized hardware and attacking the system on which the software is running, such as snooping system buses during the execution of the software. Software attacks are more commonly known. Preventing hardware attacks has been an extremely important problem in secure embedded system domain, such as smart cards, and is becoming more and more relevant in general purpose computing domains.

Protecting software from attacks has several aspects including intrusion prevention, intrusion detection and intrusion recovery. Measures can be taken to prevent certain attacks or make certain attacks computationally difficult. On the other hand, there is no perfect security so attacks or intrusions will occur thus we need mechanisms to detect those attacks. Finally, in certain situations it is critical that the software can recover from intrusions and continue its correct execution.

1.2 Traditional Software-Based Approaches

Software protection is a traditional research topic. Traditional solutions are purely software-based and mostly focus on intrusion prevention and intrusion detection. We give several examples to show the problems of a purely software-based solution.

Software-based encryption is a classical mechanism to prevent attacks to software confidentiality. A part of the software may be encrypted by an activation key only known to the legitimate user. However, the scheme can be easily broken as long as the attacker has access to one legal copy. Although a part of the program binary is encrypted, without hardware support, the processor cannot execute encrypted code so that the encrypted code has to be decrypted first in memory. The attacker can debug/reverse-engineer

[35][112] the protected software during its execution and find out the whereabouts of the decrypted code and dump its content, then he has a decrypted version of the software.

Software-based self-checksumming is a popular technique to detect tampering to software due to attacks [16][45]. Checksumming code is spread through the software and the software code/data is verified against its previously computed checksum during execution. However, the attacker can still reverse-engineer and understand the implementation of checksumming technique. Then he can learn how to tamper the software data without triggering a checksum failure or attack the checksumming code directly to remove/bypass it. The amount of work is normally insignificant for an experience attacker and is trivial for a well-funded organization.

Software obfuscation [5][19][20][21][70][80][65] has been proposed to make reverse-engineering more difficult by transforming the original program code. The techniques used include instructions shuffling, replacing, scrambling, dummy code insertion, branch function creation etc. However, software obfuscation cannot achieve any security guarantee but only increases the effort required to attack the software. Moreover, the research of deobfuscation is also active, which could help attackers greatly. As a good example, after an obfuscation technique to make static disassembly difficult is proposed in [65], it is promptly broken in [25]. More ironically, an extremely competent Java obfuscator has been subverted and used as a powerful deobfuscator [74].

General intrusion detection techniques based on detecting anomalous program behavior are also proposed [32][115][92][31][30][59][76][37][36]. The detection strength of those anomaly detection techniques largely depends on the granularity at which they monitor the target program. Since current approaches are implemented in software, they

are not able to monitor the target software at a very fine granularity due to the potential performance overhead. Currently, most of them monitor at system call level such as in [32]. Due to the coarse monitoring granularity, certain attacks will be missed by those schemes.

1.3 Existing Hardware-Based Approaches

Trusted Computing Group (TCG) [107] is an industrial alliance promoting trusted computing, founded by Microsoft, Intel, IBM, HP and AMD. The goal of it is to develop specifications for a secure computing platform against software-based attacks to software confidentiality and integrity. The so-called trusted computing platform incorporates several security components. The most noteworthy one is a secure co-processor called trusted platform module (TPM). The security of the whole platform relies on the TPM, which can be used to protect data confidentiality by encrypting/decrypting and to verify the integrity of the software by hashing. Other security components include a processor having a strong process isolation feature (implemented in Intel's LaGrande Technology [48]), a secure operating system kernel and a secure kernel in each trusted computing application (both provided by Microsoft's NGSCB infrastructure [79]). There are already several implementations of the TPM specification and TPM compliant chips are already available in many desktops and laptops models [105].

Recently, Lie et al. [64] proposed the XOM secure architecture design based on hardware supported encryption/decryption. With hardware support, program code/data is encrypted and decrypted on the fly by the secure processor during program execution. The effect is that the processor is able to execute "encrypted" program now. Only the processor has access to the plaintext of the original program code and data but the

attacker does not. Thus, their scheme prevents attacks to software confidentiality effectively through encryption/decryption. Lie et al. also proposed a hashing based integrity checking scheme to detect tampering to the software thus achieving certain intrusion detection too. Later, Gassend et al. [38] pointed out that the integrity checking scheme in [64] is vulnerable to replay attacks. Gassend et al. further proposed a hash-tree based integrity checking scheme to fix the problem. Since then, there has been some follow up work to improve over the initial work [124][102][101][95][67]. But the basic designs established in [64] and [38] are inherited.

1.4 Our Approach

In general, software protection techniques purely based on software cannot achieve strong security guarantee. Since the protection scheme is purely implemented in software, it can be easily reverse-engineered and attacked. On the other hand, lack of hardware support makes security operations too expensive to be applied aggressively, further limiting the security strength that a software-based protection scheme can achieve. As a matter of fact, software piracy and digital media piracy cost industries billions of dollars each year [12], which easily proves current software protection (in terms of confidentiality) schemes are ineffective.

Moreover, the development of computer technology and its applications has made securing critical software an extremely important and challenging problem. Although software and digital media piracy costs significant financial losses for software/entertainment industries, its damage is still limited. With computer software virtually controlling every critical service in modern society even including nuclear arms, how to secure critical software has become a problem of national priority. On the other

hand, the pervasive presence of network connections and the easy access to tremendous personal computing power make computer software face great threats and challenges never experienced before.

In particular, traditional software-based software protection approaches only tackle software-based attacks, in which attackers utilize software tools and methods to break in the software. Hardware attacks constitute another large category of attacks in which attackers utilize specialized hardware and manipulate the target system in an unintended way. Hardware attack techniques include probing, re-engineering, memory readout, bus snooping, side-channel information leakage exploitation etc. [3]. Hardware attacks are mostly concerned in embedded systems domain previously, especially in designs of tamper-resistant devices such as smart cards. However, we believe that to achieve software security with strong guarantee, hardware attacks have become realistic and dangerous enough to be considered in general secure system designs. We should indeed learn our lessons from the case of Xbox hacking [47], in which the carefully crafted security scheme was easily defeated by system bus snooping. The device utilized is commonly available in labs and can be readily purchased at a cheap price. Obviously a protection scheme purely based on software can do little to prevent hardware attacks.

In response to the ineffectiveness of traditional software protection schemes and the emerging new threats, we believe that it is time to call for micro-architecture level support for software security. With hardware support for security, the security of the system can be rooted in hardware and the system can achieve much better security guarantee than software-based ones. The reasons are two folded. First, hardware is very difficult to reverse-engineer and crack. Even an average programmer would be able to

disassemble, trace and study a software binary using numerous easily accessible tools, but reverse-engineering a hardware component is virtually impossible for common attackers. Second, hardware support can make previously infeasible security operations feasible and previously expensive security operations efficient, enabling a much stronger security strength.

We share the same view with both industry and academia. As discussed previously, there has been a strong drive to introduce micro-architecture level support for security in both industry and academia. However, the secure platform designed by TCG focuses on software-based attacks and does not concern about hardware attacks, so it will not help when hardware attacks are real threats. The protection strength of the TCG secure platform may be enough for common commercial applications such as digital rights management, but it is inadequate in areas in which the attacker is very powerful, well organized and well funded such as national security related areas. Thus, we believe to achieve a strong security guarantee, defending against common hardware attacks is necessary

The previous approaches taken by academia do have defending against hardware attacks as an important goal. The proposed secure architectures [64][38][124][102][101][95] are designed to achieve strong process isolation to prevent attacks from other processes and to prevent certain types of hardware attacks, such as using specialized hardware to read/write the external memory system directly to evade process isolation mechanism completely. However, those secure architecture designs are purely hardware-based and do not call for any software assist. The most significant problem is that they cannot defend against software attacks exploiting vulnerabilities of

the victim software. The reason is that without any software assist, the secure architecture cannot make any assumption about the expected software behavior. For example, it will not be able to tell whether a store instruction is writing inside a buffer or is overflowing the buffer. In addition, a purely hardware-based approach could introduce significant hardware cost and limit the applicability of the approach.

In this dissertation, we propose a security infrastructure called RADAR (compileR and micro-Architecture supported intrusion prevention, Detection, Analysis and Recovery) to help prevent, detect and even recover from attacks to critical software. Instead of being a pure software-based approach or a pure hardware-based approach, our approach emphasizes collaborations between compiler and micro-architecture to avoid the problems of pure software or hardware based approaches. Our infrastructure is based on micro-architecture level support and has its security rooted in hardware. At the same time, we call for compiler assist whenever it is necessary, such as to obtain expected software behavior, or whenever it is helpful to reduce the complexity of the micro-architecture support. With both micro-architecture and compiler support, our infrastructure can defend against both software and hardware attacks with superb security strength but reasonable hardware and performance cost.

We implement the widely accepted machine model established in [64][38]. Our implementation achieves similar security guarantee as in [64][38], but with a much smaller performance overhead. The details of our implementation can be found in [67]. In that way, our hardware infrastructure can achieve strong process isolation to prevent attacks from other processes and to prevent certain types of hardware attacks, such as using specialized hardware to read/tamper the system data bus traffic and the external

memory system directly. However, the machine model established in [64][38] has a serious flaw. It only protects confidentiality of the traffic going through system data bus. It cannot protect system address bus traffic, i.e., addresses of memory accesses. We show that unprotected memory access sequence acts as very dangerous side-channel and leaks critical control flow information of the protected software. The information leakage could facilitate an attack and bring significant damage to both code and data confidentiality. To our best knowledge, this flaw actually exists in all previous secure processor designs (in both industry and academia) and can be a very real threat. To enhance intrusion prevention capability of our hardware infrastructure, we present a scheme with both innovative hardware modification and extensive compiler support to eliminate most of the information leakage on system address bus with little performance overhead.

Our hardware infrastructure can prevent attacks to software confidentiality and detect attacks to software integrity from other malicious software running in the same machine (even including a malicious operating system) and hardware attacks. However, one important observation is that it cannot prevent attacks exploiting flaws/bugs in the protected software itself. The classical example of this kind of attack is buffer overflow attack [82], which can be regarded as due to a programming error (missing bound checking). Other examples include format string attacks and return-to-libc attacks etc. The hardware infrastructure alone cannot prevent such attacks. As far as the secure processor concerns, the instructions to overflow the buffer are perfectly legal instructions. In fact, to maintain the original program semantics, the buffer should be overflowed. There are numerous software/hardware solutions to buffer overflow attacks, including [23][22][108][6][50][44][89][18][111][27][33][8][83][104][43][131]. However, even

buffer overflow attacks are completely prevented, there will be other new attacks emerging, for example format string attacks [91]. In general, no security system is bullet-proof and is able to prevent all attacks. There may be design/implementation flaws in the underlying protection scheme, or the attacker may exploit the flaws such as buffer overflows in the software to be protected. To be realistic, we have to assume that some attacks will be able to evade the protection scheme implemented. To protect software from those attacks, we build a second line of defense consisted of intrusion detection and intrusion recovery mechanisms, which is able to detect both known and unknown attacks and even recover from those attacks. Previous secure architecture designs are incomplete since such an intrusion detection and recovery scheme is missing.

Our intrusion detection mechanisms are based on anomaly detection. In other words, we try to detect anomalous program behavior caused by attacks based on expected program behavior. Anomaly detection does not target to specific attacks and is able to detect novel or unknown attacks. There have been extensive research on software-based anomaly detection systems, such as [32][115][92][31][30][59][76][37][36] to name a few. However, it is impossible for software-based anomaly detection systems to monitor the software execution at a very fine granularity due to potential huge performance degradation. With micro-architecture level support, our intrusion detection mechanisms can achieve superb monitoring granularity and much stronger detection capability.

In this dissertation, we propose three anomaly detection schemes. The first one is a training based scheme to detect anomalous dynamic program paths. The scheme monitors the software execution at a very fine granularity and is able to detect attacks with high precision and neglectable performance overhead. The major concern of the

scheme is false positives. So we propose the second scheme based on compiler branch correlation analysis to detect infeasible paths without any false positives. But the detection capability is not as good. Finally, we present a third scheme based on compiler data flow analysis to detect tampering to critical software data. The third scheme achieves both zero false positives and strong detection strength, but the performance overhead is significant. We thus devise several compiler optimizations to minimize performance impact due to additional security operations and carefully trade off between security and performance. We demonstrate the effectiveness of our anomaly detection schemes thus the great potential of what compiler and micro-architecture can do collaboratively for software security.

After an intrusion is detected, most previous approaches simply shut down the attacked software to avoid any further damage. However, security implies several important properties including confidentiality, integrity and availability [3]. Availability is an equally important security property to be enforced but gets far less attention. In the sense of software security, availability means recoverability from attacks or intrusion recovery. Simply shutting down the attacked process is unacceptable for software providing critical services. A complete software protection scheme should have the ability to recover from a tampering whenever possible. Thus, intrusion recoverability is an important goal of our infrastructure.

To provide intrusion recoverability, two major tasks have to be done. First, we need to analyze the attack to identify exactly when and how the attack happens so that we can identify the tampered system state and gather useful information for later forensic analysis. Second, the tampered system state has to be recovered through certain

mechanisms. We focus on memory state in this dissertation, since most attacks break into a system by memory tampering. Intrusion analysis is a difficult problem since there could be an arbitrarily large interval between the tampering to the system state and the detection of the tampering. We propose two schemes for intrusion analysis. The execution logging based scheme incurs little performance overhead but has higher demand for storage and memory bandwidth. The external input points tagging based scheme is much more space and memory bandwidth efficient, but leads to significant performance degradation. After intrusion analysis is done and tampered memory state is identified, tampered memory state can be easily recovered through memory updates logging or memory state checkpointing.

In summary, our RADAR infrastructure aims to protect critical software from both hardware attacks and software attacks with strong security guarantee. It integrates the abilities of intrusion prevention, intrusion detection, intrusion analysis and intrusion recovery, which are the major aspects of protecting software from malicious attacks. It emphasizes collaborations between compiler and micro-architecture to achieve those abilities. With micro-architecture level support and compiler assist, our RADAR infrastructure can achieve strong security guarantee for critical software with reasonable hardware cost and performance degradation.

1.5 Contribution Statement

We have an ambitious goal to build an infrastructure to protect critical software from both hardware attacks and software attacks with strong security strength.

We argue that purely software-based approaches cannot achieve strong security strength and hardware support for software protection is necessary. We further point out

that purely hardware-based approaches lack the ability of understanding software semantics. In other words, a semantic gap exists. In this thesis, we take a novel approach to bridge the semantic gap through compiler analysis and hardware runtime support. Compiler provides information about program behavior and attributes to the hardware. Hardware utilizes the information cleverly at runtime to fight against attacks. This approach is seldom explored before in software security area. Moreover, compiler optimizations are devised to minimize performance impact due to additional security operations when necessary. Through our work, we demonstrate the effectiveness and the potential of what compiler and micro-architecture can do in concert for achieving software security strengths and guarantees.

Our infrastructure is a complete solution to the problem of software protection. It integrates the abilities of intrusion prevention, intrusion detection, intrusion analysis and intrusion recovery, which are the major aspects of protecting software from malicious attacks.

We show that unprotected address bus traffic (memory access sequence) acts as very dangerous side-channel and leaks critical control flow information of the protected software. We present a scheme with both innovative hardware modification and extensive compiler support to eliminate most of the information leakage on system address bus with little performance overhead.

We are the first to explore anomaly detection with both micro-architecture and compiler support. We propose three anomaly detection schemes. The first one is a training based scheme to detect anomalous dynamic program paths. The scheme monitors the software execution at a very fine granularity and is able to detect attacks with high

precision and neglectable performance overhead. The major concern of the scheme is false positives. So we propose the second scheme based on compiler branch correlation analysis to detect infeasible paths without any false positives. But the detection capability is not as good. Finally, we present a third scheme based on compiler data flow analysis to detect tampering to critical software data. The third scheme achieves both zero false positives and strong detection strength, but the performance overhead is higher.

We are among the first to explore how to utilize micro-architecture support to analyze then recover program memory state after an attack. We propose two schemes for intrusion analysis. The execution logging based scheme incurs little performance overhead but has higher demand for storage and memory bandwidth. The external input points tagging based scheme is much more space and memory bandwidth efficient, but leads to significant performance degradation. After intrusion analysis is done and tampered memory state is identified, tampered memory state can be easily recovered through memory updates logging or memory state checkpointing.

1.6 Dissertation Organization

In this introduction, we introduce the problem of software protection and discuss why this problem will become more challenging and more critical in the future. We discuss previous solutions to this problem, show their limitations and talk about our approach in a high-level view.

Chapter 2 will discuss the machine model and the attack model used in our work. Our basic machine model is adopted from previous research. We will explain how such a machine model achieves strong process isolation and prevents certain types of hardware attacks.

Chapter 3 will focus on the problem of address bus protection, which is largely ignored previously. We will show that unprotected address bus traffic acts as very dangerous side-channel and leaks critical control flow information of the protected software, then we will present a scheme with both innovative hardware modification and extensive compiler support to eliminate most of the information leakage on system address bus with little performance overhead.

Chapter 4 will discuss our first anomaly detection scheme, which is a training based scheme to detect anomalous dynamic program paths. The scheme monitors the software execution at a very fine granularity and is able to detect attacks with high precision and negligible performance overhead. The major concern of the scheme is false positives.

Chapter 5 will present our second anomaly detection scheme based on static compiler branch correlation analysis. The second scheme is able to detect dynamic infeasible paths without any false positives. The performance overhead is also negligible, but due to the limitation of static compiler analysis, the detection capability is not as good.

Chapter 6 will elaborate our third anomaly detection scheme based on compiler data flow analysis. The third scheme aims to detect tampering to critical software data. It can tackle a large category of attacks left unhandled by control flow monitoring based schemes such as our anomalous path checking and infeasible path detection schemes. The third scheme is able to achieve both zero false positives and strong detection strength, but the performance overhead is much higher.

Chapter 7 focuses on intrusion analysis and intrusion recovery after an attack is detected. We propose two schemes for intrusion analysis. The execution logging based

scheme incurs little performance overhead but has higher demand for storage and memory bandwidth. The external input points tagging based scheme is much more space and memory bandwidth efficient, but leads to significant performance degradation. After intrusion analysis is done and tampered memory state is identified, tampered memory state can be easily recovered through memory updates logging or memory state checkpointing.

In chapter 8, we will summarize the main contributions of this dissertation, talk about broader impact of our work, and discuss possible areas for future work.

2 MACHINE MODEL AND ATTACK MODEL

Before we proceed to present our RADAR infrastructure, we first elaborate our basic machine model and attack model to facilitate the understanding of the work presented in this dissertation.

2.1 Previous Work

Our hardware infrastructure implements a machine model widely accepted in secure architecture area. The machine model is established in [64][38]. It is designed to achieve strong process isolation to prevent attacks from other processes even including operating systems and to prevent certain types of hardware attacks, such as using specialized hardware to read/tamper the system data bus traffic and the external memory system directly to evade process isolation mechanism completely.

Lie et al. proposed the XOM secure architecture design in [64]. XOM stands for "eXecute Only Memory", which means that software in this special memory can only be executed but cannot be read or modified. The XOM secure architecture focuses on defending attacks to memory and assumes that on-chip storage is secure from hardware attacks but off-chip memory and disk are not secure. The XOM secure architecture is based on both cryptographic techniques and micro-architecture level support. From a high level point of view, access control tags are used to protect data inside the trusted hardware of the processor. On the other hand, cryptography is used to protected data that has to be stored off the processor.

XOM tries to implement the notion of compartment and runs the protected software inside the compartment to achieve strong process isolation and to prevent attackers from reading or modifying software code or data. Each compartment has a

XOM id. Each software process is assigned a XOM id which indicates what compartment it is running in. Data from the software operations is tagged with this XOM id. On-chip storage for data and tags are considered properly protected by the hardware from hardware attacks and cannot be attacked in their machine model. So the tags can be trusted and used to achieve access control and protect software code/data when the code/data resides in the on-chip storage. On the other hand, off-chip storage, such as the external memory and disk, is not trusted. Off-chip storage cannot be simply protected by tags since tags may be tampered by attacks. Instead, cryptographic ciphers and hashes are used to protected code/data in the off-chip storage.

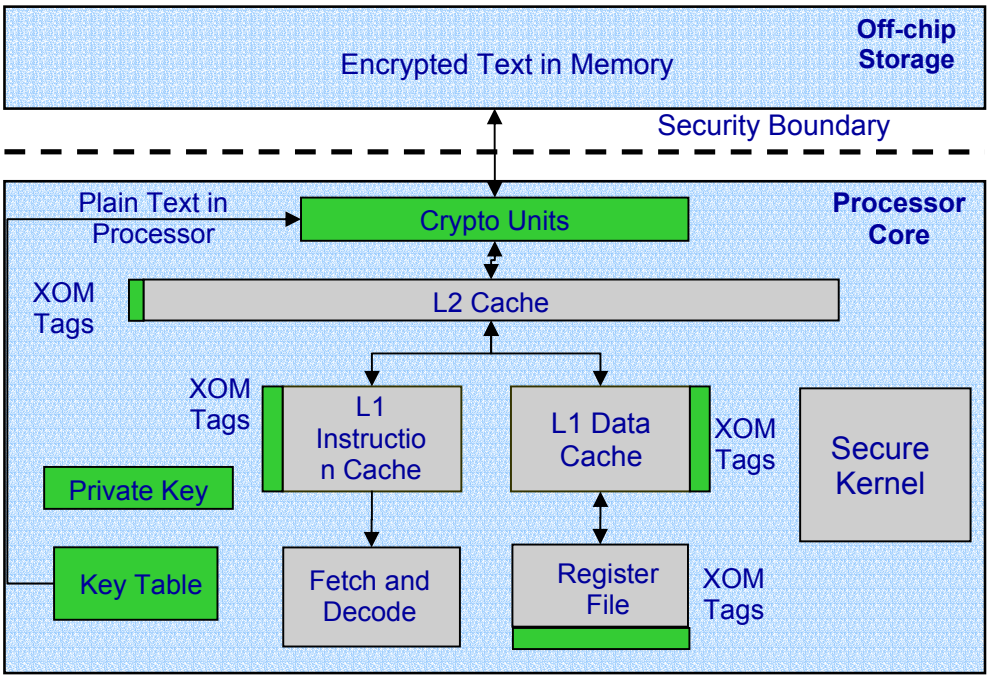


Figure 2. The XOM secure architecture.

Figure 2 shows the XOM secure architecture. The processor possesses the private part of a public/private key pair. The private key is the secret only known to the processor and is the security foundation of the security system built upon the XOM architecture.

Based on this secret, the processor can generate a session key for each protected software process. The session key then becomes the root secret for each protected process and can be used in security operations such as symmetric encryption/decryption and hashing. Session keys for processes are stored in a key table. The key table is indexed by the XOM id associated with each protected process. All data residing in on-chip storage, including register file, L1 instruction cache, L1 data cache and L2 cache, is properly tagged with the XOM id of the data owner. In general, only the data owner has access to the piece of data. Data sharing is handled specially. As mentioned previously, in the XOM architecture, the secure processor is inside the security boundary, so the on-chip data tags are considered protected from attacks and can be securely used to enforce security. On the other hand, the external memory system is outside the security boundary. An owner tag associated with each piece of data is not enough to protect the data when it resides in the external memory since the tag itself can be tampered by the attacker. To protect off-chip data, cryptographic mechanisms are used. To prevent reading attacks or protect confidentiality, data is encrypted when transferring from the secure processor to the external memory and decrypted when transferring from the external memory to the secure processor. The symmetric key used in encryption/decryption is the aforementioned session key for the process. The session key can be found in the key table in the processor using the XOM id associated with the data as index. To detect writing attacks or protect integrity, the XOM architecture calculates a hash for each piece of data when it is transferred to the external memory and verifies the hash when the data is fetched back from the external memory. The protection granularity is a cache block. The cryptographic operations are done by the crypto units inside the processor.

Under the XOM architecture, the general operating system is not trusted and is assumed possibly tampered. The software protection scheme is based on hardware. But to minimize the modification to current micro-architectures without security support, a secure kernel (XOM Virtual Machine Monitor in the XOM architecture) is introduced to implement necessary operations to avoid the complexity of the additional hardware. The secure kernel is a trusted, authorized and privileged program. The secure kernel runs at a privilege level higher than the operating system. It can be implemented in either software or micro-code. Software implementations must be authenticated by a secure booting mechanism such as described in [107][4][62][109]. There are special hardware facilities only accessible to the secure kernel, such as the private key, process key table, secure on-chip memory etc. The secure kernel supports encryption/decryption of software code/data when transferring across the security boundary. It also supports proper tagging of on-chip data, implements special instructions introduced for the security support and handles context switches and certain interrupts.

The XOM architecture aims to prevent three major types of attacks, including spoofing, splicing and replay. In spoofing attacks, the attacker tries to substitute the original cipher text with a faked one to alter software behavior. With encryption, the faked cipher text will decrypt to junk but it will alter software behavior. To prevent spoofing attacks, an integrity hash is attached to the data block. Now if the attacker wants to launch a spoofing attack, in general he has to reverse the encryption and fake the integrity hash, which is very difficult. In splicing attacks, the attacker tries to replace one valid cipher text in one location with another valid cipher text from another location. This is prevented by position-dependent hash. In other words, virtual address of a data block is

involved in the computation of the integrity hash. In replay attacks, the attacker records previous valid values and try to reuse them later. The original XOM design only handles replay attacks using register values.

Later, Gassend et al. [38] pointed out that the original XOM design [64] is vulnerable to replay attacks using memory values. Gassend et al. further proposed a hash-tree [75] based integrity checking scheme to fix the problem. Since then, there has been a lot of follow up work to improve over the initial work [124][102][101][95][67], primarily in terms of performance. But the basic designs established in [64] and [38] are inherited.

2.2 Our Machine Model and Attack Model

In our work, we implement the machine model established in [64] and [38]. When the code and data of the protected software resides in on-chip storage, it is protected in the same way as in [64]. However, when the code and data of the protected software resides off-chip, we design an efficient stream cipher based scheme to prevent attacks to software code/data confidentiality and an efficient MAC tree based scheme to detect attacks to software code/data integrity. Overall, our implementation achieves similar security guarantee as in [64][38], but with a much smaller performance overhead. The details of our implementation can be found in [67].

In summary, our machine model assumes that the secure processor possesses a secret private key. This per-processor private key is the security root of the security system built upon the secure processor. Each software process is associated with a secret key derived from the per-processor key. The per-process key can then be used in cryptographic operations such as encryption/decryption/hashing. We also assume a trusted secure kernel residing in the processor, which is implemented in software and

provides basic services such as context switching and encryption/decryption. Other software (operating system included) is not trusted. Process contexts including registers are encrypted using the per-process key during context switches thus are protected. The security boundary with respect to hardware attacks is drawn around the processor chip. That means: 1) any hardware component inside the processor chip is considered to be secure against hardware attacks; 2) anything else other than the processor itself and the process context is considered non-secure thus is susceptible to hardware attacks, including off-chip caches, system buses and external memory etc. When software code/data resides in off-chip storage, its confidentiality and integrity are properly protected against hardware attacks by cryptographic schemes.

We assume the attacker can launch both hardware attacks and software attacks to the protected software. In terms of hardware attacks, we assume that the attacker can read/write to the external memory directly without going through the secure processor, snoop over the system buses etc., but the secure processor chip is resistant to hardware attacks. The attacker can also launch software attacks. For example, he can create a malicious process to attack the victim software by exploiting the design/implementation flaws of process isolation, or attack through the bugs/flaws in the victim software using buffer overflows, format string attacks etc.

3 PREVENTING INFORMATION LEAKAGE ON ADDRESS BUS

Our hardware infrastructure implements the basic machine model established in [64][38]. However, the machine model established in [64][38] have a serious flaw. It only protects confidentiality of the traffic going through system data bus and ignores protection of system address bus traffic, i.e., addresses of memory accesses. In this chapter, we show that unprotected memory access sequence acts as very dangerous side-channel and leaks critical control flow information of the protected software. The information leakage could facilitate an attack and bring significant damage to both code and data confidentiality. To our best knowledge, this flaw actually exists in all current secure processor designs (in both industry and academia) and can be a very real threat. To enhance intrusion prevention capability of our hardware infrastructure, we present a scheme with both innovative hardware modification and extensive compiler support to eliminate most of the information leakage on system address bus. We show that our solution is both efficient and effective.

3.1 Introduction of the Problem

Although the widely accepted XOM-based machine model is successful in protecting the off-chip code and data using cryptographic techniques, it fails to protect the memory access addresses transmitted on the system address bus. In other words, the memory address sequence generated by an application may be exposed under the XOM machine model. Thus, the following two questions are left unaddressed: 1) Even though off-chip memory contents are encrypted, can the exposed address sequence lead to security breaches? 2) If so, how should we prevent those at a reasonable cost?

Although the problem has been noticed in [64] and [101], they both leave it open. [64] poses it as an open problem, and [101] largely ignores it. In [38] it is shown that the detection of loops through the information leakage on address bus can become a starting point for the replay attack. In this chapter, we point out that address bus protection is critical; otherwise control flow information might be exposed and severe security breaches might occur.

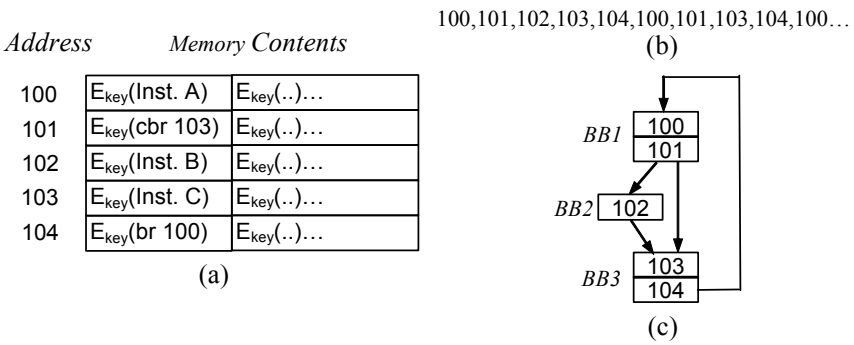


Figure 3. Control flow snooping.

In Figure 3, we illustrate how control flow can be snooped by the attacker through memory access sequence. Under the current XOM-based machine models, all fives blocks of instructions are stored in an encrypted form in memory. However, authentic addresses are readily available on the address bus. The attacker has no idea what the instructions are due to the encryption, however he can snoop on the address bus and obtain a sequence of memory access addresses (refer to Figure 3.b). From the address sequence, he can infer that the code is in a loop since addresses 100,101,103,104 appear repeatedly. He can also infer that there is a conditional branch at address 101 because sometimes the control goes to 103 directly but sometimes it goes to 103 via 102. Therefore, by identifying recurring (block) addresses, the attacker can construct a *block level control flow graph (CFG)* as shown in Figure 3.c. In fact, we found that the leakage

of the control flow can severely jeopardize the encryption of code (it may be possible to crack as much as 70% of the encrypted code through a well devised attack – refer to Section 3.2). Apart from this, address sequence on the bus may lead to exposure of the critical data (such as the secret key) as well.

Regarding the second question about how to prevent the information leakage on the address bus, it may be noted that address bus protection is a much tougher problem than it might first appear. Both industry and academia are aware of the severity of the problem and have proposed solutions. DS5000 and DS5002FP are chips produced by Dallas Semiconductors [26], which are among the most widely used security devices in credit-card terminals, pay-TVs access control systems etc. The processor incorporates bus-encryption (actually, fixed address reordering together with some random accesses) and was described as the most secure processor currently available for commercial users. However, such protection can be easily invalidated (refer to [61] and related work). The only previous solution that completely avoids such information leakage is called Oblivious RAM (ORAM), which was proposed by Goldreich [41][40]. In his papers and patent, three schemes are proposed to ensure that the addresses shown on the address bus are independent of the addresses issued by the application. Unfortunately, all three schemes are infeasible on real machines due to either significant slowdown or resulting memory explosion. Therefore a practical solution for preventing information leakage on address bus is highly desirable and valuable to improve intrusion prevention capability against hardware attacks.

In this chapter, we propose such a solution with negligible performance overhead. We call our scheme as HIDE (Hardware-support for leakage-Immune Dynamic

Execution). Through hardware support and compiler optimizations, HIDE provides a very high level of security guarantee, which means that the information leakage via the address bus is largely prevented. Also, our scheme is highly flexible. It can easily incorporate programmers' specifications of sensitive sections to be protected as well as some compiler optimizations.

3.2 Attacks via Control Flow Snooping on Address Bus

We now illustrate two kinds of attacks that are possible through the control flow information leaked on the address bus to further motivate our work. Note that the example in Figure 3 assumes that there is no "noise" in the addresses seen on the bus (i.e., all the addresses are leaked). But in practice, branches within a block are hidden and a cache can hide many memory accesses. This might lead to less than full leakage but could still be quite damaging. We first assume that there is no noise, and in Section 3.2.3, we will address the noise issues.

3.2.1 Reuse Code Identification

Due to the following two facts, leaking the CFG information can result in the complete exposure of reuse code and severely disrupt code encryption.

The first fact is ***software reuse and binary-level similarity***. With the ever-increasing amount of legacy code and time-to-market pressure, software development relies more and more on reusing existing modules or the pre-built libraries from other companies or from the public domain. For example, many classical algorithms have their standard and/or non-standard open-source implementations available online for reuse. Moreover, most compiler and development tool chains are provided by only a few 3rd party name-brand vendors that can lead to a binary-level similarity once the source code

is reused. In other words, due to reuse of source code and the existence of only a few compilers, the diversity of the generated code at binary level is diminished greatly. We measured the full set of SPEC 2000 integer program binaries targeted to Alpha to find out the percentage of code that is reused from the standard C library on Alpha target. As shown in Figure 4, the reuse percentage can be very high for some benchmarks like mcf (88%) and bzip2 (66%). On an average, 39% of the code at binary level is due to the libraries. A recent study [72] shows that nowadays, up to 70% of the code in industrial software is the reuse code. Given such a high amount of reuse code, the question is: Can it be discovered? The answer is yes. Address bus information leakage allows building a CFG and the **CFG serves as a unique fingerprint** of the underlying code. The attacker can discover the reuse by matching the CFG observed that serves as a fingerprint of the reused code with a repository of well known implementations and thus know which implementation is actually reused.

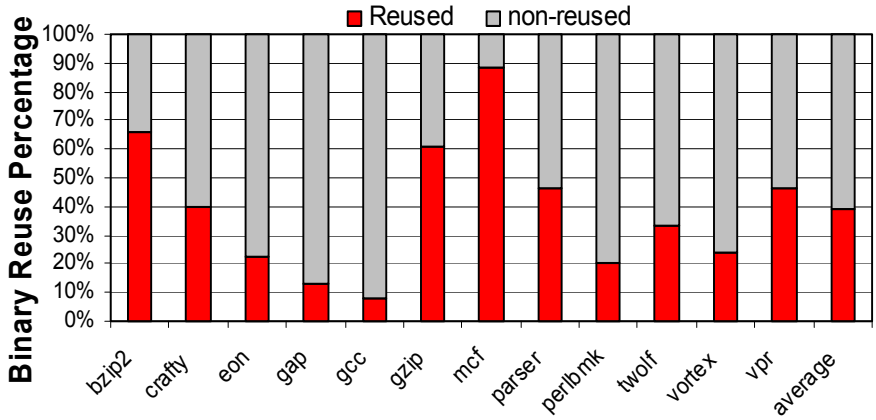


Figure 4. Binary reuse percentage for SPEC2000 integer programs.

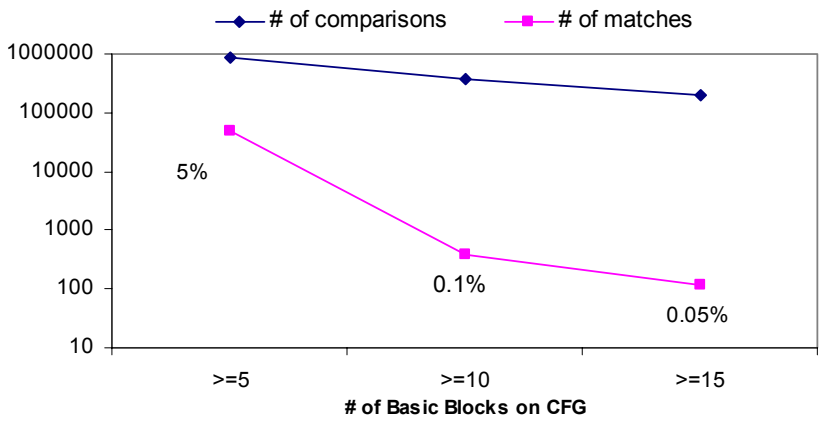


Figure 5. Isomorphic CFG pairs in the standard C library.

In order to measure the uniqueness of CFGs, we did another study and found that they indeed serve as unique fingerprints due to the following intuitive arguments. CFGs are made of basic blocks. It is widely known that the average length of basic blocks is only 6 to 10 instructions for integer programs and a large number of instructions are branches (around 12%). Conceivably, as long as the code is reasonably complex, the chance of two different pieces of code forming the same CFG is slim since they would have a good number of basic blocks and quite a few potential control flow graphs are possible with a given number of basic blocks. As an experiment, we built the CFGs for various block cipher algorithms such as DES, MARS, Rijndael, RC6, and found out their CFGs are significantly different. In Figure 5, we investigate the similarity of CFGs for the procedures in the standard C library of the Alpha compiler. There are 1334 procedures in the standard C library libc.a, with reasonable size (at least 5 basic blocks). We built the CFGs for all those procedures in which each basic block is abstracted as a node (which in fact increases the chances of two CFGs being similar). We run the famous graph isomorphism algorithm by Ullman [110] (we reuse the graph matching library developed by Univ. of Naples [113]) between all possible pairs of CFGs. In Figure 5, the

results show that only 5% of the comparisons find that the two graphs match. If we ignore the CFGs with less than 10 basic blocks, only 0.1% of them match. Finally, if we ignore the CFGs with less than 15 basic blocks, only 0.05% of them match. This study shows that each CFG can serve as a distinct fingerprint for a reasonably-sized piece of code. Therefore, if the programmer reuses a procedure in the library with ten or more basic blocks, the reuse is almost doomed to be found out by the attacker due to its distinctive fingerprint (assuming he can construct the CFG by exploiting address bus information leakage). Note that this estimation is conservative due to our abstraction of the CFGs that ignores sizes of individual basic blocks; otherwise the number of matches would decrease further. Even if some matches occur, the attacker can still narrow down his search to a few possible procedures that might be reused.

Given sufficient amount of time to experiment with the code, most CFG edges could be exposed. Theoretically only dead CFG edges will not be executed and exposed. Even if only partial CFG can be identified with subgraph matching algorithms[110][113], we can still roughly detect the reuses. It is easy to show that the number of possible CFG graphs grows exponentially with the number of basic blocks in the CFG; therefore hiding a large piece of reuse code is almost impossible. From the prior discussion, the CFG, as a matter of fact, can be regarded as an algorithm's fingerprint.

Based on the two facts described above, it is quite possible that an attacker can identify the reuse components in a program given its CFG. He can collect the CFGs of all procedures in the standard libraries, or for publicly available source code, compile them with a name-brand 3rd party compiler and build the CFGs. By graph matching the program's CFGs with his collection, the attacker can nail down the reuse parts. This not

only exposes the reuse code in its entirety, but also helps the attacker in other aspects: 1) A number of plaintext/ciphertext pairs for the reuse code are identified. If the hardware cannot afford integrity check due to its performance and memory space overhead [38], the attacker might construct a program to read out other code such as in [61]. 2) More critically, in some cases critical data could be leaked due to the discovery of re-use code. In the next subsection, we will show how critical data can be found out in some cases. 3) By watching the interactions between the reuse code and the programmer's own code such as the calling sequence, parameters, the attacker can learn more about programmer's own code.

3.2.2 Critical Data Leakage via Value-dependent Conditional Branches

Besides the potential to break code confidentiality, CFG matching can also potentially compromise data confidentiality and leak sensitive data values.

All conditional branches (around 80% among all branches) make comparison between two values and then decide which path to take. Therefore the control flow information can leak important information about the values being compared. The following example assumes that the algorithm used is known beforehand (most security systems assume that the cryptographic algorithms used are known to the attacker) or has been detected by CFG matching. It demonstrates how the critical data (secret key in this case) is revealed.

Example

Diffie-Hellman and RSA private-key operations consist of computing $R = y^x \bmod n$, where the attacker's goal is to find x, the secret key. To show the problem easily, we assume that the implementation uses the simple modular exponentiation algorithm shown

in Figure 6.a, which computes $R = y^x \bmod n$, where x is w bits long. The algorithm is widely used, therefore we can reasonably assume the attacker has identified it through CFG matching.

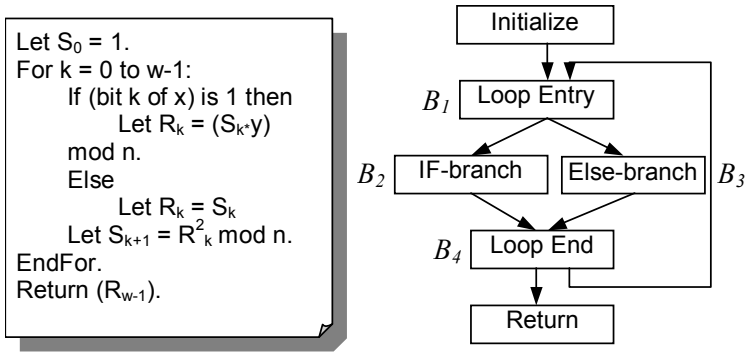


Figure 6. Modular exponentiation algorithm.

The corresponding CFG for this small piece of code is shown in Figure 6.b. From Figure 6.a, we can easily find that inside the loop body if the current examined bit of x is 1, IF-branch is executed, otherwise Else-branch is executed. We assume that IF-branch code resides at address B_2 and Else-branch code resides at address B_3 (B_2 and B_3 are different). The processor must behave as follows: if the current examined bit of x is 1, then fetch the IF-branch code at B_2, otherwise, fetch the Else-branch code at B_3. This results in a sequence of addresses of B_2 or B_3 showing up on the address bus correspondingly. By monitoring the address bus and capturing the addresses transmitted, the attacker can guess whether the respective bits of x are 0's or 1's and get the secret key x. Even if he cannot distinguish between IF-branch and Else-branch, the information on the address bus leaves only two possible values of x to guess (the correct key or its complement).

This example shows that if the conditional branch is known to the attacker, the direction of the branch can expose the outcome of the comparison which might be helpful

in determining or narrowing down the values involved. Such security sensitive conditional branches are widely seen in compression and encryption algorithms. The same piece of code shown in Figure 6 is also vulnerable to timing attack [55], which is however more complicated.

Note that missing several rounds of the for-loop can hide part of the secret key, but still help the attacker substantially narrow down his search space. It is well known that in the security domain, 64-bit encryption has much less strength than 128-bit encryption. If the attacker can capture half of the for loop, his search space will be cut from $2^{|x|}$ to $2^{|x|/2}$, which is $2^{|x|/2}$ times smaller.

As a matter of fact, these kinds of attacks, i.e., circumventing the encryption scheme indirectly through information leakage are well-known in the security domain as *side-channel attacks*. In reality, there have been many successful stories [55][56][34][2] to obtain critical information from a secure chip such as a smartcard, by monitoring the timing [55], power [56] or electromagnetic differences [34] from outside the chip.

3.2.3 Other Issues: Blocking, Caching

The attack described above is a classic side channel attack. Most side-channel attacks have to deal with noise that leads to inaccuracy: Timing attacks suffer from varied computation time of instructions, differential power attacks must count for the power consumption of other components inside the chip. However bus snooping is actually more accurate than other side-channel attacks. It is also very easy to setup [47][61], since the bus is typically exposed outside the processor chip and standard equipments (for testing purposes etc.) are readily available to snoop bus signals. Here we discuss different

types of “noises” that can affect control flow snooping and how the attacker may get around them.

Blocking

Cache accesses and misses are addressed at cache block boundaries; thus, the addresses on the address bus are block addresses. Actually, both attacks we discussed above only rely on the detection of branches. Given that each cache block contains only several instructions (8 on Alpha), it does not affect the attacks much. For reuse code identification, we tried to build block-level CFG, i.e., every block becomes a node and edges indicate possible execution paths between blocks. We found that typically block-level CFGs contain about 25% lesser edges than the regular CFGs. Graph matching the block-level CFG shows results close to those in Figure 5 with negligible differences. To find out how block size affects CFG matching, we list in Table 1 the percentage of matched block level CFGs when the block size equals 32B, 64B and 128B. The results show that larger block size does not affect CFG matching much, especially if we ignore block level CFGs with less than 10 or 15 nodes.

Table 1. Isomorphic block level CFGs with different block sizes.

	32B	64B	128B
≥5	5%	4%	3.3%
≥10	0.1%	0.19%	0.1%
≥15	0.05%	0.04%	0.05%

Caches

Modern processors typically consist of large on-chip caches which might lead to small miss rates and very few addresses exposed on the address bus. However, it does not help due to the following reasons. (1) Since the cache is a shared resource among all

processes running on the processor and our machine model assumes that the OS is not secure. It is very easy for the attacker to manipulate the OS so that the cache gets flushed upon a context switch; alternatively the attacker can ascertain that his own process occupies most cache space before switching to the process being attacked. In this manner, all memory accesses are directly exposed on the address bus due to compulsory misses. (2) Even if only one process is running, many processors have a unified L2 or L3 cache for both code and data. If the program's working set can be affected by inputs, the attacker may intentionally increase the working set size causing more instruction misses. (3) Generally, caching is not predictable, especially in a multi-tasking environment. Different parts of the control flow can be leaked during different runs. It is possible that the attacker can finally get the whole picture after many runs. (4) For low-end systems, on-chip caches are typically small. (5) The cache may be disabled on some machines. (6) An interesting study [84] was conducted by Shamir et. el on how to figure out important information (such as the secret key) leaked through cache misses.

As an experiment, we tried to flush the cache at random moments, and collected 4 block addresses immediately after the flush. After sufficient number of runs, we found that over 95% of edges on the block-level CFG were exposed. In addition, as mentioned before, even if the control flow can be partly masked, information still leaks to some extent since subgraph matching can match partial CFGs. Also, partial execution paths can still be used to prune the searching space for critical data.

The two attacks we showed above are very simple compared to some of the other side-channel attacks that involve sophisticated mathematical and statistical analyses. This indicates address bus information leakage is relatively easier to exploit as well as more

damaging and is harder to prevent. With more advanced analyses, more information leakage could result.

3.2.4 Data Address Protection

Finally, accesses to the data segment can expose control flows as well. For example, in Figure 6.a, if y is accessed, we will know that the If-branch is taken. Therefore, data address protection is equally important. However, this could induce a big overhead since the size of the data segment can be much bigger than the code size.

3.3 Basic Concepts and Components of HIDE

HIDE stands for Hardware-support for leakage-Immune Dynamic Execution. HIDE provides an infrastructure for preventing information leakage on the address bus involving both micro-architecture and compiler support. The basic idea behind HIDE is to "hide" the correlation between recurring memory addresses. This is achieved by permuting the address space and re-encrypting blocks at suitable intervals during the execution.

In this section, we first introduce the background knowledge about what kind of address sequence should appear on the address bus to avoid information leakage. Then we talk about the basic concepts and components of HIDE such as the HIDE cache and the permutation unit.

3.3.1 Fixed Address sequence

To hide the address sequence on the bus, a naïve but completely secure approach is to establish a fixed address sequence that does not change throughout the execution. It

is a foolproof approach that can prevent the attacker from learning anything by monitoring the memory access sequence. This approach is based on the following claim:

If the memory access sequence is always the same regardless of the program execution, no information will be leaked by monitoring the sequence.

Obviously, a fixed address sequence would not tell the attacker anything. To achieve fixed or independent memory access sequence, a naïve way is to read the whole memory segment from the beginning to the end repeatedly. As shown in Figure 7.a, in each round, the whole memory segment is read from beginning to the end once. No matter what is going on inside the processor, the processor always reads memory blocks in this fixed sequence. Apparently, the only thing exposed is the memory segment size, which has been assumed to be insensitive information. However, the naïve approach can lead to a tremendous slowdown. When there is a miss during the fetch request from the cache, the request cannot be satisfied immediately from the memory, but has to wait till the block is read in through the fixed sequence. When the memory segment is large, the processor may have to stall for a long time. For example, in Figure 7.b, if the current reading block is block 70, and the pending miss fetch requests are block 100, 200 and 50, the processor has to wait for 30 reads to get block 100. Again, it has to wait for another 100 reads to get block 200. To get block 50, it needs to wait till the access sequence reaches the end of the memory segment (finish this round of reading), and starts from the beginning again until block 50 is read in. This delay can be enormous given the memory space is big and one round of accesses can take tremendous amount of time. Conceivably, this naïve approach will cause an intolerable performance loss and is not viable at all.

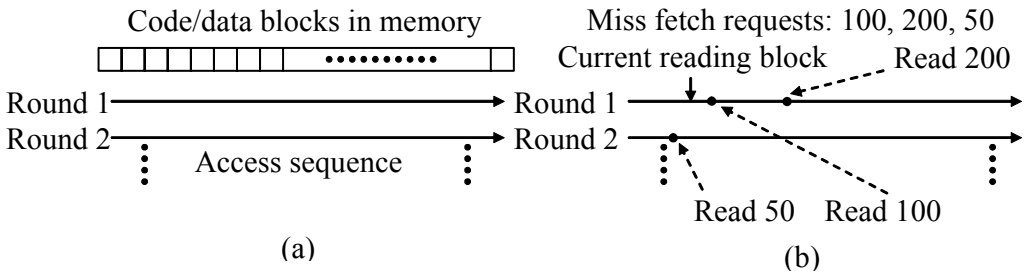


Figure 7. Independent instruction access sequence.

3.3.2 Probabilistically Fixed Address sequence

As introduced in [40] we can alternatively construct an address sequence with addresses conforming to a fixed probabilistic distribution. In other words, although the addresses on the bus do not form a fixed sequence, they are random variables conforming to a fixed distribution. It still exposes no information about which addresses are actually accessed by the processor. Next, we define a few terms about address sequences.

Original Address (Sequence)*:* The address (sequence) issued by the processor.

Actual Address (Sequence)*:* The address (sequence) that actually appears on the address bus.

Probabilistically Fixed Address Sequence*:* A kind of actual address sequence in which actual addresses follow a fixed probabilistic distribution.

