Embedded System Security: A Software-based Approach

Ang Cui

Submitted in partial fulfillment of the

requirements for the degree

of Doctor of Philosophy

in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2016

©2016
Ang Cui
All Rights Reserved

ABSTRACT

Embedded System Security: A Software-based Approach

Ang Cui

We present a body of work aimed at understanding and improving the security posture of embedded devices. We present results from several large-scale studies that measured the quantity and distribution of exploitable vulnerabilities within embedded devices in the world. We propose two host-based software defense techniques, Symbiote and Autotomic Binary Structure Randomization, that can be practically deployed to a wide spectrum of embedded devices in use today. These defenses are designed to overcome major challenges of securing legacy embedded devices. To be specific, our proposed algorithms are software-based solutions that operate at the firmware binary level. They do not require source-code, are agnostic to the operating-system environment of the devices they protect, and can work on all major ISAs like MIPS, ARM, PowerPC and X86. More importantly, our proposed defenses are capable of augmenting the functionality of embedded devices with a plethora of host-based defenses like dynamic firmware integrity attestation, binary structure randomization of code and data, and anomaly-based malcode detection. Furthermore, we demonstrate the safety and efficacy of the proposed defenses by applying them to a wide range of real-time embedded devices like enterprise networking equipment, telecommunication appliances and other commercial devices like network-based printers and IP phones. Lastly, we present a survey of promising directions for future research in the area of embedded security.

Table of Contents

List of Figures vi

List of Tables xii

I Introduction 1

1 Motivation 2

1.1 What is an embedded device? 4

2 Hypothesis 5

3 Contributions 6

4 Related Work 8

4.1 Quantification and Qualification 8

4.1.1 Large-Scale Evaluation of Firmware Vulnerabilities 8

4.1.2 Embedded Exploitation in the Wild 9

4.2 Embedded Exploitation 10

4.2.1 Cisco Router Exploitation 11

4.3 Embedded System Defense Technologies 11

4.4 Software Compaction 13

4.5 Defensive Randomization 14

5 Problem Description 16

5.1 Real-world quantification of embedded system vulnerability and exploitability 16
5.2 Generalized firmware analysis and modification 16
5.3 Generalized software environment agnostic defense 17

II The Embedded Threatscape 19

6 Quantitative Assessment of Real-World Embedded Vulnerability 20
6.1 Real-world quantification of embedded device vulnerability 20
6.2 Analysis of Results 26
6.2.1 Breakdown of Vulnerable Devices by Functional Categories 27
6.2.2 Breakdown of Vulnerable Devices by Geographical Location 29
6.2.3 Breakdown of Vulnerable Devices by Organizational Categories . . . 30
6.2.4 Community Response to Default Credential Scanner Activity 31
6.2.5 Preliminary Longitudinal Results 32
6.3 Remediation Strategy 33
6.4 Ethical Considerations of Such Studies 33
6.5 Concluding Remarks 34

7 Qualitative Assessment of Real-World Embedded Vulnerability 35
7.1 Case-Study: Reliable Cisco IOS Exploitation 36
7.1.1 Motivation 39
7.1.2 Cisco Exploitation Timeline 41
7.1.3 Two-Stage Shellcode Execution Strategy 42
7.1.4 Cisco IOS DISASM Shellcode 42
7.1.5 Interrupt Hijacker Shellcode 44
7.1.6 Stealthy Data Exfiltration 49
7.1.7 Experimental Data 51
7.1.8 Defense 52
7.1.9 Concluding Remarks 53
7.2 Case-Study Firmware Modification: HP-RFU 55
7.2.1 Overview 55

7.2.2 Firmware Modification Attack 59
7.2.3 Case Study: HP LaserJet Exploitation 61
7.2.4 Discovery Process 63
7.2.5 Proof of Concept Printer Malware 65
7.2.6 Threat Model and Assessment 67
7.2.7 Vulnerable Device Population Analysis 71
7.2.8 Vulnerable Third-Party Libraries 74
7.2.9 Recommended Defenses 77
7.3 Concluding Remarks 78
7.3.1 Poly-species propagation of advanced persistent embedded implants 78
7.3.2 Large-scale exploitation 81

III Embedded Defense **82**

8 Symbiotic Embedded Machines **83**
8.1 Introduction 84
8.2 Threat Model 86
8.3 Symbiotic Embedded Machines 87
8.3.1 Doppelgänger: A Symbiote Protecting Cisco IOS 89
8.3.2 Live Code Interception with Inline Hooks 91
8.3.3 SEMM and Execution Context Records 92
8.3.4 SEM Memory Management 93
8.3.5 Automatically Locating Control-Flow Intercept Points 94
8.3.6 Injecting Symbiotic Embedded Machines into Firmware 95
8.3.7 Rootkit Detection Payload 96
8.4 Computational Lower Bound of Successful Software-Only Symbiote Bypass 97
8.5 Symbiote Performance and Computational Overhead 99
8.5.1 Experimental Results: Doppelgänger, IOS 12.2 and 12.3, Cisco 7121 101
8.5.2 Doppelgänger, Linux 2.4.18, ARM and Qemu 103
8.6 Concluding Remarks 103

9 Autotomic Binary Structure Randomization **105**

9.1 Motivation . . . 105

9.1.1 Software Diversification . . . 106

9.1.2 Attack Surface Reduction . . . 107

9.1.3 A Hybrid Approach . . . 108

9.2 Threat Model . . . 110

9.3 Autotomic Binary Reduction . . . 111

9.3.1 Generalized $F_{et}EM$ Extraction . . . 112

9.3.2 General Autotomy Algorithm . . . 114

9.4 Code Autotomy Algorithm . . . 118

9.4.1 Return instruction injection . . . 120

9.4.2 Fast Code Autotomy Algorithm . . . 121

9.5 Binary Structure Randomization . . . 122

9.5.1 Primitive Transforms . . . 124

9.5.2 Complex BSR Transforms . . . 126

9.5.3 ABSR: Code Execution Detection (XD_{pad}) . . . 128

9.5.4 Functional Preservation . . . 129

9.6 Applied ABSR . . . 129

9.6.1 ABSR in ARM BusyBox . . . 130

9.6.2 ABSR in PowerPC Cisco IOS . . . 137

9.6.3 ABSR in MIPS Cisco IOS . . . 145

10 Case-Study: Symbiote and ABSR Defense **148**

10.1 Case-Study: Symbiote and Cisco Routers . . . 148

10.1.1 Symbiote Performance and Overhead . . . 148

10.1.2 Computational Overhead . . . 151

10.1.3 Detection Performance . . . 152

10.1.4 Control-Plane Latency . . . 152

10.1.5 Discussion . . . 154

IV Conclusion **155**

11 Future Work **156**

11.1 Qualification and Quantification 156

11.2 Symbiote 157

11.2.1 Embedded Self-Healing 158

11.2.2 Embedded Anomaly Detector 159

11.2.3 Large-Scale Embedded Sensor Grid 159

11.3 Autotomy Binary Structure Randomization 159

12 Concluding Remarks **161**

12.1 Conclusion 161

V Bibliography **165**

Bibliography **166**

VI Appendices **178**

A Appendix A **179**

A.1 Appendix: List of Embedded Device Profiles Supported by Default Credential Scanner 179

B Appendix B **180**

B.1 Cisco IOS Rookit 180

B.1.1 Disassembling Shellcode 180

B.1.2 Interrupt Hijacking Shellcode 180

C Appendix C **181**

C.1 CVE-2011-4161: HP LaserJet Firmware Modification Vulnerability 181

D Appendix D **184**

D.1 Symbiote Performance 184

List of Figures

4.1 Spectrum of proposed software diversification techniques 14

6.1 Distribution of Vulnerable Embedded Devices in IPv4 Space. Total Number of Vulnerable Devices Found: **540,435**. 21

6.2 Distribution of Vulnerable Devices Across Unique Device Types. The Top 3 Device Types Constitute 55% of the Entire Vulnerable Device Population. . 25

6.3 Embedded Device Vulnerability Rates of Monitored Countries (Threshold = 2%). 27

6.4 Discovered Candidate Devices (Left) and Vulnerable Devices (Right) by Functional Category. 27

6.5 Discovered Candidate Devices (Left) and Vulnerable Devices (Right) By Geographical Distribution. 29

6.6 Discovered Candidate Devices (Left) and Vulnerable Devices (Right) by By Organization Type. 30

6.7 Daily Page Access Analytics For Scan Project Information Page [url anonymized]. Oct 19, 2009 - April 12, 2010. 31

7.1 Timeline of two-stage attack against vulnerable IOS router of unknown hardware platform and firmware version. Attacker launches exploit with reliable shellcode (1.a). Shellcode installs rootkit and exfiltrates victim device's IOS fingerprint (1.b). The attacker finds exact IOS version from fingerprint by consulting offline database (2.a). The attacker then creates a version specific rootkit for victim and uploads it using 1.b rootkit (2.b). 38

7.2 The disassembling shellcode first locates a known string (A), then locates a xref to this string (B). Once this xref location is found, the attacker can patch the function containing the xref. This shellcode requires two linear scans of IOS memory, one through the .data section, and a second one through the .text section. 43
7.3 A disassembly of a typical f_chkpasswd. The string xref is the first highlighted block. The second highlighted block is the single instruction, which can disable password authentication in IOS. While these addresses vary greatly, they can be reliably computed at exploitation time by the disassembling shellcode. 44
7.4 The interrupt hijack shellcode first locates all *eret* (exception return) instructions within IOS's .text section. The second-stage rootkit is then unpacked inside the $gp memory area (which is unused by IOS). All *eret* instructions, and thus all interrupt service routines are hooked to invoke the second-stage code. We now have reliable control of the CPU by intercepting all interrupt handlers of the victim router. 45
7.5 Interrupt hijack second-stage rootkit. Each time any ISR (interrupt service routine) is invoked, the rootkit will seek through the latest punted packets within IOMEM for specially crafted command and control packet payloads. 47
7.6 Highlighted words, left to right, top to bottom. 1: Pointer to previous packet data node. 2. Pointer to next packet data node. 3. Exfiltration request magic pattern. 4. Beginning of next packet data entry, pointed to by 2. 48
7.7 Data exfiltration through forwarded packet payload. 1: The attacker crafts a packet with a magic pattern in its payload indicating exfiltration request. 2: Packet payload is copied into a *packet data* structure. 3: Rootkit locates magic pattern, overwrites remaining packet with exfiltrated data. 4: Packet is process-switched. The packet data entry is linked to the TX queue. 5: The requested data is sent back to the attacker inside an ICMP response packet. 50

7.8 Distribution of the location of the password authentication function. This location varies greatly across the IOS .text segment, forcing the disassembling shellcode to search a large region. 53
7.9 Distribution of the location of *eret* instructions over 162 IOS images. These locations mark the end of all interrupt service routines in IOS, and tend to be concentrated within a predictable region of IOS. 54
7.10 CPU utilization of 7204 router during the first-stage execution of both the disassembling and intercept hijack shellcodes. Note that the interrupt hijack shellcode is simpler, requires less CPU and thus avoids watchdog timer exceptions. 54
7.11 CPU intensive shellcodes will be caught by Cisco's watchdog timer, which terminates and logs all long running processes. The disassembling shellcode, although reliably bypasses password verification, consistently triggers the watchdog timer, generating the above logs, which give precise memory location of the shellcode. 55
7.12 Byte value distribution histogram of a typical RFU file. Distribution suggests that the data is compressed and not encrypted. 62
7.13 Formatter board for LaserJet P2055DN. Dump of the onboard SPI flash revealed RFU format and integrity checking algorithm. 63
7.14 Logical block diagram of the major components used on the LaserJet P2055DN formatter board. The Spansion boot flash was key to our reverse engineering effort. 64
7.15 The SPI flash chip was physically removed then connected to an Arduino for boot code extraction. 65
7.16 Boot image layout on the SPI flash chip. The level-1 boot loader contains code that validates, unpacks and decompresses the factory reset RFU allowing us to reverse engineer the binary RFU format and compression algorithm. 65
7.17 Typical advanced persistent threat attack scenario involving compromised printers. 69

7.18 Percentages of RFUs for each printer model containing known zlib and OpenSSL vulnerabilities. 76
7.19 Anatomy of a plausible poly-species malcode propagation scenario 79

8.1 Logical overview of SEM injected into embedded device firmware. SEM maintains control of CPU by using large-scale randomized control-flow interception. The SEM payload executes alongside original OS. Figure 6 shows a concrete example of how the SEM payload can be injected into a gap within IOS code. 85
8.2 Generic end-to-end process of fortifying an arbitrary host program with a Symbiote. Our proof-of-concept Symbiote for Cisco routers, Doppelgänger, is completely implemented in software and can execute on existing commodity systems without any need for specialized hardware. 88
8.3 Rendering of Symbiote structure inside a typical IOS firmware. The top of the graph shows large numbers of control-flow interceptors diverting the CPU to the SEM manager and payload, which can be seen as the small vertical structure at the bottom of the graph. 90
8.4 Live Code Regions (White) Within IOS 12.4 Firmware (Black). Code Range: 0x80008000-0x82a20000 95

