# Discovering New Vulnerabilities In Computer Systems

Zhenyu Wu

Chengdu, Sichuan, China

Master of Science, The College of William and Mary, 2007
Bachelor of Science, Denison University, 2005

A Dissertation presented to the Graduate Faculty
of the College of William and Mary in Candidacy for the Degree of
Doctor of Philosophy

Department of Computer Science

The College of William and Mary
May 2012

Copyright © 2012 Zhenyu Wu
All Rights Reserved

# APPROVAL PAGE

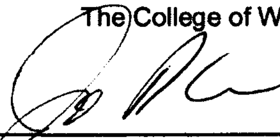
This Dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

Zhenyu Wu

Approved by the Committee, April 2012

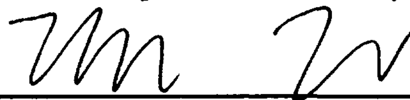
Committee Chair
Associate Prof. Haining Wang, Computer Science
The College of William and Mary



Associate Professor Phil Kearns, Computer Science
The College of William and Mary

Associate Prof. Weizhen Mao, Computer Science
The College of William and Mary

Associate Professor Qun Li, Computer Science
The College of William and Mary



Dr. Zhichun Li
NEC Laboratories America Inc.

# ABSTRACT PAGE

Vulnerability research plays a key role in preventing and defending against malicious computer system exploitations. Driven by a multi-billion dollar underground economy, cyber criminals today tirelessly launch malicious exploitations, threatening every aspect of daily computing. To effectively protect computer systems from devastation, it is imperative to discover and mitigate vulnerabilities before they fall into the offensive parties' hands. This dissertation is dedicated to the research and discovery of new design and deployment vulnerabilities in three very different types of computer systems.

The first vulnerability is found in the automatic malicious binary (malware) detection system. Binary analysis, a central piece of technology for malware detection, are divided into two classes, static analysis and dynamic analysis. State-of-the-art detection systems employ both classes of analyses to complement each other's strengths and weaknesses for improved detection results. However, we found that the commonly seen design patterns may suffer from evasion attacks. We demonstrate attacks on the vulnerabilities by designing and implementing a novel binary obfuscation technique.

The second vulnerability is located in the design of server system power management. Technological advancements have improved server system power efficiency and facilitated energy proportional computing. However, the change of power profile makes the power consumption subjected to unaudited influences of remote parties, leaving the server systems vulnerable to energy-targeted malicious exploit. We demonstrate an energy abusing attack on a standalone open Web server, measure the extent of the damage, and present a preliminary defense strategy.

The third vulnerability is discovered in the application of server virtualization technologies. Server virtualization greatly benefits today's data centers and brings pervasive cloud computing a step closer to the general public. However, the practice of physical co-hosting virtual machines with different security privileges risks introducing covert channels that seriously threaten the information security in the cloud. We study the construction of high-bandwidth covert channels via the memory sub-system, and show a practical exploit of cross-virtual-machine covert channels on virtualized x86 platforms.

# Table of Contents

**Dedication** iv

**Acknowledgements** v

**List of Tables** vi

**List of Figures** vii

**List of Algorithms** ix

**1 Introduction** 1

1.1 Malware Detection and Classification . . . . . 4

1.2 Server System Power Management . . . . . 6

1.3 Virtualization and Privacy . . . . . 8

**2 Vulnerability in Static Binary Analysis** 11

2.1 Motivation . . . . . 12

2.2 Related Work . . . . . 14

2.3 Background . . . . . 17

2.3.1 Regular Mimic Function . . . . . 18

2.3.2 High-order Mimic Function . . . . . 18

2.3.3 The Power of High-order Mimic Function . . . . . 20

2.3.4 Enhancements to High-order Mimic Function . . . . . 21

2.4 Mimimorphic Engine Design . . . . . 22

2.4.1 Digesting . . . . . 22

2.4.2 Encoding . . . . . 24

2.4.3 Decoding . . . . . 26

2.4.4 Design Issues . . . . . 28

2.5 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . 29
2.6 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . 30
2.6.1 Statistical Tests . . . . . . . . . . . . . . . . . . . . . . 31
2.6.2 Semantic Analysis Test . . . . . . . . . . . . . . . . . . . 34
2.7 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . 38
2.7.1 Artifact Generation . . . . . . . . . . . . . . . . . . . . 38
2.7.2 Robustness . . . . . . . . . . . . . . . . . . . . . . . . 38
2.7.3 Application Constraints . . . . . . . . . . . . . . . . . . 39
2.7.4 Decoder Detection . . . . . . . . . . . . . . . . . . . . 40
2.8 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . 41

**3 Vulnerability in Server Power Management 42**

3.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . 43
3.2 Background . . . . . . . . . . . . . . . . . . . . . . . . . . 44
3.2.1 Power Distribution . . . . . . . . . . . . . . . . . . . . 45
3.2.2 Energy Proportionality . . . . . . . . . . . . . . . . . . 46
3.2.3 Real Server Measurements . . . . . . . . . . . . . . . . . 47
3.2.4 Threat of Energy Attacks . . . . . . . . . . . . . . . . . 48
3.2.5 Feature of Energy Attacks . . . . . . . . . . . . . . . . . 49
3.3 Energy Attack on Server Systems . . . . . . . . . . . . . . . 51
3.3.1 Scenario Selection . . . . . . . . . . . . . . . . . . . . . 51
3.3.2 Case Study: Wikipedia Mirror Server . . . . . . . . . . . . 52
3.4 Attack Evaluation . . . . . . . . . . . . . . . . . . . . . . . 55
3.4.1 Configuration and Setup . . . . . . . . . . . . . . . . . . 55
3.4.2 Workload—Response Time Profile . . . . . . . . . . . . . 56
3.4.3 Attack Measurements . . . . . . . . . . . . . . . . . . . 57
3.4.4 Damage Assessment . . . . . . . . . . . . . . . . . . . . 58
3.5 Defending Against Energy Attacks . . . . . . . . . . . . . . 59
3.5.1 Defense Challenges . . . . . . . . . . . . . . . . . . . . 60
3.5.2 Energy-Aware Programming . . . . . . . . . . . . . . . . 61
3.5.3 Defense Experiment . . . . . . . . . . . . . . . . . . . . 66
3.6 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . 68
3.6.1 Attack Variations . . . . . . . . . . . . . . . . . . . . . 68
3.6.2 Attack Applicability . . . . . . . . . . . . . . . . . . . . 69

3.6.3 Limitation of Defense . . . . . . . . . . . . . . . . . . . . . . 69
3.7 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
3.8 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72

**4 Vulnerability in Virtualizated Public Cloud** **74**
4.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
4.2 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
4.3 Struggles of the Classic Cache Channels . . . . . . . . . . . . . . 79
4.3.1 Addressing Uncertainty . . . . . . . . . . . . . . . . . . . . 80
4.3.2 Scheduling Uncertainty . . . . . . . . . . . . . . . . . . . . 84
4.3.3 Cache Physical Limitation . . . . . . . . . . . . . . . . . . . 85
4.4 Covert Channel in the Hyper-Space . . . . . . . . . . . . . . . . . 86
4.4.1 Redesigning Data Transmission . . . . . . . . . . . . . . . . 86
4.4.2 (Re)Discovering the Memory Bus Channel . . . . . . . . . . . 89
4.4.3 Enabling Reliable Communication . . . . . . . . . . . . . . . 94
4.5 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 99
4.5.1 In-house Experiments . . . . . . . . . . . . . . . . . . . . . 99
4.5.2 Amazon EC2 Experiments . . . . . . . . . . . . . . . . . . . 101
4.6 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104
4.6.1 Damage Assessment . . . . . . . . . . . . . . . . . . . . . . 104
4.6.2 Mitigation Techniques . . . . . . . . . . . . . . . . . . . . . 105
4.7 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 107

**5 Conclusion** **109**
5.1 Contribution to Knowledge . . . . . . . . . . . . . . . . . . . . . 110
5.2 Future Research . . . . . . . . . . . . . . . . . . . . . . . . . . 111
5.2.1 Data Center and Cloud Computing . . . . . . . . . . . . . . . 111
5.2.2 Mobile System Security . . . . . . . . . . . . . . . . . . . . 113

**References** **115**

**Vita** **126**

*To my parents,*
*for their unending love and encouragement;*
*And to Vera,*
*for her companion and support.*

# ACKNOWLEDGEMENTS

The work of this dissertation would not have been possible without the guidance, encouragements and support from my committee members, professors, colleagues, friends, and family.

I owe my deepest gratitude to my advisor, Dr. Haining Wang, who fostered my interests in research, and provided me with invaluable guidance and support in every step of my graduate study.

I would like to express my sincere gratitude to the professors in the Computer Science department, for their excellent and challenging coursework. I am thankful to Dr. Phil Kearns, for providing research insights and serving on my dissertation committee; Dr. Qun Li and Dr. Weizhen Mao, for their encouragements and support as my committee members; Dr. Evgenia Smirni and Dr. Virginia Torczon, for their kindness and considerations as graduate director and department chair.

I am grateful to my former colleagues, who shared with me their research passion and inspirations. Dr. Mengjun Xie and Dr. Steven Gianvecchio, with whom I have really enjoyed years of collaborations and co-authorship; Dr. Chuan Yue, who encouraged my with cheerful and supporting words. I am especially thankful to Dr. Mengjun Xie, who has also provided precious mentor-ship and assistance along most part of my graduate study.

I am also grateful to Dr. Zhichun Li, for his mentor-ship and valuable input as my external committee member; Dr. Yueping Zhang, for his mentor-ship and encouragements on my research.

I am thankful to the staff of the Computer Science Department, and especially, to Ms. Vanessa Godwin and Ms. Jacqulyn Johnson, for their persistent and unfailing administrative support.

I am also thankful to my colleagues, Zhang Xu, for his timely assistance on my research project for the final chapter; Zi Chu, for his unselfish support of sharing office desk with me during my final semester.

Finally, I want to give my heartfelt gratitude to my family, whose love and encouragement allowed me to finish this journey. I hope that this work makes you proud.

# List of Tables


2.1 Mimicry English Text . . . . . . . . . . 20

2.2 Mimic Function Runtime Analysis . . . . . . . . . . 20

2.3 Mimimorphic Terms . . . . . . . . . . 22

2.4 Kolmogorov-Smirnov Results . . . . . . . . . . 32

2.5 Byte Entropy Results . . . . . . . . . . 33

2.6 Bad Fingerprints for $M_7$ and $M_8$ Instances . . . . . . . . . . 35

2.7 Shared/Good/Bad Fingerprints of All $M_7$ and $M_8$ Instances . . . . . 37

3.1 Server System Configurations . . . . . . . . . . 47

3.2 Percentage of Power Increases due to Attack . . . . . . . . . . 59

4.1 Experimental System Configurations . . . . . . . . . . 82

4.2 Cache Latencies vs. Access Pattern Lengths . . . . . . . . . . 82

4.3 Various Invalid Scheduling Patterns . . . . . . . . . . 85

# List of Figures


1.1 The Progression of a Vulnerability . . . . . . . . . . . . . . . . . . . . 2
1.2 Vulnerable Malware Detection System Designs . . . . . . . . . . . . . 5

2.1 The Prefixed Symbol Tables . . . . . . . . . . . . . . . . . . . . . . . 19
2.2 The Prefixed Huffman Forest . . . . . . . . . . . . . . . . . . . . . . 19
2.3 The Digesting Phase . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
2.4 The `CommonInst` Structure . . . . . . . . . . . . . . . . . . . . . . . 23
2.5 The Instruction Digest Table . . . . . . . . . . . . . . . . . . . . . . . 23
2.6 The Encoding Phase . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
2.7 An Encoding Example . . . . . . . . . . . . . . . . . . . . . . . . . . 26
2.8 The Decoding Phase . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
2.9 An Decoding Example . . . . . . . . . . . . . . . . . . . . . . . . . . 28
2.10 Kolmogorov-Smirnov Results . . . . . . . . . . . . . . . . . . . . . . 32
2.11 Byte Entropy Results . . . . . . . . . . . . . . . . . . . . . . . . . . 33
2.12 A Sample of $M_7$ Control Flow Graph . . . . . . . . . . . . . . . . . . 34
2.13 Bad Fingerprints in Collections of $M_7$ and $M_8$ Instances . . . . . . . 34
2.14 Shared Bad Fingerprints in Collections of $M_7$ and $M_8$ Instances . . . 36

3.1 System Power Consumptions . . . . . . . . . . . . . . . . . . . . . . . 47
3.2 Power Consumption and Response Time vs. Caching Scenarios . . . . 53
3.3 Power Consumption and Response Time Profiles of Victim Server . . 56
3.4 Effects of Attack Under Selective Benign Workload . . . . . . . . . . 57
3.5 Collective Effects of Attack Under Different Benign Workload . . . . 58
3.6 Example Component Flow Chart . . . . . . . . . . . . . . . . . . . . 62
3.7 System Power Measurement and Attack Detection . . . . . . . . . . . 64
3.8 Maintaining Client Power History for Defensive Throttling . . . . . . 64
3.9 MediaWiki Flow Chart . . . . . . . . . . . . . . . . . . . . . . . . . . 66
3.10 Defense Measurement Results . . . . . . . . . . . . . . . . . . . . . . 66

4.1 Memory Address to Cache Line Mappings for L1 and L2 Caches . . . 81
4.2 Memory Address to L2 Cache Line Mapping with Virtualization . . . 83
4.3 Timing-based Cache Channel Bandwidth Test . . . . . . . . . . . . . . 88
4.4 Timing-based Memory Bus Channel Bandwidth Tests . . . . . . . . . 94
4.5 Memory Bus Channel Quality Sample on EC2 . . . . . . . . . . . . . . 95
4.6 Effects of Non-participating Workload on Bandwidth and Error Rate 100
4.7 Memory Bus Channel Capacities of EC2 . . . . . . . . . . . . . . . . . 102
4.8 Reliable Transmission with Adaptive Rates . . . . . . . . . . . . . . . . 103

# List of Algorithms


2.1 Mimimorphic Encoding . . . . . . . . . . . . . . . . . . . . . . . . . . 25
2.2 Mimimorphic Decoding . . . . . . . . . . . . . . . . . . . . . . . . . . 27
4.1 Classic Cache Channel Protocol . . . . . . . . . . . . . . . . . . . . . . 80
4.2 Timing-based Cache Channel Protocol . . . . . . . . . . . . . . . . . . 87
4.3 Timing-based Memory Bus Channel Protocol . . . . . . . . . . . . . . 93
4.4 Reliable Timing-based Memory Bus Channel Protocol . . . . . . . . . . 98

Chapter 1

# Introduction

**Computer system security** is *not* just a static term—it also represents an ongoing and unending warfare, with battles taking place in the cyberspace at any moment. Started as vandalism or for personal glories in the early days of computing, the act of computer system exploitations quickly evolves into organized cyber-crime as computers and network spread through the modern society. Nowadays, driven by a multi-billion dollar underground economy [7, 29], malicious computer system exploitations are sub-divided into specialized categories, such as system hijacking (e.g., bot farming), data stealing (e.g., theft of personal or business information), spamming, and even digital terrorism (e.g., denial-of-service blackmailing), threatening every aspect of our daily computing. Fortunately, defensive technologies and security research have also made comparable improvements and corresponding specializations, protecting the cyberspace infrastructure as well as most of our everyday computing activities from the harms of malicious exploits. However, enticed by strong financial incentives, attackers are constantly probing and developing new offensive technologies, and the war will continue on for the foreseeable future.

Central to both offensive and defensive technologies, vulnerability research plays a key role in crafting malicious exploitations as well as developing corresponding pro-

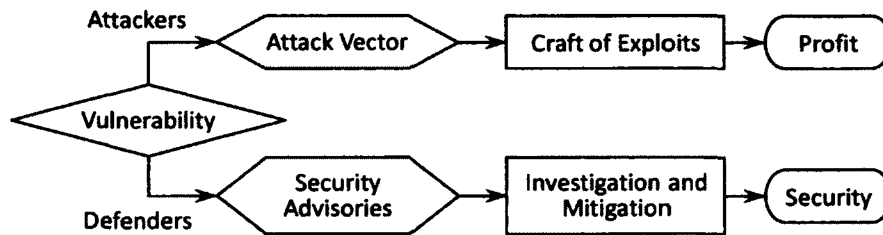



Figure 1.1: The Progression of a Vulnerability

tections. As show in Figure 1.1, a vulnerability may go through different stages of transformations on both the offense side and the defense side. On the offense side, the attackers first turn a vulnerability into a workable attack vector. Then they proceed to create exploits that leverage the attack vectors to achieve their malicious goals. And finally, the attackers obtain profits through successfully launched exploits. On the defense side, in response to a vulnerability discovery, security advisories are first published to alert the system administrators, the vendors of the vulnerable product, as well as the security service providers. Then, the responsible parties investigate the vulnerability and develop countermeasures, such as security patches, workarounds, and exploit prevention techniques. And finally, with the deployment of countermeasures, the vulnerable system is secured from the attackers' exploit. In order to protect computer systems from the attackers' devastation, it is imperative for the defenders to reach the end on the progression chart first. And thus the timely discovery of vulnerabilities is critical to successful defenses.

Vulnerabilities of computer systems, by their causes of introduction, can be classified into the following three general categories:

1. **Vulnerable design:** Vulnerabilities can be introduced into a system at the design stage through various errors and failures, such as logic flaws, use of unwarranted assumptions, and failure to incorporate certain security aspects.

2. **Vulnerable implementation:** Even with a secure design, vulnerabilities can still enter a system at its implementation stage. Vulnerable implementations take forms of a wide spectrum of "bugs", ranging from accidental errors (e.g., typos) to bad practices (e.g., insufficient checking or normalization of user input), and to rogue features (i.e., unspecified or documented implementations).

3. **Vulnerable deployment:** Malpractice in system deployment, such as deploying a system out of its original designed context, can also introduce vulnerabilities. Deployment errors subject a system to unanticipated operation conditions, such as failed premises, malformed input, and ambiguous command interpretations, which may either directly lead to or increase the likelihood of vulnerability manifestation.

While the most commonly encountered vulnerabilities belong to the category of vulnerable implementation, those vulnerabilities typically have simple corrections that can be developed and deployed within a short time frame. The now pervasive on-line software update systems provided by major operating system and application vendors are good examples of mature industrial solutions that could rapidly respond to implementation vulnerability discoveries. Compared to implementation vulnerabilities, the other two categories of vulnerabilities, namely the design and deployment vulnerabilities, are less often discovered. However, those vulnerabilities are much more difficult to respond to and to mitigate in a timely manner. And thus design and deployment vulnerabilities pose more security hazard, if they were to be employed by the offensive parties.

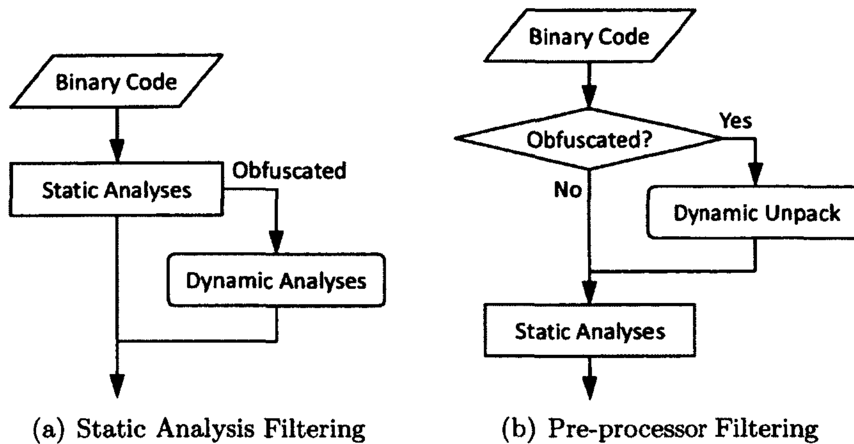
This dissertation is dedicated to the research and discovery of new design and deployment vulnerabilities in three very different types of computer systems that are or are becoming important parts of our computing infrastructure.

## 1.1 Malware Detection and Classification

Malicious software, known as "Malware", is a class of power weapons used by attackers to assault computer systems. It is commonly employed by attackers to hijack computer systems and steal information. Taking forms of worms, viruses, and Trojan horses, malware propagates on to a large number of personal computers as well as servers either by automated exploits of vulnerabilities, or by exploiting the human factors (e.g., curiosity, trust, management flaws, etc.).

The first line of defense against malware is automatic malicious binary detection and classification. With the prevalence of anti-malware software nowadays, binary executables are subjected to a number of detection scans during transportation and before execution. In addition, automatic malware analysis and classification systems are employed by security product vendors and researcher to facilitate quick responses to malware outbreaks. However, we have found a vulnerable design, buried deep in many automatic malicious binary detection and classification systems.

Automatic malicious binary detection works by extracting the characteristics, a.k.a. signatures, of an unknown binary executable, and comparing the results against a pool of signatures from known malware. And if enough similarities are found, the binary executable in question is asserted as malicious. As a central piece of the automatic malicious binary detection, binary analysis techniques can be coarsely divided into two classes—*static analysis* and *dynamic analysis*. Static analysis techniques analyze binary executables without executing them. Instead, a number of static metrics are derived from the binary data, ranging from statistical features such as byte entropy and n-gram distributions, to semantic features such as control flow and system call patterns. Dynamic analysis techniques, in contrast, perform analysis by executing the subject binary in a controlled fashion, such as emulation, sand-boxing or symbolic




(a) Static Analysis Filtering (b) Pre-processor Filtering

Figure 1.2: Vulnerable Malware Detection System Designs

execution, and monitoring the subject's run-time activity. Each class of techniques has its own strengths and weaknesses. Static analyses feature high throughput and low resource consumptions; but they suffer binary obfuscation evasion attacks, particularly, polymorphism and metamorphism. Dynamic analyses enjoy high detection accuracy and suffer less from binary obfuscation; however, they demand significantly longer process time and more computing resources than static analyses.

In order to increase both detection accuracy and performance, state-of-the-art automatic malicious binary detection and classification systems employ both static analyses and dynamic analyses. And these two classes of components are commonly seen organized in design patterns similar to those shown in Figure 1.2(a) or 1.2(b). The rational of using these designs is that dynamic analyses are too "expensive" to be placed on the critical path of the detection procedure. Instead, static analyses or static pre-processing can be employed to handle or filter out most unobfuscated binary executables, leaving only those particularly-difficult-to-determine binaries for dynamic analyses to process.

These designs suffer unwarranted assumption and flawed logic. In particular, the problematic assumption is that static analysis can always differentiate obfuscated

programs from unobfuscated ones. However, it is perceivable that some meticulously crafted binary transformations can produce obfuscated binaries that are indifferentiable from unobfuscated binary in terms of static features. Such kind of binary transformations invalidates the above assumptions and reveals a logic loophole within these designs—dynamic analyses, incorporated for the purpose of complementing static analyses' weaknesses against code obfuscation, could be effectively bypassed due to the very same weaknesses.

In Chapter 2, we present an exploit of this design vulnerability in automatic malicious binary detection and classification systems. We introduce mimimorphism, a novel binary obfuscation technique that camouflages malware binaries as legitimate executables. Inspired by a steganographic technique—the mimic functions, we create the mimimorphic transformation by augmenting a high-order mimic function with customized assembler and disassembler. Mimimorphic transformation is capable of encoding arbitrary data into "mimimorphic binaries"—pseudo-executable binaries that are indistinguishable from unobfuscated benign binary code in terms of statistical features such as byte entropy and frequency distributions, as well as semantic features such as control flow finger prints. As a result, mimimorphism defeats a range of static analysis techniques by obfuscating the malicious code and misguiding the analyses to assert the the obfuscated binary as obfuscated, allowing mimimorphic malware to evade the vulnerable detection systems.

## 1.2 Server System Power Management

Power management has become increasingly important for server systems nowadays. As the cost of commodity hardware decreases, energy cost has become a major factor in server system total cost of ownership (TCO). To address the increasing energy

concern and demand for power saving, the concept of energy proportional computing has been introduced.

Energy proportional computing aims to improve energy efficiency by making servers consume energy proportional to its workload. Numerous techniques have been developed to facilitate energy proportional computing, covering a variety of aspects, from low-level hardware features such as processor Dynamic Voltage and Frequency Scaling (DVFS) and hard disk spin-down, to high-level system-wise management schemes such as cluster load provisioning and virtual machine consolidation. While the server system power savings have been significantly improved thanks to these technological advancements, a new type of vulnerability, *energy targeted exploits*, has quietly emerged from behind the scene.

Historically, performance and security have been considered as primary metrics for server system evaluation and operation, while the power management has been largely down-played or even ignored. For this reason, server systems used to be power-inefficient and energy-disproportional, that is, a server consumes the same or similar amount of power regardless of the workload it processes. And for the very same reason, power management has never been considered as a security concept for server systems. However, the adaptation to energy proportional computing has changed the scope of server system security concept, and this change have not yet been paid attention to.

Switching power consumption profiles from "constant" to "workload-proportional" has introduced a new variable into the server system operation metrics, the power usage. And more significantly, while other operation variables, such as network bandwidth and security privileges, are either audited or access-restricted to system administrators, the newly introduced system power usage variable is largely unaudited, and its value can be heavily influenced by remote users (i.e., whoever submits workload to

the server) outside of the system administrators' control. The lack of incorporation of the power management factor in the security framework results in a serious design vulnerability, leaving current server systems completely open to (i.e., no means to detect or defend) energy targeted attacks.

In Chapter 3, we demonstrate a realistic energy attack on a standalone web server system, exploiting this server system power management vulnerability. By profiling request serving energy cost of an open Web service under different operation conditions, we identify an attack vector that an anonymous remote user can exploit to abuse the server's power consumption. Then, leveraging knowledge of human Web browsing behaviors, we proceed to design a stealthy attacking strategy, which ensures low attack traffic volume as well as statistically indistinguishable request timing signatures (i.e., request inter-arrival time) from those of the benign human users. We launch the energy attack against our testbed server, and systematically measure the extent of damage. We find out that this attack is able to significantly increase the power consumption of victim servers under typical workloads.

## 1.3 Virtualization and Privacy

Server virtualization are widely deployed in today's data centers. Providing the benefit of dynamic workload consolidation and simplified resource management, virtualization technologies greatly reduce data center operation cost, enabling low-cost access to utility based cloud computing services, and bringing pervasive cloud computing a step closer to the general public.

However, a major factor that impedes the adaptation of public cloud computing is the concern of privacy, or information security in general. The public cloud is a heavily commuted, dynamic, and possibly hostile environment, and thus private data

stored in the cloud are more susceptible to loss or leakage. Cloud vendors and users battle data leakage by orchestrating a variety of technologies, such as network isolation (such as VLAN and VPN), encryption, firewalls, traffic filtering, intrusion detection, etc. Despite the efforts being spend on information safeguarding, the potential risks of data leakage still loom the cloud, one of which is *the covert channels.*

Covert channels exploit imperfections in the isolation of shared resources between two unrelated entities, and enable communications between them via unintended channels, bypassing mandatory access controls placed on standard communication channels. Previous research have shown that on non-virtualized systems, covert channels can be constructed using a variety of media, such as file system objects, network interfaces, shared processor cache, etc. Although to date there is no known practical exploit of covert channels in the cloud, recent research have precautionarily pointed out the potential risk of physical co-residency on virtualized systems.