Thus, the actual address sequence must be different from the original addresses sequence issued by the processor. Note that the same code/data block must be fetched even if their addresses are different. Thus, changing the addresses means we need to relocate those blocks on the fly. Also, to prevent the attacker to correlate blocks from their ciphertext, blocks are always re-encrypted after relocation. Recent studies indicate

that counter-mode encryption/decryption [124][102][95][96][123] can be applied to achieve low-overhead re-encryption.

There could be many possibilities of constructing the probabilistically fixed address sequences. The following lemma gives one such example that will be used for HIDE.

LEMMA 1: A memory space of size M is randomly permuted repeatedly; assume a block originally at address T is relocated to $P_k(T)$ after the k^{th} permutations. If between the k^{th} and $(k+1)^{th}$ permutation, the processor issues the original addresses T_1, T_2,…$T_{n(k)}$, and these addresses are all different, then the address sequence on the bus is probabilistically fixed.

REMARK: Lemma 1 says that, between two permutations, all original addresses should be different, or we should randomly permute the memory space before the same original address is issued again. Note that since different blocks cannot be permuted to the same location (mapping is one-to-one), all actual addresses between two permutations are different too.

PROOF: Lemma 1 is derived from [40]. We can prove this lemma as follows. Since permutation P_k is completely random and one-to-one, for two different original addresses T_i and T_j , $P_k(T_i)$ and $P_k(T_j)$ are different (two different addresses cannot be permuted to the same place) and are independent random variables. Therefore the addresses in the sequence based on the same permutation P_k, i.e. $P_k(T_1)$, $P_k(T_2)$… $P_k(T_{n(k)})$ are independent to each other. Similarly, since any two permutations P_k and P_l are random and independent, $P_k(T_i)$ and $P_l(T_j)$ are always independently distributed. Thus the addresses on the bus are all independently distributed variables. Note that between two

permutations, original addresses must be distinct; otherwise we will see the same actual address recurring on the bus (this is because the same permutation must permute the same original address to the same actual address). From control flow point of view as mentioned in Figure 3, recurring addresses help the attacker to identify loops and branches. For a better understanding of Lemma 1, we will give an example during the discussion of the *HIDE cache*. □

3.3.3 HIDE cache

To fulfill Lemma 1, intuitively we must "remember" the original addresses that have been issued by the processor after the previous permutation. Before an original address recurs, we need to permute the memory space again. In other words, we must remember the original address sequence to detect recurrence of an address. It is clear that if we "remember" only a small number of original addresses, the memory space must be permuted more frequently. On the other hand, if we "remember" a lot of original addresses, extra space is required to store them and more latency is incurred to check if a new original address has been issued before. These overheads pose the barrier to the deployment of a solution based on Lemma 1 as discussed below.

The *square root algorithm* in the ORAM paper [40] stores all such original addresses in an off chip memory space called *shelter buffer*. The size of the shelter buffer is the square root of the memory space being protected. However, during each access, the processor must read the entire shelter buffer to check if the address has been accessed since last permutation. Since the shelter buffer is in the insecure memory, the processor must read the entire shelter buffer such that the attacker cannot tell whether or where the access has hit in the shelter buffer.

With large on-chip space available on modern processors, we may want to move the shelter buffer on-chip. However, it is still space inefficient to occupy a separate on-chip area for this purpose. Given caches are readily available to store memory blocks accessed before, in this work we propose the *HIDE cache*; which adds address bus protection on top of a normal cache to achieve a low space and performance overhead solution.

***HIDE cache**:* A cache that is same as a normal cache except that blocks fetched after the previous permutation are all locked i.e. they cannot be replaced until the memory space they belong to is permuted again. Also, blocks that are dirty after the previous permutation must be held from the write back until the next permutation.

How the HIDE cache Works

In a HIDE cache, we intentionally lock all blocks that are fetched after the previous permutation. Therefore accesses to the same original address between two permutations always hit in the cache without going out to the external memory. Similarly, blocks that become dirty after the previous permutation are locked as well. If such dirty blocks are allowed to be written back, there could be read accesses to the same address later causing the same address to appear on the bus again; thus, such blocks must be locked as well. If a block is locked, it cannot be evicted. After the memory space is permuted again, all blocks belonging to that memory space are unlocked.

Next we show an example in Figure 8, which helps to understand Lemma 1 and the HIDE cache. In Figure 8.a, we assume that the cache is 2-set 2-way. Figure 8.b shows

the original address sequence borrowed from Figure 3. The memory space contains 5 blocks as shown in Figure 8.c. Note that all blocks are initially permuted randomly after they are loaded from the disk to the memory. The initial random permutation prevents the attacker from correlating information across different runs (since random permutations are done before code and data are initially loaded). We also assume all accesses are read accesses in this example. If blocks are not locked and permuted after the initial permutation, we will observe the cache contents and actual access sequence as illustrated in Figure 8.d. On the left side of Figure 8.d, we show the state of the cache after 4 fetches. All addresses shown inside the blocks are the original addresses. Since the four addresses are all different, they are loaded from memory due to compulsory misses. Also, since blocks are already permuted, the actual address sequence on the bus is 102,100,104,101 instead of 100,101,102,103 for the 4 accesses. If the blocks were not locked (as in normal caches) we will see the address 102 (which corresponds to the original address 100) appearing again. This means that even if the block 100 is randomly permuted to address 102, its recurrence can still be detected on the bus. Thus, a normal cache cannot hide such a recurrence. With the HIDE cache—as shown in Figure 8.e—the 2nd permutation is triggered before the 5th access, because all blocks in the set 0 are locked and we cannot evict a locked block. Note that if a locked block is evicted, we lose tracking of the block. It might be read in again causing the same address to appear on the bus; or if the block is locked because it is dirty, evicting the block will incur a writeback immediately causing re-appearance of its address on the bus. Thus, a permutation ***must*** be conducted to unlock blocks when all the blocks are locked in a set. From the address sequence at the lower right part of Figure 8.e, we can observe that after the 2nd permutation, the original address

100 is now 104. Since the two permutations are random and independent, the attacker only observes random numbers on the bus. Also, recurring original addresses become different random numbers after the 2nd permutation. By locking the blocks, we make sure that the same address does not appear again on the bus *before* another permutation. For example, if we were to access block 104 again, it is in the cache; therefore no access goes out on the bus due to a hit. This scheme not only meets the requirement of Lemma 1, but also preserves the functionality of a cache.

Another observation in Figure 8.e is that in set 1, both blocks are unlocked after the second permutation. Now they behave like normal cache blocks and can be evicted if necessary. Since their mapping and addresses have been changed (their original address is now mapped to a different one) during the second permutation, they can be safely evicted. After the blocks are evicted, they will be locked the next time they are fetched in. Finally, for this example, we discuss the latency costs that might be incurred. When all blocks are locked in a set, we must permute to unlock at least one block before a new block can get in and replace the unlocked one. Since permutation takes a long time, it is not wise to permute at the last minute. We thus permute and unlock ahead of time – before all the entries in a set are locked (this issue will be addressed in detail shortly).

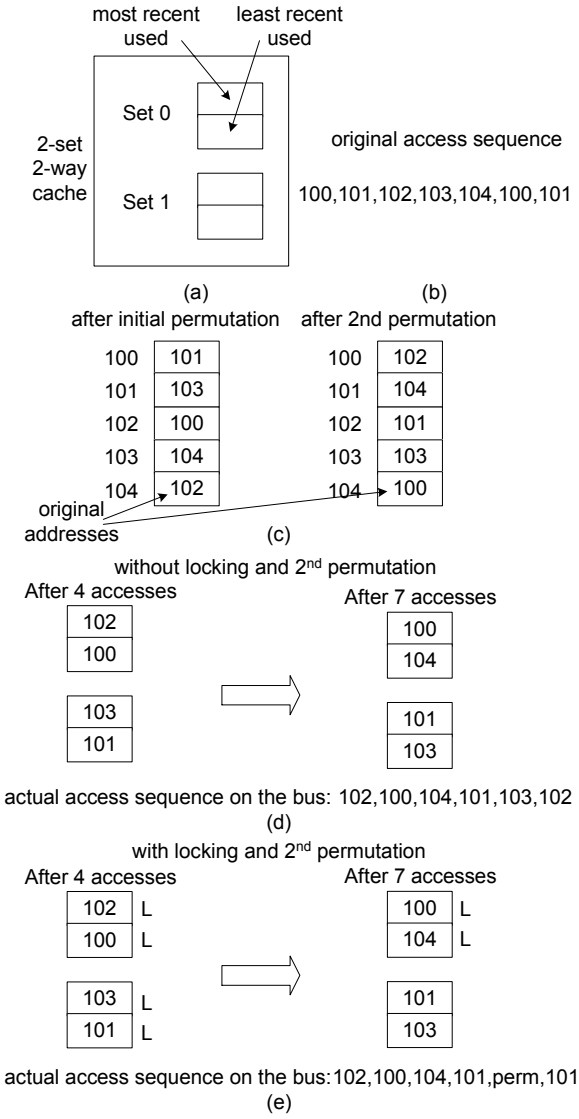


Figure 8. Example for Lemma 1 and HIDE cache.

Implementation of the HIDE cache

We list all HIDE cache operations in Table 2, assuming LRU is the original cache replacement policy. Blocks are still ordered and updated according to the LRU policy, except that when evicting a block, one should choose the least recently used block among the unlocked blocks. Note that the LRU block is not necessarily the last block that is

locked in a set, because a block could be brought in (thus locked) and used frequently, whereas other blocks might be in the set for a long time without being used.

Table 2. HIDE cache operations.

Read and hit	Update LRU order of blocks
Write and hit	Set dirty bit and lock the block, update LRU order of blocks
Cache miss, fetch a new block into the cache	Picked the LRU block among all unlocked blocks in the same set. Fetch and lock the new block. Set dirty bit for write access. If all blocks are locked, put it to fetch buffer.
A memory space is permutated	All blocks belonging to that memory space in cache are unlocked

In our implementation, we use a bitmap (separately stored) to record whether a block is locked or not, i.e. each bit represents one block. After the permutation, the whole bitmap is cleared.

Latency Hiding via Fetch Buffer and Pre-permutation

Prefetch buffer is an on-chip buffer used to store cache blocks temporarily. Since HIDE cache operations could render the cache unavailable in some cases for a short period, cache blocks can be first put in the prefetch buffer and used to serve the application requests and then moved to the HIDE cache once it can be accessed. There are two scenarios in which the fetch buffer is necessary. In one case, the missed block request comes before we find out which block should be replaced; the second case is related to the pre-permutation. Both will be discussed in details later.

Note that locking blocks, i.e. setting the bits in the bitmap, is not on the critical path since it can be conducted after the cache access. Upon cache misses, for each block in the set we need to determine whether it is locked or not. This involves reading several bits in the bitmap. Fortunately it is not on the critical path either. The missed block can be

fetched in parallel when we access the bitmap. Even if the missed block is fetched faster than the bitmap accesses, we can simply put the block in the *fetch buffer* and let the cache access return first. Once we find out all unlocked blocks in the set and the LRU one among them, the block in the fetch buffer can then be moved into the cache to replace the LRU block. In this way, all the operations for the HIDE cache can have the same latency as a normal cache.

However, the performance of a HIDE cache could still be worse than a normal cache because: 1) A locked block may otherwise be replaced when it becomes least recent used. 2) Once all the blocks in a set are locked, a new block cannot be fetched into the set unless we permute and unlock some of the blocks. Due to the long latency of permutation, it may happen that a set is fully locked before the permutation releases a locked block in the set, leading to stalls.

We propose *pre-permutation* to solve the above two problems. Pre-permutation attempts to start permutation before all the blocks are locked. In our design, we start pre-permutation when half of the blocks in a set are locked. Pre-permutation increases the chance that a block is unlocked before it becomes LRU, and greatly reduces the possibility that all the blocks in a set are locked when a new block needs to be fetched in. Even if the pre-permutation does not complete in time, we can put newly fetched blocks in the fetch buffer until the permutation completes and unlocks blocks for replacement. These latency hiding techniques successfully cut down the performance loss as shown in our experiments.

3.3.4 The Permutation Unit

The permutation unit randomly permutes the memory space. It should avoid exposing the correlation between any block's old and new locations. Therefore, blocks are always re-encrypted after relocation. Recent studies indicate that counter-mode encryption/decryption [124][102][95][96][123] can be applied to achieve low-overhead re-encryption. We show the pseudo-code for the permutation unit in Figure 9. A memory space with M blocks is to be permuted using an on-chip space with P blocks called *out_buffer*. In addition, a *permutation vector (pv)* consisting of a random permutation of numbers from 1 to M is generated and stored on-chip.

CASE I: If M is less than or equal to P, we only need to sequentially read the M blocks once. After reading in block number s, we put it to *out_buffer*[*pv*[s]]. Finally, *out_buffer* is written out sequentially to the original memory space. Since the attacker only sees one sequential read and one sequential write to all blocks and everything is re-encrypted, he cannot build any correlation between blocks' old and new locations.

CASE II: If M is larger than P, without loss of generality, let $M=k*P$, where k is an integer larger than 1. We split the M block memory into k equal-sized partitions. During an iteration s, all blocks destined to partition s are permuted and put in the *out_buffer*. At the end of iteration s, the *out_buffer* is written out to a temporary memory space as a new partition s. Upon finishing all permutations, we overwrite the original memory space with blocks in the temporary memory space. Overall, the M blocks are read k+1 times and written 2 times. Alternatively, we can change the page table such that the temporary memory space is mapped to the original memory space, which saves copying from *temp_mem* to *mem*.

```
DATA STRUCTURE:
//memory space to be permuted
block  mem[1..M]
//temporary space in memory
block  temp_mem[1..M]
//permutation vector, on-chip
Int      pv[1..M]
//output buffer, on-chip
block out_buffer[1..P]
```

```
M≤P
pv[1..M]⇐a random permutation of numbers from 1 to M;
for s=1 to M do
  out_buffer[pv[s]] ⇐mem[s]
endfor
mem[1..M] ⇐re-encrypt(out_buffer[1..M])
```

```
M=k*P
pv[1..M] ⇐a random permutation of numbers from 1 to M;
for s=0 to k-1 do
  for t=1 to M do
    read mem[t];
    if s*P<pv[t] ≤(s+1)*P then out_buffer[pv[t]-s*P] ⇐mem[t]
  endfor
  temp_mem[(s*P+1)..(s+1)*P] ⇐re-encrypt(out_buffer[1..P])
endfor
mem[1..M] ⇐temp_mem[1..M]
```

Figure 9. Pseudo-code for the permutation unit.

Although the size of the *out_buffer*, i.e. *P,* cannot be very large, the permutation unit can permute a very large memory space, i.e. *M* can be large. In this algorithm, the size of *pv* still depends on *M*, however *pv* is actually very small because it only stores a small integer for each block (8 bits for 8K pages with 32B block).

The pseudo-code in Figure 9 does not show the algorithm to generate a random permutation of numbers from 1 to *M*. We follow the shuffle algorithm in [54], which requires *M* swaps to generate a complete random permutation as long as a hardware-based true random number generator [49] is available. Finally the time for random permutation generation can be completely masked when the permutation unit reads the memory space.

There might be pending accesses to the memory space being permuted. If the block has not been read-in, we can still issue the access to memory (Lemma 1 is still enforced, because we do not unlock until the permutation finishes). If the block is read in and in the out_buffer, it is fetched from the out_buffer immediately. If the block has been written out, we can fetch it from the memory, but need to mark that it will remain locked after the permutation.

Overhead Analysis

With pre-permutation, permutation is normally not on the critical path; critical reads by the processor are always given higher priority than the permutation traffic. Note that although permutation is the main source of bus traffic increase, such traffic is very regular and predictable and therefore can be easily pipelined. In addition, we can take advantage of memory banking and parallelizing memory accesses to different banks. If a normal access is going to access the chunk that is being permutated (this should rarely happen, since a chunk only takes a small portion of the address space), we can first try to locate it in the out_buffer if that block has been read into the permutation chip. To amortize the initial overhead for each bus transaction (which could take the majority of the access time if only a small number of bytes are transferred in each transaction), we read/write many consecutively located memory blocks during each transaction. Finally, we can completely offload the permutation traffic from the front-side bus with a separate permutation chip (or it can be combined with the memory controller). Communication occurs between the processor chip and the permutation chip, e.g. the processor chip should send/receive data to the permutation chip if the address falls in the chunk that is being permuted. Also the permutation chip returns the new mapping once a permutation

has finished. Obviously, such communication should be encrypted to be immune from bus tapping and the amount of bus traffic will be much smaller. Finally, as mentioned earlier, re-encrypting blocks is necessary to avoid block correlation based on block cipher texts. This overhead has become negligible with recent studies on counter-mode encryption/ decryption.

We now show how we put together the HIDE cache and the permutation unit to offer chunk level protection.

3.4 HIDE at Chunk-Level

This section gives the hardware infrastructure of HIDE, which provides so-called chunk-level protection. The hardware interface is supplied as simple instructions.

3.4.1 Chuck Level Protection and Transition Coverage

A chunk is defined as one or more pages that are protected and permuted together. Protecting a large piece of memory is prohibitive due to the high permutation cost, especially. when the out_buffer cannot hold the entire piece of the memory space, we have to access the memory space multiple times. The goal of chunk level protection is to limit the size and the cost of the permutation. With chunk-level protection, the permutation unit only permutes blocks within a chunk. Once a chunk is permuted, all cache blocks in that chunk are unlocked.

We can split an address sequence into a series of transitions from address to address, e.g. the address sequence in Figure 3 has transitions like: 100→101, 101→102, 102→103…. If the transition is between two addresses in the same chunk, we call it intra-chunk transition, otherwise it is inter-chunk transition. Since all intra-chunk transitions are protected with chunk-level protection, the percentage of intra-chunk

transitions among all transitions is called transition coverage, which is a good indication of how well the address sequence is protected and the level of security guarantee we can provide.

We found that chunk-level protection is powerful. As observed from our experiments, even with the smallest chunk size, i.e. a page, over 75% of the transitions are intra-chunk. Given that not all memory contents are security sensitive, protecting chunks of a reasonable size should suffice if we can slightly narrow down the protection domain with either compiler analyses or user specifications as shown in Section 3.5. Besides, chunk-level protection is flexible. Since our infrastructure supports chunks with different sizes, the user can choose to protect some memory space in big chunks and some in small chunks. Building chunks on top of pages facilitates the implementation, since pages are supported by both the hardware and OS.

3.4.2 Hardware Components

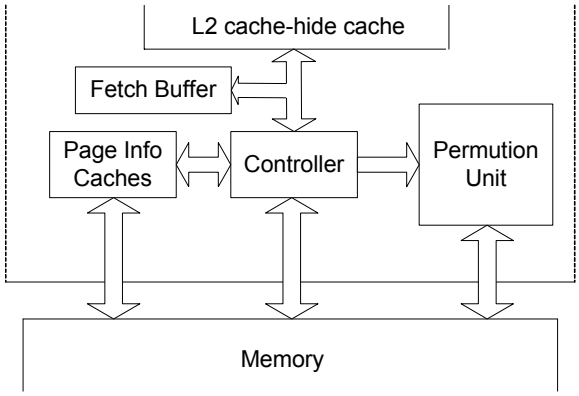


Figure 10. Hardware flowgraph.

Figure 10 shows the hardware structure to provide chunk level protection. The hide cache is implemented at the L2 level (in other words, L2 is a HIDE cache). The

fetch buffer and the permutation unit were introduced in Sections 0 and 3.3.4. The controller coordinates all components.

Before addressing the page info cache, we first describe the page info record in Figure 11, which contains 7 fields for each page. Page info records store extra information for the L2 cache so that it can function as a hide cache. As shown below, the if_hide field indicates whether the given page should be protected. Blocks in a protected page are permuted together with other pages in the same chunk and accesses must go through the address translation to reach the new locations. The second field is the page number in the virtual address space. The next two fields specify the chunk this page belongs to. Note that each chunk must take a contiguous piece of memory in the virtual address space – this is done to facilitate compiler optimization and user specification. In other words, a chunk must take several consecutive pages in the virtual (also physical) address space; begin_virtual_page# and chunk_size (number of pages in a chunk) uniquely define a chunk.

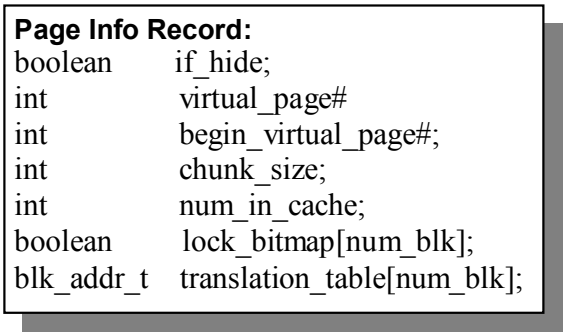


Figure 11. Data structure for page info record.

num_in_cache counts the number of blocks of the chunk that are currently locked in the *hide cache*. For chunks with multiple pages, only the first page's page info record stores such information. This field is used for pre-permutation described in Section 3.3.3.

When half of the blocks in a set are locked, for each locked block we find out the percentage of locked blocks in their chunks. The one with the highest percentage is chosen to be permuted. The *lock_bitmap* field contains *num_blk* bits, where *num_blk* is the number of blocks in a page. Each bit indicates if the block is in the cache and is locked. Finally, the translation table translates each block to its new block address after a permutation. The translation table is updated by the permutation unit after each permutation. For chunks with multiple pages, the *blk_addr_t* includes a *page_ID* $\in$[0,*chunk_size*) to indicate which page in the chunk this block is permuted to i.e. *begin_virtual_page#* +*page_ID*.

The size of the page info record is small compared with the size of a page. For 8KB page size on Alpha, it only adds 3.5% memory space overhead. However, for big chunks, the translation table takes more space due to the bigger *page_ID* field.

Page info records are stored separately in a reserved memory space so that they can be accessed only by the hardware. There is one page info record for each physical page. For each block, the hardware can find the corresponding page info record via its physical page number. To speed up the access to the page info records, we put a page info cache on chip. Due to the small size of the page info records, a cache of 8~16KB is typically enough to achieve a very high hit rate. However, since the page info records are stored in the external memory, accesses to it may leak information on the address bus too although the records are encrypted under our machine model. Conceivably, such information leakage is indirect and also very limited due to the small size of the page info records. For complete security, we can build several layers of protection, i.e., the first layer page info cache becomes a hide cache which protects the first layer page info

records for the pages used by the program. Since the first layer page info cache is a hide cache, we need second layer page info records for this hide cache and for the pages taken by the first layer page info records. Obviously, the sizes of the page info cache and page info records decrease exponentially. At layer 3, the page info records are typically small enough to be stored on-chip.

3.4.3 Some Other Issues

Since chunks are defined in the virtual space, actual accesses to the physical memory must go through the page table and TLB look up. Recall that the OS could be malicious, therefore both the page table and the TLB could be manipulated by the OS. In fact, this problem has been solved by integrity checking the application's virtual address space.

Moreover, TLB only provides page level translation, thus a malicious OS might be able to figure out intra-trunk page-level transitions by observing the virtual page addresses sent to the TLB (e.g. by flushing the TLB and trapping the TLB misses). One possible solution is to use large pages. Large pages are available on some platforms like IBM Power series. However, typically very few sizes are available. For chunk level protection, it is desirable that a variety of chunk sizes can be supported. To prevent information leakage through the TLB, we propose a simple approach without changing the existing TLB architecture. When a virtual page address needs to be converted, we always use the beginning virtual page address of the chunk that the page belongs to. Figure 12 illustrates how the translation is done. Each chunk takes a continuous memory space in the virtual and in the physical memory address space. To translate any page

inside a chunk, the page address of the first page in the chunk is sent to the TLB. After getting the physical address of the first page, the offset is added to get the final address.

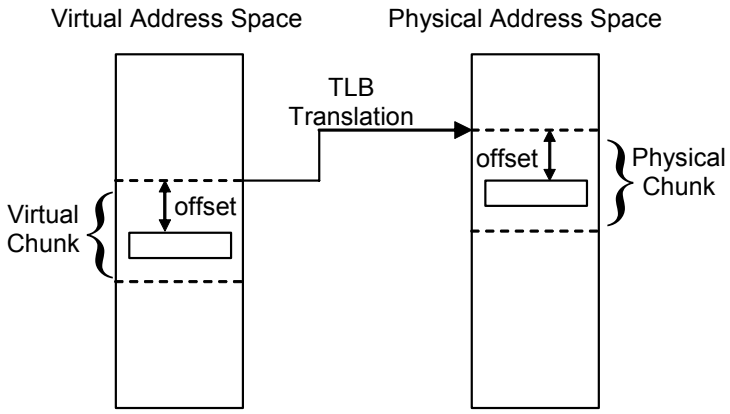


Figure 12. Virtual to physical address translation.

The OS should guarantee that all pages for a chunk are stored continuously in the physical address space, otherwise it will fail the integrity checking and get detected. Also, all pages in a chunk should be permuted together, therefore it is better they are swapped in and out together, such that extra delay could be avoided. In our implementation, these suggestions are conveyed to the OS. Although the OS could be malicious and does not comply with these suggestions, it only leads to worse performance but no security breaches.

Finally context switches should not affect the hide cache, since it works at the physical page level, i.e., page info records are designed for physical pages and are addressed with physical page numbers. It works in tandem with a L2 hide cache where all the addresses are physical addresses.

3.4.4 Interface to the Application

Chunks can be specified statically in the program code or at runtime with special instructions. In the former case, such information is inserted in the header of the program

binary telling which chunks should be created initially. Upon loading a chunk, the hardware initializes the fields in the page info record and performs an initial permutation before loading it into the memory. The initial permutation prevents the attacker from gaining information across different runs of the program. At run time, we can use three instructions to manage chunks; their syntax and operational semantics are as follows:

♦*hide_chunk (begin_virtual_page#, chunk_size)*

Operational semantics: For all pages in the chunk, set the *if_hide* and other fields accordingly. Get a new translation table from the permutation unit without performing real permutation and clear the lock bitmap. Later accesses to the chunk must go through the address translation. This instruction can be used when a new memory space is allocated. Since all old contents will be overwritten, no real permutation is needed.

♦*unhide_chunk (begin_virtual_page#)*

Operational semantics: Clear the *if_hide* fields of all pages in the chunk so that the later accesses will go to the external memory directly. The old contents are discarded.

♦*unlock_block (virtual_page#, start_block_num, num_block)*

Operational semantics: Clear *num_block* consecutive bits in the *lock_bitmap* of the virtual page, starting from *start_block_num*.

Next we discuss some optimizations developed to boost the level of security guarantee offered as well as the performance.

3.5 Optimizations

As mentioned in Section 3.4, chunk-level protection still exposes inter-chunk transitions since permutations are confined to the blocks within a chunk. This section talks about compiler and runtime techniques to achieve a higher level of security

guarantee based on the hardware infrastructure and through the interface proposed earlier. Through compiler analyses, user specifications and runtime support, we can effectively improve transition coverage and reduce address bus leakage, especially for sensitive contents, with very small overhead.

3.5.1 Compiler Layout Optimization

Compiler layout optimization aims to properly layout code or data to chunks such that transitions are covered with small permutation cost. There are a number of issues we should consider during the compiler layout optimization.

1) Intuitively, we should put functions that frequently call each other in the same chunk. Similarly, consecutive data accesses that are frequent should be put in the same chunk as well.
2) Adding more functions or data to the same chunk tends to increase the chunk size. However a big chunk is expensive to permute. Therefore, we must factor in the tradeoff between transition coverage and the permutation cost.

In our compiler optimization, we attempt to properly layout code and static data to minimize inter-chunk transitions. This approach is also applicable to heap space that is redistributed to the program – refer to Section 3.5.2. For static data, we assume that arrays and structures are laid out as a whole. For code, functions are laid out as a whole. Occasionally, we may encounter large functions [1] or arrays, which can cause big overheads if they have to be covered with large sized chunks. In such cases we hope that the compiler or the programmer is able to divide them into smaller pieces that are

[1] In our benchmarks, we only observe several functions that exceed page size.

unlikely to transit frequently from one to another. Next we give definitions of *Transition Graph and Permutation Cost.*

Transition Graph: undirected graph with weighted nodes and edges.

P-Cost(m): *Permutation Cost* for a memory space of size *m*. It should monotonously increase with the size of the memory space.

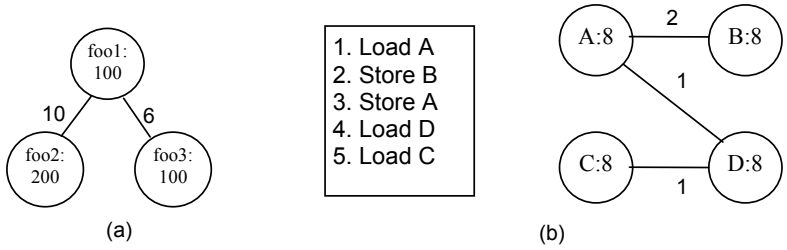


Figure 13. Examples for transition graph.

On the transition graph, each node represents a function. The weight of the node is the size of the function. Edge weights between nodes represent the call/return frequency between the two functions. If function A calls B or returns from B once, the weight of the edge between node A and B is increased by 1. For the example in Figure 13.a, the transition graph tells us that the function foo1 calls or returns from foo2 10 times, whereas foo1 calls or returns from foo3 6 times. The sizes of the 3 functions are 100,200 and 100 respectively. For static data, each node represents a *data unit*, i.e. a scalar variable, a structure or an array, etc. The weight of the node is the size of the data unit. The weight of the edge is the number of times the two data units are accessed consecutively. For example, if we find a path on which data unit A is accessed immediately after data unit B, the edge weight between them is incremented by 1. An example is shown in Figure 13.b, 4 data units are loaded or stored as in the code segment.

The edge between A and B gets weight 2 because B is accessed right after A once and A is also accessed right after B once. Similarly, D is accessed immediately after A and followed by C. The sizes of these variables are all 8 bytes. The transition graph is a rough estimation of how frequently nodes transit from one to another, because some transitions can be hidden by the cache.

For a node n, Let $P\_Cost(n)$ be the permutation cost of the memory with all data units in the node.
$P\_Cost(n) \propto$ size of the minimal chunk that can hold n.

Let $G(N,E)$ be the transition graph with node set N and edge set E.
Let W be the total edge weights of the original graph (before node merging)

LAYOUT ALGORITHM

1. $C = \sum_{n \in N} P\_Cost(n)$
2. *while* (total edge weight of G)/W>5% *do*
3. For any two nodes n_1 and n_2 find the minimal value of ($P\_Cost(n_1+$
4. n_2) - $P\_Cost(n_1)$- $P\_Cost(n_1)$)/(edge weight between n_1 and n_2).
5. merge n_1 and n_2.
6. *od*
7. return the transformed graph

Figure 14. Algorithm to layout code and data.

Figure 14 shows the code and data layout algorithm. We start with the transition graph assuming each node will be assigned to a separate chunk. Then we merge chunks into bigger one if it is beneficial. Our goal is to get minimal total *chunk cost* to *cover* most edge weights. At this point, we need to clarify two things: 1) The total chunk cost is the sum of the costs of all the chunks and a chunk's cost is empirically calculated to be proportional to its size, since the time to permute grows with the chunk size. 2) If two nodes are in the same chunk, then the edge weight between them is covered.

Initially, each node is assigned to a distinct chunk with minimal size, i.e. a single page. Empirically we loop until over 95% of edge weights are covered. During each

iteration, we find a pair of chunks to merge with minimal ratio increase (ratio is defined to be the total chunk cost divided by the covered edge weights). Note that if we merge two chunks, all nodes in the two chunks are now assigned to a new chunk that can hold all nodes and its size is minimal.

3.5.2 Managing Stack and Heap

Stack and heap are dynamically managed memory spaces that must be tackled at runtime. We now discuss several optimizations for protecting stack and heap accesses.

Since stack size is typically small and most activities occur at the top of the stack, we should always try to put the top of the stack inside a chunk boundary. To avoid the exposure of accesses to the top of the stack, the application checks if the callee's frame will cross chunk boundary. If so, it sets the frame pointer to the boundary of the next chunk. Figure 15 shows that when lesser space is available in a chunk than the stack frame size of the callee function, we allocate the callee's stack frame at the start of the next chunk. In this way, we can avoid callee's stack frame from crossing the chunk boundary which might lead to information leakage. Since the parameters have to be passed from the caller to the callee, there might still be some inter-chunk leakage. We suggest parameters are put into the callee's chunk. Although this might still cause several inter-chunk transitions at the beginning of a function call, we believe that the amount of leakage introduced is minor. On the other hand, the accesses within a stack frame are more frequent and thus, we choose the above solution. In cases where there are accesses across stack frames (such as indirect references through a pointer to caller's data etc.), we attempt to put those two functions' stack frames in the same chunk.

Since the location and the size of the chunk as well as the end of the stack frame are all known at runtime, the compiler simply inserts comparison instructions at the function call site and advances the stack frame accordingly. The new chunk is protected with the *hide_chunk* instruction. Given that the stack frame size is typically small, space loss due to this is negligible compared with all memory space taken by a program.

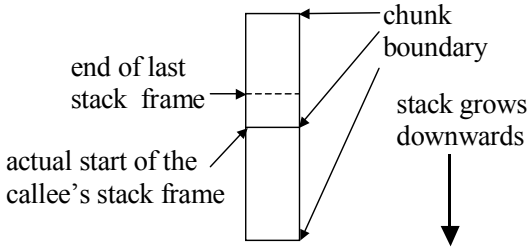


Figure 15. Protecting stack space under HIDE.

On the other hand, heaps can be large; therefore more specifications by the programmer are desirable. Next we give the *unlock rule*, which has been found very useful to reduce the number of locked blocks and save unnecessary permutations.

Unlock Rule: If a block will not be referenced after a certain point and it is not dirty in the cache, we can directly unlock it.

If a block will no longer be referenced and is not dirty, there is no need to lock it in cache, because its addresses will not appear on the address bus any more. The unlock rule suggests that data can be unlocked in the cache as soon as we are sure it will not be referenced later. Especially when only a few blocks in a chunk are locked in the cache, it will be inefficient to perform a permutation of the entire chunk. Thus, unlocking those blocks without incurring a permutation is most desired in this situation.

To unlock a set of blocks, we can use the *unlock_block* instruction. This instruction simply clears the lock bits for those blocks if they are not dirty. *unlock_block* instruction can be inserted by the compiler or the programmer after data values are determined to be dead through offline analysis. Some heap allocation schemes grab large pieces of memory from the heap then redistribute them to the application. Thus, techniques presented in Section 3.5.1 might be similarly applied to minimize inter-chunk transitions during the above heap allocations.

3.5.3 Adaptive Chunking

Chunks can also be formed adaptively. If transitions frequently happen between two adjacent chunks, it is probably better to merge them together. Although we pay more cost to permute them together, it achieves better coverage. On the other hand, if two adjacent chunks are rarely accessed consecutively, merging them is not beneficial at all. As mentioned earlier, for ease of implementation we require chunks to occupy a contiguous piece of memory in the address space. Thus, we only need to track the transitions between adjacent chunks. To facilitate adaptive chunking, we add *transition counters* for each chunk to keep track of the transitions between adjacent chunks. Those counters monitor the number of transitions for pairs of chunks that could potentially be merged. For each chunk, there is a transition counter which records the number of transitions from or to the next adjacent chunk in the virtual address space. Note that the counters keep increasing as time goes by. For a fair measurement, we also record the time-stamp when the counter was reset last time. Empirically if *Trans_Counter*/(*TS_now*-*TS_last*) is above certain threshold *merge_threshold*, it means the two chunks should be merged. Periodically we reset the counters and update *TS_last* so that the counters keep

track of recent transition activities. To reduce overhead, we only check against the *merge_threshold* after *TS_now*-*TS_last* is large enough (more than 1 million cycles) and the transition counter is being updated. In other words, after we get enough statistics (from *TS_last* to *TS_now*), there is a check point to decide if a merger is needed. If not, both *Trans_counter* and *TS_last* are reset. The checking is done in a lazy manner. It is designed so because we do not want to eagerly merge chunks, which definitely leads to higher cost. Therefore, only in cases a really good candidate is identified, should we perform the merger. Once we decide a merger, the corresponding page info tables of the underlining pages are updated to reflect the change.

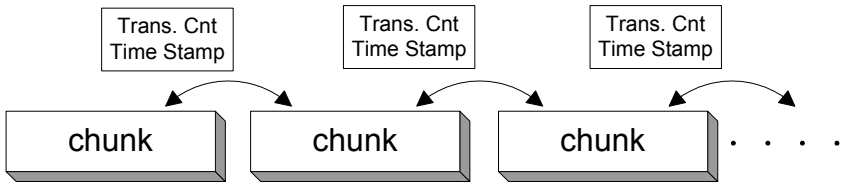


Figure 16. Illustration of adaptive chunking.

As page info table, the transition counters and time stamps are put in a reserved memory space that can only be accessed by the hardware. The space overhead is considered very small, as the information is only maintained at chunk level. To update the counters, the hardware simply keeps track of the previous chunk that was accessed, if the current chunk is the same as the previous one, or the two chunks are not adjacent to each other, we do not need to do anything, otherwise the transition counter of the previous chunk is updated to reflect this transition. In other words, the updates only happen during some of the chunk level transitions, which actually happen rarely as most of the transitions are covered inside chunks (we will show this in the results section). Meanwhile, updating the counters can be done in parallel with chunk accesses and permutation, further reducing the overhead. When a merger is decided, extra latency is

incurred to update corresponding page info records. However chunk mergers do not happen frequently, especially when we do it lazily. In this manner, the runtime overhead for the involved operations is almost negligible.

3.6 Other Considerations

Information leakage prevention is a broad topic, whereas here we only tackle a particular problem. For instance, system calls can somewhat leak control flow information, however the interactions between the application and the operating system are unavoidable. This problem is actually left to the programmer to not to put system calls at sensitive sections of the code. Also, execution time cannot be hidden due to its tight association with performance. It is normally unreasonable to require the program to run for the same amount of time regardless of inputs. Under our scheme, the attacker observing the address bus still gains some information, such as the moments when permutations take place, the number of accesses between two permutations, etc. However such leakage is much less than that from an unprotected address bus. So far we cannot conceive of any feasible attacks that might benefit from this type of information leakage.

In a multi-processor system, one block may be present in the caches of other processors, therefore locking and permutation information must be shared and made consistent across multiple processors. However, the communications among processors are transmitted through the bus that is subject to attacks. Therefore our scheme cannot work without extensions. Currently, we regard address bus information leakage prevention in a multiprocessor environment as our future work.

3.7 Evaluation

We evaluate our scheme on a processor model with default parameters shown in Table 3, in which all 8K chunks are protected. The entire SPEC2000 integer benchmark suite is used as representative applications. The standard reference input is used for each benchmark. Implementation is done with the Simplescalar toolset [11] and experiments are based on SimPoint [94]. Each benchmark is fast-forwarded according to SimPoint then simulated by 100M instructions. The 200M Memory bus is 8B wide and fully pipelined. The permutation latency of a chunk is not a constant number. It has three major components as discussed in section 3.3.4. The first component is the latency to sequentially read the blocks inside a chunk and the latency to decrypt the blocks. With a counter mode based encryption/decryption scheme, the decryption latency can be largely hidden. The second component is the latency to place the block read in to a proper location in the temporary output buffer according to the permutation vector. The latency of this step is small. The third component is the latency to re-encrypt and write out the output buffer. Again, the encryption latency can be largely hidden.

Table 3. Default architectural parameters.

Clock frequency	1 GHz	Unified L2	4way, 32B block 1M (12 cycles)
Fetch queue	32 entries	Memory latency	80(1^{st}), 5(inter) cycles
Decode/issue/ commit width	8/8/8	Chunk size	8 K, all chunks protected
RUU/LSQ size	128/64	Fetch buffer Page info caches	8 blocks 8K/1K
TLB miss	30 cycle	Permutator Outbuf	64K
L1 I/D	8K DM 1 cycle 32B block	Encryption Latency	40 cycles
Memory bus	200M, 8 Byte wide	Encryption Throughput	3.2GB/s

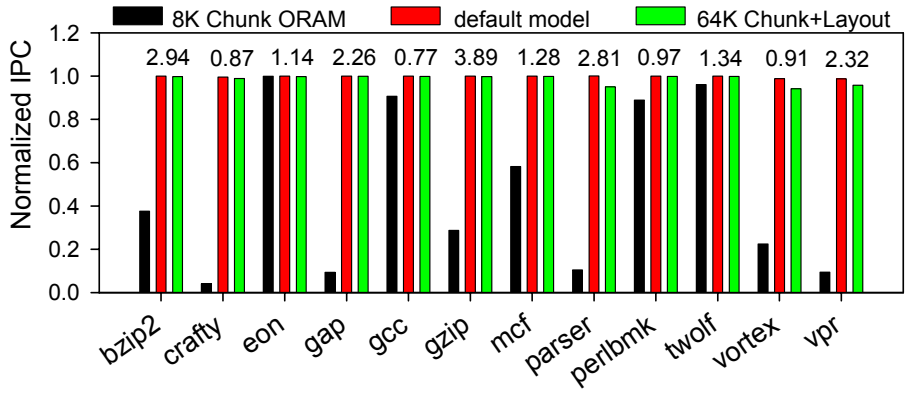


Figure 17. IPC results normalized to the baseline without address bus protection.

As a summary, first we compare IPC, bandwidth usage and transition coverage (i.e. percentage of intra-chunk transitions—Section 3.4) for three models: 1) the one with shelter buffer in the ORAM paper [41][40]; 2) default model implemented as a HIDE cache with parameters as listed in Table 3, 8KB chunk, no layout optimizations; 3) a more secure model with 64KB chunk and compiler layout optimizations described in Section 3.5. Note that, the ORAM work was theoretical therefore no implementation details were provided. We directly implemented their scheme. Further improvements might be possible but should be accompanied by a careful study from the security aspects.

Figure 17 shows performance results. For comparison, we normalize all IPC numbers to the baseline without address bus protection. We also list absolute values of the baseline IPC values. The ORAM model incurs significant slowdown (44.2%), although we only implemented it at the smallest chunk, i.e. single-page level with 8-block shelter buffer for a fair comparison with the default model. Both default model and the "64K chunk+layout" model shows little slowdown: 0.3% and 1.5% on average. Comparing with a normal cache, our HIDE cache has to permutate a memory chunk when necessary thus resulting a larger average L2 cache access latency and a lower

performance. Figure 18 shows the average L2 cache access latency under the baseline without address bus protection and the default model of our HIDE scheme. The hit latency of the L2 cache is 12 cycles. Without address bus protection, the average access latency of L2 cache over all benchmarks is 13.27 cycles. Under HIDE default model, the average access latency of L2 cache over all benchmarks is 15.43 cycles.

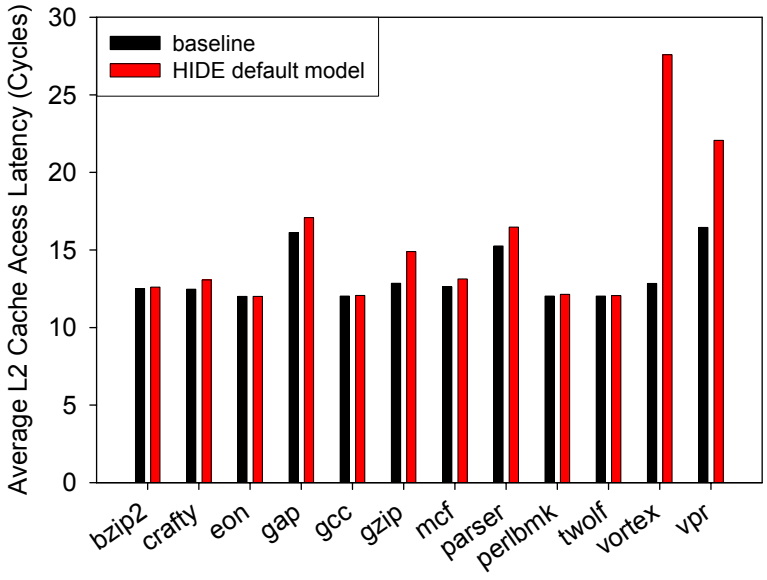


Figure 18. Average L2 cache access latency under the baseline and the HIDE default model.

Next, we look at the bandwidth usage. We show the percentage of overall bandwidth (i.e. 1.6GB/s) being taken in Figure 19. ORAM uses over 60% of the bandwidth due to its scanning of the whole shelter buffer during each access, the default model and 64K chunk one only use 9% and 15% of the bandwidth. For most benchmarks in SPEC2K, the memory traffic is not a big issue since they take only about 5% of the bandwidth to begin with. To get an idea of the worst-case bandwidth consumption, we reduce the L2 cache size to 512KB and 256KB (due to the nature of SPEC benchmarks,

we cannot find one that experiences memory problem with 1MB L2). For 512KB L2, the bandwidth consumption increases by 130% from the default model, whereas 256KB L2 leads to 529% memory traffic increase. As mentioned in earlier sections, the majority of the memory traffic comes from permutations and therefore can be properly pipelined and parallelized with memory banking. For memory-bound applications, it is recommended to use a separate permutation chip to offload the traffic from the front-side bus—Section 3.3.4.

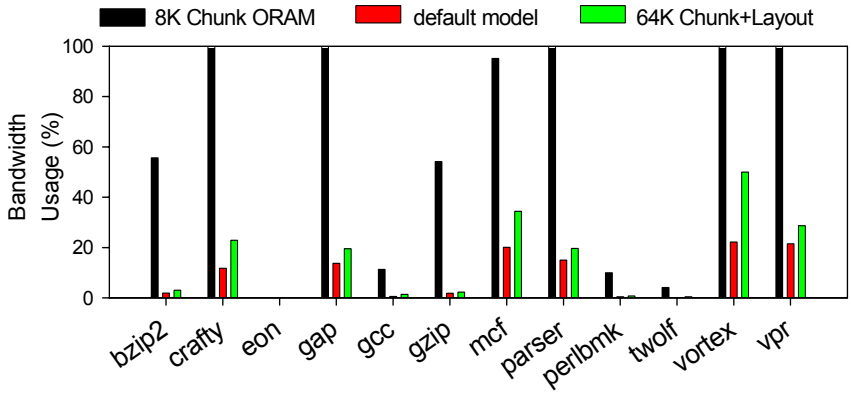


Figure 19. Percentage of total memory bandwidth used.

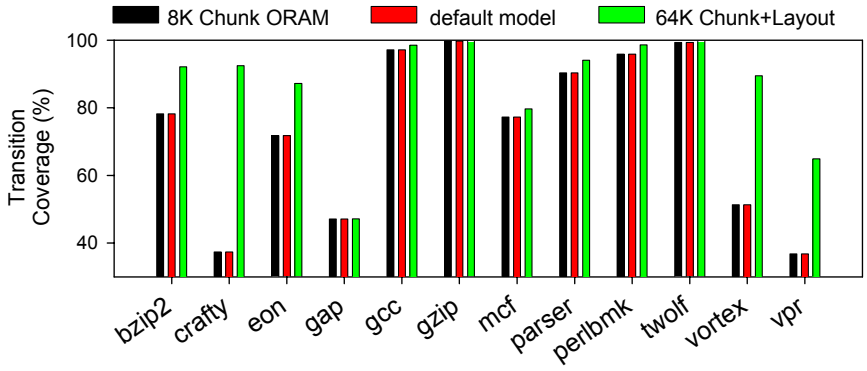


Figure 20. Percentage of address transitions covered.

Figure 20 gives transition coverage of the 3 models, which is an indication of the level of security guarantee that is achieved (Section 3.4). The first two models show the

same 75% transition coverage on average, because they both protect at 8K chunk level. With 64K chunk and layout optimization, 87% transition coverage is achieved. Notice that for some benchmarks such as crafty and vortex there is a significant increase in the transition coverage (and thus the security). This is due to the layout optimizations undertaken by the compiler. Some benchmarks do not benefit from layout optimizations due to the inherent nature of accesses esp. those resulting in gap due to the heap accesses. This also causes more permutations, since the number of blocks each permutation can unlock is lower.

Thus, one can see that significantly higher transition coverage can be achieved in the 3rd model with almost insignificant performance degradation. The layout optimization provides around 95% transition coverage for code and static data, while it only causes negligible slowdown due to the small size of that part. The rest of the slowdown is due to large chunk-size and overall transition coverage is much higher with little performance penalty.

It is important to understand if the user can provide specifications to exclude a small percentage of code and data as non-security-sensitive, our scheme will eliminate almost all the leakage on the address bus with negligible slowdown. For the default model, the user needs to identify roughly 25% such code and data, while under the "64K Chunk+Layout" model, he only needs to exclude about 5% of the code and static data (Figure 14) and 13% of the data on the heap, assuming stack is protected as in Section 3.5.2. This means the user specification can be very rough or in same cases, a nice compiler approach will probably do the work too, making our scheme practical to achieve a good level of security guarantee almost automatically.

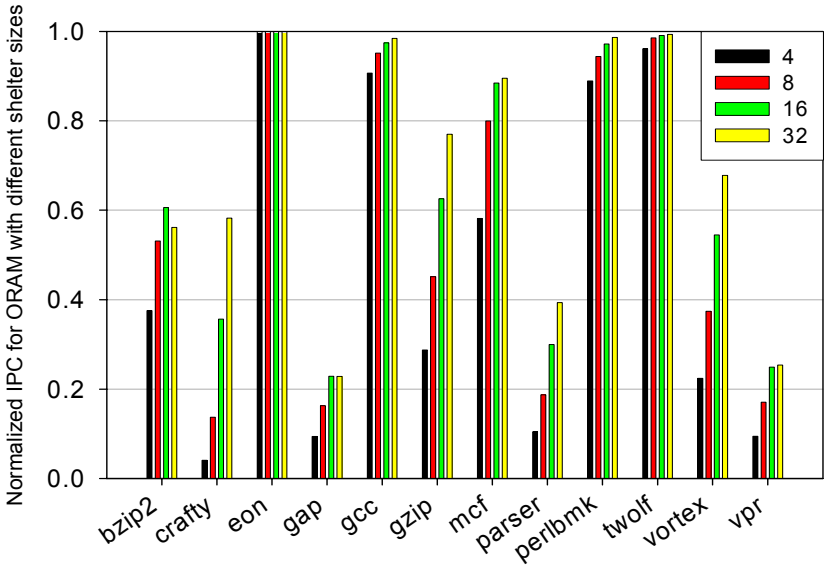


Figure 21. Normalized IPC results for the ORAM scheme with different shelter sizes.