9.1 Autotomic Binary Reduction 111
9.2 Generalized code structure useful for identifying Feature Entry-Point Maps in embedded firmware. 113
9.3 Typical finite graph representation of Feature Entry Map logic 114
9.4 Autotomy of Feature Entry Point A. The red areas denote code regions which can be safely removed. 116
9.5 Three example flow-graphs to illustrate the **Dominator** function 117
9.6 Three example flow-graphs to illustrate the **Inverse Dominator** function . 118
9.7 Autotomy of both code and data of Feature Entry Point A. The red areas denote code regions that can be safely removed. 119
9.8 Feature Entry-Point Return Value Identification 120

9.9 Binary Structure Randomization 122
9.10 General 123
9.11 Binary Structure Randomization: Primitive Transform 124
9.12 Binary Structure Randomization: Basic block Relocation 125
9.13 Binary Structure Randomization: Basic block splitting 127
9.14 Binary Structure Randomization: Basic block swapping 127
9.15 Code Execution Detection Detector Pads 128
9.16 Feature selection control-flow structure for Busybox 131
9.17 Data associated with feature selection control-flow structure for Busybox . 131
9.18 BusyBox $F_{et}EM$ Enumeration 131
9.19 ABR applied to busybox. Non-black regions represent removed code regions. 132
9.20 ABR applied to busybox, feature-set size=1 133
9.21 ABR applied to busybox, feature-set size=3 133
9.22 ABR applied to busybox, feature-set size=5 134
9.23 ABR applied to busybox, feature-set size=7 134
9.24 ABR applied to busybox, feature-set size=9 135
9.25 ABR applied to busybox, feature-set size=11 135
9.26 ABR applied to busybox, feature-set size=13 136
9.27 ABR applied to busybox, feature-set size=353 136
9.28 Binary Diff Rate vs Original Busybox Binary 137
9.29 Binary Diff Rate vs Original Busybox Binary 137
9.30 BusyBox **unzip** utility runtime over 1MB random data 138
9.31 BusyBox **unzip** utility runtime over 1MB random data 139
9.32 BusyBox **sha512** utility runtime over 1MB random data 140
9.33 BusyBox **sha512** utility runtime over 10MB random data 141
9.34 BusyBox **sha512** utility runtime over 100MB random data 142
9.35 IOS Create Process function 142
9.36 IOS Create Process function call-graph 142
9.37 ABR applied to Cisco 3750 IOS 12.1, feature-set size = 1 143
9.38 ABR applied to Cisco 3750 IOS 12.1, feature-set size = 3 143

9.39 ABR applied to Cisco 3750 IOS 12.1, feature-set size = 5 144
9.40 ABR applied to Cisco 3750 IOS 12.1, feature-set size = 248 144
9.41 IOS Create Process function 145
9.42 ABR applied to Cisco 2821 IOS 12.3, feature-set size = 1 146
9.43 ABR applied to Cisco 2821 IOS 12.3, feature-set size = 3 146
9.44 ABR applied to Cisco 2821 IOS 12.3, feature-set size = 3 146
9.45 ABR applied to Cisco 2821 IOS 12.3, feature-set size=669 147

10.1 Symbiote-based Cisco IOS Detector Testing and Verification Environment . 149
10.2 CPU Utilization : Fixed Burst-Rate SEM Manager 151
10.3 CPU Utilization : Inverse-Adaptive SEM Manager 152
10.4 Detection Latency : Fixed Burst-Rate SEM Manager 153
10.5 Detection Latency : Inverse-Adaptive SEM Manager 153
10.6 Ping Latency : Fixed Burst-Rate SEM Manager 154
10.7 Ping Latency : Inverse-Adaptive SEM Manager 154

C.1 Hex dump of a typical HP-RFU. For P2055DN, using the undocumented PJL/ACL language. 181
C.2 "UAT" table structure. Contains a checksum value, followed by a directory manifest describing various compressed components of the binary update package. 182
C.3 RFU binary embedded inside a typical PostScript file. This illustrates the most straightforward reflexive attack. 182

D.1 CPU Utilization on Cisco 7121 Router Using Different SEM Payload Execution Bursts Rates ($g(\alpha_i, \tau_q)$) for IOS 12.2 and 12.3. Note the Direct Relationship Between $g(\alpha_i, \tau_q)$, SEM Payload Execution Time and Total CPU Utilization. Terms Low, Med, High, and Really High Utilization Corresponds to Varying SEM Payload Burst Rates, $g(\alpha_i, \tau_q)$. 185
D.2 Inverse Relationship between SEM Payload Burst Rate ($g(\alpha_i, \tau_q)$) and Detection Latency. 185

List of Tables

6.1 Scale and Result of the Latest Global Default Credential Scan. 21

6.2 Vulnerability Rate by Device Category. 28

6.3 Total Discovered Candidate Embedded Devices and Corresponding Vulnerability Rates By Geographical Location (Continental). 28

6.4 Vulnerability Rate By Organization Type. 30

6.5 Email Correspondences Received from Network Operators Regarding Scanning Activity. 32

6.6 Preliminary Longitudinal Study Tracking 102,896 Vulnerable Devices Over 4 Months. 32

7.1 Reliability of the disassembling shellcode and interrupt hijack shellcode when tested on 159 IOS images. 51

7.2 Reliability of exfiltration mechanism when the number of packet-data nodes searched per invocation varies. Searching more than 64 nodes caused the test router to behave erratically. 52

7.3 Observed population of printers vulnerable to the HP-RFU attack on IPv4. 72

7.4 Observed population of printers vulnerable to attacks other than HP-RFU on IPv4. 73

7.5 Organizational distribution of vulnerable printers. 73

7.6 Geographical distribution of vulnerable printers. 74

7.7 Lifespan of vulnerabilities in third-party libraries used by LaserJet firmware. 76

7.8 Third-party library vulnerability analysis observations. 77

8.1 Doppelgänger Implementation Stats 101
8.2 Detection Latency at Different SEM Payload Burst Rates IOS 12.2 102
B.1 MIPS-based disassembling rootkit statistics. 180
B.2 MIPS-based interrupt hijack rootkit statistics. 180
C.1 Printer models and firmware images analyzed for vulnerable libraries. . . . 183

Acknowledgments

I thank my advisor, Salvatore J. Stolfo, for a great number of things. I thank him for giving me the opportunity to embark on the intellectual journey that culminated in the findings presented in this dissertation. I am grateful for his ever-present support throughout my time at Columbia University. I am also thankful for Sal's reverend guidance in all matters scientific. Most importantly, I am thankful for his confidence in my endeavors and his friendship.

I thank Joel Rosenblatt and the entire Columbia University Information Security team. Without their support, the world-wide quantification of embedded device vulnerability rates would not have been possible.

I must also thank Daisy Nguyen and the Columbia University Computing Research Facilities team. Daisy's constant encouragement, kind advice, and spare parts pushed me through many difficult times. I am grateful for Daisy's support and friendship.

I thank all the dedicated researchers and students who helped me with the difficult task of turning the ideas presented in this dissertation into reality, of which two must be mentioned specifically. Thank you, Jatin Kataria and Michael Costello, for all your help in overcoming the innumerable technical challenges we faced together.

I also thank my labmates, Nathaniel Boggs, Yingbo Song, Malek Ben Salem, Gabriella Cretu-Ciocarlie, Peter Du, Jon Voris, Jill Jermyn and David Tagatac for your companionship, your open ears and brilliant open minds. Thank you for all your insightful input, suggestions and inspirations that shaped my research. Thank you for putting up with me as an officemate.

Thank you, Adrian Grabicki, for your constant friendship and much needed support.

I am also grateful for the comments and suggestions of many anonymous reviewers of the various papers I have published while at Columbia University. Much of the materials

presented in this dissertation is based on work supported by DARPA, IARPA and DHS.

Lastly, and most importantly, I thank my parents Ran Zhao and Zidu Cui for their life-long support, constant encouragement and love.

For my parents.

-

For the breath in my breath.

Part I

Introduction

Chapter 1

Motivation

The world is monitored, powered, supported, and controlled by billions of embedded computers. These devices constitute the substrate of our modern technological infrastructure. Embedded computers are used to control the production and distribution of energy, the operation of modern vehicles, the functions of devices that range from miniature implantable medical devices to grand weapon systems that operate over land, air, sea, and space.

Cyber attacks against embedded systems can have profound consequences. Successful exploitation of embedded systems can allow the attacker to manipulate financial markets, conduct espionage, disrupt global communication, damage critical infrastructure, and impact the outcome of armed conflicts.

Is it possible, in our increasingly connected and automated world, that software vulnerabilities in black-box embedded devices can be used to cause disruption and destruction in the physical world? Can embedded systems be exploited? If so, do vulnerable devices exist in significant quantity or perform significantly critical functions such that the exploitation of such devices should be considered a real threat? Can we trust that the embedded devices we depend on today have not already been compromised? Can we reliably detect the consequence of exploitation after the fact? Most importantly, what can be done to improve the security of embedded systems given their unique and constrained nature?

The remainder of this thesis represents a body of scientific study aiming to answer these questions. In short, embedded systems can be, and have been, exploited. They exist in vast numbers and perform critical functions in the world. The exploitation of embedded

systems should be considered a real and present threat. Most importantly, we propose several techniques in this thesis that can be applied to real-world embedded systems to make them more secure against cyber attack.

To better understand the nature of embedded security, we first present quantitative and qualitative evidence of the existence of vulnerable embedded devices in the world. Specifically, a internet-wide scan was carried out to study the make-up and measure the lower-bound on the quantity of vulnerable embedded devices accessible over the public internet. We discovered that approximately 20% of all embedded devices on the internet was configured with a well-known default root credentials, making them trivially vulnerable to attack. Furthermore, we predicted, through quantitative measurements and qualitative analysis, that the size of an embedded device botnet would most likely be around 540,000 devices. This prediction was validated two years after the initial publication of our work by the public announcement of the Carna botnet [5], which compromised over 480,000 devices.

Next, we present a series of case-studies of exploitation against ubiquitous embedded devices to gain greater insight into the nature of common embedded vulnerabilities. These case-studies resulted in the public disclosures of 4 vulnerabilities that affected millions of devices in the world; CVE-2012-5445, CVE-2013-6685, ASA-2014-099, CVE-2011-4161. While the exact vulnerabilities varied from device to device, several common traits can be extracted about their nature. These vulnerabilities have existed for many years, the complexity of the vulnerabilities are low, and successful exploitation can be done using offensive techniques that can be considered obsolete by modern exploitation standards. As we begin to define and analyze the nature of embedded systems, we will see that devising effective host-based defenses is uniquely challenging. Chapter 5 of this thesis discusses these challenges in detail.

Lastly, we propose and discuss two software-based defensive techniques, Software Symbiote and Autotomic Binary Structure Randomization, that are designed to improve the security posture of embedded devices. We demonstrate the efficacy of these two defensive techniques by applying them to real-world embedded devices that are known to be vulnerable. We present experimental data confirming the safety of our prosed defenses, as well as their efficacy against real-world exploitation.

1.1 What is an embedded device?

Before we begin, we present a definition of what an embedded device is. Embedded devices cannot be categorically defined by the architecture of their CPU, the computing power of the device, the device's cost, size, shape, color, weight, power consumption, manufacturer, the device's operating system, or the language in which the device's software is written. Instead, we propose a qualitative definition for what an embedded device is.

An embedded device is *a general purpose computer who's software is intended to function in a specific and confined way.*

Unlike general-purpose computers, an embedded device is expected to perform a narrow range of functions. This range of functionality is confined by hardware and software, and for both technical and non-technical reasons. Generally speaking, users and operators of commercial embedded systems may *reconfigure* the behavior of the device, but are prohibited from adding, deleting, or modifying the firmware of the device. In other words, operators of embedded systems typically cannot *install* or *uninstall* code from the device. This constraint is enforced at a contractual level through EULAs and warranty agreements. More importantly, this constraint is enforced at a software design level. Thus, embedded systems typically lack the technical mechanisms necessary for operators of these devices to install custom programs, which makes the implementation of third-party security solutions uniquely challenging.

Chapter 2

Hypothesis

Embedded devices are vulnerable to large-scale exploitation. The use of software defensive techniques that take into consideration the hardware and software constraints imposed by such systems can provide effective and efficient detection of and defense against the exploitation of several classes of software vulnerabilities, as well as the injection of persistent software implants in legacy embedded devices. Such software-based defensive techniques can be automatically realized by making modifications to the firmware that do not alter the original functionality of the firmware but introduce various security capabilities to the embedded device at a cost of an acceptable level of resource overhead. Most importantly, such software-based defenses should be realizable at the binary level, without requiring access or modification to source-code, and should not require any hardware modification.