Co-hosting virtual machines (VMs) with different security privileges on the same physical hardware risks introducing deployment vulnerabilities. Server virtualization technologies meet the objective of computing consolidation by creating multiple logically separate, virtual computing platforms, multiplexing the shared underlying physical hardware. And the "logically-separate-but-physically-shared" design choice clearly indicates that server virtualization does not intend to provide computing environment identical to that of physically separated machines. As a result, it is inevitable for virtualized environment to have subtle but non-negligible differences from non-virtualized ones, especially in non-standardized aspects, such as covert channel security. However, few applications or security mechanisms today are prepared to accommodate these differences. Therefore, physically co-hosting VMs with different security privileges put systems (especially security systems) on the VMs out of their originally designed context, thereby introducing deployment vulnerabilities.

Based on our above assertion, in Chapter 4 we study the construction of high-bandwidth cross–VM channels on the virtualized x86 platform, focusing on the memory sub-system. We first analyze existing low-bandwidth cache covert channel techniques, and understand their inefficiencies and limitations in a virtualized environment. Then we perform an in-depth study of x86 processor cache and memory architecture, and design novel cache and memory bus covert channels that overcome the obstacles of existing techniques. We then conduct realistic experiments, and show that our covert channels can achieve high bandwidth, low cache footprint and reliable data transmission. Our study is the first among its kind to prove that the threat of covert channel attacks in the cloud is both realistic and practical.

# Chapter 2

# Vulnerability in Static Binary Analysis

Automatic malicious binary detection is the first line of defense against malicious software. With the prevalence of anti-malware software nowadays, a piece of binary code is subjected to a number of static analysis and detection scans during transportation and before execution. Consequently, binary obfuscation is critical for malware to succeed in propagation. The widely used code obfuscation techniques, such as polymorphism and metamorphism, focus on evading syntax based detection. However, statistic test and semantic analysis techniques have been developed to thwart their evasion attempts. More recent binary obfuscation techniques are divided in their purposes of attacking either statistical or semantic approach, but not both. In this chapter, we introduce mimimorphism, a novel binary obfuscation technique with the potential of evading both statistical and semantic detections. Mimimorphic malware uses instruction-syntax-aware high-order mimic functions to transform its binary into mimicry executables that exhibit high similarity to benign programs in terms of statistical properties and semantic characteristics. We design and implement a prototype mimimorphic engine on the Intel x86 platform, and prove its capability of evading statistical anomaly detections, using byte frequency distributions and entropy tests, as well as semantic analysis detection techniques, using control flow fingerprinting.

## 2.1 Motivation

To date, real-time malware detection largely relies on static binary analysis, due to its significant speed and resource consumption advantages over dynamic executable analysis [45, 48, 53, 64, 83]. Malware mainly evades static analysis detections through binary obfuscations, namely oligomorphism, polymorphism, and metamorphism [84]. Oligomorphism is used to evade byte sequence signature detections on the malware functional code. It utilizes simple operations such as XOR to scramble malware functional code before propagation, and decodes it while executing. Evolved from oligomorphism, polymorphism encodes malicious code by "packing" (i.e., compressing or encrypting), and then camouflages the "unpacker" (the decompression or decryption code) by using binary mutation techniques, such as instruction substitution and register remapping. Instead of packing program binaries, metamorphism generates different instruction combinations to represent the same functional part of a malicious program in its variants. The major techniques employed by metamorphism are binary-level mutations and meta-level transformations. A meta-level transformation first translates binary code into an intermediary representation called P-code, then manipulates the P-code, and finally composes new instances from the P-code. In this way, metamorphic malware can significantly shuffle its program contents and escape substring signature based detections.

Although the classic polymorphism and metamorphism enable malware to generate many binary instances with different byte patterns, they cannot effectively disguise the presence of malicious code in terms of statistical properties and program semantics. Compression and encryption in polymorphism usually change the statistical characteristics of a program in such a dramatic manner that the malware program can be easily classified as suspicious and be further scrutinized. Exploiting this prop-

erty, byte-frequency based detection methods such as Anagram [93] and PAYL [94], and entropy based detection methods such as Bintropy [54] have been proposed to uncover polymorphic malware. Additionally, because compressed or encrypted code segments are no longer executable, they can be easily identified by advanced disassemblers [51]. Such a filtering strategy has been applied to extract the unpackers of polymorphic worms [50]. Meanwhile, metamorphism preserves semantic equivalences between different variants. This property is thus exploited by semantic analysis techniques. For example, MetaAware [104] detects variants of metamorphic malware based on analysis of system call and library call instructions.

With the advancements in detection, state-of-the-art evasion techniques are moving beyond polymorphism and metamorphism. Targeting byte-frequency-based static anomaly detection, polymorphic blending attacks [27] manipulate the statistics of malware through byte padding and substitution. Designed to thwart semantic analysis, [60] mutates a program's control flow by transforming constants into an NP-complete problem. However, while flying under the radar of their targeted detection methods, these evasion techniques are ineffective against other analysis techniques. Polymorphic blending can hardly escape semantics based detection because byte padding and substitution destroy the executable semantics, making it easy to single out the unpacker code. Similarly, encoding control flow with opaque constants induces identifiable syntax patterns, which can be used as signatures.

We introduce mimimorphism, a new approach to binary obfuscation. Mimimorphism is unique in that instead of targeting at a specific detection approach, it aims to camouflage malware binaries as legitimate executables and thus significantly increases malware's resistance against a range of static statistical and semantic analyses. Leveraging a steganographic technique called the mimic function, mimimorphism transforms a malware executable into "mimic-binaries" that resemble ordinary

benign programs. To achieve this goal, we augment a high-order mimic function with customized assembler and disassembler, creating an instruction-syntax-aware mimic function—the core of the mimimorphic engine. The mimimorphic engine captures the high-order instruction-level characteristics of a given set of benign programs, and encodes malicious binaries based on the captured characteristics. As a result, a mimimorphic binary acquires highly similar statistical properties and semantic structures to those of ordinary benign programs.

## 2.2 Related Work

Attackers increasingly employ polymorphic and metamorphic techniques [84] to disguise their attacks and evade intrusion detection systems. The core of these techniques is to change the appearance of malicious code. Although the bit patterns of polymorphic attacks are distinctly different, their malicious functions remain the same. A number of tools have been developed for generating polymorphic shellcode [20, 55] and polymorphic executables [43, 70, 103]. Since polymorphic malware and metamorphic malware are able to significantly transform their contents in propagation, as mentioned by Newsome *et al.* [64] and Crandall *et al.* [14], they can effectively circumvent the perimeter of the network intrusion detection systems that are based on contiguous byte string signatures [45, 48, 83].

A basic approach to detecting polymorphic worms is based on byte statistics, such as byte frequency [94] and byte entropy [54]. Wang *et al.* [94] developed a payload-based anomaly detector, PAYL, which profiles the byte distribution of packet payloads and detects the abnormal byte distributions of polymorphic worms. Lyda *et al.* [54] demonstrated that the byte entropy of executables can be used to effectively identify packed or encrypted malware. Tang *et al.* [86] introduced the position-aware

distribution signature (PADS), which is capable of detecting polymorphic worms by recording a byte frequency distribution for each position in the signature "string".

There are several advanced polymorphic attacks [20, 27] designed to evade detections based on byte statistics. Detristan *et al.* [20] built a polymorphic engine called CLET, which uses byte padding to approximately match the normal byte distribution. Fogla *et al.* [27] introduced the polymorphic blending attack that exploits byte substitution and byte padding to achieve a very close match to normal profiles. The polymorphic blending attack is effective in evading 1-gram and 2-gram PAYL, as well as other detection methods based on low-order byte distributions. However, the problem of generating optimal polymorphic blending attacks is shown to be NP-complete, and a near-optimal heuristic approach must be used.A drawback to these mimicry attacks, similar to basic polymorphic attacks, is that the encrypted regions do not contain valid instruction sequences, while the attack vector and decryption routines are still executable, making these regions easily differentiated.

To counter mimicry attacks, higher-order byte patterns have been used in recent detection methods [69, 93]. Wang *et al.* [93] presented a new anomaly detector called Anagram, which is capable of detecting a modified polymorphic blending attack [27]. Anagram employs a Bloom filter to reduce the computation and storage requirements for modeling higher-order n-grams, in particular, n-grams 2–9 are chosen for experiments. While higher-order n-grams tend to produce better signatures, their training costs are much higher. Perdisci *et al.* proposed a multi-classifier system [69]. It summarizes higher-order n-grams as pairs of non-consecutive bytes, reducing the dimensionality of fully modeling higher-order n-grams. A clustering algorithm, originally proposed for text classification, is also used to reduce the dimensionality. The experimental results demonstrate that the proposed detector is as robust against evasion as a hypothetical 7-gram PAYL.

A different approach for polymorphic worm detection is based on syntactic signatures composed of multiple invariant substrings. The rationale behind this approach is that invariant substrings such as protocol framing substrings and high-order bytes of overwritten addresses often occur in all variants of polymorphic malware. Polygraph [64] proposes three types of syntactic signatures and related automatic signature generation algorithms. Hamsa [53] shares a similar design principle and signature scheme with Polygraph, but is faster, more noise-tolerant and attack-resilient. Both Polygraph and Hamsa require innocuous and suspicious traffic pools for signature generation, and thus, are vulnerable to training attacks. Perdisci *et al.* [68] presented a noise injection attack, in which injecting just one fake anomalous flow per real worm flow can prevent Polygraph from generating an accurate worm signature. Similarly, Newsome *et al.* [65] stated that malicious training can cause problems even when all of the training data are correctly labeled, and demonstrated that this type of attacks in general can be effective against both Polygraph and Hamsa. Gundy *et al.* [32] developed a polymorphic engine for PHP code and a polymorphic PHP-based worm that is able to evade Polygraph and Hamsa. Venkataraman *et al.* [89] presented the fundamental limits on the accuracy of a class of pattern-extraction algorithms for signature-generation in an adversarial setting.

More recent research has begun to focus on semantic analysis methods that extract higher-level meaning from executables[13, 46, 50, 102, 104]. Christodorescu *et al.* [13] proposed a semantic-aware malware detection system, which essentially exploits the uniform behavior exhibited by the variants of the same malware. A program is classified as malicious if it contains a sequence of instructions exhibiting the behavior specified by a malware template. Kruegel *et al.* proposed a polymorphic worm detection scheme [50] by utilizing the structural information. Based on the facts that the decryption routines of polymorphic worms are usually executable and their control

flow graphs (CFG) are fairly stable across worm mutations, the proposed method employs the static analysis and comparison of binary's CFG for worm detection. As another semantic analysis method, MetaAware [104] detects metamorphic worms by matching call instruction patterns. A pattern usually comprises multiple sub-patterns, each constituting library and system call instructions with corresponding parameter setting instructions.

Due to the fundamental roles of control flow and data flow analyses in static analysis, Moser *et al.* [60] designed a binary obfuscation scheme based on the concept of opaque constants. They demonstrated that advanced semantics-based detections (such as model checking [46]) can be effectively thwarted by hiding key constants of the control flows using obfuscation transformations. Barak *et al.* [2] discussed the theoretical limits of program obfuscation. In particular, they proved that it is impossible to hide certain families of properties via function-preserving obfuscation.

The concept of mimicry attack has also been applied in dynamic host based intrusion detection systems (IDS). In the context of system call monitoring, a mimicry attack is defined as a sequence of malicious system calls that exploits flaws in the IDS program model [91], and is thus considered as legitimate. Traditionally, these attacks are manually constructed [85, 91], but recent research [28, 49, 66] has shown that they can be automatically developed.

## 2.3 Background

The idea of mimic functions was first introduced by Peter Wayner [98] as a steganographic technique. A mimic function transforms given input data into certain output that assumes the statistical properties of a different type of data, thereby concealing the true identity of the original data.

### 2.3.1 Regular Mimic Function

The Huffman mimic function [98], referred to as the "regular mimic function," is the functional inverse of the Huffman coding. The use of a mimic function involves three phases, digesting (i.e., Huffman tree building), encoding and decoding.

Like Huffman coding, a mimic function requires a Huffman tree to operate. In the digesting phase, a Huffman tree is constructed based on the frequency of each symbol appearing in a given piece of mimicry target data. In the encoding phase, the mimic function applies the *Huffman decoding* operation on the input data, and produces the mimicry output by referring to the Huffman tree. In the decoding phase, the mimic function applies the *Huffman encoding* operation, referring to the same Huffman tree, and uncovers the original input data from the mimicry output. In order to produce the mimicry output with a symbol frequency distribution similar to that of the mimicry target data, the input data must be random (i.e., follow uniform distribution). To meet this requirement, the input data can be randomized, such as XORing with a sequence of random numbers.

However, the regular mimic function suffers from a limitation that the symbol frequency of the mimicry output is limited to negative powers-of-2, e.g., 0.5, 0.25, 0.125, and so on. There are several techniques to overcome this limitation and we choose to use the high-order mimic function.

### 2.3.2 High-order Mimic Function

High-order mimic function differs from regular mimic function mainly in the digesting phase. Instead of building a single Huffman tree, an $n_{th}$-order mimic function constructs a collection of Huffman trees for a detailed "profile" of the mimicry target.

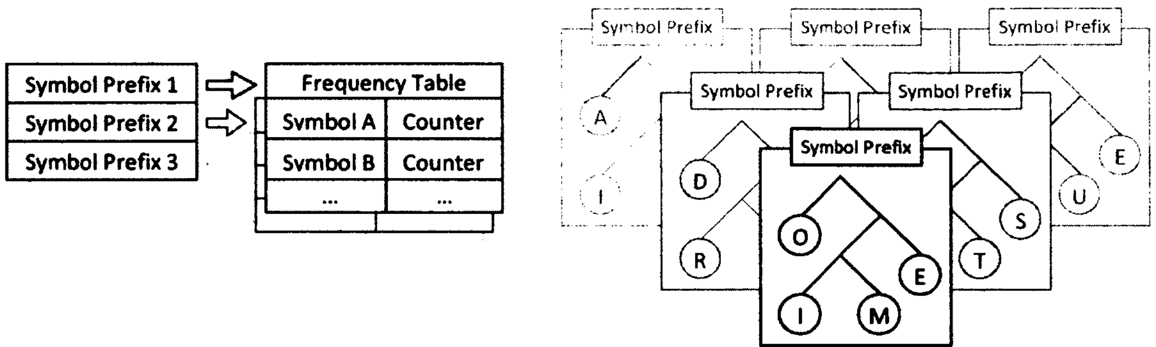



Figure 2.1: The Prefixed Symbol Tables Figure 2.2: The Prefixed Huffman Forest

Specifically, as shown in Figure 2.1, each observed unique symbol prefix of length $n-1$ is associated with a frequency table, which records occurrences of symbols with the given prefix. At the end of the digesting phase, each table is converted into a Huffman tree. This results in a forest of Huffman trees, each labeled by its symbol prefix, as shown in Figure 2.2.

Correspondingly, in the encoding and decoding phases of an $n_{th}$-order mimic function, a symbol prefix cache of length $n-1$ is maintained, recording the sequence of symbols that have just been encoded or decoded. For each symbol to be encoded or decoded, the high-order mimic function first locates the Huffman tree whose label corresponds to the current symbol prefix, and then performs Huffman decoding and encoding operations respectively, using the located Huffman tree.

Compared to a single Huffman tree in a regular mimic function, the Huffman forest in a high-order mimic function contains more detailed symbol frequency distributions as well as interdependencies among a number of adjacent symbols. As a result, the output produced by an $n_{th}$-order mimic function consists of n-grams that are observed in the mimicry target; and the occurrence of each n-gram is close to that of the mimicry target.

Table 2.1: Mimicry English Text

| ...I don't recommend using gA(t) to choose the safe. These one-to-one encoded with n leaves and punctuation. The starting every intended to find the same order mimic files. A Method is to break the trees by constructing the mimics the path down the most even though, offer no way that is, in this paper. Figure will not overflow memory. These produced by truncating letter. This need to handle n-th ordered compartment of nonsense words cannot bear any resemblance to B because... |
| --- |

Table 2.2: Mimic Function Runtime Analysis

| Digesting = $O(n)$ | | |
| --- | --- | --- |
| Table building | Reading a symbol | 1 |
| | Prefix lookup | 1 |
| | Recording frequency | 1 |
| | Input length | $n$ |
| Tree conversion | Sort | $C$ |
| | Construct tree | $C$ |
| | Number of tables | $C$ |
| Encoding / Decoding = $O(n)$ | | |
| For each symbol | Locate Huffman tree | 1 |
| | Huffman de(en)coding | 1 |
| | Input length | $n$ |

### 2.3.3 The Power of High-order Mimic Function

Compared to the polymorphic blending attack, the state-of-the-art payload mimicry technique, the high-order mimic function holds two major advantages.

**Structural and semantic mimicry**: The output of a high-order mimic function manifests structural and even semantic similarities to the mimicry target. Table 2.1 lists a sample mimicry text output produced by a $6_{th}$-order mimic function, using Wayner's paper [98] as the mimicry target. Without the concept of "word" or "grammar," the mimic function manages to produce the paragraphs with correctly spelled words and semi-sensible sentences. In addition, it also successfully reproduces the grammatical feature that every sentence starts with a capitalized letter. While a human reader may eventually realize that the output is mere mimicry, it is very difficult to differentiate the output from "normal" English text by using statistical tests, such as byte frequency (spectrum) and entropy. Some of the sentences can even trick grammar parsers.

**Run-time efficiency**: While polymorphic blending attack on large n-grams is a hard problem [26], the high-order mimic function have a linear time computational

complexity, as shown in Table 2.2. Let $R$ denote the order of a mimic function, and $M$ denote the number of possible symbols in a given language. In the digesting phase, using a hash table for prefix lookup, collecting symbol usages and constructing symbol frequency tables take linear time. Converting all symbol frequency tables into Huffman trees takes sub-linear time, with a constant bound[1]. And thus the digesting phase overall runs in linear time. The encoding and decoding phases essentially consist of a prefix lookup followed by a Huffman decoding or encoding, which are constant time operations for each input or output symbol. Therefore, the encoding and decoding phases also run in linear time.

### 2.3.4 Enhancements to High-order Mimic Function

The high-order mimic function is a powerful evasion technique against statistical anomaly detection, thanks to its ability to transform any data and reproduce statistical and structural features of the mimicry target. However, without proper enhancements, the mimic function falls short against semantic analysis detection.

Compared to human languages, binary machine languages (i.e., executables) have higher density and less structural flexibility. Without the knowledge of instruction syntax, the mimic function is unable to generate continuous long sequences of legitimate instructions. The control flows in the mimicry output are very often interrupted by malformed instructions, and thus fail to reproduce semantic properties of the mimicry target. We resolve this problem in our proposed mimimorphic engine by helping the mimic function understand the machine language. We augment the mimic function with customized assembler / disassembler. The enhanced mimic function is

[1] The total number of entries in each table is bounded by $M$ and the total number of tables is bounded by $Min(n, R^M)$. In theory, the constant $R^M$ can be very large. However, the upper bound is reached only when the input data is completely random. For meaningful data such as English text or executable binaries, the actual bound is much lower because the number of possible fixed-length substrings is limited.

Table 2.3: Mimimorphic Terms

| Terms | Description |
|---|---|
| Mimicry target | The target binaries to be mimicked |
| Mimicry digest | A high-order instruction "profile" produced by digesting the mimicry target |
| Mimicry output | The output of the mimic function |
| Mimicry instance | A fake executable composed from the mimicry output (contains malware encoded by the mimimorphic engine) |

aware of instruction syntax, and thus is capable of generating executable instructions as well as mimicking control flows.

## 2.4 Mimimorphic Engine Design

The mimimorphic engine consists of four major components: assembler, disassembler, high-order mimic function, and pseudo-random number generator. In this section, we describe the function of each component and detail the three operational phases of the mimimorphic engine: digesting, encoding and decoding. Table 2.3 defines a few important terms used throughout this section.

### 2.4.1 Digesting

In the digesting phase, the mimimorphic engine takes a set of binary executables as the mimicry target, and produces a mimicry digest—a high-order machine language "profile." Two components, the disassembler and the mimic (digesting) function, are involved in this phase, as shown in Figure 2.3.

Preparing for the digesting function, the disassembler decodes instructions in the executable binary streams into `CommonInst` structures, as shown in Figure 2.4. This structure is designed to provide a generalized abstraction from platform-specific ma-

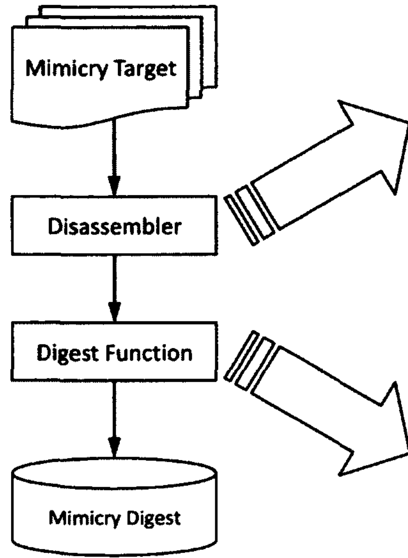



Figure 2.3: The Digesting Phase

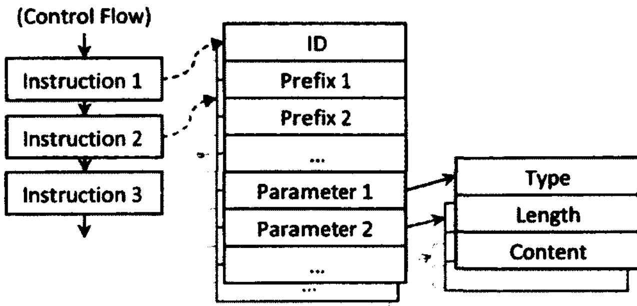



Figure 2.4: The `CommonInst` Structure

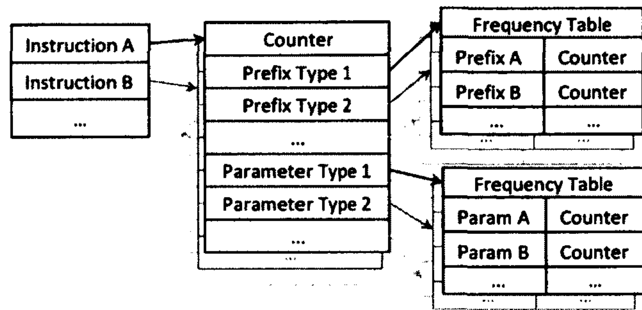



Figure 2.5: The Instruction Digest Table

chine instructions, making the mimimorphic engine easily deployable on any instruction set architecture. The ID field contains an index to identify each unique instruction. The mimic function treats this field as a symbol in the machine language. The prefix fields, not to be confused with the "symbol prefix" of the mimic function, correspond to the fields within an instruction that alter the instruction behaviors, such as atomic operation and address size override. The parameter fields record instruction parameters. Each parameter further includes three fields: type name, length, and content, indicating the type, size and content of a parameter, respectively.

After the disassembly, the digesting function processes the decoded instructions in a sequential manner. Internal to the digest function, a sequence of most recently processed instruction IDs, called *InstPfx*, is maintained, acting as the "symbol prefix" of the mimic function. For each `CommonInst`, the digest function first tries to locate an *instruction digest table (IDT)* associated with the *InstPfx*. And if absent, a new

table is created. Then, the digest function records the information of the `CommonInst` into the *IDT*. Finally, it appends the current instruction ID to *InstPfx* before moving onto the instruction.

The *IDT* consists of *instruction digest records (IDRs)*, indexed by the instruction ID. Each record includes a frequency counter of the instruction, as well as frequency counters of each type of prefixes and parameters, in the form of nested tables. To record the information of a `CommonInst`, we locate the *IDR* (or create a new one) with the matching instruction ID and increment its frequency counters and all the frequency counters corresponding to each of the prefixes and parameters noted in the `CommonInst`. Figure 2.5 illustrates the structure of an *IDT* and its *IDRs*.

At the end of the digesting phase, each *IDT* is converted into an *instruction Huffman tree (IHT)*, based on the frequency counter of each *IDR* inside the table. Correspondingly, each *IDR* is turned into an *instruction encoding template (IET)* by converting all the frequency tables associated with the prefixes and parameters into Huffman trees.

### 2.4.2 Encoding

Utilizing the mimicry digest, the encoding phase transforms an arbitrary piece of binary into a sequence of executable instructions that resembles the mimicry target. Three components of the mimimorphic engine—the pseudo-random number generator (PRNG), the mimic (encoding) function, and the assembler—are involved in this phase, as shown in Figure 2.6.

Algorithm 2.1 provides a high level overview of the mimimorphic encoding operations. Similar to the digesting function, the encoding function also maintains an *InstPfx*, recording the sequence of the most recently encoded instruction IDs. In the

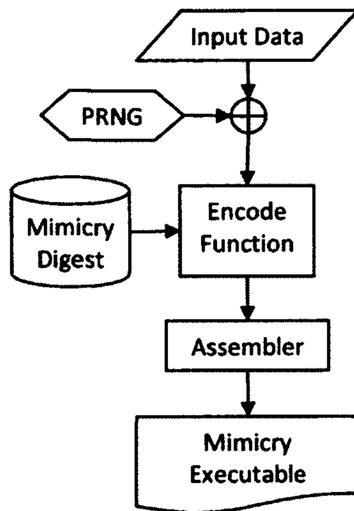



Figure 2.6: The Encoding Phase

---

**Algorithm 2.1** Mimimorphic Encoding

---

***Bin***: Input binary data
***Digest***: Mimicry digest
***RSeed***: Pseudo-RNG seed

Initialize $InstPfx$;
$SBin = Randomize(Bin, RSeed)$;
**while** $SBin$ **is not** empty **do**
  $IHT = Lookup(Digest, InstPfx)$;
  $IET = TreeWalk(IHT, SBin)$;
  $Inst = InstEncode(IET, SBin)$;
  Append $Inst$ to $InstCollection$;
  Update $InstPfx$ with $Inst$;
**end while**
$Result = Assemble(InstCollection)$;

---

`Randomize` function, the input data (i.e., malicious binary) is randomized by XORing with a pseudo-random data stream generated by the PRNG. This randomization is a dual purpose operation: on one hand, it ensures that the input data satisfies the requirement of the mimic function (i.e., uniformly distributed); on the other hand, it completely erases all the characteristics of the original binary. The `TreeWalk` function searches for an *IET* from the *IHT* by "walking" down the Huffman tree from the root node, taking left or right branches according to the (randomized) input bits—this is essentially a Huffman decoding operation. Then, the `InstEncode` function constructs mimicry instructions based on the *IET*. Each *prefix* or *parameter* field in the *IET* is associated with a Huffman tree, and thus the generation of a prefix or parameter is essentially a Huffman decoding operation as well. The constructed mimicry instructions are stored in the form of `CommonInst` structures, which are later converted to binary machine instructions by the assembler.