Figure 21 shows IPC results for the ORAM scheme with different shelter sizes. Results are normalized to the baseline without address bus protection. In the experiment, only the size of the shelter buffer in the ORAM scheme is varied. The shelter buffer sizes used in the experiments are 4 blocks, 8 blocks, 16 blocks and 32 blocks respectively. The chunk size is 8KB. From the results, a larger shelter area generally leads to a better performance under the ORAM scheme. The reason is that with a larger shelter buffer, the chance of a miss cache block found in the shelter buffer is larger, thus the cache miss can be satisfied faster. On average, the performance degradation for a 4-block shelter buffer, an 8-block shelter buffer, a 16-block shelter area and a 32-block shelter area is 53.7%, 44.2%, 35.6% and 30.6% respectively. Although a larger shelter buffer improves performance, the performance degradation is still significant and much larger than the proposed HIDE scheme. Moreover, each chunk requires its own shelter buffer, so the memory overhead is increased rapidly when the shelter buffer is enlarged.

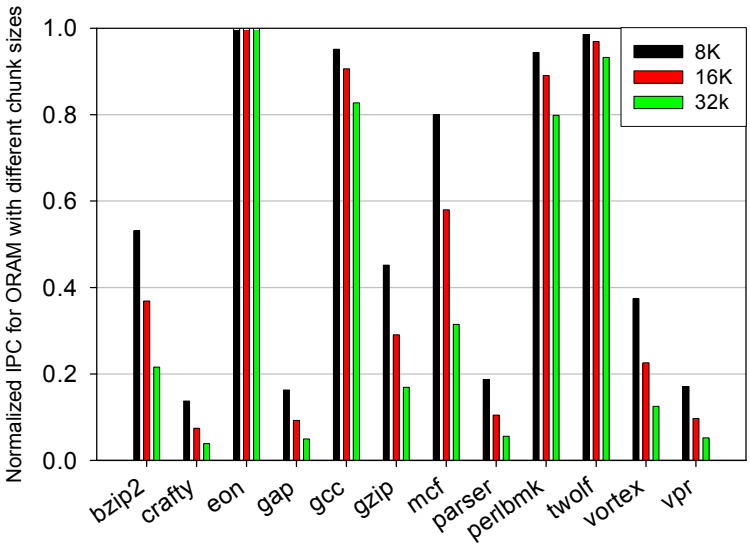


Figure 22. Normalized IPC results for the ORAM scheme with different chunk sizes.

Figure 22 shows IPC results for the ORAM scheme with different chunk sizes. Results are normalized to the baseline without address bus protection. The size of the shelter buffer is fixed at 8 cache blocks. From the results, it is very clear that with a larger chunk size, the performance under the ORAM scheme deteriorates quickly, showing that the ORAM scheme does not have a good scalability. When the chunk size is 8K, the average performance degradation is 44.2%; when the chunk size is 16K, the average performance degradation is 53.4%; when the chunk size is 32K, the average performance degradation becomes 61.7%.

In addition to the heavy performance degradation, the ORAM scheme also consumes much more bandwidth than our proposed HIDE scheme, as shown briefly in Figure 19. Thus, the ORAM scheme is apparently inferior to our HIDE scheme. We will now focus on more evaluation of the HIDE scheme.

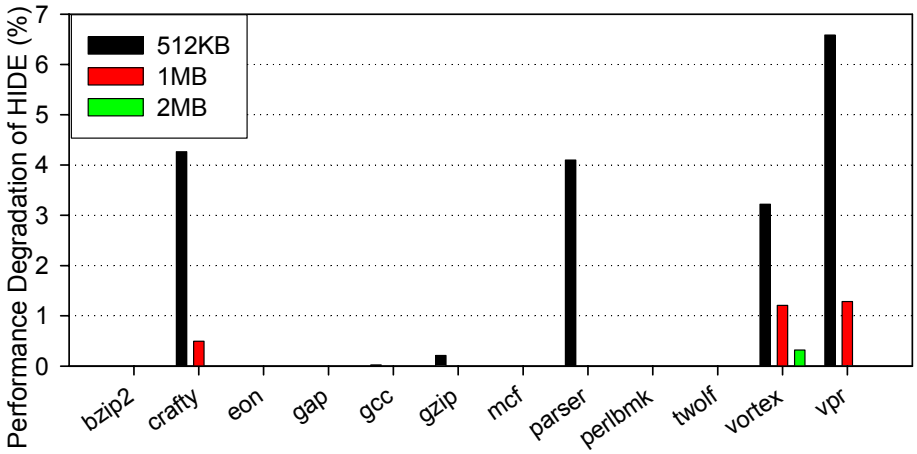


Figure 23. Performance degradation for the HIDE scheme with different L2 cache sizes.

In Figure 23, we vary the size of the L2 cache under the default model of our HIDE scheme to see how performance degradation changes. From Figure 23, only some benchmarks have observable slowdown, typically those with relatively large working sets. Small cache leads to bigger slowdown due to more L2 misses causing more permutations. On average, the slowdown for 512K, 1M, 2M L2 is 1.5%, 0.3%, 0.03% respectively. Our experiments show that the slowdown is within 20% even for 64K caches, making it applicable to low-end systems with smaller caches where information leakage might be more severe. Figure 24 shows L2 cache miss rate for the HIDE scheme with 512KB, 1MB and 2MB L2 cache sizes respectively.

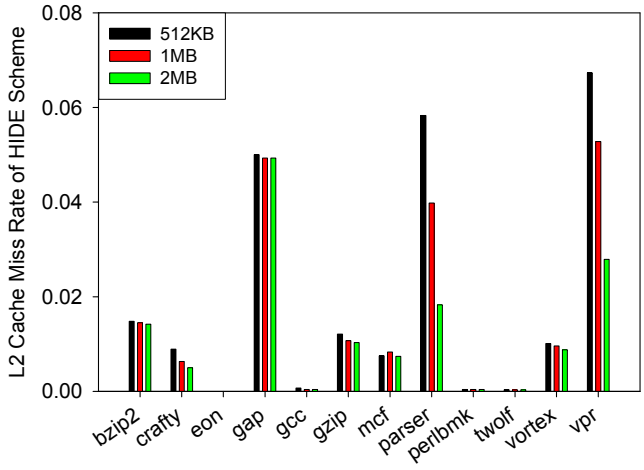


Figure 24. L2 cache miss rate for the HIDE scheme with different L2 cache sizes.

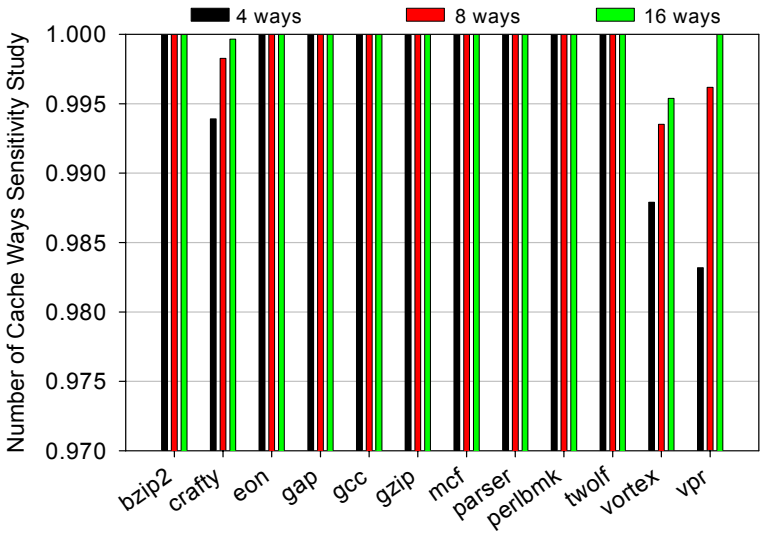


Figure 25. Performance degradation for the HIDE scheme with different L2 cache ways.

In Figure 25, we perform sensitivity study for the number of cache ways of the L2 cache under the default model of our HIDE scheme. IPC results are normalized to the baseline without address bus protection. By default, our customized L2 cache has 4 ways. Intuitively, when the number of ways is increased, the chance that all ways in a set are locked is smaller, thus the number of forced permutations is smaller, leading to a better

performance. From our results, if the 1MB L2 cache has 4 ways, the average performance degradation is 0.3%. When the L2 cache has 8 ways, the average performance degradation is only 0.07%. When the L2 cache has 16 ways, the average degradation is near zero. Thus, increasing the number of ways of the L2 cache is effective to reduce the performance overhead further if the cost is affordable.

Next, we do sensitivity study for chunk sizes under the default model. All comparisons are against the 8K-chunk default model and all numbers are normalized to the 8K-chunk default model. A better protection granularity leads to more slowdown and bandwidth consumption at the same time a better coverage rate. Figure 26 shows the performance degradation with a larger chunk size. On average, the slowdowns relative to the default model for 16K, 32K, 64K chunk are 0.13%, 0.55%, 1.05% respectively. Figure 27 shows the increase of bandwidth consumed. For 16K, 32K and 64K chunk, bandwidth increases are 18%, 40%, 73% respectively. Again, a permutation chip could offload this bandwidth increase from front-side bus.

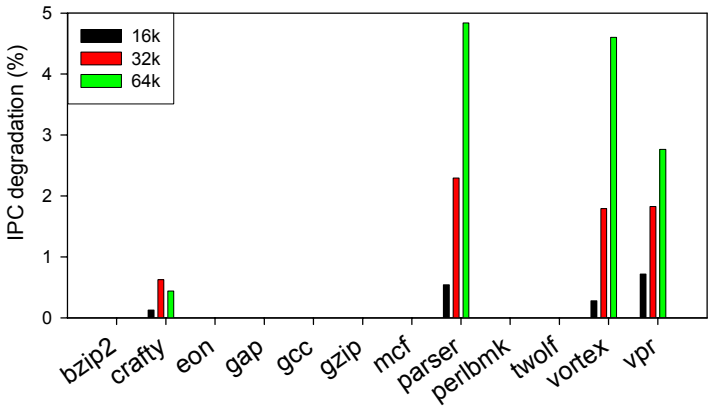


Figure 26. Performance degradation with larger chunk sizes comparing with the default model.

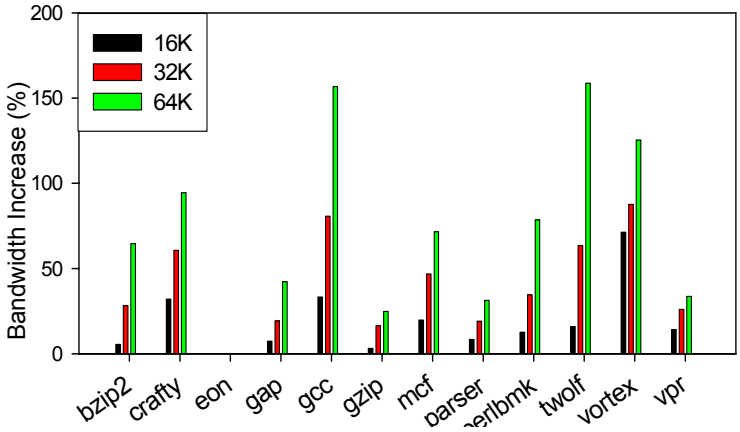


Figure 27. Bandwidth consumption increase with larger chunk sizes comparing with the default model.

Figure 28 shows the improvement of transition coverage rate with a larger chunk size. For 16K, 32K and 64K chunk size, transition coverage rate is improved over the default model by 1.6%, 4.3% and 5.9% respectively. From the results, without compiler optimizations, a larger chunk size improves coverage rate but not significantly. The major reason is that the misses from the L2 cache are already quite irregular and 64K chunk size is still not large enough to reduce inter-chunk transitions significantly. Thus, compiler layout optimization is critical to reduce inter-chunk transitions and improve coverage rate, which is shown in Figure 20. On the other hand, it is critical not to be confused by the small improvement brought by a larger chunk size. It does not show that a larger chunk size is very ineffective. A larger chunk size achieves better security guarantee, which is the main point of our HIDE scheme. All intra-chunk transitions can be protected. Thus, 64K-chunk achieves much better security guarantee than 8K-chunk, regardless of the improvement over transition coverage rate.

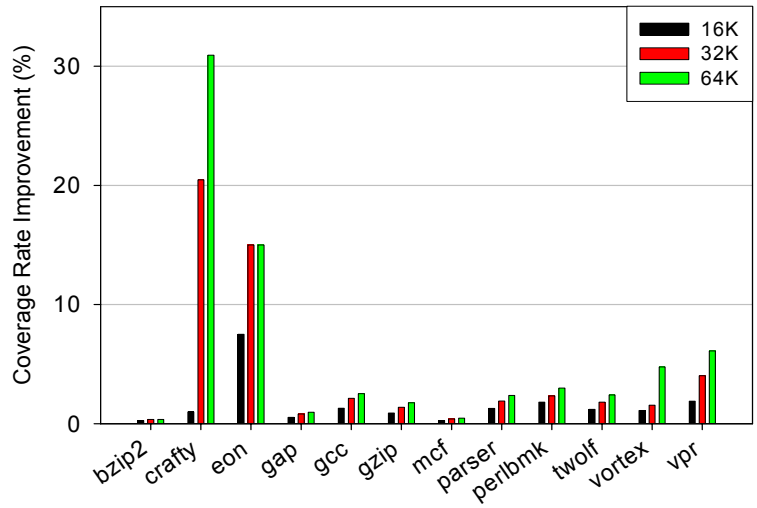


Figure 28. Transition coverage rate improvement with larger chunk sizes comparing with the default model.

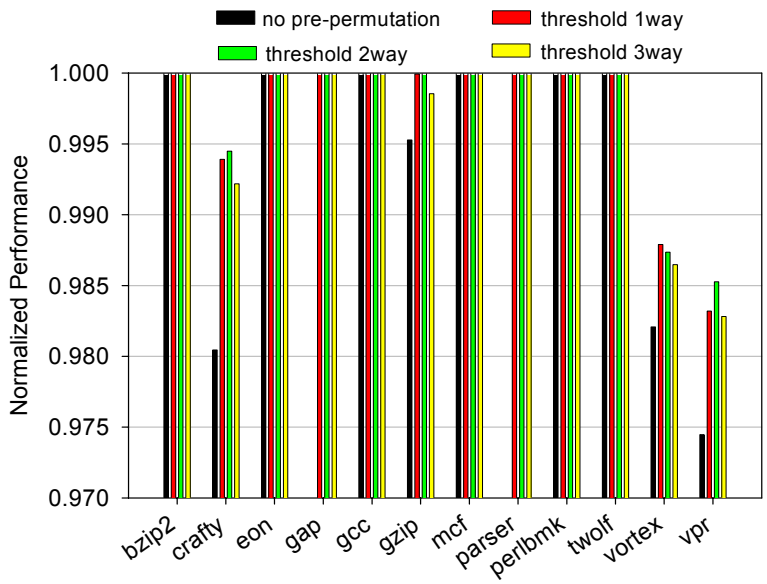


Figure 29. Effects of pre-permutation optimization under different pre-permutation strategies.

Figure 29 shows the effects of pre-permutation optimization under different pre-permutation strategies. In our default model, the L2 cache has 4 ways. We studied four cases: without pre-permutation, pre-permute when one way of a set is locked, pre-

permutate when two ways of a set are locked, and pre-permutate when three ways of a set are locked. IPC numbers are normalized to the baseline without address bus protection. For many benchmarks, performance degradation is negligible even no pre-permutation is done. For those benchmarks having observable performance degradation, Figure 29 shows that pre-permutation is effective to reduce runtime overhead. Without pre-permutation, the average performance degradation is 1.2%; if we pre-permutate when one way of a set is locked, the average performance degradation is reduced to 0.33%; if we pre-permutate when two ways of a set are locked, the average performance degradation is 0.3%; if we pre-permutate when three ways of a set are locked, the average performance degradation is 0.36%.

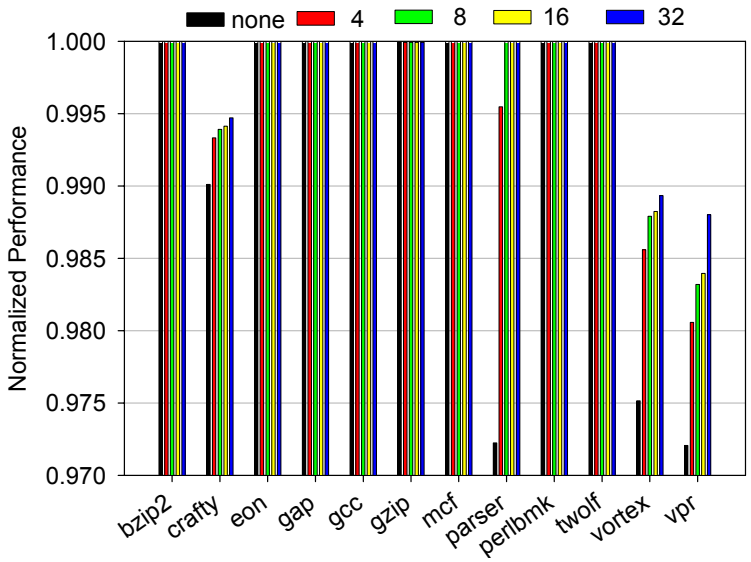


Figure 30. Effects of fetch buffer optimization with different buffer sizes.

Figure 30 shows the effects of fetch buffer optimization. When all ways in a set are locked, the requested cache block can be put into the fetch buffer temporarily rather than waiting for the completion of a permutation. From the results, fetch buffer optimization is also very helpful to benchmarks having observable performance

degradation. Without fetch buffer optimization, the average performance degradation is 0.75%; with a 4-entry fetch buffer, the average degradation is 0.37%; with an 8-entry fetch buffer, the average degradation is 0.3%, which is our default configuration; with a 16-entry fetch buffer, the degradation is 0.27%; with a 32-entry fetch buffer, the degradation is 0.22%. Thus, a small fetch buffer is already enough to tolerate permutation latency.

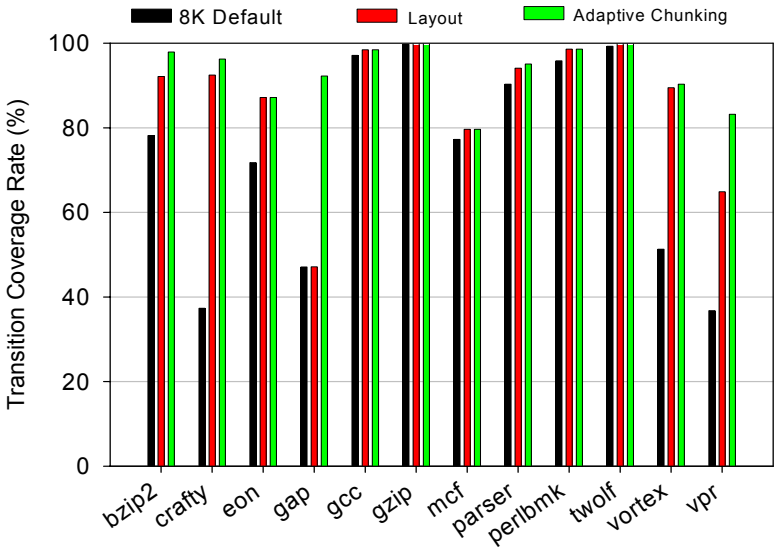


Figure 31. Effects of adaptive chunking.

Figure 31 shows the improvement of coverage rate due to adaptive chunking. Compiler layout optimization can also improve coverage rate, but compiler can only make static decisions and can normally do little about inter-chunk transitions caused by heap accesses. Adaptive chunking, on the other hand, can merge together chunks causing many inter-chunk transitions between each other at runtime. Moreover, adaptive chunking can handle heap accesses without problems. Thus, adaptive chunking is an effective supplement to compiler layout optimization. From the results, for those cases in which a good coverage rate cannot be achieved even with compiler layout optimization,

adaptive chunking helps significantly. On average, in the default model, the coverage rate is 75%; with compiler layout optimization, the coverage rate is improved to 87%; with adaptive chunking, the coverage rate is improved to 93%. We found that adaptive chunking helps data accesses more. It is more difficult for the compiler to determine data access patterns than to determine code access patterns, thus the compiler layout algorithm is more effective on code access. We also found that the performance and bandwidth overhead of adaptive chunking is always negligible and is not shown here.

3.8 Related Work

It is important to distinguish between "security guarantee" and "seemingly secure". For the latter, there are many ways such as reordering the blocks at runtime, reading blocks to a buffer then writing out after some time to new places, obfuscating the code, issuing random accesses etc.; however all these approaches provide no guarantee on how much information can be leaked. A "seemingly secure" approach fails to achieve any security guarantee, which is a severe flaw because it is hard to fathom how powerful (smart) an attacker might be. Only security guarantees establish the security strength of a system without making assumptions about the attacker, which is the essence of this work.

Code obfuscation techniques [5][19][20][21][70][80][65] are only "seemingly secure". Code obfuscation only makes reverse-engineering and cracking relatively harder. No security guarantee is provided as such.

The DS5000 series processors support so-called address bus encryption, which is equivalent to the initial permutation in Figure 8.c. However, it does not permute the memory space repeatedly at runtime. Therefore the attacker can still construct the CFG in the same way as mentioned in Section 3.1. The DS5000 also issues random fetches in

order to confuse the attacker (seemingly secure). However, random fetches can be easily discerned from true accesses in loops, which repeat more frequently. Actually, DS5002FP has been completely cracked [61].

Goldreich [41][40] proposed three approaches to guarantee no information leakage on the address bus, however all of them incur big overhead. For example, the "square-root solution" needs to read the entire shelter buffer before each access; the "hierarchical solution" takes O(t*log(t)*log(t)) memory space after t accesses, causing memory explosion.

The leakage-proof program partitioning work done by us previously [126][125] tackles a similar problem. Our previous work focuses on combating control flow information leakage due to dynamic sequences of program partitions transmitted through network in a networked embedded system environment. On the other hand, this work focuses on eliminating the control flow information leakage due to code/data blocks transmitted through system address bus. Our previous work inspired our address bus protection work, but both the assumption and the solution of the leakage-proof program partitioning work are fundamentally different from this one.

3.9 Summary

In this chapter, we present a lightweight solution to prevent information leakage on the address bus due to external memory accesses and to boost intrusion prevention capability of our software protection infrastructure. We show that information leakage prevention is critical for a secure architecture to defend against important side-channel attacks to software code and data confidentiality. However, all known solutions with

enough security guarantee [41][40] suffer from large performance degradation and memory overhead.

In this work, we propose the HIDE scheme including the hide cache with cache block locking and permutation mechanisms. HIDE provides chunk-level protection and interface for compiler optimizations and user specifications. Then we propose compiler optimizations for code and data layouts as well as other runtime optimizations to reduce overheads and improve the level of security guarantee.

The main contributions of this work are 1) Cache locking mechanism to indicate the timing to permute the memory space. Since our lock cache is a simple extension to the common set-associative cache, we are able to reuse most of the resources to reduce hardware cost. 2) Chunk level protection to selectively protect important data and code in a lightweight way. 3) Compiler techniques including layout optimization and stack/heap management optimizations to improve the security guarantee. 4) Adaptive chunking to dynamically adjust chunk sizes and merge chunks with frequent transitions.

Our results show that with 64KB chunk level protection and the layout optimization, we can guarantee that 87% of the address sequence is protected, in which 95% of the accesses to code and static data are hidden. Under the HIDE scheme, interfaces are provided for the compiler or user to further improve the level of security guarantee or to narrow down the protection domain to achieve almost complete protection. In that way, all security sensitive code/data could be identified and effectively protected in terms of the information leakage on the address bus. The protection strength can be further improved through adaptive chunking.

The performance overhead is at most 1.5% in our experiments for a general purpose processor. The increase of the bus traffic takes a very small part of the total bandwidth available for our benchmarks. The majority of the traffic increase is due to permutations. Such traffic is very regular therefore we can reduce its overhead in multiple ways as suggested in this chapter. Finally, most on-chip hardware components for HIDE are small. The largest component, i.e. the permutation unit with 64KB out_buffer can be shifted to the permutation chip as well. Due to the low overhead of the HIDE infrastructure, it is possible to apply it to low-end systems with smaller caches where leakage on the address bus might be more severe.

4 TRAINING-BASED ANOMALOUS PATH DETECTION

Our hardware infrastructure enhanced with address bus information leakage prevention can prevent attacks to software confidentiality and detect attacks to software integrity from other malicious software (even including a malicious operating system) and common hardware attacks. However, one important observation is that it cannot defend against attacks exploiting flaws/bugs in the protected software itself. The classical example of this kind of attack is buffer overflow attack [82], which can be regarded as due to a programming error (missing bound checking). Other examples include format string attacks and return-to-libc attacks etc. The hardware architecture alone cannot prevent such attacks. As far as the secure processor concerns, the instructions to overflow the buffer are perfectly legal instructions. In fact, to maintain the original program semantics, the buffer should be overflowed.

There are numerous software/hardware solutions to buffer overflow attacks, including [23][22][108][6][50][44][89][18][111][27][33][8][83][104][43][77][131]. However, even buffer overflow attacks are completely prevented, there will be other new attacks emerging, for example format string attacks [91]. In general, it is impossible to build a bullet-proof secure system and no secure system is able to prevent all attacks. There may be design/implementation flaws in the underlying protection scheme, or the attacker may exploit the flaws such as buffer overflows in the software to be protected. To be realistic, we have to assume that some attacks will be able to evade the protection scheme implemented. To protect software from those attacks, we build a second line of defense consisted of intrusion detection and intrusion recovery mechanisms, which is able to detect both known and unknown attacks and even recover from attacks. Such an

intrusion detection and recovery scheme is missing from the existing secure architectures and is one of the major contributions of our work.

In this chapter, we give some background knowledge on intrusion detection and present a training-based intrusion detection scheme to detect anomalous dynamic program paths. The scheme monitors the software execution at a very fine granularity and is able to detect attacks tampering program control flows with high precision and negligible performance overhead. However, the major concern of the scheme is the requirement of training and possible false positives.

4.1 Introduction

Intrusion detection is a critical component of an overall security solution. Due to the following reasons, intrusion detection has become an indispensable means to help secure a computer system. (1) A completely secure system is impossible to build. (2) Protecting software through cryptographic mechanisms, such as in the XOM [64] machine model, cannot prevent software's internal flaws (such as buffer overflows), which can be exploited by attackers. (3) Other operational mistakes, such as misuses, misconfigurations etc., may jeopardize the system as well.

Traditionally, intrusion detection can be classified into misuse detection and anomaly detection. Misuse detection tries to identify known patterns of intrusions and detect intrusions with pre-identified intrusion signatures. On the other hand, anomaly detection assumes that the nature of an intrusion is unknown, but will somehow deviate from the program's normal behavior. Misuse detection could be more accurate, but suffers from its inability to identify novel attacks. Anomaly detection can be applied to a wide variety of unknown (new) attacks, however normal behavior and abnormal behavior

have to be properly distinguished to reduce the number of false positives (or false alarms). We will focus on anomaly detection in this dissertation since it is more general and powerful.

A number of anomaly detection techniques have been proposed. Most early anomaly detection approaches analyze audit records against profiles of normal user behavior. Forrest et al. [32] discovered that a system call trace is a good way to depict a program's normal behavior, and anomalous program execution tends to produce distinguishable system call traces. A number of papers [59][76] focus on representing system call sequence compactly with finite-state automata (FSA), but these schemes are easily evadable because they use very little information and monitor with coarse granularity. Recent advances [115][92][31] propose including other program information to achieve faster and more accurate anomaly detection.

In general, approaches that monitor the program with finer granularity should be able to detect more subtle attacks. Both [31] and [30] expose important attacks that existing anomaly detection mechanisms are unable to detect. To detect such anomalies, a very fine monitoring granularity is necessary. However, software-based anomaly-detection systems already suffer from huge performance degradation even when operating at system call level. Thus, refining the granularity further in a software-based solution could lead to severe performance loss and is thus not viable. Due to inability to work at finer granularities, software-based intrusion detection cannot offer strong detection strength. Moreover, the anomaly detection software itself can be attacked like any other software.

In this chapter, we present a hardware-based scheme to detect anomalies by checking program execution paths dynamically. As far as we know, it is the first hardware-based intrusion detection scheme ever proposed. With hardware support, our anomalous path checking scheme offers multiple advantages over purely software-based solutions including near zero performance degradation, much stronger detection capability and fast reaction upon an anomaly.

4.2 Previous Work and Their Limitations

First, we would like to clarify the difference between anomaly detection and buffer overflow detection. There has been a lot of work on using specialized software and hardware to detect buffer overflow attacks such as [22][103]. These techniques focus on attack mechanisms rather than symptoms and plug the hole so that the attacker cannot apply the particular exploit. However, due to many sources of vulnerabilities, it is possible to start an attack using other exploits and thus, a mechanism that detects an attack based on its symptoms rather than a particular exploit is always desirable. As against those schemes specialized to defend buffer overflow attacks, our scheme can be used to detect any kind of attack, as long as the attack changes normal program control flow somehow (and thus, our technique is a symptom-based technique as against targeted to a particular exploit). Buffer overflow is merely one exploit that can cause an anomaly. Other examples include format string attacks, Trojan horses and other code changes, maliciously crafted input, unexpected/invalid arguments/commands to divert control flow into buggy/rare paths etc. Our scheme can also detect viruses alternating normal behavior of other applications, like a WORD MARCO virus. Actually, many viruses cause other legal applications (WORD, Internet Explorer etc.) to behave illegally to cause damage

since the virus itself is very compact and has very limited code. Such altered behavior can be detected as an anomaly from the normal one. In short, our scheme, as a general anomaly detection scheme, deals with a much broader problem than defending against buffer overflow attacks. Moreover, one big benefit of anomaly detection is its ability to detect future/new attacks besides countering existing attacks. It is certain that new attacks will be developed and it is obviously a bad strategy to research on countermeasures to them only after they have caused a huge damage.

Researchers have shown that a great number of attacks can be detected by analyzing program behavior during its execution. Such behavior could include system calls, function calls, control and data flows etc. We will first discuss basic system call based anomaly detection schemes, and then discuss improved solutions that provide finer monitoring granularity thus better detection capabilities.

Anomaly Detection via System Call Monitoring

System calls are generated as the program interacts with the operating system kernel during its execution, examples of which are fopen(), fgets(), and fclose(). Forrest et al. [32] argue that a system call trace appears to be a good starting point for anomaly detection. A system call trace can be considered as a distilled execution trace leaving many program structures out. Although a system call trace is a great simplification of the complete program activities, storing and checking against all normal system call traces is still a significant design effort. Exemplary approaches include [59][76].

To motivate our solution, we first discuss Finite State Automata (FSA) based system call monitoring techniques. To construct the FSA for system call monitoring, each program statement invoking a system call becomes a state on the state machine diagram.

The transitions between states are triggered by system calls. Each transition edge in the FSA is labeled by the triggering system call and the target state is determined by the feasible control flow. The state machine can be easily constructed through static analysis of the program [115]. One significant property of establishing the FSA based on static program analysis is that no false positives will be generated due to the conservative nature of static analysis. In that way, none of the normal executions will trigger alarms and if the state machine reaches the error state, there is a guarantee that something is anomalous.

Towards Finer Granularity

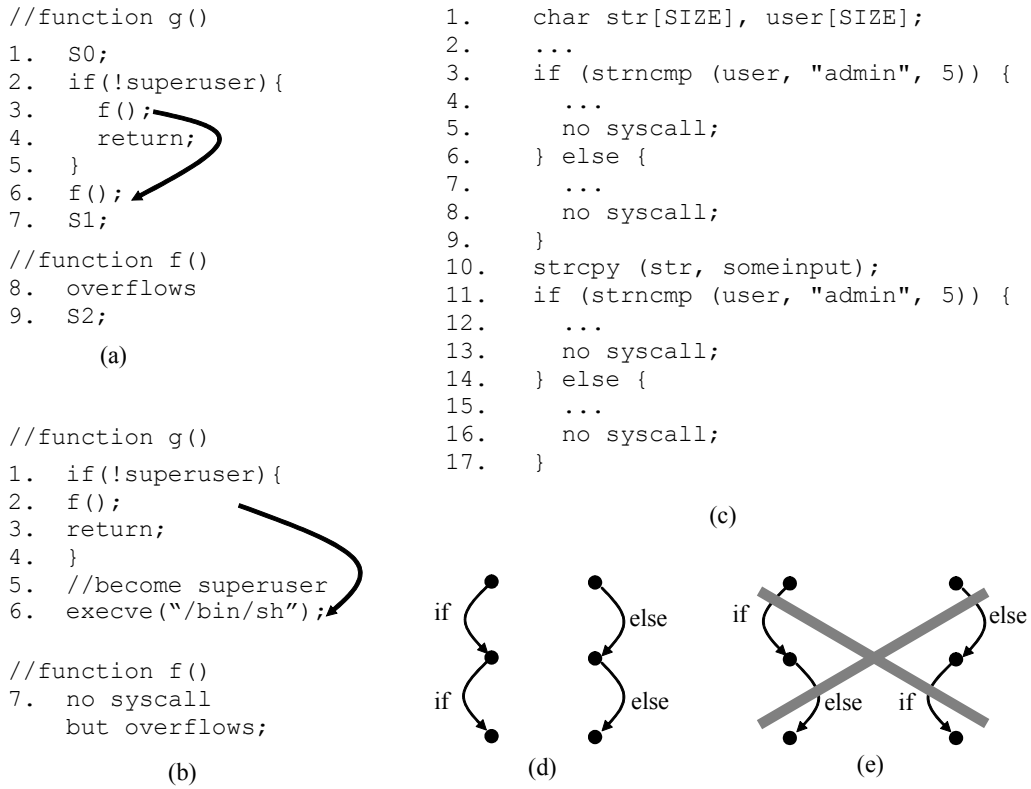


Figure 32. Attacks prompting detection at finer granularity. (From [31] and [30]).

Although the FSA-based approach in [115] has a nice property of zero false positives, later work points out that simply checking the FSA for system calls is not

sufficient and may cause false negatives in some cases. For instance, Wagner et al. notice that impossible paths may result at call/return sites which the attacker can exploit [115]. This can cause anomalous control flows along those paths being undetected. They propose a second model called abstract stack model, which records call stack information to solve the problem. In [31], the authors find out that even the abstract stack model can miss important anomalies. In Figure 32.a, although the condition in line 2 is false (i.e. the person executing the program is not a super user), the attacker can cause a buffer overflow to return from f() to line 7 instead of line 4, which actually grants him super user privilege. Since the system call trace is unchanged, no alarm will be triggered. This serious drawback can leave attacks getting through.

[31] provides a makeup solution called vtPath to detect the case in Figure 32.a by considering more program information. By constructing two hash tables to store not only the possible return addresses but also the so-called virtual paths (i.e. the sequence of procedure entry/exit points and return addresses traversed between two system calls), the false negative can be avoided. However, [31] also acknowledges that other anomalies may be left undetected. As shown in Figure 32.b, when there is no system call in function f(), a buffer overflow attack can easily grab undue privilege without being detected. This example shows that the granularity at system call level is not fine enough to detect many anomalies in reality, since an anomaly can very likely take place between system calls.

[60] proposes incorporating system call arguments into the detection model, which is also an example of incorporating more information. However, the attack shown in Figure 32.b does not depend on system call arguments. Most recently, [37] categorizes system call monitoring based anomaly detection systems into “black box”, “gray box”

and “white box” approaches. Systems relying on not only system call numbers but also extra information extracted from process memory fall into the “gray box” category. [37] further systematically studies the design space of “gray box” approaches and acknowledges the importance of monitoring granularity to the accuracy of the system. Their follow up work [36] proposes a new “gray box” anomaly detection technique called execution graph, which only accepts system call sequences consistent with the program control flow graph. However, similar to [31], due to the limitation of monitoring granularity, execution graph is not able to detect the attack in Figure 32.b either.

One possible solution to the problem illustrated in Figure 32.b is to verify all jump instructions, which are instructions that can cause non-sequential execution, including all conditional/ unconditional branches, indirect jumps, function calls/returns, etc. Compiler can record whether each instruction in the original program binary should be a jump instruction and collect all valid target addresses for each jump instruction. It is straightforward for the compiler to obtain valid target addresses for conditional branches. For indirect jumps, the compiler could obtain a set of normal target addresses using profiling. Then at runtime the program execution can be monitored based on the information collected by the compiler. Function calls/returns are special and can be checked efficiently by a return address stack (RAS). RAS is simply a stack for pushing/popping return addresses. Each return address of a dynamic return instruction is compared against the return address popped up from the top of the RAS. If they do not match, an anomaly is detected. In summary, we can choose to check each jump instruction individually and achieve very fine-grained monitoring granularity. This kind of approach will be able to detect a portion of attacks occurred during two system calls.

For example, it easily detects the attack shown in Figure 32.b. since the attack is based on stack smashing and is detected by the RAS. The essentially same attack can also be done by tampering a function pointer target by buffer overflows. Most likely, the indirect function call will have an invalid target and will be detected by the indirect-jump checking in this approach that tries to check each jump instruction individually.

4.3 Anomalous Path Checking

4.3.1 Motivation

Unfortunately, checking jump instructions individually still faces difficulties in detecting certain anomalies. Figure 32.c illustrates the problem. In this example, we have two strings defined, namely *str* and *user*. If the string *user* equals admin, the application will be granted super-user privilege. The code consists of two if-else statements. Between the two if-else statements, there is a strcpy library function call to copy some inputs to string *str*. Under a normal execution, the user can be either admin or guest. Thus, if we focus only on individual branches, since each if-else is regarded as an independent jump and since both directions of the jumps are possible, a scenario where if branch is taken in the first conditional branch and else branch is taken in the second one will be regarded as a normal execution. However, string *user* is not changed throughout this code segment, thus in fact either the two if branches are both taken or the two else branches are both taken in any legal execution, as shown in Figure 32.d. In other words, it is not possible to have one if-else statement taking if branch and the other taking else branch, as shown in Figure 32.e. Nevertheless, since the strcpy function may cause a buffer overflow and overwrite the string *user*, the attacker can change the contents in *user* after the first if-else statement, guiding the program into the two invalid paths shown in Figure 32.e.

The key aspect of this attack is that the attacker is not tampering return addresses or function pointers. Instead, he tampers critical control decision data. Unfortunately, as pointed out in [30], none of the existing techniques mentioned earlier can detect this type of anomalies. Even individually checking each jump instruction as mentioned earlier fails. Because such an approach is only aware that for the first if-else statement, both if and else branches might be taken in normal cases; and the same is true for the second if-else statement. Therefore, it will not trigger an alarm for the case shown in Figure 32.e. In other words, without correlating multiple jump instructions, anomalous execution paths cannot be discovered when each jump instruction jumps to a target that is considered normal. Therefore, in order to identify this kind of attacks, we should enhance anomaly detection with anomalous path checking, in which the program dynamic execution paths are checked against their normal behavior collected through profiling/training. A program's dynamic execution path is determined by the jump instructions along the path and their target addresses.

As concluded in [31], the detection ability of anomaly detection systems heavily relies on the granularity level it monitors. An anomaly detector solely relying on system call trace is crippled if an attack takes place between two system calls. It is easy to think about other examples that can foil the anomaly detector even if limited program control flow information is considered. Checking each jump instruction individually can lead to very fine-grained monitoring and anomaly detection with strong detection strength. With anomalous path checking, we further enhance detection capability, because multiple jump instructions on a path can be correlated to find path anomalies that will otherwise be left undetected. A simple example of such anomaly is shown above (Figure 32.c). In reality,

there could be many similar or even more subtle anomalies, requiring the strongest anomaly detection technique we can offer.

4.3.2 N-jump Path Checking Motivation

Checking program execution paths is much more complicated than checking each jump instruction separately. Theoretically, the number of possible paths could be exponential to the number of jump instructions in the path. This is so since each jump instruction (if assumed a conditional branch) could have two possible directions, i.e. taken or not taken. In other words, checking the whole execution path can be very expensive and infeasible to be implemented for runtime monitoring. In this work, we propose to analyze the whole execution path using sliding windows, where each window is a segment (say, n jumps) of the execution path. The anomaly detector verifies whether these path segments follow the normal execution path. In our scheme, the jump target of each direct jump instruction is checked separately in individual jump instruction checking component (detailed later) and an unconditional branch has only one possible target. Thus, we only need to care about conditional branches and indirect jumps in anomalous path checking, which we call *multi-target* jump instructions. We further define an n-jump path as an execution path segment on which exactly n multi-target jump instructions are encountered. An n-jump path can be uniquely decided by the address of the starting jump instruction and the directions of the n multi-target jump instructions along the path. In our work, the direction of an indirect jump is represented by its target address. N-jump path checking involves collecting n-jump paths seen during training to build a normal n-jump path set and checking n-jump paths during detection against normal n-jump paths. If an

n-jump path never seen during training is detected during detection, we regard it as an anomaly.

4.3.3 Training Phase

During training, we run the program to be monitored in a clean environment and use an existing technique [130] to record whole program paths (WPPs) of it during its execution. A WPP records the entire program execution path under a specific program input. Each input will generate a WPP, which represents a normal execution of the program (without intrusions). From each dynamic multi-target jump instruction recorded in a WPP, an n-jump path for that jump instruction can be extracted. The n-jump path records directions of all the n multi-target jumps along the path starting from the current jump instruction. It is identified by the address of the current multi-target jump instruction and the directions of all the n multi-target jumps along the path. Essentially, a WPP is segmented into a collection of n-jump paths using a size n sliding window with sliding step 1. N-jump paths obtained from different WPPs are finally aggregated together. This means as long as an n-jump path is normal for any one of the inputs, then it is regarded as normal. The aggregated set is used to detect anomalous paths at runtime.

The length or amount of training is controlled by the user. The goal is to have adequate training so that the detector has reasonably low false positive rate. In our experiments, we will show training for our scheme is effective.

4.3.4 Detection Phase

At runtime, the anomaly detector maintains a record of the latest n multi-target jumps and their directions, and creates a dynamic n-jump path based on these recorded n jumps. The dynamic n-jump path is identified using the address of the first jump

instruction (the oldest of the n jumps) and the directions of the following n jumps. The anomaly detector then checks whether this dynamic n-jump path exists in the set of n-jump paths collected during training. If not, the detector reports an anomaly. We will show later that the detection process can be optimized using hardware at run-time.

4.3.5 Detection Capability

As discussed in [31], the detection capability of an anomaly detection system depends on its monitoring granularity. For anomaly detection systems operating at system call granularity, attacks between two system calls will not be detected. An example is shown in Figure 32.c. Ideally, we want to collect the dynamic instruction sequence of a program and check whether the sequence of instructions is a normal one. Anomalous path checking technique approaches this ideal case by monitoring dynamic program paths. A dynamic program path is determined by the jump instructions along the path and their target addresses. Thus, with anomalous path checking, although the attacker may be able to modify instructions between two jumps, he cannot create new jump instructions in his attack code. The control flow of his attack code has to be a straight line and the attack code cannot include any function call, which is extremely restrictive to the attacker.

Anomalous path checking actually subsumes system call trace monitoring based anomaly detection because each function call is a jump instruction itself and each system call is normally invoked through a wrapper function call in standard C library libc.a [37]. Once the dynamic program path is known, the system call sequence along the path is determined too. The sequence of system calls made along a normal program path has to be a normal system call sequence, which means system call trace monitoring based schemes cannot detect any attack missed by anomalous path checking. On the other hand,

a normal system call trace does not guarantee a normal program path. The distribution of system calls in a program is sparse and irregular. There may be millions of jump instructions between two system calls and the program path between two system calls can be tampered without being detected under system call trace monitoring based scheme.

However, checking the whole program path imposes too much cost in terms of both storing normal paths and checking them at runtime. Thus, we choose to check n-jump paths instead. Checking all n-jump paths cannot guarantee that the entire program execution path is normal due to the limited length of n-jump paths. It is necessary to understand the cases when path anomalies cannot be detected with n-jump path checking. Figure 33 illustrates one such case. Here, two normal paths (1) and (2) share at least n-1 jump instructions in the middle. Path (3) is an anomalous path but it cannot be detected with n-jump path checking. When path (3) enters the shared part, the detector will think it is on path (1). When path (3) leaves the shared part, it is considered to be on path (2) with n-jump path checking. The reason is that n-jump path checking can only check a path traversing no more than n multi-target jump instructions. Fortunately, our experiments show that the scenario in Figure 33 rarely happens.

We do not try to design a general algorithm to decide the value of n for an arbitrary application, since it is highly application-dependent. We will show later that a larger n can help detect more attacks but also incurs more false positives and more performance overhead. The criterion to choose n is to select as big an n as possible given that both false positive rate and performance overhead are acceptable. Thus, the user can start with a large n then gradually reduces it until those two metrics are acceptable.

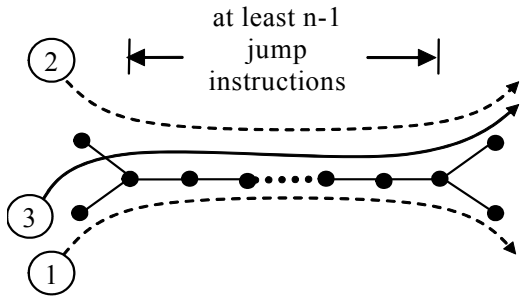


Figure 33. An anomaly path that cannot be detected.

4.4 Hardware Implementation

4.4.1 The Advantages

Performance

Although it is obvious that finer granularity brings better detection capability, the performance overhead it introduces is formidable under software-based approaches, e.g., [115][92][31]. Software-based anomaly detection systems suffer from large performance degradation even when operating at the system call granularity [115]. With hardware support, performance degradation can be made very small since the monitor is entirely implemented in hardware and works in parallel with instruction execution. In comparison, in system call monitoring based software approaches, intercepting system call alone can incur 100% to 250% overhead [92].

Anomaly Detector Tamper Resistance

Anomaly detection software, as any software, might itself be attacked. To alleviate this problem, software anomaly detectors are normally implemented as a separate software module. In that way, the detector is not affected by the vulnerabilities of the monitored program. For example, system call based monitoring normally depends

on special system tools such as strace to trace system calls rather than instrumenting the program binary directly.

However, such separation is not possible when we monitor the program at a very fine granularity. Normally there will be a dynamic jump instruction in every 10 instructions. Imagine the performance degradation of the monitored program when there is a context switch after every 10 instructions. Thus, we have to transform the monitored binary to insert monitoring code into it, which means the anomaly detector faces potential attacks due to the vulnerabilities not only in the detector itself but also in the monitor program. It is possible that the monitoring code is tampered or bypassed due to vulnerabilities inside the monitored program. On the other hand, our hardware anomaly detector resides in a secure processor, achieving tamper resistance.

4.4.2 Hardware Architecture Overview

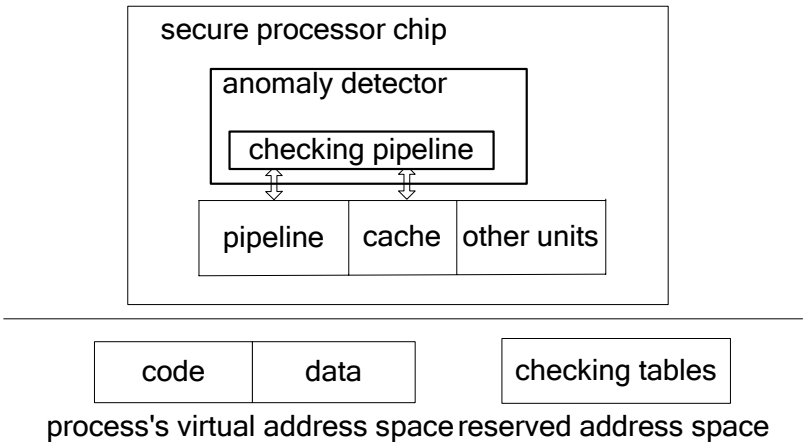


Figure 34. Architectural overview.

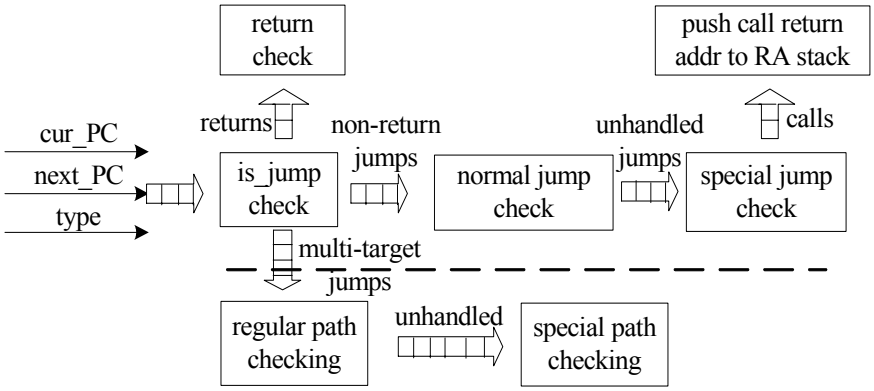


Figure 35. Checking pipeline.

Figure 34 shows an overview of our hardware components for anomalous path checking. Our anomalous path checking mechanism is built into our secure hardware infrastructure described earlier. The hardware infrastructure guarantees the confidentiality and integrity of all code and data residing off-chip in the external memory. Hardware anomaly detector resides in the secure processor and is tamper-resistant. Checking tables record normal behavior of the program (including information for each individual jump instruction and normal n-jump paths) and are utilized by the checking pipeline. Upon a program start, checking tables are loaded into a reserved segment of the program virtual address space and are only accessible to the anomaly detector itself. The program itself has no access to checking tables. Attempts to read/write checking tables from programs will be detected and prevented by the secure processor. Moreover, hardware attacks to checking tables are detected by the integrity-checking scheme of the secure architecture. Thus, no software or hardware attack can tamper program normal behavior data.

The checking pipeline illustrated in Figure 35 checks committed jump instructions against the normal behavior recorded in the checking tables. The checking pipeline is designed to be isolated from the original execution pipeline to minimize implications to

the latter. There are data paths from the execution pipeline to the checking pipeline to transmit information required by anomaly detection including current operation code, current PC address and computed next PC address etc.

The internal of the checking pipeline is detailed in Figure 35. The pipeline stages at the top of the figure (marked by the dotted line) check individual jump instructions. Individual jump instruction checking is relatively simple, so we just briefly describe it here and focus on anomalous path checking later. The first stage in individual jump checking checks whether each instruction is and should be a jump instruction. The information is recorded in a bitmap indexed by the instruction address. If it is a valid jump instruction, the second stage further checks each direct jump instruction (a conditional or an unconditional branch) to make sure its jump target is valid. The information is stored in a specially optimized hash table. Whether an instruction should be a jump instruction and the valid targets of direct jumps can be easily collected by examining the original program binary. At runtime, dynamic instructions are checked against those tables. There are some space optimizations to reduce the size of checking tables used in individual jump instruction checking, but they are also used in anomalous path checking so we will describe them later in the context of anomalous path checking. More detailed description about individual jump checking can be found in [129].