Chapter 3

Contributions

The body of scientific study presented in this thesis makes the following contributions to the study of the security of embedded devices:

- Established measurements of embedded insecurity through large-scale quantitative data collection and analysis.
- Systematically analyzed significant bodies of embedded system firmware for vulnerability and exploitability.
- Identified and analyzed several prototypical embedded vulnerabilities, and collaborated with device manufacturers to produce the appropriate security patches.
- Devised the theoretical operation of Software Symbiote, a software and hardware agnostic software construct that can provide host-based defenses to many embedded devices.
- Validated the safety and efficacy of Software Symbiote implementations on real-world embedded devices.
- Devised the theoretical operation of Autotomic Binary Structure Randomization, a binary-level, non-localized, in-place code randomization technique that requires no operating system or hardware support.
- Validated the safety and efficacy of Autotomic Binary Structure Randomization on real-world embedded devices.

We hope this line of research will lead to substantial future research in the area of understanding and improving the security posture of embedded systems, and ultimately, of our computational and communications infrastructure as a whole.

Chapter 4

Related Work

We examine a body of work related to embedded security through several perspectives. First, we present prior studies aimed at the quantification and qualification of the feasibility of embedded exploitation. Next, we present a selection of prior academic and real-world studies of the exploitation of embedded devices. Lastly, we survey prior works related to the defense of embedded devices.

4.1 Quantification and Qualification

Accurate quantification and qualification of the security posture of existing embedded devices in the wild is crucial to the science of embedded security. However, the collection of reliable and accurate data is challenging. Nonetheless, several large-scale studies have shed light on the level of insecurity of existing embedded devices.

4.1.1 Large-Scale Evaluation of Firmware Vulnerabilities

One way to measure the level of vulnerability within embedded devices is to analyze the firmware content of devices for known vulnerabilities. For example, vulnerabilities found within common software components within general-purpose computers [20, 95] can sometimes be used to exploit embedded devices that also use such components [10]. A quantitative study [25] scanned 32,356 firmware images for software components containing known vulnerabilities. The study found that approximately 2.14% of analyzed firmware instances

contained at least one known vulnerability. [31] presented a similar analysis within a more specific corpus of firmware instances yielded a vulnerability of over 65%. While vulnerability rate reported by [25] appears to be relatively low, realistic data on the quantitative distribution of embedded device types is required to deduce the real-world impact of this study.

4.1.2 Embedded Exploitation in the Wild

The "Internet Census of 2012" [5] is perhaps the most well-documented example of real-world, large-scale exploitation of embedded devices. The size and scope of this exploitation was largely predicted by our quantitative study [36] published several years prior.

This work was done by an anonymous party. The stated purpose of the research was to map the scale and structure of the Internet. To this end, the author(s) engineered and deployed a botnet that targeted OpenWRT-based devices. The malware implantation process involved an internet-wide scan in order to identify vulnerable embedded devices with well-known default passwords, followed by the injection of malcode binary previously built for the device's hardware architecture. The malicious payload was reportedly written in C and compiled to execute within 9 hardware variants of OpenWRT devices.

This work provided definitive proof that large-scale exploitation of heterogenous bodies of embedded devices is occurring in the world in several ways. First, the documented quantifications of this work closely correlates to predictions made by previous work [37]. Second, the authors of this work inadvertently discovered a second botnet already present on a small population of embedded devices called Aidra [47]. This evidence further proves that large-scale exploitation of embedded devices has taken place even prior to 2012.

In recent years, numerous attacks once demonstrated by the security community as hypothetical, or proof-of-concept against embedded systems have been seen in the wild. For example, evidence [4] suggests that persistent malware implants that reside in Cisco routers, leveraging the ROMMON region, have been used and recently discovered in real-world scenarios. The actual malware samples and detailed technical information related to such attacks are currently being withheld from the general security community. However, judging from the information released by the device vendor, it is likely that such exploits

have been predicted and demonstrated by [80] and others for nearly a decade.

4.2 Embedded Exploitation

There exists a large body of work describing the exploitation of embedded devices. While it is likely that the rate of proliferation of offensive embedded technology outpaces that of defensive embedded technology, relatively few bodies of offensive work have been scientifically documented. The following list is a representative sample of publicly documented studies of the exploitation of embedded devices.

1. Medical Devices: [52, 65]
2. Peripherals and COTS components: Network Cards [109], Smart Battery Controller [86], Keyboard [21]
3. Computer Peripherals: [21, 40, 120]
4. Networking Equipment: [34, 77, 93]
5. Voting Machines: [8]
6. GSM Baseband: [118]
7. Consumer Electronics: Gaming Console: [13]
8. Office appliances: [27, 28, 29, 31]
9. Industrial Control Systems: [50, 90]
10. Financial/Commercial Systems: [64]
11. Cars: [111]

Network Bluepill, also known as 'psyb0t', is perhaps the most direct evidence of the feasibility of large scale exploitation of network embedded devices. According to dronebl.org [42], this home router-based botnet, which principally targets MIPS-based devices running OpenWRT and DD-WRT, was discovered after dronebl.org became the victim of a DDOS attack levied by the same botnet. This botnet uses password guessing as the principal attack vector, contains shell codes for several popular MIPS-based network embedded devices, and is packed with UPX for binary obfuscation. Once compromised, the router or DSL modem is used to sniff user credentials, scans for vulnerable network embedded devices as well

as exploitable phpMySQLAdmin and MySQL installations, and carries out DDOS attacks. While no detailed analysis of the malicious code was released, it is suspected that the psyb0t botnet observed in 2008 was merely a proof of concept test of the technology [3], as the botnet was quickly shutdown by its operators following dronebl.org's public announcement of its existence. Evidence suggests that a recent incarnation of psyb0t now contains shellcode for over 55 different home router models as well as a list of 6,000 usernames and 13,000 passwords [2].

4.2.1 Cisco Router Exploitation

As a demonstrative example, we present the timeline of published offensive works against Cisco routers.

FX, 2003: FX analyzes several IOS vulnerabilities and various exploitation techniques [78].

Lynn, 2005: Lynn described several IOS shellcode and exploitation techniques, demonstrating VTY binding shellcode [85].

Lynn, 2005: Cisco and ISS Inc. files injunction against Michael Lynn [1].

Uppal, 2007: Uppal releases IOS Bind shellcode v1.0 [110].

Davis, 2007: Davis releases IOS FTP server remote exploit code [38].

Muniz: 2008 Muniz releases DIK (Da IOS rootKit) [87].

FX, 2009: FX demonstrates IOS diversity, demonstrates reliable disassembling shellcode and reliable execution methods involving ROMMON [80].

Muniz and Ortega, 2011: Muniz and Ortega releases GDB support for the Dynamips IOS emulator, and demonstrates fuzzing attacks against IOS [89].

4.3 Embedded System Defense Technologies

Relatively little work has been done to detect and capture sophisticated attacks against embedded devices. However, such problems have been well studied for general purpose computers and operating systems. A multitude of rootkit and malware detection and mitigation mechanisms have been proposed in the past but largely target general purpose computers.

Sophisticated detection and prevention strategies have been proposed by the research community. Virtualization-based strategies using hypervisors, VMM's and memory shadowing [96] have been applied to kernel-level rootkit detection. Others have proposed detection strategies using binary analysis [71], function hook monitoring [117] and hardware-assisted solutions to kernel integrity validation [112].

Numerous rootkit and malware detection, and mitigation mechanisms have been proposed for general purpose computers and operating systems (virtualization-based[96], binary analysis [71], function hook monitoring [117], *etc.*). Traditional anomaly detection methods, such as monitoring sys-call patterns [46] have also been applied to embedded devices [119] .

These strategies may perform well within general purpose computers and well-known operating systems, but they have not been adapted to operate within the unique characteristics and constraints of embedded device firmware (limited storage, memory and processing; absence of memory management units; real-time operating systems; *etc.*). Effective prevention of binary exploitation of embedded devices requires a rethinking of detection strategies and deployment vehicles.

DynamoRIO [44] is a runtime code manipulation system that supports code transformations on any part of program. An application launched by DynamoRIO can be analyzed and manipulated through its API. DynamoRIO is designed for general purpose operating systems like Windows and Linux on the X86 architecture.

Much work has been done in the area of remote software attestation as a defense against firmware modification. SWATT: Software-Based Attestation for Embedded Devices, proposed by [101], and SBAP: Software-Based Attestation for Peripherals, proposed by Li *et al.*[74], involve the external validation of embedded devices through the use of a challenge-response protocol. VIPER, proposed by Li *et al.*[76], can be applied directly to mitigate a real-world firmware modification attack against keyboards[22]. While promising, timing-based attestation techniques tend to be vulnerable to hardware over-clocking and time of check vs time of use (TOCTTOU) attacks. Most importantly, such defense mechanisms are generally stop-the-world algorithms, requiring a full halt of the system while remote attestation is in progress. While perhaps adequate for keyboards, it would be difficult to directly apply such techniques to embedded devices with hard real-time requirements, like

routers and firewalls, which must deliver uninterrupted availability.

Guards, originally proposed by Chang and Atallah[19], are simple pieces of code that are injected into the protected software using binary rewriting techniques. Once injected, a guard can perform tamper-resistance functionality like self-checksumming and software repair. Guards are localized modifications that monitor control-flow and data-flow integrity of the immediate predecessors of each modification. Symbiote, on the other hand, takes a non-localized approach and performs integrity verification checks that are not in-line to the execution of the host program. In a sense, if Guards can be thought of as inline monitors, Symbiote can be thought of as *orthogonal* monitors.

Furthermore, the following bodies of prior work discusses defensive techniques and approaches that may be applied to some sub-classes of embedded devices:

Software-only binary rewriting-based defenses: [19, 45]

Co-processor and hypervisor-based defenses: [43, 67, 116]

Dynamic binary translation-based defenses: [12, 48, 70]

Static firmware attestation defenses: [14, 75, 76, 99, 100, 101, 106]

Medical device senses and its technical and legislative consequences: [23, 41]

4.4 Software Compaction

A large body of work exists that deals with software optimization and compaction techniques. These techniques largely focus on the reduction of software size and the enhancement of software efficiency ([39]).

[97] posits that security vulnerabilities are frequently the consequence of unwanted features in a software system, which results from overly general (bloated) software, feature accretion, subsystem reuse and development errors on the part of designers and implementors and vulnerability insertion on the part of attackers. Rinard also outlined several potential remedies, including feature replacement or excision, input rectification and dynamic modification. Furthermore, [83] proposed techniques to manipulate and shift the attack surface of computer systems in a game theoretic framework.

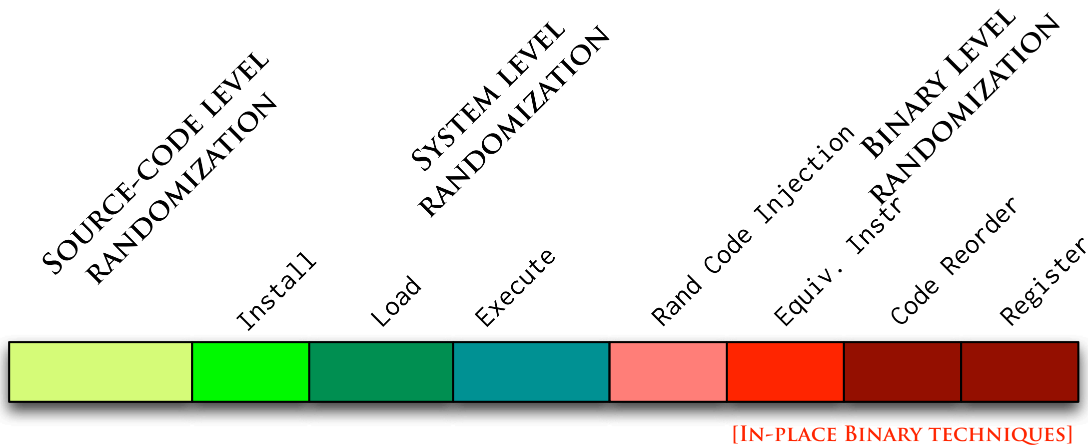

Figure 4.1: Spectrum of proposed software diversification techniques

4.5 Defensive Randomization

Randomization techniques have been applied to numerous aspects of software to increase its resiliency against reliable exploitation of known vulnerabilities. For a comprehensive systemization of knowledge paper surveying the spectrum of known software diversification techniques, we refer the reader to [72]

Figure 4.1 shows the general spectrum of randomization techniques. Given the constraints of embedded systems, in-place binary-level randomization techniques are the most promising. However, the space and locality constraints caused by making only localized in-place modifications limit the effectiveness of such techniques.

If source-code is available, a plethora of randomization techniques can be added to the compilation process. Such techniques include the randomization of static data layout, heap and stack randomization as well as instruction-level techniques involving register reassignment, instruction reordering and randomized nop injection [66]. Similar techniques can also be applied to JIT compilers [54].

When source-code is not available, system-level randomization techniques can be used, provided that the operating-system can provide support for such techniques. System-level diversification techniques include Address Space Layout Randomization (ASLR) and Instruction Set Randomization (ISR). ASLR aims to randomize the in-memory base-address

layout of code and data at execution time [102]. ISR aims to randomize the encoding of the op-codes during execution[68].