Figure 2.7 shows an example of a $7_{th}$-order mimimorphic engine generating an instruction in a function prologue. First, an *IHT* is looked up based on the six

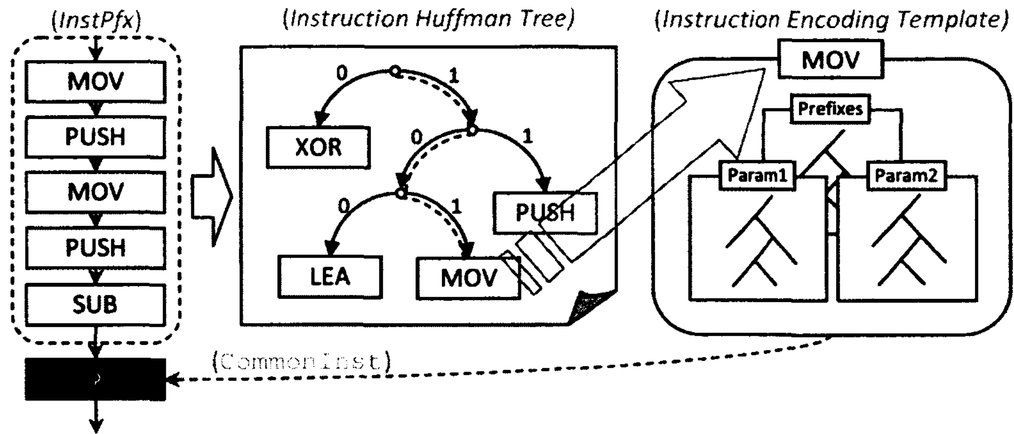



Figure 2.7: An Encoding Example

previously-generated instructions. Then the engine searches the tree branches according to the input bits, until a leaf node is reached. The leaf node is an *IET* of a "MOV" instruction, which contains the information of this instruction used after this particular prefix in the mimicry target. The encode function then leverages this information to generate a mimicry "MOV" instruction.

### 2.4.3 Decoding

The decoding phase is the inverse of the encoding phase, as shown in Figure 2.8. Based on the same mimicry digest, the decoding phase uncovers the input data from the mimicry output. There are three components of the mimimorphic engine involved in this phase: the disassembler, the mimic (decoding) function, and the PRNG.

The high level description of the mimimorphic decoding operations is given in Algorithm 2.2. Again, the *InstPfx* is used to record the most recently decoded instruction IDs. A mimicry instance is first disassembled into `CommonInst` structures, before being processed sequentially. The **NodeLookup** function locates the *IET* in the *IHT* with the matching instruction ID. Meanwhile, it produces a stream of data bits that corresponds to the branches to take from the root of the Huffman tree to the leaf

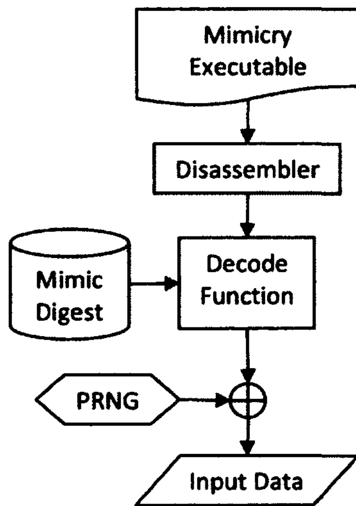



Figure 2.8: The Decoding Phase

**Algorithm 2.2** Mimimorphic Decoding

$\boldsymbol{Mimicry_{In}}$: Mimicry instance
$\boldsymbol{Digest}$: Mimicry digest
$\boldsymbol{RSeed}$: Pseudo-RNG seed

Initialize $InstPfx$;
$InstCollection = Disassemble(Mimicry_{In})$;
**for** each $Inst$ in $InstCollection$ **do**
  $IHT = Lookup(Digest, InstPfx)$;
  $(IET, IData) = NodeLookup(IHT, Inst)$;
  $IData = InstDecode(IET, Inst)$;
  Append $IData$ to $Data_{Rand}$;
  Update $InstPfx$ with $Inst$;
**end for**
$Result = Derandomize(Data_{Rand}, RSeed)$;

node—this is essentially a Huffman encoding operation. The `InstDecode` function further retrieves the data bits encoded in each mimicry instruction by performing similar Huffman encoding operations for all the prefixes and parameters with their corresponding Huffman trees in the *IET*. Finally, the `Derandomize` function uncovers the original data by XORing the decoded data with a pseudo-random data stream, which are generated by the PRNG with the same seed used in the encoding phase.

Figure 2.9 shows an example of a $7_{th}$-order mimimorphic engine decoding the instruction produced in the previous encoding example. First, an *IHT* is located based on the six previously-generated instructions. Then the engine looks up the leaf node *IET* that corresponds to the current instruction to be decoded, in this example, the "MOV" instruction. The path from the *IHT* root to this leaf node is then converted to data bits. The similar operations are performed for each of the prefixes and parameters of the "MOV" instruction, using the corresponding Huffman trees in the *IET* and producing a stream of data bits.

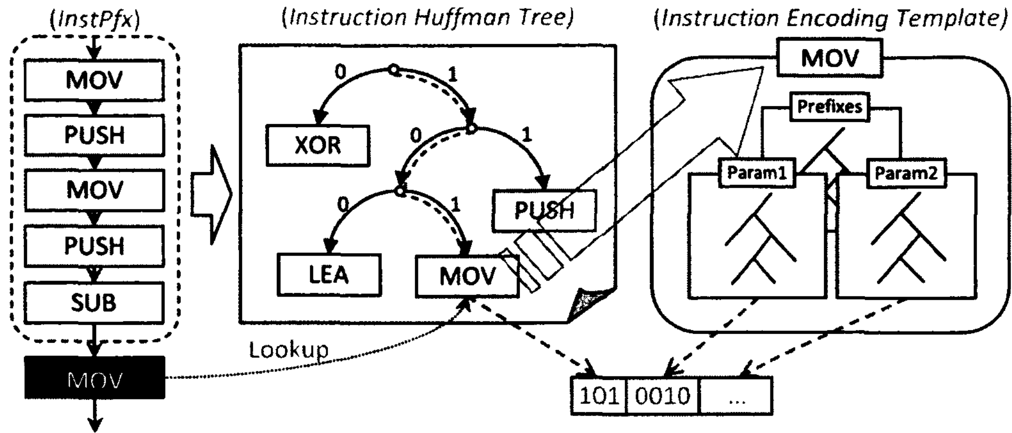



Figure 2.9: An Decoding Example

### 2.4.4 Design Issues

We now discuss a few important design issues in the digesting and encoding phases, which affect the quality of mimicry.

**Handling of embedded data in digest binaries:** In the digesting phase, the mimicry target binaries are first disassembled into `CommonInst` structures before digesting. However, in most legitimate executable binaries, there are a small but non-negligible amount of embedded data, which mainly consist of constants and jump address tables. Simply ignoring these embedded data might cause the statistical properties of the mimicry output to deviate from those of the mimicry target, resulting in the degradation of mimicry quality. We resolve this problem by masquerading embedded data as special one-byte-no-parameter "instructions" and digesting them along with other real instructions.

**Selecting a good random source:** Recall that, in the encoding phase, a regular mimic function requires input data to be uniformly distributed, so as to produce the mimicry output with the statistical properties approximating those of the mimicry target. Correspondingly, a high-order mimic function also requires the input data to be randomized on high-order. In our mimimorphic engine design, we select MT19937

PRNG [57], which claims to have equidistribution in 623 dimensions. Other PRNGs that can pass high dimensional distribution tests could be used as well.

**Ensure valid control flow generation:** Although the mimimorphic engine ensures valid instruction generation, it does not guarantee to produce valid control flows. This is because branch/call instructions use byte offset to redirect control flows. However, the lengths of x86 instructions are not fixed. And in addition, when the mimimorphic engine produces a branch/call instruction, there is no prior knowledge of subsequent instruction to be generated. As a result, a byte offset could point to the middle of a following instruction, invalidating the control flow. We resolve this problem by performing control flow correction on the intermediate data after the encoding phase. Instead of outputting the binary as soon as each instruction is generated, we keep all the `CommonInst` structures in a linked list. Then, for each branch and call instruction, we inspect whether its referring offset aligns to an instruction, and make corrections if necessary. We have verified the effectiveness of the solution by performing control flow analysis and basic block identification [50] on the mimicry output with and without control flow correction. We have observed that the number of valid basic blocks increases by nearly seven times with control flow correction.

## 2.5 Implementation

We have implemented a prototype of the mimimorphic engine on the Intel x86 architecture. While the current implementation works on the Windows XP, its core component is OS-independent and can be easily ported to Unix variants. In the following, we briefly describe some non-trivial implementation details.

First, a mimicry target is required for the mimimorphic engine to perform transformations. We randomly select 100 executable files from the `system32` folder of

the Windows XP, and extract their "text" sections to form a representative set of "normal" executables. System executables and libraries make good candidates of the mimicry target, because they exist on the majority of victim hosts, and they are also commonly delivered over the Internet (i.e., in forms of security patches).

Second, based on our observation of the basic block size of the mimicry target, we set the order of mimic functions to 7-8. Considering the unique feature of mimimorphism, we attempt to generate mimicked control flows that can be used to evade advanced semantic analysis detection. Because control flows are formed by basic blocks, the success of mimicking basic blocks is essential to the generation of mimicry control flows. We profile the basic block size of our selected mimicry target executable files, and observe that 89% of the executable files have the average basic block size less than or equal to eight instructions.

Third, we use a hash table to provide fast prefix lookup of *IHT*. Although the number of possible "symbol prefix" grows exponentially as the order of the mimic function increases, the number of observed unique prefixes is bounded by the size of input. With a relatively large hash table (22 bits), we are able to achieve reasonably low collision rate. In our experiments, the utilization rates of the hash table are below 20% and 25% for $7_{th}$-order and $8_{th}$-order mimic functions, respectively. For both mimic functions, 85% of entries are collision free, and over 99% of entries have less than or equal to one collision.

## 2.6 Evaluation

We use $7_{th}$-order and $8_{th}$-order mimic functions in the mimimorphic engine ($M_7$ and $M_8$ for short). An 83KB executable file is used as a hypothetical malware program, on which we apply mimimorphic transformations. For each $M_7$ and $M_8$, we generate

100 instances of the mimicry output, and each instance uses a different seed value for input data randomization. We evaluate the effectiveness of mimicry from two detection aspects: statistical test and semantic analysis test.

Note that, whether the executable file is a "real" malware or not is irrelevant to our evaluation. This is because (1) as stated in Section 2.4.2, the input randomization in the encoding phase has completely erased all the characteristics of the input data, thereby any input data would yield equivalent output; and (2) the detections we apply in our experiments are generic anomaly and similarity tests, instead of specific malware detections (such as commercial malware/virus scanners).

### 2.6.1 Statistical Tests

We run our mimimorphic output, $M_7$ and $M_8$ files, against statistical tests, namely the Kolmogorov-Smirnov and byte entropy tests. The Kolmogorov-Smirnov test is a general purpose statistical test frequently used in steganalysis—the analysis of steganographic techniques, whereas the byte entropy test is proposed for detecting packed or encrypted malware [54]. Although the Kolmogorov-Smirnov test is more powerful, it can only determine if a sample is anomalous, whereas the byte entropy can determine if a sample is, with high probability, a compressed or encrypted file.

The Kolmogorov-Smirnov test evaluates how much two samples (or a sample and a distribution) differ by measuring the maximum distance between two empirical distribution functions: $KSTEST = \max \mid S_1(x) - S_2(x) \mid$, where $S_1$ and $S_2$ are the empirical distribution functions of the two samples. This test is distribution free—in other words, the test statistic is not dependent on a specific distribution, and thus, is very general in applicability. For our experiments, we perform Kolmogorov-Smirnov test between samples of either mimicry or legitimate files and a collection of

| | Mean | Std. Dev. |
|---|---|---|
| Legitimate | 0.074 | 0.045 |
| $M_7$ | 0.109 | 0.007 |
| $M_8$ | 0.093 | 0.006 |

Table 2.4: Kolmogorov-Smirnov Results

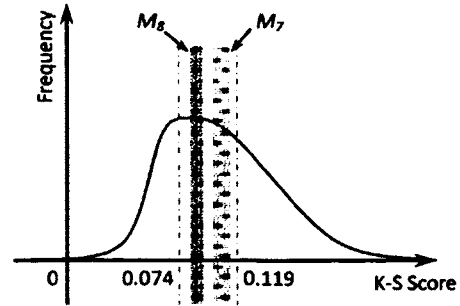



Figure 2.10: Kolmogorov-Smirnov Results

legitimate files. If the test statistic is low, the sample is classified as normal, otherwise it is classified as suspicious.

The mean and standard deviation of the Kolmogorov-Smirnov test scores are listed in Table 2.4. Although the test scores of the mimicry files are higher on average, the majority of these test scores fall within one standard deviation of the legitimate mean, as shown in Figure 2.10[2]. For the legitimate files, the mean score is 0.074 and the standard deviation is 0.045. For $M_7$ and $M_8$ files, the mean scores are 0.109 and 0.093, respectively. Therefore, the Kolmogorov-Smirnov test is unable to reliably differentiate mimicry files from legitimate files. The standard deviation of the mimicry files is very low compared to that of the legitimate files. This is mainly due to the size of the mimicry files. $M_7$ and $M_8$ files are approximately 2.4MB and 3.3MB, whereas the legitimate files range from 1KB to 0.5MB. As smaller files are statistically more likely to vary from the expected value, the variance of the mimicry files, whose sizes are larger on average, is very small.

The byte entropy detects compressed or encrypted data by measuring the randomness of the distribution of bytes: $entropy(X) = -\sum_{X} P(x) log P(x)$, where $X$ is a byte sample and $P(x)$ is the probability $P(X = x)$. For our experiments, we measure the byte entropy of different test samples, either mimicry or legitimate files. If the

[2] This figure is for illustration purpose only, and is not drawn to scale.

|  | Mean | Std. Dev. |
|---|---|---|
| Legitimate | 6.353 | 0.258 |
| $M_7$ | 6.545 | 0.021 |
| $M_8$ | 6.528 | 0.021 |

Table 2.5: Byte Entropy Results

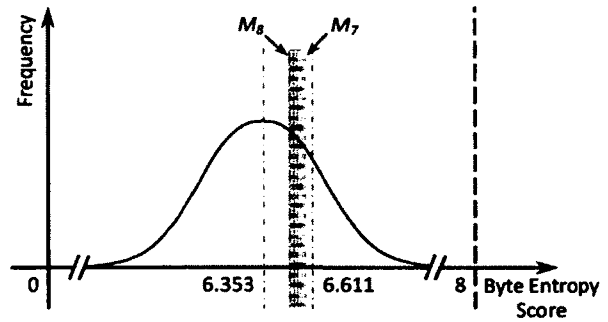



Figure 2.11: Byte Entropy Results

entropy is high, then the sample is suspected as compressed or encrypted malware, which may be further examined by unpacking via emulation or other dynamic analysis. However, if the entropy is low, i.e., in the range of typical executables, then the sample is classified as uncompressed and unencrypted.

The mean and standard deviation of the byte entropy test scores are listed in Table 2.5. For the legitimate files, the mean score is 6.353 and the standard deviation is 0.258. For $M_7$ and $M_8$, the mean scores are 6.545 and 6.528, respectively. Like the Kolmogorov-Smirnov results, the standard deviation of the mimicry files is very low, again due to their file sizes. In comparison to those of legitimate files, the test scores of $M_7$ and $M_8$ are slightly higher on average, but fall well within one standard deviation of the legitimate mean, as shown in Figure 2.11[3]. Based on these results, the byte entropy test is unable to differentiate mimicry files from legitimate files. Moreover, because packed and encrypted executables are identified by their high byte entropy scores [54], and thus $M_7$ and $M_8$ are also successful in disguising their packed content as normal executables.

[3]This figure is for illustration purpose only, and is not drawn to scale.

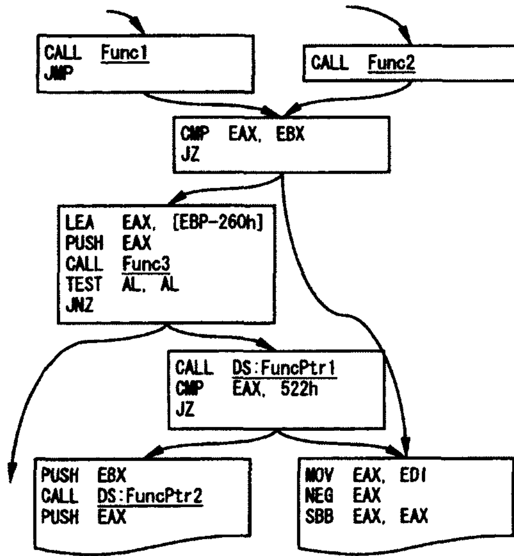



Figure 2.12: A Sample of $M_7$ Control Flow Graph

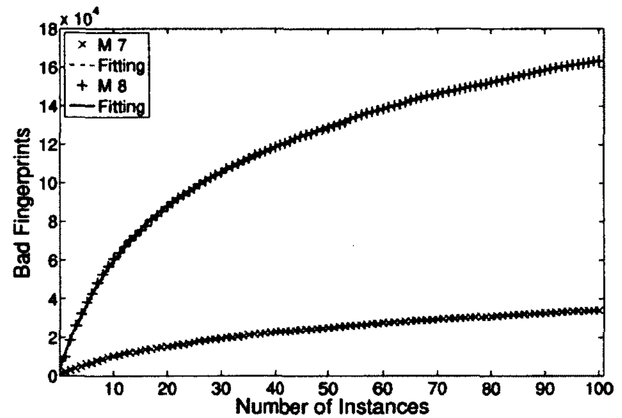



Figure 2.13: Bad Fingerprints in Collections of $M_7$ and $M_8$ Instances

### 2.6.2 Semantic Analysis Test

We use $M_7$ and $M_8$ to evaluate mimimorphic attacks against semantic analysis detection, particularly, the detection based on control flow fingerprinting [50]. This detection technique analyzes the control flows of binaries, and generates "fingerprints" for those control flows. To detect polymorphic malware, the system compares the fingerprints for suspicious network traffic against the fingerprints of known malware instances. If a sufficient number of fingerprints match, the detection system asserts that the traffic contains malware.

Mimimorphism attacks the control flow fingerprinting detection by introducing a large number of mimicked control flows resembling those of legitimate binaries. The detection system generates a number of fingerprints from a database of malware, i.e., $M_7$ and $M_8$ instances. A fingerprint is defined as "good" if it matches only malware files, but if the fingerprint matches both malware and legitimate files, it is considered as "bad." Figure 2.12 presents an example of "mimicry control flow graph" in an

| | $M_7$ | | | | $M_8$ | | | |
|---|---|---|---|---|---|---|---|---|
| | Mean | Std. Dev. | Max. | Min. | Mean | Std. Dev. | Max. | Min. |
| Bad fprts. | 1856.46 | 372.5 | 3321 | 1057 | 11407.99 | 912.42 | 14216 | 9606 |
| Matched files | 72.93 | 14.53 | 92 | 44 | 81.37 | 4.06 | 91 | 70 |

Table 2.6: Bad Fingerprints for $M_7$ and $M_8$ Instances

$M_7$ instance. Except for the underlined function addresses, the instruction sequence matches that of a system library file. As a result, the fingerprint generated from this segment of code is "bad." When the majority of fingerprints generated by the detection system are bad, it would suffer high false positives.

As a basic test, we first measure fingerprints that are common in the original hypothetical malware program and the $M_7/M_8$ instances. We observe that only one file from each set of instances, $M_7$ and $M_8$, has one or more common fingerprints with the hypothetical malware. The $M_7$ file shares three common fingerprints, while the $M_8$ file shares only one. Thus, overall $M_7$ and $M_8$ are successful in erasing the signatures from the original malware. We then proceed to measure the number of bad fingerprints produced from $M_7$ and $M_8$ instances, and the number of good and bad fingerprints shared by all $M_7$ and $M_8$ instances.

Table 2.6 presents the results of fingerprint comparisons in terms of mean, standard deviation, maximum and minimum counts between the legitimate files and $M_7/M_8$ files, respectively. The "bad fingerprints" row shows the number of bad fingerprints. The "matched files" row shows the number of legitimate files that share one or more fingerprints with a mimicry file. For all rows, larger numbers indicate that mimicry attacks are more successful.

On average an $M_7$ mimicry file contains 1856.46 bad fingerprints, and shares one or more bad fingerprints with 72.93% of the legitimate files. The most successful $M_7$ mimicry instance shares one or more fingerprints with 92 legitimate files, while

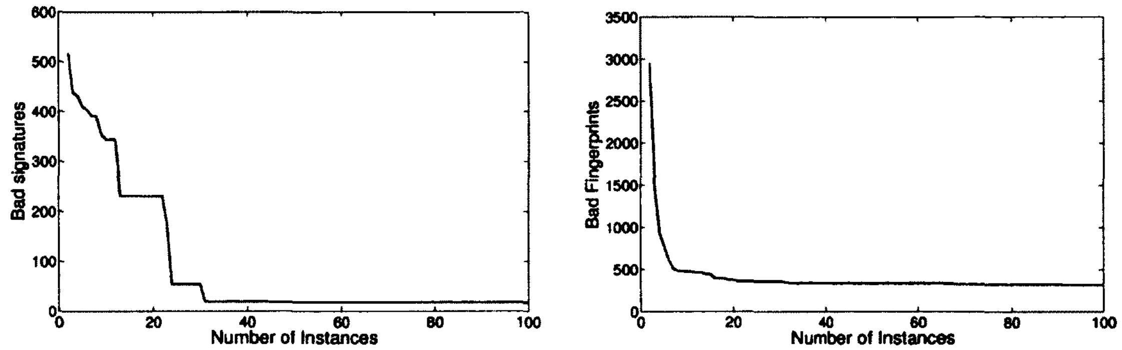



Figure 2.14: Shared Bad Fingerprints in Collections of $M_7$ and $M_8$ Instances

the least successful instance reproduces fingerprints in only 44 legitimate files. On average, an $M_8$ mimicry file contains 11407.99 bad fingerprints, and shares one or more fingerprints with 81.37% of the legitimate files. The most successful $M_8$ mimicry instance shares fingerprints with 91 legitimate files, while the least successful instance reproduces fingerprints in only 70 legitimate files.

Figure 2.13 illustrates the total number of bad fingerprints contained in a collection of $N$ $(1 \leq N \leq 100)$ $M_7$ and $M_8$ files. It highlights the mimimorphic engine's capability to mimic fingerprints from the legitimate files. The dashed and solid lines are curve fittings of the $M_7$ and $M_8$ data points, respectively. We can see that for both $M_7$ and $M_8$ files, as the number of instances increases, the total number of bad fingerprints increases, following a polynomial distribution.

Results in Table 2.6 and Figure 2.13 indicate that both $M_7$ and $M_8$ are successful in mimicking control flows of the mimicry target binaries. An $M_7$ or $M_8$ mimimorphic malware instance contains thousands to tens of thousands of bad fingerprints. As a result, the high false positive rates make it impractical to use the control flow fingerprints of a mimimorphic malware instance for detecting the other instances.

With greater efforts, a number of mimimorphic malware instances can be collected and analyzed, and their shared fingerprints can be extracted. However, our results

| | Shared fprnts. | Bad fprnts. | Good fprnts. |
|---|---|---|---|
| $M_7$ | 161 | 18 | 143 |
| $M_8$ | 339 | 321 | 18 |

Table 2.7: Shared/Good/Bad Fingerprints of All $M_7$ and $M_8$ Instances

show that such an approach can only achieve limited improvements on detecting mimimorphic malware. Figure 2.14 presents the results of the fingerprint comparisons between the legitimate files and a collection of $N$ instances ($2 \leq N \leq 100$) of $M_7$ and $M_8$ files, respectively. The line signifies the number of bad fingerprints. While the bad fingerprint counts for $N = 2$ decrease dramatically compared to the results in Table 2.6, the decrease slows down and the bad fingerprints stabilize at a non-zero value. More specifically, when $N = 100$, there are 18 bad fingerprints for $M_7$ files, and 321 bad fingerprints for $M_8$ files.

While the above two figures indicate very positive results for both $M_7$ and $M_8$, the number of shared fingerprints among 100 mimicry files presented in Table 2.7 gives us some surprises. When $N = 100$, there are 161 fingerprints shared by all $M_7$ files, but only 18 match legitimate files. This implies that the $M_7$ mimimorphic engine generates 143 fingerprints that can be used to identify the mimimorphic instances! However, the results are much better for $M_8$ files. Whereas there are 339 shared fingerprints in all $M_8$ files, 321 of them match legitimate files, leaving only 18 additional fingerprints. The polymorphic instances of a malware normally have tens to hundreds of shared fingerprints [50]. Thus, even with 100 instances of $M_8$ mimimorphic malware, there are still comparable amount of bad signatures mixed with the good signatures of the malware. Therefore, even given a large number of identified instances, $M_8$ mimimorphic malware can still maintain enough bad fingerprints to render the control flow fingerprints unusable.

## 2.7 Discussion

In this section we first present an interesting phenomenon observed in our evaluations. Then we discuss several issues related to the real world applications of Mimimorphism.

### 2.7.1 Artifact Generation

Table 2.7 shows an interesting phenomenon that the mimimorphic engine produces shared fingerprints in all instances that do not belong to any legitimate file. This phenomenon is caused by digesting data with limited order mimic functions.

When the mimic function digests two sequences of symbols that share a common subsequence with interdependencies longer than the order of the mimic function, the interdependencies will be partially merged. And thus the mimicry transformation may produce erroneous symbol sequences that mix and match the two original symbol sequences. For an intuitive example, the $6_{th}$-order English mimic function sometimes generates erratic words, such as "operationale" and "instructural", which are the combinations of the words "operational" and "rationale", and "instruction" and "structural", respectively. Because the the number of such long symbol sequences are very limited, the mimic function tends to reproduce the same erratic words persistently. This problematic phenomenon can be greatly reduced by increasing the order of the mimic function, as demonstrated by the $M_8$ files. Back to the previous example, a $7_{th}$-order English mimic function does not produce the word "instructural", because "instruction" and "structural" do not have any common 7-grams.

### 2.7.2 Robustness

Mimimorphism is robust against a range of static analysis detection methods, such as automatic n-gram signature generation, and certain types of semantic analysis

techniques. An $N_{th}$-order mimimorphic engine digests the mimicry target binaries in units of $N$ adjacent instructions, and thus its mimicry output consists of series of $N$ consecutive instructions observed from the mimicry target. As a result, the mimicry output could evade any $n$-byte-gram detection ($n \leq N \times b$, where $b$ is the instruction length in bytes). Based on our observation, the average length of Intel x86 instructions in a program lies between 2.1-2.8 byte. Thus, in theory, a 16-gram or higher byte test would be needed to reliably generate signatures for an $M_8$ mimimorphic malware. Semantic analysis techniques, which make decision based on short-range semantic similarities [104], are also vulnerable to mimimorphic attacks, due to the large number of randomly generated control flows that are similar to legitimate binaries.

The mimimorphic engine in our current design has limited ability to mimic program-level syntactic and semantic characteristics, such as function prologue, epilogue, and boundaries. We manually inspected an $M_8$ file, and found that about 45% of "functions" miss function prologue or epilogue sequence, and some relative jumps go across function boundaries. While it is possible to identify our mimimorphic instances leveraging these anomalies in program-level properties, we do not consider it a reliable detection approach. This is because the common program-level properties exist only by convention, and there are many legitimate programs that deviate from the norm, especially copyright protected executables that employ non-conventional protection techniques [21]. Detection malware based on non-conforming of programming conventions would suffer high false positive rates.

### 2.7.3 Application Constraints

There are two constraints for applying mimimorphism: memory consumption and payload inflation. Currently, the $M_7$ and $M_8$ mimimorphic transformations on average

consume 600MB and 1.2GB memory, and increase the payload size by 20 and 30 times, respectively. However, both constraints can be effectively mitigated.