Function call/return instructions are checked specially by a hardware managed return address stack (RAS). Return address stack is simply a stack for pushing/popping return addresses. As shown in Figure 35, function returns are isolated after the is_jump check stage. Return address of each return instruction is compared against the return address popped up from the top of the RAS. If they do not match, an anomaly is detected.

A function call is nothing but a direct or indirect jump instruction that sets return address. It is first checked as a normal jump instruction. After it passes the checking, the anomaly detector pushes the return address of the function to the RAS for later verification when the function returns.

The checking of indirect jumps is incorporated in the path checking stages shown at the bottom of Figure 35. Our path checking stages follow the first stage and consist of two stages, namely regular path checking and special path checking. Among the checking tables, the ones that are used for our anomalous path checking are called path checking tables, which record all normal n-jump paths. The details of the path checking stages will be discussed in the next section.

4.4.3 Implementation Details

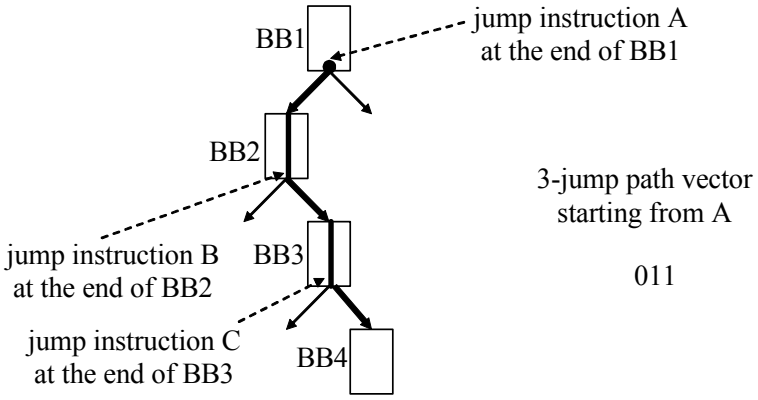


Figure 36. An example of n-jump path.

The goal of the anomalous path checking hardware is to efficiently check dynamic n-jump paths against normal n-jump paths collected during training. An n-jump path is a segment of the execution path with n multi-target jump instructions on it. For example, in Figure 36 the thick line shows a path starting from the end of BB1 going through three basic blocks. Three conditional branch instructions are encountered along

that path, i.e. the last instructions of BB1, BB2 and BB3. Obviously, we can uniquely identify an n-jump path by the address of its first jump instruction and the directions of the n multi-target jump instructions on the path. To indicate the directions of the n multi-target jump instructions, we define n-jump path vector as follows.

***N-jump Path Vector**:* a vector marking the directions of the n multi-target jump instructions on the n-jump path.

Thus, if we only consider conditional branches that have two target addresses, e.g. either fall-through or jump to a particular target address, an n-bit vector is enough to represent an n-jump path vector. For the example in Figure 36, the 3-jump path can be uniquely identified as starting at jump instruction A with 3-jump path vector 011. Here, we assume that left branch is marked as 0 and right branch is marked as 1. The most significant bit is the direction of the starting jump instruction. Therefore, an n-jump path can be represented succinctly because only the directions instead of the addresses of jump instructions are recorded except for the header of the path. For this example, the path checking table only records jump instruction A as the header and directions of the 3 jump instructions which are left-right-right, i.e. 0-1-1. This simplification is possible because the correct jump targets of each direct jump instruction (e.g. the left branch of instruction A should jump to the beginning of BB2 but not other places) are verified by other pipeline stages at the top of Figure 35 — refer to section 4.4.2.

Therefore, n-jump paths can be grouped based on their headers. For example, in Figure 36, all 3-jump paths starting from instruction A can be grouped and stored together. We group and store all normal n-jump paths in the path checking tables (Figure 34). There are two tables used in anomalous path checking. Regular path checking table

is used in regular path checking stage and special path checking table is used in special path checking stage.

One difficulty on path checking table design is that the data in it cannot be entirely regular, since we have to consider indirect jumps that may reach more than two targets and whose directions are represented by their target addresses. Although indirect jumps are not easy to handle, the majority of dynamic jump instructions are direct jumps, either conditional or unconditional. Unconditional branches have only one possible target and are handled by other stages. Conditional branches have two jump targets that are known by examining program binary. For conditional branches, only 1 bit is needed to indicate whether the branch is taken or not taken. Moreover, a large portion of indirect jump instructions are actually return instructions, which are checked separately as mentioned in section 4.4.2 and are not considered in path checking. Normally, only less than 2% of dynamic jump instructions are non-return indirect jumps. In conclusion, the jump instructions requiring irregular path data (rather than a single bit) are rare, which inspired our two-stage path checking scheme. The regular path checking stage handles most n-jump paths consisting of only conditional branches (those causing collisions after hashing are not handled), while all other cases are left to the special path checking stage.

Regular Path Checking

This stage handles dynamic n-jump paths consisting of only conditional branches. Thus, each jump instruction in the path has two directions. The n-jump path vector becomes an n-bit vector. The data structure for regular path checking is called regular path checking table and is shown in Figure 37.

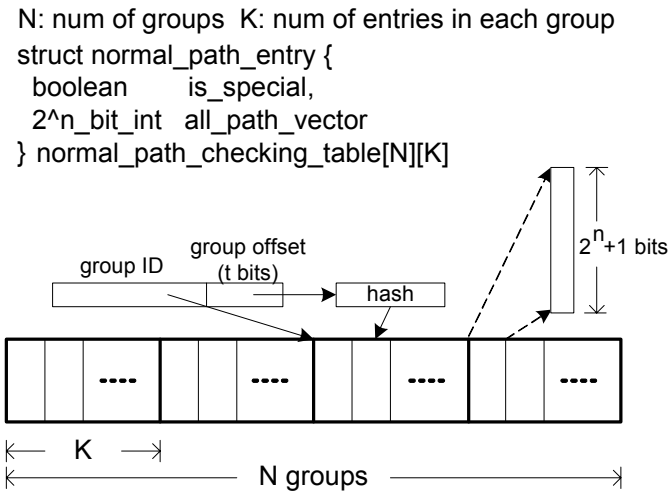


Figure 37. Data structure for regular path checking.

For each jump instruction, all normal n-jump paths starting from that jump instruction are stored together. Here, we use an all_path_vector to record all normal n-jump paths for each jump instruction. In this case, each n-jump path vector is an n-bit number, and therefore corresponds to a unique number within the range of [0, 2^n-1]. There are a total of 2^n possible distinct paths. The all_path_vector contains 2^n bits, in which each bit corresponds to an n-jump path. With a bit set in the all_path_vector, the corresponding n-jump path is indicated to be normal and vice versa. Essentially, all_path_vector is a bit vector recording which n-jump path is normal. The least significant bit represents path 0…0, the next bit presents path 0…1, and so on. It is noteworthy that we store the all_path_vector instead of all normal n-jump path vectors. This can be explained from two aspects. First, the all_path_vector is of fixed size, i.e. 2^n bits, while the number of normal n-jump paths is not fixed. Fix-sized entries are much easier to manage with hardware, leading to smaller hardware cost and performance degradation. Secondly, the number n we are currently dealing with is small. We will show that a reasonably small n is good enough to achieve very low false negative rate.

For small n, the space taken by the all_path_vector is comparable to that taken by all the normal n-jump path vectors. For example, if n=5, the all_path_vector requires 2^5=32 bits, whereas each 5-jump path vector needs 5 bits. In other words, if there are more than 6 normal 5-jump paths start with the same jump instruction, the all_path_vector will cost less space.

Refer to Figure 37 again. The structure regular_path_entry consists of the all_path_vector and a Boolean variable called is_special, which if set indicates that the corresponding jump instruction should be handled by the special path checking stage. Entries in the regular path checking table have to be retrieved using the address of the starting jump instruction. Hash table is the natural way to organize the entries in the regular path checking table. To retrieve a regular_path_entry, we first calculate a hash value of its jump instruction address then use the hash value to index into the hash table. However, there are a couple of problems with such a simple hash table design. First, a simple hash table cannot exploit code locality. The hash table entries for two adjacent jump instructions could be far away. Moreover, the processor always fetches a cache block of data from the external memory. Thus, if one uses a simple hash table mechanism, among multiple regular_path_entry records in one cache block it is very possible that only one record in the fetched block is touched before the block's eviction. This is very inefficient. Second, hashing can cause collisions. To avoid fetching the wrong information for a jump instruction, each hash entry has to be tagged which could waste space significantly. Due to the above reasons, our anomaly detection system abandons a simple hash table design and instead deploys a specially optimized data structure called groupwise hash table.

In Figure 37, the instruction address is divided into two parts: group ID and group offset. Assume that there are N groups and each group contains K entries, then the ith group starts at the address i*K. Inside each group, the group offset is first hashed then indexed into one of the entries in the group. The advantage of groupwise hashing is: 1) hashing saves space for non-uniformly distributed addresses of jump instructions; 2) hashing is only performed inside each group and each group has a number of sequentially stored entries. In this way, we can exploit spatial locality, because adjacent jump instructions are most likely located in the same group, thus their information will be stored close to each other. Some jump addresses may be hashed to the same location causing collisions though. As we observe, with a reasonable hashing function and a proper setup of N and K, collisions rarely happen. In the case of a collision, we indicate that the jump should be handled in the special path checking stage, which is done by setting the is_special bit in the corresponding entry for the jump instruction after hashing. Therefore, no tag is necessary to distinguish jumps hashed to the same entry since all the colliding jumps will be handled separately by the special path checking stage. With this customized hash table design, regular path checking table records all normal n-jump paths that are consisted of only conditional branches for non-colliding jumps.

To decide the parameters for the hash table, assume that the group offset is t bit long and the instructions are 4 bytes long, then $2^t*¼$ instruction addresses are hashed into K slots. The number of jump instructions is about 12% of all instructions, thus the actual number of jump instructions that will be hashed into a group is about $2^t*¼*12\%$. Thus, we can select K to be $1.5*2^t*¼*12\%$, which largely avoids conflicts and the space wastage is below 30%. We can choose a proper t, so that K entries only occupy a few

cache blocks, which improves locality among the entries of the same group. After K is decided from t, N should be $\lceil M/2^t \rceil$, where M is the code size in bytes.

As discussed above, a groupwise hash table not only improves cache performance, but also is an important optimization to reduce the memory requirement of checking tables. We also observe an important fact that there is no need to record the whole PC address to identify a branch target. We only need to record the base address of the code section and the offset to the base address to identify a branch target. In our scheme, we assume that the architecture is a 32-bit one and the code size is smaller than 2^{22} bytes (4MB - a large number for most applications). Thus, we only need 22 bits to identify a branch target instead of 32 bits. This is another important technique to reduce sizes of checking tables.

The above two space optimization techniques are also applied to checking tables for individual jump checking. Due to the group-wise hashing table optimization, there are also a normal jump checking stage and a special jump checking stage in individual jump checking.

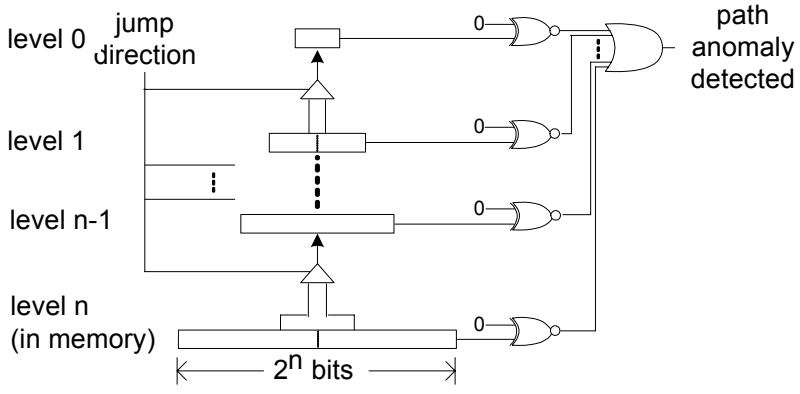


Figure 38. Regular path checking hardware diagram.

The path checking hardware for *regular path checking* is shown in Figure 38. The hardware maintains n bit vectors on chip. The n bit vectors are organized into n levels (level 0 to level n-1). The sizes of the bit vectors vary from 1 bit, 2 bits…to 2^{n-1} bits, where the level k bit vector contains 2^k bits. Level n is assumed to be in the memory and it is the *all_path_vector* of the current jump instruction. Upon reaching a jump instruction, the *all_path_vector* of the jump instruction is fetched. Then the direction of the jump instruction (only two possibilities in regular path checking, left branch or right branch) is used to select half of the bits (either higher 2^{n-1} bits or lower 2^{n-1} bits) in the *all_path_vector* to be stored in the level n-1 bit vector. Meanwhile, at each level of the bit vectors, half of the bits are selected according to the direction of the current jump and moved one level up, i.e. half of the level k bit vector (either higher 2^{k-1} bits or lower 2^{k-1} bits) is selected and moved to the k-1 level bit vector, where $k \in [1,n-1]$. Finally, if any bit vector becomes 0, a path anomaly is detected. A 0 bit in a bit vector means the corresponding n-jump path was never recorded in the normal runs during training. All bits in a bit vector being 0 means no path from this point is normal thus there is an anomaly. Thus, our hardware implementation of n-jump path checking *detects the anomaly as soon as possible* rather than waits for another n-1 jump instructions following the current jump and builds the n-jump path then checks whether the n-jump path is seen during training. For example, if the selected half bits of the *all_path_vector* of the current jump are all 0, then an anomaly is detected at once since the dynamic n-jump path starting from the current jump goes to an anomalous direction at the first step.

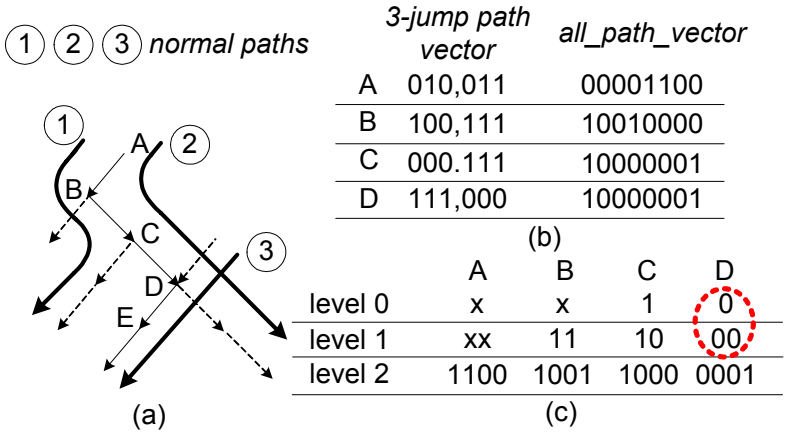


Figure 39. An example for regular path checking: checking anomalous path A→B→C→D→E.

To explain the above scheme further, we give an example in Figure 39. There are three normal paths marked as thick lines. The actual execution follows A→B→C→D→E. Figure 39.b shows 3-jump path vector and all_path_vector for each jump instruction. As in Figure 36, left branch is marked as zero and right branch is marked as one. The most significant bit is the direction of the starting jump instruction. Figure 39.c shows the contents of the on-chip bit vectors after reaching each jump instruction. After instruction A is reached and we know its direction to be zero, i.e. left branch, the lower 4 bits of A's all_path_vector are loaded into the level-2 bit vector. Since A goes to the left branch, no 3-jump path vectors going towards the right branch are needed. At B, the bit vector in level two is moved to level 1 and further pruned based on the direction B takes, i.e. the upper 2 bits are moved to the level-1 bit vector. Upon reaching D, its left branch is taken, however both level 0 and level-1 bit vectors are 0 indicating a path anomaly. This is so because no normal 3-jump path starting from B follows the actual execution path B→C→D→E (causing level 0 to be 0), and no normal 2-jump path starting from C follows the actual execution path C→D→E (causing the level 1 bit vector to be 0).

Special Path Checking

When the *regular_path_entry* loaded during regular path checking has *is_special* bit set (which indicates that the corresponding jump instruction leads to a collision during hashing), the n-jump path is sent to the *special path checking* stage. Also, an indirect jump instruction can cause all the current pending n-jump paths in regular path checking stage be sent to the special path checking stage including the n-jump path starting from the indirect jump instruction. The data structure in the special path checking stage is a hash table called *special path checking table*. It hashes the address of the starting jump instruction to index into the table and retrieves all the normal n-jump paths for the jump. The hash table is tagged and collisions are handled by a linked list. Special path checking table records all normal n-jump paths for colliding jumps and normal n-jump paths containing indirect jumps for non-colliding jumps. To support special path checking, the starting jump instruction corresponding to each level in regular path checking hardware is recorded and updated when the vector in the level is updated. In addition, when the special path checking stage is not idling, the direction of conditional branches and the target address of indirect jumps have to be sent to it. The former is used to check the conditional branches in an n-jump path and the latter is used to check indirect jumps.

Since an indirect jump could have multiple targets, we use the target address to identify the direction of an indirect jump (as against 0/1 to specify the direction of a conditional branch with true or false outcome). The target of an indirect jump along a program path is collected during training. With the presence of indirect jumps, an n-jump path vector cannot be an n-bit vector any more. The actually length of n-jump vector

depends on the number of indirect jumps in the path. In addition, we have to record the positions of indirect jump target addresses in an n-jump path vector for proper processing. The irregularity of data structure complicates special jump checking stage and there is no interesting optimization opportunity. Thus, the latency in special path checking stage is longer. However, in most cases, only a small percentage of n-jump paths reach this stage, thus the performance impact is small. During our experimentation, we find that in most cases less than 20% of n-jump paths reach this stage.

Our staged anomaly detection hardware is carefully designed to reduce performance overhead by pipelining the anomaly detection. Most requests are handled in the first and the fast stage. Only a small portion will enter the slower stage. Since our design can achieve very good performance without further architectural optimizations, the requirement of expensive hardware resources such as large on-chip buffers could be avoided. In addition, the checking tables are also particularly optimized to reduce the pressure on physical memory.

4.4.4 Other Considerations

A function call is just one type of jump instruction; therefore, paths across function boundaries are checked in the same manner. DLLs or dynamic libraries are shared by many processes and are loaded on demand. They can be checked in the same way as normal programs as long as checking tables are loaded together with the DLLs. The jump instruction addresses in the checking tables for a DLL should be relative to the beginning of the DLL, so that they are independent to the actual location the DLL is loaded into the program's virtual address space. System calls such as fork and exec family can create copies of the running process or overwrite it completely. A copy of the

checking tables can be created if the process has been forked. In addition, new checking tables are loaded to the memory space if an exec family system call is executed.

The setjmp()/longjmp() functions are used in exception and error handling. setjmp() saves the stack context and other machine state for later recovery by invoking longjmp(). Thus, after longjmp(), the program resumes as if the setjmp() just returned. To handle longjmp(), the anomaly detector has to be aware of the execution of setjmp() and saves the current state of the anomaly detector upon a setjmp() call. The anomaly detector state includes the state of the return address stack and the state of the path checking hardware, such as the n-1 on-chip bit vectors (Figure 38). After longjmp(), the state of path checking hardware is restored accordingly.

4.5 Evaluation

The evaluation of an anomaly detection system consists of two aspects: precision (detection capability) and performance. In this section, we show that our anomalous path checking scheme achieves both good precision and low performance overhead.

Table 4. Daemon programs and vulnerabilities.

Server Program	***total # of vulnerabilities***	***buffer overflow***	***format string***	***other bugs***
telnetd	2	1	0	1
wu-ftpd	5	2	2	1
xinetd	2	2	0	0
crond	1	1	0	0
syslogd	1	0	1	0
atftpd	1	1	0	0
httpd (CERN)	2	1	0	1
sendmail	4	4	0	0
sshd	2	1	0	1
portmap	1	0	0	1

In our experiments, we choose 10 daemon programs as benchmarks, as listed in Table 4. Those daemon programs are very common in UNIX/LINUX based systems. They have well-known vulnerabilities so that we can evaluate our scheme in a standard way. These programs are important since they offer essential system services for networked systems. The same programs will be used in evaluation of all of our intrusion detection and intrusion recovery schemes. We intentionally choose old versions of these programs with well-known vulnerabilities. Those vulnerabilities will be used in false negative measurement. Table 4 also lists type and number of vulnerabilities in each of these daemon programs used. To make our results more convincing, we also provide results for SPEC 2000 integer benchmark programs in a couple of important experiments.

To evaluate precision of our anomalous path checking scheme, we implement it in an open-source IA-32 system emulator Bochs [9] with Linux installed. Good precision means both low false positive rate and low false negative rate. False positive is acknowledged as a very difficult problem in anomaly detection. False positives are generally inevitable for training-based anomaly detection systems. However, in our experiments, we show that training for our scheme is effective and as long as user performs reasonable amount of training, false positive rate is very small.

To train our daemon programs, we used training scripts that generate commands to exercise the programs as in [92][31]. These scripts generate a random sequence of mostly valid commands, interspersed with some invalid commands. The distribution of these commands along with their parameters is set to mimic the distributions observed under normal operations. We keep exercising the daemon programs using training scripts until no new normal n-jump paths are discovered for a long time. Then we think training

is done and n-jump paths learnt during training are all normal n-jump paths of the program. Figure 40 shows convergence graph in terms of percentage of normal 9-jump paths learnt for three relatively big daemon programs (sshd, httpd and sendmail). The figures are plotted against the number of jump instructions executed in the program being learnt. The graph uses a linear scale on Y-axis (percentage learnt) and a logarithmic scale on X-axis (number of jump instructions processed). The curves are smoothed.

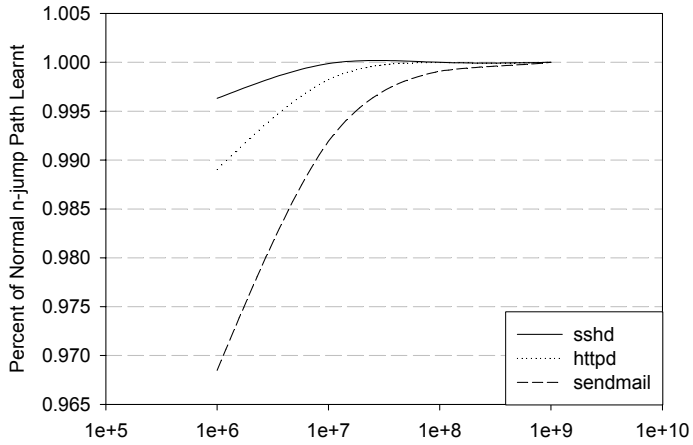


Figure 40. Convergence on selected benchmarks.

Convergence speed is an important measurement since it governs the amount of training time required to achieve a certain level of false positive rate. Slower convergence speed means that the user has to spend more time to train the system. Figure 40 shows for all three big daemon programs, normal 9-jump paths converge after about a few hundred million jump instructions are encountered and processed at runtime. No new n-jump paths can be learnt even we train the system for a much longer time. Convergence on smaller daemon programs are not shown since they converge even faster.

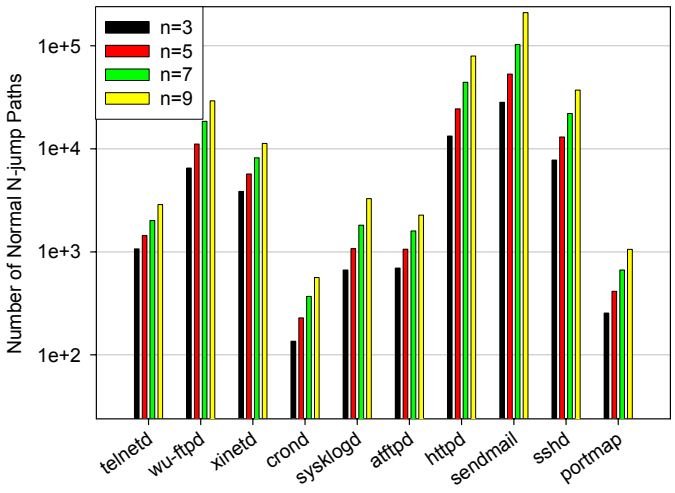


Figure 41. Number of n-jump paths.

The above results show that training for our anomalous path detection is very effective. One fundamental reason of the fast convergence rate is that although the number of potential n-jump paths is huge (the number of static branches multiplies 2^n), the actual n-jump paths seen during program execution is very limited. Figure 41 shows the absolute numbers of n-jumps paths learnt after convergence. It is clear that most benchmarks have a very small number of n-jump paths even they are run by billions of instructions. The result is not surprising. A lot of work, for example [42], has shown that a significant part of branches are highly biased and many of them are always taken or not-taken during the program execution. Another important reason is a well-known fact that most program execution time is spent on a small number of program hot paths, which can be easily learnt through training.

In [92] and [31], the authors also estimated false positive rate of their systems. They modified the training scripts slightly in terms of commands distribution and other parameters, and then used the modified scripts to test against their anomaly detection system running in detection mode after training. We tried the same method in our

experiments. We found that even with modified training scripts, no false positives are observed for all daemon programs after sufficient amount of training (after convergence). We are aware that training scripts cannot capture normal user behavior completely. When applied in real world, it is very likely that our anomalous path checking scheme will incur false positives. However, we are confident that the false positive rate would be very low and our system does not require excessive human interferences.

A good anomaly detection system has to achieve very low false negative rate too, i.e., very high anomaly detection rate. However, it is expected to be futile to measure detection rate of our anomaly detection system against traditional attacks (stack smashing etc.) as in the previous work, because those attacks should be easily prevented under our scheme. To prove that, we tested our scheme against our benchmarks with known vulnerabilities listed in Table 4. Most vulnerabilities examined are due to buffer overflows. Other vulnerabilities include format string attacks and unexpected program options etc. We found that our system successfully identified all the attacks because they cause anomalous paths. In addition, we tried the testbed of 20 different buffer overflow attacks developed by John Wilander [118]. The authors claim that the combination covers all practically possible buffer overflow attacks in which attack targets are the return address, the old base pointer, function pointers or long jump buffers. In all cases, the attacks were detected.

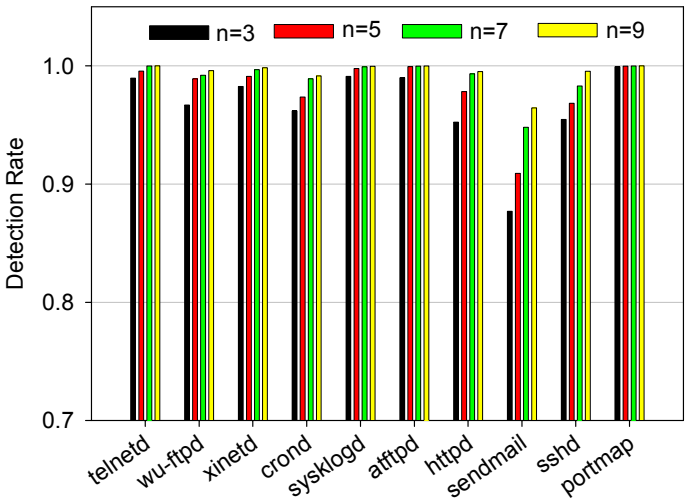


Figure 42. Detection rate of randomly inserted anomalous paths (daemon programs).

To further measure detection capability of our scheme, we choose to perform random fault injection experiments. Fault injection is a classical method for security analysis. A detailed discussion of it can be found in [39]. During the execution of benchmark programs, we randomly pick up branches to inject faults to divert their directions. The total number of faults injected during the execution is huge. For each randomly chosen branch, we divert the branch from its correct direction/target. Such diversion may or may not create an anomalous path. If it does, we check whether our anomaly detection system is able to detect that anomalous path. If our anomaly detection system fails, that is a false negative. Note that each fault is isolated to others. For each fault (each branch direction diversion), we check whether our scheme can detect it. Thus, we achieve a similar effect of running the benchmark numerous times and in each execution injecting only one fault and detecting it using our scheme.

We cannot find enough number of real attacks so we choose to use fault injection to evaluate our anomalous path checking scheme more rigorously. Although a fault is not a real attack, we believe that our fault injection model provides an extensive coverage to

the real attacks altering program control flows. Our fault injection model assumes that any dynamic branch direction/target can be tampered with any value. Thus it represents the worst case. For example, classical buffer overflow attacks overflow a buffer to tamper interesting control-deciding data, such as function return addresses, function pointers, exception handlers etc. The final effect that those attacks aim to achieve is to change the target address of an indirect branch instruction to a desired value, which is obviously covered by our fault injection model. Moreover, buffer overflow attacks normally are not able to tamper a specific memory location in an isolated way. They tend to tamper a continuous memory region. So buffer overflow attacks tend to lead to multiple anomalous branches under our anomalous path checking scheme, which makes detecting those attacks much easier actually. Another category of attacks including heap overflows, format string attacks and etc. enable the attacker to tamper any memory location in an isolated way. The effect achieved is that the attacker is able to modify specific control deciding data with a desired value at a certain program point. Such attacks are more accurate than buffer overflows but they are still covered in our fault injection model.

Figure 42 shows the results of our random fault injection experiments. The detection capability of our system depends on the value *n* in n-jump path. As illustrated in section 4.3.5, a larger n leads to a higher detection rate. Our results show that when n=3, the average detection rate of randomly inserted anomalous paths is 94.1%; when n=5, the average detection rate is 97.4%; when n=7, the average detection rate is 98.9%; when n=9, the average detection rate is 99.5%. From the above results, our anomalous path checking scheme achieves very high detection rate and thus exhibits very low false negative rate. The fundamental reason is again that the number of normal n-jump paths is

very limited, as shown in Figure 41. Thus, the cases illustrated in Figure 33 are rare. The cases are even rarer with a larger n. False negatives are due to the reason explained in section 4.3.5.

To enforce our results further, we also performed the same fault injection experiment on SPEC2000 integer programs. Normal profile for each program is collected using a single standard training input data set. We found that our scheme got very good detection rate again for such complicated programs and with such limited training. For example, with n=9, the average detection rate is 99.3%.

Next, we evaluate the performance aspect of our anomalous path checking scheme. To measure performance accurately, we collect the instruction trace during program execution and feed the instruction trace into a cycle accurate processor simulator Simplescalar [11] targeted to X86 [114]. All hardware modeling is done in Simplescalar. Each benchmark is simulated in a cycle accurate way by 2 billion instructions. The default parameters of our processor model are shown in Table 5. We select x86 based processor model primarily due to its popularity. Another important reason for using X86 is that we could easily find a full system emulator for it.

Table 5. Default Parameters of the processor simulated.

Clock frequency	600MHZ	L1 I-cache	32K, 32 way, 1cycle 32B block
Fetch queue	8 entries	L1 D-cache	32K, 32 way, 1cycle 32B block
Decode width	1	L2 Cache	none
Issue width	2	Memory latency	first chunk: 30 cycles, inter chunk: 1 cycles
Commit width	2	Branch predictor	bimod
RUU size	8		
LSQ size	8	TLB miss	30 cycles

There is no performance comparison with software-based approaches since no current software-based anomaly detection system is able to achieve the same security strength as our scheme. Moreover, even with weaker detection capability, software-based solutions already suffer from huge performance degradation, please refer to [115] for detailed numbers. On the other hand, our design only degrades performance slightly.

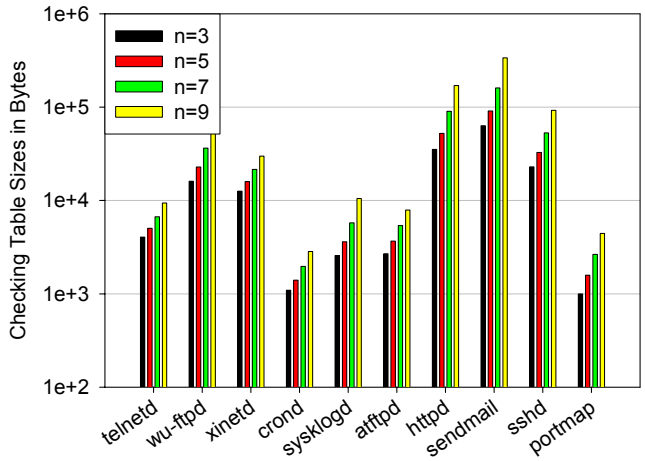


Figure 43. Optimized Checking table sizes.

Performance degradation under our scheme is mainly due to additional memory accesses to checking tables. Checking tables record a program's normal behavior. Intuitively, larger checking tables lead to higher memory system pressure and larger performance degradation. Figure 43 shows optimized sizes (in bytes) of checking tables for each benchmark with different values of n. Checking tables become large with a larger n, since there are more n-jump paths to record and information for each n-jump path takes more space. Generally, the size of checking tables is in terms of tens of KB and is small.

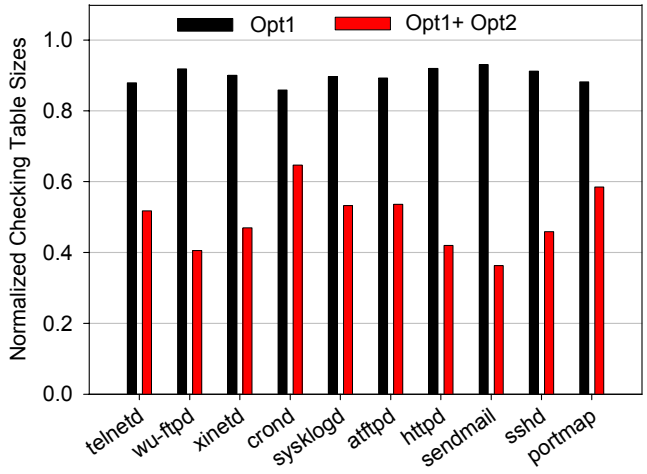


Figure 44. Effects of optimizations to checking table sizes.

Figure 44 shows the effects of our optimizations to reduce checking table sizes. Opt1 denotes the optimization that uses offsets to represent target addresses. Opt2 denotes the groupwise hash table optimization. Checking table sizes are normalized to the naïve case without any optimization. From the results, by using the offset to the base address of the code section to represent a branch target address, we reduce checking table sizes by 10.1% on average. If both optimizations are enabled, we reduce checking table sizes by 50.7% on average. The results show that our optimizations, especially groupwise hash table optimization, are highly effective to reduce memory requirement of checking tables.

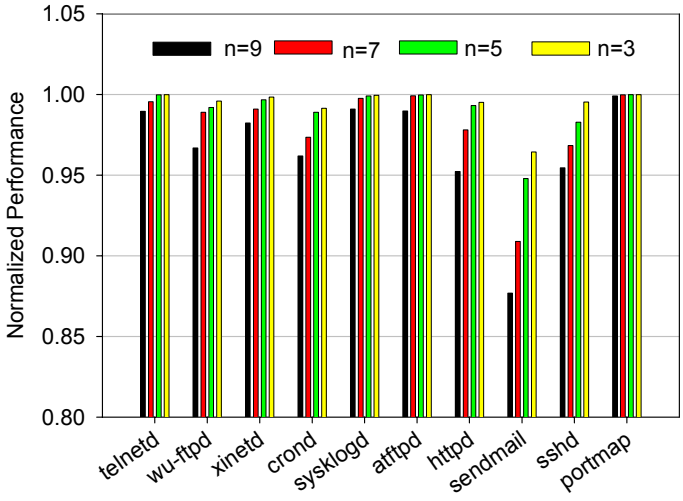


Figure 45. Performance results.

Figure 45 shows performance results. Performance degradation under different values of n (the length of paths checked) is shown. The IPCs (instruction per cycle) are normalized to the baseline processor without anomalous path checking. With a larger n, detection strength is stronger, but the size of path checking tables will be larger, leading to degraded cache performance and overall performance. When n=9, the average performance degradation is 3.4%; when n=7, the average performance degradation is 2.0%; when n=5, the average performance degradation is 1.0%; when n=3, the average performance degradation is 0.6%. It is clear that our hardware implementation keeps performance degradation minor even when n is large. Benchmark sendmail is a lot larger than other benchmarks used, thus it always has significantly larger performance degradation than the others. We also measured performance degradation for SPEC2000 integer programs. As in the fault injection experiment, training is done using standard training input data sets. We found that the performance degradation is also minor for SPEC2000 integer programs. In particular, when n=9, the average performance degradation is only 2.7%.

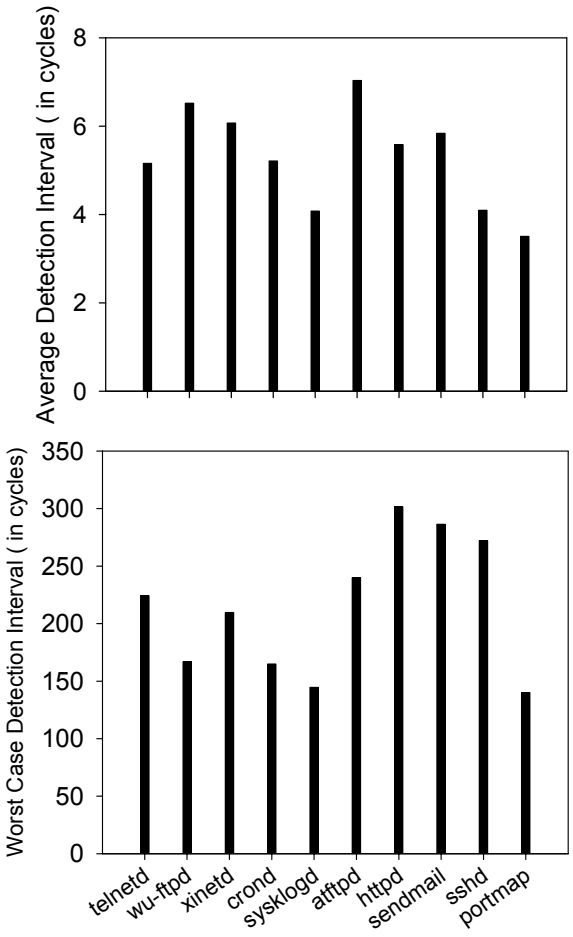


Figure 46. Intervals between fault injection and fault detection when n=9.

Finally, fast attack detection and response is another important advantage of a hardware-based anomaly detection scheme over software-based schemes. In our fault injection experiments, we also measured the time interval between fault injection and fault detection when n=9. Figure 46 shows the intervals in cycles in the average and worst cases. Cycle numbers are collected in the same way as in performance evaluation. We found that in most cases the simulate attack is detected immediately after the branch direction is tampered, so the average interval is only several cycles. In worst cases in which we can still detect the simulated attack, the attack is detected only after additional n-1 jumps are executed. The interval in worst cases can reach a couple of hundreds of

cycles. The intervals will be much larger for a software-based anomaly detection scheme due to its coarse monitoring granularity. For example, a buffer overflow attack will not be detected by a system call monitoring based scheme until the next system call is encountered, which may be millions of cycles away.

4.6 Summary

In this chapter, we show that a fine monitoring granularity is critical for an anomaly detection system to achieve a strong detection capability and it is infeasible for a purely software-based anomaly detection system to achieve a very fine monitoring granularity. Thus, we introduce micro-architecture level support for anomaly detection in this chapter and propose a training based scheme to detect anomalous program paths caused by attacks. Anomaly detection based on training is a classical approach. But with carefully designed hardware support, our approach offers multiple advantages over software-based solutions including negligible performance degradation; much stronger detection capability and fast reaction upon an anomaly thus much better security.

We show that our anomalous path checking scheme can detect almost all of the simulated attacks (above 99%) as long as the attack tampers program control flows. Moreover, the performance degradation of our anomalous path checking scheme is very low mainly due to the efficient design of our staged checking pipeline. A second reason for low performance degradation is that checking for attacks is done in parallel with the instruction execution, causing almost no extra overheads. The hardware components introduced by our anomalous path checking scheme are lightweight and through optimizations, we are able to reduce the memory space overhead of checking tables recording normal program behavior by almost 50%.

Our hardware-based anomalous path checking scheme can be applied to both general-purpose computing systems and embedded systems due to its low performance cost and hardware cost. At the same time, we also acknowledge that the requirement of good training and potential false positives in this scheme would limit its applicability in reality. However, the fact is that most existing anomaly detection schemes are training-based. Both training and false positives imply user involvement, thus increasing the overall cost of deploying such a system. When the target applications are easy to train or the user is willing to invest time on training, our anomalous path checking scheme could be an attractive choice to detect intrusions.

5 INFEASIBLE PATH DETECTION WITHOUT FALSE POSITIVES

Training-based anomalous path checking scheme is the first hardware-based anomaly detection scheme we proposed. Although the anomalous path checking scheme achieves strong detection capability with minor performance degradation, it is based on training thus good training is required to reduce false positives. However, training brings great burden to user and user may not even know how to train the anomaly detection system properly. Even the detection system is trained in a proper way, it is infeasible to exercise every possible normal input to the protected program, especially for large and complicated programs, so false positives are inevitable. The problem of false positives will be even more severe when we monitor the program at a much finer granularity. Take the anomalous path checking scheme as an example. There will be a dynamic n-jump path roughly every 10 dynamic instructions. Even though the false positive rate is extremely low, the absolute number of false positives could be still significant.

False positives have to be handled by human and will bring great burden to the management of the security system. False positive is the most important reason why current intrusion detection systems are still dominated by signature-checking based approaches. The potential false positives of the anomalous path checking scheme could limit its applicability significantly, which prompts us developing an anomaly detection scheme without false positives at all.

Zero false positives means that every alarm raised indicates a real attack. Training based approaches can never guarantee zero false positives. To achieve zero false positives, the anomaly detection system has to be based on static program analysis. In this

chapter, we propose a novel anomaly detection scheme utilizing both compiler static analysis and hardware runtime support. We still focus on monitoring dynamic program control flows. However, instead of trying to differentiate normal dynamic control flows from anomalous ones, we identify invalid control flows caused by attacks. By invalid, we mean that the dynamic control flow is infeasible according to the original program semantics. Thus, whenever such an infeasible control flow occurs, there must be an attack. Our observation is that attacks tend to lead to invalid program control flows because the attacker's goal is to manipulate the program state and lead the program to an execution state that it is not supposed to be.

Thus, we propose another anomaly detection scheme called infeasible path detection. We systematically analyze how to utilize compiler analysis to construct the correlations between branches. The collected information is then made available to the runtime system and is used to detect dynamic infeasible program paths caused by attacks with additional runtime information. The infeasible path detection scheme works with low runtime overhead and all alarms reported guarantee that certain memory content must be corrupted thus an attack must happen. In other words, the scheme is able to avoid false positive completely which is one of the major limitations of current anomaly detection schemes.

5.1 Background and Motivation

Let us revisit the example we used to motivate our anomalous path checking scheme. In this attack, the attacker tampers certain program data through buffer overflows and manipulates dynamic program paths maliciously. Through training, the anomalous path checking scheme can learn that the manipulated dynamic path is never

seen during normal executions thus is able to detect the attack. However, a close examination reveals that we do not actually need training to figure out the paths shown in Figure 47.c are anomalous. According to the program code, variable *user* should not be modified between the two if statements, thus the two if statements should always give the same result. This is just a simple example of a common phenomenon called branch correlation, which has been extensively studied previously [78][10]. Branch correlation is the foundation for the branch predictor inside a modern processor to predict branch directions. Even without runtime information, a compiler can still derive a lot of branch correlation information statically, which is the foundation of our infeasible path detection scheme.

```
1.    char str[SIZE], user[SIZE];
2.    ...
3.    if (strncmp (user, "admin", 5)) {
4.      ...
5.      no syscall;
6.    } else {
7.      ...
8.      no syscall;
9.    }
10.   strcpy (str, someinput);
11.   if (strncmp (user, "admin", 5)) {
12.     ...
13.     no syscall;
14.   } else {
15.     ...
16.     no syscall;
17.   }
```

(a)

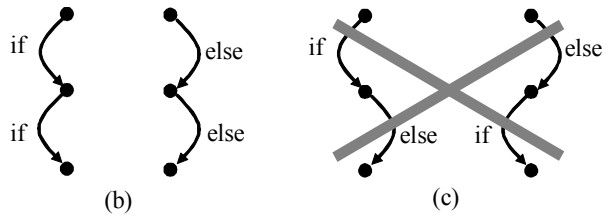


Figure 47. Example attack revisited.

Our basic idea is to detect dynamic infeasible paths with compiler's help at runtime, which could be caused by attacks tampering program state. Our basic assumption is that data should remain the same when it is fetched back from the memory after the previous access. The assumption is obvious. The compiler derives which paths are feasible or infeasible based on this assumption. We give a simple example in Figure 48. In Figure 48, if the path goes from BB1, BB2 to BB4, then the backward branch must be taken at the end of BB4 looping back to BB1 because we know x<0 at the end of BB1, so it must be less than 10 as well. If we see a path goes from BB1, BB2 to BB4, but not from BB4 to BB1, the variable x must be corrupted when it is loaded back from the memory. Also, the execution must goes to BB2 in the second iteration, since we know x is not changed, therefore the branch at the end of BB1 should take the same direction.

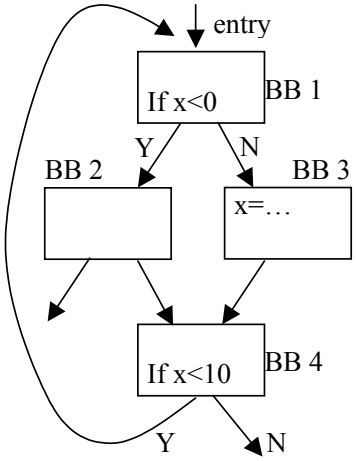


Figure 48. An infeasible path caused by memory tampering.

Note that in general program state or memory tampering may not necessarily lead to infeasible paths. However the attacker frequently needs to tamper critical variables and cause critical branches to deviate from its normal direction to achieve the desired (malicious) intent. Such deviations could manifest themselves through infeasible paths (as shown in the example in Figure 47). In the example, the attacker overwrites the

variable “user” by buffer overflows, so that it is different from its previous definition. However, the program is not aware of the tampering and still uses it to grant the privileged access. Thus, memory tampering caused by attacks could lead to dynamic program paths that are infeasible with respect to the program semantics. So the motivation is that if we can detect those infeasible paths, we can detect a range of attacks based on memory tampering, which is the most common starting point of attacks.

Comparing with previous software-based anomaly detection schemes, our scheme monitors program execution at a very fine granularity (dynamic control flows vs. system call traces) thus our scheme is able to detect certain attacks missed by previous system call monitoring based approaches. Another significance of our scheme is that our scheme is able to achieve a zero false positive rate because it detects anomalies based on knowledge collected by compiler infeasible path analysis. False positives are major obstacles for the wide deployment of training-based anomaly detection systems. A zero false positive rate is especially important for server applications where service continuation is crucial. Zero false positive is a crucial advantage of infeasible path detection over anomalous path detection elaborated in the last chapter.

5.2 Overview

Our objective is to provide a practical solution to efficiently detect infeasible program paths caused by attacks at runtime. Those infeasible paths indicate that an attack must have happened. There are several important issues to be considered in the design of such a solution. First, we want to avoid false positives completely, which is our major motivation for this scheme. In other words, our scheme is designed to report an alarm only when we are completely sure that there is an attack. Another issue is the detection

rate or the false negative rate. The system has to be able to detect a sufficient number of memory tampering based attacks to justify its cost. However, static compiler analysis has to be conservative. Thus, if we rely on static information solely, we could miss a lot of infeasible paths and memory tampering attacks at runtime. Finally, the performance overhead of the system has to be reasonably small; otherwise it is not practical. However, the number of dynamic program paths is huge. How to verify such a large number of dynamic paths with only small performance cost is also a challenging problem.

In this work, we propose to perform offline compiler analysis and then store the information collected compactly. The runtime system is armed with hardware support and detects infeasible program paths due to attacks by combining such static information collected by the compiler together with the runtime information collected during the program execution. By using dynamic information, we are able to detect dynamic infeasible paths more precisely and efficiently within the current execution context. In this manner, we not only improve the detection capability significantly (reduce false negatives) but also maintain good performance.

5.3 Branch Correlations

Each path consists of a number of branches and thus, branches provide the basic unit in path verification. In this section, we illustrate how branches affect each other with examples. It is well known that branches are not totally independent to each other but could be correlated. There are multiple ways how branches affect each other. Branch correlations have been extensively studied previously such as in [78][10]. We give an example in Figure 49.

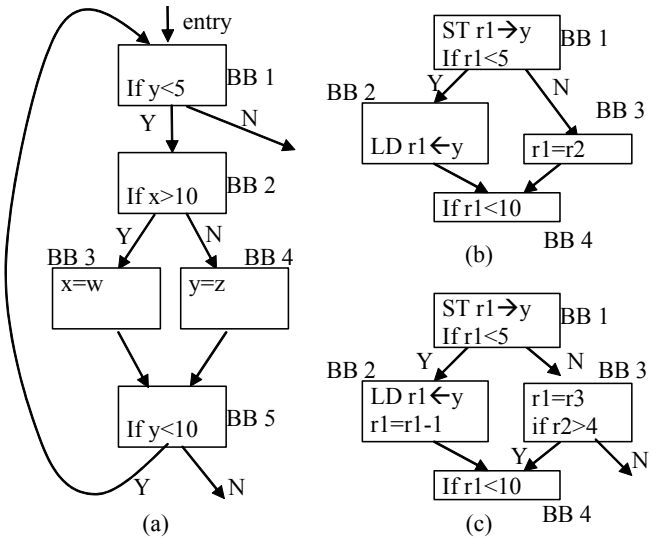


Figure 49. An example of branch correlation.

In Figure 49.a, we assume that all variables are memory resident and are not constants. The figure shows several interesting cases. If the branch in BB1 is taken and the path follows BB1→BB2→BB3, then BB5 should be taken as well, because y must be less than 5 if BB1 is taken, and along the path, y is not modified. Thus, the condition y<10 must be true. However, if the path follows BB1→BB2→BB4, then the direction of BB5 cannot be determined statically, since y is assigned a new value and normally we do not know whether this value is smaller than 10 or not. For variable x, it is used in BB2 in the conditional branch and is changed in BB3. Suppose the branch in BB2 is not taken for the first time, when the execution goes back to BB2 in the next loop iteration, we know this branch will not be taken, since variable x is loaded again without being changed from the last definition. If x is tampered in memory, it could lead to a different branch outcome and we should be able to detect that. However, if BB2 is taken in one iteration, x will be assigned a new value, which causes the branch outcome in BB2 to be unknown since normally we do not know whether the new value of x is larger than 10 or not.