When source-code is not available and operating-system-level support does not exist, some in-place instruction-level randomization techniques have been demonstrated to be feasible by applying standard static analysis on disassembled executable binary. Such prior works were largely limited to making small localized alterations, such as instruction re-ordering and register reassignments. Prior work has demonstrated the feasibility of randomization techniques such as randomized nop-instruction injection[66], randomized instruction reordering and register reassignment[92]. While this is effective against code reuse techniques like ROP, much of the the binary structure of the diversified binary remains unchanged. As a result, targeted patching of known memory-locations can still have reliable effect. The Cisco IOS attack presented in Section 7.1 is a good demonstration of why instruction-level diversification alone is insufficient to prevent reliable exploitation.

[49] and others have proposed the injection of functionality preserving instructions into software as a defensive technique against reliable shellcode execution.

Chapter 5

Problem Description

5.1 Real-world quantification of embedded system vulnerability and exploitability

While embedded systems are anecdotally believed to be vulnerable to attack, empirical data to support this belief is lacking. Large-scale survey of existing embedded devices for vulnerabilities and exploitability is needed to quantify the state of embedded insecurity in real-world environments. Large scale analysis of embedded firmware images for vulnerabilities is needed to quantify and qualify the exploitability of bugs within embedded device software. This is currently difficult because the unpacking of firmware binaries of unknown formats have not been automated, making large-scale analysis of embedded firmware labor intensive and error-prone.

5.2 Generalized firmware analysis and modification

The proprietary and non-standard nature of embedded device firmware makes the analysis a largely manual and labor intensive process. Furthermore, making modifications to embedded device firmware for the purpose of testing on actual embedded hardware poses numerous challenges. For example, the modified binary must be re-packed into the vendors proprietary format before the new embedded firmware can be loaded onto the intended device. In order to overcome these challenges, embedded security researchers must spend a

large amount of time to painstakingly reverse-engineer proprietary compression, encryption, integrity-checking and signature verification algorithms before any actual security analysis can be done. Software integrity features such as trusted boot and firmware signing can even hinder the development, testing and deployment of host-based security features in physical embedded devices.

5.3 Generalized software environment agnostic defense

Commercially viable embedded device fortification technology does not currently exist. While third-party end-point protection software exists for general-purpose computers, no host-based solution exists which allows the operators of embedded devices to defend these devices against exploitation. Furthermore, no solution exists which will alert the operators when these devices have become compromised. For example, Cisco IP phones and HP printers are ubiquitous fixtures within the modern office environment. Serious security vulnerabilities have been found in these devices. The currently accepted practice in the operational embedded security field is to rely entirely on the device vendor to identify and fix vulnerabilities. The operators of embedded devices have no real method of protecting these devices against exploitation or to audit these devices to ensure that they have not been compromised. Bringing effective host-based security into embedded systems is challenging for at least the two reasons.

The manufacturer problem: Embedded systems are typically not designed with security as a significant focus. Vendors of embedded systems typically do not have sufficient expertise to engineer secure devices. Furthermore, when security flaws are identified, vendors will typically attempt to fix the immediate issue to the best of their ability. Since their security expertise is limited, these fixes take a significant amount of time to reach their customers; they are likely to be stop-gap fixes that do not address the root cause of the vulnerability and may even introduce additional bugs and vulnerabilities into their product.

The operator problem: No existing technology allows the operator of embedded devices to fortify these devices according to their own internal policy. Furthermore, no com-

> mercially available technology allows the operator to audit embedded devices to ensure that the software running in each device has not been altered or compromised. While enterprise security professionals have the expertise and ability to lock down and harden embedded systems according to their own internal environment and security policy, the necessary technology and tools that can make the application of security expertise possible do not exist.

In order for an embedded defense technology to be viable, it must be able to defend a diverse set of proprietary software running within unknown software environment on diverse types of resource-constrained black-box hardware. For a defense to be practical and economical, it needs to perform its defensive functions without requiring source code or intellectual property from the vendor. It must not require major rewrite of the software and it cannot require any modification of existing hardware.

In other words, what is required is a software environment agnostic structure that can be injected into embedded device firmware of unknown design at the binary level. Such a defense must be able to execute properly and safely on a wide range of software environments and hardware platforms using only existing commodity hardware capabilities. It must also operate in a way that does not significantly impact the real-time responsiveness of the protected device and in a way that does not impact existing functionality in a negative way.

The two host-based defensive techniques presented in this thesis, Symbiote and Autotomic Binary Structure Randomization, are designed to satisfy the above requirements in an automated fashion.

Part II

The Embedded Threatscape

Chapter 6

Quantitative Assessment of Real-World Embedded Vulnerability

6.1 Real-world quantification of embedded device vulnerability

We seek to quantify and trend the level of insecurity of embedded devices currently in the wild. To this end, we first establish an observed **lower bound** on the number of trivially vulnerable embedded devices on the Internet. We do this by assuming the role of the least sophisticated malicious attacker, who only tries to log into publicly reachable embedded devices using well known **default root credentials**. The default credential scanner, which we we developed using standard tools such as **nmap**, positively identified over **540,000** wide open embedded devices.

Vulnerable devices were detected in **144 countries**, in enterprise, ISP, government, educational, satellite provider as well as residential network environments[1]. We discovered vulnerable devices across a diverse spectrum of product types, including consumer appliances, home networking devices, office appliances, enterprise and carrier networking

[1] US Military networks are intentionally excluded from our scan.

Total IPs Scanned	Devices Targeted	Vulnerable Devices Found	Overall Vulnerability Rate
3,223,358,720	3,954,620	540,435	13.67%

Table 6.1: Scale and Result of the Latest Global Default Credential Scan.

equipment, data-center power management devices, network security appliances, server lights-out-management controllers, IP camera surveillance systems, VoIP devices, video conferencing appliances, ISP issued modems, and set-top boxes.

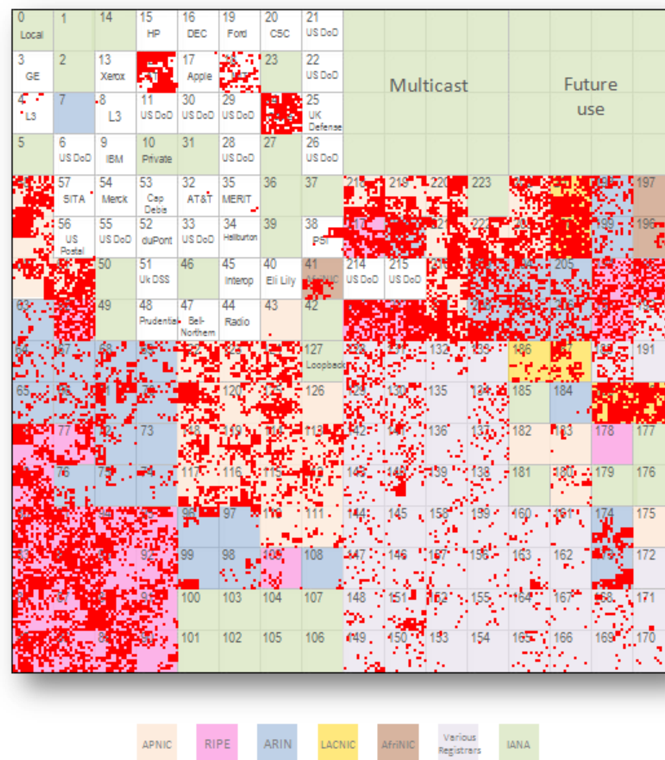

Figure 6.1: Distribution of Vulnerable Embedded Devices in IPv4 Space. Total Number of Vulnerable Devices Found: **540,435**.

While the observed quantity and distribution of embedded devices configured with default root passwords demonstrate a global, pervasive phenomenon, we believe the data presented in this chapter represent a conservative lower bound on the actual population of vulnerable devices in the wild.

We present the first quantitative measurement of embedded device insecurity on a global

scale, along with preliminary results from an ongoing longitudinal study of the same subject. By assuming the role of the least sophisticated attacker, we present an observed **lower bound** on the distribution of trivially exploitable network embedded devices over functional (Section 6.2.1), spatial (Section 6.2.2), organizational (Section 6.2.3), and temporal (Section 6.2.5) domains.

The embedded device default credential scanner created for this experiment is designed to identify efficiently and safely the vulnerable embedded devices on the network. It does this by testing whether one can remotely log into a device using well known default root credentials. The verification process is designed to use minimal resources on the target embedded device. The scanner currently supports 73 unique embedded device types including consumer appliances, home networking devices, office appliances, enterprise and carrier networking equipment, data-center power management devices, network security appliances, server lights-out-management controllers, IP camera surveillance systems, VoIP devices, video conferencing appliances , ISP issued modems and set-top boxes.

While the embedded security threat has been generally known for some time, the data presented in this chapter provides a real-world quantitative assessment of the scale and scope of the embedded threat on a global level. Analysis of our results yields several interesting features within the observed vulnerability distributions. The features presented in Section 6.2 presents insights into the root causes of the existence of vulnerable embedded devices. By combining the observed vulnerability distributions and its potential root causes, we formulate a set of mitigation strategies and hypothesize about its quantitative impact on reducing the global vulnerable device population.

Many forces will undoubtably change the observable lower bound of embedded device insecurity as time goes on. For example, the out-of-the-box security of new embedded products may change. Network operators controlling large homogeneous sets of devices may improve their security, as may small and medium size organizations like private enterprises and educational organizations. The level of malicious exploitation will also indirectly contribute to the overall effort dedicated to improving embedded device security. Lastly, it is our hope that the data and mitigation strategies reported in this chapter will generate more awareness of this pervasive threat. In order to quantify the scope of the embedded

device insecurity threat over time and detect such forces at work, we plan to continue our scanning activities to conduct an ongoing longitudinal study over the next year. Section 6.2.5 discusses the preliminary results of our longitudinal study over the past four months.

The default credential scanner is designed to quickly sweep large portions of the Internet. Each scan takes approximately four weeks and involves two or three sweeps of the entire monitored IP space.

Multiple sweeps across the same IP space is desirable for two reasons. First, embedded devices on residential networks have unpredictable availability. Therefore, multiple sweeps increase the scanner's probability of observing a vulnerable device when it is connected to the network. Second, multiple sweeps across the same address space over months and years allow us to conduct a **longitudinal** study on the vulnerability rates of embedded devices around the world.

In Section 6.2, we present the results of our latest scan, containing over 540,000 observed vulnerable devices as well as the analysis of preliminary data gathered by tracking approximately 102,000 vulnerable embedded devices over a span of four months. This is an ongoing study, and we plan to publish the results of a detailed longitudinal study over the next year when the data becomes available.

For the sake of establishing a lower bound on the state of embedded device insecurity in the wild, we assume the role of the least sophisticated malicious attacker. The attacker has unrestricted access to the Internet but is unable to exploit any vulnerabilities found on any devices. Instead, the attacker has access to the network scanner nmap and a list of well known factory default root credentials for popular network embedded devices.

For the remainder of the chapter, we define a *vulnerable* device as any device that is reachable on the Internet and allows the attacker to gain root privileges by using factory default credentials.

The default credential scan process is straightforward and can be broken down into three sequential phases: **recognizance**, **identification**, and **verification**.

Recognizance: First, nmap is used to scan large portions of the Internet for open TCP ports 23 and 80. The results of scan is stored in a SQL database.

Identification: Next, the device identification process connects to all listening Telnet and

HTTP servers to retrieve the initial output of these servers[2]. The server output is stored in a SQL database then matched against a list of signatures to identify the manufacturer and model of the device in question.

Verification: Once the manufacturer and model of the device are positively identified, the verification phase uses an automated script to attempt to log into devices found in the identification phase. This script uses only well known default root credentials for the specific device model and does not engage in any form of brute force password guessing. We create a unique *device verification profile* for each type of embedded device we monitor. This profile contains all information necessary for the verification script to automatically negotiate the authentication process, using either the device's Telnet or HTTP administrative interface. Each device verification profile contains information like the username and password prompt signatures, default credentials as well as authentication success and failure conditions for the particular embedded device type. Once the success or failure of the default credential is verified, the TCP session is terminated, and the results are written to an encrypted flash drive for off-line analysis.

6.1.0.1 Malicious Potential of Embedded Device Exploitation

The heterogeneous nature of embedded administrative interfaces makes orchestrating large DDOS attacks using embedded devices a logistic challenge. Vulnerable embedded devices clearly exist in large numbers in the wild. However, it is often believed that embedded operating systems are too diverse, and capturing the long tail of this diversity is required to carry out large scale exploitation. Data gathered by our default credential scanner reveal that many large vulnerable homogenous device groups exist in the wild. In fact, the top **3** most vulnerable device types represent over **55%** of all vulnerable devices discovered by our latest scan. In other words, there exists at least 300,000 vulnerable embedded devices that can be controlled via 3 similar Telnet-based administrative interfaces. Figure 6.1.0.1 shows the distribution of the top 12 most frequently encountered vulnerable embedded device types.