To reduce memory consumption, we suggest implementing an on-demand, disk-based Huffman forest structure, which loads Huffman trees into memory as needed. With reasonable sized input for a mimimorphic transformation, only a small portion of Huffman trees will be traversed (bounded by input size) and thus the memory consumption is significantly reduced. To limit the payload size increase, we suggest applying compression to the input data before randomization, which could decrease the mimicry output size by up to 30% [17, 18]. Because the size inflation of mimimorphic transformation only occurs at the encoding phase, after input data randomization, compressing input data before randomization does not affect the inflation ratio.

### 2.7.4 Decoder Detection

Like polymorphic malware, mimimorphic malware requires to ship its decoder with the payload. The decoder needs to be directly executable and thus cannot be transformed into non-executable data. Mimimorphism is not designed to obfuscate the decoder. However, it does provide improved protection to the decoder binary compared to Polymorphism. The decoder is the common weakness of polymorphic malware, because its packed payloads have abnormal statistical properties and are *not executable*, making the decoder binaries easily extracted and analyzed. In contrast, mimimorphic payloads are indistinguishable from *executable binaries*, and thus correctly identifying the decoder binary for signature extraction becomes a non-trivial task. Techniques that "blend" the decoder control flow into the mimimorphic payload, side-by-side with hundreds of thousands of fake-but-legit-like mimicry control flows, can effectively thwarting attempts to extract the decoder statically.

## 2.8 Summary

Automatic malicious binary detection is the first line of defense against malicious software. To succeed in propagation, malware employ various binary obfuscation techniques to evade static detection, which real-time malware detection largely relies on. In this chapter we have presented a novel binary obfuscation technique, called mimimorphism.

Mimimorphism transforms a binary executable into a mimicry executable, with statistical and semantic characteristics highly similar to those of the mimicry target. Exploiting mimimorphism, malware can successfully evade a range of statistical anomaly detections, automatic substring signature generation, as well as some state-of-the-art semantic analysis techniques.

We have implemented a prototype mimimorphic engine on the Intel x86 platform. Our experimental results validate its efficacy in evading statistical anomaly detection—the byte frequency distribution test and entropy test—and a semantic analysis technique—the control flow fingerprinting. The mimimorphic binaries produced by the mimimorphic engine are indistinguishable from benign executables in the byte frequency distribution test and entropy test. And for control flow fingerprinting, the mimimorphic binaries are found to contain a large number of fingerprints that match legitimate binaries, leading to high detection false positives or false negatives. Moreover, even subjected to harder tests by training the detection system with a large number of mimicry executables for common fingerprints extraction, the mimimorphic engine can still introduce a significant quantity of benign-program-matching fingerprints, and thereby cause difficulties in accurate detections.

# Chapter 3

# Vulnerability in Server Power Management

Power management has become increasingly important for server systems. Numerous techniques have been proposed and developed to optimize server power consumption and achieve energy proportional computing. However, the security perspective of server power management has not yet been studied. In this chapter, we investigate energy attacks, a new type of malicious exploits on server systems. Targeted solely at abusing server power consumption, energy attacks exhibit very different attacking behaviors and cause very different victim symptoms from conventional cyberspace attacks. First, we unveil that today's server systems with improved power saving technologies are more vulnerable to energy attacks. Then, we demonstrate a realistic energy attack on a standalone server system in three steps: (1) by profiling energy cost of an open Web service under different operation conditions, we identify the vulnerabilities that subject a server to energy attacks; (2) exploiting the discovered attack vectors, we design an energy attack that can be launched anonymously from remote; and (3) we execute the attack and measure the extent of its damage in a systematic manner. Finally, we highlight the challenges in defending against energy attacks, and propose energy-aware programming, an effective defense scheme, to meet the challenges and evaluate its effectiveness.

## 3.1 Motivation

Power management is one of the critical issues for server systems nowadays. To date energy cost has become a major factor in the total cost of ownership (TCO) of large-scale server clusters [4, 34]. According to EPA [87], more than 100 billion kilowatt hours, representing a $7.4 billion annual cost, are estimated to be consumed by data centers in U.S. by 2011. As the prices of hardware components keep dropping, while their performance continuously improve, the proportion of energy cost in overall expense of server systems tends to grow even larger [4, 34].

Previous research on server power management mainly focus on reducing power consumption while maintaining acceptable quality of service. Numerous techniques have been proposed to improve energy efficiency in a variety of aspects, from low-level hardware features such as processor Dynamic Voltage and Frequency Scaling (DVFS) [23, 37] and hard disk spin-down [11, 33], to high-level system-wise management schemes such as cluster load provisioning [12, 71] and virtual machine consolidation [62]. While these power management advancements have significantly improved power savings[1], they have also opened up spaces for energy misuse. However, the security aspect of server system power management has not yet been paid attention to.

In this chapter, we investigate energy attacks, a new type of malicious exploits on server systems. Stealthily launched from remote by anonymous attackers, energy attacks increase the power consumption of a server system non-proportionally to its effective workload. Energy attacks are distinct from conventional cyber-space attacks in three interrelated aspects: objectives, attacking behaviors, and victim symptoms. First, energy attacks aim solely at abusing power consumption. They do not attempt to disrupt the normal service operations of the victim servers, nor

[1] For example, our study shows that a mainstream server in idleness consumes less than half of the energy consumed in full utilization.

to acquire sensitive information from the victim servers. Second, energy attacks are mounted in a stealthy manner, and they deliver damages over a relatively long period of time. An attacker's network flow is indistinguishable from those of the normal clients, in terms of traffic patterns or data fingerprints. Third, energy attacks manifest on victim servers only as increased energy usage, and no other induced anomalies such as significant performance degradation.

Although no immediately observable damages ensue, the consequences of energy attacks are serious. A successfully launched energy attack can cause a victim system to waste a large amount of energy, which in turn becomes waste heat, resulting in significantly increased power and cooling expense, shortened hardware lifespan, reduced reliability, and even permanent hardware failure. Current power management and security mechanisms provide virtually no defense against energy attacks.

To demonstrate the feasibility of launching an energy attack, we perform a step-by-step design and execution of a realistic energy attack on a Wikipedia mirror server. First, we profile the power consumption of the victim Web server under different page serving conditions, and identify a condition that triggers high energy consumption We then proceed to design an energy attack technique, achieving stealthiness by leveraging knowledge of human Web browsing behaviors. Finally, we evaluate our design by executing the attack and systematically measure the power consumption increases of the victim server under different load conditions.

## 3.2 Background

In this section, we first discuss the impact of energy proportional computing on a server system and present power measurements on our own server systems. Then, we describe the threat of energy attacks exposed on today's server systems.

### 3.2.1 Power Distribution

The power consumption in a server is mainly attributed to two sources, system component powering and cooling, with the latter heavily dependent on the former. Server system components mainly fall into the following categories: power supply, motherboard (chipset), processor, memory, and disk storage.

The power supply is responsible for transforming high voltage electricity input from a power outlet to a proper form of electricity (e.g., 5V, 12V DC) for all other server components. Although the power supply does not directly participate in service processing, it consumes a portion of input power due to conversion loss. The state-of-the-art power supplies guarantee over 90% efficiency at normal nominal loads (i.e., 20% - 100% of rated output) [88].

The motherboard and chipset provide the basic interconnection for all other system components. Modern server system chipset has nominal TDP[2] of 25 to 35W. Processors are usually the component that is capable of consuming the most of power per unit. A typical server processor's TDP is rated at 65 to 130W, and a server is usually equipped with one to four processors. Memory, typical Fully-Buffered DRAM (FBDIMM) for servers, has per-unit TDP of approximately 12W [41]. A server system usually has four to eight memory modules installed, which add up the overall memory power consumption to 48 to 96W.

The disk storage for a server system usually consists of multiple hard drives. The power consumption of a hard drive is mainly determined by its disk rotation speed: 7,200 and 10,000 RPM (Rotations Per Minute) drives on average consume 8W power while idling and 12W during operation [79], and 15,000 RPM drives on average consume 12W idle and 16W during operation [80].

[2]Thermal Design Power, a reference number of the typical amount of power a processor or chipset draws during full utilization, which is usually lower than its peak power consumption.

### 3.2.2 Energy Proportionality

Energy proportional computing [5] is an important concept in today's server systems. It aims to address the increasing energy concern and demand for power saving by making servers consume energy proportional to its workload. This goal is normally achieved by conditionally trading off performance for power savings.

Processors are the primary targets for power optimization, because of their high maximum power consumption (hundreds of watts per unit). Nowadays, the majority of server-class CPUs have employed power saving techniques that are already used in desktop and mobile processors, such as DVFS, multiple power states with reduced performance, and even power-down of idle cores. Motherboard and chipset feature the shutdown of unused circuitry, and memory chips also have several standby states with reduced power for no read/write cycles. Hard drives can only save a small portion of energy at idleness, due to their power demanding internal mechanical parts (spinning platters). However, they have another power saving mechanism called "spin-down", which shuts down the motor and thereby cutting down the majority of the power consumption, at a high (latency) cost of resuming service.

The ACPI (Advanced Configuration and Power Interface) specifications [36] are introduced to unify the power management of various types of devices in computer systems and provide well defined power management interfaces for both hardware and software. Within the specifications, multiple performance states are defined for a computer component. Each performance state corresponds to a specification of the expected performance and power consumption. At least one state is well defined: a full power state corresponds to the maximum performance. Depending on device type and manufacturing technology, additional number of lower states with reduced performance can be defined.

| | System A | System B |
|---|---|---|
| CPU | 2 × Xeon 5130 Dual Core | 2 × Xeon 5520 Quad Core |
| Memory | 4 × 1GB DDR2 FBDIMM | 6 × 1GB DDR3 FBDIMM |
| HDD | 4 × 7200RPM SATA | 6 × 7200RPM SATA |

Table 3.1: Server System Configurations

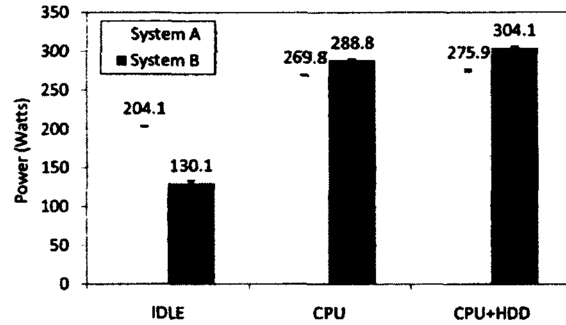



Figure 3.1: System Power Consumptions

Although modern operating systems are all capable of utilizing the ACPI to conserve energy under light load or in idleness, previous generations of server systems (such as our System A below) are not very energy proportional. This is because performance and security used to be the primary concerns for server systems, and thus the underlying hardware provides little or no support of additional performance states with reduced power consumption. However, as energy concerns weigh increasingly heavily, today's server systems have been becoming more energy proportional.

### 3.2.3 Real Server Measurements

We perform a small measurement study on system power consumption, using two server systems with different generations of hardware configurations, which are listed in Table 3.1. System A was bought in 2006 and System B was bought in mid-2009. We believe that both servers are representative of the mainstream system configurations at the time of purchase.

We measure the whole system power consumption in three different load scenarios: completely idle (IDLE), processors being fully utilized (CPU), and processors and hard drives being fully utilized (CPU+HDD). The "CPU" workload is generated by running multiple instances of a classic CPU benchmark program "linpack", and the number of instances corresponds to the number of logic cores. The "CPU+HDD"

workload is generated by running the "CPU" workload with the highest `nice` value and, at the same time, writing a large volume of data to the hard drives using the `dd` utility. The power consumption data is collected using a "Watts up? .Net" digital power meter [97], which is capable of measuring power usage with accuracy of $\pm 1.5\%$, at a time granularity of one second. The average power consumption of each scenario is obtained from a one minute long measurement, collecting 60 data points.

Two observations can be made from our measurement results shown in Figure 3.1: first, in high utilization scenarios System B (the newer server) consumes slightly more power than System A; second, and more interestingly, in the IDLE scenario, the power consumption of System B is significantly less than that of System A. While the first observation can be explained by System B having increased overall computation power than System A, the second observation presents us the direct proof that newer server system is becoming more energy proportional than previous generations. With higher computation power and improved energy proportionality, one can expect System B to yield more energy saving than System A under the same workload. However, we make an additional, alarming observation when we look at the advancements in energy proportional computing from a security perspective.

### 3.2.4 Threat of Energy Attacks

The improved energy proportionality has significantly changed the power profile of today's server systems. For example, our measurement data in Figure 3.1 shows that compared to IDLE, the CPU+HDD power consumption of System A increases by only 35%, while that of System B increases by 134%. The larger power consumption increase of System B indicates that it has a wider dynamic power range than System A. In other words, the power consumption of System B (the energy proportional

server) is more alterable than that of System A (the non-energy proportional server). And the increased power consumption alterability represents a new threat to server systems. The increased power consumption alterability represents a new threat to server systems. The power management mechanism of a server can be attacked by maliciously crafted workloads that target at consuming disproportional amount of energy, rendering the power saving ineffective, and resulting in significant energy wastage of the victim server.

Alarmingly, we realize that the threat of energy attacks is in fact an exploitable vulnerability because currently there is no effective defense against it. Existing power management schemes mainly focus on improving energy efficiency under normal operating conditions with benign workload, and thus they do not provide any defense against energy attacks. Moreover, most server systems do no have an efficient mechanism to measure power consumption, and thus could not even detect energy attacks, let alone defend against them.

### 3.2.5 Feature of Energy Attacks

Energy attacks on server systems target at exploiting the aforementioned power management vulnerability, and increasing a victim server's power consumption disproportional to its effective workload. Compared to other cyber-attacks, the damage of increased power consumption is delivered in an accumulative fashion over a relatively long period of time. As a result, energy attacks must meet two stealthiness requirements—low network-level signature and low performance degradation.

First, the attack should not exhibit traffic anomalies or have unique traffic patterns, since network traffic is often monitored for security purposes. Second, the attack should cause minimal performance impact on the victim server, because un-

usual performance degradation is a very visible sign that the server is under attack. The first requirement precludes high service request rate attacks, due to their obvious traffic anomalies. The malicious requests in an energy attack need to be sent at low to normal rate, and hence should be crafted to ensure a high per-request energy cost. In order to fulfill the second requirement, energy attacks must be adaptive to the workload condition of the victim server. Because the victim hosts an open service, its normal workload tends to vary significantly in time (e.g., correlated to the diurnal and weekly cycles). Inflicting a fixed malicious workload on the victim may either result in performance anomaly during high-load periods, or fail to incur the maximum damage during low-load or idle periods.

Note that energy attacks on server systems belong to a new attack class, which is very different from DoS (Denial of Service) attacks [42, 75, 92] in terms of their purpose, methodology and effects. Energy attacks aim to stealthily abuse a victim server's power consumption, and try hard to avoid causing any tangible service irregularities. In contrast, DoS attacks target at complete disruption of the victim's service, leveraging relatively simple attack strategies such as request flooding.

Moreover, because old generations of server systems are not energy proportional, to date energy has never been a target for DoS attacks mounted on server systems. And consequently, DoS attacks have mixed energy effects. In other words, not all DoS attacks result in increased power consumption of the victim server, and some could even lead to power consumption decrease. As an intuitive example, a TCP SYN flooding DoS attack exhausts the victim server's socket resource, and thus prevents the victim from receiving normal service requests. This attack causes the major components (e.g., CPUs and hard drives) of the victim serve to become idle, and hence significantly reduces its power consumption.

## 3.3 Energy Attack on Server Systems

In this section, we demonstrate the feasibility of launching an energy attack. First, we describe the scenario selection. We then design a realistic energy attack against an open Web server as a case study, covering the attack vector discovery, exploitation, and detection avoidance.

### 3.3.1 Scenario Selection

A great variety of tactics can be used to mount energy targeted attacks against server systems. For example, if attackers obtain administrator privilege on a victim system, they can deliberately mis-configure drivers and/or firmware, e.g., over-clock processor and memory, to operate the hardware components out-of-specs. Even with the privilege of a normal user, attackers can still easily increase the power consumption by running badly behaving programs such as a tight dead loop. However, the above-mentioned scenarios are not the focus of our study, because they are generally difficult to implement from remote (e.g., requiring privileged or physical access to the system).

We are interested in more commonly encountered scenarios, in which energy attacks can be launched without any special privileges. We assume that (1) the victim server runs an open service, which accepts service requests from the Internet; (2) the attackers have no physical access to the victim server; (3) the attackers only have equivalent privileges of "anonymous users" on the victim server (for example, they cannot change system configurations or execute arbitrary code); and (4) there are no exploitable security vulnerabilities on the victim system to escalate the attackers' privileges. In other words, the attackers communicate with the victim server using the same method as legitimate users, and the major variable they can manipulate is the server's workload, by crafting and submitting malicious service requests.

Thanks to the generic setting of attack environment, we believe that our scenarios are applicable to a wide range of servers, particularly, public Web services such as news, blogs, forums, public data services including file and image sharing sites, and search engines.

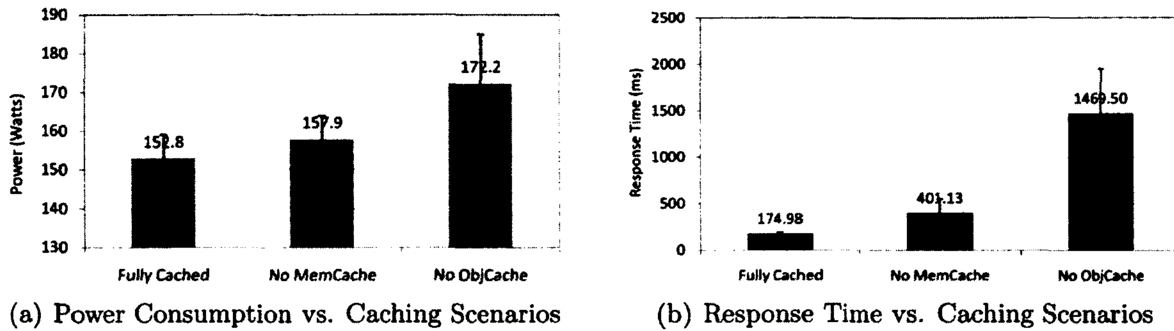
## 3.3.2 Case Study: Wikipedia Mirror Server

We perform a case study of designing an energy attack on an open Web server. We use System B (i.e., the newer, energy-proportional server) as the victim, running a Wikipedia service with setup detailed in Section 3.4.1. We choose Wikipedia mirror as our attack target because it is a freely available, content-rich Web service—a representative of real world production-use open Web services.

### 3.3.2.1 Identifying an Attack Vector

The Wikipedia mirror is powered by MediaWiki, a large-scale content management system. The contents of all MediaWiki pages are stored in a marked up format different from standard HTML, and pages are dynamically generated when they are requested. Two levels of caching, "object cache" and memory cache, help to optimize the performance.

MediaWiki stores the dynamically generated HTML contents in an "object cache"—a database table. When a page is requested repeatedly, the HTML content is retrieved directly from the object cache without being repeatedly generated. A cached HTML page expires either after a period of inactivity or the associated page content has been modified. In addition to the object cache, the MySQL database speeds up operations by storing a portion of frequently queried table entries, as well as table search indices and query results in a memory, employing a modified LRU replacement algorithm.




(a) Power Consumption vs. Caching Scenarios (b) Response Time vs. Caching Scenarios

Figure 3.2: Power Consumption and Response Time vs. Caching Scenarios

We profile the power consumption and service latency characteristics of the two caching mechanisms on the target server, using the "Watts up? .Net" digital power meter. Figures 3.2(a) and 3.2(b) show the average power usage and average response time for serving page requests from a single client in three different caching scenarios: pages being fully cached (in both memory and object cache), pages only in object cache, and pages not being cached. The power consumption averages are obtained from a one minute long measurement, collecting 60 data points; the response time averages are derived from the observed server response time during the power measurement, and the number of samples vary depending on the request servicing delay. The lower bound of Y-axis in Figure 3.2(a) is set to 130 watts, the system idleness power consumption. Thus, the columns in the figure represent the additional power consumption caused by the service requests.

From this measurement, we can observe that compared to fully cached requests, requests with memory cache misses incur 3% power increase and 129% processing time increase, and requests with object cache misses incur 12.7% power increase and 840% processing time increase. Because energy is defined as the product of power and time, the effect of cache misses on energy consumption increase is multiplicative. The high energy cost rendered by cache misses forms an effective energy attack vector to our Wikipedia mirror server.

#### 3.3.2.2 Exploiting the Attack Vector

Our next step is devising a method to exploit the discovered attack vector, that is, to generate requests that can cause cache misses, especially object cache misses. We examine previous studies in Web browsing behaviors. According to Barford and Crovella [3], Web page accesses on a Web server follow Zipf distribution, i.e. access frequency of a page correlates to its rank, and most accesses concentrate on a small number of pages while a large number of pages are rarely accessed. It is clear that the caching mechanisms in our Web server work well in handling such an access pattern because they are designed to optimize for similar access patterns. However, this knowledge also hints at a practical cache attack scheme. To generate page requests with high probability of cache miss, we may access pages in patterns following a very different distribution from Zipf. For the ease of study and implementation, we choose a uniform random page access pattern to exploit our attack vector.

#### 3.3.2.3 Detection Avoidance

The selected attack vector enables us to increase the victim's energy consumption without sending a large amount of requests. To avoid generating abnormal traffic patterns, we model the attacking request rate after "normal" Web clients. The study by Barford and Crovella [3] also shows that Web browsing exhibits an "active-inactive" behavioral pattern. During the active period, a client submits requests in a bursty manner, which is attributed to the browser downloading multiple resources (images, scripts, etc.) linked to a document. During the inactive period, the client pauses sending requests, presumably because of the user reading the page content. The length of the inactive period follows Pareto distribution.

For our experiments, we simplify the user behavior by "condensing" the active period into a single request, and only model the inactive period for request interarrival time. This is because all Wikipedia pages are text-oriented and structurally-alike. The client behaviors in all the active periods would be very similar.

In addition to traffic shaping, we also need to adaptively adjust the injection of malicious requests based on the workload of the victim server. This is achieved by associating the victim server's workload with the response time. During the attack, we monitor the response time, with which we can infer the server's workload, and adjust the amount of malicious requests accordingly.

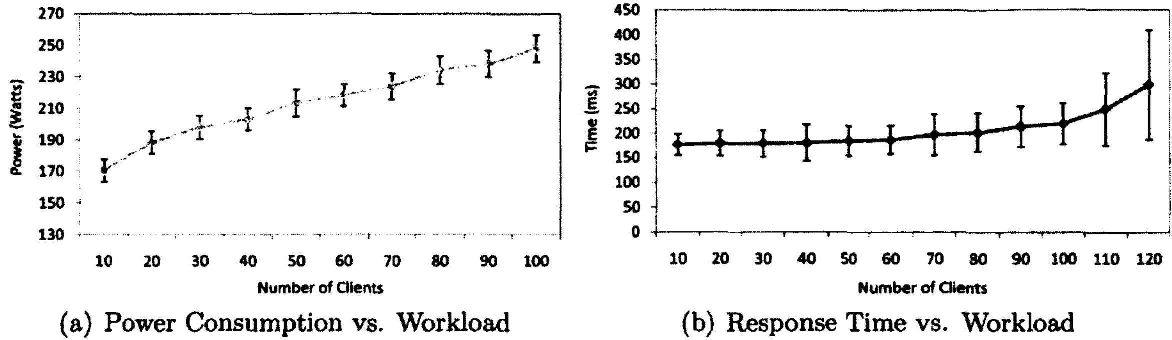
## 3.4 Attack Evaluation

In this section, we first describe the experimental setup. Then we present the detailed preparation and measurement results of the energy attack. And finally we assess the achievable damage of the energy attack.

### 3.4.1 Configuration and Setup

We set up a Wikipedia mirror server on System B using the classical LAMP combination (Linux, Apache, MySQL, and PHP). The database is imported from a Wikipedia dump containing 9,053,725 page entries. With a number of tests, we find that the server is capable of caching about 10,000 pages in memory. Therefore, we randomly pick 50,000 pages for use in our experiment.

We simulate client requests using a custom program running on a desktop computer. The client program simulates multiple clients each running in a separate thread. The "normal" clients are configured to access selected pages following Zipf distribution with $\alpha = 1$, and the request interarrival time follows Pareto distribution




(a) Power Consumption vs. Workload

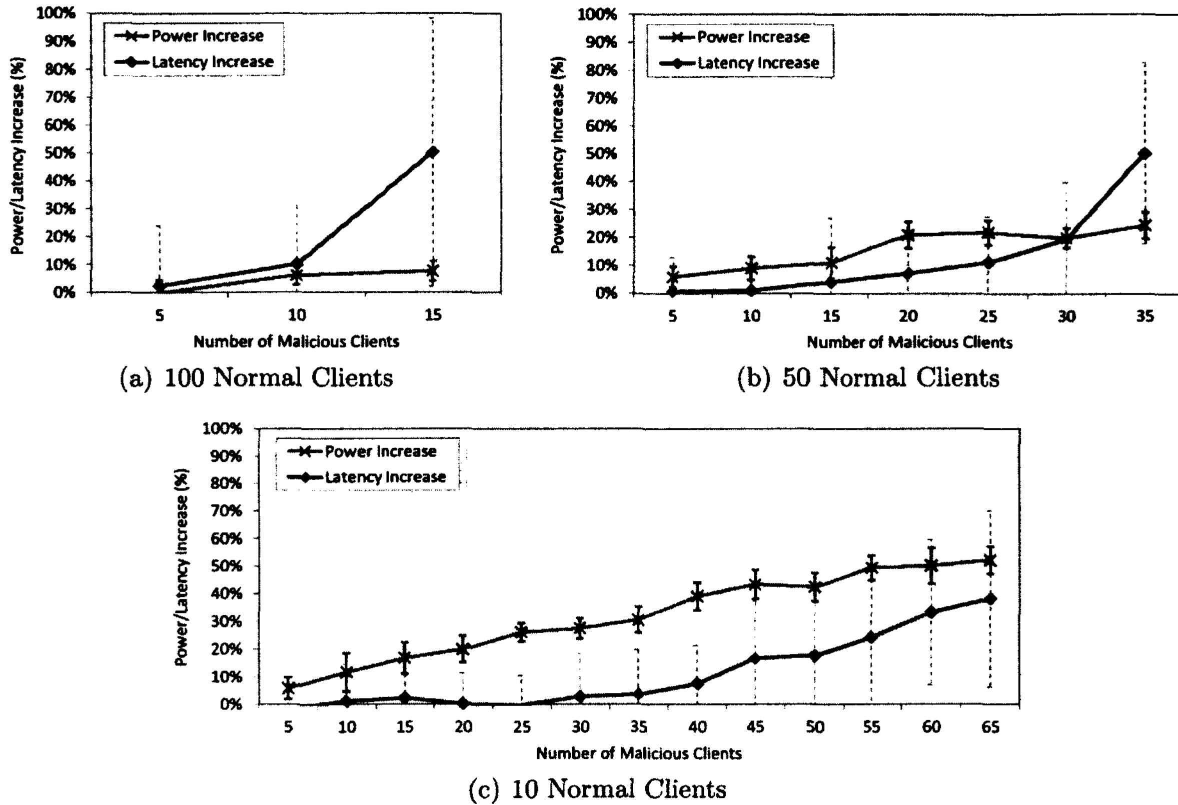
(b) Response Time vs. Workload

Figure 3.3: Power Consumption and Response Time Profiles of Victim Server

with $k = 1$ and $\alpha = 1.5$. The "malicious" clients are configured to access selected pages with the same request interarrival time distribution as the "normal" clients, but in a uniform random manner.