From the above example, we can identify three scenarios in which branches affect each other. 1) The variables involved in a conditional branch are redefined somewhere before the branch is reached again. In such cases the outcome of the branch will normally become unknown. 2) The variables involved in the conditional branch are not redefined anywhere before the branch is reached again. In such cases the outcome of the branch should be the same as the last outcome. 3) A branch's condition subsumes another branch's. Here "subsume" is used [78] to indicate that if a variable in one range, then it must be in the other range, e.g., range [0, 5] subsumes range [0, 10]. The branch in BB1 subsumes the branch in BB5, because range $y<5$ subsumes range $y<10$. Note that scenario 2 can be regarded as a special case of scenario 3, in which one dynamic instance of a branch subsumes another instance (their ranges are same).

Figure 49.b shows how a piece of real code may look like as another example. In BB1, variable y is stored in memory. If the branch in BB1 is taken, we know that the value of y is less than 5. We also know that the load of y in BB2 should get a value less than 5 and the branch in BB4 should be taken. However, if the dynamic program path is BB1→BB3→BB4, r1 is redefined in BB3 and generally we do not have information for the new value of r1. Thus, to be conservative, we have to regard both taken and not-taken outcomes of the branch in BB4 as possible.

Finally, Figure 49.c shows that sometimes we are able to analyze more complicated cases. In Figure 49.b, after r1 is reloaded in BB2, there is no further redefinition to r1. However, in some cases, even after a variable is redefined, we can still have certain information about its new value, which is especially true if the new definition is done through simple arithmetic operations. In this example, r1 is decreased

by 1. After r1 is decreased, we still know that the branch in BB4 must be taken because y is known to be less than 5; after it is loaded and decreased, it should be still less than 5 and thus less than 10 as well.

Revisit the example in Figure 47, the second if-branch is subsumed by the first if-branch. Once the first if-branch is taken or not taken, the second one should follow the same direction. If it does not, there must be a memory tampering attack.

5.4 Implementation Details

This section elaborates our infeasible path detection scheme. We first introduce the main data structures such as Branch Status Vector (BSV) and Branch Action Table (BAT) then discuss how they are constructed and updated. Finally we give details of our Infeasible Path Detection System (IPDS).

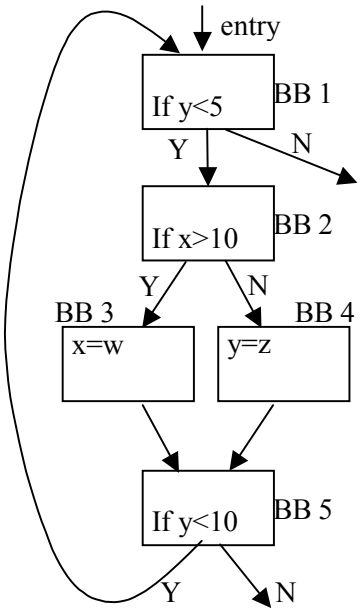
5.4.1 Branch Status Vector and Branch Action Table

The first task is to design proper data structures for the infeasible path detection scheme. In our scheme, we need to record the expected direction for each branch (obtained by combining both static information collected by the compiler and the runtime information), so that at runtime after the branch is executed and the actual direction is known, we can compare the actual direction with the expected direction detect anomalies. We introduce Branch Status Vector (BSV) for that purpose. Also, from the discussion of branch correlations, it is clear that whether a branch correlation exists at runtime heavily depends on whether certain variables get redefined and how they are redefined, which in turn depends on which dynamic path the program takes or the directions of dynamic branches. Thus, we need another data structure to record the interactions (correlations)

between branches. We introduce Branch Action Table (BAT) for that purpose, which is generated by the compiler through static analysis.

Since the majority of branches are conditional branches with two possible outcomes, we first focus on them. Multi-target branches will be discussed later. For each conditional branch, we need to keep two bits to encode three possibilities, namely taken, not-taken and unknown.

At runtime, after a branch instruction is executed, we first verify its actual direction with the expected direction. If they do not match, an infeasible path has been detected. If no mismatch is found, we update the branch status vectors (possibly including current branch's branch status vector) properly according to the actual direction of the branch and the branch action table.



Path	BR1	BR2	BR5
Init	UN	UN	UN
BB1	T	UN	T
BB2	T	UN	T
BB3	T	UN	T
BB5	T	UN	T
BB1	T	UN	T
BB2	T	NT	UN
BB4	UN	NT	UN
BB5	UN	NT	NT

Figure 50. An example on how to update the branch status vector.

We give an example on how to update the branch status vector in Figure 50. The example is the same as the one in Figure 49.a. In the example, three branches, located at

the end of BB1, BB2 and BB5 are present. To facilitate our discussion, we name them BR1, BR2 and BR5 respectively. Their branch status vectors are tracked on the right side of the figure. Initially, all branch statuses are UN, i.e. unknown. Assume the first dynamic instance of BR1 is taken. After the execution of the branch, we first verify the branch. Since "unknown" matches any direction so no mismatch is found. Then we update the status vectors. The status vectors of BR1 and BR5 need to be updated since BR1 subsumes BR5 and of course BR1 subsumes itself. So we set status vectors of BR1 and BR5 to "taken" as the current expected direction. Those expected directions are obviously optimistic. For example, BR1 will take the same direction when it is executed again only if y is not redefined in between. The basic idea here is that we first set expected directions optimistically (assuming that depending variables of branches will not be redefined), then we update the status vectors properly when we encounter redefinitions of those depending variables, which invalidate the previous expectations.

Let us go back to the example. The program path goes to BB2 and BR2 is taken at runtime. The status vector of BR2 is then set to "unknown" instead of "taken". The reason is that entering BB3 causes variable x to be redefined to an unknown value, which means, the direction of BB2 should become unknown now. Then the program path goes into BB5 and BR5 is taken. For the second execution of BR1, since there is no redefinition to its depending variable y, the branch is taken again and this matches the status vector. If BR1 is not taken at runtime, then an attack is detected. When BR2 is executed again, the status vector shows its direction to be unknown, therefore any direction is correct. This time, it goes to BB4, i.e. not-taken. This causes the status vector

of BR5 to be unknown, since variable y is redefined to some unknown value. The verification of BB5 is again correct since the status vector gives “unknown”.

During the discussion of the above example, we talked about various actions to update status vectors after a dynamic branch is executed and its direction is known. Those actions encode correlations between branches. How to construct and store those actions is the major job of the compiler. The actions are stored in a data structure called Branch Action Table (BAT). It records which branches’ branch status vectors should be updated and how to update them after a dynamic branch is executed.

In the simplest form (without concerns to the table size), we can store the BAT as a two dimensional array with br_num*br_num elements. Here br_num is the number of branches (we temporarily assume all branches are recorded, later we will talk about how to reduce that number).

After a branch is executed, we find out which branches are affected by this branch. For each affected branch, there are four possible actions: SET_T, SET_NT, SET_UN, NC, which means: “set to taken”, “set to not-taken”, “set to unknown” and “no change” respectively. It is clear that these four represent all possibilities. The reader can find examples of these actions in the discussion of the example in Figure 50. The actually number of actions for each [current branch, affected branch] pair is doubled, since there are two possible outcomes for the current branch, i.e. taken or not-taken. Different outcomes lead to different actions. In addition, as previously mentioned, not all branches need to be checked. If the compiler cannot infer anything in terms of correlation about the branch outcome, the branch can be excluded from checking. Therefore, we set up a

vector called Branch Checking Vector (BCV). It only stores which branches should be checked.

```
INPUT:   Program code in a function (CFG form),
         br_num: Num of Branches
OUTPUT:  BAT: The Branch Action Table
         BCV: Branch Check Vector

enum br_action { SET_T, SET_NT, SET_UN,NC}
BAT:= array [1..br_num][1..br_num][1..2] of br_action
BCV:=array [1..br_num] of Boolean

ALGORITHM: BAT_Construction
1.  Alias analysis and identify memory resident values.
2.
3.  Assume each store is a definition of the memory variable, construct
    the reaching definition information.
4.
5.  Foreach load l do
6.    Foreach branch bl whose outcome is inferrable from l's range do
7.      Foreach store s, whose definition reaches l do
8.        foreach branch bs whose outcome can infer s's range and s's
    range subsumes bl's range do
9.          set the action in BAT for bs to bl; mark bl in BCV;
10.       od
11.     od
12.     Foreach load lp, whose use immediately precedes l do
13.       foreach branch blp whose outcome can infer lp's range and lp's
    range subsumes l's range do
14.         set the action in BAT for bs to blp; mark blp in BCV;
15.       od
16.     od
17.   od
18. od
19.
20. Foreach branch br marked in BCV do
21.    Find all branches with definitions (other than the correlated
    loads) to the register used in br. Or other stores(other than the
    correlated ones) to the variable. Mark the action to br as UN in the
    corresponding entries in BAT.
22. od
23. return BAT, BCV
```

Figure 51. Algorithm to construct BAT and BCV.

The algorithm to construct the BAT and the BCV is shown in Figure 51. Note that the algorithm works on functions rather than on the whole program. It starts with alias analysis to identify all possible memory resident variables that are accessed by each load/store instruction. We first focus on those uniquely aliased variables. Multiple-aliased ones will be addressed later. Next, we perform reaching definition analysis for all store

instructions after alias analysis. Here, we regard each store as a definition of the variable in the particular memory location.

The main component of the algorithm is a nested loop. The first half of the loop handles correlation between one store and one load. We first need to clarify the relationship between a load/store and a conditional branch. Here we examine each branch whose outcome is inferable from a load's range, which means the branch direction is determinable if the loaded value is in a certain range. For example, for a code sequence like {… ld r1, x; … ble r1, 100; …}, if x is less than 100 and is not redefined between the load and the branch, then the branch must be taken. For the example in Figure 49.b, the outcome of branch at the end of BB1 is clear if we know whether y is less than 5 or not. Thus, we first obtain a collection of branch-load correlations. For each of them, the branch outcome depends on the value range of the loaded variable.

For each load, we find the stores that define the same variable and whose definitions reach this load. We then find branches that are related to the store. We want to identify branches whose outcomes infer a value range of the same variable and the range (of the store) subsumes the range of the load. To put it in a simple form, our goal is to discover the follow relationship:

branch bs's direction➔store st's range;

store st's range subsumes load ld's range;

load ld's range➔ branch bl's direction;

In other words, such a relationship indicates that from branch instruction bs's direction we can know bl's direction exactly. Revisit the example shown in Figure 49.c. If the branch in BB1 is taken, it infers y<5. y<5 subsumes y<11. For the load in BB2, we

then know that y is less than 11. This infers that the branch in BB4 is taken, since r1=y-1<10. Thus, this formulation is also able to take the following case into consideration: after a variable is loaded into a register, the register participates in further calculations before it is used in a conditional branch.

The second half of the loop handles correlation between two consecutive loads. The two loads must access the same variable, and after the first load the variable must keep alive between the two loads. The reasoning here is similar to the one discovering correlation between one store and one load. The goal is to discover the following relationship:

branch blp'direction➔load lp's range;

load lp's range subsumes load l's range;

load l's range➔ branch bl's direction;

In the example in Figure 49.a, if the path follows BB1→BB2→BB3→BB5, variable y is encountered again. The fact that the branch in BB1 (blp) is taken infers range y<5 (lp's range). y<5 subsumes y<10 (l's range). And y<10 infers that the branch at BB5 (bl) should be taken.

Note that the branch blp could be the same as bl. For example, in Figure 49.a, if the path follows BB2→BB4→BB5→BB1→BB2, variable x is encountered again. The variable's range is not changed. Therefore the direction of the branch at the end of BB2 should not change.

These two schemes basically follow our fundamental assumption that variables residing in the memory between either two loads or between one store and one load should not change. If the change happens due to tampering in memory caused by attacks

(such as by buffer overflows), it might lead to infeasible paths that will be detected by the above schemes.

Note that we also mark branches that should be checked in the branch check vector (BCV). We only mark branches for which the compiler can infer knowledge. In this way, the runtime system does not need to check every branch, which helps reduce overhead.

The last part of the algorithm checks all the definitions other than the correlated loads to the registers used in the branches that are marked in BCV. That is because if there are other paths coming to the branch and the register that is involved in the branch is changed along one of those paths, we must take proper actions accordingly, otherwise the checking mechanism will fail. We mark the branch direction as unknown if such as path is taken since either the compiler fails to derive a range for the redefined value or the range is not related to memory resident variables, which is not interesting in our case. This step also checks if there are other stores to the same variable depended by the branch, for which the compiler cannot establish their relationship with the branch. In such cases, we also mark the branch direction as unknown.

An example is shown in Figure 49.c. If the path follows BB3→BB4, we should mark the branch in BB4 as unknown, since r1 is defined in BB3. So when the branch in BB1 is not taken, we should immediately mark BB4's branch as unknown.

The complexity of the algorithm can be analyzed as follows. The main computation is in the nested loop with a depth of 4. It can be roughly estimated as O((#branches)2*(#loads)*MAX(#loads, #stores)). It is a polynomial-time algorithm. The execution time of the algorithm is normally small as measured in our experiments.

Till now, we have only considered variables that are not multiple aliased. However, sometimes the branch predicates are constructed out of pointer dereferences that could point to multiple memory locations. In such cases, our scheme must be conservative in order to avoid false positives. If the memory access instruction is a load, we simply remove it from further analysis. In this way, no branch will be incorrectly checked, since we do not infer anything from the loaded value. It might be possible that a load comes after this one is now (live range) connected to a load or store before it. It is still fine because the load (use of the variable) will not change the variable. If the memory access instruction is a store, we have to assume all aliases of the stored variable might be defined. Therefore all incoming definitions to those aliases from previous stores must end here. Also, loads that are reached by this store should not be considered to infer anything from this store, because we do not know which memory location the store actually modifies.

5.4.2 Storing BSV, BCV and BAT Information Efficiently

In last sub-section, we discussed what kind of information should be stored in BSV, BCV and BAT and how to collect the information. Next we need to find an efficient binary representation for those tables. We want to make the tables as small as possible to reduce the space overhead and other associative cost. In this sub-section, we discuss several ways to reduce the table sizes.

Using Proper Data Structures

The information stored in the tables is with respect to branches identified by PC addresses. Branches are non-uniformly distributed in the code and there could be an arbitrary interval between two adjacent branches. So the natural way to store the

information for branches is a hash table. The PC address of a branch is hashed then used as an index into the hash table. However we have to be careful when using hash tables. A common hash table requires tags to handle collisions. If we assume the function contains 1K instructions, the tag will have 10 bits. In other words, the size of the tag could be much larger than the information to be store in BSV and BCV.

To solve the problem, the compiler always finds out a hash function that leads to no collisions for the current function. The compiler achieves this by a trial-and-error method. It utilizes a parameterizable hash function with only shift and XOR operations. The hash function of course is the same one implemented by the runtime system. It first tries different hash function parameters to hash the branches into the optimally sized hash space. If that fails after several tries, the compiler enlarges the hash space and tries again. Obviously the hashing result should depend on the size of the hash space. The hash function has a lower probability to produce collision with a larger space. Since the number of branches in a function is normally not large, in most cases, the compiler can find a proper combination of hash function and hash space quickly. In that way, no tags are required in the hash tables since there is no collision at all. Note that the hash function parameter chosen by the compiler needs to be passed to the runtime system as a part of the information associated with a function.

Storing BAT information is more complicated since for each branch there are multiple and unfixed number of actions to be stored. We can use a large array to store actions for each branch but it is obviously inefficient, especially when there are a large number of branches in a function. Intuitively, each branch could only take action on a small number of other branches. Therefore, we chose a link list representation to avoid

space overhead for non-existing actions between two branches. The link list is implemented in an array. Each link list element is tagged by the branch index and stores the index of the next link list element. The collision free hash table for BAT contains the index of the first link list element for a branch.

Remove Unrelated Branches

Unrelated branches are those neither being checked nor acting on any other branches. Those are the branches that have nothing to do with our scheme. We can simple remove them from the BAT, BCV and BSV.

Reduce Branches Actions

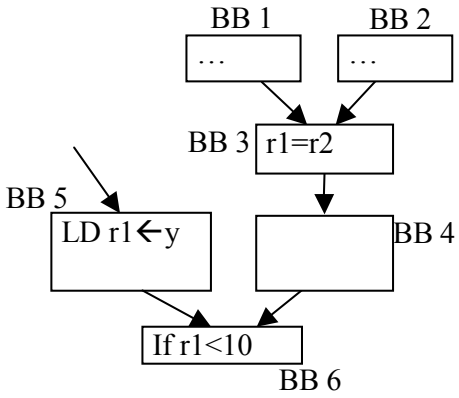


Figure 52. Reduce branch actions.

We found that in some cases actions for branches can be reduced. We show an example in Figure 52. We initially put an action in the BAT entries for both branches in BB1 and BB2, since whenever their path to BB3 is taken we should set the branch in BB6 as unknown. However we can instead put the action on the branch in BB3 and remove the original actions for the branches in BB1 and BB2. Note by doing that we do not degrade the detection capability because the status vector of the branch in BB6 is

only used when that branch is executed. On the other hand, we reduce the total number of actions, which could be helpful when the BAT is implemented as a linked list.

5.4.3 Handling Function Calls

Our BAT construction algorithm works on functions. We have not talked about how to handle function calls in the function processed. When there are function calls in the function processed, it becomes a bit more complicated, since variables might be modified inside the callee function. Therefore the function call site can act like a store to variables and in some cases statically we cannot know exactly which variables are modified by the callee function.

We choose to handle function calls in a simple and conservative way. First, we try to prove that the callee function only modifies non-local memory state through its pointer parameters, which should be true for most functions under good programming styles. Local memory state modified by the callee function will be discarded after the function returns thus does not matter. All standard C library function calls are specially handled since we know the exact semantics of those functions. For example, we know strcmp() will not change any non-local memory state and scanf() will only modify dereferenced objects of the second parameter and following. We try to prove the property for user-defined functions by examining the function bodies individually without relying on a full-fledged inter-procedural analysis. If the function modifies non-local memory state through global variables and/or pointer dereferences, to avoid the requirement of an advanced inter-procedural analysis, we can simply act conservatively and assume the call site can modify any variable. Of course it would be easy to optimize over this extreme conservativeness for the cases in which the function only modifies non-local memory

state by directly accessing global variables. After the above analysis, we can convert the function call into a list of pseudo store instructions for the purpose of our correlation analysis. For functions that do not modify non-local memory state, the list is empty. For functions that modify non-local memory state through pointer parameters, we create a pseudo store instruction for each dereferenced actual pointer parameter. For other functions, we create a store that could modify any variable. In that way, we convert the function calls into a list of (possibly multiple aliased) store instructions and how to handle (aliased) store instructions has been covered in section 5.4.1.

5.4.4 Other Considerations

Multi-target branches, typically indirect jumps, must be specially handled. The major difficulty is that multi-target branches can have an arbitrary number of targets. The BSV and BAT data structures for conditional branches are generally not usable by multi-target branches. Moreover, it is difficult to design special data structures for multi-target branches since to guarantee zero false positives, the data structures have to be able to accommodate an arbitrarily large number of targets. We believe that the additional detection capability does not justify the additional cost and complexity. First, the number of multi-target branches is relatively small. Second, many multi-target branches actually have only one or two possible targets, which could be identified by our pointer analysis pass. For example, in many cases function pointers are not mandatory but are merely used to achieve a flexible programming style. Thus, in our scheme we choose to only handle multi-target branches having at most two targets identified by the pointer analysis, and ignore other multi-target branches. In that way, multi-target branches handled can reuse the data structures for conditional branches. We only need to add a small table to

map one direction of the multi-target branch to “not-taken” and the other direction to “taken”. We also need to know when we should perform such mapping. Fortunately, for conditional branches, the 2 bits for each branch status vector are only used to represent 3 states: taken, not-taken and unknown. We can use the available fourth state to indicate that the branch is a multi-target one. Of course, to avoid the above additional overheads, the user can choose to ignore all multi-target branches. According to our experience, the degradation of detection capability is insignificant.

We assume that there is a pass to properly identify memory resident variables. This includes all scalar global/local variables, plus strings. We consider arrays on an aggregate basis (which is sufficient to tackle strings) but we do not undertake element by element analysis of arrays. Note that attacks seldom go through array elements. Instead, they change strings (aggregates) and important variables that make critical decisions such as indicating the user level etc. (see the previous examples in Section 5.1). Moreover, array comparison through strcmp(), strncmp() etc. are specially handled as well.

5.4.5 IPDS

Upon this point, we have introduced the basic components in our infeasible path detection scheme. Putting them together, the entire framework is called IPDS (Infeasible Path Detection System) and is shown in Figure 53.

The infeasible path detection works at the function level. BSVs, BCVs and BATs are constructed on a function basis. They are attached to the program binary by the compiler and mapped into a reserved memory space of the program once the program is loaded. The compiler conveys basic information for each function to the runtime system

through a function information table. The information includes entry addresses of BSV, BCV and BAT, the entry address of the function, hash function parameters etc.

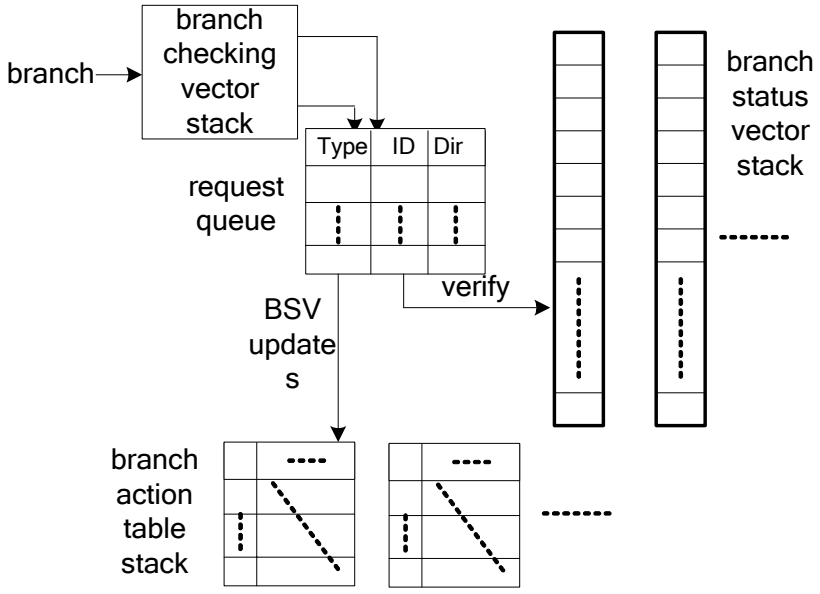


Figure 53. IPDS framework.

At runtime, each committed branch is sent to the IPDS. The IPDS first checks whether this branch is marked in the BCV. If it is, the IPDS queues a request to verify if the actual direction of the branch matches the expected direction in the BSV. The IPDS queues a request to update the BSV according to the current branch and the BAT table no matter whether the branch is marked in the BCV.

The sizes of BSV, BCV and BAT tables for each function are not fixed. The tables naturally form a stack at runtime. When we enter a new function, its corresponding BSV, BCV and BAT tables are push into the top of BSV, BCV and BAT stacks respectively. When the function returns, its BSV, BCV and BAT tables are popped up from the stacks then we can continue the checking based on the caller's BSV, BCV and BAT tables. To save on-chip space, we only keep the top of the stack on-chip but spill the non-active tables to their home location, which is similar to Itanium's register stack

engine. The spilled BSV, BCV and BAT tables require proper protection from tampering. The protection can be easily achieved by mapping them to a reserved space. Since only the IPDS is supposed to access those tables and the program is never supposed to access them, any attempted access from the program will be detected and prevented by the processor.

All requests are put in a request queue according to the order in which they are issued. One important observation is that as long as the requests are properly ordered, we can guarantee that the checking is correct. Even if the process of requests gets delayed due to long latency operations such as loading the spilled tables, we can allow the program execution to continue without any delay but queue all the requests in their originally order. Also, it is shown in our experiments that the average checking speed is normally higher than the program execution. In other words, our scheme normally will not slow down the program execution.

The hardware overhead of IPDS includes additional infeasible path detection logic and small on-chip buffers holding BCVs, BSVs and BATs. We believe the hardware overhead is reasonable as the advance of IC technology allows packing millions of transistors on a single chip. There will be plenty of hardware resources available to implement IPDS in a single chip. Moreover, chip multiprocessors become more and more popular and it is possible to program one of the cores to implement security functionalities.

5.5 Evaluation

All compiler implementations are done in SUIF [100] and MachSUIF [68] research compiler infrastructure. SUIF and MachSUIF provide extensive compiler

optimizations and analyses that are convenient to use. In particular, there is a publicly available pointer analysis pass for SUIF, which is based on the algorithm proposed in [119]. It is both context-sensitive and flow-sensitive. We found that the pointer analysis pass gives very accurate points-to information especially for non-heap data accesses, which is greatly helpful to our work. The compiler infrastructure also provides flexible frameworks to develop new analyses and optimizations. The compilation time of the algorithm in Figure 51 is up to a few seconds on a Pentium 4 2GHZ machine.

The evaluation of the IPDS system consists of two aspects: precision and performance. In this section, we show that the IPDS system achieves both good precision and low performance overhead.

Good precision means both low false positive rate and low false negative rate. The IPDS achieves a zero false positive rate since it always acts conservatively and only raises an alarm when it is completely sure that an attack is ongoing.

Good precision also means low false negative rate, i.e. high detection rate of dynamic infeasible paths due to attacks. Besides showing that IPDS can detect exemplary attacks illustrated in Figure 47, we want to measure the ability of IPDS more systematically. It would be ideal if we can evaluate our system against real-world attacks. Unfortunately most publicly available attacks are traditional code injection attacks through buffer overflows, format string attacks etc. Those code injection attacks have been extensively studied and numerous solutions have been proposed, such as our anomalous path checking scheme and [108][23] [33][63][73] to only name a few. Those code injection attacks are not the focus of our work. We expect them become less important when protection mechanisms against them are widely deployed. We are

interested in the type of attacks in which attackers tamper critical data such as decision making data, user IDs etc. rather than return addresses or function pointers. This kind of attacks does not involve code injection and are much harder to detect. Chen et al. presented an experimental study of this kind of attacks and shows that they are very realistic [17]. There are fives examples in Chen et al.'s work, which are the only real examples we can find. But after code injection attacks become less effective, we expect that attackers will focus more on devising this new kind of attacks. Among the five examples given in [17], four of them involves tampering data that does not change dynamic program control flow, for which our scheme is not designed to detect. For the attack leading to program control flow changes (integer overflow attack against decision-making data in [17]), our scheme is able to detect the attack successfully.

To measure the detection rate further, we choose to simulate real attacks that tamper memory state by exploiting program vulnerabilities. We then check whether our scheme is able to detect the simulated attacks. In our experiments, we implement our IPDS system in an open-source IA-32 system emulator Bochs [9] with Linux installed. We choose 10 server programs having well-known vulnerabilities as benchmarks to perform simulated attack experiments. The server programs and their original well-known vulnerabilities are telnetd(buffer overflow), wu-ftpd(format string), xinetd (buffer overflow), crond(buffer overflow), sysklogd(format string), atftpd(buffer overflow), httpd(buffer overflow), sendmail(buffer overflow), sshd(buffer overflow) and portmap(buffer overflow).

Format string vulnerability allows us tamper an arbitrary memory location, so we can launch as many attacks as we want to the vulnerable program and tamper a different

memory location in each attack. But buffer overflow attacks normally tamper a continuous block of memory and only allow us tamper local stack data. Moreover, each of the above server programs only contains a few buffer overflow vulnerabilities. To be able to perform enough number of attacks to measure the detection rate accurately, we manually introduce more buffer overflow vulnerabilities into the server programs originally only having a few buffer overflow vulnerabilities. We further devise our attack to tamper only a specific local stack location rather than a continuous memory block. Otherwise, we would not be able to make attacks and results independent to each other in the buffer overflow cases. Buffer overflowing a specific location is possible in reality if the attacker knows the original local stack state, for example, by running the same program in his own machine. In the cases in which the attacker is not able to only tamper a specific local stack location, he will tamper multiple stack locations at the same time. Each tampered location can be used to detect the attack. Thus, our results for buffer overflow attacks represent the worst case. We make each of our attacks independent to each other to get a better feeling of the detection capability of our system.

Each server program is attacked 1000 times independently as explained above. The caused memory tampering may or may not change program control flow. If it does, we check whether the IPDS is able to detect that infeasible path. If it fails, that is a false negative. Our scheme is not designed to handle the cases in which the memory tampering does not change program control flow.

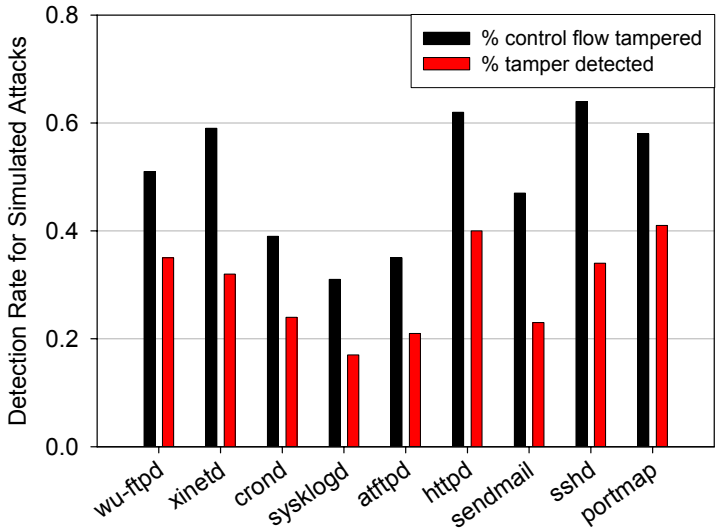


Figure 54. Detection Rate for Simulated Attacks.

Figure 54 shows the results of our simulated attack experiments to measure the detection capability of our system quantitatively. Our work is the first one to provide a quantitative view of detection capability and our results shown represent worst cases. None of the previous work quantifies the percentage of false negatives as we do. We show two results for each benchmark: the percentage of memory tamperings caused by attacks that actually change program control flow and the percentage of memory tamperings caused by attacks that are detected by our scheme. Intuitively, some memory tamperings do not have an effect on the program control flow. Our scheme is not designed to handle those cases. Actually, any scheme focuses on program control flow property will fail to detect those cases. How to detect memory tamperings that do not change program control flow is an open and tough problem [17]. From the results, on average 49.4% of memory tamperings cause changes in the program control flow and the IPDS can detect 29.3% of memory tamperings overall. Thus, the IPDS system is able to detect 59.3% of those memory tamperings that change program control flow. The

detection rate is very good, considering the IPDS is conservative and achieves a zero false positive rate. The high detection rate shows that strong correlation exits among branches. Especially, in many cases, IPDS knows dynamically the outcome of a branch should be the same as its previous outcome since the depended values should not be changed in between. However, after the tampering one depended value may be modified, then the branch may have a different outcome thus the tampering is detected.

Although the IPDS cannot prevent all memory tampering attacks, it detects a good percentage of memory tampering attacks that change program control flow. Moreover, the IPDS achieves a zero false positive rate, which is a critical property that makes the IPDS practical. False positive is the primary obstacle for the wide deployment of anomaly detection systems. That is why currently most commercial products still use signature based misuse detection rather than executing monitoring based anomaly detection. An anomaly detection system with a very high detection rate is meaningless if the user does not want to deploy it due to its frequent false positives. Also note that none of anomaly detection systems can detect all the anomalies. Our system is designed to combat a type of attacks largely ignored previously, which means it acts as a very good complement to the current anomaly detection systems. We also believe that since our system works on a much finer granularity, its detection capability is generally superior to the previously proposed system call monitoring based systems – we have in fact shown some attacks in the motivation section that are missed by those systems. Overall, we believe that our work provides a useful anomaly detection component to build the complete security system.

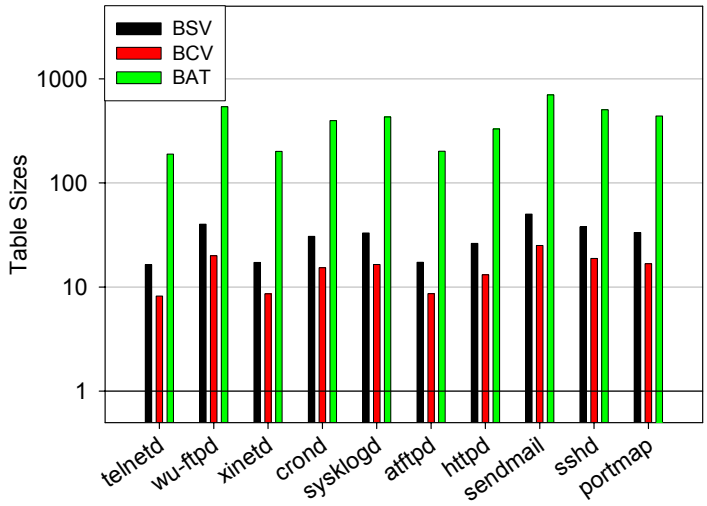


Figure 55. Average sizes (in bits) of BSV, BCV and BAT tables.

Figure 55 shows the average sizes of BSV, BCV and BAT tables in bits for functions inside each benchmark. Remember that BSV, BCV and BAT are constructed on a function basis. The graph is shown in log-scale. As discussed in section 5.4.2, the binary representation of the tables is carefully design to avoid the huge tag space overhead associated with normal hash tables. The average number of branches inside a function is normally smaller than 15. Each branch in a function only needs two bits for BSV information and one bit for BCV information. Although we cannot achieve optimal sizes in some cases, the sizes of BSV and BCV are very small. The average size of the BSV table for a function is 34 bits and the average size of the BCV table is 17 bits. Normally they can be packed into a couple of machine words. Now the advantage of our special hash table design should be clear. On the other hand, the size of BAT is much larger due to more complicated data structures. The average size of the BAT for a function is 393 bits. On average, 45% of the bits in BCV are marked. That is, the dynamic instances of those 45% of static branches will be checked at runtime.

Table 6. Default Parameters of the processor simulated.

Clock frequency	1 GHz	L1 I/D	64K, 2 way, 2 cycle 32B block
Fetch queue	32 entries	Unified L2	512K, 4way, 32B block Latency 10 cycles
Decode width	8	Memory bus	200M, 8 Byte wide
Issue width	8	Memory latency	first chunk: 80 cycles, inter chunk: 5 cycles
Commit width	8	TLB miss	30 cycles
RUU size	128	BSV stack	2K bits
LSQ size	64	BCV stack	1K bits
Branch predictor	2 Level	BAT stack	32K bits

Next, we focus on the performance aspect of the IPDS. To measure performance of the IPDS, we run the benchmarks in the Simplescalar toolset [11], which a cycle-accurate processor simulator. Each benchmark is simulated in a cycle accurate way by 2 billion instructions. The default parameters for the simulated processor are shown in Table 6. The IPDS hardware is also modeled in Simplescalar. We simulate a wide-issue high-performance processor to stress the IPDS. The on-chip buffers for BSV, BCV and BAT stacks are 2K bits, 1K bits and 32K bits respectively. We make the on-chip buffers large enough that in most cases they are enough to contain the information for branches in the active call chain, even for large server applications. For example, a BCV stack with 1K bits can record information for 1K branches. Assume there is one branch in every eight program instructions and each instruction is 4 bytes. Then the 1K BCV stack can cover 32KB code. The performance cost due to spilling of the stacks is minor. The total on-chip buffer space is only 35K bits, which is very small considering the on-chip cache size has reached several MB in modern superscalar processors. The access latency to the tables is one cycle. Note that we may need to access the BAT table (which implements a link list) several times to handle a BSV update request.

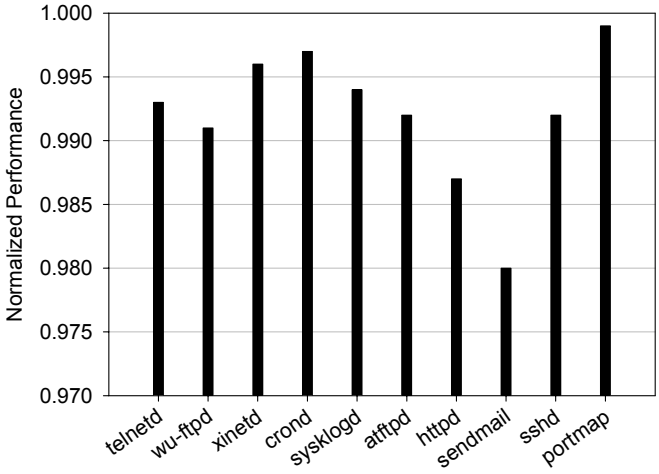


Figure 56. Normalized Performance.

In Figure 56, we show the performance for all benchmarks normalized to the baseline case without infeasible path detection. The average performance degradation is only 0.79%. In most cases, the performance degradation is negligible. The fundamental reason is that checking of infeasible paths normally does not need to block the original program execution. We design a request queue to handle the burst of requests. The normal program execution is only blocked when the request queue is full. Whether the queue becomes full highly depends on the speed of processing each request. Each branch check request only needs a read of the expected branch and a comparison. Each BSV update request needs a small number of reads from the BAT and writes to BSV. The request for a branch normally can be completed with a very short latency before next branch is processed. Note that we pay additional overhead during function calls/returns and multi-target branches. Finally, tables used in anomaly detection are treated as a part of process context thus we pay additional overhead during context switches. Context switches overhead is also modeled in our experiments. Because context switches are rare

events and the size of the additional process context is only around 4KB, we found that the performance overhead due to additional process context is minor.

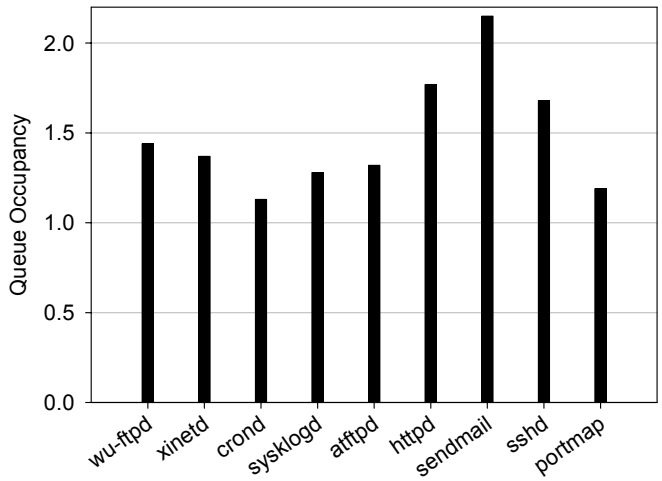


Figure 57. Queue Occupancy

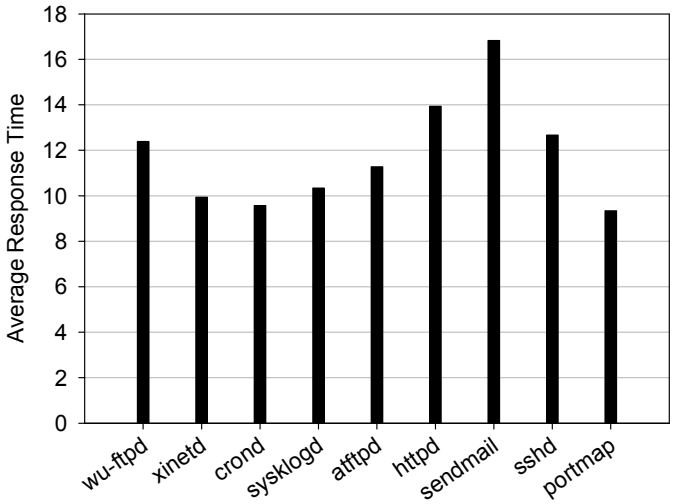


Figure 58. Detection Response time.

Next, we look at the occupancy of the request queue, which is important to understand the performance characteristics of our system. Figure 57 shows the average queue occupancy across benchmarks. It is about 1.47, which means at most of the time,

there are few requests being queued because normally the requests for a branch can be completed within a short latency before next branch has to be processed.

Finally, in Figure 58, we give the average response time in cycles, i.e. the time from a committed branch instruction is sent to the IPDS to the time the infeasible path is detected. On average, the response time is 11.7 cycles. In other words, the IPDS can quickly detect an infeasible path before it causes severe damage. This is an important feature especially for systems that would offer some intrusion-recovery mechanisms – if an attack is not detected quickly enough it would be too late to recover from. We perceive this is another important dimension of our solution – an ability to detect the attack in near real time.

5.6 Summary

Although our anomalous path checking scheme achieves strong detection capability and negligible performance degradation, it is based on training and potential false positive is a big concern that will limit its applicability in reality. In response to the concern of false positives, in this chapter we develop an infeasible path detection scheme to detect attacks based on static compiler branch correlation analysis and hardware runtime support. Our observation is that most attacks start from memory tampering and memory tampering could lead to infeasible dynamic program paths in terms of program semantics. In this scheme, the compiler analyzes correlations between branches and then the collected information is conveyed to the runtime system. The runtime system detects dynamic infeasible program paths by combining the compiler collected information and the runtime information.

The infeasible path detection scheme achieves a zero false positive rate which improves its applicability greatly. All alarms reported are guaranteed to indicate that certain memory content is corrupted and an attack has happened. We show that a good percentage of memory tampering (around 60%) can be detected as long as the tampering actually causes a change in the control flow. Moreover, the infeasible path detection scheme requires only a modest amount of hardware resources and the performance penalty of it is almost negligible. Overall, we believe that the infeasible path detection scheme can be a very practical intrusion detection solution, especially when low-cost and zero false positives are required at the same time. On the other hand, in this chapter we intentionally show that a large percentage of random memory tamperings (around 50%) actually do not alternate program control flows at all. Any control flow monitoring based anomaly detection scheme is completely helpless in detecting those memory tamperings. This problem prompts us to rethink the problem and seek a more complete solution.

6 DATA TAMPERING DETECTION THROUGH DYNAMIC ACCESS CONTROL

Most of the previous anomaly detection schemes, including our anomalous path checking scheme and infeasible path detection scheme, focus on control flow monitoring. They try to detect attacks by detecting anomalous dynamic control flows. However, the root cause of anomalous control flows is due to tampering to critical program data. In essence, anomaly detection schemes based on control flow monitoring try to infer data tampering by monitoring anomalous control flows caused by data tampering. However, in last chapter, we showed that a large percentage (about 50%) of random data tamperings do not alter program control flows at all. Recently Chen et al. [17] also discovered that a large category of attacks tamper program data but do not alter program control flows. Those attacks are not only realistic, but are also as important as classical attacks tampering control flows. Anomaly detection schemes based on control flow monitoring cannot detect those attacks at all. Detecting these attacks is a critical issue but has received little attention so far.

In this chapter, we propose an intrusion detection scheme with both compiler and micro-architecture support that detects data tamperings directly. The basic idea is that the compiler identifies program regions in which the data should not be modified according to the program semantics. The compiler performs analyses to determine modifications of data in different program regions and conveys this information to the hardware and the hardware checks the data accesses based on the information. If the compiler asserts that the data should not be modified but there is an attempt to do so at runtime, an attack is detected. The compiler starts with a basic scheme achieving maximum data protection

but such a scheme also suffers from significant performance overhead. We then attempt to reduce the performance overhead through different compiler optimization techniques. The optimization techniques include lazy protection points hoisting, protection points selecting, and protection operations aggregation. Our experiments show that this scheme achieves zero false positive rate, strong memory protection, and tight control over the performance degradation.

6.1 Background and Motivation

Most of the previous anomaly detection work [32][115][92][31][59][76][37][36][60][128] focuses on monitoring program control flows and program paths. Thus, one important question is whether monitoring program control flow is good enough to detect the incidence of an attack. First of all, memory tampering or data tampering is the primary starting point of attacks. Most if not all attacks start from memory tampering, such as buffer overflow attacks, format string attacks etc. In effect, most previous anomaly detection work tries to detect memory tampering by detecting anomalous program paths caused by the memory tampering. We checked the relationship between memory tamperings and control flow modifications. During our experiments, we found that around half of the memory tamperings do not alter program control flows. For those memory tamperings, monitoring control flows as in system call trace monitoring and anomalous path checking does not help at all. In [17], Chen et al. showed a large category of real attacks without any control flow tamperings. One real example is shown in Figure 59.

```
FILE * getdatasock( … ) {
        ...
        seteuid(0);
        setsockopt( ... );
        ...
        seteuid(pw->pw_uid);
        ...
}
```

Figure 59. Attacks without control flow tampering.

Wu-ftpd version 2.6 contains format string vulnerability. Format string vulnerability allows an attacker modify any memory location. Figure 59 shows a piece of innocent code from the Wu-ftpd source code. In normal situations, this piece of code will temporarily escalate the privilege using seteuid(0) to perform the setsockopt operation. Then it will restore the original user privilege after the setsockopt operation. But consider a scenario in which pw→pw_uid is tampered and is set to 0 through a format string attack. In this case, the second seteuid operation will always seteuid to 0 again, although it is supposed to restore the original user's privilege. So the attacker is able to retain root privilege and do all the damage subsequently. Note that there is no control flow tampering involved in this attack.

In [17], Chen et al. showed that this type of attacks is very realistic and should be given a serious attention. Generally, tamperings to various security-critical data could lead to such attacks. Security-critical data includes configuration data, user input, user identification data, decision making data and etc. In many cases, tamperings to the first three categories of critical data does not alter program control flows. *So the important point is that we have to detect tamperings to the critical data directly rather than trying to infer the data tamperings from control flow tamperings.*

```
ptr = NULL;
… …
if ( user_input1 ) {
        ptr = & obj1;
}
else if (user_input2) {
        ptr = & obj2;
}
… …
*ptr = …  // deref_inst
… …
```

Figure 60. Potential security holes in the AccMon approach.

The research on combating memory tampering attacks has been very limited. AccMon [131] is a software debugging tool with hardware support to detect software bugs leading to memory corruptions. The scheme is based on the observation that a memory location is typically only accessed by a few instructions. It works at the data object level. First, some data objects are selected to be monitored. Next they use profiling to build up a PC-based invariants table, recording the set of instructions that normally access a given data object. At runtime, the program execution is monitored through the compiler inserted monitoring operations. If a data object is accessed by an instruction not observed during training, an alarm is raised. Although AccMon is designed for software debugging, it can also be used in combating memory corruption attacks. The scheme relies on profiling/training, so it will have false alarms. Moreover, since it is based on profiling, an instruction will be regarded legal to access a data object when it does that during any normal program execution, even though under a specific execution, it may be illegal for that instruction to access the data object. This leaves potential holes to be

exploited by the attacker. Figure 60 shows a simple example. Assume that the simple program shown can accept two kinds of user inputs. If the user input is user_input1, the pointer ptr will point to obj1; if the user input is user_input2, the pointer ptr will point to obj2. Later, the pointer is dereferenced at deref_inst and the object pointed by the pointer is modified. Under the AccMon scheme, during profiling, both kinds of user inputs will be exercised and the deref_inst will be regarded as legal to access both obj1 and obj2. But during a specific execution of the program, the pointer can only either point to obj1 or obj2 but never point to both obj1 and obj2. So if the user input is user_input1, ptr is illegal to access obj2 but AccMon will regard it is legal. The attacker could tamper the value of ptr to the address of obj2 and gain illegal access to it without being detected by AccMon. Likewise, if the user input is user_input2, ptr is illegal to access obj1 but AccMon will regard it is legal.

Xu et al. propose a technique to detect memory corruptions attacks in [121]. Their basic assumption is that a randomized program usually crashes upon a memory corruption attack. They utilize the crash information to automatically identify the faulty instructions. In other words, they cannot detect attacks that do not result in a process crash. An example of such attacks is shown in Figure 59. So the protection strength of their scheme is limited.

Mondrian memory protection system [120] is a micro-architectural framework that enables fine-grained memory protection supporting multiple protection domains with acceptable performance degradation. It uses a single, shared address space with access permissions possibly at the granularity of per word defined in a permission table. At runtime, the value in the address register is used to lookup the permission table to see if

the domain has appropriate access permissions. The access permissions are preset by the user. Mondrian memory protection system provides an important reference in design of the micro-architecture component of our scheme.

In our scheme, we aim to tackle the root cause of memory corruption attacks by detecting corruptions to data directly. Our scheme achieves this by providing very fine-grained data access protection. Different from the Mondrian Memory Protection system, the access permissions are generated by the compiler automatically and access permission for a data object is managed and changed properly at runtime as per the program semantics.

Our basic idea is as follows. The compiler first identifies program regions in which certain critical data should remain unmodified performing a data flow analysis. Then the compiler sets the data as read-only in those regions through special operations with hardware support. Any attempt to modify the data marked as read-only triggers an alarm. This leads to superior attack detection strength. Moreover, since it is based on static data flow analysis, there are no false positives. Zero false positive rate greatly improves the applicability of the scheme. In the extreme case, we can always set the data as read-only right after each write to it and set the data as writable just before each write to it. This should provide best security but at the cost of performance degradation. There are many implementation issues and optimization opportunities, which will be detailed in the next section.

6.2 Compiler Analysis and Optimizations

Before we present the details of our scheme, we first introduce some important definitions.

Definition 1: A *memory object* is a data object defined in a program and resides in the memory. Memory objects include local stack data objects, static global data objects and dynamically allocated heap data objects. Common memory objects are scalar variables, arrays and aggregated structures etc.

Definition 2**:** An *access permission level* of a memory object is either writable, or read-only.

Definition 3**:** A *protection point* is a program point where the access permission levels of some memory objects are changed by setting the corresponding access bits for those memory objects.

Definition 4**:** A *protection operation* at a protection point is denoted as the action of changing the access permission levels of some memory objects at this point. Generally, there are two types of protection operations – changing from writable to read-only and changing from read-only to writable.

Baseline Scheme

First, we discuss a most basic implementation of our idea. In such a baseline scheme, every memory object has a corresponding access bit containing the access permission level for it. There are two types of protection points – just before a store instruction and right after a store instruction. Assume that a memory object "O" is written by the store instruction. The access permission level for O is changed to writable before the store instruction and is changed back to read-only after the store instruction. Initially, the access permission of all memory objects is set to be read-only. During the execution of the program, whenever a store instruction is to be executed, the hardware checks the access bit table to see whether this instruction violates the memory access permission, i.e.

writing to a memory object with access permission as read-only. If there is a violation, it alerts the attempted attack. Note that the memory corruption has to be done by some store instructions in a software-based attack. This store instruction could be some existing instruction in the program but writing to a memory object that it is not supposed to. A second possibility is that this store instruction could be injected by an attacker. In either case the tampering is detected by our scheme.

The baseline scheme can prevent memory corruption attacks completely. In other words, *all* memory objects can be protected from being attacked. But it is infeasible to realize this complete protection mechanism due to the very large overhead of setting and checking access permissions. Thus, we propose three compiler optimizations to reduce the performance overhead in a significant way while maintaining the memory protection strength at a fine-granularity level. We empirically show how strong protection can be achieved for typical attack models with minimized performance degradation.