[2]In case of HTTP, we issue the 'get /' request

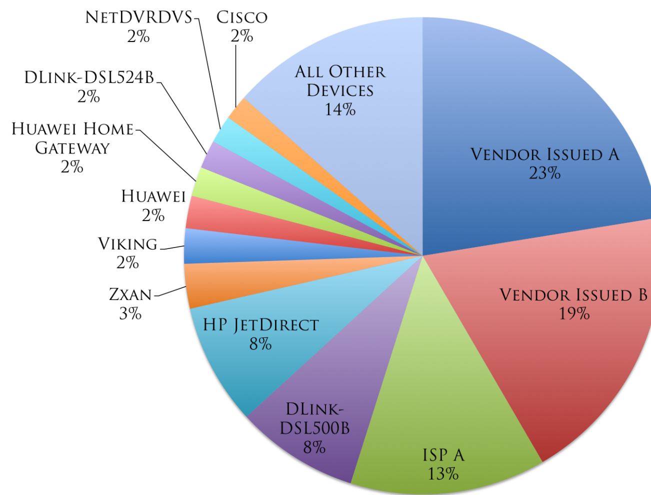

Figure 6.2: Distribution of Vulnerable Devices Across Unique Device Types. The Top 3 Device Types Constitute 55% of the Entire Vulnerable Device Population.

6.1.0.2 VoIP Appliance Exploitation

VoIP adapters like the Linksys PAP2, Linksys SPA and Sipura SPA are consumer appliances, which provide a gateway between standard analog telephones and VoIP service providers. In many cases, the publicly accessible HTTP interface of such devices will display diagnostic information without requiring any user authentication. This information usually includes the name of the customer, their phone number(s), a log of incoming and outgoing calls, and relevant information regarding the SIP gateway to which the device is configured to connect. Once authenticated as the administrative user, an attacker can usually retrieve the customer's SIP credentials, either by exploiting trivial HTTP vulnerabilities[3] or redirecting the victim to a malicious SIP server.

6.1.0.3 Data Leakage via Office Appliance Exploitation

Enterprise printers servers and digital document stations are ubiquitous in most work environments. According to our data, network printers also constitute one of the most vulnerable types of embedded devices. For example, our default credential scanner identified over

[3]Credentials are sometimes displayed in clear-text within HTML password fields. While this appears to hide the passwords in the web browser, it does not hide it in the HTML source.

44,000 vulnerable HP JetDirect Print Servers in **2,505** unique organizations worldwide. Since high-end print servers and document stations often have the capability of digitally caching the documents it processes, we posit that an attacker can use such devices not only to monitor the flow of internal documents but also to exfiltrate them as well.

6.1.0.4 Enterprise Credential Leakage via Accidental Misconfiguration

It is a common practice for organizations that operate large homogenous collections of networking equipment to apply the same set of administrative credentials to all managed devices. While this significantly reduces the complexity and cost of managing a large network, it also puts the network at risk of total compromise. Using a single master root password for all networking devices is safe so long as every device is correctly configured at all times, and the master password is not leaked. If an enterprise networking device is brought online with both factory default credentials, as well as the master credentials of the organization, an attacker can easily obtain the master root password for the entire network. While this event is unlikely, the probability of such a misconfiguration quickly increases with the size and complexity of the organization, specially when human error is taken into account. We have not verified that such an attack is feasible; however, our data indicate that enterprise networking devices residing within large homogenous environments have been misconfigured with default root credentials.

6.2 Analysis of Results

In this section we present latest data gathered by our default credential scanner and preliminary results from our ongoing longitudinal study, tracking approximately 102,000 vulnerable devices over a span of four months. We also present statistics on the level of human and organizational responses received by the University regarding our scanning activities. Figure 6.3 shows a heat map of embedded device vulnerability rates across monitored countries.

Section 6.2.1 shows the breakdown of vulnerable embedded devices across **9 functional categories**: Enterprise Devices, VoIP Devices, Home Networking Devices, Camera/Surveillance, Office Appliances, Power Management Controllers, Vendor Issued Equipment, Video

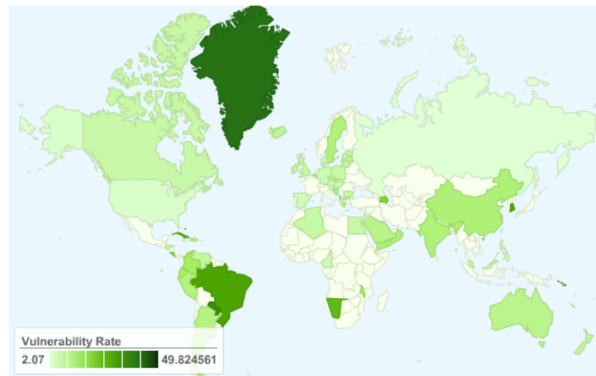

Figure 6.3: Embedded Device Vulnerability Rates of Monitored Countries (Threshold = 2%).

Conferencing Units, and Home Brew Devices. Section 6.2.2 shows the breakdown of vulnerable embedded devices across **6 continents**. Section 6.2.3 shows the breakdown of vulnerable devices across **5 types of organizations**: Educational, ISP, Private Enterprise, Government, and Unidentified.

6.2.1 Breakdown of Vulnerable Devices by Functional Categories

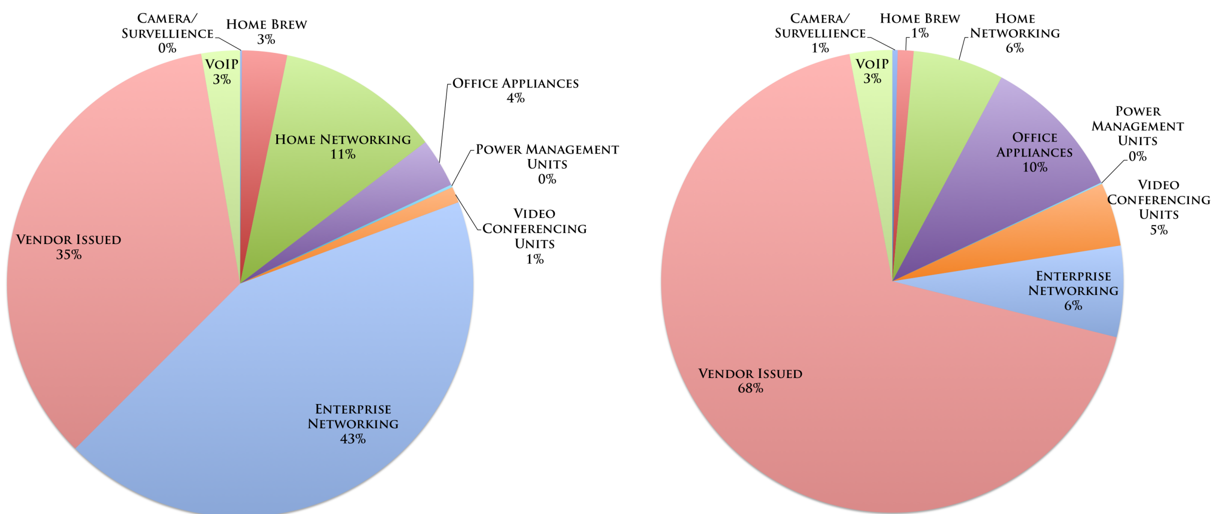

Figure 6.4: Discovered Candidate Devices (Left) and Vulnerable Devices (Right) by Functional Category.

We organized 73 unique embedded device types monitored by our scan into 9 functional

	Enterprise Devices	VoIP Devices	Home Networking
Vul. Rate	2.03%	15.34%	7.70%
Total Devices	1,689,245	104,827	445,147
	Camera / Surveillance	Office Appliances	Power Management
Vul. Rate	39.72%	41.19%	7.23%
Total Devices	5,080	132,991	7,429
	Vendor Issued Equipment	Video Conferencing	Home Brew
Vul. Rate	27.02%	55.44%	4.93%
Total Devices	1,362,347	43,349	122,159

Table 6.2: Vulnerability Rate by Device Category.

	Africa	Asia	Europe	North America	South America	Oceania
Vul. Rate	5.36%	21.69%	4.76%	4.12%	0.37%	17.98%
Total Devices	19,363	1,731,089	450,019	1,335,575	402,163	85,941

Table 6.3: Total Discovered Candidate Embedded Devices and Corresponding Vulnerability Rates By Geographical Location (Continental).

categories. Appendix A.1 contains the details of this categorization. Figure 6.2.1 shows the distribution of all discovered candidate embedded devices (left) and the distribution of vulnerable embedded devices (right) across the different functional categories. Table 6.2 shows the total number candidate embedded devices discovered within each functional category as well as their corresponding vulnerability rate.

- While **Vendor Issued Equipment** accounts for only 35% of all discovered candidate embedded devices, it represents 68% of all vulnerable embedded devices.
- While **Enterprise Networking Equipment** accounts for 43% of all discovered candidate embedded devices, it only represents 6% of all vulnerable embedded devices.

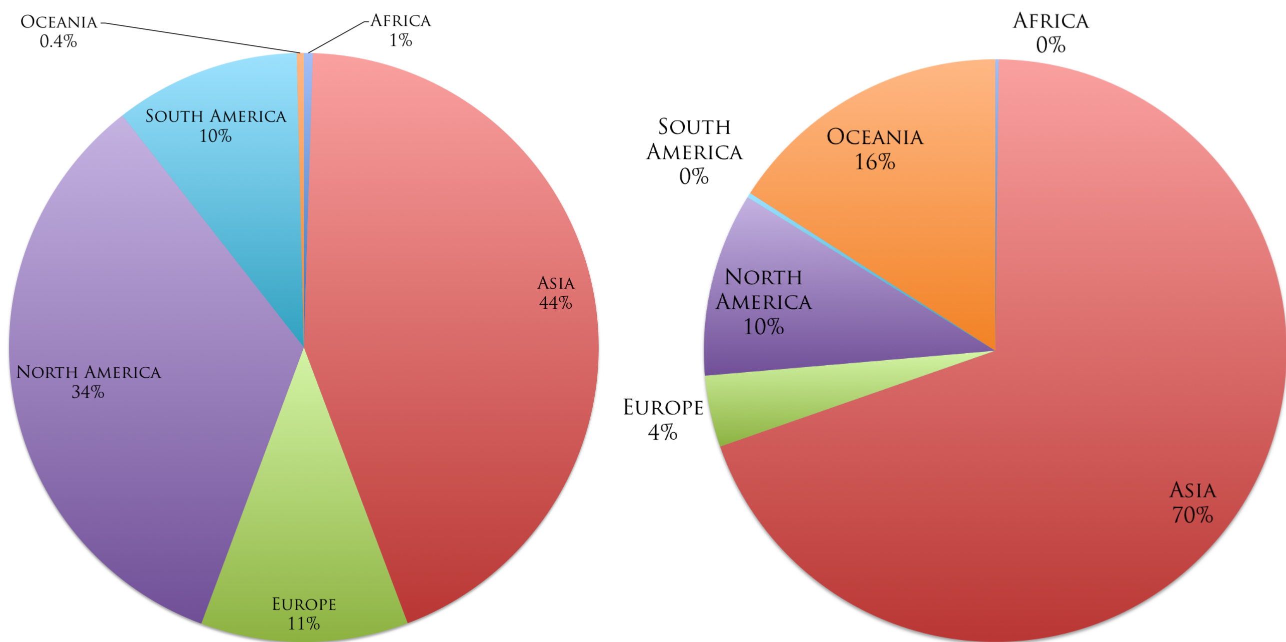

Figure 6.5: Discovered Candidate Devices (Left) and Vulnerable Devices (Right) By Geographical Distribution.

6.2.2 Breakdown of Vulnerable Devices by Geographical Location

Using the MaxMind GeoIP database, we categorized all discovered candidate and vulnerable embedded devices according to the continent in which they are located. Figure 6.2.2 shows the distribution of all discovered embedded devices (left) and the distribution of vulnerable embedded devices (right) across 6 continents. Table 6.2.2 shows the total number of candidate embedded devices as well as the corresponding vulnerability rate within each continent.

- **Asia** represents the continent with the most number of candidate embedded devices and contains over 70% of all discovered vulnerable embedded devices.
- **South Korea** contains the most number vulnerable embedded devices out of all monitored nations.
- While 34% of all discovered candidate embedded devices reside within **North America**, only 10% of all vulnerable embedded devices are found there.

	Educational	ISP	Private Enterprise	Government	Unidentified
Vulnerability Rate	32.83%	17.43%	16.40%	10.38%	2.54%
Total Devices	156,992	2,095,292	554,101	44,460	1,103,775
Unique Organizations	1371	2374	4070	494	9118

Table 6.4: Vulnerability Rate By Organization Type.