### 3.4.2 Workload—Response Time Profile

Before launching the attack, we first profile the victim server and establish the correlation between its workload and response time, as shown in Figure 3.3(b). Each data point is the average of 250 samples of service response time obtained under the corresponding workload. The error bar represents the standard deviation of response time. For light and moderate workloads (up to 50 clients), the server's response time increases quite slowly. When the workload increases beyond 60%, or 60 clients, the response time starts to rise significantly. With workloads in which the number of active clients is beyond 100, the server starts to show symptoms of being overloaded—all clients experience intermittent short burst of request failures in the form of "HTTP 500" errors. Therefore, we determine that the server is capable of stably supporting up to 100 normal clients. Figure 3.3(a) shows the correlation between stable workload and system power consumption, from which we can see that the server system power consumption is indeed proportional to its workload.




(a) 100 Normal Clients

(b) 50 Normal Clients

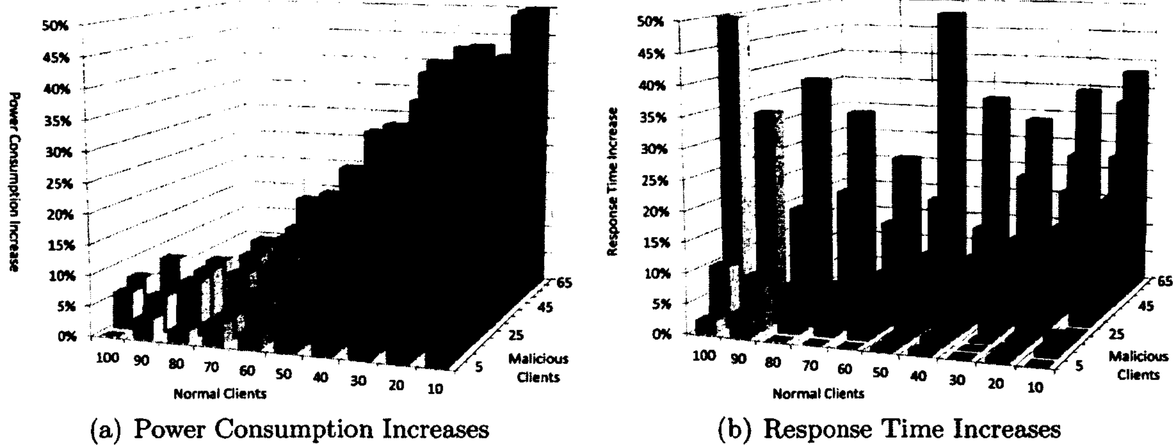
(c) 10 Normal Clients

Figure 3.4: Effects of Attack Under Selective Benign Workload

### 3.4.3 Attack Measurements

We use server-side power consumption and client-side perceived response latency to measure the effects of the energy attack. We conduct the experiments using different server workloads, which range from 10 to 100 normal clients with the increment of ten clients. For each workload, we inject energy attack traffic by adding a number of malicious clients. Due to the large volume of data, we only present the results corresponding to 100, 50, and 10 normal clients and depict them in Figures 3.4(a), 3.4(b), and 3.4(c), respectively. These figures show the increases in power consumption and response latency caused by the introduction of malicious workloads.

At 100% of the full load,the response latency of the victim server is very sensitive to the addition of malicious workloads. The malicious workload of ten malicious




(a) Power Consumption Increases (b) Response Time Increases

Figure 3.5: Collective Effects of Attack Under Different Benign Workload

clients increases the response latency by 7.6%, and the workload of 15 malicious clients increases the response latency by 50.2%. The power consumption, however, does not increase with the response latency, as the server is already fully loaded. At 50% of the full load,with 20 malicious clients, the attack results in 20.9% of extra power being consumed while only incurs 7.1% increase in response latency. However, with 30 or more malicious clients, the response latency increase surpasses the power consumption increase. At 10% of the full load,the energy increase caused by the attack becomes very evident. With 40 malicious clients, the victim server's power consumption increases by 39.0%, while the response latency only increases by 7.4%.

### 3.4.4 Damage Assessment

Our measurement results show that, at any stable workload, energy attacks will cause increased power consumption on the victim server. The more malicious clients, the larger the power increase. However, a larger number of malicious clients also results tangible performance degradation. Figure 3.5(a) and 3.5(b) presents the collective results of power consumption and service response time increases for all ten different

Table 3.2: Percentage of Power Increases due to Attack

| Utilization | 10% | 20% | 30% | 40% | 50% | 60% | 70% | 80% | 90% | 100% |
|---|---|---|---|---|---|---|---|---|---|---|
| Power Increase | 39.0% | 42.3% | 36.3% | 31.6% | 21.7% | 14.8% | 11.6% | 9.0% | 11.3% | 6.2% |

workloads with varying numbers of malicious clients. Note that samples with response time increment larger than 50% are omitted due to their unimportance.

To guarantee the success of an energy attack, low attack profile takes precedence over the power consumption increment. Therefore, the number of malicious clients needs to be limited to avoid significant response time impact. We refer to the workload—response time profile for a reasonable threshold. The standard deviation of response time at stable workloads varies between 12.3% and 21.1% of the measured values. We thus use the smallest percentage, 12.3%, as the upper limit of response time increases. With the chosen constraint, we determine the maximum power consumption achievable by the attacks for each workload, and present them in Table 3.2. We observe that the power impact of the energy attack is inversely correlated to the benign workload of the victim—an idle server suffers significant extra power consumptions, while a busy server only incurs small power consumption increases.

To assess the nominal damage of this energy attack to a server, we refer to the study of typical server workloads. Barroso and Hölzle [5] observe that most servers have average utilization between 10% and 50%. Correspondingly, under such utilization, our energy attack can result in 21.7% – 42.3% power consumption increase.

## 3.5 Defending Against Energy Attacks

The high potential damage of the energy attacks calls for an effective defense. In this section, we first discuss the difficulties in defending against energy attacks. Then we present our solution to meet the challenges, and validate its effectiveness.

### 3.5.1 Defense Challenges

To defend against energy attacks, it is necessary to differentiate malicious users from benign users, by the amount of energy consumed in serving their requests. Unfortunately, although the power consumption of the whole system can be measured in a coarse time granularity, today's servers are unable to provide fine-grained power consumption measurement due to the lack of hardware support. As a result, currently it is not possible to measure and account for the actual power consumption of servicing each individual request. And consequently, it is a very challenging task to devise an effective and generalized protection mechanism against energy attacks.

One may be tempted to suggest detecting an energy attack by other metrics in place of fine-grained power instrumentation. For example, the energy attack used in our case study can be uncovered by detecting abnormal page visit patterns, instead of referring to power consumption measurements. However, this naïve solution suffers in terms of soundness and completeness. First, the cause-and-effect relationships are not definitive, and thus detecting an energy attack by other metrics may lead to high false positives. For instance, our case study exploits an abnormal page visit pattern. But not all page visit patterns that deviate from the norm result in energy attacks. Second, energy attacks could exploit a great variety of alternative attack vectors, and render the monitoring system ineffective. Unlike buffer overflow or code injection vulnerabilities, the energy security issue is rooted deep in the server system's design, and it can manifest itself as very different, unrelated attack vectors. We discuss two alternative attack vectors in Section 3.6.1, and the exploitation of each attack vector requires a separate metric to discover.

Compared to the aforementioned "side-metric" monitoring strategy, a more holistic approach is to build the defense system based on fine-grained power consumption

information, and then derive the needed information by measuring hardware performance metrics. Neugebauer and McAuley [63] suggest using performance counter data such as CPU cycles, disk operations, and screen pixels to approximate power consumption for laptops and mobile devices. Buennemeyer *et al.* [8] present a battery-sensing intrusion protection system for mobile computers, which correlates device power consumption with Wi-Fi and Bluetooth communication activities. Kim *et al.* [44] propose a power-aware malware detection framework by collecting application power consumption signatures. While this approach achieves good generalization, it suffers low accuracy on server systems. Designed for personal use, mobile devices run few applications concurrently. In contrast, server systems are designed to process a large number of requests from multiple users in parallel. As a result, performance counter readings of independent request-serving processes can be heavily coupled (especially at fine granularity) and inaccurate for power approximation. For example, on a multi-processor system, processes competing for shared resources, such as the memory bus and PCI devices, can lead to heavy interferences of each other's cycle count readings. For another example, modern hard drives can intelligently reorder the sequence of operations to improve efficiency. However, this optimization can cause the operation latency disproportional to the complexity of a data request.

### 3.5.2 Energy-Aware Programming

The lack of hardware support makes fine-grained power measurement on today's server systems unachievable, while the parallel processing nature of request servicing renders low-level counter-based power approximation inaccurate. To work around these limitations and enable the design of an effective and generalized defense system, we take an application-oriented approach and propose energy-aware programming.

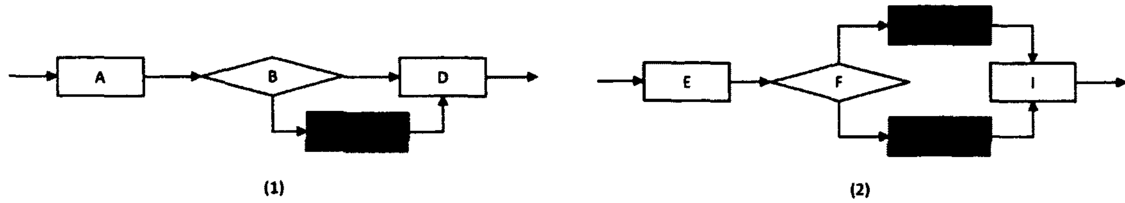



Figure 3.6: Example Component Flow Chart

The key idea of energy-aware programming is to capture the power consumption of individual request servicing in the form of application code execution, and enable an application to differentiate power consumption of service requests. Energy-aware programming infers power consumption information by leveraging high-level application context. Thus, this technique is much less prone to interferences than those low-level counter-based power approximation techniques.

#### 3.5.2.1 Power Consumption Characterization

In order to create an energy-aware application, application developers need to characterize the power consumption of the components in their applications, and embed such information into the program code. This can be accomplished in three steps.

The first step is the analysis and collection of conditionally invoked components. As shown by the two example flow charts in Figure 3.6, components C, G and H are conditionally invoked while all others are mandatory. The rationale behind this design is that the mandatory executed code contributes to the *baseline* power consumption for all request services, but the conditionally invoked code is responsible for the *dynamic* power consumption, which can increase dramatically for servicing power-extensive requests.

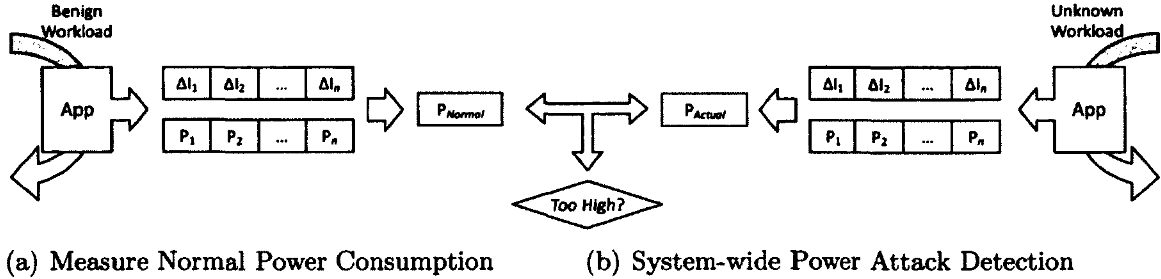
The second step is to characterize the power consumptions of component collection $\{C_1, C_2, \ldots, C_n\}$. This can be done using basic profiling techniques. Given a fixed time interval $t$ and a specific number of invocations $r$, each component $C_i$ is

invoked $r$ times in $t$ seconds, and the average system power consumption $Psys_i$ is measured. The effective power consumption $P_i$ corresponding to the component $C_i$ is then derived by contrasting $Psys_i$ with the system idle power consumption $P_{idle}$. The power consumption readings do not have to be very precise, because (1) in absence of fine-grained hardware support, it can be difficult to obtain accurate power consumption readings; and (2) the goal of this profiling is to differentiate components by their power consumption, and then use this information to infer the system dynamic power consumption state, as well as the nature of future workload.

The final step is to annotate the power information in the application. This is fulfilled by embedding the component power consumption table and inserting *power counters* into the program. For each component $C_i$, a counter $I_i$ is assigned, and each invocation of the component results in an increment of its associated counter.

#### 3.5.2.2 Power State Inference

With embedded power information in an application, the dynamic power consumption can be inferred by computing $P = \sum(P_i \cdot \Delta I_i)$, referring to the embedded component power consumption table and the increments of *power counters*. This calculation applies to the entire system as well as individual request servicing. In other words, the dynamic power consumption of the whole system at any given interval can be calculated by collecting the *power counter* increments during that interval; and the dynamic power consumption for servicing a specific request can be calculated by using the *power counter* increments caused by servicing this request. Therefore, an energy-aware application can self-monitor its power states at coarse-grained and fine-grained levels, and thereby is capable of detecting energy attacks, identifying attackers, and reacting accordingly.




(a) Measure Normal Power Consumption (b) System-wide Power Attack Detection

Figure 3.7: System Power Measurement and Attack Detection

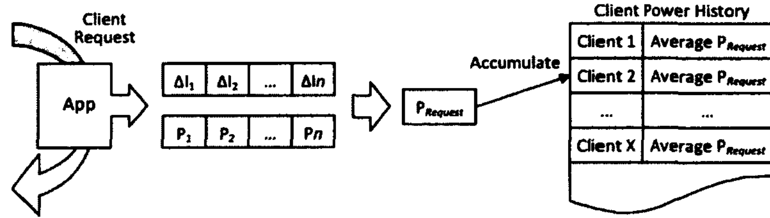



Figure 3.8: Maintaining Client Power History for Defensive Throttling

#### 3.5.2.3 Defense System Design

With the help of energy-aware programming, we design a simple and effective defense system to shield applications from energy attacks. The system is composed of three components, attack detection, power history maintenance, and defensive reaction.

The first component is responsible for detecting energy attacks on the server system. Before an energy-aware application is deployed, we first subject it to benign workloads, and record the system's normal `Power`, $P_{Normal}$, as shown in Figure 3.7(a). After the application has been deployed, the attack detection component monitors the `Power` of the entire system, and compares it with $P_{Normal}$. As shown in Figure 3.7(b), when the system power consumption surges significantly above the normal value, it asserts that the system is under an energy attack.

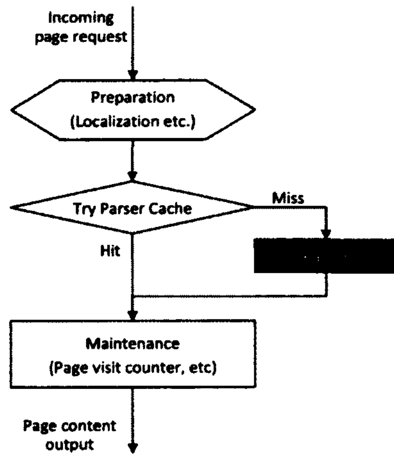
The second component is used for maintaining client power history records. Illustrated in Figure 3.8, for each communicating client, its average power consumption for request servicing is maintained. During an energy attack, these records are used

by the defensive reaction component as a reference to classify malicious clients. To scale with the client population growth, if necessary, individual clients' power history records can be aggregated on a per-subnet or per-domain basis.

The third component is designed for providing defensive reactions when an energy attack is detected. It sorts the power history records, and identifies the clients on top of the sorted list as malicious, because they represent the major sources of increased power consumption. It then applies defensive operations to these clients to reduce their energy impacts. For example, throttling down or blocking their request servicing for a period of time, or until the energy attack is stopped. The choices of defensive reactions are flexible, and customizable according to the tolerance of false positives.

Thanks to energy-aware programming, our defense system can uncover the stealthiness of malicious energy attack clients, and thus prevent potential evasion attempts. An attacker may utilize a large number of compromised machines (such as a botnet) and make each client very low profile. However, the attacking clients can still be identified and reacted upon by our defense system. This is because the service requests from each malicious client, although low in intensity, still consist of a high percentage of power-consuming service requests.

Although determined attackers might evade our defense system by lowering the concentration of malicious requests, and thus making a malicious client's power consumption comparable to that of a benign client. However, doing so would also significantly reduce the effectiveness of the energy attack, and require the attacker to exploit a much larger number of compromised clients and to generate much higher traffic volume to achieve the same effect. Such a practice would degrade an energy attack to a regular DDoS (Distributed DoS) attack, which can be defended by various previously proposed work [42, 75, 92]. The discussion of defending against DDoS attacks is beyond the scope of this chapter.



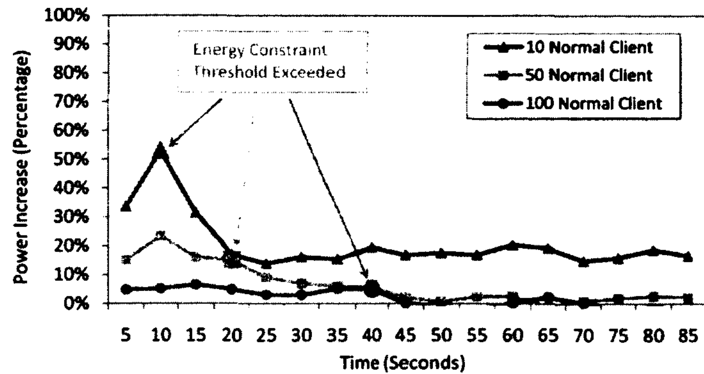




Figure 3.9: MediaWiki Flow Chart    Figure 3.10: Defense Measurement Results

### 3.5.3 Defense Experiment

To validate the effectiveness of our defense scheme, we have implemented a prototype defense system in our Wikipedia mirror server. In the following, we first briefly describe the implementation and then present the experimental results.

According to our profile measurements in Section 3.2, an object cache miss in page request servicing incurs high power consumption. We thus analyze the MediaWiki page request handling routine, and present its abstract flow chart in Figure 3.9. The main difference between object cache hit and miss lies in the invocation of the "Parser" component, which performs a processor intensive operation that dynamically generates HTML content from the Wiki style mark up text. Because the execution of the parser component is major source of additional power consumption, we can build an effective defense against energy attacks by protecting this single component.

We augment MediaWiki with "energy-awareness" by counting the parser invocations. Since there is only one power-extensive component, we simply omit power profiling, and assign 1.0 as this component's symbolic dynamic power consumption (`Power` for short). That is, if the parser is invoked while serving a page, the ap-

plication's `Power` is 1.0; otherwise, the `Power` is 0.0 . Effectively, the `Power` can be expressed as the parser invocation ratio.

The storage and computation overhead of the defense system is minor. For each client, a 12-byte power history record is used to maintain the client's identity, number of page visits, and its cumulative `Power`. For each request servicing, a hash table look-up is performed to retrieve the client's power history record, and then several arithmetic operations are performed to update the record. Overall, the overhead is negligible compared to the storage and computations required to serve a Wikipedia page.

We first test our defense system's ability to differentiate benign and malicious clients, as well as collecting the system's normal `Power` for defense purposes. When we subject the server to benign requests, we observe that the system `Power` stays around 0.04–0.05. However, when we inflict malicious requests on the server, the system `Power` is increased to 0.6–0.8. We heuristically set the system normal `Power` to 0.3, and set the protection threshold to be 30% parser invocations with 128 accesses. Therefore, when the victim server's system wide `Power` exceeds 0.3, users will be throttled down if they have over 128 recorded accesses, and have triggered the page parser invocation over 30% of the time.

We then configure and activate our defense system online. We measure the actual power consumption increases of the victim server under the same energy attack, and present the results in Figure 3.10. We use the same server workloads in Section 3.4. For 10 normal clients (i.e., low workload), the increase of server power consumption reaches about 50% at the beginning of attack. However, the attacker is unable to sustain the high power consumption increase. The parser invocation ratio quickly exceeds the threshold of our defense system, and the attacker's service requests are discarded afterwards. Correspondingly, the power consumption increase of the server

is reduced to 19%. We can observe similar effects for 50 and 100 normal clients—our defense lowers the power consumption increments from 21% to 4%, and from 6% to near zero, respectively. Meanwhile, page accesses from normal clients at all workloads are served without interruption.

## 3.6 Discussion

In this section, we first describe two other possible energy attack vectors, then discuss the applicability and scope of energy attacks, and finally discuss the limitation of energy-aware programming and the possibility of hardware-based mitigation.

### 3.6.1 Attack Variations

In addition to using cache miss as an attack vector, energy attacks can also be launched by exploiting other energy related vulnerabilities. For example, a file depositing server running an unmodified Linux kernel and allows users to control the names for stored files (such as a public FTP server) is vulnerable to energy attacks. The attacker can exploit a well known *nix kernel file name resolution vulnerability, and launch a low-rate algorithmic complexity attack [10, 15], which could significantly increase the victim server's processor utilization. Because a file depositing service is storage and network bandwidth bound, a well-controlled energy attack can avoid generating any throughput anomalies.

Besides the processors, other components with large dynamic power range can also be exploited by energy attacks. For example, hard drives normally consume 12 to 16 watts during operation, but their power consumption can be reduced to under one watt by spin-down the platters during long period of idleness. As a result, an energy attack on hard drives can be mounted by performing sleep deprivation attack

[56, 72] to prevent expected spin-down. Although the additional energy cost of a single attacked hard drive seems to be insignificant, the damage can accumulate to a significant amount when the energy attack targets at a decent sized storage server with 10 to 20 installed hard drives.

### 3.6.2 Attack Applicability

We have thoroughly investigated the proposed energy attack against a standalone server system. We use the case of single standalone server as the first step to study energy attack, because it is relatively easy to perform a clear analysis and repeatable evaluations. However, the attack vectors on a standalone server are not applicable to other hosting configurations, such as clustered servers and load balanced server farm. For example, our proposed energy attack on our Wikipedia mirror server is not effective on the actual Wikipedia website, which employs load balanced server clusters and heavy proxy caching techniques. In order to launch energy attack against a service configured in multi-server setup, one needs to discover and exploit new attack vectors.

Nevertheless, we believe energy attacks also pose serious threats to large scaled systems. For example, in a cloud hosting environment [24], competing cloud vendors may use energy attack as a powerful weapon to increase the operation cost of their opponents, making the attackers' service rates more attractive. To extend the scope of this work, we plan to study the interactions of workload and power consumption of multi-server systems, discover viable attack vectors, and devise defending techniques.

### 3.6.3 Limitation of Defense

We acknowledge that our defense strategy places the nontrivial duty of power profiling and energy accounting onto the application developers. However, because energy-

aware programming is currently the most effective holistic defense approach, we argue that its additional complexity is an acceptable trade-off for better security.

For more scalable and accurate solutions, we advocate enabling fine-grained power measurement at the hardware and operating system level, and make energy consumption information as accessible as the performance data. For example, the processor can include an "energy counter" similar to performance counters, and account the amount of energy consumed by a particular thread in given time period, based on the amount and variety of circuitry being activated.

## 3.7 Related Work

As energy cost of server systems takes a significant proportion of IT expenditures, research on power management techniques for server systems has been very active in recent years. Bianchini and Rajamony [6] perform a survey of power and energy research for server systems. Elnozahy *et al.* [23] present two power management mechanisms—dynamic voltage scaling (DVS) and request batching— to reduce energy consumption in Web servers. Horvath *et al.* [37] explore the benefits of using DVS in multi-stage service pipelines for power management in server farms. Felter *et al.* [25] study power shifting, which reduces peak power with minimal performance impact by dynamically reallocating power to performance critical components. Meisner *et al.* [59] design a system called Powernap, which can reduce server idle power by rapidly transitioning the entire system between a high-performance active state and a near-zero power idle state.

Barroso and Hölzle [5] present the concept of energy-proportional computing. They call for improvements in the energy usage profile of every server system component, particularly the memory and disk subsystems to achieve energy proportionality.

Barroso and Hölzle also point out that server systems may not benefit as much as the mobile systems from the energy-efficiency schemes targeting mobile devices, due to the distinct behavior of server workloads. And to make the entire server system energy-efficient, energy proportionality must be included in the design objectives for each component.

Energy management in server clusters has also been extensively studied. Chase *et al.* [12] design a resource management system called Muse with a primary focus on energy for large server clusters. Muse promotes energy efficiency of server clusters by balancing the cost of resources against the achieved benefit. Pinheiro *et al.* [71] propose a load concentration technique that can dynamically distribute the load and set some hardware resources in low-power modes to conserve energy. Elnozahy *et al.* [22] evaluate different combinations of cluster reconfiguration and dynamic voltage scaling. Rajamani and Lefurgy [74] investigate the key factors in the system-workload context that affect energy saving policies in server clusters. Heath *et al.* [35] study the energy conservation in heterogeneous server clusters using a model-based cooperative Web server. Fan *et al.* [24] present the aggregate power usage characteristics of several large-scale workloads from a data center over a period of six months and find that the opportunities for power and energy savings at the cluster-level are greater than at the rack-level.

Different from the research on power management in server systems that mainly focus on energy conservation, the security issue of power and energy has gained much attention in mobile computing community, because the power of battery is a critical and scarce resource for mobile devices. Dagon *et al.* [16] categorize a number of security problems caused by mobile malware and pointed out that battery exhaustion, a type of denial-of-service (DoS) attacks, is a serious threat to mobile computing. Martin *et al.* [56] present three types of battery depletion attacks. Racic *et al.* [73]

demonstrate that the attack on mobile phones' battery can be stealthily launched by exploiting the vulnerability of cellular service MMS (Multimedia Messaging Service) and that the attack can drain the power of batteries up to 22 times faster.

A number of research efforts have been spent in detecting and preventing attacks on battery power of mobile devices. Martin *et al.* [56] propose a power-secure architecture, which employs multi-level authentication and energy signatures, to counter power attacks. Buennemeyer *et al.* [8] present a battery-sensing intrusion protection system for mobile computers that correlates device power consumption with Wi-Fi and bluetooth communication activities. Kim *et al.* [44] propose a power-aware malware detection framework that can detect previously unknown energy-depletion attacks by collecting power consumption information of applications and comparing their power signatures with the signatures of normal applications.

## 3.8 Summary

Server systems have become more power efficient and energy proportional as power management technologies advance. However, the security aspect of power management has not yet been studied. In this chapter, we investigated the potential vulnerabilities in server power management.

First, we exposed the threat of energy attacks by measuring the power consumption of real server systems. Then, we designed and evaluated an energy abusing attack on server systems. In particular, we validated the threat of energy attacks on an open Web server running Wikipedia mirror service. By profiling power consumption of the target server under different operation conditions, we realized a viable energy attack vector. We conducted a series of experiments, in which energy attacks with varying attack intensities were carefully mounted to avoid incurring tangible degradation of

server performance. Our experimental results show that the proposed energy attack can incur significant increase of power consumption on the victim server. Finally, we presented an application-oriented defense approach to work around the current limitations of the hardware, and effectively protect a server against energy attacks.