Compiler Framework Overview

The compiler's task is to analyze the program, optimize the baseline scheme, collect the information about how to set up access permissions for memory objects properly and convey the information to the runtime component.

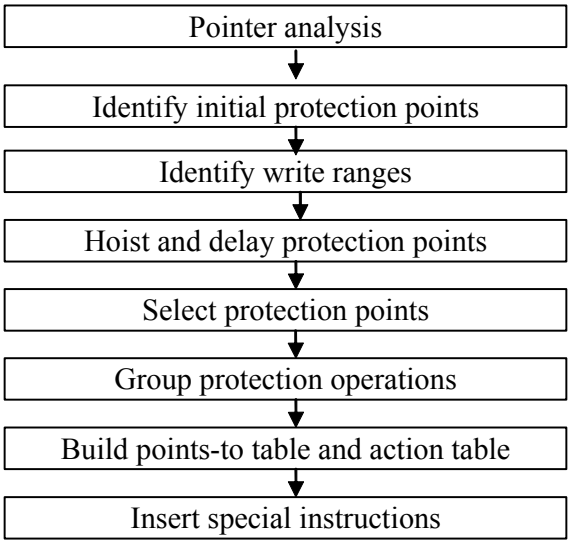


Figure 61. Compiler framework overview.

Figure 61 depicts our compiler framework. There are eight steps. First, the pointer analysis is performed trying to statically find out which memory objects a pointer is pointed to. Second, all store instructions are identified by the compiler. All program points right before/after store instructions are treated as initial protection points as we stated in the baseline. Third, all write range information is collected to facilitate further optimizations (we will talk about write ranges later). Forth, the hot protection points are hoisted/delayed to cold basic blocks. Then given some performance degradation constraints, the least beneficial protection points are removed. Next, remaining protection operations are grouped together whenever such opportunities exist to reduce performance overhead further. Next, an action table recording protection points and the corresponding protection operations is created. So is a pointed-to table used to assist the handling of pointer dereferences. Finally special instructions are inserted into the code to inform the processor when to look up the action table. Details about the above steps follow.

6.2.1 Write Ranges Identification

A write range is similar to the live range of a variable, except that the starting point of a write range is a store instruction on a memory object, and the ending point is the next closest store instructions on the same memory object. A write range is used to identify protection pairs – given a "set to writable" protection operation, what are its corresponding "set to read-only" protection operations, or on the other hand, what are the related "set to writable" protection operations for a "set to read-only" protection operation. The information is collected to assist the later optimization phases.

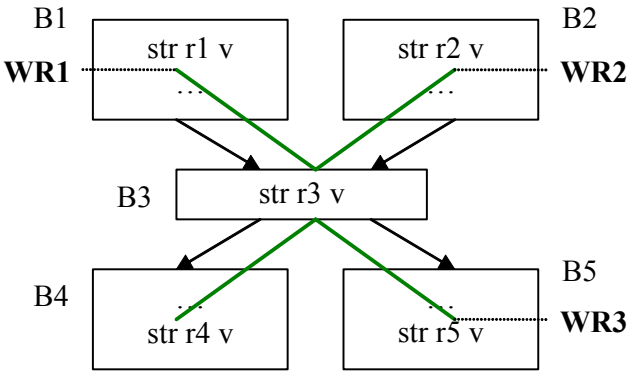


Figure 62. Write range example.

Figure 62 gives an example to show the write ranges of a variable v. There are five store instructions in five basic blocks, resulting in three write ranges – WR1, WR2 and WR3. One from "str r1 v" to "str r3 v", one from "str r2 v" to "str r3 v", and another from "str r3 v" to "str r4 v" and "str r5 v". Note that the three write ranges do not include the store instructions themselves. In other words, a write range starts at the point right after a store instruction and ends just before the nearest next store instruction for the same variable.

We utilize the standard framework of webs to identify write ranges. The starting and ending points of a write range are program points where protection operations should

be performed initially. In the above example, the access permission level for v should be set to read-only at the start of WR1, and it should be changed back to writable at the end of WR1. It is the same for WR2 and WR3. Thus, protection points are the boundaries of write ranges.

6.2.2 Protection Points Hoisting and Delaying

The large overhead of complete memory protection by setting access permissions at all protection points is mainly due to that some store instructions are executed many times. For example, in some multimedia encoding applications, the encoding process is done in a major loop executed a very large number of times; while other code outside the loop body is seldom touched. Hoisting/delaying a protection point out of the loop body would afford us more efficient protection mechanisms. This observation motivates us to come up with an optimization to move the protection operations from a hot basic block to a cold basic block whenever possible.

```
Func:     HoistPP
Input:    PP – Protection point set of a variable
          CFG – control flow graph of the program
          Profile – Hot/cold basic block information in the CFG
          Threshold – a threshold to make cost-benefit tradeoff
Output:   Optimized protection point set

For each variable v
    For each P ∈ PP[v] in a basic block B
        If B is hot
            For each preprocessor BP of B
                _push (BP, S)
            EndFor
            HoistSet = ProcStack
            //BP ∈ HoistSet
            Hoist_benefit = B.freq – ∑ BP.freq
            Hoist_cost = ∑ dynamic stores from BP to B
            If (Hoist_benefit/Hoist_cost > threshold)
                PP[v] = PP[v] – {B}
                PP[v] = PP[v] ∪ HoistSet
            EndIf
        EndIf
    EndFor
EndFor

Func:     ProcStack
Input:    PP – Protection point set of a variable
          CFG – control flow graph of the program
          Profile – Hot/cold basic block information in the CFG
          S – Stack for processing
Output:   HoistSet – Set of protection points that the original
           protection point is going to be hoisted to

HoistSet = Null
While (B = pop(S) != NULL )
    B.visit = true;
    If B is hot
        If ( ∃ P ∈ PP[v]) && (P is in B)
            return Null
        Else
            For each preprocessor BP of B
                If B.visit = false
                    _push(BP, S)
                EndIf
            EndFor
        EndIf
    Else
        HoistSet = HoistSet ∪ {B}
    EndIf
EndWhile
return HoistSet
```

Figure 63. Algorithm for hoisting protection points.

Figure 63 gives the pseudo code for our hoisting algorithm. Starting from a hot basic block containing a protection point, we go back along the edges in the control flow graph. Its predecessor basic blocks are pushed onto a stack for further processing.

ProcStack is an auxiliary function to explore the algorithm along backwards edges. It first examines the block on the top of the stack to see if it is also hot. A basic block is said to be hot if its execution frequency exceeds some preset threshold. If it is cold, then we get one desired location to insert the protection operation. Otherwise, we should check whether we have reached a boundary – a basic block that contains another protection point for the same memory object. In other words, the algorithm determines if we have reached a program point where the object's access permission level is just changed from writable to read-only, so we have to stop here and leave the hot protection point for further optimizations. If the current basic block that is being processed is hot and it does not include any protection operation on the same memory object, we continue pushing its predecessor basic blocks onto the stack. A control flow graph may include loops, so we use a flag to indicate whether a basic block has been visited before.

After obtaining the potential locations for hoisting a protection point, we determine whether to actually move the protection point out of the hot basic block based on a cost/benefit analysis. The benefit is denoted as the saving in executing protection operations dynamically. The cost is defined in terms of the security degradation. Hoisting a protection point to a predecessor point means that the object could be corrupted between the two points. So it is not protected in that region. If the ratio of benefit/cost is bigger than a threshold, then the protection point is hoisted. Currently the threshold is

determined by the system manually, which can be optimized using some heuristic solutions.

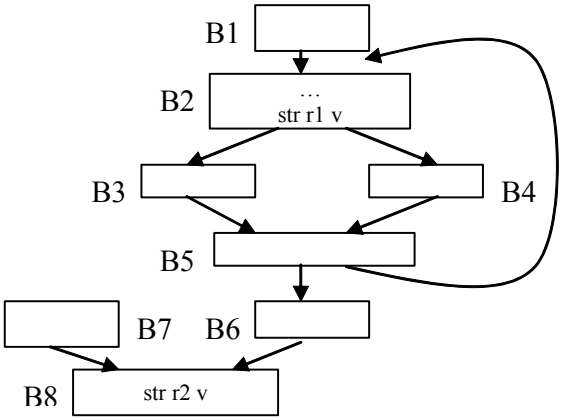


Figure 64. An example of hoisting protection points.

Figure 64 gives a sub control flow graph to show how the hoisting algorithm works. There are two store instructions in blocks B2 and B8, generating four protection points, right before and after the two stores. Assume all blocks in the loop body are hot. It is easy to see that the protection point before the store in B2 can be taken out of the loop to the block B1. That means we can set the access permission level for v1 to writable at the end of B1 instead of in B2. Similarly, if B6 is cold, the protection point can also be lifted from B8 to B6. On the other hand, if B6 is also hot, the hoisting algorithm has to proceed by processing blocks B5, B4, B3 and B2 until it meets another store for v in B2. Thus there are two possibilities for moving protection points from hot blocks to cold blocks in this example.

The benefit also comes with penalty – the degradation in security. If a write is only permitted just before the instruction "str r1 v", the variable v is well protected. If the variable v is set to be writable at the end of B1, the attacker can possibly corrupt variable v during the program's execution between the end of B1 and the store instruction. So a

tradeoff must be made to balance the benefit and the cost. On the other hand, if there is no other store instruction between the end of B1 to the store instruction to variable v1, the protection remains intact or does not degrade at all. This is because the attacker has to execute a store instruction to corrupt a memory object. Thus, some motions of protection points actually do not result in security degradation if they don't move the protection point past another store (which could be malignant).

The similar algorithm is applied to delay the protection operation of changing the access permission from writable to read-only from a hot basic block to a cold one. Look at the example in Figure 64 again. Initially, variable v is set to be read-only right after "str r1 v" in B2. It could involve great overhead since B2 is hot. So if B6 is cold, we could delay this protection to B6. However, there is a relatively long path from B2 to B6. So the protection to variable v may be degraded significantly. In this case, the compiler determines whether it should be delayed or not after considering both the benefit and the cost similar to that of hoisting a protection operation (i.e. if benefit/cost is bigger than the threshold, it is decided to be delayed).

6.2.3 Protection Points Selection

Due to the big overhead, completely protecting all memory objects may not be feasible. So how to determine where and which memory object should be protected are the key issues addressed in this section.

We build a cost/benefit analysis model to select protection points based on the profile data. The analysis unit is the set of protection points of a write range. All protection points for a write range have to be analyzed together since they are related operations. In other words, if a "set to read-only" operation for a memory object is removed, the

corresponding “set to writable” operations are not necessary any more. On the other hand, if only the “set to writable” operation is deleted, its corresponding writes are not checked. Otherwise, the corresponding writes will be regarded as illegal and there will be false alarms.

```
Func:    SelectPP
Input:   AllWR – write ranges set
         Threshold – performance degradation constraint
Output:  Optimized write ranges set

CompCostBenefit
// WR ∈ AllWR
performance_degradation = ∑ WR.cost
While (performance_degradation > threshold)
      WR ∈ AllWR with minimum weight
      AllWR = AllWR – {WR}
      Performance_degradation -= WR.cost
EndWhile

Func:    CompCostBenefit
Input:   AllWR – write ranges set
         Profile – Hot/cold basic blocks information in the CFG
Output:  Weight of each write range

For each variable v
    For each WR ∈ AllWR[v]
          WR.benefit = ∑ (dynamic stores within WR)
          WR.cost = ∑ (instruction at protection point in WR)
          WR.weight = tradeoff(WR.beneft, WR.cost)
      EndFor
EndFor
```

Figure 65. Algorithm for selecting protection points.

The benefit of removing a write range is denoted as the sum of the dynamically executed store instructions within the write range. In other words, the benefit corresponds to the protection offered to the number of stores in a given range and is therefore proportional to the number of stores. The cost of a write range is defined as the sum of

the dynamic execution times of all protection points in this write range. The cost is equal to the sum of the frequencies of basic blocks where the protection points are located.

Figure 65 shows the pseudo code for the algorithm to choose the most profitable memory objects to be protected. The algorithm first analyzes the cost/benefit for all write ranges. The weight of a write range is based on the cost/benefit tradeoff which indicates its priority in terms of protection. The weight is equal to WR.benefit/WR.cost. If WR.cost is zero, WR.weight is set to be infinity. Then, driven by the performance degradation constraint, the algorithm repeats removing write ranges until the degradation requirement is satisfied.

As an example, consider the control flow graph in Figure 64. If the "set to read-only" operation cannot be delayed from B2, we re-evaluate its benefit and cost in this compiler pass to determine if such a protection point should be deleted.

6.2.4 Grouping Protection Operations

In this section, we introduce a compiler technique called protection operations grouping. Normally, each protection point corresponds to one protection operation that incurs certain runtime overhead. However, interesting optimization opportunities exist when multiple protection points performing the same type of operations are clustered together and the memory objects they protect are adjacent in the memory. In these cases, instead of incurring one protection operation for each protected memory object, the protection operations can be grouped together as one protection operation for all memory objects at the *clustering point*. Figure 66 shows a simple example. Assume variables v1 and v2 are adjacent to each other. The start address of v1 is addr1 and the size of it is size1. The start address of v2 is addr1+size1 and the size of it is size2. Instead of setting

access permission levels of v1 and v2 separately, we can group the operations and set the access permission level of the memory region with start address as addr1 and with size as size1+size2. The grouped protection operation achieves better performance since it reduces the number of dynamic protection operations and the associated runtime overhead.

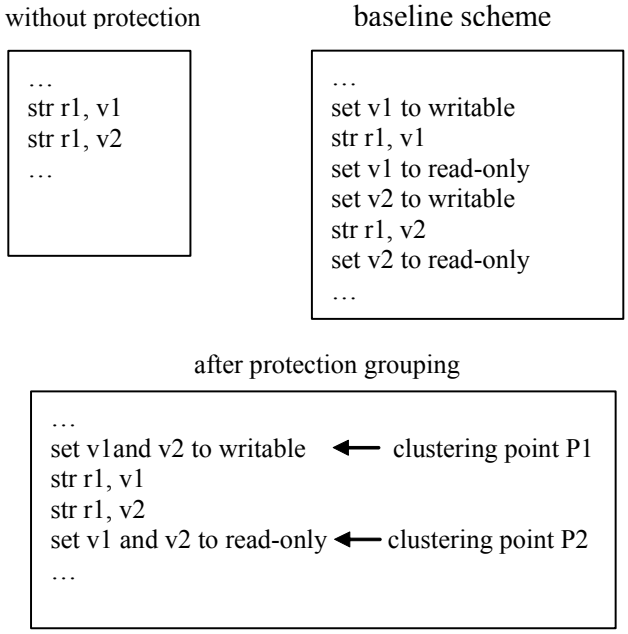


Figure 66. An example of protection operations grouping.

Such optimization opportunities occur when there are multiple protection operations clustered together and when the memory objects protected at the clustering point are adjacent to each other. We can easily create clustered protection operations when there are multiple adjacent store instructions as shown in Figure 66. In our scheme, the compiler tries to move store instructions together as long as their dependencies are still satisfied. Also, after the protection point hoisting/delaying algorithm is performed, many protection operations tend to be grouped at the same program point, such as the beginning/ending of a cold basic block.

After clustered protection operations are identified, we need to lay out the memory objects involved properly to exploit the optimization opportunities. Note that the layout of the memory objects is a global decision rather than a local decision with respect to a given protection point. Different memory layouts may be required to completely exploit protection operation grouping opportunities at different clustering points, but there could be only one memory layout for the program. Our goal is to find a layout of memory objects that can exploit the opportunities of protection operations grouping maximally.

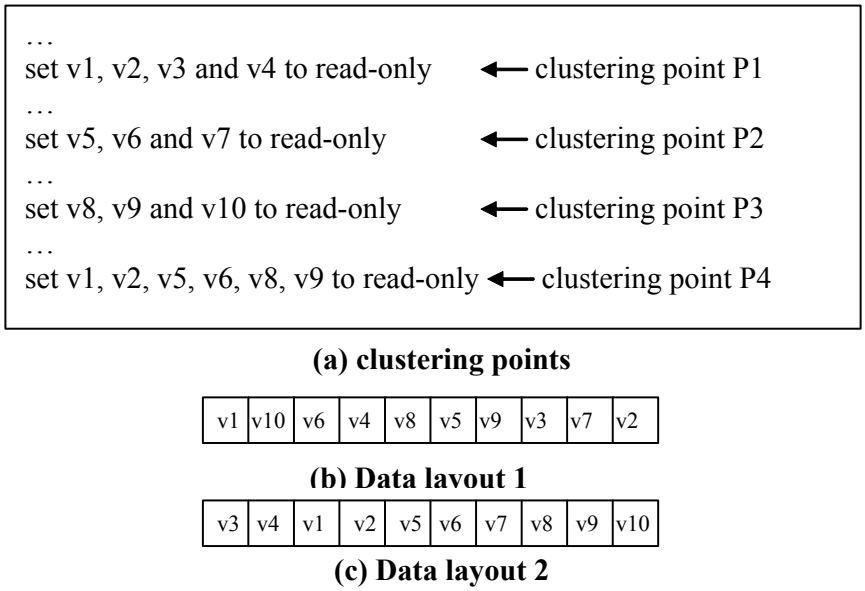


Figure 67.Data layout and protection operations grouping.

Figure 67 gives an example showing how the data layout impacts the protection operations grouping. There are nine variables and four clustering points in this example. Figure 67 (b) and (c) show two possible data layouts of these variables in the virtual memory. Data layout 1 only allows us to perform the protections on v5, v8 and v9 together at P4 since only these variables are adjacent and there are protection operations on them at the same clustering point (i.e. P4). On the other hand, data layout 2 allows us

to protect v1, v2, v3 and v4 together at P1; to protect v5, v6 and v7 together at P2; to protect v8, v9 and v10 together at P3; to protection v1, v2, v5 and v6 together at P4; and to protect v8 and v9 together at P4. The two data layouts show substantial difference in grouping protection operations, so our goal is to find out the best way to layout the data so that protection operations can be grouped maximally.

The pseudo code of our algorithm to determine memory object layout is shown in Figure 68. First, a variable group at a clustering point is identified. All objects with the same type, whose access permissions are changed in the same way at a clustering point form the variable group for the clustering point. Types of an object include initialized global object, un-initialized global object, heap object, and stack object. So there are at most four variable groups at a clustering point. Access permissions can be changed either from read-only to writable or from writable to read-only. The weight of a variable group is defined as the saving of dynamic protection operations if the protection operations for all of the variables in the variable group are grouped together at the given point. Then the algorithm starts from the variable group (say CurrVG) with the maximum weight. Note that we only deal with variable groups whose size is bigger than one since one element group does not provide any opportunity for optimization. Let ProcessedVG be the set of variable groups that have been processed. It is initialized to be null. CommonVG includes variable groups that have common variables with CurrVG. The motivation behind identifying the set of common variables is to maximize linearization of the variables in the group greedily so that more protection operations can be removed. We illustrate these concepts below through an example.

```
Func:   GroupVar
Input:  AllVG – set of all variable groups
Output: Optimized variable groups

ProcessedVG = null
While AllVG != null
      CurrVG = SelectMaxWeightVG
      AllVG = AllVG – {CurrVG}
      If (CurrVG.size > 1)
            CommonVG = {VG' | (VG' ∈ AllVG} & (VG' ∩ CurrVG != null)}
            ProcessedVG = ProcessedVG – CommonVG
            If (CommonVG != null)
                  ProcessedVG = ProcessedVG ∪ ArrangeData
            Else
                  ProcessedVG = ProcessedVG ∪ {CurrVG}
            EndIf
      EndIf
EndWhile

Func:   ArrangeData
Input:  CommonVG – variable groups having common element with CurrVG
Output: Arranged variable group

ListGroup = null
For each VG' in CommonVG
      CommonVar = VG' ∩ CurrVG
      For each v in CommonVar
            CurrVG = CurrVG – {v}
            ListGroup = ListGroup –VG'
            VG' = VG' – {v}
            NewListGroup = CreateListGroup(v, VG')
            ListGroup = CombineListGroup(ListGroup, NewListGroup)
      EndFor
EndFor
```

Figure 68. Pseudo code for the grouping algorithm.

We use the example in Figure 69 to explain our algorithm step by step. In this example, there are four groups VG1, VG2, VG3 and VG4 corresponding to the example in Figure 67. Their weights are decreasing. That means VG1 should be dealt with first, then VG2, VG3 and VG4. In processing VG1, both ProcessedVG and CommonVG are null, so VG1 is included in ProcessedVG. In processing VG2, since Processed VG only contains VG1 and there is no common variable for VG1 and VG2, CommonVG is also empty. VG2 is inserted into VG. Similarly, VG3 is also included in VG. Now VG

contains VG1, VG2 and VG3. The last variable group is VG4, which has common variables with VG1, VG2 and VG3, so CommonVG includes VG1, VG2 and VG3.

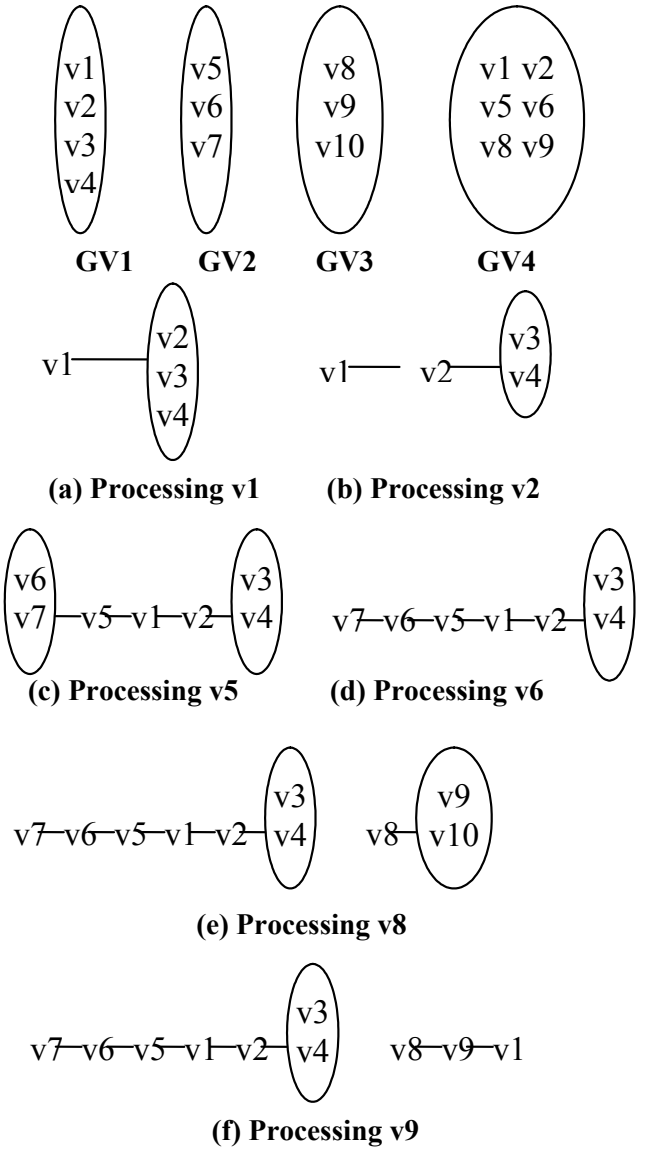


Figure 69. An example of protection operations grouping algorithm.

Next we show how to use the common variables to clarify the spatial relationship of variables. For each variable group VG' in CommonVG, let CommonVar be the set of common variables of CurrVG and VG'. For each variable v in CommonVar, it is taken out of the variable group first. The remaining variables in the variable group form a new

variable group. Next v is connected to the new variable group to create NewListGroup that is in a special form – listgroup. Listgroup is a special structure containing both lists and groups. A group means that variables in it could be grouped together, but it does not tell us the exact layout of the variables, i.e. which variable should be adjacent to which variable in the virtual memory. So the variable can be linearized in many ways. But considering all variable groups, a variable should be preferred to be the neighbor of another variable to optimize overall protection operations. So a list is designed to represent the linear data layout of the variables. Finally NewListGroup is combined with original ListGroup to build current ListGroup.

To make it more clear, look at the example in Figure 69 again. VG1 is taken from CommonVG first. CommonVar includes v1 and v2. Initially ListGroup is set to be empty. v1 is first removed out of VG1. Note that although the variable group has priority based on their weights, variables in the same group have the same priority. That means it does not matter if we deal with v1 or v2 first. Now the new VG1 only has v2, v3 and v4. NewListGroup is created containing two elements, v1 and the new VG1. Then NewListGroup is combined with ListGroup. Since the original ListGroup is empty, current ListGroup is the same as NewListGroup as shown in Figure 69 (a). v1 is taken out as a special element connecting to the new VG1 since VG4 indicates that it is preferable for v1 to be grouped with other variables later, such as v2, v5 to reduce protection operations.

The next variable in Common to be processed is v2. In this case, VG1 is picked from ListGroup. Currently VG1 is the set of v2, v3 and v4, so ListGroup only has v1. Then v2 is taken out of VG1 connecting to the new VG1 creating NewListGroup. Next

NewListGroup and ListGroup are combined as follows. In CombineListGroup, if both NewListGroup and ListGroup have ending variables (such as v2 in NewListGroup and v1 in ListGroup) and they are both in P, then the ending variables are connected by an edge. An ending variable is a variable in a listgroup that has only one neighbor and not in a group. Thus, by connecting v1 and v2, we get the final ListGroup as shown in Figure 69 (b).

It works similarly in dealing with VG2. Note that if a variable group has only one variable, it becomes a variable automatically. For example, in Figure 69 (d), after taking v6 from VG2, new VG2 only has v7, so new VG2 degenerates to a variable, thus there is no group circle around v7. Now the layout of v1, v2, v5, v6 and v7 are quite clear. If we keep the variables layout as shown in ListGroup in Figure 69 (d), all variables in VG1/VG2 can be protected together at the clustering point P1/P2. Only one protection operation is needed for v1, v2, v5 and v6 at P4. On the other hand, assume the data layout is like v3–v1–v4–v2–v5–v7–v6, any two of v1, v2, v5 and v6 cannot be protected together at P4 since the grouped protection operations only works for adjacent memory objects as discussed earlier.

It is the same for VG3. But in processing v8, ListGroup and NewListGroup are not connected though they both have ending variables (v7 and v8) since v7 and v8 are not both in VG4. That means VG4 does not suggest that v7 and v8 should be protected together. We choose to leave two listgroups instead of combing them into one because more listgroups have more ending variables that could provide more opportunities for further listgroup combination. Now how to proceed further is very clear, so we do not list all the intermediate data structures in Figure 69 (e) and (f). The final ListGroup is shown

in Figure 69 (f). There is still one variable group in the final ListGroup. Either v3 or v4 can be put next to v2. So the final data layout is v7–v6–v5–v1–v2–v3–v4, v8–v9–v10 or v7–v6–v5–v1–v2–v4–v3, v8–v9–v10.

Besides scalar variables, we can also group array accesses. For example, if an array is written/read in a stride of 2, e.g., it accesses elements 1, 3, 5, 7, 9, 11…, we may group array elements 1, 3, 5 as a group and array elements 7,9,11 as a group. Array access analysis is an important topic in compiler optimizations especially for loop parallelization. It has been extensively studied in [13][24][88][106][85]. With array access information, array accesses can be grouped using standard techniques. For array accesses with a stride of 1, the problem is very similar to loop vectorization. For array accesses with a stride greater than 1, loop scatter-gather can be done first to create a loop accessing the same data with a stride of 1. Of course any transformation cannot violate the original program dependencies. All the techniques involved are elaborated in 1[53] . Being able to handle arrays is important to reduce security cost for the whole program since a large percentage of dynamic memory accesses go to aggregated data structures, most of them being arrays.

6.2.5 Points-to Table

Dealing with pointer dereferences is a difficult problem. Static compiler pointer analysis has a lot of limitations and in some cases cannot determine an accurate points-to set for a pointer dereference. If the points-to set of a pointer dereferencing store instruction contains multiple possible memory objects, then before the pointer dereferencing store instruction, the access permission levels of all possible pointed-to memory objects have to be set to writable. Also, after the pointer dereferencing store

instruction, the access permission levels of all possible pointed-to memory objects have to be set to read-only. This could increase the protection overhead significantly. Moreover, in some cases, the points-to set can be very big, even including all possible memory objects, which means the compiler cannot derive any useful points-to information for this dereference. If we choose not to handle such cases; then we have to give up protecting the dereferencing store instructions and a large number of locations would remain unprotected which sacrifices security.

In our scheme, we propose a profile-driven solution to the above problem. Our observation is that due to the limitation of static analysis, although a pointer dereferencing store instruction could possibly access a large set of memory objects according to static compiler analysis, at runtime the number of memory objects actually accessed by a pointer dereferencing store instruction is very limited. The same observation is made in [131]. Thus, we use profiling to identify the most likely accessed memory object by a pointer dereferencing store instruction. Based on this information, a *points-to table* is created. The points-to table is basically a hash table indexed by the PC addresses of pointer dereferencing store instructions for which the compiler computed points-to set is large. Normally there are only tens of such entries in the hash table, so we can easily create a collision free hash table by tweaking the hashing function parameters to avoid the overhead to maintain a spilled list.

When handling these pointer dereferencing store instructions, all the compiler algorithms proceed as if the compiler knows the store instruction will access that most likely accessed memory object. Thus, the most likely accessed memory object will be properly protected. At runtime, the hardware component checks whether the pointer

actually points to the expected memory object by looking up in the points-to table. If the pointer happens to point to a different memory object, the hardware component will avoid checking access permissions for this pointer dereference to avoid potential false alarms. In this way, we are able to handle most dynamic instances of pointer dereferencing store instructions at the same time maintain a zero false positive rate.

From our results, the average size of the points-to set for each pointer dereferencing store instruction is 1.58. The average miss prediction rate of the dynamically accessed memory object for those specially handled pointer dereferencing instructions is only 5%. This shows our profiling based scheme is highly effective and does not degrade protection significantly.

6.2.6 Action Table and Special Instruction

We now describe in detail how the compiler actually inserts protection operations to be executed at runtime by the hardware component. The central data structure involved is an action table. The action table records which action should be performed at a given program point on a given memory object at runtime. An example of an action table is shown as Table 7. The action table is a hash table that uses the PC address of an instruction as its key. We always create a hash table without collisions to avoid the overhead of maintaining a spill list. Each entry of the table has three fields – action, memory object starting address, and memory object size. The action field has only one bit – bit value 1 denotes the action of changing the corresponding access permission level for the memory object from read-only to writable; bit value 0 denotes the action of changing the access permission level for the memory object from writable to read-only. The memory object is identified by its starting address and its size.

A special instruction is inserted at every protection point to inform the processor that some protection operation needs to be done at the given program point. Whenever the processor encounters such a special instruction, it uses its PC address to index into the action table and to execute the desired action.

Table 7. Action table.

Action	Starting address	Size
0	Addr1	4
1	Addr2	8
0	Addr3	30
	…	

Another possibility is to insert instructions to implement the intended protection operations directly instead of recording them in a separate action table. However, the action would require several instructions to implement, which indicates more changes to the standard ISA of the processor and more code space overhead. Using a separate action table plus one special instruction appears to be a better solution to us.

The action table requires the start address and the size of the modified memory object for each action. The important problem is that the information may not be available during compilation time. In general, during compilation, we do not know the address of the local stack objects and the heap objects. For some heap objects, we may not even know their sizes. To solve this problem, the action table has to be made writable and the compiler has to insert instructions to fix the action table for stack and heap objects. Such modifications of the action table occur after each dynamic memory allocation and each function entry point. The inserted code will update the action table with dynamic obtained address and size information when necessary.

Making the action table writable brings some security complications. Ideally, the action table should be put into a reserved address space and made read-only to the user program, so that it can only be written by the hardware component of our protection scheme to avoid malicious corruptions from the user program. Now how to protect the action table becomes an issue. Our solution is to apply the protection mechanism in a hierarchical way. That is, we use the same method to protect the action table residing in the memory. But in this case, the scale of the problem is much smaller and the problem is much simpler. We know the action table should only be modified by those fix-up instructions inserted by the compiler. We can regard the action table as one single memory object and its address is predetermined. Thus, the additional action table for protecting the original action table can be easily constructed. It can be put into a reserved address space, thus is not accessible to the user program but only accessible to the hardware component. So it cannot be corrupted by the attacker. Then the original action table can be protected properly.

6.3 Architectural Support

Figure 70 illustrates the necessary architecture support for our work. There are three major data structures. Action table and points-to table are simple small hash tables as explained previously. They are easy to manage due to their sizes and accesses to them are cached as data in data caches. The access bit table records the access permission level for each memory object. It is allocated into a reserved space and can only be accessed by the hardware component of our protection scheme, thus is protected from tampering by the user program. Access bit table is large and accessed frequently. It has to be carefully managed to avoid significant space and performance overhead. A very similar problem

exists in the Mondrian Memory Protection system work [120] and it provides an excellent reference on how to manage this large access permission table. We largely follow their design, regarding each memory object as a memory segment in their work. We deploy multi-level permission table with mini-SST entries that are elaborated in [120]. To improve performance, a protection lookaside buffer and sidecar registers are also deployed. Utilizing the design in [120] greatly reduces the space and performance overhead of our access bit table.

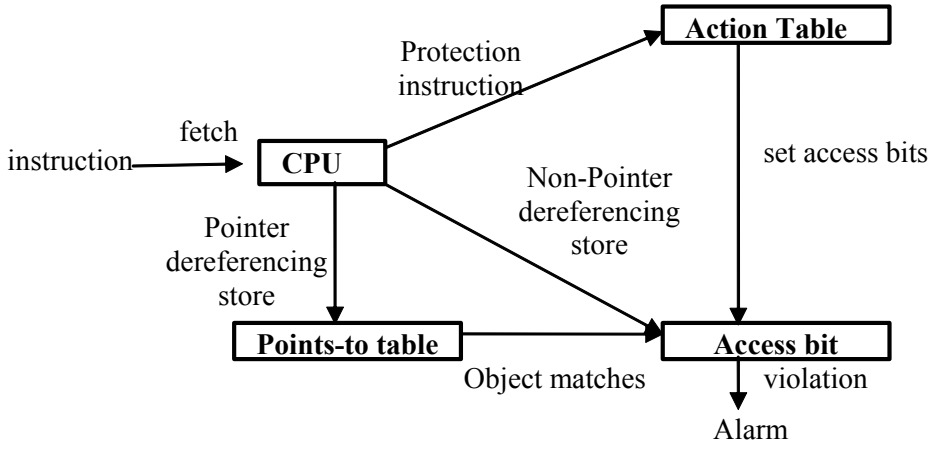


Figure 70. Architecture support overview

There are two kinds of operations performed by the hardware component. Upon fetching a protection instruction, the processor uses the PC address of the protection instruction as the key to index the corresponding entry in the action table. Then the access bits for the memory object will be set according to the starting address and the size information. Another case is when a store instruction is fetched. The processor first checks whether this store instruction is one of the specially handled pointer dereferencing instruction by looking up in the points-to table. If the instruction is in the points-to table, then it is specially handled. If the memory object dynamically pointed by the pointer does not match with the object recorded in the points-to table, then the processor will skip the

access permission checking for this store instruction to avoid any false alarm. Otherwise, the access bit table is looked up to find out whether the memory location is allowed to be written. If the instruction violates the access permission, then an alarm is raised indicating the program is under attack.

6.4 Evaluation

All compiler implementations are done in the Machine SUIF compiler infrastructure [68], a research compiler infrastructure for profile-driven and machine-specific optimizations. Machine SUIF can be used to instrument programs for profiling and carry out various code transformations and optimizations.

Our experiments are based on x86 Pentium architecture. In our experiments, we first train our benchmarks to gather the basic block profile which is fed back to compiler passes for memory protection. After the compiler passes are performed, the generated binaries are evaluated by input data sets different from the ones used in training runs as the test run. The benchmarks used are the same daemon programs used in the evaluation of the anomalous path checking scheme and the infeasible path detection scheme.

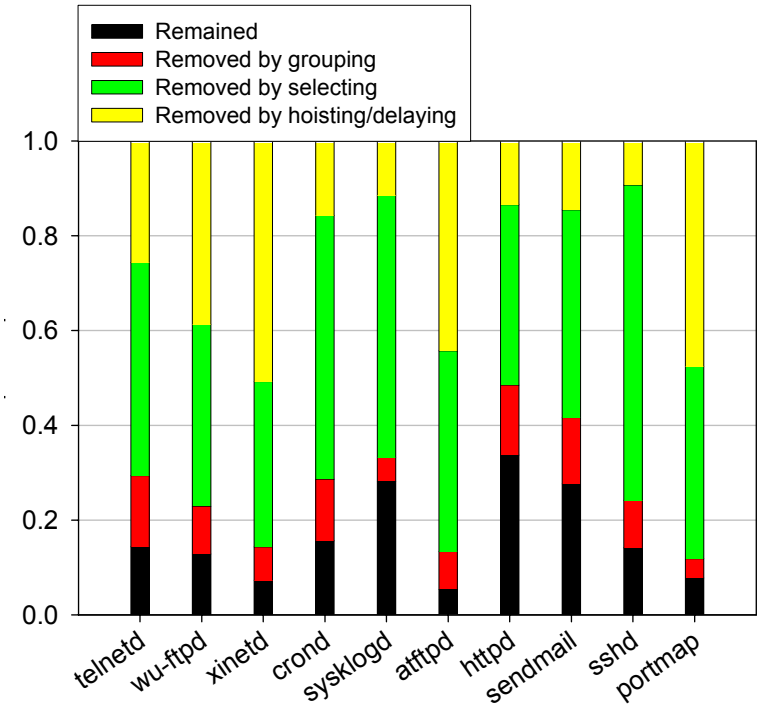


Figure 71. Effects of compiler optimizations.

Figure 71 shows the effects of our compiler optimizations. Three compiler optimizations, including protection point hoisting/delaying, protection point selecting and protection operations grouping, are proposed to reduce the number of dynamically executed protection points. On average, protection point hoisting/delaying reduces the number of dynamic protection points by 27.1%; protection point selecting reduces the number of dynamic protection points by 46.1%; protection operations grouping reduces the number of protection points by 10.1%. Overall, those optimizations reduce the number of dynamic protection points by 83.3% on average. The reduction results in significant savings in performance overhead.

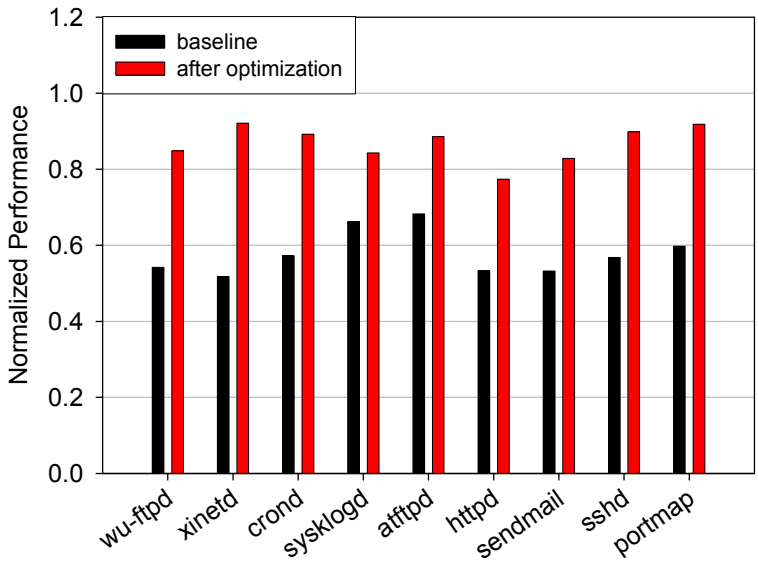


Figure 72. Performance degradation.

Figure 72 shows the performance degradation under our scheme. All hardware support modeling is done inside SimpleScalar [11] targeted to x86 [114]. The parameters for the processor modeled are shown in Table 8. The performance under the baseline protection scheme and the performance after our optimizations are shown in the figure. Performance numbers are normalized to the original program binary without protection. From the results, the average performance degradation under the baseline protection scheme is 41.4%. The performance degradation mainly comes from the overhead to access the access bit table. Other sources of performance degradation include accessing the action table and the points-to table, and executing the compiler inserted instructions to fix up the action table. The average performance degradation after optimizations is 13.0%. So our optimizations are able to reduce the performance degradation due to data protection significantly because optimizations are designed to reduce the number of dynamic protection points greatly.

Table 8. Parameters of processor simulated.

Clock frequency	1 GHz	Branch predictor	2 Level
Fetch queue	16 entries	BTB	512 entries, 4 -way
Decode width	4	PLB	128 entries
Issue width	4	L1 I/D	DM, 32K, 1 cycle 32B block
Commit width	4	Unified L2	8way, 32B block 1M (16 cycles)
RUU size	64	Memory bus	200M, 8 Byte wide
LSQ size	32	Memory latency	first chunk: 120 cycles, inter chunk: 10 cycles

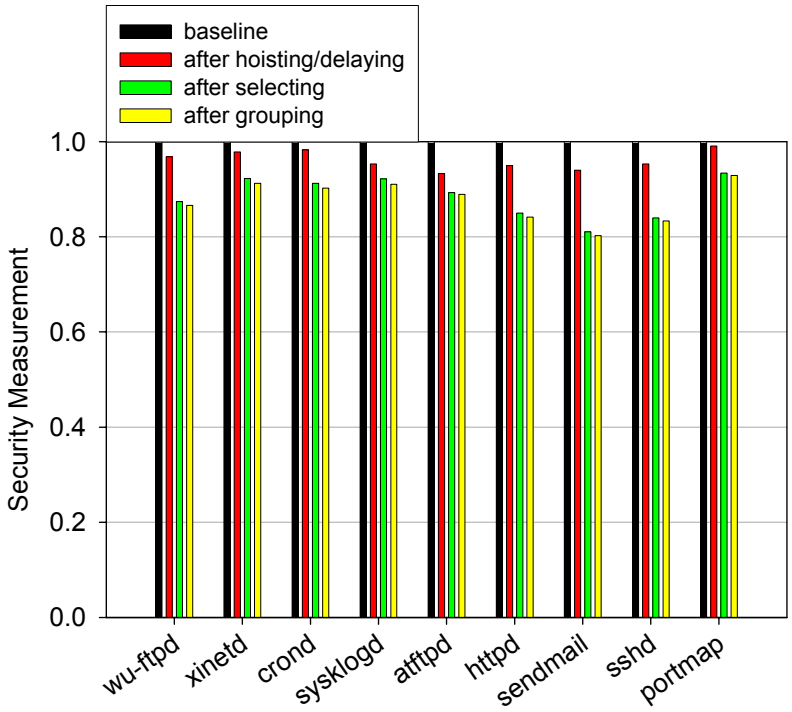


Figure 73. Security measurement.

Figure 73 shows the security measurement. Under our baseline protection scheme, the access permission level for a memory object is set to writable just before a store instruction writing to it and set to read-only right after the store. Thus the baseline protection scheme can detect all memory corruption attacks exploiting the flaws in the program such as buffer overflows and format string vulnerabilities, since an attack needs a store instruction to do the tampering and that attacking store instruction has to be

between the pair of “set to writable” and “set to read-only” operations. The baseline scheme corresponds to the 100% protection shown in Figure 73. Due to the motion and removal of protection points caused by compiler optimizations, some memory locations could be left unprotected. The reason is that another store instruction str2 may be between the “set to writable” operation and the “set to read-only” operation for str1 now, and str2 could be the attacking store instruction and could possibly tamper the memory object to be written by str1. Also, if a pointer points to an unexpected memory object at runtime, the write will not be checked against the access permission table, thus there could be false negatives too. We count the number of dynamic store instructions like str2 and the number of dynamic pointer dereferencing store instructions with an unexpected target memory object, then subtracted those from the total dynamic store instructions to measure the protection strength after optimizations. As per this calculation, a higher number of dynamic store instructions after subtraction means a higher protection strength.

The results in Figure 73 indicate that the compiler optimizations do not lead to much security degradation. On average, protection points hoisting/delaying degrades the protection strength by 3.6%; protection points selection degrades the protection strength by 7.8%; protection points grouping degrades the protection strength by 0.7%. Overall, after optimizations we achieve an average of 87.8% protection over the baseline scheme, ranging from 80.2% to 92.9%. So our scheme achieves a good memory protection with reasonable performance degradation. It should also be noted that the above method to measure protection strength represents the worst case. The above method assumes that every store instruction moved into a pair of “set to writable” and “set to read only” operations could be an attacking store instruction or an attacking point. In reality,

normally a program only has a few possible attacking instructions. For example, for a buffer overflow attack exploiting the strcpy function, the store instruction in the strcpy function implementation is the possible attacking instruction. As long as our optimizations do not move that possible attacking instruction into a pair of "set to writable" and "set to read only" operations (which is very unlikely), the security will not be harmed at all. Thus, our scheme should have much stronger detection strength against real-world attacks than represented by the above worst case.

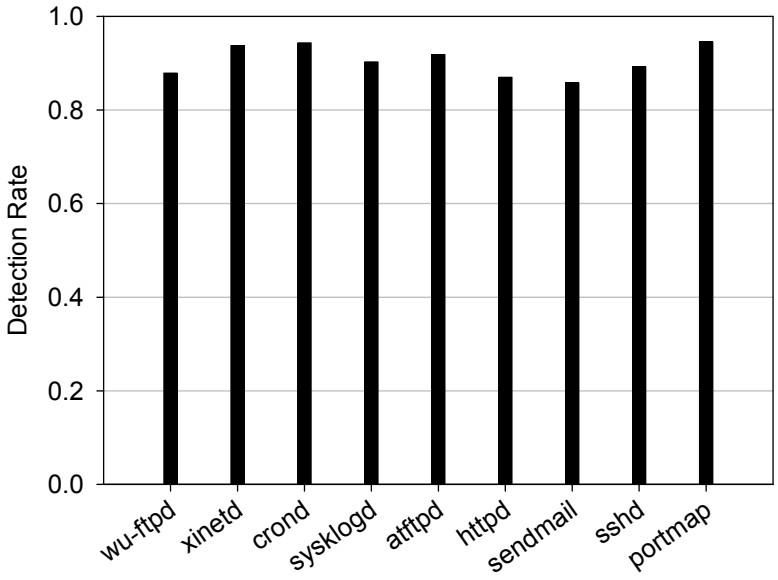


Figure 74. Detected simulated attacks (extreme condition).

We also performed simulated attack experiments to further evaluate our data tampering detection scheme. We run the benchmarks in the simulator and then randomly tamper a memory location by making a dynamic store instruction write to a different memory object to see whether our scheme is able to detect the tampering. For each benchmark, we perform such simulated attacks by tampering at 1000 different memory locations.

Although such a fault injected is not a real attack, we believe that our fault injection model provides an extensive coverage of real world attacks and should represent the worst case. Our fault injection model makes no assumptions about the attack behavior and assumes that any memory object can be tampered by any dynamic store instruction. The ability of real-world attacks is much more limited than this. For example, the classical stack buffer overflow attacks cannot tamper arbitrary memory locations and can only tamper memory locations following the unchecked and overflowed buffer. Moreover, such stack buffer overflows normally tamper a continuous region of memory instead of a specific memory location, making the detection of those attacks much easier. Some attacks, such as format string attacks and heap overflow attacks, actually enable the attacker to tamper arbitrary memory locations. However, the tampering is done by a specific attacking store instruction. Only when that instruction is executed, could possibly memory tampering be done. Which store instructions could be attacking instructions are attack dependent. So to do a limit study of our scheme, in this experiment, we again assume that every store instruction could be an attacking instruction and our results represent the worst case. Figure 74 shows the percentage of attacks detected. On average 92.7% of the randomly injected memory tamperings are detected, which shows that our scheme is very effective to protect memory tampering attacks even under the worst case.

We also tested our scheme against several real attacks discussed in [17], including format string attacks against user identification data, heap corruption attacks against configuration data, stack buffer overflow attacks against user input data and integer overflow attacks against decision-making data. In all of those attacks, there are some attacking store instructions used to corrupt memory state and they achieve the memory

corruption by writing to a memory object that should not be modified by those instructions at all. In the baseline scheme, such memory corruptions by illegally tampering a memory object are easily detected since a memory object is only marked as writable during the span of the execution of a legal store instruction to the memory object. After our optimizations, as previously discussed, as long as the attacking store instruction of an attack does not result in a pair of "set to writable" and "set to read-only" operations for the target memory object, the attack will detected. The miss detection is a rare event since normally there are only a few possible attacking store instructions inside a program and for the attack to succeed, some specific memory object has to be tampered rather than any memory object. We found that even after optimizations, our scheme can detect all of these exemplary attacks shown in [17].

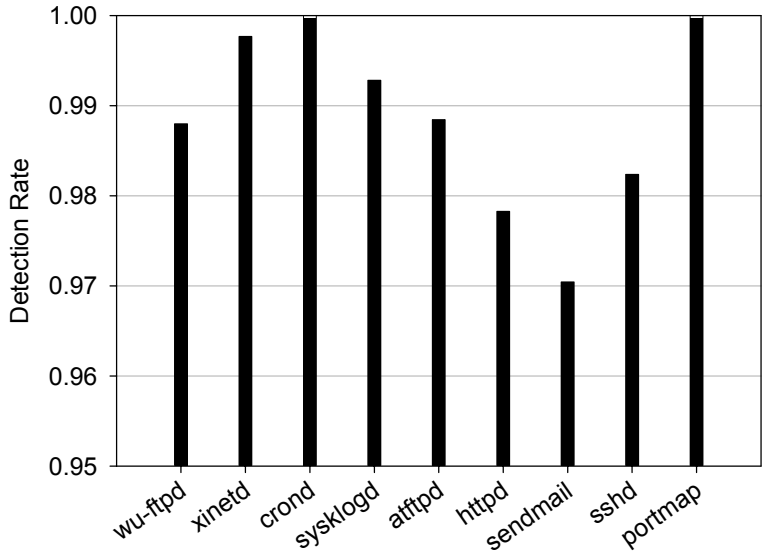


Figure 75. Detected simulated attacks (more realistic condition).