6.2.3 Breakdown of Vulnerable Devices by Organizational Categories

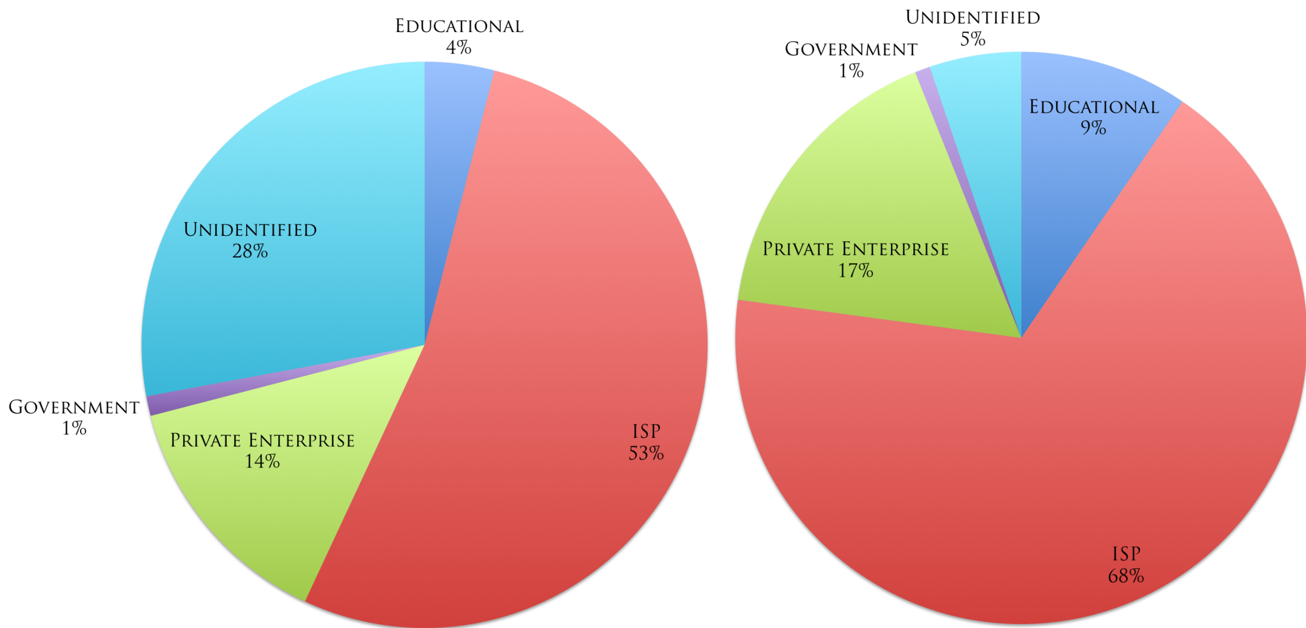

Figure 6.6: Discovered Candidate Devices (Left) and Vulnerable Devices (Right) by By Organization Type.

Using the MaxMind GeoIP Organization database, we categorized all monitored network ranges into 17,427 individual organizations. This was then divided into 4 general organization types: Educational, Internet Service Provider (ISP), Private Enterprise, and Government. 9118 organizations could not be accurately classified and were left in Unidentified category. Figure 6.2.3 shows the distribution of all discovered embedded devices (left) and the distribution of vulnerable embedded devices (right) across the 5 organization

types. Table 6.2.3 shows the total number of candidate embedded devices as well as the corresponding vulnerability rate within each organization type.

- **ISP** networks contain the most number of candidate embedded devices and house over 68% of all discovered vulnerable embedded devices.
- While **Educational** networks contain only a modest number of candidate embedded devices, they have the highest per category vulnerability rate of 32.83%

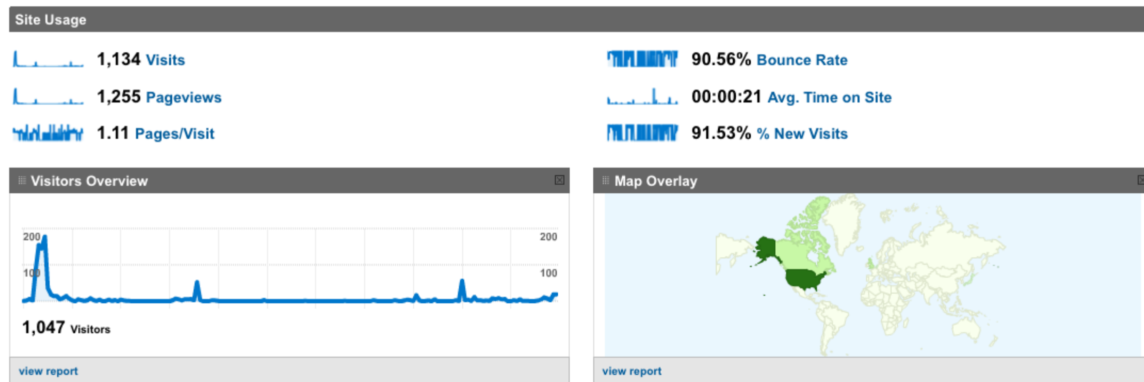

Figure 6.7: Daily Page Access Analytics For Scan Project Information Page [url anonymized]. Oct 19, 2009 - April 12, 2010.

6.2.4 Community Response to Default Credential Scanner Activity

The default credential scanner is designed to direct interested parties to a public webpage which describes the intent and methodology of our project[4]. Each IP address used by the scanner also hosts a public HTTP server which redirects visitors to the public project webpage. We tracked access to this webpage using Google Analytics as a way to gauge the global community's awareness of our scanning activities. Figure 6.7 shows the number and geographical distribution of visitors over the past six months. The initial spike of visitors in October 2009 coincided with the publication of an article regarding preliminary results of our project. Since then, our continuous scanning activity attracted **87 visitors** over the last 5 months.

[4]http://hacktory.cs.columbia.edu

Total Conversations	Opt-Out Requests	Request for Information, but Not Opt-Out
36	14	22
Tone of Counter-Party		
Supportive	Neutral	Hostile
14	15	7

Table 6.5: Email Correspondences Received from Network Operators Regarding Scanning Activity.

Devices Tracked	Currently Online	Tracked, Currently Vulnerable
102,896	54,429	52,661

Table 6.6: Preliminary Longitudinal Study Tracking 102,896 Vulnerable Devices Over 4 Months.

Table 6.5 shows a breakdown of all communications between the operators of the networks monitored by our scanner and our research team. The conversations were all initiated by the counter-party via email, usually requesting further information or to be excluded from the scan. We answered 36 conversations in total, 14 of which requested certain IP ranges to be permanently excluded. 1,798 /24 networks were excluded as a result of these requests. **61%** of all interested parties that detected our scanning activity decided to allow the scan to continue. The geographical location of the counter-parties correlates closely to the heat map in Figure 6.7. We did not receive any correspondence from ISP organizations or organizations from Asia, even though the majority of vulnerable devices were discovered within such IP ranges.

6.2.5 Preliminary Longitudinal Results

Table 6.6 shows the preliminary results of our longitudinal study. We retested 102,896 vulnerable embedded devices discovered at the end of December, 2009. As of April 20, 2010, 54,429 of the retested devices are still publicly accessible; out of which 52,661 devices remain vulnerable.

In other words, approximately **96.75%** of accessible vulnerable devices are still vulner-

able after a 4 month period, and factory default credentials have been removed from only **3.25%** of the same set of devices.

6.3 Remediation Strategy

The least sophisticated attacker modeled in this experiment can be defeated by simply discontinuing the use of well-known default credentials on embedded devices. However, the overall cost of implementing this naive mitigation strategy will likely be quite high in reality. In the unlikely event that all embedded device manufacturers universally agree to discontinue the use of well-known default passwords henceforth, we are still faced with the challenge of retroactively fixing the vulnerable legacy embedded devices in use throughout the world today. Therefore, it is reasonable to assume that the embedded security threat will likely persist and grow endemically for the near future. In order to effectively reduce the total population of vulnerable embedded devices in the wild, we must carefully consider the best methods for securing existing legacy devices. Since existing devices are by definition under the administrative control of some individual or organization, successful mitigation strategies must actively engage these network operators in order to fix the problem.

According to the data presented in Section 6.2, a few groups of network operators contribute disproportionally large numbers of vulnerable embedded devices to the global population. For example, we discovered over 300,000 vulnerable embedded devices operating in homogenous environments within two ISP networks in Asia. Overall, embedded devices operated by residential ISPs constitute over 68% of the entire vulnerable population. Since ISPs centrally manage large numbers of vulnerable embedded devices, they are the ideal candidates to engage to mitigate the embedded security threat.

6.4 Ethical Considerations of Such Studies

The scientific value of accurate, large-scale quantification of emerging threats in cyber-security is self-evident. In order to devise effective defenses against emerging security threats, such as the one presented in this section, one must first have an accurate understanding of the nature, scale and scope of the underlying causes of the problem. The

collection of such data raises numerous questions about how the security community should conduct such studies in a way that is both ethnical and legal.

For a more in-depth discussion of such matters, we refer the reader to [84].

6.5 Concluding Remarks

We presented the first quantitative measurement of embedded device insecurity on a global scale as well as a preliminary longitudinal study tracking vulnerable embedded devices over a 4 month period. We developed an embedded device default credential scanner capable of efficiently and safely identifying vulnerable embedded devices on the network. The scanner does this by testing whether one can remotely log into a device using its well-known manufacturer supplied default credentials. Using this scanner, which currently monitors 73 common embedded device types, we identify over **540,000** publicly accessible vulnerable devices in 144 countries. Vulnerable embedded devices were discovered in 17,427 unique organizations on 6 continents including government, ISP, private enterprise, educational and satellite provider networks. Preliminary results from our longitudinal study that tracked 102,896 vulnerable devices discovered in December 2009. Out of the 54,429 devices currently online from the original population, **96.75%** such devices still remain vulnerable today. By breaking down the observed vulnerable embedded device population across functional, geographical and organizational categories, we were able to identify key groups that contribute a disproportionately large number of vulnerable devices to the global population. Lastly, using observations derived from the presented data, we proposed a set of realistic mitigation strategies to effectively reduce the total population of vulnerable embedded devices. This study demonstrates that there is a very large population of trivially vulnerable embedded devices available for exploitation by the least sophisticated adversary. We posit that the size of this vulnerable population can be significantly increased by escalating the level of sophistication of the assumed attacker. Since no widely available host-based defenses exist, vulnerable embedded devices constitute a serious and pervasive security problem.

Chapter 7

Qualitative Assessment of Real-World Embedded Vulnerability

"To the past, or to the future. To an age when firmware is transparent. From the age of the DMCA, from the age of the tyrannous embedded device vendor, from a de-obfuscator of the secret sauce... greetings!"

7.1 Case-Study: Reliable Cisco IOS Exploitation

Over the past decade, Cisco IOS has been shown to be vulnerable to the same types of attacks that plague general purpose computers [78, 85]. Various exploitation techniques and proof-of-concept rootkits [80, 88] have been proposed. However, all current offensive techniques are impeded by an unintended security feature of IOS: diversity. As Felix "FX" Linder points out, Cisco IOS is not a homogenous collection of binaries, but a collection of approximately 300,000 diverse firmwares [80]. Although never intended as a defense against exploitation, this diversity makes the creation of reliable exploits and rootkits difficult.

Known proof-of-concept rootkits operate by patching specific locations within IOS. In the case of DIK [88], the rootkit intercepted a specific function responsible for checking password. The major drawback of this approach is that it relies on *a priori* knowledge of the location of this function. As previously noted, this knowledge is generally difficult to obtain with accuracy prior to attack. Therefore, any rootkit that depends on specific memory locations cannot be used reliably in large-scale attacks against the Internet substrate. Conversely, version-agnostic shellcode, combined with known vulnerabilities in IOS, makes such large-scale attacks against Cisco routers a feasible reality.

Cisco IOS firmware diversity, the unintended consequence of a complex firmware compilation process, has historically made reliable exploitation of Cisco routers difficult. With over *300,000* unique IOS images in existence, a new class of version-agnostic shellcode is needed in order to make the large-scale exploitation of Cisco IOS possible. We show that such attacks are now feasible by demonstrating two different reliable shellcodes that will operate correctly over many Cisco hardware platforms and all known IOS versions. We propose a novel two-phase attack strategy against Cisco routers and the use of offline analysis of existing IOS images to defeat IOS firmware diversity. Furthermore, we discuss a new IOS rootkit that hijacks *all* interrupt service routines within the router and its ability to use intercept and modify process-switched packets just before they are scheduled for transmission. This ability allows the attacker to use the payload of innocuous packets, like ICMP, as a covert command and control channel. The same mechanism can be used to stealthily exfiltrate data out of the router, using response packets generated by the router itself as the vehicle. We present the implementation and quantitative reliability measure-

ments by testing both shellcode algorithms against a large collection of IOS images. As our experimental results show, the techniques proposed in this chapter can reliably inject command and control capabilities into arbitrary IOS images in a version-agnostic manner. We believe that the technique presented in this chapter overcomes an important hurdle in the large-scale, reliable rootkit execution within Cisco IOS. Thus, effective host-based defense for such routers is imperative for maintaining the integrity of our global communication infrastructures.