Chapter 4

# Vulnerability in Virtualizated Public Cloud

Privacy and information security in general are major concerns that impede enterprise adaptation of shared or public cloud computing. Specifically, the concern of virtual machine (VM) physical co-residency stems from the threat that hostile tenants can leverage various forms of side channels (such as cache covert channels) to exfiltrate sensitive information of victims on the same physical system. However, on virtualized x86 systems, covert channel attacks have not yet proven to be practical, and thus the threat is widely considered a "potential risk". In this chapter, we present a novel covert channel attack that is capable of high-bandwidth and reliable data transmission in the cloud. We first study the application of existing cache channel techniques in a virtualized environment, and uncover their major insufficiency and difficulties. We then overcome these obstacles by (1) redesigning a pure timing-based data transmission scheme, and (2) exploiting the memory bus as a high-bandwidth covert channel medium. We further design and implement a robust communication protocol, and demonstrate realistic covert channel attacks on various virtualized x86 systems. Our experimental results show that covert channels do pose serious threats to information security in the cloud. Finally, we discuss our insights on covert channel mitigation in virtualized environments.

## 4.1 Motivation

As x86 virtualization technologies mature and being widely deployed, utility based cloud computing services are becoming increasingly attractive for enterprises. Cloud vendors today are known to utilize virtualization heavily for consolidating workload and reducing management and operation cost. However, due to the relinquished control from the data owner, data in the cloud is more susceptible to leakage by operator errors or theft attacks. Cloud vendors and users battle data leakage with network isolation (such as VLAN and VPN), encryption, traffic filtering, and intrusion detection. Despite the efforts being spend on information safeguarding, there remains potential risks of data leakage, namely the covert channels.

Covert channels exploit imperfections in the isolation of shared resources between two unrelated entities, and enable communications between them via unintended channels, bypassing mandatory auditing and access controls placed on standard communication channels. Previous research has shown that on a non-virtualized system, covert channels can be constructed using a variety of shared media [9, 52, 67, 78, 90]. However, to date there is no known practical exploit of covert channels on virtualized x86 systems.

Exposing cloud computing to the threat of covert channel attacks, Ristenpart *et al.* [77] have implemented an L2 cache channel in Amazon EC2 [77], achieving a bandwidth of 0.2 bps (bits-per-second), far less than the one bps "acceptable" threshold suggested by the Trusted Computer System Evaluation Criteria (TCSEC, a.k.a. the "Orange Book") [19]. A subsequent measurement study of cache covert channels [101] has achieved slightly improved speeds—a theoretical channel capacity of 1.77 bps[1]. Given such low reported channel capacities from previous research, it

[1]This value is derived from results presented in the original paper—a bandwidth of 3.20 bps with an error rate of 9.28%, by assuming a binary symmetric channel.

is widely believed that covert channel attacks could only do very limited harm in the cloud environment. Coupled with the fact that the cloud vendors impose non-trivial extra service charges for providing physical isolation, one might be tempted to disregard the concerns of covert channels as only precautionary, and choose the lower cost solutions.

In this chapter, we show that the threat of covert channels in the cloud is real and practical. We first study existing cache covert channel techniques and their applications in a virtualized environment. We reveal that these techniques are rendered ineffective by virtualization, due to three major insufficiency and difficulties, namely, *addressing uncertainty*, *scheduling uncertainty*, and *cache physical limitations*. We tackle the addressing and scheduling uncertainty problems by designing a pure timing-based data transmission scheme featuring relaxed dependencies on precise cache line addressing and scheduling patterns. Then, we overcome the cache physical limitations by discovering a high-bandwidth memory bus covert channel, exploiting the atomic instructions and their induced cache–memory bus interactions on x86 platforms. Unlike cache channels, which are limited to a physical processor or a silicon package, the memory bus channel works system-wide, across physical processors, making it a very powerful channel for cross–VM covert data transmission.

We further demonstrate the real world exploitability of the memory bus covert channel by designing a robust data transmission protocol and launching realistic attacks on our testbed server as well as in the Amazon EC2 cloud. We observe that the memory bus covert channel can achieve (1) a bandwidth of over 700 bps with extremely low error rate in a laboratory setup, and (2) a real world transmission rate of over 100 bps in the Amazon EC2 cloud. Our experimental results show that, contrary to previous research and common beliefs, covert channels can achieve high bandwidth and reliable transmission on today's x86 virtualization platforms.

## 4.2 Related Work

Covert channel is a well known type of security attack in multi-user computer systems. A covert channel is formed by a pair of colluding parties, with the sender operates a shared resource in ways that allow the receiver to make distinguishable observations of the same resource, thereby conveying information in a stealthy manner.

Originated in 1972 by Lampson [52], the threats of covert channels are prevalently present [9, 19, 58, 67, 78, 81, 82, 90, 95] in systems with shared resources. Lampson [52] described a covert channel leveraging the file system locking mechanism, with which the locking states of a shared file are used to transport information (i.e. storage channel). Van Vleck [90] discussed a covert channel based on the memory paging mechanisms of the Multics operating systems. The data are covertly transmitted by the presence or absence of memory pages in public libraries, which is inferred by the paging performances (i.e. timing channel). Rowland [78] and Cabuk *et al.* [9] explored construction of covert channels over legitimate communication channels, exploiting the network protocols. Shah *et al.* uncovered covert channels in wireless network channels [81] and input devices [82].

Cache-based covert channels have attracted more attention in recent studies. Compared to other covert channel media, the processor cache is more attractive for exploitation, because its high operation speed could yield high channel bandwidth, and the low level placement in the system hierarchy can bypass many high level isolation mechanisms.

Percival [67] introduced a technique to construct inter-process high bandwidth covert channels using the L1 and L2 caches, and demonstrated a cryptographic key leakage attack through the L1 cache side channel. Wang and Lee [95] deepened the study of processor cache covert channels, and pointed out that the insufficiency of

software isolation in virtualization could lead to cache-based cross–VM covert channel attacks. Ristenpart *et al.* [77] further exposed cloud computing to covert channel attacks by demonstrating the feasibility of launching VM co-residency attacks, and creating an L2 cache covert channel in the Amazon EC2 cloud. Xu *et al.* [101] conducted a follow up measurement study on L2 cache covert channels in a virtualized environment. Based on their measurement results, Xu *et al.* concluded that the harm of data exfiltration from cache covert channels is quite limited due to low achievable channel capacity.

In response to the discovery of cache covert channel attacks, a series of architectural solutions have been proposed to limit cache channels, including RPcache [95], PLcache [47], and Newcache [96]. RPcache and Newcache employ randomization to prevent data transmission by establishing a location-based coding scheme. PLcache, in contrast, is based on enforcing resource isolation by cache partitioning. However, a drawback of hardware-based solutions is their high adaptation cost and latency.

Offering immediately deployable protection, HomeAlone [105] proposes to proactively detect the co-residence of unfriendly VMs. Leveraging the knowledge of existing cache covert channel techniques [67, 77], HomeAlone detects the presence of a malicious VM by acting like a covert channel receiver and observing cache timing anomalies caused by another receiver's activities.

The industry took a more pragmatic approach to mitigating covert channel threats. The Amazon EC2 cloud provides a featured service called dedicated instances [1], which ensures VMs belonging to each tenant of this service do not share physical hardware with any other cloud tenants' VMs. This service effectively eliminates various covert channels induced by the shared platform hardware, including cache covert channel. However, in order to enjoy this service, the cloud users have to pay a significant price premium. As of the time of writing (March, 2012), each dedicated instance

incurs a 23.5% higher per-hour cost than regular usage. In addition, there is a $10 fee per hour/user/region. Effectively, for a user of 20 small instances, the overall cost of using dedicated instances is 6.12 times more than that of using regular instances.

Of historical interest, the study of covert channels in virtualized systems is far from a brand new research topic—legacy research that pioneered this field dates back over 30 years. During the development of the VAX security kernel, a significant amount of effort has been paid to limit covert channels within the Virtual Machine Monitor (VMM). Hu [38, 39] and Gray [30, 31] have published a series of follow up research on mitigating cache channels and bus contention channels, using timing noise injection and lattice scheduling techniques. However, this research field has lost its momentum until recently, probably due to the cancellation of the VAX security kernel project, as well as the lack of ubiquity of virtualized systems in the past.

## 4.3 Struggles of the Classic Cache Channels

Existing cache covert channels (namely, the classic cache channels) employ variants of Percival's technique, which uses a hybrid timing and storage scheme to transmit information over a shared processor cache, as described in Algorithm 4.1.

On non-virtualized hyper-threaded systems, the classic cache channels work very well, achieving transmission rates as high as hundreds of kilobytes per second [67]. However, when applied in today's virtualized environments, the achievable rates drop drastically, to only low single-digit bits per second [77, 101]. The multiple orders of magnitude reduction in channel capacity clearly indicates that the classic cache channel techniques are no longer suitable for cross–VM data transmission. Specifically, the data transmission scheme of a classic cache channel suffers three major obstacles—addressing uncertainty, scheduling uncertainty, and cache physical limitation.

**Algorithm 4.1** Classic Cache Channel Protocol

***Cache*[*N*]**: A shared processor cache, conceptually divided into $N$ regions;
Each cache region can be put in one of two states, *cached* or *flushed*.
**$D_{Send}[N]$**, **$D_{Recv}[N]$**: $N$ bit data to transmit and receive, respectively.

| **Sender Operations:** | **Receiver Operations:** |
|---|---|
| (Wait for receiver to initialize the cache) | **for** $i := 0$ **to** $N-1$ **do**<br>{Put $Cache[i]$ into the *cached* state}<br>Access memory maps to $Cache[i]$;<br>**end for** |
| **for** $i := 0$ **to** $N-1$ **do**<br>**if** $D_{Send}[i] = 1$ **then**<br>{Put $Cache[i]$ into the *flushed* state}<br>Access memory maps to $Cache[i]$;<br>**end if**<br>**end for** | (Wait for sender to prepare the cache) |
| (Wait for receiver to read the cache) | **for** $i := 0$ **to** $N-1$ **do**<br>Timed access memory maps to $Cache[i]$;<br>{Detect the state of $Cache[i]$ by latency}<br>**if** $AccessTime > Threshold$ **then**<br>$D_{Recv}[i] := 1$; {$Cache[i]$ is *flushed*}<br>**else**<br>$D_{Recv}[i] := 0$; {$Cache[i]$ is *cached*}<br>**end if**<br>**end for** |

### 4.3.1 Addressing Uncertainty

Classic cache channels modulate data by the states of cache regions, and hence a key factor affecting channel bandwidth is the number of regions a cache being divided. From information theory's perspective, a specific cache region pattern is equivalent to a transmitted symbol. And the number of regions in a cache thus corresponds to the number of symbols in the alphabet set. The higher symbol count in an alphabet set, the more information can be passed per symbol.

On hyper-threaded single processor systems, for which classic cache channels are originally designed, the sender and receiver are executed on the same processor core,

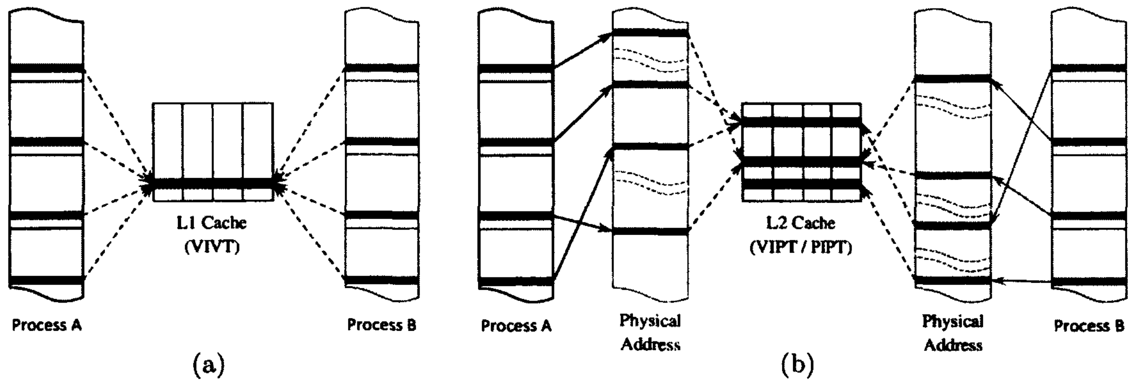



Figure 4.1: Memory Address to Cache Line Mappings for L1 and L2 Caches

using the L1 cache as the transmission medium. Due to its small capacity, the L1 cache has a special property that its storage is addressed purely by virtual memory addresses, a technique called VIVT (virtually indexed, virtually tagged). With a VIVT cache, two processes can impact the same set of associative cache lines by performing memory operations with respect to the same virtual addresses in their address spaces, as illustrated in Figure 4.1(a). This property enables processes to precisely control the status of the cache lines, and thus allows for the L1 cache to be finely divided, such as 32 regions in Percival's cache channel [67].

However, on today's production virtualization systems, hyper-threading is commonly disabled for security reasons (i.e., eliminating hyper-threading induced covert channels). Therefore, the sender and receiver could only communicate by interleaving their executions. Since the L1 cache is completely flushed at context switches, only those higher level caches (e.g., the L2 cache) whose contents are preserved across a context switch can be leveraged for classic cache channel transmission. Unlike the L1 cache, the storage in these higher level caches is not addressed purely by virtual memory addresses, but either by physical memory addresses (PIPT, physically indexed, physically tagged), or by a mixture of virtual and physical memory addresses (VIPT, virtually indexed, physically tagged). With physical memory addresses involved in

Table 4.1: Experimental System Configurations

| | System A | System B |
|---|---|---|
| CPU | Core2 Q8400, 2.66GHz,<br>Caches: (size, set-associativity)<br>L1D – 32KB, 8-way<br>L2 – 2MB, 8-way | 2 * Xeon E5520, 2.26GHz,<br>Caches: (size, set-associativity)<br>L1D – 32KB, 8-way<br>L2 – 256KB, 8-way<br>L3 – 8MB, 16-way |
| Memory | DDR2 DIMM, 1621MHz | DDR3 FBDIMM, 2153MHz |

Table 4.2: Cache Latencies vs. Access Pattern Lengths

System A (L2 Associative Set Size = 256KB)

| Accesses | 1–6/7 | 8–24/32/64 | More than 64 |
|---|---|---|---|
| Latency | 8 cycles (const.) | 8 cycles/access | 48 cycles/access |

System B (L2 Associative Set Size = 32KB)

| Accesses | 1–18 | 19–64 | More than 96 |
|---|---|---|---|
| Latency | 4 cycles (const.) | 2 cycles/access | 33 cycles/access |

cache line addressing, given only knowledge of its virtual address space, a process cannot be completely certain of the cache line a memory access would affect due to address translation.

We demonstrate the phenomenon of addressing uncertainty by conducting experiments on two systems with configurations shown in Table 4.1. We first calculate the *associative block* sizes of the L2 caches by dividing the total cache capacities over their corresponding set-associativity counts. Then we measure the latencies of a repeating sequence of random memory accesses[2], with each access spaced multiple *associative blocks* apart. The repeating sequence length starts at one, and is incremented by one for each measurement.

As shown in in Table 4.2, when the repeating sequence size is small (i.e., the first data column), the access latencies on both systems are small constants, due to the

[2]The randomness is introduced to avoid the interference of hardware prefetching.

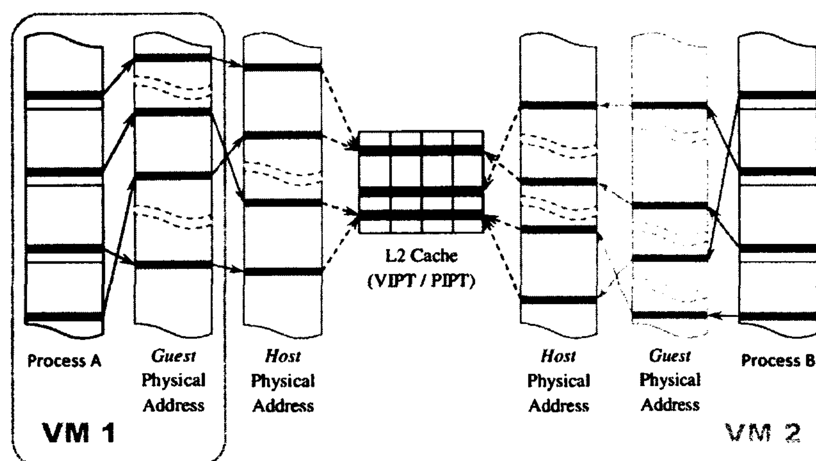



Figure 4.2: Memory Address to L2 Cache Line Mapping with Virtualization

caching effect of the L1 data cache[3]. As the repeating sequence size grows larger (i.e., the second data column), the access latencies begin to increase at slow rates, evidencing that the L2 cache addressing involves physical memory addresses. With a VIVT cache, one would expect that the access latency begins to increase sharply, since all memory accesses would collide onto the same associative set of cache lines and induce a thrashing-like behavior. However, with physical address involved in cache addressing, the memory accesses tend to spread over more than one set of cache lines because physical memory tends to be less continuous than virtual memory. As a result, the memory accesses continue to be cached by the L2 cache while missing the L1 cache. Finally, when the repeating sequence length grows beyond a threshold (i.e., the last data column), even the L2 cache runs out of associative cache lines. Thus, the memory accesses begin to hit the memory or higher level caches, and the latencies increase at significantly higher rates.

Server virtualization has further complicated the addressing uncertainty by adding another layer of indirection to memory addressing. As illustrated in Figure 4.2, the

[3]System B could sustain minimum access latency for up to 18 random accesses, possibly because of a more adaptive (online-learning) prefetch algorithm implemented in newer processors.

"physical memory" of a guest VM is still virtualized, and access to it must be further translated. As a result, it is very difficult, if not impossible, for a process in a guest VM (especially for a full virtualization VM) to discover the actual physical memory addresses of a memory region. Due to the addressing uncertainty, for classic covert channels on virtualized systems, the number of cache regions is reduced to a minimum of only two [77, 101].

### 4.3.2 Scheduling Uncertainty

Classic cache channel data transmission depends on a cache pattern "round-trip"—the receiver completely resets the cache and correctly passes it to the sender; and the sender completely prepares the cache pattern and correctly passes it back to the receiver. Therefore, to successfully transmit one cache pattern, the sender and receiver must be strictly round-robin scheduled.

However, without special scheduling arrangements (i.e., collusion) from the hypervisor, such idealistic scheduling rarely happens. On production virtualized systems, the physical processors are usually oversubscribed in order to increase utilization. In other words, each physical processing core serves more than one virtual processor from different VMs. As a result, there exist many scheduling patterns that prevent successful cache pattern "round-trip", as listed in Table 4.3.

Xu *et al.* [101] have clearly illustrated the problem of scheduling uncertainty in two of their measurements. First, in a laboratory setup, the error rate of their covert channel increases from near 1% to 20–30% after adding a non-participating VM with moderate workload. Second, in the Amazon EC2 cloud, they have discovered that only 10.5% of the cache measurements at the receiver side are valid (correctness not considered), due to the hypervisor's less-than-ideal scheduling.

| Scheduling Pattern | Failure |
|---|---|
| The receiver is de-scheduled before it finishes resetting the cache. | Channel not cleared for send |
| The receiver finishes resetting the cache, but another unrelated VM is scheduled to run immediately after. | Channel invalidated for send |
| The sender is de-scheduled before it finishes preparing the cache. | Sending incomplete |
| The sender finished preparing the cache, and another unrelated VM is scheduled to run immediately after. | Symbol destroyed |
| The receiver is de-scheduled before it finishes reading the cache. | Receiving incomplete |
| The sender and receiver are executed in parallel, on processor cores that share the L2 cache. | Channel operation collision |

Table 4.3: Various Invalid Scheduling Patterns

### 4.3.3 Cache Physical Limitation

Besides the two uncertainties, classic cache channels also face an insurmountable limitation—the necessity of a *shared* and *stable* cache.

If the sender and receiver of classic cache channels are executed on processor cores that do not share any cache, obviously no communication could be established. On a multi-processor system, it is quite common to have processor cores that do not share any cache, since there is usually no shared cache between different physical processors. And sometimes even processor cores residing on the same physical processor do not share cache, such as an Intel Core2 Quad processor in System A, which contains two dual-core silicon packages with no shared cache in between.

Even if the sender and receiver could share cache, external interferences can make the cache unstable. Modern multi-core processors often include a large last-level cache (LLC) shared between all processor cores. To facilitate a simpler cache coherence protocol, the LLC usually employs an inclusive principle, which requires that all

data contained in the lower level caches must also exist in the LLC. Thus, any non-participating processes executing on those processor cores that share the LLC with the sender and receiver can interfere with the communication by indirectly evicting the data in the cache used for the covert channel. The more cores on a processor, the higher the interference.

## 4.4 Covert Channel in the Hyper-Space

Virtualization induced changes to cache operations and process scheduling render the data transmission scheme of classic cache channels obsolete. First, the effectiveness of data modulation is severely reduced by addressing uncertainty. Second, the critical procedures of signal generation, delivery, and detection are frequently interrupted by less-than-ideal scheduling patterns. And finally, the fundamental requirement of stably shared cache tends to be invalidated.

In this section we present our techniques to tackle the existing difficulties, and develop a high-bandwidth, reliable covert channel on virtualized x86 systems. We first describe our redesigned, pure timing-based data transmission scheme, which overcomes the negative effects of addressing and scheduling uncertainties by a simplified design. After that, we detail our findings of a powerful covert channel medium, exploiting the atomic instructions and their induced cache–memory bus interactions on x86 platforms. And finally, we specify our designs of a high error-tolerance transmission protocol for cross–VM covert channels.

### 4.4.1 Redesigning Data Transmission

We first question the reasoning behind using cache state patterns for data modulation. Originally Percival [67] designed this transmission scheme mainly for the use of cryp-

**Algorithm 4.2** Timing-based Cache Channel Protocol

***CLines***: Several sets of associative cache lines picked by both the sender and the receiver; These cache lines can be put in one of two states, *cached* or *flushed.*
$\boldsymbol{D_{Send}[N]}$, $\boldsymbol{D_{Receive}[N]}$: $N$ bit data to transmit and receive, respectively.

**Sender Operations:**

**for** $i := 0$ **to** $N-1$ **do**
  **if** $D_{Send}[i] = 1$ **then**
    **for** an amount of time **do**
      {Put *CLines* into the *flushed* state}
      Access memory maps to *CLines*;
    **end for**
  **else**
    {Leave *CLines* in the *cached* state}
    Sleep of an amount of time;
  **end if**
**end for**

**Receiver Operations:**

**for** $i := 0$ **to** $N-1$ **do**
  **for** an amount of time **do**
    Timed access memory maps to *CLines*;
  **end for**
  {Detect the state of *CLines* by latency}
  **if** $Mean(Acc.Time) > Threshold$ **then**
    $D_{Receive}[i] := 1$; {*CLines* is *flushed*}
  **else**
    $D_{Receive}[i] := 0$; {*CLines* is *cached*}
  **end if**
**end for**

tographic key stealing on a hyper-threaded processor. In this specific usage context, the critical information of memory access patterns are reflected by the states of cache regions. Therefore, cache region-based data modulation is an important source of information. However, in a virtualized environment, the regions of the cache no longer carry useful information due to addressing uncertainty, making cache region-based data modulation a great source of interference.

We therefore redesign a data transmission scheme for the virtualized environment. By removing cache region-based encoding, data is modulated by the state of cache lines over time, resulting in a pure timing-based transmission protocol, as described in Algorithm 4.2.

Besides removing cache region-based data modulation, the new transmission scheme also features a significant change in the scheduling requirement, i.e., signal generation and detection are performed instantaneously, instead of being interleaved. In other words, data are transmitted while the sender and receiver run in parallel. This requirement is more lenient than strict round-robin scheduling, especially with the

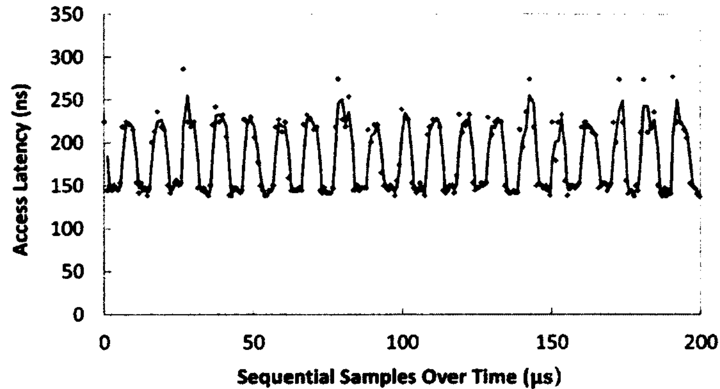



Figure 4.3: Timing-based Cache Channel Bandwidth Test

trend of increasing number of cores on a physical processor, making two VMs more likely to run in parallel than interleaved.

We conduct a simple raw bandwidth estimation experiment to demonstrate the effectiveness of the new cache covert channel. In this experiment, interleaved bits of zeros and ones are transmitted, and the raw bandwidth of the channel can thus be estimated by manually counting the number of bits transmitted over a period of time.

We build the cache covert channel on an Intel Core2 system with two processor cores sharing a 2 MB 8-way set-associative L2 cache. Using a simple profiling test, accessing a random[4] sequence of memory addresses separated by multiples of 256KB, we observe that these memory addresses can be mapped to up to 64 cache lines. Therefore, we select $CLines$ as a set of 64 cache lines mapped by memory addresses following the pattern $M + X \cdot 256K$, where $M$ is a small constant and $X$ is a random positive integer. The sender puts these cache lines into the *flushed* state by accessing a sequence of $CLines$-mapping memory addresses. The receiver times the access latency of another sequence of $CLines$-mapping memory addresses. The length of the receivers access sequence should be smaller than, but not too far away from the cache line set size, for example, 48.

[4]The randomness is introduced to avoid the interference of hardware prefetching.

As shown in Figure 4.3, the x-value of each sample point is the observed memory access latency by the receiver, and the trend line is created by plotting the moving average of two samples. According to the measurement results, 39 bits can be transmitted over a period of 200 micro-seconds, yielding a raw bandwidth of over 190.4 kilobits per second, about five orders of magnitude higher than the previously studied cross–VM cache covert channels.

Having resolved the negative effects of addressing and scheduling uncertainties and achieved a high raw bandwidth, our new cache channel, however, still performs poorly on a system with non-participating workloads. We discover that a stable communication channel could not be established, due to the frequent migration of virtual processors across physical processor cores, which is also observed by Xu *et al.* [101]. The outgrowth of this behavior is that the sender and receiver frequently reside on processor cores that do not share any cache, making our cache channel run into the insurmountable physical limitation just like the classic cache channels.

### 4.4.2 (Re)Discovering the Memory Bus Channel

The prevalence of virtual processor core migration handicaps cache channels in cross–VM covert communication. In order to reliably establish covert channels across processor cores that do not share any cache, a commonly shared and exploitable resource is needed as the communication medium. And the memory bus comes into our sight as we extend our scope beyond the processor cache.

#### 4.4.2.1 Background

Interconnecting the processors and the system main memory, the memory bus is responsible for delivering data between these components. Because contentions on

the memory bus results in a system-wide observable effect of increased memory access latency, a covert channel can be created by programmatically triggering contention on the memory bus. Such a covert channel is called a bus-contention channel.