To further get a sense of how our data tampering detection scheme performs under a more realistic condition rather than under the extreme condition assuming that every store instruction could be an attacking instruction, we tried to limit the possible

attacking store instructions. Instead of considering every store instruction as a possible attacking store instruction, we first tried to regard only store instructions depending on external inputs as possible attacking instructions. Those store instructions can be affected by the attacker and are thus dangerous. However, we found that actually most store instructions depend on external inputs directly or indirectly so this method does not really give different results. Then we tried to survey existing vulnerabilities and got a list of well-known dangerous standard library functions, such as strcpy, free, printf and a lot of others, and then got the attacking store instructions in those functions. We performed a similar simulated attack experiment but this time regarded only those store instructions as possible attacking store instructions. Since the number of possible attacking store instructions is greatly reduced, we got much better results as shown in Figure 75. On average, 98.9% of simulated attacks are detected. This experiment is more realistic than the limit study. However, it is optimistic in that it only considers existing popular vulnerabilities in standard libraries. On the other hand, it is also very pessimistic since it assumes that tampering any memory object would be useful to the attacker. In reality, some very specific memory object has to be tampered for the attacker to launch a successful attack. However, which memory objects are interesting is highly application dependent.

The space cost is shown in Table 9. There are four fields in this table. The first three fields represent the sizes of the action table, the access bit table, and the points-to table. The last field shows the total code size increase of the program due the insertion of protection operations. The additional action table to protect the original action table only requires a little space, so we do not show its size here. We use a multi-level permission

table as in [120]. The average sizes of the action table, the access bit table, the points-to table, and the code increase are 12361 bytes, 319 KB, 776 bytes and 7721 bytes respectively. The access permissions for one object are compacted using the Mini-SST [120] technique, which could save a lot of space. The results show that the space cost of our protection technique is at most several hundred kilo bytes, which is very acceptable for modern computers.

Table 9. Space cost measurement.

Benchmark	Action table (byte)	Access bit table (KB)	Points-to table (byte)	Code size increase (byte)
Telnetd	1580	193	388	1216
wu-ftpd	5972	376	700	3980
xinetd	3084	151	652	2204
crond	588	263	304	492
sysklogd	1728	124	312	1232
atftpd	508	231	336	364
httpd	41384	847	1920	25864
sendmail	57784	524	1268	33992
sshd	10804	360	1624	7716
portmap	176	122	260	148

6.5 Summary

A large category of realistic memory tampering attacks do not alter program control flows at all, which makes control flow monitoring based anomaly detection schemes completely helpless when combating with those attacks. In this chapter, we propose a compiler and micro-architecture collaboration framework to perform memory protection and to detect memory tamperings directly instead of trying to infer memory tamperings from anomalous control flows. Three optimizations involving hoisting, selecting and grouping of memory protection operations are designed to reduce the performance degradation of the baseline scheme. By carefully crafting these

optimizations, our empirical study shows that the security of the scheme is not reduced much whereas the performance degradation is reduced significantly. Our experiments prove that our scheme achieves strong memory protection with tight control over the performance degradation.

Comparing with the anomalous path checking scheme and the infeasible path detection scheme, the data tampering detection scheme presented in this chapter is able to detect attacks not altering program control flows, while those two schemes based on control flow monitoring cannot. The data tampering detection scheme also achieves zero false positive rate since it is based on static compiler data flow analysis. Thus, the security strength of the data tampering detection scheme is superior to the previous two schemes we discussed. However, the hardware support required for the data tampering detection scheme is more complicated thus introducing more hardware cost. More importantly, the performance degradation (13% on average) is much more significant than the previous two schemes. Thus, the data tampering detection scheme should be applied in situations in which the maximum security strength is the first priority or the performance degradation incurred is affordable. Finally, in the next chapter we will show that the data tampering detection scheme has significant advantage in terms of intrusion recovery. The reason is that both the anomalous path checking scheme and the infeasible path detection scheme try to infer data tampering from anomalous control flows, thus it is impossible for them to detect the data tampering until the tampered data is used by the program. In other words, there could be an arbitrarily long interval between memory tampering and attack detection, making intrusion recovery a very difficult problem. Such a problem does not exist in the data tampering detection scheme. Detailed discussions

regarding this issue can be found in the next chapter. Thus, the data tampering detection scheme should be given a higher priority if intrusion recovery is highly desired.

So far we have proposed three anomaly detection schemes. Each of them has its advantages and disadvantages. It is necessary to give some guidance on how user should decide to choose which anomaly detection scheme. In general, due to its security advantage, the data tampering detection scheme should be given the highest priority as long as the performance overhead incurred is acceptable. If the performance overhead is not affordable, the anomalous path checking scheme and the infeasible path detection scheme should be then considered since they incur only minor performance degradation. Then the user needs to evaluate the cost of deploying the anomalous path checking system. The system has to be properly trained, bringing significant initial installation cost. Also there will be false positives and managing a possibly large number of false positives is also an important concern. The installation and maintenance cost generally depends on the complexity of the protected software. If the cost is acceptable, then the anomalous path checking scheme should be chosen due to its better detection strength. Otherwise, the infeasible path checking scheme should be chosen since it does not have false positives at all. As a summary, the pseudo code to decide an anomaly detection scheme is shown in Figure 76.

```
IF estimated performance overhead (through simulation) of the data
    tampering detection scheme is affordable THEN
        Choose data tampering detection scheme;
ELSE IF the initial installation cost and the maintenance cost of the
        anomalous path checking scheme is affordable THEN
        Choose anomalous path checking scheme;
ELSE
        Choose infeasible path detection scheme;
```

Figure 76. How to decide an anomaly detection scheme.

7 INTRUSION ANALYSIS AND INTRUSION RECOVERY

After spending significant efforts on devising intrusion detection schemes, we change our focus into another important aspect of software protection – intrusion recovery – in this chapter. No security system is bullet-proof and is able to prevent all attacks. To be realistic, we have to assume that some attacks will be able to evade the intrusion prevention mechanisms implemented. To protect software from those attacks, we build a second line of defense consisted of intrusion detection, intrusion analysis and intrusion recovery mechanisms. We have discussed several efficient intrusion detection schemes in previous chapters. In this chapter, we focus on intrusion analysis and intrusion recovery.

After an intrusion is detected, most previous approaches simply shut down the attacked process to avoid any further damage. However, security implies several important properties including confidentiality, integrity and availability [3]. Availability is an equally important security property to be enforced but gets far less attention. In the sense of software security, availability means recoverability from attacks or intrusion recovery. Simply shutting down the attacked process is unacceptable for software providing critical services. A complete software protection scheme should have the ability to analyze an attack and recover from the attack whenever possible. Thus, intrusion analysis and intrusion recovery is an important goal of our secure infrastructure.

7.1 Background and Motivation

Intrusion recovery is critical since no software system can be made perfect. There are always software bugs, configurations errors etc. to be exploited by the attacker. Upon detection of an attack, the simplest response is fail-stop, i.e., shutting down the system to

avoid further damage. However, shutting down the system completely leads to inconvenience for the users and denial of services. For systems providing critical services, it may cause huge financial losses. In areas such as aircraft control, battlefield control, it could be disastrous.

Intrusion recovery enables software to recover from attacks and continue correct software execution even when tampering occurs. Intrusion analysis is an indispensable step for intrusion recovery. We can recover from an attack only after we understand the attack. Unfortunately, although failure recovery is often the most important aspect of security engineering, it has not been carefully addressed. Most of the computer security research has dealt with confidentiality and most of the rest with authenticity and integrity. However, the actual expenditure of systems providing critical services may go the other way around [3].

Availability is a traditional research topic in fault tolerant computing. Traditional solutions to provide availability in fault tolerance area are replication [81][90][66][69][14] and checking-pointing/roll back [57][116][29][86][99]. Those solutions cannot be simply copied to security area. Replication is very effective to combat random faults since the chances that multiple replicated systems experience random faults at the same time are extremely low. However, it is very likely that multiple replicated systems are attacked coordinately at the same time since attacks can be well organized. Software versioning [7] could be used to mitigate the problem. Overall, replication is a very expensive way to provide availability since the system cost with n-replication is order of n times larger. Checkpointing backs up the program state for later recovery. The size of the program state could be large. If done in software, checkpointing could incur significant overhead.

For example, in [116], the checkpointing interval is 30 minutes and each checkpointing costs up to several minutes with program state size up to tens of megabytes. The resulting overhead could prevent fine-grained checkpointing. To reduce performance overhead, checkpointing can be done incrementally through logging of changes of in program state.

Although fault tolerance is a traditional research area, there has been limited research on intrusion recovery for secure software. DIRA [98] is a GCC compiler extension to combat buffer overflow attacks. Instead of merely detecting buffer overflow attacks, DIRA unifies the functionality of attack detection, attack identification (intrusion analysis) and attack repair. DIRA achieves attack identification and repair by a memory updates logging based scheme. DIRA is among the first to make an effort to analyze and recover from attacks, but it has many limitations. It only handles a limit set of buffer overflow attacks targeting to tamper return addresses and function pointers. As we have shown previously, those attacks are only the most basic ones and real attacks can be much more sophisticated. The memory updates logging algorithm of DIRA is designed for simplicity with little compiler analysis and frequently sacrifices security strength. It only logs updates to global variables and ignores local variables. It is not clear that whether the scheme is able to handle heap objects. Moreover, the current DIRA compiler only tracks data dependencies carried by simple assignment operations such as A=B and proxied functions, it cannot identify dependencies that involve any arithmetic expressions, e.g., B=A+C. This means that DIRA's recovery scheme will not be able to trace a corrupted data structure back to the attack point as long as the data structure is computed through some form of transformation other than assignment operations. Even if DIRA is enhanced with powerful compiler analysis, the limitation of static compiler analysis is

well understood, especially when dealing with aliased data. The performance degradation of the scheme is up to 60% with an average of 25% for the benchmarks tested even it only deals with return addresses and function pointers in a limited way. As shown in [17], a lot of other software data is critical to security. If all security critical data is handled in this scheme, the performance overhead will be unacceptable.

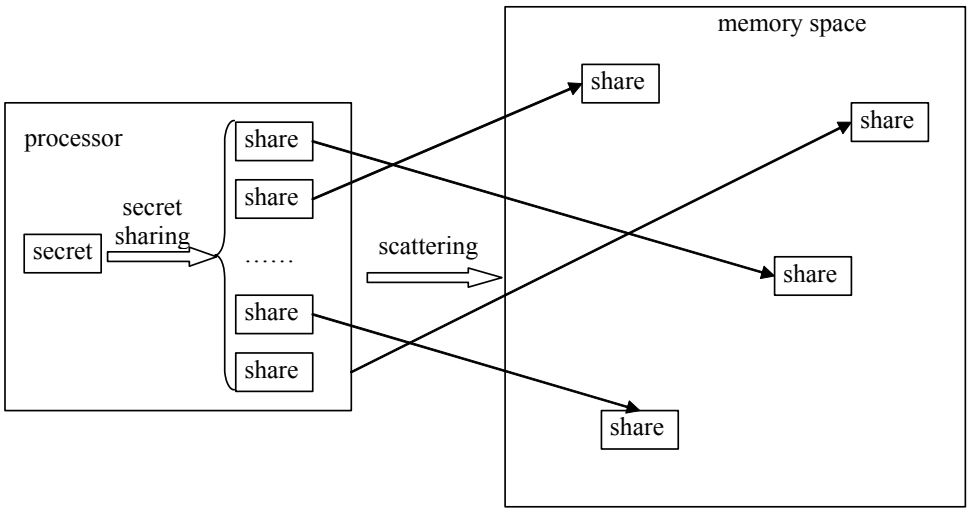


Figure 77. Data replication/scattering based on secret sharing.

We are also among the first to introduce intrusion recoverability into secure software. Our previous work [127] proposed applying secret sharing [93] in the process memory space to achieve confidentiality, integrity and availability at the same time for critical data. It can be regarded as one example of data replication/scattering technique. Data replication/scattering creates and distributes data replicas in the software process memory space. The goal is to achieve recoverability with redundancy. To recover from an attack, all replicas are gathered and the value given by the majority is regarded as the original value. The basic idea of our secret sharing scheme is illustrated in Figure 77. The data to be protected is first secret shared and multiple shares for the data are obtained. Then the shares are distributed to random locations in the memory space only known to

the process to which the data belongs. If the number of shares tampered is below a threshold and other shares are intact, the scheme is able to recover the tampered data.

During our secret sharing work, we were able to understand the data replication/scattering technique better. We realized that the technique has several important limitations. First, the recoverability is not as good as checkpointing and memory updates logging, since the replicas reside in a limited buffer space. The attacker can wipe out all replicas in the extreme case and the data replication/scattering technique has no way to recover from that. Second, the recoverability strength largely depends on the size of the buffer space in which the replicas are randomly distributed. A stronger recoverability requires a larger memory space to defend brute-force guessing, which could introduce significant memory cost. Finally, the technique can recover from hardware attacks and attacks from other malicious processes, but it has difficulty to recover from attacks exploiting software flaws like buffer overflows. If the attacker can tamper some software data by impersonating the software, all the replicas for the data would be automatically tampered.

For data replication/scattering technique to defend against attacks exploiting software flaws, the data protected by replication/scattering has to be treated differently from the data without protection. One possibility is to introduce a new pair of scattering load/store instructions. The new scattering load/store instructions can be implemented either by software or by hardware with instruction set architecture (ISA) support. The user first decides what to be protected then the compiler generates proper instructions. Protected data is accessed with scattering loads/stores and unprotected data is accessed with normal loads/stores. Then, we can sandwich protected data with unprotected data to

defend against attacks exploiting software flaws. How this could help is illustrated in Figure 78.

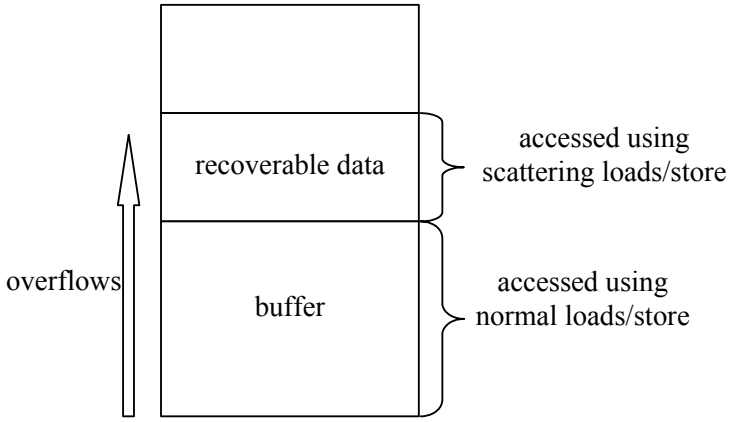


Figure 78. How data replications/scattering defends against buffer overflows.

As shown in the example, by providing an oversized input, the attacker can still overflow the buffer. However, the buffer is written using normal stores. Thus, by overflowing, the attacker tries to tamper the protected data using normal stores, which is futile since to access the protected data correctly, scattering loads/stores have to be used. This above idea helps data replication/scattering technique combat buffer overflow type of attacks. However, the above tweak is not a clean solution and in general data replication/scattering is vulnerable to attacks exploiting software flaws.

Although the protection strength of a data replication/scattering scheme is not satisfying, it can recover tampered data in real-time. The latency of recovering data by program state checkpointing or memory updates logging could be in terms of milliseconds or even seconds, but data replication/scattering technique can recover data in tens of or hundreds of cycles as shown in our previous work. Real-time intrusion recovery is an important property which may be critical to real-time applications. Thus, data replication/scattering scheme could be an option desired in certain circumstances.

But in this dissertation work, we focus on developing a more general intrusion analysis and intrusion recovery scheme with strong recovery strength thus will not explore more into data replication/scattering schemes.

Recently, Shi et al. proposed a system design using a chip multi-processor to provide intrusion tolerance and self-recovery for server applications [97]. They use a checkpointing based approach to recover server applications under attack. Their scheme takes a snapshot of the application context and memory state before the server application handles the next request. If the request turns out to be malicious, the system can discard the malicious request and rollback the application's state to a known good one through checkpointing. Their scheme essentially is a checkpointing based approach with architecture support to achieve low performance overhead.

However, their strategy of checkpointing is overly simplistic. In their scheme, the server takes a snapshot of its process context and incremental memory state upon receipt of a new network request. The request is then handled by the application. If later, an intrusion is detected by the intrusion detection software, the application's memory state is rolled back to a state before the malicious request was handled. The problem is that there could be an arbitrarily long interval between the damage to the process state and the detection of the attack. For example, an attacking packet exploiting a buffer overflow vulnerability can tamper a memory location, but under current intrusion detection schemes the tampering most likely will not be detected until the tampered memory location is used by the software. After all, if such a tampering can be detected immediately, attack recovery would be a much simpler problem. Thus, it is not clear when it is safe to conclude that a packet is not malicious. Before the detection of the

attack caused by a malicious packet, hundreds of innocent packets may have arrived and it is likely that the proposed scheme will identify an innocent packet as the malicious one and roll back the process state to a corrupted one. Moreover, it is not clear how to define a request to the server. Assuming one packet is a request is overly simplistic. An attack can be easily launched by multiple network packets.

In this dissertation work, we aim to build an intrusion analysis and intrusion recovery mechanism with strong recovery strength and small performance degradation by utilizing micro-architecture level support. In our work, we assume that the attacker exploits the software flaws through its interfaces to the outside world, for example, program arguments, program input files, keyboard inputs, incoming packets, external events etc. In other words, if the software has absolutely no interactions with the outside world, it is impossible to attack the software by exploiting its flaws. We believe this assumption is very realistic. If the software tampers itself even without any interaction with the outside world, it probably should be regarded as a software bug rather than an attack.

We believe that to recover from an attack after its detection, two major tasks have to be done. First, we need to analyze the attack to identify exactly when and how the attack happens so that we can identify the tampered system state and gather useful information for later forensic analysis. We call this step intrusion analysis. Proper intrusion analysis is the basis for later recovery and cannot be arbitrarily done as in [97]. Otherwise, we could roll back the program into a tampered state. Second, the tampered system state has to be recovered through certain mechanisms such as checkpointing. We focus on the attacks tampering memory state and recovery of memory state in our work.

Memory state tampering is the most common starting point of an attack and we can limit our problem scope greatly by focusing on it.

It is very easy to identify memory state tampered by hardware attacks and attacks from other malicious processes in the presented hardware infrastructure. In the hardware infrastructure, data residing in the untrusted external memory is protected cryptographically with a process specific key and is always verified before used by the secure processor. If the data is tampered by hardware attacks or other malicious processes, it will be detected by the hardware integrity checking scheme immediately. Also, the hardware integrity checking scheme knows exactly which memory block is tampered. Thus, the possible tampering by hardware attacks or other malicious processes is very limited.

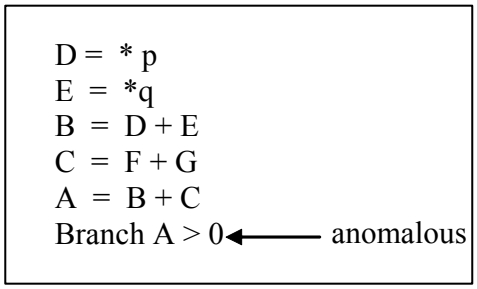


Figure 79. Difficulty of identifying tampered data.

However, the hardware integrity checking scheme does not help against attacks exploiting flaws in the protected software itself, such as buffer overflows. Our anomaly detection schemes can monitor the software execution and detect attacks exploiting software flaws by detecting anomalous program paths, infeasible program paths or data access anomalies. Once an anomaly is detected, which individual instruction caused the anomaly is identified. The anomaly detection scheme can provide such information. Then we need to identify which data is possibly tampered and which external inputs cause the

anomalous instruction. That is a difficult problem as illustrated in Figure 79. The figure shows a piece of pseudo code, in which branch A>0 has been identified as a jump instruction causing an anomaly. The branch instruction directly depends on the value of A. But can we simply assume that A is tampered by the attacker? Unfortunately we cannot. From the pseudo code, the value of A depends on the values of B and C. The value of B depends on the values of D and E, and so on. For example, the anomalous branch can be caused by a tampering to D instead of A. Such a dependency chain can be arbitrarily long and tampering to any data on the dependency chain could be the cause of the anomalous path.

Before we elaborate the details of our intrusion analysis schemes, we would like to point out the advantage of our data tampering detection scheme in terms of intrusion recovery. In general, there are three phases from the initiation of an attack to the detection of the attack. During the first phase, the attacker injects certain malicious input into the program to initiate the attack. Such malicious input may or may not cause tampering to the memory state immediately. For example, the malicious input can be copied several times and only one of the temporary buffers does not have enough space to hold the input. In that case, tampering to memory state will not happen until the malicious input is copied into the vulnerable buffer. During the second phase, tampering to the memory state is actually done. During the third phase, the tampered memory state is used by the victim program then the attacker is able to take control of the program or cause other damages.

Since most intrusion detection schemes including our anomalous path detection and infeasible path detection are based on control flow monitoring, they are not able to

detect the attack until the tampered memory state is used by the program and leads to some anomalous program behavior. There could be an arbitrarily long interval between the tampering of the memory state and the use of the tampered memory state, which brings difficulty during intrusion analysis and intrusion recovery. On the other hand, our data tampering detection scheme detects memory tampering directly instead of trying to infer memory tampering from control flow tampering. It can detect the attack in the second phase of the attack. In many cases, the malicious input causes memory tampering immediately. In those cases, our data tampering detection scheme can detect the attack immediately thus has significant advantage in terms of intrusion analysis and intrusion recovery.

7.2 Intrusion Analysis Based on Logging

Intrusion analysis and tampered memory state identification can be done through dependency backtracking of the anomalous instruction. As shown in Figure 79, there are two major issues in such a scheme. First, we need a mechanism to enable dependency backtracking and identify which memory values the anomalous instruction depends on. Second, we need to be able to stop backtracking dependencies as early as possible but safely. Otherwise, it is very likely that all memory state has to be regarded tampered and to recover from the attack, the program has to restart. Compiler can help on dependency tracking, but it has important limitations. First, it can only track dependencies statically, thus it will not be as efficient and precise as dynamic tracking. Second and more important, compiler analysis has difficulty to handle pointer dereferences. An example is shown in Figure 79. The value of D depends on the value pointed by the pointer p. In many cases, compiler cannot figure out which data pointer p is actually pointing to. In

those cases, we have to regard the value of D possibly depending on any data, which means all data is suspicious and could have been tampered. Thus, we believe that a hardware based dynamic dependency backtracking mechanism is necessary since at runtime we could know the exact target of a pointer.

To enable dependency backtracking from any program point, we devise a logging based scheme. The logging scheme records two kinds of dynamic program information. The first is dynamic control flow trace. The second is memory reference trace. With these two kinds of information, we can know the complete dynamic program path followed by the execution thus we know how to backtrack the execution. At each memory reference point, we also know exactly which memory address is referenced.

To stop tracking dependency safely, we need to make certain assumptions on the attacks. As mentioned earlier, in our work, we assume that the attacker exploits the software flaws through its interfaces to the outside world, for example, program arguments, program input files, keyboard inputs, incoming packets, external events etc. Under this assumption, the dependency backtracking can stop when 1) the data value does not depend on the external inputs at all, for example, the data value is a constant; 2) the data value is directly defined by the external input. The second case happens when the program interacts with the outside world, for example, the values of the program arguments (argv in the C language). We know the data values are defined by some external input and that input is suspicious, but in our work we limit our backtracking inside the victim process and we will not backtrack the external input further.

We elaborate our logging scheme below. To record dynamic program paths, the hardware logs all dynamic jump instructions changing the program PC in a non-

sequential way. For each such dynamic jump instruction, we record the PC address of the jump instruction and its target address. Also, for each dynamic memory reference, we record the memory address referenced and the type of memory operations. Obviously, types of memory operations should include memory reads and memory writes. In our scheme, for the purpose of dependency backtracking, we also introduce a special type of memory operation – external input. External input is a special type of memory writes in which the memory location is indicated to be directly defined by some external input, through I/O operations, system calls etc. In our work, we assume that I/O port operations are done through a memory-mapped I/O scheme. External input logs facilitate the backtracking to stop when the memory location is defined by some external input directly, as discussed previously. External input logs are generated by special system support when the program receives outside inputs. Also note that for memory reference records, we do not need to log the PC address for the reference as long as we create a record for every dynamic memory reference. Since we have complete dynamic program path, the memory reference records can be easily mapped to the memory references of the program code. We will illustrate that by an example. The data structure of a log record under our logging scheme is shown in Figure 80.

```
enum  log_type_t { PATH, MEM};
enum  mem_type_t  {READ, WRITE, EXT_INPUT};
struct log_t {
      log_type_t log_type;
      union {
            struct  {
                  addr_t    pc;
                  addr_t    target;
            } path_info;
            struct  {
                  mem_type_t mem_type;
                  addr_t     address;
            } mem_info;
      } log_info;
};
```

Figure 80. Data structure of a log record.

Based on the information logged, we can derive the dependency backtracking algorithm. The algorithm is formalized in Figure 81. The algorithm assumes a RISC instruction set architecture.

```
Input: The anomalous instruction and the log of program execution
Output: A set of suspicious external data input points

set W = ∅              // working set of registers/memory addresses
set REG_DEF (inst) = defined register of the instruction
set REG_USE (inst) = used register of the instruction
set INST(pc) = instruction at instruction address pc
set cur_inst = INST(cur_pc)

Algorithm: Dependency backtracking
1.  cur_pc = anomalous_pc
2.  W = W ∪ USE( cur_inst )
3.  While (W<>∅) do
4.     switch cur_inst->type
5.     case LOAD/STORE:
6.        if (cur_log->log_type <> MEM)
7.           ERROR("no corresponding log record for the memory reference");
8.        endif
9.        if (cur_inst is LOAD)
10.          Assert(cur_log->mem_info.mem_type == READ)
11.          if (REG_DEF(cur_inst) ∈ W)
12.             W = W – REG_DEF(cur_inst); W = W ∪ cur_log->mem_info.address;
13.          endif
14.       else  // STORE
15.          Assert(cur_log->mem_info.mem_type == WRITE)
16.          if (cur_log->mem_info.address ∈ W)
17.             W = W – cur_log->mem_info.address; W = W ∪ REG_USE(cur_inst);
18.          endif
19.       endif
20.       cur_log = PREV(cur_log)
21.    case OTHER:
22.       if (REG_DEF(cur_inst) ∈ W)
23.          W = W – REG_DEF(cur_inst); W = W ∪ REG_USE(cur_inst);
24.       endif
25.    end switch
26.    if (cur_log->log_type == MEM) && (cur_log->mem_info.mem_type == EXT_INPUT)
27.       W = W – cur_log->mem_info.address
28.       add program point corresponding to cur_log into suspicious_set
29.       cur_log = PREV (cur_log);
30.    endif
31.    if (cur_log->log_type == PATH) && (cur_log->path_info.target == cur_pc)
32.       cur_pc = cur_log->path_info.pc
33.    else
34.       cur_pc = cur_pc – 4 ;
35.    endif
36. endw
37. return suspicious_set
```

Figure 81. Dependency backtracking algorithm.

Assume the size of one instruction is four bytes. The dependency backtracking algorithm takes the anomalous instruction as input and finds out a set of suspicious dynamic external input points, which possibly cause the anomalous instruction. The algorithm examines the program instructions in the reverse order of the dynamic control

flow. If the current instruction is a load instruction, the algorithm removes the defined register from the working set and puts the accessed memory address into the working set. The accessed memory is obtained from the memory reference logs. If the current instruction is a store instruction, the algorithm removes the accessed address from the working set and puts the register used to define the memory location into the working set. For other instructions, the defined registers are removed from the working set and the used ones are put into the working set. Note that other instructions mentioned above include jump instructions to incorporate control dependence. Also note that there are instructions for which the used register set is empty, such as load constant, load address etc. The working set could be reduced after processing of those instructions. The algorithm also checks the existence of path logs and external input memory logs. For external input memory logs, the algorithm removes the memory address directly defined by some external input from the working set and stops backtracking for that memory address. Then the dynamic external input point is added to the set of suspicious external input points. The algorithm also checks whether the target address of the current path log record is same as current PC. If it is same, it changes current PC to the PC of the jump instruction; else it just subtracts four (the size of one instruction) from the current PC since the control flow was sequential.

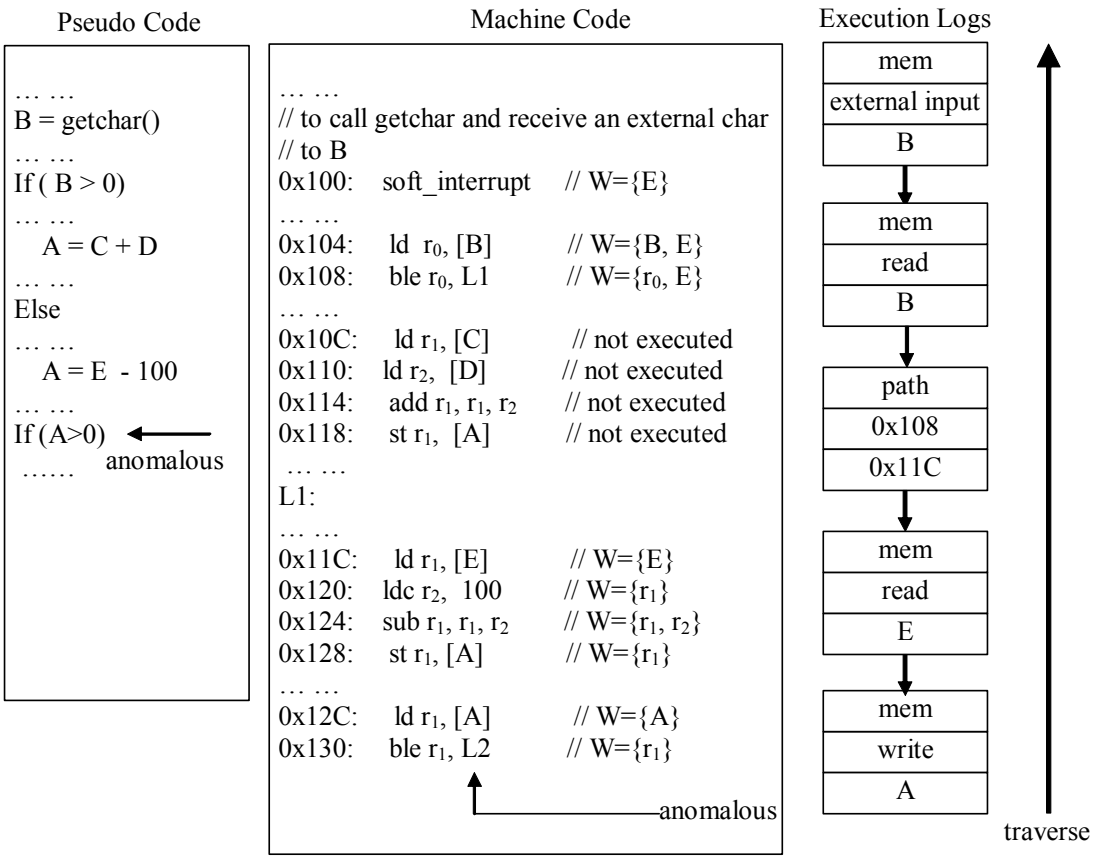


Figure 82. An example of dependency backtracking.

Figure 82 shows a simple example of dependency backtracking. In the figure, a piece of pseudo code, the corresponding simplified machine code and the program execution logs are shown. The branch instruction at address 0x130 causes an anomalous path. The dependency backtracking algorithm traverses the program backward from the anomaly detection point, and processes the execution logs along the way. The working set at each program point during backtracking is shown besides the machine code. After processing the instruction at 0x100, the external input to define B is identified as suspicious and the backtracking continues to trace dependencies for E, which is still in the working set.

7.3 Intrusion Analysis Based on External Input points Tagging

The major concern of the logging based scheme is that the storage required by logging could be huge and is generally unbounded, since the program can execute for an arbitrarily long time and generate an arbitrarily long execution trace. We could of course set an upper limit to the storage used in logging and discard old execution traces, but the storage limitation will certainly limit the ability of backtracking. The backtracking process may not be able to determine the point of tampering based on the limited information and the program may have to restart from the beginning. This concern prompts us to seek for an alternative scheme that possibly requires less storage. One important observation regarding the logging based scheme is that many entries in the execution trace become useless during the program execution, which is illustrated by a simplified example shown in Figure 83.

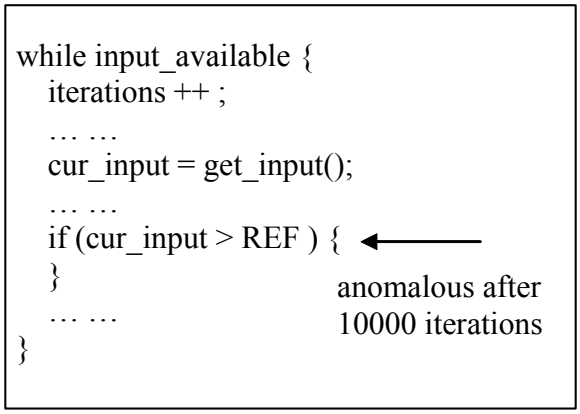


Figure 83. The problem of the logging based scheme.

In the simplified example, the variable cur_input is redefined in every loop iteration. Assume the home location of the variable is in memory, a memory-write log and a memory-read log will be generated for the variable during each loop iteration. Assume an anomaly occurs after 10,000 iterations. At that time, the execution trace

already has 10,000 memory-read logs and 10,000 memory-write logs for the variable but they are completely useless in this case, because the anomalous instruction only depends on the value defined in the current loop iteration. The vast amount of useless logs leads to great space inefficiency in the execution trace. Moreover, there is no good way to delete those useless logs from the execution trace.

Thus, we want to develop a scheme to avoid the huge space inefficiency and reduce the storage requirement imposed by the execution trace. One critical point is that if we aim to keep an execution trace of the program, the storage requirement will inevitably be huge since the program can execute for an arbitrarily long time and at a very high speed in modern processors. We can perform all kinds of optimizations to reduce the events to be logged and the size of the log, but the storage requirement would still increase largely linearly with the instructions executed. So we probably have to examine the problem from a different perspective.

Let us revisit our goal. Our final goal is to identify a set of suspicious external input points that possibly cause the anomalous instruction identified. In other words, we want to find out what external data the anomalous instruction depends on. One way to do that is to keep a dynamic execution trace then backtrack the trace when an anomalous instruction is identified, as discussed previously. The important observation is that another way to achieve the goal is to dynamically update the depended external input data for every register and every memory location. In that way, the information is computed along the program execution. When an anomalous instruction is identified, we know which set of external input data it depends on immediately. The advantage of this scheme is possibly less storage requirement. The number of registers is fixed and very small. The

number of memory locations could be huge but normally the memory space used by the program is much smaller than the virtual address space limit. Moreover, the growth rate of the used memory space is magnitudes smaller than the growth rate of the instructions executed.

We know the number of registers and memory locations is bounded. The important question remaining is that how much storage space is required for each register or memory location. For each register or memory location, we need to record its depended external input data point. There are two important points here. First, the number of static external data input points is very limited. External data is normally received through system calls and sometimes through I/O port operations. Both system calls and I/O port operations are relatively rare events in the program. Second, we have to track dynamic instances of static external input points. Essentially we want to find out a time point from which we can recover the tampered program safely. So we have to identify each dynamic instance of external input points.

To identify dynamic external input points, we could use a global counter initialized to zero. The global counter is increased by one after each dynamic external input point and is used to identify the current dynamic external input point. During execution, the hardware updates the depended external input points for registers and memory locations properly along the program execution. The algorithm is shown in Figure 84. For clarity, the algorithm assumes a RISC instruction set architecture.

```
Input: The dynamic instruction stream
Output: For each register or memory location, a set of external input points
        depended by it

set REG_DEF (inst) = defined register of the instruction
set REG_USE (inst) = used registers of the instruction
set MEM_DEF(inst) = defined memory location
set MEM_USE(inst) = used memory location
set global_counter = 0

Algorithm: External input points tagging
1.  While program_executing do
2.    switch cur_inst->type
3.    case EXT_INPUT:
4.      for each register reg defined directly by the external input
5.        cur_rec = fetch_ext_points_record(reg)
6.        *cur_rec = {global_counter};
7.      end for
8.      for each memory address mem defined directly by the external input
9.        cur_rec = fetch_ext_points_record(mem)
10.       *cur_rec = {global_counter};
11.     end for
12.     global_counter ++
13.   case LOAD:
14.     mem_rec = fetch_ext_points_record(MEM_USE(cur_inst)
15.     reg_rec = fetch_ext_points_record(REG_DEF(cur_inst)
16.     *reg_rec = *mem_rec
17.   case STORE:
18.     reg_rec = fetch_ext_points_record(REG_USE(cur_inst)
19.     mem_rec = fetch_ext_points_record(MEM_DEF(cur_inst)
20.     *mem_rec = *reg_rec
21.   case OTHER:
22.     def_rec = fetch_ext_points_record(REG_DEF(cur_inst)
23.     *def_rec = ∅
24.     for each reg in REG_USE(cur_inst)
25.       use_rec = fetch_ext_points_record(reg)
26.       *def_rec = *def_rec ∪ *use_rec
27.     end for
28.   end switch
29. endw
```

Figure 84. External input points tagging algorithm.

During program execution, the external input points tagging algorithm processes each committed instruction. For each register or memory location, it maintains a set of external input points depended by the register or the memory location. If the instruction receives external inputs, for example, a system call reading keyboard inputs, the memory locations or the registers defined are tagged by the ID of the external input point. At such

an external input point, a block of memory locations may be defined and all memory locations defined are tagged. For a load instruction, the algorithm copies the external input points set of the referenced memory location to the set of the register. For a store instruction, the algorithm copies the external input points set of the referenced register to the set of the memory location. For other instructions, the external input point sets of the referenced registers are merged then copied into the set of the defined register.

The algorithm is straightforward. Conceivably the algorithm could reduce the storage requirement of the logging based scheme since the number of memory locations and registers is much smaller comparing with the number of dynamic instructions. However, as always, the benefit comes with a cost. In the logging scheme, we only need to append newly generated logs to the execution trace. This no longer holds for external input points tagging scheme. Under the scheme, each memory location or register has a corresponding set of external input points. This set is manipulated frequently during tagging, which brings challenges to the design of the data structure. Moreover, although the number of memory locations and registers are bounded, the number of dynamic external input points possibly tagged for each memory location or register is not bounded. The problem is shown in Figure 85. In this simplified example, the memory location of the variable result will be tagged with all dynamic external input points generated during get_input. This number can be arbitrarily large. Although this kind of situation should be rare in reality, the fact that there is no upper bound of the number of elements in the set adds to the difficulty of the data structure design.

```
while input_available {
   iterations ++ ;
   … …
   cur_input = get_input(); // external input
   … …
   result = result + cur_input;
   … …
}
```

Figure 85. The difficulty of external input points tagging.

Next, we discuss the data structure design of the external input points set. We only discuss the handling of memory locations. Registers are very limited and can be modeled as a tiny piece of data memory and handled in the same way. We assume that there is a segment of process memory space reserved to the external input points set. This reserved memory space cannot be accessed through machine instructions and can only be accessed by the hardware directly thus is protected. First, we want to be able to locate an EIP set by the memory location easily. In other words, the memory location can be transformed into its corresponding EIP set address through simple operations. To achieve that, the general principle is that we lay out the EIP sets by the same order as their corresponding memory locations. That is, the EIP set of the first memory location is laid out first in the EIP address space. Then the EIP set of the second memory location follows. The non-continuous data address space brings extra complications. Normally the program data space is divided into static data space, heap space and stack space. Figure 86 shows a common layout. Static data space is easy to handle since its size is fixed. Stack space and heap space grow at runtime and have no fixed size. The best we can do is to pre-allocate the amount of memory corresponding to the expected upper limit of the

stack and heap space, then allocate more memory and repair the data structure if at runtime this upper limit is broken.

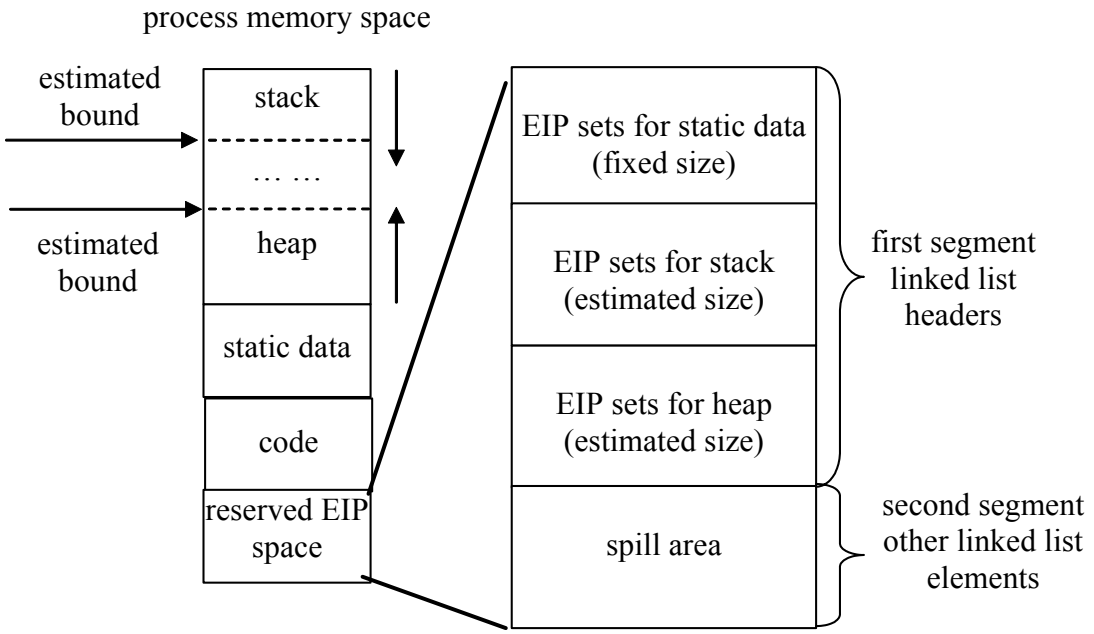


Figure 86. A typical process space layout.

EIP_base = base address of EIP space
static_base = base address of static data space

EIP_address of static_address =

$$EIP\_base + \frac{(static\_address - static\_base)}{mem\_tagging\_unit\_size} * EIP\_unit\_size$$

Figure 87. Finding the corresponding EIP address for static memory address.

Figure 87 shows how we compute the corresponding EIP set address for a static memory address. The handling of stack and heap addresses is similar. The EIP tagging unit is normally a machine word, but we could choose a larger tagging unit to reduce the memory requirement of the EIP tagging scheme. To locate the corresponding EIP set quickly as shown in the equation, we need to make each EIP set record fix-sized. At the

same time, we have to be able to handle EIP sets containing an arbitrarily large number of elements. To achieve that, we design a linked-list based data structure. Each EIP set is effectively a linked list. Each linked list element contains a fixed number of EIP IDs and a pointer to the next linked list element if there is one. We divide the EIP space into two segments. The first segment only contains headers of the linked lists representing EIP sets. The headers are laid out by the same order of the corresponding memory locations thus the header of the linked list representing the EIP set for a memory location can be easily located. The second segment is the spill area. Only when one linked list element is not enough to record all EIP Ids, is the spill area utilized. When one header element is not enough, the tagging scheme finds out an unallocated space in the spill area and creates an additional linked list element, then appends the new element to the end of the list. There are no layout constraints for spilled linked list elements. They are allocated wherever a free space is found. Figure 86 shows what the EIP space looks like.

```
struct link_list_elm {
   id_t  EIP_ids[MAX_ID_PER_ELM];
   addr_t next;
};

struct EIP_space_t {
   struct link_list_elm header_space[MAX_HEADER_ELMS];
   struct link_list_elm spill_area[MAX_SPILL_ELMS];
} EIP_space;
```

Figure 88. EIP space data structures.

For further clarification, Figure 88 shows the pseudo code for our data structure design. Initially, the EIP space is cleared. A zero EIP id in a linked list element means that slot is unused. If the first EIP id in a linked list element in the spill area is zero, that means an unused linked list element space. The pseudo code to add an EIP id to the EIP

set of a memory location is shown in Figure 89. Note that both header space and spill area space can become too small at runtime. To expand spill area is straight forward since it is at the end of the EIP space. We can simply allocate more memory to the EIP space. To expand header space will invalidate all pointers to the spill area, thus the data structures have to be repaired. The pseudo code to clear an EIP set or to delete the linked list representing the EIP set is not shown but is also easy to implement.

```
unsigned int spill_area_ptr = 0;

void init() {
   mem_zero(&EIP_space, sizeof(EIP_space));
}

void add_EIP_id ( addr_t mem_addr, id_t id) {

   addr_t EIP_addr = get_EIP_set_addr( mem_addr );
   struct link_list_elm * list_header = (struct link_list_elm*) EIP_addr;
   struct link_list_elm * cur_list_elm = list_header;

   while (cur_list_elm) {
      for(int i=0; i<MAX_ID_PER_ELM; i++) {
         // zero id means unused
         if(!cur_list_elm→EIP_ids[i]) {
            cur_list_elm→EIP_ids[i] = id;
            return;
         }
      }
      cur_list_elm = (struct link_list_elm*) cur_list_elm→next;
   }
   // have to create a new linked list element
   // first find an unused slot
   saved_spill_ptr = spill_area_ptr;
   while ( EIP_space.spill_area[spill_area_ptr].EIP_ids[0]) {
      spill_area_ptr = (spill_area_ptr + 1) % MAX_SPILL_ELMS;
      if(spill_area_ptr == saved_spill_ptr)
          PANIC("spill area full")
    }
    // found an empty slot
    EIP_space.spill_area[spill_area_ptr].EIP_ids[0] = id;
}
```

Figure 89. Algorithm to add an EIP id into the proper EIP set.

7.4 Intrusion Recovery

After intrusion analysis is properly done and tampered memory state is identified, intrusion recovery for the memory state becomes a simple problem.

If intrusion analysis is done by the execution trace logging based algorithm, to enable intrusion recovery, we can simply record the overwritten value for each memory write operation. There will be an additional field for saving the old value of the memory location. The modified data structure for the execution trace log is shown in Figure 90. The difference is marked as bold.

```
enum  log_type_t { PATH, MEM};
enum  mem_type_t  {READ, WRITE, EXT_INPUT};
struct log_t {
      log_type_t log_type;
      union {
            struct  {
                  addr_t    pc;
                  addr_t    target;
            } path_info;
            struct  {
                  mem_type_t mem_type;
                  addr_t     address;
                  data_t     old_val;
            } mem_info;
      } log_info;
};
```

Figure 90. Modified data structure of a log record.

To recover from tampered memory state, during dependency backtracking, when a memory write log record is processed, the value of the memory location is set to the old value recorded in the memory write log record. When the backtracking process is finished, the memory state will be recovered to a safe one. The relevant modification to the backtracking algorithm is shown in Figure 91.

```
14.       else  // STORE
15.         Assert(cur_log->mem_info.mem_type == WRITE)
16.         if (cur_log->mem_info.address ∈ W)
17.             W = W – cur_log->mem_info.address; W = W ∪ REG_USE(cur_inst);
18.         endif
            // recover possibly tampted memory state
19.          MEM[cur_log->mem_info.address] = cur_log->mem_info.address.old_val;
20.       endif
```

Figure 91. Modified backtracking algorithm.

If intrusion analysis is based on external input points tagging, intrusion recovery can be done by a checkpointing based scheme. The memory state can be checkpointed incrementally at a certain interval, such as one millisecond. Each checkpoint does not need to save the whole memory state. It just records the memory writes in between the previous checkpoint and the current checkpoint. Each checkpoint has a timestamp, which is just the value of the global counter counting the number of dynamic external input points. With the set of identified suspicious dynamic external input points, the recovery algorithm is straightforward. Each dynamic external input point is identified by the value of the global counter when the external input happens. Assume the smallest ID in the set of suspicious dynamic external input points is MIN, to recover tampered memory state, we just need to process the checkpoints until its timestamp is smaller than MIN.

7.5 Experiments and Results

To evaluate the recovery strength of the proposed logging and tagging based intrusion recovery schemes, we implement them in an open-source IA-32 system emulator Bochs [9] with Linux installed. Then we perform simulated attacks to our daemon programs. We run each benchmark in the emulator for a long period of 10 billion instructions. During its execution, we tamper its memory state by corrupting a random memory location at a random external input point, even though originally the external

input point is innocuous. If the memory corruption is detected, we then check whether our intrusion analysis and intrusion recovery scheme is able to identify the malicious external input point and recover the tampered memory state. Each server program is attacked 10000 times independently.

We have discussed three intrusion detection schemes in this dissertation, including anomalous path detection, infeasible path detection and data tampering detection. The introduced memory tampering may or may not be detected by our intrusion detection schemes. If our data tampering detection scheme is able to detect the memory corruption, it will detect it immediately. In other words, if the data tampering detection scheme is deployed, the malicious external input point can be identified immediately and the tampered memory state can be recover easily. Thus, we will not show the intrusion recovery results for the data tampering scheme. For anomalous path detection and infeasible path detection schemes, they are based on control flow monitoring and they try to infer memory tampering from control flow tampering. Thus, they will not be able to detect the attack until the tampered memory state is used. There could be an arbitrarily long interval between the memory tampering and the reference of the tampered memory state, which brings great difficulty to the intrusion analysis and intrusion recovery schemes. After the tampered state is used, both anomalous path detection and infeasible path detection schemes can detect the attack within a very small interval if they can detect it at all. For the anomalous path detection scheme, the attack will be detected with n dynamic branches if the scheme can detect the attack. For the infeasible path detection scheme, the interval between the tampered memory state reference and the attack detection is also very small generally since it is very hard for the

compiler to detect a branch correlation between two branches far away from each other. Thus, the anomalous path detection scheme and the infeasible path detection scheme behave very similarly in terms of intrusion recovery, since the interval between the memory tampering and the reference of the tampered memory state is the dominating factor. Thus, we will only show the results for the anomalous path detection scheme. The infeasible path detection scheme achieves almost identical results.

For the logging based scheme, the storage space required for a complete logging of the program execution is linear to the number of dynamic instructions executed. Thus, a complete logging of the program execution is generally not possible. We have to set an upper limit of the storage space reserved for logging. If the time interval between the memory tampering and the attack detection is too large, the space reserved for logging may not be enough to record all the necessary information to identify the attack and recover the memory state completely. In general, the more space reserved for logging, the better intrusion identification and recovery strength.

Figure 92 shows the results of intrusion identification under the logging based scheme. In this experiment, we assume only intrusion identification is required but not intrusion recovery. The intrusion identification rate of our simulated attacks under the logging scheme is shown in the figure. We show the results under a 10MB reserved space, a 100MB reserved space and a 1GB reserved space respectively. To merely identify the cause of an attack, we do not need to save the old value of a memory location, thus the storage requirement is smaller. By identifying an attack, we mean to find out the exact malicious external input points causing the attack. From the results, with a 10MB reserved space for logging, on average we can successfully identify 72.9% of attacks.