For reliable, large-scale payload execution in IOS to be feasible, we must construct attacks and shellcodes that are version and platform agnostic. Towards this end, we outline a two-stage attack methodology as follows:

Stage 1: Leverage some IOS invariant to compute a host fingerprint. Using computed information, inject stage-2 shellcode. Furthermore, exfiltrate host fingerprint back to attacker.

Stage 2: Persistent rootkit with covert command and control capability. The attacker will use exfiltrated fingerprint data to construct a version specific rootkit, which is loaded via the second-stage shellcode.

The attacker is at a disadvantage when attempting an online attack. However, since all IOS images can be obtained, and since such images are not polymorphically mutated, an attacker can construct a large collection of version specific rootkits offline. If the attacker is able to simultaneously inject a simple rootkit and exfiltrate a host-environment fingerprint during the first phase of the attack, the attacker can then load a rootkit specifically parameterized for the exact IOS version of the victim router. Figure 7.1 shows the timeline of our proposed attack, which is intentionally broken into two phases to shift the advantage towards the attacker.

The two requirements of our first-stage shellcode, the need to reliably inject a basic second-stage rootkit and the need to accurately fingerprint the victim device, can be satisfied simultaneously. Both shellcodes presented in this chapter compute a set of critical memory locations within IOS's .text section. These memory addresses are used both as intercept points for our second-stage code, but also used to uniquely identify the exact micro-version

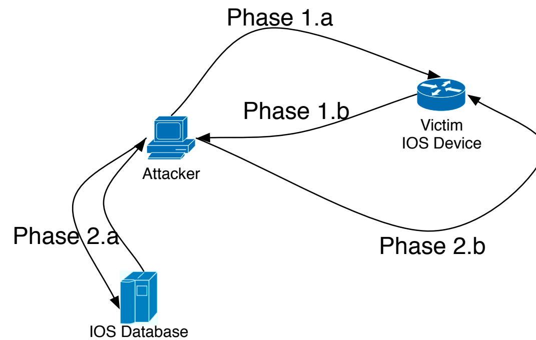

Figure 7.1: Timeline of two-stage attack against vulnerable IOS router of unknown hardware platform and firmware version. Attacker launches exploit with reliable shellcode (1.a). Shellcode installs rootkit and exfiltrates victim device's IOS fingerprint (1.b). The attacker finds exact IOS version from fingerprint by consulting offline database (2.a). The attacker then creates a version specific rootkit for victim and uploads it using 1.b rootkit (2.b).

of the victim's firmware. As figure 7.1 shows, this fingerprint data is exfiltrated back to the attacker and compared to a database of pre-computed fingerprints for all known IOS firmwares. As Section 7.1.7 shows, the fingerprints can be computed using simple linear-time algorithms and efficiently stored in a database. Pre-computing such fingerprints for all 300,000 IOS images should not take more than a few days on a typical desktop.

We present two different techniques for implementing this two-stage attack. The disassembling shellcode is discussed in Section 7.1.4. A novel interrupt hijack shellcode is discussed in Section 7.1.5. A stealthy exfiltration technique that modifies process-switched packets just before it is scheduled for transmission is discussed in Section 7.1.6. The intercept hijacking shellcode and the exfiltration mechanism built on top of it has several interesting advantages over existing rootkit techniques. First, the command and control protocol is built into the payload of incoming packets. No specific protocol is required, as long as the packet is punted to the router's control-plane. This allows the attacker to access the backdoor using a wide gamut of packet types, thus evading network-based intrusion detection systems. Hiding the rootkit inside interrupt handlers also allows it to execute *forever* without violating any watchdog timers. Furthermore, the CPU overhead of this shellcode will be distributed across a large number of random IOS processes. Unlike with

shellcodes that take over a specific process, the network administrator can not detect unusual CPU spikes within any particular process using commands like *show proc cpu*, making it very difficult to detect by conventional means.

The remainder of this chapter is organized as follows: Section 7.1.1 outlines the challenges of reliable IOS rootkit execution and provides motivation for the need for version-agnostic shellcodes. Section 7.1.2 presents a survey of advancements in Cisco IOS exploitation over the past decade and provides a timeline of public disclosures of significant vulnerabilities and exploitation techniques. Section 7.1.3 outlines a general two-stage attack strategy against unknown Cisco devices. Section 7.1.4 presents our first reliable IOS shellcode, a disassembling shellcode, which was first proposed by Felix Linder for PowerPC based Cisco devices. Section 7.1.5 presents our second reliable IOS shellcode. This shellcode hijacks all interrupt handler routines within the victim device and is faster, stealthier, and more reliable than our first shellcode. Experimental data, performance, overhead, and reliability measurements are presented in Section 7.1.7. Potential defenses against our proposed shellcodes are discussed in Section III. Concluding remarks are presented in Section 8.6. Lastly, the full source code of both shellcodes are listed in Appendix A.

Please note that the remainder of this chapter will focus on MIPS-based Cisco IOS. All code examples will be shown in MIPS. However, the techniques presented can be applied to PowerPC, ARM and even x86-based systems.

7.1.1 Motivation

Consider the availability of proof-of-concept exploits and rootkits, the wide gamut of high-value targets that can be compromised by the exploitation of devices like routers and firewalls, and the lack of host-based defenses within close-source embedded device firmwares. Such conditions should make the vast numbers of vulnerable embedded devices on the Internet highly attractive targets. Indeed, we have observed successful attempts to create botnets using Linux-based home routers [42]. As Section 7.1.2 shows, the necessary techniques of exploiting Cisco IOS and installing root-kits on running Cisco routers are well understood. However, an obstacle still stands in the way of reliable large-scale exploitation of Cisco devices: *firmware diversity.*

As Felix Linder and others have pointed out [80], there are over 300,000 unique versions of Cisco IOS. Diverse hardware platforms, overlapping feature-sets, cryptography export laws, licensing agreements and varying compilation and build procedures all contribute to create an operating environment that is highly diverse. Although unintentional and not strictly a defense mechanism, this firmware diversity has made the deployment of reliable attacks and shellcodes difficult in practice. Therefore, in order for IOS exploitation to be feasible and practical, reliable shellcode that operate correctly across large populations of IOS versions is needed.

As Lindner [80] demonstrates, certain common features within Cisco routers can be used to improve the chances of reliable execution of IOS shellcode. The disassembling shellcode concept was proposed in the same work. Building off this insight, we first tested the reliability of the proposed disassembling shellcode. While this technique works smoothly across all versions of IOS for several major hardware platforms, it failed on all versions of IOS for several popular platforms, including the Cisco 2800 series routers. Furthermore, its computational complexity frequently triggered watchdog timer exceptions, which logs a clear trace of the shellcode. Section 7.1.4 discusses the reason for this failure and several other drawbacks of this disassembling approach.

Looking to improve reliability and performance, we constructed a different shellcode by leveraging a common invariant of not only Cisco IOS, but all embedded systems, *interrupt handler routines*. Hijacking interrupt handlers is advantageous for several reasons. First, such routines can be identified by a single 32-bit instruction, **eret**, or *exception return*. The search for a single **eret** instruction reduces the computational complexity of the first-stage shellcode. Whereas the disassembling shellcode frequently causes watchdog timer exceptions on busy routers (See Section 7.1.4), the interrupt-handler hijacking first-stage shellcode executes quickly enough to avoid such timer exceptions, even on heavily utilized routers. Second, there are approximately two dozen interrupt handler routines on any IOS image, all of which are clustered around a common memory range. By using offline analysis of large numbers of IOS images, we can safely reduce the memory range searched by the first-stage shellcode to a small fraction of IOS's .text section, further improving the efficiency of the shellcode (See Figures 7.8 and 7.9).

As our experimental data shows, the two proposed shellcodes, along with our proposed data exfiltration mechanism presented in Section 7.1.6, combined with available vulnerabilities of Cisco IOS makes the large-scale of Cisco routers feasible. Weaponizing the techniques presented in this chapter to create worms that target routers is possible and can seriously damage the Internet substrate. Therefore, the development of advanced host-based defense mechanisms to mitigate such techniques should now be considered a necessity. Section III discuss potential host-based defenses for Cisco IOS and other similar embedded devices.

7.1.2 Cisco Exploitation Timeline

A timeline of significant advancements in offensive technologies against Cisco IOS is listed below.

FX, 2003: FX analyzes several IOS vulnerabilities and various exploitation techniques [78].

Lynn, 2005: Lynn described several IOS shellcode and exploitation techniques, demonstrating VTY binding shellcode [85].

Lynn, 2005: Cisco and ISS Inc. files injunction against Michael Lynn [1].

Uppal, 2007: Uppal releases IOS Bind shellcode v1.0 [110].

Davis, 2007: Davis releases IOS FTP server remote exploit code [38].

Muniz: 2008 Muniz releases DIK (Da IOS rootKit) [88].

FX, 2009: FX demonstrates IOS diversity, demonstrates reliable disassembling shellcode and reliable execution methods involving ROMMON [80].

Muniz and Ortega, 2011: Muniz and Ortega releases GDB support for the Dynamips IOS emulator, and demonstrates fuzzing attacks against IOS [89].

The techniques presented in this chapter extend the above line of work by introducing novel methods of constructing reliable IOS shellcodes and stealthy exfiltration, making large-scale exploitation feasible across all IOS-based devices.

7.1.3 Two-Stage Shellcode Execution Strategy

Sections 7.1.4 and 7.1.5 discusses two reliable shellcode techniques. Unlike existing IOS shellcodes, these two examples are designed to work in a two-phase attack. Figure 7.1 illustrates the attack process. In general, this attack first computes a series of memory locations that the second-stage shellcode will intercept to obtain minimal rootkit capability. This series of memory locations is also exfiltrated back to the attacker after the first-stage rootkit finishes execution. Using this information as a host fingerprint, the attacker queries a database of pre-computed fingerprints for all known IOS images to determine the exact micro-version of firmware running on the victim router. Once this is known, a version specific rootkit can be constructed automatically, then loaded onto the victim router via the rootkit installed by the first-stage shellcode.

Each shellcode computes a different set of features. In the case of the disassembling shellcode, a 2-tuple is computed: the address of an invariant string and the address of the password authentication function. In the case of the interrupt hijacking shellcode, a n-tuple is exfiltrated, containing a list of memory address for all interrupt handler routines on the victim device. Section 7.1.7 will discuss how accurately each feature-set can uniquely identify the micro-version of the victim IOS environment.

7.1.4 Cisco IOS DISASM Shellcode

First proposed by Felix Linder [80] for PowerPC-based routers, the disassembling shellcode scans the victim router's memory twice in order to locate and patch a target function based on some functional invariant, and works as follows:

A. Find String Addr: Scan through memory, looking for a specific string pattern. For example, '%Bad Secrets'.

B. Find String-Xref: With the string's memory location known, construct the instruction that loads this address. Rescan through memory, looking for code that references this string.

C. Patch Function: The data reference is located within the function we wish to find. Search within this function for the desired intercept point. For example, the function

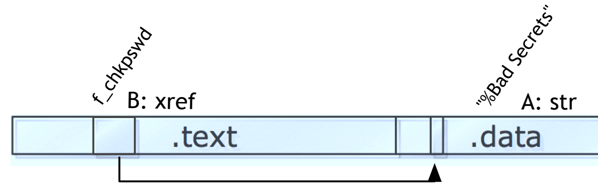

Figure 7.2: The disassembling shellcode first locates a known string (A), then locates a xref to this string (B). Once this xref location is found, the attacker can patch the function containing the xref. This shellcode requires two linear scans of IOS memory, one through the .data section, and a second one through the .text section.

entry point, or a specific branch instruction.

Any function that prints a predictable string can be identified and patched in this manner. A particularly useful function is the credential verification function, which prints '%Bad Secrets' when the wrong enable password is entered 3 times.

Figure 7.3 shows the disassembly of this function. We can bypass password authentication by overwriting a single *move* instruction, highlighted in red.

As experimental results in Section 7.1.7 shows, this first-stage shellcode reliably disables password authentication for all versions of Cisco 7200 and 3600 IOS images tested. However, it failed for all Cisco 2800 series IOS images.

In general, this type of disassembling shellcode is suitable for finding *direct* data references, and will fail to find *indirect* references. Indirect references can be identified at the price of computational complexity. In the case of Cisco routers, this limit is a very practical one. A watchdog timer constantly monitors every process within IOS, terminating any process running for longer than several seconds.[1] As Figure 7.11 shows, our implementation of the disassembling shellcode frequently caused watchdog timer exceptions to be thrown, leaving clear evidence of the attack in the router's logs.