The bus contention channels have long been studied as a potential security threat for virtual machines on the VAX VMM, on which a number of techniques have been developed [30, 31, 38] to effectively mitigate this threat. However, the x86 platforms we use today are significantly different from the VAX systems, and we suspect similar exploits can be found by probing previously unexplored techniques. Unsurprisingly, by carefully examining the memory related operations of the x86 platform, we have discovered a bus-contention exploit using atomic instructions with exotic operands.

Atomic instructions are special x86 memory manipulation instructions, designed to facilitate multi-processor synchronization, such as implementing mutexes and semaphores —the fundamental building blocks for parallel computation. Memory operations performed by atomic instructions (namely, atomic memory operations) are guaranteed to complete uninterrupted, because accesses to the affected memory regions by other processors or devices are temporarily blocked from execution.

#### 4.4.2.2 Analysis

Atomic memory operations, by their design, generate system-wide observable contentions in the target memory regions they operate on. And this particular feature of atomic memory operations caught our attention. Ideally, contention generated by an atomic memory operation is well bounded, and is only evident when the affected memory region is accessed in parallel. Thus, atomic memory operations are not exploitable for cross–VM covert channels, because VMs normally do not implicitly share physical memory. However, we have found out that the hardware implementations of atomic memory operations do not match the idealistic specification, and memory

contentions caused by atomic memory operations could propagate much further than one might have expected.

Early generations (before Pentium Pro) of x86 processors implement atomic memory operations by using bus lock, a dedicated hardware signal that provides exclusive access of the memory bus to the device who asserts it. While providing a very convenient means to implement atomic memory operations, the sledgehammer-like approach of locking the memory bus results in system-wide memory contention. In addition to being exploitable for covert channels, the bus-locking implementation of atomic memory operations also causes performance and scalability problems.

Modern generations (before Intel Nehalem and AMD K8/K10) of x86 processors improve the implementation of atomic memory operations by significantly reducing the likelihood of memory bus locking. In particular, when an atomic operation is performed on a memory region that can be entirely cached by a cache line, which is a very common case, the corresponding cache line is locked, instead of asserting the memory bus lock [40]. However, on these platforms, atomic memory operations can still be exploited for covert channels, because the triggering conditions for bus-locking are not eliminated. Specifically, when atomic operations are performed on memory regions with an exotic[5] configuration—unaligned addresses that span two cache lines, atomicity cannot be ensured by cache line locking, and bus lock signals are thus asserted.

Remarkable architecture evolutions have taken place in the latest generations (Intel Nehalem and AMD K8/K10) of x86 processors, one of which is the removal of the shared memory bus. On these platforms, instead of having a unified central memory storage for the entire system, the main memory is divided into several pieces, each

[5]The word "exotic" here only means that it is very rare to encounter such an unaligned memory access in modern programs, due to automatic data field alignments by the compilers. However, manually generating such an access pattern is very easy.

assigned to a processor as its local storage. While each processor has direct access to its local memory, it can also access memory assigned to other processors via a high-speed inter-processor link. This non-uniform memory access (NUMA) design eliminates the bottleneck of a single shared memory bus, and greatly improves processor and memory scalability. As a side effect, the removal of the shared memory bus has seemingly invalidated memory bus covert channel techniques at their foundation. Interestingly, however, the atomic memory operation exploit continues to work on the newer platforms, and the reason for this requires a bit more in-depth explanation.

On the latest x86 platforms, normal atomic memory operations (i.e., operating on memory regions that can be cached by a single cache line) are handled by the cache line locking mechanism similar to that of the previous generation processors. However, for exotic atomic memory operations (i.e., operating on cache-line-crossing memory regions), because there is no shared memory bus to lock, the atomicity is achieved by a set of much more complex operations: all processors must coordinate and completely flush in-flight memory transactions that are previously issued. In a sense, exotic atomic memory operations are handled on the newer platform by "emulating" the bus locking behavior of the older platforms. As a result, the effect of memory access delay is still observable, despite the absence of a shared memory bus.

#### 4.4.2.3 Verification

With the memory bus exploit, we can easily build a memory bus covert channel by adapting our timing-based cache transmission scheme with minor modifications, as shown in Algorithm 4.3.

Compared with Algorithm 4.2, there are only two differences in the memory bus channel protocol. First, we substitute the set of cache lines ($CLines$) with the memory bus as the transmission medium. Similar to the cache lines, the memory bus can also

**Algorithm 4.3** Timing-based Memory Bus Channel Protocol

$M_{Exotic}$: An exotic configuration of a memory region that spans two cache lines.
$D_{Send}[N]$, $D_{Recv}[N]$: $N$ bit data to transmit and receive, respectively.

**Sender Operations:**

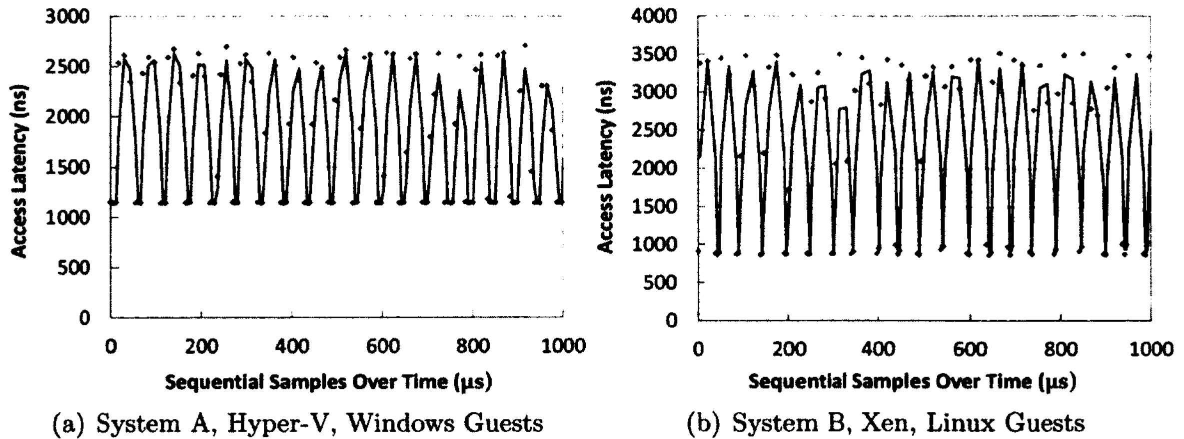
```
for i := 0 to N − 1 do
  if D_Send[i] = 1 then
    for an amount of time do
      {Make memory bus contended}
      Atomic operation with M_Exotic;
    end for
  else
    {Leave memory bus contention-free}
    Sleep of an amount of time;
  end if
end for
```

**Receiver Operations:**

```
for i := 0 to N − 1 do
  for an amount of time do
    Timed uncached memory access;
  end for
  {Detect memory bus state by latency}
  if Mean(Acc.Time) > Threshold then
    D_Recv[i] := 1; {Bus is contended}
  else
    D_Recv[i] := 0; {Bus is contention-free}
  end if
end for
```

be put in two states, *contended* and *contention-free*, depending on whether exotic atomic memory operations are performed. Second, instead of trying to evict contents of the selected cache lines, the sender changes the memory bus status by performing exotic atomic memory operations. And correspondingly, the receiver must make uncached memory accesses to detect contentions.

We demonstrate the effectiveness of the memory bus channel by performing bandwidth estimation experiments, similar to the one in Section 4.4.1, on two systems running different generations of platforms, hypervisors and guest VMs. Specifically, the first system uses an older shared memory bus platform and runs Hyper-V with Windows guest VMs, while the second system utilizes the newer platform without a shared memory bus and runs Xen with Linux guest VMs. Shown in Figures 4.4, the x-value of each sample point is the observed memory access latency by the receiver, and the trend lines are created by plotting the moving average of two samples. According to the measurement results, on both systems, 39 bits can be transmitted over a period of 1 millisecond, yielding a raw bandwidth of over 38 kilobits per second. Although an order of magnitude lower in bandwidth than our cache channel, the memory bus



(a) System A, Hyper-V, Windows Guests (b) System B, Xen, Linux Guests

Figure 4.4: Timing-based Memory Bus Channel Bandwidth Tests

channel enjoys its unique advantage of working across different physical processors. Notably, the same covert channel implementation works on both systems, regardless of the guest operating systems, hypervisors, and hardware platform generations.

### 4.4.3 Enabling Reliable Communication

We have demonstrated that the memory bus channel is capable of achieving high speed data transmission on virtualized systems. However, the preliminary protocol described in Algorithm 4.3 is prone to errors and failures in a realistic environment, because the memory bus is a very noise channel, especially on virtualized systems running many non-participating workloads.

Figure 4.5 presents a realistic memory bus channel sample, taken using a pair of physically co-resident VMs in the Amazon EC2 cloud. From this figure, we can observe that both the "contention free" and "contended" signals are subject to frequent interferences. The "contention free" signals are intermittently disrupted by workloads of other non-participating VMs, causing the memory access latency to moderately raise above the baseline. In contrast, the "contended" signals experience much heavier interferences, which originate from two sources: scheduling and

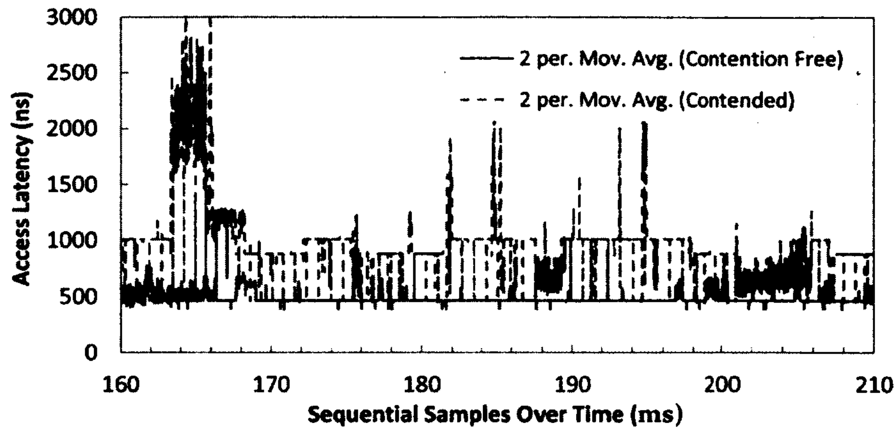


Figure 4.5: Memory Bus Channel Quality Sample on EC2

non-participating workloads. The scheduling interference is responsible for the periodic drop of memory access latency. In particular, context switches temporarily de-schedule the sender process from execution, and thereby briefly relieving memory bus contention. The non-participating workloads executed *in parallel* with the sender process worsen memory bus contention and cause the spikes in the figure, while non-participating workloads executed *concurrently* with the sender process reduce memory bus contention, and result in the dips in the figure. All these interferences can degrade the signal quality in the channel, and make what the receiver observes different from what the sender intends to generate, which leads to *bit-flip* errors.

Besides the observable interferences shown in Figure 4.5, there are also unobservable interferences, i.e., the scheduling interferences to the receiver, which can cause an entirely different phenomenon. When the receiver is de-scheduled from execution, there is no observer in the channel, and thus all data being sent is lost. And to make matters worse, the receiver could not determine the amount of information being lost, because the sender may also be de-scheduled during that time. As a result, the receiver suffers from *random erasure* errors.

We summarize three important issues need to be addressed by the communication protocol in order to ensure reliable cross–VM communication: receiving confirmation, clock synchronization, and error correction.

1. **Receiving Confirmation**

   The *random erasure* errors can make the transmitted data very discontinuous, significantly reducing its usefulness. To alleviate this problem, the sender needs to be aware of whether the data it sent out has been received.

   We avoid using send-and-acknowledge, a commonly employed mechanism for solving this problem, because this mechanism requires the receiver to actively send data back to the sender, reversing the roles of sending and receiving, and subjects the acknowledgment sender (i.e., the data receiver) to the same problem. Instead, we leverage the system-wide effect of memory bus contention to achieve simultaneous data transmission and receiving confirmation. In particular, the receiver signifies its presence to the sender by generating increased memory access latencies on the sender side.

   The corresponding changes to the data transmission protocol include:

   (a) Instead of making uncached memory accesses, the receiver performs exotic atomic memory operations, just like the sender transmitting a one bit.

   (b) Instead of sleeping when transmitting a zero bit, the sender performs uncached memory accesses. In addition, the sender always measures its memory access times.

   (c) While the receiver is in execution, the sender should always observe high memory access latencies; otherwise, the sender can assume the data has been partially lost, and retry at a later time.

2. **Clock Synchronization**

   Because the sender and receiver belong to two independent VMs, scheduling differences between them tend to make the data transmission and detection procedures de-synchronized, which can cause a significant problem to pure timing-based data modulation. We overcome clock de-synchronization by using self-clocking coding—a commonly used technique in telecommunications. Specifically, we choose to transmit data bits using differential Manchester encoding, a standard network coding scheme [99].

3. **Error Correction**

   Even with self-clocking coding, *bit-flip* errors are expected to be common. Similar to resolving the receiving confirmation problem, we again avoid using acknowledgment-based mechanisms. Assuming only a one-way communication channel, we resolve the error correction problems by applying forward error correction (FEC) to the original data, before applying self-clocking coding. More specifically, we use the Reed-Solomon coding [76], a widely applied block FEC code with strong multi-bit error correction performance.

In addition, we strengthen the communication protocol's resilience to clock drifting and scheduling interruption by employing data framing. We break the data into segments of fixed-length bits, and frame each segment with a start-and-stop pattern. The benefits of data framing are twofold. First, when the sender detects transmission interruption, instead of retransmitting the whole piece of data, only the affected data frame is retried. Second, some data will inevitably be lost during transmission. With data framing, the receiver can easily localize the erasure errors and handle them well through the Reed-Solomon coding.

The finalized protocol with all the improvements is presented in Algorithm 4.4.

**Algorithm 4.4** Reliable Timing-based Memory Bus Channel Protocol

$M_{ExoticS}$, $M_{ExoticR}$: Exotic memory regions for the sender and the receiver, respectively.
$D_{Send}$, $D_{Recv}$: Data to transmit and receive, respectively.

```
Sender Prepares D_Send by:
{DM_Send[]: Segmented encoded data to send}
RS_Send := ReedSolomon_Encode(D_Send);
FD_Send[] := Break RS_Send into segments;
DM_Send[] := DiffManchester_Encode(FD_Send[]);

Sending Encoded Data in a Frame:
{Data: A segment of encoded data to send}
{FrmHead, FrmFoot: Unique bit patterns
 signifying start and end of frame, respectively}
Result := SendBits(FrmHead);
if Result is not Aborted then
   Result := SendBits(Data);
   if Result is not Aborted then
      {Ignore error in sending footer}
      SendBits(FrmFoot);
      return Succeed;
   end if
end if
return Retry;

Sending a Block of Bits:
{Block: A block of bits to send}
{Base_1, Base_0: Mean contention-free access
 time for sending bit 1 and 0, respectively}
for each Bit in Block do
   if Bit = 1 then
      for an amount of time do
         Timed atomic operation with M_ExoticS;
      end for
      Latency := Mean(AccessTime) - Base_1;
   else
      for an amount of time do
         Timed uncached memory access;
      end for
      Latency := Mean(AccessTime) - Base_0;
   end if
   if Latency < Threshold then
      {Receiver not running, abort}
      return Aborted;
   end if
end for
return Succeed;

Receiver Recovers D_Recv by:
{DM_Recv[]: Segmented encoded data received}
FD_Recv[] := DiffManchester_Decode(DM_Recv[]);
RS_Recv := Concatenate FD_Recv[];
D_Recv := ReedSolomon_Decode(RS_Recv);

Receiving Encoded Data in a Frame:
{Data: A segment of encoded data to receive}
Wait for frame header;
Result := RecvBits(Data);
if Result is Aborted then
   return Retry;
end if
Result := Match frame footer;
if Result is not Matched then
   {Clock synchronization error, discard Data}
   return Erased;
else
   return Succeed;
end if

Receiving a Block of Bits:
{Block: a block of bits to receive}
for each Bit in Block do
   for an amount of time do
      Timed atomic operation with M_ExoticR;
   end for
   {Detect the state of memory by latency}
   if Mean(AccessTime) > Threshold then
      Bit := 1; {Bus is contended}
   else
      Bit := 0; {Bus is contention-free}
   end if
   {Detect sender de-schedule}
   if too many consecutive 0 or 1 bits then
      {Sender not running}
      Sleep for some time;
      {Sleep makes sender abort, then we abort}

      return Aborted;
   end if
end for
return Succeed;
```

## 4.5 Evaluation

We evaluate the exploitability of memory bus covert channels by implementing the reliable Cross–VM communication protocol, and demonstrate covert channel attacks on our in-house testbed server, as well as on the Amazon EC2 cloud.

### 4.5.1 In-house Experiments

We launch covert channel attacks on system B, a state-of-the-art virtualization server (configurations listed in Table 4.1). The experimental setup is simple and realistic. We create two Linux VMs, namely VM-1 and VM-2, each with a single virtual processor and 512 MB of memory. The covert channel sender runs as an unprivileged user program on VM-1, while the covert channel receiver runs on VM-2, also as an unprivileged user program.

We first conduct a quick profiling to determine suitable parameters for the communication protocol. We find that a data frame size of 32 bits (including an 8 bit preamble), and an error correction strength of 4 parity symbols (bytes) per 4 data bytes works well on the experimental system. Effectively, each data frame consists of 8 bits of preamble, 12 bits of data, and 12 bits of parity, yielding a transmission efficiency of 37.5%. In order to minimize the impact of burst errors, such as multiple frame losses, we group 48 data and parity bytes, and randomly distribute them across 16 data frames using a linear congruential generator (LCG).

We then assess the capacity (i.e., bandwidth and error rate) of the covert channel by performing a series of data transmissions using these parameters. For each transmission, a one kilobyte data block is sent from the sender to the receiver. With 50 repeated transmissions, we observe a stable transmission rate of $781.6 \pm 13.6$ bps. Data errors are observed, but at a very low rate of 0.27%.

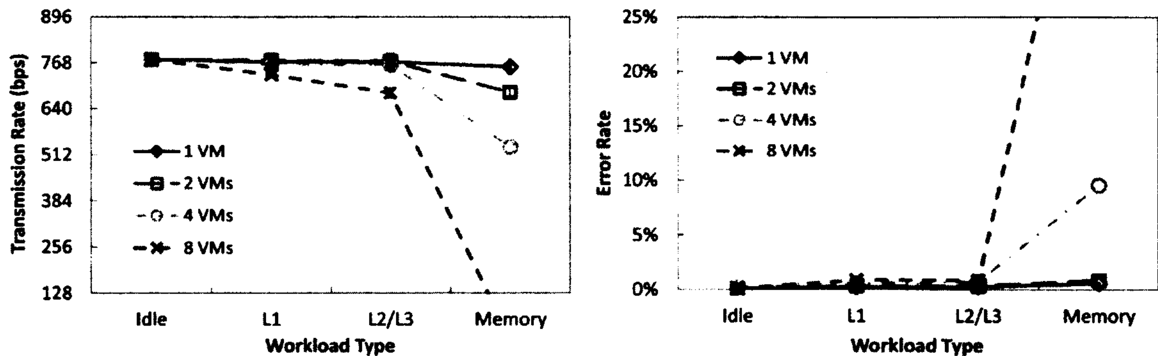



Figure 4.6: Effects of Non-participating Workload on Bandwidth and Error Rate

We further evaluate the impact of covert channel performance by interfering workload, in particular, the workload on the memory sub-system, from non-participating VMs ("other VMs" for short). We define four levels of interferences, *idle*, *L1*, *L2/L3*, and *Memory*, listed in ascending order by the weight of impact to the memory subsystem. The *idle* interference is generated by spawning other VMs and leaving them idle. The *L1* interference is generated by running in the other VMs a program with a tight infinite loop, which only stresses the processor L1 cache due to the very small amount of memory involved in execution. Both *L2/L3* and *Memory* interferences are generated by running `cachebench` [61], a processor cache and memory benchmarking utility: for the *L2/L3* interference, the amount of memory access is limited to the size of the processor L3 cache; and for the *Memory* interference, the amount of memory access is set to be slightly larger than the size of the processor L3 cache.

As shown in Figure 4.6, we measure the bandwidth and error rate of the covert channel when it is subjected each level of interferences generated by up to eight non-participating VMs. We observe that the covert channel is very resilient to *idle*, *L1*, and *L2/L3* interferences. More specifically, while these interferences do exert negative impacts on the covert channel (i.e., decreased bandwidths and increased error rates), the effects are minimal—except for the moderate decrease of bandwidth with

eight VMs running *L2/L3* workload, the bandwidth and error rate reductions in all other cases are negligible. The robustness against cache-based interferences is well expected, since the processor cache is not used as a medium for this covert channel. However, when subjected to *Memory* interferences, the covert channel performances degrades significantly with more than four VMs running non-participating workload. Especially, with eight VMs, no data could be transmitted without error. This dramatic reduction of performance is also well expected, because the memory benchmark program inflicts extreme workload on the memory bus, and thereby rendering this medium unusable for the covert channel. Because normal applications would rarely generate such an intense memory workload for an extended period of time, the memory bus covert channel is still practical in the real world.

### 4.5.2 Amazon EC2 Experiments

We prepare the Amazon EC2 experiments by spawning physically co-hosted Linux VMs. Thanks to the operational experiences presented in [77, 101], using only two accounts, we successfully uncover two pairs of physically co-hosted VMs (micro instances) in four groups of 40 VMs (i.e. each group consists of 20 VMs spawned by each account). Information disclosed in `/proc/cpuinfo` shows that these servers use the shared-memory-bus platform, one generation older than our testbed system B.

Similar to our in-house experiments, we first conduct a quick profiling to determine the suitable protocol parameters for the EC2 system. Compared to our in-house system profiles, memory bus channels on Amazon EC2 VMs have a higher tendency of clock de-synchronization. We compensate for this deficiency by reducing the data frame size to 24 bits. The error correction strength of 4 parity symbols per 4 data bytes still works well. And the overall transmission efficiency thus becomes 33.3%.

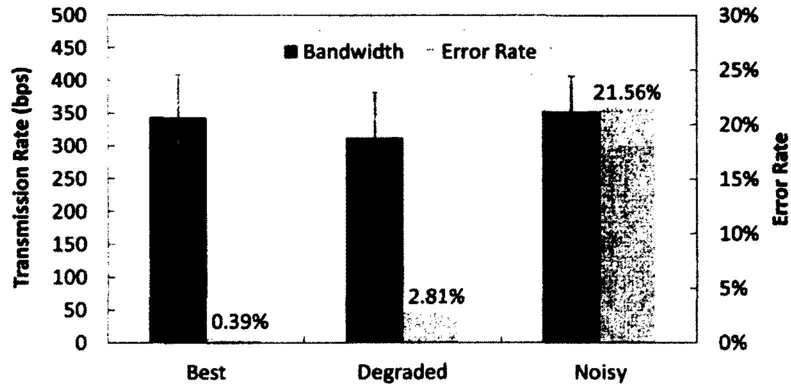


Figure 4.7: Memory Bus Channel Capacities of EC2

We again perform a series of data transmissions and measure the bandwidth and error rates. Our initial results are astonishingly good. A transmission rate of $343.5 \pm 66.1$ bps is achieved, with error rate of 0.39%. However, as we continue to repeat the measurements, we observe an interesting phenomenon. As illustrated in Figure 4.7, three distinct channel performances are observed through our experiment. The best performance is achieved during the initial 12–15 transmissions. After that, for the next 5–8 transmissions, the performance degrades. The bandwidth slightly reduces, and the error rate slightly increases. Finally, for the rest of the transmissions, the performance becomes very bad. While the bandwidth is still comparable to that of the best performance, the error rate becomes unacceptably high.

By repeating this experiment, we uncover that the three-staged behavior can be repeatedly observed after leaving both VMs idle for a long period of time (e.g., one hour). Therefore, we believe that the cause of this behavior can be explained by scheduler preemption [100] as discussed in [101]. During the initial transmissions, the virtual processors of VMs at both the sender and receiver sides have high scheduling priorities, and thus they are very likely to be executed in parallel, resulting in a very high channel performance. Then, the sender VM's virtual processor consumes all its scheduling credits and is throttled back by the Xen scheduler, causing the channel

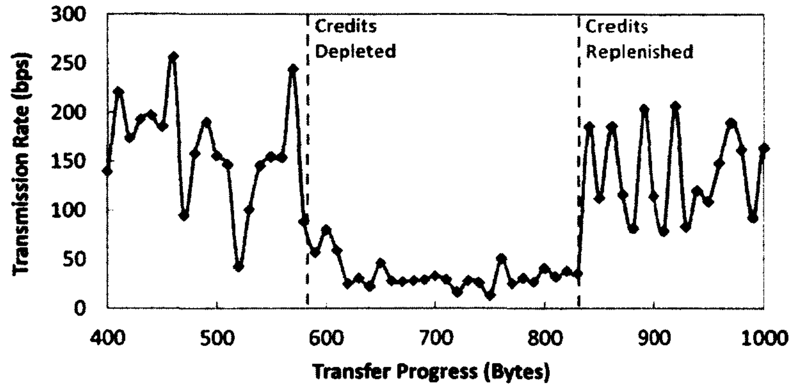



Figure 4.8: Reliable Transmission with Adaptive Rates

performance to degrade. Soon after that, the receiver VM's virtual processor also uses up its scheduling credits. Since both the sender and receiver are throttled back, their communication is heavily interrupted. This "offensive" scheduling pattern subjects the communication channel to heavy random erasure beyond the correction capability of the FEC.

Fortunately, our communication protocol is designed to handle very unreliable channels. We adapt to the scheduler preemption by tuning two parameters to be more "defensive". First, we increase the ratio of parity bits to 4 parity symbols per 2 data bytes. Although reducing transmission efficiency by 11.1%, the error correction capability of our FEC is increased by 33.3%. Second, we reduce the transmission symbol rate by about 20%. By lengthening the duration of the receiving confirmation, we effectively increase the probability of discovering scheduling interruptions. After the parameter adjustment, we can achieve a transmission rate of 107.9 ± 39.9 bps, with an error rate of 0.75%, even under scheduler preemption.

Figure 4.8 shows the adjusted communication protocol in action. During the first period of preemption-free scheduling, the transmission rate can be as high as 250 bps. However, when preemption starts, the sender responds to frequent transmission failures with increased retries, allowing the receiver continue to receive and

decode data without uncorrectable error. And correspondingly, the transmission rate drops to below 50 bps. Finally, when the harsh scheduling condition is alleviated, the transmission rate is automatically restored. The capability of adaptively adjusting transmission rates to channel conditions, evidences the versatility of our reliable communication protocol.

## 4.6 Discussion

In this section, we first reassess the threat of covert channel attacks based on our experimental results. Then, we discuss possible means to mitigate the covert channel attacks in virtualized environments.

### 4.6.1 Damage Assessment

Due to their very low channel capacities [77, 101], previous studies conclude that covert channels can only cause very limited harms in a virtualized environment. However, the experimental results of our covert channel lead us to a different conclusion that covert channels indeed pose realistic and significant threats to information security in the cloud.

With over 100 bits-per-second high speed and reliable transmission, covert channel attacks can be applied to a wide range of mass-data theft attacks. For example, a hundred byte credit card data entry can be silently stolen in less than 30 seconds; a thousand byte private key file can be secretly transmitted under 3 minutes. Working continuously, over 1 MB of data, equivalent to tens of thousands of credit card entries or hundreds of private key files, can be trafficked every 24 hours. In addition to high channel capacity, memory bus covert channel has two other intriguing properties:

1. **Stealthiness:** Because processor cache is not used as channel medium, memory bus covert channel incurs negligible impact on cache performance, making it transparent to cache based covert channel detections, such as HomeAlone [105].