With a 100MB reserved space for logging, on average we can identify 84.6% of attacks. With a 1GB reserved space for logging, on average we can identify 94.3% of attack. We can see that a small reserved space such as 10MB can achieve a good intrusion identification rate, since if the tampered memory value is going to be referenced, it is more likely to be reference sooner than later. Also, the intrusion identification rate increases with the reserved space for logging.

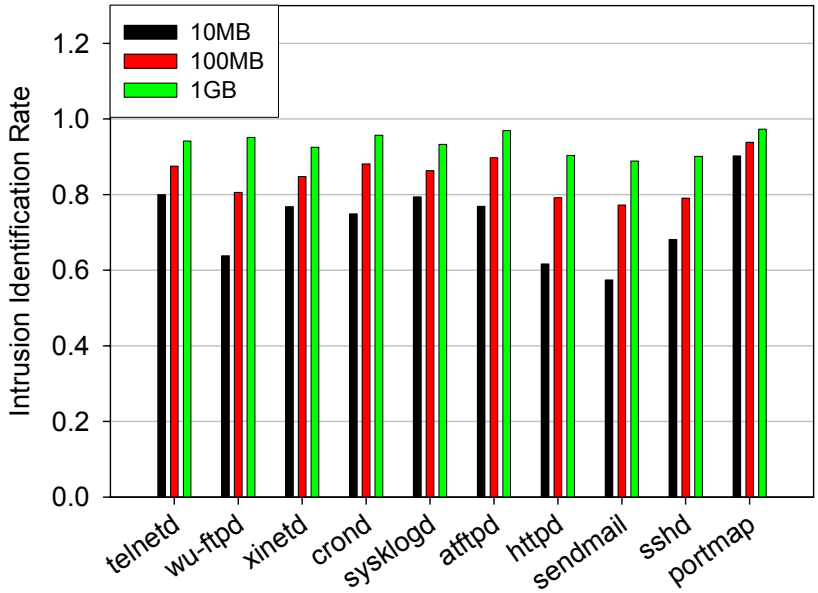


Figure 92. Intrusion identification under the logging based scheme.

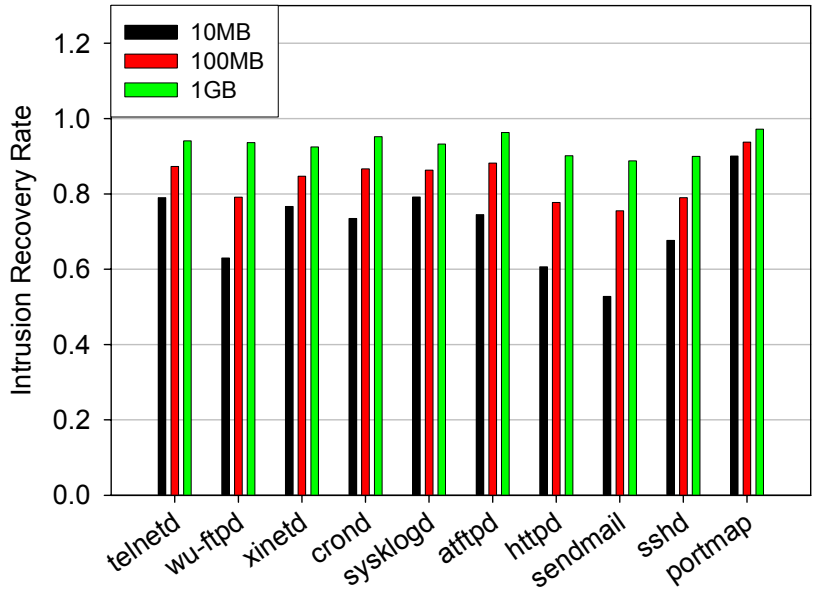


Figure 93. Intrusion recovery rate under the logging based scheme.

To both identify and recover from an attack, the execution log has to save the old value of a memory location if there is a write to the memory location. Thus, the size of a record in the execution log is increased, so is the space overhead for logging. Figure 93 shows the intrusion recovery rate of the simulated attacks under the logging scheme. We show the results under a 10MB reserved space, a 100MB reserved space and a 1GB reserved space respectively. With a 10MB reserved space for logging, on average we can successfully identify and recover 71.7% of attacks. With a 100MB reserved space for logging, on average we can identify and recover 83.8% of attacks. With a 1GB reserved space for logging, on average we can identify and recover 93.1% of attack. From the results, given the same reserved space for logging, enabling recovery increases space overhead but only slightly. The fundamental reason is that the interval between memory tampering and attack detection for a large part of attacks is small enough so that the reserved space is enough for both intrusion identification and intrusion recovery.

The external input points tagging scheme works in a different way. The space requirement of the EIP tagging scheme does not increase linearly with the number of dynamically executed instructions. Instead, the space requirement is related to the amount of data memory used by the program since each memory location could possibly be tagged. The data memory used by the program is generally very limited. The size of it will become stable and will not increase forever as the number of dynamically executed instructions. Also, we found that most memory locations depend on zero or only one external input point at runtime. Thus, the EIP tagging scheme has a better space efficiency than the logging based scheme.

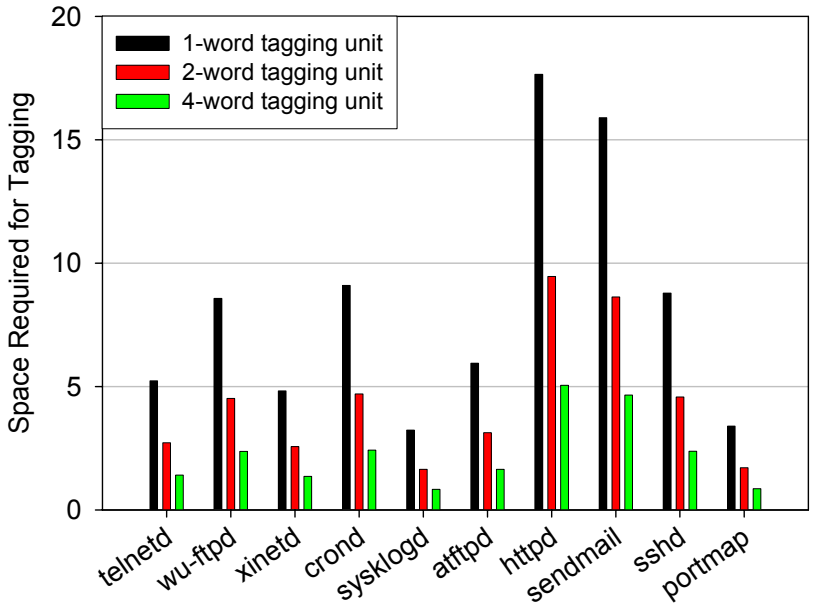


Figure 94. EIP tagging space requirement.

Figure 94 shows the space requirement of the EIP tagging scheme under three different tagging units. The unit of tagging has a great impact on the space requirement. Our finest tagging granularity is one machine word. In other words, each machine word will be tagged separately with the depended external input points. We also show the

results for the tagging units of two machine words and four machine words. With the tagging unit of two machine words, two adjacent machine words share one tag. The tag will record depended external input points for both words. As long as those two machine words share some depended external input points, we have space savings. The tagging unit of four machine words works in a similar way. The space requirement under the tagging unit of one machine word is 8.26MB on average, ranging from 3.24MB to 17.65MB. The space requirement under the tagging unit of two machine words is 4.37MB on average, ranging from 1.64MB to 9.46MB. The space requirement under the tagging unit of four machine words is 2.30MB on average, ranging from 0.84MB to 5.05MB. Doubling the tagging unit can roughly cut the space requirement to half, showing that many memory locations share the same depended external input points. It is expected since normally an external input set the values for a block of memory at the same time instead of a single machine word.

A larger tagging unit brings adverse effects to intrusion identification. Assume the attacker always tampers at least a machine word, the tagging unit of one machine word can identify an attack immediately and accurately, and the EIP tagging scheme can achieve 100% of intrusion identification rate. With a tagging unit greater than one machine word, we will not know whether one tagged external input point is depended by the memory location investigated or by other locations in the same tagging unit. We have to be conservative, which leads to inaccuracy in intrusion identification.

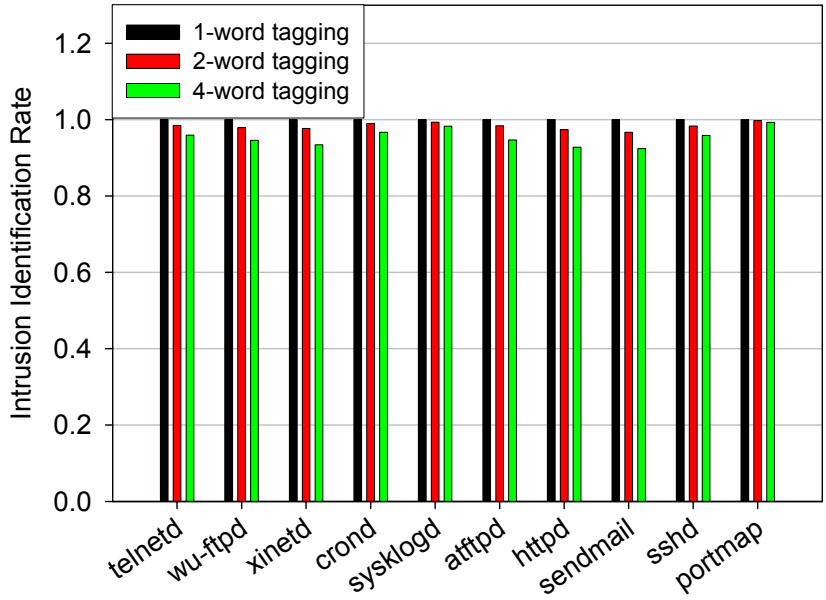


Figure 95. Intrusion identification rate of EIP tagging scheme.

Figure 95 shows intrusion identification of the EIP tagging scheme under different tagging units. If we assume the attacker always tampers at least a machine word, the tagging unit of one machine word brings us 100% intrusion identification rate. Moreover, the EIP tagging scheme can identify the attack immediately without processing a large amount of logging data as in the logging scheme, since all suspicious external input points have been tagged properly. With a tagging unit of two machine words, the intrusion identification rate is 98.3% on average. With a tagging unit of four machine words, the intrusion identification rate is 95.4% on average.

To enable intrusion recovery, the EIP tagging scheme has to checkpoint memory state so that the memory state can be recovered after the attack point is identified. Memory checkpointing can be incrementally done by logging memory writes and recording the old values of the memory location. Figure 96 shows the intrusion recovery rate of the EIP tagging scheme under a certain space limitation for checkpointing. We

show the results under a 10MB reserved space, a 100MB reserved space and a 1GB reserved space respectively. With a 10MB reserved space for checkpointing, on average we can recover 76.7% of attacks. With a 100MB reserved space for checkpointing, on average we can recover 86.7% of attacks. With a 1GB reserved space for checkpointing, on average we can recover 95.1% of attacks. The EIP tagging scheme is more space efficient than the logging based scheme. Under a certain space limitation, the EIP tagging scheme can achieve better intrusion recovery rate. The improvement is attack and application-dependent and is seen to be up to 24% in our experiments.

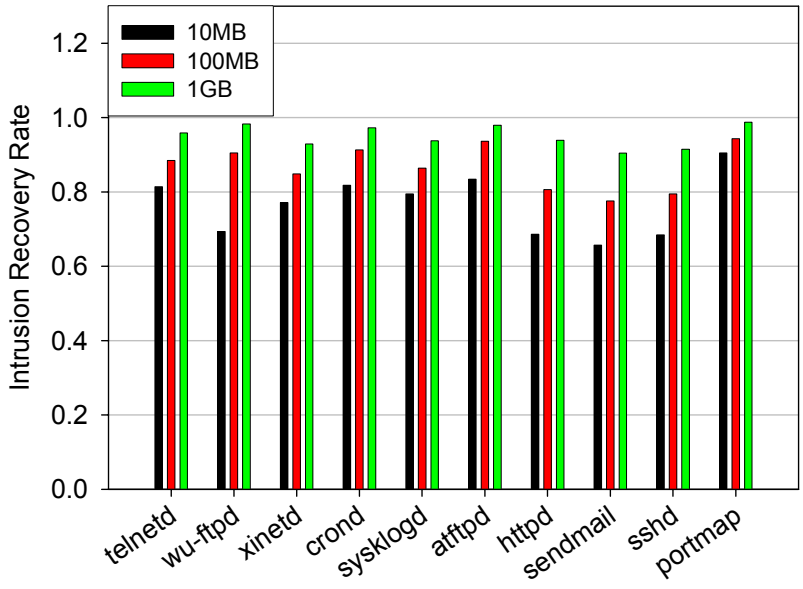


Figure 96. Intrusion recovery rate of EIP tagging scheme.

Next we discuss the performance aspect of our intrusion identification and intrusion recovery scheme. For the logging based scheme, there is no computation involved during logging. As long as there is enough bandwidth to transfer the log data, there will be no performance degradation. For the external input points tagging scheme, the tag for each memory location has to be computed at runtime. The computation is very

simple as shown previously, especially most memory locations are tagged by no more than one external input point. But we assume that the tag data will share cache with the program data, thus accessing the tags for memory locations increase cache pressure and degrades performance. Also, the external input points tagging scheme needs to checkpoint memory state to recover tampered state later, thus it also requires certain memory bandwidth for the checkpointing.

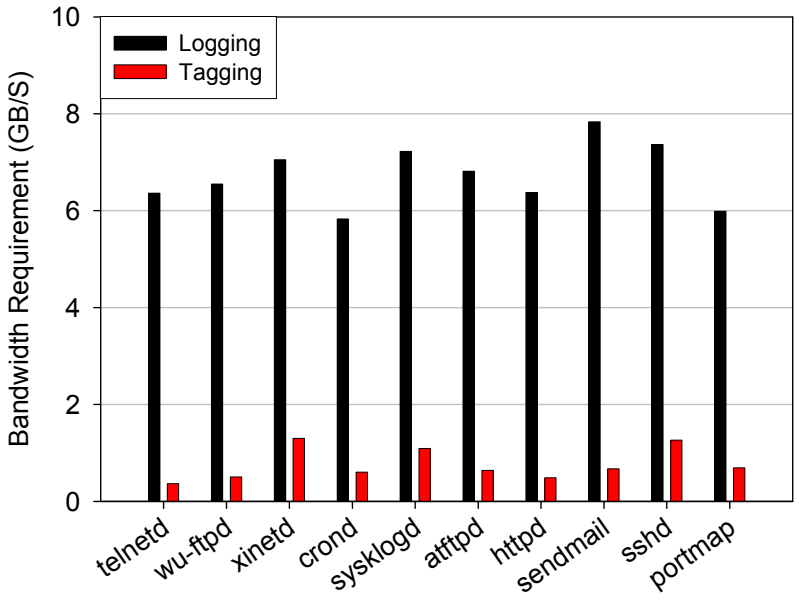


Figure 97. Bandwidth requirement.

Figure 97 shows the bandwidth requirement for the logging based schema and the EIP tagging scheme. Under the EIP tagging scheme, we only need to record memory writes. Under the logging scheme, we have to record both memory reads and memory write, plus control flow transfers. Since the number of memory writes is normally only a fraction of memory reads, the EIP tagging scheme requires much smaller bandwidth. On average, the logging scheme requires a memory bandwidth of 7.8GB/s and the EIP tagging scheme requires a memory bandwidth of 673MB/s. We can see that due to its

space efficiency, the EIP tagging scheme requires much smaller bandwidth. A memory bandwidth of about 8GB/s is quite realistic in the future. In fact, Xbox 360 game console has 278.4GB/s of memory system bandwidth [87] and the IBM CELL architecture has a theoretic 204.8GB/s peak memory bandwidth [117].

To measure the performance degradation of the external input points tagging scheme, we model the tagging hardware using SimpleScalar [11] tool set. The parameters for the processor modeled are shown in Table 10.

Table 10. Parameters of processor simulated.

Clock frequency	1 GHz	Branch predictor	2 Level
Fetch queue	16 entries	BTB	512 entries, 4 -way
Decode width	4	PLB	128 entries
Issue width	4	L1 I/D	DM, 32K, 1 cycle 32B block
Commit width	4	Unified L2	8way, 32B block 1M (16 cycles)
RUU size	64	Memory bus	200M, 8 Byte wide
LSQ size	32	Memory latency	first chunk: 120 cycles, inter chunk: 10 cycles

Figure 98 shows the performance results for the EIP tagging scheme normalized to the baseline without tagging enabled. The results under three different tagging units are shown. Tagging with finer granularity requires more tagging data space and incurs more cache pressure, thus leading to worse performance. With 1-word tagging unit, the average performance degradation is 20.8%. With 2-word tagging unit, the average performance degradation is 15.1%. With 4-word tagging unit, the average performance degradation is 10.9%. We can always choose a larger tagging unit in exchange for a better performance and a worse intrusion identification rate.

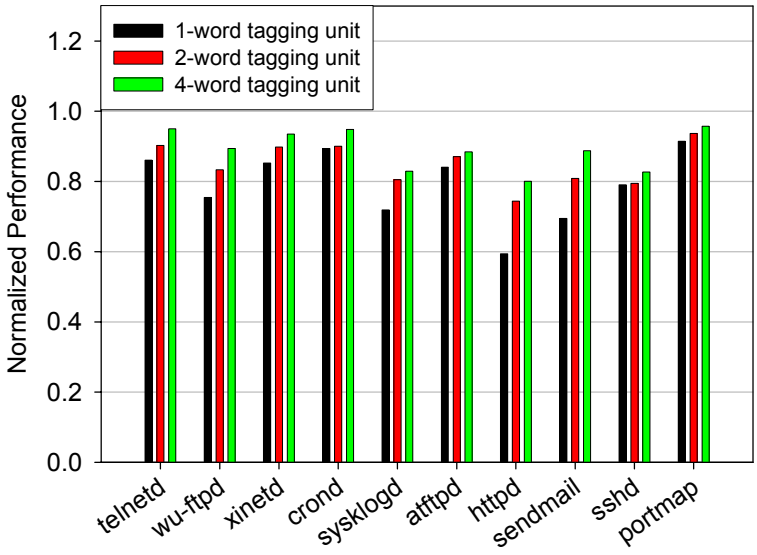


Figure 98. Performance degradation of the EIP tagging scheme.

7.6 Summary

In this chapter, we focus on intrusion analysis to identify the cause of an attack and the tampered memory state and intrusion recovery to recover the tampered memory state. Intrusion analysis and intrusion recovery is generally a difficult problem because there could be an arbitrarily long interval between the tampering of the memory state and the detection of the attack, depending on the underlying intrusion detection mechanism.

We propose two intrusion analysis and intrusion recovery schemes in this chapter. The first is based on execution trace logging. The second is based on external input points tagging. The logging based scheme requires a high memory bandwidth, is less space efficient but has little performance degradation. The tagging based scheme is more space and bandwidth efficient, but brings significant performance degradation. Given enough storage space, both schemes can achieve a satisfying intrusion identification and intrusion recovery rate. User can choose a scheme to deploy according to the properties of the schemes. Overall, our study on compiler and micro-architecture supported intrusion

analysis and intrusion recovery is still preliminary. Both schemes proposed have their drawbacks and how to make them more space or performance efficient is the future work.

8 CONCLUSIONS AND FUTURE WORK

8.1 Conclusions

Computer systems virtually control every aspect of our modern society. The rapid increase of the society's dependence on computer systems brings great interest to break in and attack those systems. Computer software determines the functionality of computer systems and is the primary target of attacks. With today's pervasive presence of computer systems and network connections, securing critical software from attacks has become an extremely important problem and has never been as challenging.

In this dissertation, we build a software protection infrastructure by utilizing both compiler and micro-architecture support. We name our infrastructure RADAR – compileR and micro-Architecture supported intrusion prevention, Detection, Analysis and Recovery. Our infrastructure aims to achieve a complete solution to software protection. It is designed to defend against both common hardware attacks and common software attacks and tackles all three major aspects in software protection – intrusion prevention, intrusion detection and intrusion recovery. Our infrastructure emphasizes collaborations between compiler and micro-architecture. It is based on micro-architecture level support and has its security rooted in hardware. At the same time, it calls for compiler assist whenever it is necessary, such as to obtain expected software behavior, or whenever it is helpful to reduce the complexity of the micro-architecture support. With both micro-architecture and compiler support, our infrastructure can defend against both software and hardware attacks with superb security strength but reasonable hardware and performance cost.

Traditional software protection approaches are purely based on software. The protection strength provided by a purely software-based approach is far from enough considering the ever increasing importance of critical software protection and the constantly emerging new threats faced by critical software, especially realistic hardware attacks.

Introducing hardware support for software protection has been shown is the future direction to go. There has been active research on micro-architecture level support for software protection. But up until now, the previous secure architecture research mainly focuses on using cryptographic mechanisms to protect software confidentiality and software integrity from hardware attacks and attacks from other malicious software processes. The solutions proposed are purely hardware based.

In this dissertation, we first show that even though the problem scope of the previous secure architecture research is quite limited, the solution proposed to the problem is not complete and has a serious flaw. Previous secure architectures only protect confidentiality of the traffic going through system data bus. They do not protect system address bus traffic, i.e., addresses of memory accesses. However, unprotected memory access sequence acts as very dangerous side-channel and leaks critical control flow information of the protected software. We show that the information leakage could bring significant damage to both code and data confidentiality. Thus, we propose an improved hardware infrastructure enhanced with address bus information leakage prevention. Our address bus protection scheme relies on both innovative hardware modification and extensive compiler support to eliminate most of the information leakage on system

address bus with little performance overhead. Our scheme is the first practical scheme to prevent information leakage on the system address bus.

In general, no security system is bullet-proof and is able to prevent all attacks. Even our enhanced hardware infrastructure cannot prevent attacks exploiting flaws/bugs in the protected software itself, such as buffer overflow attacks. To protect software from attacks missed by the intrusion prevent mechanism, we build a second line of defense consisted of intrusion detection and intrusion recovery mechanisms, which is able to detect both known and unknown attacks and even recover from those attacks.

Intrusion detection with both compiler and micro-architecture support is another major contribution of this thesis work. We show that intrusion detection based on compiler and micro-architecture collaboration can achieve a much finer monitoring granularity than software-based intrusion detection schemes thus much stronger detection strength but only with small hardware and performance cost. We have done extensive research on this area. We have greatly developed our understanding to the problem during our research and devised three intrusion detection schemes accordingly.

We first propose a training based scheme to detect anomalous program paths caused by attacks. Anomaly detection based on training is a classical approach. But with carefully designed hardware support, our approach is able to achieve both strong detection capability and negligible performance degradation.

The requirement of good training and potential false positives in the anomalous path checking scheme would limit its applicability in reality, which prompts us to devise a scheme with zero false positives. Thus, we develop an infeasible path detection scheme to detect attacks based on static compiler branch correlation analysis and hardware

runtime support. The scheme achieves zero false positives and only incurs small hardware and performance cost, which boosts its applicability in the real world. But this scheme is still based on control flow monitoring and the detection strength is significantly worse than the anomalous path checking scheme due to the limitation of static compiler analysis.

Both anomalous path checking and infeasible path detection are based on control flow monitoring. They cannot detect attacks tampering memory but not altering control flows at all. However, such attacks have been shown to be very realistic. This problem prompts us to work on the root cause of attacks – memory tampering. We thus devise a compiler and micro-architecture collaboration framework to detect memory tamperings directly instead of trying to infer memory tamperings from anomalous control flows. The data tampering detection scheme is based on compiler data flow analysis thus achieves zero false positives too. Moreover, it attacks the root cause of intrusions thus has a very strong detection strength. However, the performance overhead is significant even after extensive compiler optimizations.

As a conclusion, among three properties of detection strength, zero false positives and performance, anomalous path checking prioritizes detection strength and performance; infeasible path detection prioritizes zero false positives and performance; finally data tampering detection prioritizes detection strength and zero false positives. All three schemes should find their applications in reality according to the different user requirements under different scenarios.

Finally, a complete software protection scheme should have the ability to analyze an attack and recover from the attack whenever possible. Thus, intrusion analysis and

intrusion recovery is an important goal of our software protection infrastructure. We propose two intrusion analysis and intrusion recovery schemes in this dissertation. The first is based on execution trace logging. The second is based on external input points tagging. The logging based scheme requires a high memory bandwidth, is less space efficient but has little performance degradation. The tagging based scheme is more space and bandwidth efficient, but brings performance degradation. Given enough storage space, both schemes can achieve a satisfying intrusion identification and intrusion recovery rate. Our research on intrusion analysis and intrusion recovery is still in the beginning stage and future work should be done to improve the efficiency of our current intrusion analysis and intrusion recovery schemes.

We draw the following higher level conclusions. Compiler analysis is able to provide information about program behavior and generate attributes that could be verified at runtime by the hardware. On the other hand, clever hardware techniques allow tractable management of large amounts of information at runtime to track and synthesize essential security attributes that must be verified. Semantic gap between software behaviors that get exploited and hardware support that provides security substrate can be thus effectively bridged for a variety of attacks based on side channel information, memory tampering that may or may not alter control flow, and etc. This compiler and micro-architecture collaboration approach leads to better security guarantees and strengths.

8.2 Future Work

Regarding to compiler and micro-architecture supported intrusion detection, we believe that more work should be done on the data tampering detection scheme due to its

multiple advantages. First of all, the scheme works on the root cause of attacks directly and can achieve very strong detection strength and detect memory tampering attacks not altering program control flows at all, which will be missed by control flow monitoring based schemes. Second of all, the scheme is able to achieve zero false positives, which is critical to the applicability of an anomaly detection scheme. Finally, the scheme also has significant advantages in terms of intrusion recovery as discussed in the last chapter. The current bottleneck of the deployment of the scheme is primarily its performance cost. More compiler and micro-architecture optimizations could be investigated to reduce the performance overhead thus making the scheme generally affordable. We also need to investigate the impact of multithreading to the scheme. Multithreading could lead to false positives if multiple threads share the same access permission information. The most naïve way to tackle this problem is to make each thread have its own access permission information, but it is obviously too expensive and inefficient.

Our research on intrusion analysis and intrusion recovery is still on its preliminary stage. Neither the execution trace logging based scheme nor the external input points tagging base scheme is a satisfying solution to the problem of intrusion analysis and intrusion recovery yet. The execution trace logging based scheme requires a high memory bandwidth. On the other hand, the external input points tagging base scheme incurs significant runtime overhead. Optimizations to both schemes should be explored in the future work. A more efficient solution is yet to be devised.

REFERENCES

[1] Agrawal, H., DeMillo, R. A., and Spafford, E. H., "An execution-backtracking approach to debugging", In IEEE Software, May 1981.

[2] Anderson, R. J., and Kuhn, M., "Low Cost Attacks on Tamper Resistant Devices," Security Protocols Workshop, 1997.

[3] Anderson, R. J., "Security Engineering", John Wiley and Sons publishers, February 2001.

[4] Arbaugh, W. A., Farber, D. J., and Smith, J. M., "A Secure and Reliable Bootstrap Architecture", in Proceedings 1997 IEEE Symposium on Security and Privacy , pp. 65-71, May 1997.

[5] Aucsmith, D., "Tamper Resistant Software: An Implementation", Information Hiding: 1st Int. Workshop (Lecture Notes in Computer Science), vol. 1174, R.J. Anderson, Editor, Springer Verlag, 1996, Berlin, pp.317-333.

[6] Austin, T. M., Breach, S. E., and Sohi, G. S., "Efficient detection of all pointer and array access errors", ACM SIGPLAN Notices, 29(6), 1994.

[7] Avizienis, A., "The N-version approach to fault-tolerant software," IEEE Trans. Software Eng., Vol. SE-11, No. 12, pp. 1491–1501, Dec. 1985.

[8] Baratloo, A., Tsai, T., and Singh, N., "Transparent run-time defense against stack smashing attacks", In Proceedings of USENIX Annual Technical Conference, June 2000.

[9] Bochs. "Bochs: the Open Source IA-32 Emulation Project," http://bochs.sourceforge.net (Accessed July 18, 2006).

[10] Bodik, R., Gupta, R., and Soffa, M. L., "Interprocedural Conditional Branch Elimination", ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 146-158, Las Vegas, Nevada, June 1997.

[11] Burger, D., and Austin, T. M., "The SimpleScalar Tool Set Version 2.0," TR. 1342, Univ. of Wisconsin--Madison, May 1997.

[12] Business Software Alliance, “Second Annual BSA and IDC Global Software Piracy Study”, http://www.bsa.org/globalstudy/upload/2005-Global-Study-English.pdf (Accessed July 18, 2006).

[13] Callahan, D., and Kennedy, K., “Analysis of Interprocedural Side Effects in a Parallel Programming Environment,” Journal of Parallel and Distributed Computing, 5:517–550, 1988.

[14] Castro, M., and Liskov, B., “Practical Byzantine fault tolerance,” In Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation, February 1999.

[15] CERT. www.cert.org (Accessed July 17, 2006).

[16] Chang, H., and Atallah, M., “Protecting software code by guards”, In Proc. 1st ACM Workshop on Digital Rights Management (DRM 2001), volume 2320 of Lecture Notes in Computer Science, pages 160–175. Springer-Verlag, 2002.

[17] Chen, S., Xu, J., Sezer, E. C., Gauriar, P., and Iyer, R. K., “Non-Control-Data Attacks Are Realistic Threats,” in Proc. USENIX Security Symposium, Baltimore, MD, August 2005.

[18] Chiueh, T. C., and Hsu, F. H., “RAD: A compile-time solution to buffer overflow attacks”, In Proc. of 21st Intl. Conf. on Distributed Computing Systems, 2001.

[19] Chow, S. T., Gu, Y., Johnson, H. J., and Zakharov, V. A., “An Approach to the Obfuscation of Control-Flow of Sequential Computer Programs”, In the Proc. of International Security Conference 2001, Malaga, Spain.

[20] Collberg, C., Thomborson, C., and Low, D., “A Taxonomy of Obfuscating Transformations”, Technical Report Number 148, Dept. of Computer Science, University of Auckland, New Zealand, July 1997.

[21] Collberg, C., Thomborson, C., and Low, D., “Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs”, Dept. of Computer Science, University of Auckland, New Zealand, 1998.

[22] Cowan, C., Beattie, S., Johansen, J., and Wagle, P., “Pointguard: Protecting pointers from buffer overflow vulnerabilities”, In 12th USENIX Security Symposium,

Washington DC, August 2003.

[23] Cowan, C., Pu, C., Maier, D., Hinton, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., and Zhang, Q., "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks", In Proceedings of the 7th USENIX Security Symposium, pages 63-78, January 1998.

[24] Creusillet, B., and Irigoin, F., "Exact vs. Approximate Array Region Analyses," In Lecture Notes in Computer Science. Springer Verlag, NY, August 1996.

[25] Cruegel, C., Robertson, W., Valeur, F., and Vigna, G., "Static Disassembly of Obfuscated Binaries", Proc. 13th. USENIX Security Symposium, August 2004, pp. 255-270.

[26] Dallas Semiconductor, "DS5002FP secure microprocessor chip data sheet".

[27] Etoh, H., "GCC extensions for protecting applications from stack-smashing attacks", http://www.trl.ibm.com/projects/security/ssp (Accessed July 18, 2006).

[28] Federal Trade Commission, "Federal Trade Commission – Identity Theft Survey Report", September 2003.

[29] Feldman, S., and Brown, C., "Igor: A system for program debugging via reversible execution", In ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging, January 1989.

[30] Feng, H. H., Giffin, J. T., Huang, Y., Jha, S., Lee, W., and Miller, B. P., "Formalizing Sensitivity in Static Analysis for Intrusion Detection," In Proceedings of the 2004 IEEE Symposium on Security and Privacy, 2004.

[31] Feng, H. H., Kolesnikov, O. M., Fogla, P., Lee, W., and Gong, W., "Anomaly Detection Using Call Stack Information," IEEE Symposium on Security and Privacy, May, 2003.

[32] Forrest, S., Hofmeyr, S. A., Somayaji, A., and Longstaff, T. A., "A Sense of Self for Unix Processes," In Proceedings of the 1996 IEEE Symposium on Security and Privacy, 1996.

[33] Frantzen, M., and Shuey, M., "StackGhost: Hardware facilitated stack protection", In Proceedings of the 10th USENIX Security Sumposium, August 2001.

[34] Gandolfi, K., Mourtel, C., and Olivier, F., "Electromagnetic Analysis: Concrete Results," In Workshop on Cryptographic hardware and Embedded Systems, 2001.

[35] Gannod, G. C., and Cheng, H. C., "Using Informal and Formal Techniques for Reverse Engineering of C Programs", Proc. of the 3 rd Working Conference on Reverse Engineering, Monterey, California, Nov 8-10, pp. 249-258.

[36] Gao, D., Reiter, M. K., and Song, D., "Gray-Box Extraction of Execution Graphs for Anomaly Detection", the 11th ACM CCS conf., pages 318-329, October 2004.

[37] Gao, D., Reiter, M. K., and Song, D., "On Gray-Box Program Tracking for Anomaly Detection", 13th USENIX Security Symposium, pages 103-118, August 2004.

[38] Gassend, B., Suh, G. E., Clarke, D., v.Dijk, M., and Devadas, S., "Caches and Hash Trees for Efficient Memory Integrity Verification", International Symposium on High Performance Computer Architecture, Feb. 2003.

[39] Ghosh, A. K., O'Connor, T., and McGraw, G., "An automated approach for identifying potential vulnerabilities in software", 1998 IEEE Symposium on Security and Privacy, pp. 104-114.

[40] Goldreich, O., and Ostrovsky, R., "Software Protection and Simulation on Oblivious RAMs," Journal of the ACM, Vol.43, No.3, 1996.

[41] Goldreich, O., "Towards a Theory of Software Protection and Simulation by Oblivious RAMs," The 19th Annual ACM Symposium on Theory of Computing (STOC), 1987.

[42] Grunwald, D., Lindsay, D., and Zorn, B., "Static methods in hybrid branch prediction," PACT 1998. Pages: 222 – 229.

[43] Hangal, S., and Lam, M. S., "Tracking down software bugs using automatic anomaly detection", In Proceedings of Int. Conf. Software Engineering, May 2002.

[44] Hastings, R., and Joyce, B., "Purify: Fast detection of memory leaks and access errors", In Proceedings of the Winter USENIX Conference, 1992.

[45] Horne, B., Matheson, L., Sheehan, C., and Tarjan, R., "Dynamic self-checking techniques for improved tamper resistance", In Proc. 1st ACM Workshop on Digital Rights Management (DRM 2001), volume 2320 of Lecture Notes in Computer Science, pages 141–159. Springer-Verlag, 2002.

[46] Householder, A., Houle, K., and Dougherty, C., "Computer Attack Trends Challenge Internet Security", IEEE security and Privacy, Apr. 2002.

[47] Huang, A., "Keeping Secrets in Hardware: the Microsoft Xbox (TM) Case Study," MIT TR. AIM-2002-008, May 26, 2002.

[48] Intel LaGrande Technology, http://www.intel.com/technology/security/ (Accessed July 18, 2006).

[49] Intel Corporation, "Intel Random Number Generator," 1999.

[50] Jones R., and Kelly. P., "Bounds checking for C", http://www-ala.doc.ic.ac.uk/~phjk/BoundsChecking.html (Accessed July 18, 2006).

[51] Kanellos, M., "Intel's accidental revolution", http://news.com.com/2009-1001-275806.html (Accessed July 17, 2006).

[52] Kelsey, J., Schneier, B., Wagner, D., and Hall, C., "Side channel cryptanalysis of product ciphers", European Symposium on Research in Computer Security, Sep. 1998.

[53] Kennedy, K., and Allen, J. R., "Optimizing compilers for modern architectures: a dependence-based approach," Morgan Kaufmann Publishers Inc, 2002.

[54] Knuth, D. E., "Seminumerical Algorithms," The Art of Computer Programming, Vol. 3, Addison Wesley 1981.

[55] Kocher, P. C., "Timing attacks on implementations of Die-Hellman, RSA, DSS, and other systems," International Cryptology Conference, 1996.

[56] Kocher, P., Jaffe J., and Jun, B., “Differential Power Analysis”, International Cryptology Conference, 1999.

[57] Koo, R., and Toueg, S., “Checkpointing and rollback recovery for distributed systems,” IEEE Trans. Software Eng., Vol. SE-13, No. 1, pp. 23–31, Jan. 1987.

[58] Koopman, P., "Embedded System Security," IEEE Computer, July 2004.

[59] Kosoresow, A., and Hofmeyr, S., “Intrusion Detection via System Call Traces,” IEEE Software, vol. 14, pp. 24-42, 1997.

[60] Krügel, C., Mutz, D., Valeur, F., and Vigna, G., “On the Detection of Anomalous System Call Arguments”, In Proceedings of ESORICS 2003, pages 326-343, Norway, 2003.

[61] Kuhn, M. G., “Cipher Instruction Search Attack on the Bus-Encryption Security Microcontroller DS5002FP,” IEEE Transaction on Computers, Vol.47, No.10, pp.1153-1157, 1998.

[62] Lampson, B., Abadi, M., Burrows, M., and Wobber, E., “Authentication in distributed systems: Theory and practice”, ACM Transactions on Computer Systems, volume 10, issue 4, Nov. 1992, pp 265-310.

[63] Lee, R. B., Karig, D. K., McGregor, J. P., and Shi, Z., “Enlisting hardware architecture to thwart malicious code injection,” In Proceedings of the International Conference on Security in Pervasive Computing (SPC-2003), March 2003.

[64] Lie, D., Thekkath, C., Mitchell, M., Lincoln, P., Boneh, D., Mitchell, J., and Horowitz, M., “Architectural Support for Copy and Tamper Resistant Software,” International Conference on Architectural Support for Programming Languages and Operating Systems, Nov. 2000.

[65] Linn, C., and Debray, S., “Obfuscation of Executable Code to Improve Resistance to Static Disassembly”, In Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS), pages 290--299, October 2003.

[66] Liskov, B., Ghemawat, S., Gruber, R., Johnson, P., and Shrira, L., “Replication in the Harp File System,” In ACM Symposium on Operating System Principles, 1991.

[67] Lu, C., Zhang, T., Shi, W., and Lee, H. S., " M-TREE:A High Efficiency Security Architecture For Protecting Integrity and Privacy of Software", Journal of Parallel and Distributed Computing for a special issue on Security in Grid and Distributed Systems, 2006.

[68] Mach-SUIF Backend Compiler, "The Machine-SUIF 2.1 compiler documentation set", Harvard University, Sep. 2000, http://ececs.harvard.edu/hube/research/machsuif.html (Accessed July 18, 2006).

[69] Malkhi, D., and Reiter, M., "Secure and Scalable Replication in Phalanx," In IEEE Symposium on Reliable Distributed Systems, 1998.

[70] Mambo, M., Murayama T., and Okamato, E., "A tentative approach to constructing tamper-resistant software", In Proceedings of the 1997 Workshop on New Security Paradigms, Langdale, Cumbria, United Kingdom, September 23 - 26, 1997.

[71] Marlin, S., "Security Breach Exposes Data On Millions Of Payment Cards", Information Week, June 17, 2005.

[72] McClure, C., "Software Reuse Planning by Way of Domain Analysis", Technical Paper, Extended Intelligence Inc.

[73] McGregor, J. P., Karig, D. K., Shi, Z., and Lee, R. B., "A processor architecture defense against buffer overflow attacks," In Proceedings of IEEE International Conference on Information Technology: Research and Education (ITRE 2003), pages 243–250, August 2003.

[74] Melanson M., "Deobfuscating Obfuscated Code with Retroguard", available at http://www.multimedia.cx/pre/re-retroguard.html (Accessed July 18, 2006).

[75] Merkle, R. C, "Protocols for public key cryptography", In IEEE Symposium. On Security and Privacy, pages 122–134, 1980.

[76] Michael, C., and Ghosh, A., "Using Finite Automate to Mine Execution Data for Intrusion Detection: A preliminary Report", RAID 2000.

[77] Min, S. L., and Choi, J. D., "An efficient cache-based access anomaly detection scheme", In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 1991.

[78] Mueller, F., and Whalley, D. B., “Avoiding Unconditional Jumps by Code Replication,” by in the Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation, June 1992, pages 322-330.

[79] Next-Generation Secure Computing Base, http://www.microsoft.com/resources/ngscb/default.mspx (Accessed July 18, 2006).

[80] Nickerson, J. R., Chow, S. T., and Johnson, H. J., “Tamper Resistant Software: Extending Trust into a Hostile Environment”, Multimedia and Security Workshop at ACM Multimedia 2001, Ottawa, October 5, 2001.

[81] Oki, B., and Liskov, B., “Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems”, In ACM Symposium on Principles of Distributed Computing, 1988.

[82] One, A., "Smashing the Stack for Fun and Profit", Phrack volume 7, issue 49.

[83] Openwall project, http://www.openwall.com (Accessed July 18, 2006).

[84] Osvik, D. A., Shamir, A., and Tromer, E., “Cache Attacks and Countermeasures: The Case of AES,” The Cryptographers' Track at the RSA Conference 2006, February 13-17, 2006.

[85] Paek, Y., Hoeflinger, J., and Padua, D., “Efficient and Precise Array Access Analysis”, ACM Transactions on Programming Languages and Systems (TOPLAS). Vol. 24, Issue 1, pp. 65-109, January 2002.

[86] Pan, D. Z., and Linton, M. A., “Supporting reverse execution of parallel programs,” In Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, January 1989.

[87] Perry, D. C., “Microsoft's Xbox 360 vs. Sony's PlayStation 3”, http://xbox360.ign.com/articles/617/617951p1.html (Accessed July 18, 2006).

[88] Pugh, W., “A Practical Algorithm for Exact Array Dependence Analysis,” Communications of the ACM, 35(8), August 1992.

[89] Ruwase, O., and Lam, M. S., “A practical dynamic buffer overflow detector”, In

Proceedings of the 11th Network and Distributed System Security Symposium, February 2004.

[90] Schneider, F., "Implementing Fault-Tolerant Services Using The State Machine Approach: A Tutorial," ACM Computing Surveys, 22(4), 1990.

[91] Scut, "Exploiting format string vulnerabilities". TESO Security Group, Sep. 2001.

[92] Sekar, R., Bendre, M., Dhurjati, D., and Bollineni, P., "A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors," In Proceedings of the 2001 IEEE Symposium on Security and Privacy, 2001.

[93] Shamir, A., "How to Share a Secret", Communications of the ACM, 22(11): 612-613, November 1979.

[94] Sherwood, T., Perelman, E., Hamerly, G., and Calder, B., "Automatically Characterizing Large Scale Program Behavior," International Conference on Architectural Support for Programming Languages and Operating Systems Oct. 2002.

[95] Shi, W., Lee, H. S., Ghosh, M., Lu, C., and Boldyreva, A., "High Efficiency Counter Mode Security Architecture via Prediction and Precomputation." In the Proceedings of the 32nd International Symposium on Computer Architecture, pp.14-24, Madison, Wisconsin, June 2005.

[96] Shi, W., Lee, H. S., Ghosh, M., and Lu, C., "Architectural Support for High Speed Protection of Memory Integrity and Confidentiality in Multiprocessor Systems," In Proceedings of the International Conference on Parallel Architecture and Compilation Techniques, pp.123-134, September 2004.

[97] Shi, W., Lee, H. S., Gu, G., Ghosh, M., Falk, L, and Mudge, T. N., "Intrusion Tolerant and Self-Recoverable Network Service System Using Security Enhanced Chip Multiprocessors," In the Proceedings of the 2nd International Conference on Autonomic Computing, pp.263-273, Seattle, Washington, June 2005.

[98] Smirnov, A., and Chiueh, T., "DIRA: Automatic Detection, Identification, and Repair of Control-Hijacking Attacks," In Proceedings of NDSS'05, San Diego, CA, February 2005.

[99] Srinivasan, S., Kandula, S., Andrews, C., and Zhou, Y., “Flashback: A lightweight extension for rollback and deterministic replay for software debugging”, In Proceedings of USENIX Annual Technical Conference, June 2000.

[100] Stanford SUIF Compiler Infrastructure, “The SUIF 2 compiler documentation set”, Stanford University, Sep.2000. http://suif.stanford.edu/suif/index.html (Accessed July 18, 2006).

[101] Suh, G. E., Clarke, D., Gassend, B., v.Dijk, M., and Devadas, S., “AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing,” International Conference on Supercomputing, Jun. 2003.

[102] Suh, G. E., Clarke, D., Gassend, B., v.Dijk, M., and Devadas, S., "Efficient Memory Integrity Verification and Encryption for Secure Processors", International Symposium on Microarchitecture, Dec. 2003.

[103] Suh, G. E., Lee, W., and Devadas, S., “Secure Program Execution via Dynamic Information Flow Tracking”, ASPLOS 2004.

[104] Team, P., “Non-executable pages design and implementation”, http://pax.grsecurity.net/docs/noexec.txt (Accessed July 18, 2006).

[105] TPM vendors, http://www.tonymcfadden.net/tpmvendors.html (Accessed July 18, 2006).

[106] Triolet, R., Feautrier, P., and Irigoin, F., “Direct Parallelism of Call Statements,” ACM SIGPLAN Symposium on Compiler Construction, pages 176–185, 1986.

[107] Trusted Computing Group, https://www.trustedcomputinggroup.org/home (Accessed July 18, 2006).

[108] Tuck, N., Calder B., and Varghese, G., “Hardware and Binary Modification Support for Code Pointer Protection from Buffer Overflow”, 37th International Symposium on Microarchitecture, Dec. 2004.

[109] Tygar, J. D., and Yee B., “Dyad: A system for using physically secure coprocessors”, Technical report, Carnegie Mellon University, May 1991. CMU-CS-91-140R.

[110] Ullman, J. R., “An Algorithm for subgraph Isomorphism,” Journal of the ACM, Vol.23, pp.31-42, 1976.

[111] Vendicator, “StackShield GCC compiler patch”, http://www.angelfire.com/sk/stackshield (Accessed July 18, 2006).

[112] Verton, D., “Hacking syndicates threaten banking”, http://www.computerworld.com/securitytopics/security/cybercrime/story/0,10801,75584,00.html (Accessed July 17, 2006).

[113] VFLib Graph Matching Library, http://amalfi.dis.unina.it/graph/db/vflib-2.0/doc/vflib-1.html (Accessed July 18, 2006).

[114] Vlaovic, S., and Davidson, E. S., “TAXI: Trace Analysis for X86 Interpretation”, In Proc. Of 2002 IEEE International Conference on Computer Design.

[115] Wagner, D., and Dean, D., “Intrusion Detection via Static Analysis,” In Proceedings of the 2001 IEEE Symposium on Security and Privacy, 2001.

[116] Wang, Y. M., Huang, Y., Vo, K. P., Chung, P. Y., and Kintala, C., “Checkpointing and its applications,” In 25th International Symposium on Fault-Tolerant Computing, pages 22--31, Pasadena, CA, June 1995.

[117] Wikipedia, “Bandwidth Assessment of CELL Processor”, http://en.wikipedia.org/wiki/Cell_(microprocessor)#Bandwidth_Assessment (Accessed July 18, 2006).

[118] Wilander, J., and Kamkar. M., “A comparison of publicly available tools for dynamic buffer overflow prevention,” In 10th NDSSS, 2003.

[119] Wilson, R., and Lam, M., “Efficient context-sensitive pointer analysis for C programs”, In Proceedings of the SIGPLAN '95 Conference on Program Language Design and Implementation, La Jolla, CA, June 1995.

[120] Witchel, E., Cates, J., and Asanovic, K., “Mondrian Memory Protection,” Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X) , San Jose, CA, October 2002.

[121] Xu, J., Ning, P., Kil, C., Zhai, Y., and Bookholt, C., “Automatic Diagnosis and Response to Memory Corruption Vulnerabilities,” In Proceedings of 12th ACM Conference on Computer and Communications Security (CCS), Nov 2005.

[122] Xu, M., Bodik, R., and Hill, M. D., “A "flight data recorder" for enabling full-system multiprocessor deterministic replay,” In Proceedings of the 30th Annual international Symposium on Computer Architecture, pp. 122-135, San Diego, California, June 09 - 11, 2003.

[123] Yan, C., Englender, D., Prvulovic, M., Rogers, B., and Solihin, Y., “Improving Cost, Performance, and Security of Memory Encryption and Authentication,” Proceedings of the 33rd Annual International Symposium on Computer Architecture, Jun. 2006.

[124] Yang, J., Zhang, Y., and Gao, L., “Fast Secure Processor for Inhibiting Software Piracy and Tampering,” International Symposium on Microarchitecture, Dec. 2003.

[125] Zhang, T., Pande, S., and Valverde, A., “Tamper-resistant Whole Program Partitioning”, International Conference on Languages, Compilers, and Tools for Embedded Systems, Jun. 2003.

[126] Zhang, T., Pande, S., Santos, A. D., Bruecklmayer, F. J., “Leakage-proof Program Partitioning,” International Conference on Compiler, Architecture and Synthesis for Embedded Systems, Oct. 2002.

[127] Zhang, T., Zhuang, X., and Pande, S., “Building Intrusion-Tolerant Secure Software”, In Proceedings of 3rd ACM/IEEE International Symposium on Code Generation and Optimization (CGO 2005), San Francisco, CA, March, 2005, pp. 255-266.

[128] Zhang, T., Zhuang, X., Pande, S., and Lee, W., "Anomalous Path Detection with Hardware Support", International Conference on Compilers, Architectures and Synthesis for Embedded Processors (CASES), San Francisco, CA, Sep. 2005.

[129] Zhang, T., Zhuang, X., Pande, S., and Lee, W., “Hardware Supported Anomaly Detection: down to the Control Flow Level,” Technical Report GIT-CERCS-04-11.

[130] Zhang, Y., and Gupta, R., “Timestamped Whole Program Path Representation and its Applications”, PLDI 2001.

[131] Zhou, P., Liu, W., Long, F., Lu, S., Qin, F., Zhou, Y., Midkiff, S., and Torrellas, J., “AccMon: Automatically Detecting Memory-Related Bugs via Program Counter-based Invariants,” The Proceedings of 37th Annual IEEE/ACM International Symposium on Micro-architecture (Micro'04), December, 2004.

[132] Zhou, P., Qin, F., Liu, W., Zhou, Y., and Torrellas, J., “iWatcher: Efficient architectural support for software debugging”, In Proceedings of the 31st Annual International Symposium on Computer Architecture, 2004.

[133] Zhuang, X., Zhang, T., Pande, S., "HIDE: An Infrastructure for Efficiently Protecting Information Leakage on the Address Bus", International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Boston, MA., Oct 2004.