2. **"Future proof":** Our in-house experiment shows that even on a platform that is one generation ahead of Amazon EC2's systems, memory bus covert channel continues to perform very well.

### 4.6.2 Mitigation Techniques

Realistic threat of covert channel attacks calls for effective and practical countermeasures. We discuss several plausible mitigation approaches from three different perspectives—tenants, cloud providers, and device manufactures.

#### 4.6.2.1 Tenant Mitigation

Mitigating covert channels on the tenant side enjoys the advantages of trust and deployment flexibility. With the implementation of mitigation techniques inside tenant owned VMs, the tenant has the confidence of covert channel security, regardless whether the cloud provider addresses this issue.

However, due to the lack of lower level (hypervisor and/or hardware) support, the available options are very limited, and the best choice is performance anomaly detection. Because memory bus covert channels result in observable memory performance degradation, an approach similar to that of HomeAlone [105] may be taken. In particular, the defender continuously monitors memory access latencies, and asserts alarms if significant anomalies are detected. However, since memory accesses incur much higher cost and non-determinism than cache probing, this approach may suffer from high performance overhead and high false positive rate.

#### 4.6.2.2 Cloud Provider Mitigation

Compared to their tenants, cloud providers are much more resourceful. They control not only the hypervisor and hardware platform on a single system, but also the entire network and systems in a data center. As a result, cloud providers can tackle covert channels through either preventative or detective countermeasures.

The preventative approaches, e.g., the dedicated instances service provided by the Amazon EC2 cloud [1], thwart covert channel attacks by eliminating the exploiting factors of covert channels. While the significant extra service charge of the dedicated instances service reduces its attractiveness, the "no-sharing" guarantee may be too strong for covert channel mitigation. We envision a low cost alternative solution that allows tenants to share system resources in a controlled and deterministic manner. For example, the cloud provider may define a policy that each server might be shared by up to two tenants, and each tenant could only have a predetermined neighbor. Although this solution does not eliminate covert channels, it makes attacking arbitrary tenants in the cloud very difficult.

In addition to preventative countermeasures, cloud providers can easily take the detective approach by implementing low overhead detection mechanisms, because of their convenient access to the hypervisor and platform hardware. For both cache and memory bus covert channels, being able to generate observable performance anomalies is the key to their success in data transmission. However, modern processors have provided a comprehensive set of mechanisms to monitor and discover performance anomalies with very low overhead. Instead of actively probing cache or accessing memory, cloud providers can leverage the hypervisor to infer the presence of covert channels, by keeping track of the increment rates of the cache miss counters or memory bus lock counters [40]. Moreover, when suspicious activities are detected,

cloud providers can gracefully resolve the potential threat by migrating suspicious VMs onto physically isolated servers. Without penalizing either the suspect or the potential victims, the negative effects of false positives are minimized.

#### 4.6.2.3 Device Manufacture Mitigation

The defense approaches of both tenant and cloud providers are only secondary in comparison to mitigation by the device manufactures, because the root causes of the covert channels are imperfect isolation of the hardware resources.

The countermeasures at the device manufacture side are mainly preventative, and they come in various forms of resource isolation improvements. For example, instead of handling exotic atomic memory operations in hardware and causing system-wide performance degradation, the processor may be redesigned to trap these rare situations for the operating systems or hypervisors to handle, without disrupting the entire system. A more general solution is to tag all resource requests from guest VMs, enabling the hardware to differentiate requests by their owner VMs, and thereby limiting the scope of any performance impact.

## 4.7 Summary

Covert channel attacks in the cloud have been proposed and studied. However, the threats of covert channels tend to be down-played or disregarded, due to the low achievable channel capacities reported by previous research. In this chapter, we presented a novel construction of high-bandwidth and reliable cross–VM covert channels on the virtualized x86 platform.

By studying existing cache channel techniques, we uncovered their application insufficiency and limitations in a virtualized environment. We then resolved these

obstacles by designing a pure timing-based data transmission scheme, and discovering the bus locking mechanism as a powerful covert channel medium. Leveraging the memory bus covert channel, we further designed a robust data transmission protocol, and demonstrated the real-world exploitability of our covert channel by launching attacks on our testbed system and in the Amazon EC2 cloud. Our experimental results show that, contrary to previous research and common beliefs, covert channel attacks in a virtualized environment can achieve high bandwidth and reliable transmission. As a result, covert channels pose formidable threats to information security in the cloud, and they must be carefully analyzed and mitigated.

# Chapter 5

# Conclusion

Vulnerability research is essential to the protection of computer system security. In particular, the discovery of a new vulnerability on a system represents a strategic vantage point that both the malicious attackers and security providers strive to hold. This dissertation presented the research and discovery of new vulnerabilities in the design and deployment of complex systems, which pose higher security hazard compared to the more commonly seen implementation-caused vulnerabilities.

Three types of computer systems have been studied—automatic malicious binary detection systems, server power management systems, and server virtualization systems. Although very different from each other, these systems play important roles in our everyday life, from personal computing to the information technology infrastructure. By uncovering vulnerability with potentials of serious consequences from each type of systems, this dissertation revealed the prevalence of security vulnerabilities, and demonstrated effective approaches to tackle them.

The following sections first summarize the contribution to knowledge of these studies, and then discuss the outlook of the future security research.

## 5.1 Contribution to Knowledge

Automatic malicious binary detection systems are critical for the defense against malware, preventing its fast spreading and protecting computer systems from its exploitation. We studied and identified a design flaw in commonly seen design patterns of automatic malicious binary detection systems, leveraging the weaknesses of static analysis against obfuscation evasion attacks. Inspired by a steganographic technique, we designed and implemented a novel binary obfuscation technique, mimimorphism, which enables malware to evade a wide range of static analysis detections. Mimimorphism transforms a binary executable into a mimicry executable, with statistical and semantic characteristics highly similar to those of the mimicry target.

Power management plays a central role in server system power saving. Recent industrial progresses in energy proportional computing have significantly improved server system energy efficiency. However, we discovered that the significant changes of power profile on server systems are not matched with comparable modifications in the security considerations, leaving the server systems vulnerable to energy abuses. We demonstrated a realistic energy attack on a standalone web server system. Leveraging the knowledge of Web request servicing energy profile as well as human Web browsing behaviors, we designed a stealthy energy attack that significantly increases the power consumption of victim servers under typical workloads. We further proposed an application-oriented defense approach that works around the hardware limitations and effectively protects victim servers against energy attacks.

Server virtualization technologies are heavily deployed in data centers today, providing the benefit of workload consolidation and simplified resource management. However, the co-hosting of VMs introduces covert channel attack vulnerabilities in shared computing utilities, such as public clouds. We presented a novel covert channel

attack that achieved orders-of-magnitude higher bandwidth than previous research. Based on our in-depth study of x86 processor cache and memory architecture, we designed a pure timing based covert channel, exploiting the shared memory bus to transmit information across physical processors. We also implemented a robust data transmission protocol which ensures over 99% reliability in data transmission in realistic environments. In addition, we contributed our insights on mitigating of cross–VM covert channel attacks in virtualized systems.

## 5.2 Future Research

Computer system security is not a static term. While the cyberspace warfare between malicious attackers and security defenders is ever-lasting, their battle fields, in time, shift from one target to another, following the dynamic trends of industrial and personal computing.

There is no "silver bullet" for general computer system security protection, because each type of system has its own complexity and uniqueness, as exemplified by the studies presented in this dissertation. And the best practice to ensure security, as alluded in the introduction chapter of this dissertation, is thus to be ahead of the attackers—study the emerging systems, discover and mitigate their vulnerabilities before they fall into the offensive parties' hands. In the following, we discuss two promised fields for the future security research.

### 5.2.1 Data Center and Cloud Computing

As the trend of industrial computing quickly shifts toward high consolidation and rapid scaling, research on privacy and security in data center and cloud computing has been gathering momentum.

Covert channel attacks pose realistic and severe privacy threats on virtualized platforms, as shown in Chapter 4 of this dissertation. And thus providing practical and effective safeguard for their tenants becomes critical for cloud vendors, because ensuring covert channel security gives them a clear edge over their competitors. Existing solutions, such as Amazon Dedicated Instances, are quite limited in terms of affordability and level of protection. However, there is a wide range of alternative avenues waiting to be explored. For an example, a time-slotted deterministic VM placement algorithm can be developed. Instead of completely eliminating covert channel attacks, this approach makes it very difficult to launch such attacks against a specific victim VM, while enables better resource pooling and thereby lowering tenant costs. For another example, a processor modification can be implemented, which enables dynamic hardware resource partitioning. While incurring high cost in hardware upgrades, this approach results in a cloud provider and tenant transparent countermeasure, and lowers the performance penalties and overall cost.

Besides privacy concerns, data center and cloud are also facing other security challenges. Driven by the ever-increasing bandwidth and computation demands, the network and computing equipments in data centers are expanding in quantity and variety. On one hand, the massive types and amount of hardware deployed in a data center call for scalable solutions to discover, diagnose and even automatically mitigate security problems. On the other hand, the constant evolution of data center networks and equipments demands solutions to monitor and maintain the data center security in a flexible manner.

Moreover, as the industry adapts to cloud computing, software and service providers strive to migrate existing applications to the cloud. Techniques and guidelines to assist developers and administrators to develop and deploy cloud applications in a secure, scalable and seamless fashion are highly desired.

### 5.2.2 Mobile System Security

The never before seen popularity of smart phones brings personal computing into a new chapter. Portable computing devices and high-speed cellular networks provide us with great convenience of location-free information access and communication. However, just like any other technologies, the double-edged sword of mobile computing exposes us to new risks and threats.

Functioning as personal information central, smart phones gather and harbor a rich set of data about its owner, from online activities to social interactions, and to physical identities, and even to financial information. As a result, these information "gold mines" are on the trend of superseding personal computers and becoming prime targets for cyber attacks. However, the security of smart phones is no stronger than that of personal computers. In particular, to facilitate rich and customizable functionalities, smart phone designs incorporate user programmable applications, called "apps". And this design choice makes smart phones vulnerable to malicious software, just like personal computers. Adding insult to injury, due to the limited computation power and energy reserve, hosting client-side malware protections on smart phones is impractical or highly undesirable, making malware protection on a mobile system particularly challenging.

Currently some device manufactures employ manual vetting to screen untrusted, potentially malicious "apps". However, the number of new "apps" increases by thousands daily and is still on the rise, making manual vetting impractical in the near future. An alternative solution is to host automated "app" vetting on the service provider side, harvesting vast amount of computation power and energy reserves from the data center and cloud. However, provider-side malware detections are performed without real life usage context and thus may suffer degraded accuracy. In addi-

tion, malware writers can craft targeted evasion techniques, such as device emulation and/or human user detection. Future security solutions for mobile system should bridge the provider-side and device-side malware defenses, allowing these techniques to complement their weaknesses, and thereby achieve robust and efficient protection.

# References

[1] Amazon Web Services. Amazon EC2 dedicated instances. `http://aws.amazon.com/dedicated-instances/`.

[2] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In *Proceedings of the 21st Annual International Cryptology Conference (CRYPTO)*, August 2001.

[3] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 151–160, 1998.

[4] L. A. Barroso. The price of performance. *ACM Queue*, 3(7):48–53, September 2005.

[5] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *IEEE Computer*, 40(12):33–37, Dec. 2007.

[6] R. Bianchini and R. Rajamony. Power and energy management for server systems. *IEEE Computer*, 37(11):68–74, Nov. 2004.

[7] C. Bryan-Low. Cybercrime costs mount in U.K. `http://online.wsj.com/article/SB10001424052748703561604576150353058208060.html`.

[8] T. K. Buennemeyer, M. Gora, R. C. Marchany, and J. G. Tront. Battery exhaustion attack detection with small handheld mobile computers. In *Proceedings of the IEEE International Conference on Portable Information Devices (PORTABLE)*, 2007.

[9] S. Cabuk, C. E. Brodley, and C. Shields. IP covert timing channels: design

and detection. In *Proceedings of the 11th ACM conference on Computer and communications security (CCS'04)*, pages 178–187, 2004.

[10] X. Cai, Y. Gui, and R. Johnson. Exploiting unix file-system races via algorithmic complexity attacks. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, May 2009.

[11] E. V. Carrera, E. Pinheiro, and R. Bianchini. Conserving disk energy in network servers. In *Proceedings of the 17th annual international conference on Supercomputing (ICS)*, pages 86–97, 2003.

[12] J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle. Managing energy and server resources in hosting centers. In *Proceedings of the 18th ACM symposium on operating systems principles (SOSP)*, pages 103–116, 2001.

[13] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (S&P)*, May 2005.

[14] J. R. Crandall, Z. Su, S. F. Wu, and F. T. Chong. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, pages 235–248, 2005.

[15] S. A. Crosby and D. S. Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th conference on USENIX Security Symposium*, 2003.

[16] D. Dagon, T. Martin, and T. Starner. Mobile phones as computing devices: The viruses are coming! *IEEE Pervasive Computing*, 3(4):11–15, Oct.–Dec. 2004.

[17] S. Debray. Code compression. In *Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages*, 2005.

[18] S. Debray and W. Evans. Profile-guided code compression. In *Proceedings of SIGPLAN '02 Conference on Programming Language Design and Implementation (PLDI'02)*, pages 95–105, 2002.

[19] Department of Defense. TCSEC: Trusted computer system evaluation criteria. Technical Report 5200.28-STD, U.S. Department of Defense, Dec 1985.

[20] T. Detristan, T. Ulenspiegel, Y. Malcom, and M. Underduk. Polymorphic shellcode engine using spectrum analysis. *Phrack Issue 0x3d*, 2003.

[21] T. E. Dube, B. D. Birrer, R. A. Raines, R. O. Baldwin, B. E. Mullins, R. W. Bennington, and C. E. Reuter. Hindering reverse engineering: Thinking outside the box. *IEEE Security and Privacy*, 6(2):58–65, 2008.

[22] M. Elnozahy, M. Kistler, and R. Rajamony. Energy-efficient server clusters. In *Proceedings of the 2nd Workshop on Power-Aware Computing Systems*, pages 179–196, 2002.

[23] M. Elnozahy, M. Kistler, and R. Rajamony. Energy conservation policies for web servers. In *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems (USITS)*, 2003.

[24] X. Fan, W.-D. Weber, and L. A. Barroso. Power provisioning for a warehouse-sized computer. In *Proceedings of the 34th annual international symposium on Computer architecture (ISCA)*, pages 13–23, 2007.

[25] W. Felter, K. Rajamani, T. Keller, and C. Rusu. A performance-conserving approach for reducing peak power consumption in server systems. In *Proceedings of the 19th annual international conference on Supercomputing (ICS)*, pages 293–302, 2005.

[26] P. Fogla and W. Lee. Evading network anomaly detection systems: formal reasoning and practical techniques. In *Proceedings of the 13th ACM conference on Computer and communications security (CCS)*, pages 59–68, 2006.

[27] P. Fogla, M. Sharif, R. Perdisci, O. Kolesnikov, and W. Lee. Polymorphic blending attacks. In *Proceedings of the 15th USENIX Security Symposium*, July 2006.

[28] J. Giffin, S. Jha, and B. Miller. Automated discovery of mimicry attacks. In *Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2006.

[29] D. Goldman. Cybercrime: A secret underground economy. `http://money.cnn.com/2009/09/16/technology/cybercrime/index.htm`.

[30] J. W. Gray III. On introducing noise into the bus-contention channel. In *Proceedings of the 1993 IEEE Symposium on Security and Privacy (S&P'93)*, pages 90–, 1993.

[31] J. W. Gray III. Countermeasures and tradeoffs for a class of covert timing channels. Technical report, Hong Kong University of Science and Technology, 1994.

[32] M. V. Gundy, D. Balzarotti, and G. Vigna. Catch me, if you can: Evading network signatures with web-based polymorphic worms. In *Proceedings of 1st USENIX Workshop on Offensive Technologies*, August 2007.

[33] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Franke. DRPM: Dynamic speed control for power management in server class disks. In *Proceedings of the 30th annual international symposium on Computer architecture (ISCA)*, pages 169–182, 2003.

[34] J. Hamilton. Where does the power go and what to do about it? In *Proceedings of the USENIX Workshop on Power Aware Computing and Systems (HotPower)*, December 2008.

[35] T. Heath, B. Diniz, E. V. Carrera, W. M. Jr., and R. Bianchini. Energy conservation in heterogeneous server clusters. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, pages 186–195, 2005.

[36] Hewlett-Packard, Intel, Microsoft, Phoenix, and Toshiba. Advanced configuration and power interface. `http://www.acpi.info`.

[37] T. Horvath, T. Abdelzaher, K. Skadron, and X. Liu. Dynamic voltage scaling in multitier web servers with end-to-end delay control. *IEEE Trans. Comput.*, 56(4):444–458, 2007.

[38] W. Hu. Reducing timing charmers with fuzzy time. *Proceedings of the 1991 IEEE Symposium on Security and Privacy (S&P'91)*, page 8, 1991.

[39] W. Hu. Lattice scheduling and covert channels. In *Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy, 1992*, pages 52 -61, may 1992.

[40] Intel. The Intel 64 and IA-32 architectures software developer's manual. `http://www.intel.com/products/processor/manuals/`.

[41] Intel. *Intel 6400/6402 Advanced Memory Buffer: Thermal/Mechanical Design Guide*, December 2006.

[42] S. Kandula, D. Katabi, M. Jacob, and A. Berger. Botz-4-Sale: Surviving organized DDoS attacks that mimic flash crowds. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, May 2005.

[43] M. Khafir. Trident polymorphic engine. `http://vx.netlux.org/lib/vx.php?id=et06`.

[44] H. Kim, J. Smith, and K. G. Shin. Detecting energy-greedy anomalies and mobile malware variants. In *Proceedings of the 6th international conference on Mobile systems, applications, and services (MobiSys)*, pages 239–252, June 2008.

[45] H.-A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of 13th USENIX Security Symposium*, 2004.

[46] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Detecting malicious code by model checking. In *Proceedings of the 2nd International Conference Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 174–187, July 2005.

[47] J. Kong, O. Aciicmez, J.-P. Seifert, and H. Zhou. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *Proceedings of the 15th IEEE International Symposium on High Performance Computer Architecture 2009 (HPCA'09)*, pages 393–404, feb. 2009.

[48] C. Kreibich and J. Crowcroft. Honeycomb: creating intrusion detection signatures using honeypots. In *Proceedings of 2nd Workshop on Hot Topics in Networks (Hotnets-II)*, 2003.

[49] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *Proceedings of the 14th USENIX Security Symposium*, 2005.

[50] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2005.

[51] C. Kruegel, W. K. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

[52] B. W. Lampson. A note on the confinement problem. *Commun. ACM*, 16:613–615, October 1973.

[53] Z. Li, M. Sanghi, B. Chavez, Y. Chen, and M.-Y. Kao. Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P)*, May 2006.

[54] R. Lyda and J. Hamrock. Using entropy analysis to find encrypted and packed malware. *IEEE Security and Privacy*, 5(2):40–45, 2007.

[55] S. Macaulay. Admmutate: Polymorphic shellcode engine. `http://www.ktwo.ca/security.html`.

[56] T. Martin, M. Hsiao, D. Ha, and J. Krishnaswami. Denial-of-service attacks on battery-powered mobile computers. In *Proceedings of the 2nd IEEE International Conference on Pervasive Computing and Communications (PerCom)*, 2004.

[57] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[58] F. G. G. Meade. A guide to understanding covert channel analysis of trusted systems. Manual NCSC-TG-030, U.S. National Computer Security Center, Nov 1993.

[59] D. Meisner, B. T. Gold, and T. F. Wenisch. PowerNap: eliminating server idle power. In *Proceedings of the 14th ACM ASPLOS*, pages 205–216, March 2009.

[60] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC)*, pages 421–430, 2007.

[61] P. J. Mucci, K. London, and P. J. Mucci. The CacheBench report. Technical report, Nichols Research Corporation, 1998.

[62] R. Nathuji and K. Schwan. VirtualPower: coordinated power management in virtualized enterprise systems. In *Proceedings of the 21st ACM SIGOPS symposium on Operating Systems Principles (SOSP)*, pages 265–278, 2007.

[63] R. Neugebauer and D. McAuley. Energy is just another resource: Energy accounting and energy pricing in the nemesis os. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HOTOS)*, 2001.

[64] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (S&P)*, May 2005.

[65] J. Newsome, B. Karp, and D. X. Song. Paragraph: Thwarting signature learning by training maliciously. In *Proceedings of the 9th International Symposium on Recent Advanced in Intrusion Detection (RAID)*, September 2006.

[66] C. Parampalli, R. Sekar, and R. Johnson. A practical mimicry attack against powerful system-call monitors. In *Proceedings of the 2007 ACM Symposium on Information, Computer and Communications Security (ASIACCS'07)*, 2007.

[67] C. Percival. Cache missing for fun and profit. In *Proceedings of the BSDCan 2005*, 2005.

[68] R. Perdisci, D. Dagon, W. Lee, P. Fogla, and M. Sharif. Misleading worm signature generators using deliberate noise injection. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P)*, May 2006.

[69] R. Perdisci, G. Gu, and W. Lee. Using an ensemble of one-class SVM classifiers to harden payload-based anomaly detection systems. In *Proceedings of the Sixth International Conference on Data Mining*, pages 488–498, 2006.

[70] F. Perriot, P. Ferrie, and P. Szor. Striking similarities: Win32/Simile. http://securityresponse.symantec.com/avcenter/reference/simile.pdf.

[71] E. Pinheiro, R. Bianchini, E. V. Carrera, and T. Heath. *Dynamic cluster reconfiguration for power and performance*, pages 75–93. Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[72] M. Pirretti, S. Zhu, V. Narayanan, P. Mcdaniel, and M. K. The sleep deprivation attack in sensor networks: analysis and methods of defense. In *Proceedings of the ICA DSN 2005*, 2005.

[73] B. R. Racic, D. Ma, and H. Chen. Exploiting MMS vulnerabilities to stealthily exhaust mobile phone's battery. In *Proceedings of the 2nd International Conference on Security and Privacy in Communication Networks (SecureComm)*, pages 1–10, September 2006.

[74] K. Rajamani and C. Lefurgy. On evaluating request-distribution schemes for saving energy in server clusters. In *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 111–122, 2003.

[75] S. Ranjan, R. Swaminathan, M. Uysal, and E. Knightly. DDoS-resilient scheduling to counter application layer attacks under imperfect detection. In *Proceedings of the 25th IEEE Conference on Computer Communications (INFOCOM)*, Apr. 2006.

[76] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[77] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security (CCS'09)*, pages 199–212, 2009.

[78] C. H. Rowland. Covert channels in the TCP/IP protocol suite. *First Monday*, 2, 1997.

[79] Seagate. Barracuda es.2 data sheet. http://www.seagate.com/docs/pdf/datasheet/disc/ds_barracuda_es_2.pdf.

[80] Seagate. Cheetah 15k.6 data sheet. `http://www.seagate.com/docs/pdf/datasheet/disc/ds_cheetah_15k_6.pdf`.

[81] G. Shah and M. Blaze. Covert channels through external interference. In *Proceedings of the 3rd USENIX conference on Offensive technologies*, WOOT'09, pages 3–3, Berkeley, CA, USA, 2009. USENIX Association.

[82] G. Shah, A. Molina, and M. Blaze. Keyboards and covert channels. In *Proceedings of the 15th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2006. USENIX Association.

[83] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *Proceedings of the 6th ACM/USENIX Symposium on Operating System Design and Implementation (OSDI)*, December 2004.

[84] P. Szor. *The Art of Computer Virus Research and Defense.* Symantec Press, 2005.

[85] K. M. C. Tan, K. S. Killourhy, and R. A. Maxion. Undermining an anomaly-based intrusion detection system using common exploits. In *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2002.

[86] Y. Tang and S. Chen. Defending against internet worms: a signature-based approach. In *Proceedings of the 24th INFOCOM*, March 2005.

[87] U.S. Environmental Protection Agency. Report to congress on server and data center energy efficiency, 2007.

[88] U.S. Environmental Protection Agency. The ENERGY STAR version 5.0 specification for computers, 2008.

[89] S. Venkataraman, A. Blum, and D. Song. Limits of learning-based signature generation with adversaries. In *Proceedings of the 15th Annual Network and Distributed Systems Security Symposium (NDSS)*, February 2008.

[90] T. V. Vleck. Timing channels. Poster session, IEEE TCSP conference, May 1990.

[91] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM conference on Computer and communications security (CCS)*, pages 255–264, 2002.

[92] H. Wang, C. Jin, and K. G. Shin. Defense against spoofed ip traffic using hop-count filtering. *IEEE/ACM Transactions on Networking*, 15(1), Feb. 2007.

[93] K. Wang, J. J. Parekh, and S. J. Stolfo. Anagram: A content anomaly detector resistant to mimicry attack. In *Proceedings of the 9th International Symposium on Recent Advanced in Intrusion Detection (RAID)*, September 2006.

[94] K. Wang and S. Stolfo. Anomalous payload-based network intrusion detection. In *Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2004.

[95] Z. Wang and R. B. Lee. Covert and side channels due to processor architecture. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 473–482, 2006.

[96] Z. Wang and R. B. Lee. A novel cache architecture with enhanced performance and security. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture (MICRO'41)*, pages 83–93, 2008.

[97] Watts up? Watts up? .Net digital power meter. `https://www.wattsupmeters.com/secure/products.php?pn=0`.

[98] P. Wayner. Mimic functions. *Cryptologia*, 16(3):193–214, July 1992.

[99] J. Winkler and J. Munn. Standards and architecture for token-ring local area networks. In *Proceedings of 1986 ACM Fall joint computer conference*, ACM '86, pages 479–488, 1986.

[100] XenSource. Xen credit scheduler. `http://wiki.xensource.com/xenwiki/CreditScheduler`.

[101] Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting. An exploration of L2 cache covert channels in virtualized environments. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop (CCSW'11)*, pages 29–40, 2011.

[102] V. Yegneswaran, J. T. Giffin, P. Barford, and S. Jha. An architecture for generating semantics-aware signatures. In *Proceedings of the 14th USENIX Security Symposium*, 2005.

[103] Z0mbie. Automated reverse engineering: Mistfall engine. `http://vx.netlux.org/lib/vzo21.html`.

[104] Q. Zhang and D. S. Reeves. MetaAware: Identifying metamorphic malware. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC)*, pages 411–420, 2007.

[105] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter. HomeAlone: Co-residency detection in the cloud via side-channel analysis. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy (S&P'11)*, May 2011.

# VITA

Zhenyu Wu was born in Chengdu, Sichuan, China, on August 3, 1982, the son of Jian Wu and An Yang. Graduated from High School attached to Sichuan Normal University in Chengdu, Sichuan, China, he entered Denison University in Ohio, USA in August 2001, where he received the degrees of Bachelor of Science in Computer Science, and Bachelor of Arts in Physics, in May 2005. He proceeded with graduate studies in the Department of Computer Science at the College of William and Mary in Virginia, in August 2005. He received a Master of Science degree in Computer Science from the College of William and Mary in Virginia, in May 2007.

This dissertation was defended on April 13, 2012 at the College of William and Mary in Virginia.