Once the first-stage completes execution, the attacker can connect to the victim router with level 15 privilege, bypassing authentication. The attacker can then identify the exact

[1]The default watchdog timer value is 2 seconds.

```
text:829EB638                  move    $a2, $zero
text:829EB63C                  jal     sub_829EB50C
text:829EB640                  nop
text:829EB644                  bnez    $v0, loc_829EB66C
text:829EB648                  li      $v0, 1
text:829EB64C
text:829EB64C loc_829EB64C:                                  # CODE XREF: sub_829EB5C4+70|j
text:829EB64C                  slti    $v0, $s0, 3
text:829EB650                  bnez    $v0, loc_829EB60C
text:829EB654                  move    $a0, $s5
text:829EB658                  lui     $v1, 0x6396
text:829EB65C                  addiu   $a0, $v1, aBadSecrets  # "\n%% Bad secrets\n"
text:829EB660
text:829EB660 loc_829EB660:                                  # CODE XREF: sub_829EB5C4+2C|j
text:829EB660                  jal     sub_806607AC
text:829EB664                  nop
text:829EB668                  move    $v0, $zero
text:829EB66C
text:829EB66C loc_829EB66C:                                  # CODE XREF: sub_829EB5C4+80|j
text:829EB66C                  lw      $ra, 0x90+var_8($sp)
text:829EB670                  lw      $s5, 0x90+var_C($sp)
text:829EB674                  lw      $s4, 0x90+var_10($sp)
text:829EB678                  lw      $s3, 0x90+var_14($sp)
text:829EB67C                  lw      $s2, 0x90+var_18($sp)
text:829EB680                  lw      $s1, 0x90+var_1C($sp)
text:829EB684                  lw      $s0, 0x90+var_20($sp)
text:829EB688                  jr      $ra
text:829EB68C                  addiu   $sp, 0x90
text:829EB68C  # End of function sub_829EB5C4
```

Figure 7.3: A disassembly of a typical f_chkpasswd. The string xref is the first highlighted block. The second highlighted block is the single instruction, which can disable password authentication in IOS. While these addresses vary greatly, they can be reliably computed at exploitation time by the disassembling shellcode.

IOS version with a number of methods by using the router's administrative interface. While this backdoor gives the attacker persistent control of the device, it is not covert. Section 7.1.5 shows our interrupt hijack shellcode, which installs an equivalent backdoor through a covert channel, using payloads of IP packets punted[2] to the router's CPU. In our demonstration, we use a large collection of arbitrary UDP and ICMP packets to load complex rootkits into the router's memory.

The video demonstration of the disassembling shellcode running on a Cisco 7204 and 12.4T IOS can be found at [30].

7.1.5 Interrupt Hijacker Shellcode

As Section 7.1.4 shows, the disassembling shellcode can be used reliably, at least for several major hardware platforms, to locate and intercept a critical function, which handles credential verification in IOS. However, this shellcode must search through large portions of the router's memory *twice* in order to identify the target string reference and the target function. This required computation frequently triggered the router's watchdog timer,

[2]A packet is punted to a router's CPU when it cannot be handled by its linecards, and must be inspected and process-switched.

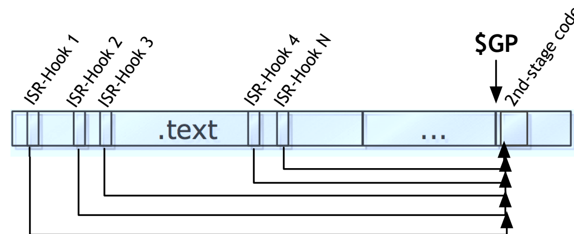

Figure 7.4: The interrupt hijack shellcode first locates all *eret* (exception return) instructions within IOS's .text section. The second-stage rootkit is then unpacked inside the $gp memory area (which is unused by IOS). All *eret* instructions, and thus all interrupt service routines are hooked to invoke the second-stage code. We now have reliable control of the CPU by intercepting all interrupt handlers of the victim router.

leaving evidence of the shellcode in the router's log. In general, we want to minimize the amount of computation required by the first-stage shellcode to evade the watchdog timer and avoid any perceivable CPU spike or performance degradation.

7.1.5.1 First-stage shellcode

The interrupt hijacking shellcode performs a single scan through the router's .text section, locating and intercepting the end of *all* interrupt handler routines, and works as follows:

Unpack second-stage: The second-stage shellcode, which contains a basic rootkit, is unpacked and copied to the location pointed to by **$gp**, the general purpose register.

Locate ERET instructions: Scan through memory, looking for all [**eret**] instructions. All such addresses are stored and exfiltrated for offline fingerprinting (See Section 7.1.6).

Intercept all interrupt handler routines: Hijack all interrupt handler routines by replacing all **eret** instructions with the [**jr $gp**] instruction.

The **eret**, or *exception return* instruction takes no operands, and is represented by the 32-bit value [0x42000018]. As the name suggests, eret is the last instruction in any interrupt handler routine, and returns the CPU context back to the previous state before the interrupt

was serviced. Once intercepted, any interrupt serviced by the CPU will invoke our second-stage code, giving us persistent, perpetual control of the CPU to execute our second-stage rootkit.

7.1.5.2 Second-stage shellcode

The second-stage is essentially a simple code loader, which continuously monitors the router's IOMEM range, looking for incoming packets with a specific format. The second-stage rootkit locates packet payloads marked with a 32-bit magic-number. Such packets contain a 4-byte target address value, a 1-byte flag and variable length data (up to the MTU of the network).

When such a packet is found, the second-stage either copies the variable length data to the 4-byte memory location as indicated by the packet, or jumps the PC to a specified location. In order to load such packets into the victim router's IOMEM, the attacker simply needs to craft IP packets that will be *punted* to the router's CPU. Any packets that must be inspected by the router's control-plane will suffice.[3] For demonstration purposes, we used a variety of UDP and ICMP packets. Such packets need not even be destined to the router's interface. Various malformed broadcast and multicast packets are automatically punted to CPU and copied to the router's IOMEM region (on the 7200 platform).

When the first-stage shellcode completes, the attacker has:

Host fingerprint: The list of *eret* addresses is exfiltrated to the attacker, and will uniquely identify the micro-version of the victim's IOS (See Section 7.1.7).

Perpetual CPU control: The second-stage code, copied to the global-scope memory area, is invoked each time an interrupt is serviced by IOS.

We now present a second-stage rootkit that monitors all incoming packet-data entries, or payloads of packets that have been *punted* to the router's control-plane for process switching, continuously scanning incoming packets for commands from the attacker. Using

[3]Different router platforms have different packet handling capabilities, trying to reduce the number of packets that must be punted to CPU. However, packets destined to routing processes, like BGP, OSPF, along with ICMP and SNMP packets are typically punted to CPU.

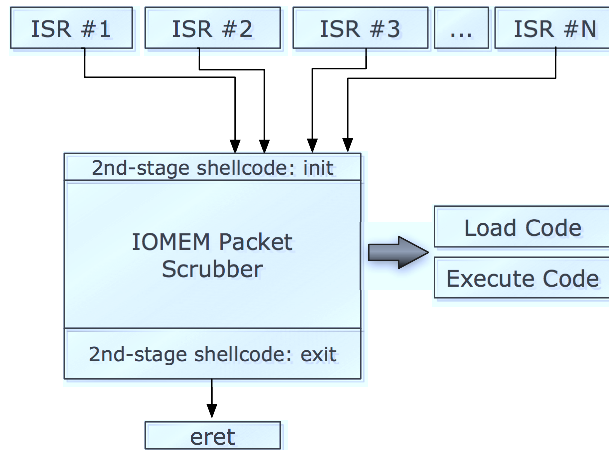

Figure 7.5: Interrupt hijack second-stage rootkit. Each time any ISR (interrupt service routine) is invoked, the rootkit will seek through the latest punted packets within IOMEM for specially crafted command and control packet payloads.

the second-stage rootkit presented below, the attacker can load and execute arbitrary code by crafting command and control packets in the payload of *any* IP packet that will be punted to the router's CPU. The attacker can stealthily assemble large programs within the router's memory by using a wide spectrum of different packet types, like ICMP, DNS, mDNS, etc.

Since we intercept *all* interrupt handlers, the second-stage code is invoked whenever *any* interrupt is serviced, including timer interrupts, interrupts from linecards, etc. Therefore, a very limited amount of computation (under a hundred instructions) can be done inside interrupt handlers without seriously impacting the router's stability and performance. Figure 7.5 illustrates a second-stage rootkit that is designed specifically for high-frequency execution within interrupt handlers. Each time the second-stage code is invoked, the rootkit scans through the linked-list of *packet data* entries located within IOMEM. Figure 7.6 shows a snapshot of this data structure in IOMEM. Each time the second-stage code is invoked, it scans through a fixed number of packet-data entries, looking for specially marked packets containing a 32-bit magic number. The number of packet data entries scanned at each iteration directly impacts the reliability of this method (See Section 7.1.7).

Once such an entry is found, the second-stage code does the following:

```
0F000190  AB 12 34 CD FF FE 00 00  00 00 00 00 62 CB C8 30  .4......b0
0F0001A0  60 6E 53 7C 0F 00 02 D0  0F 00 00 64 80 00 00 88  `nS|......d..
0F0001B0  00 00 00 01 00 00 00 00  00 00 00 01 64 6D 20 B8  ............dm
0F0001C0  AF AC EF AD 00 00 00 00  00 00 00 00 00 00 00 00  ............
0F0001D0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0F0001E0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0F0001F0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0F000200  00 00 00 00 00 00 00 00  00 00 BA 27 1A B2 7C 6C  ..........'.|l
0F000210  CA 00 5D 10 00 00 08 00  45 00 00 2E 00 01 00 00  .].....E.......
0F000220  FF 01 A7 CB 0A 00 00 02  0A 00 00 01 00 00 DF 3C  ...........<
0F000230  00 00 00 00 58 66 31 4C  54 38 33 44 00 00 00 21  ....Xf1LT83D...!
0F000240  20 52 6F 31 34 32 00 00  00 00 00 00 00 00 00 00   Ro142..........
0F000250  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0F000260  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0F000270  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0F000280  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0F000290  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0F0002A0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0F0002B0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0F0002C0  00 00 00 00 00 00 00 00  00 00 00 00 FD 01 10 DF  ..............
0F0002D0  AB 12 34 CD FF FE 00 00  00 00 00 00 62 CB C8 30  .4......b0
0F0002E0  60 6E 53 7C 0F 00 04 10  0F 00 01 A4 80 00 00 88  `nS|.........
```

Figure 7.6: Highlighted words, left to right, top to bottom. 1: Pointer to previous packet data node. 2. Pointer to next packet data node. 3. Exfiltration request magic pattern. 4. Beginning of next packet data entry, pointed to by 2.

Parse OpCode: Parse the packet data entry, looking for a 1-byte opcode, along with a 4-byte target address value.

If OpCode = Load: The second-stage code will copy the content of the remainder of the packet-data entry to the 4-byte address indicated by the packet.

If OpCode = Run: The second-stage code will jump the PC to the target address indicated by the packet.

The second-stage code is designed to execute with high frequency, but in small bursts. It will execute approximately 100 instructions each time it is invoked, which allows us to scan through several *dozen* packets before returning control of the CPU back to the interrupt handler, and thus the preempted IOS code.

Note that the head of the packet-data linked-list structure is located in a well-known address within the IOMEM region, which is mapped to the same virtual-memory address regardless of router model or IOS version [11], making this packet-scrubbing technique reliable across all IOS versions on many router platforms.

The intercept hijacking shellcode has several interesting advantages over existing rootkit techniques. First, the command and control protocol is built into the payload of incoming packets. No specific protocol is required, as long as the packet is punted to the router's control-plane. This allows the attacker to access the backdoor using a wide gamut of packet types, thus evading network-based intrusion detection systems. Hiding the rootkit inside

interrupt handlers also allows it to execute *forever* without violating any watchdog timers. Furthermore, the CPU overhead of this shellcode will be distributed across a large number of random IOS processes. Unlike with shellcodes that take over a specific process, the network administrator cannot detect unusual CPU spikes within any particular process using commands like *show proc cpu*, making it very difficult to detect by conventional means.

7.1.6 Stealthy Data Exfiltration

After the first-stage shellcode completes, it yields a sequence of memory addresses where the *eret* instruction is located. As Section 7.1.7 shows, this data can serve as a host fingerprint, allowing the attacker to identify the exact micro-version of the victim's IOS firmware. Several known methods can be used to exfiltrate this fingerprint back to the attacker. Note that the entire memory sequence need not be transmitted, as a simple hash of the data will suffice. The attacker can carry out a VTY binding [110] to open a reverse shell back to the attacker, or simply use the console connection to generate an ICMP packet back to the attacker. Depending on which services are publicly accessible on the router, the attacker can inject the fingerprint data into the server response. For example, the HTTP server's default HTML can be modified.

These methods will most likely leave some detectable side-effect that can trigger standard network intrusion detection system. We present a new exfiltration technique that modifies the payload content of process-switched packets just prior to transmission. The data is exfiltrated using packets generated by router itself, thus making the detection of this covert channel more difficult.

Once a packet is punted to the router's control-plane, it is copied from the network interface hardware to the router's IOMEM region. For efficiency, when such a packet is process-switched, the packet-data entry is not copied. Instead, the pointer to this data is simply moved from the router's RX queue to its TX queue. Once there, the packet is scheduled for transmission, then forwarded appropriately. If the attacker can *modify* the contents of the packet-data entry before it is transmitted, such payloads can be used as a vehicle for stealthy exfiltration. Figure 7.7 illustrates this exfiltration process.

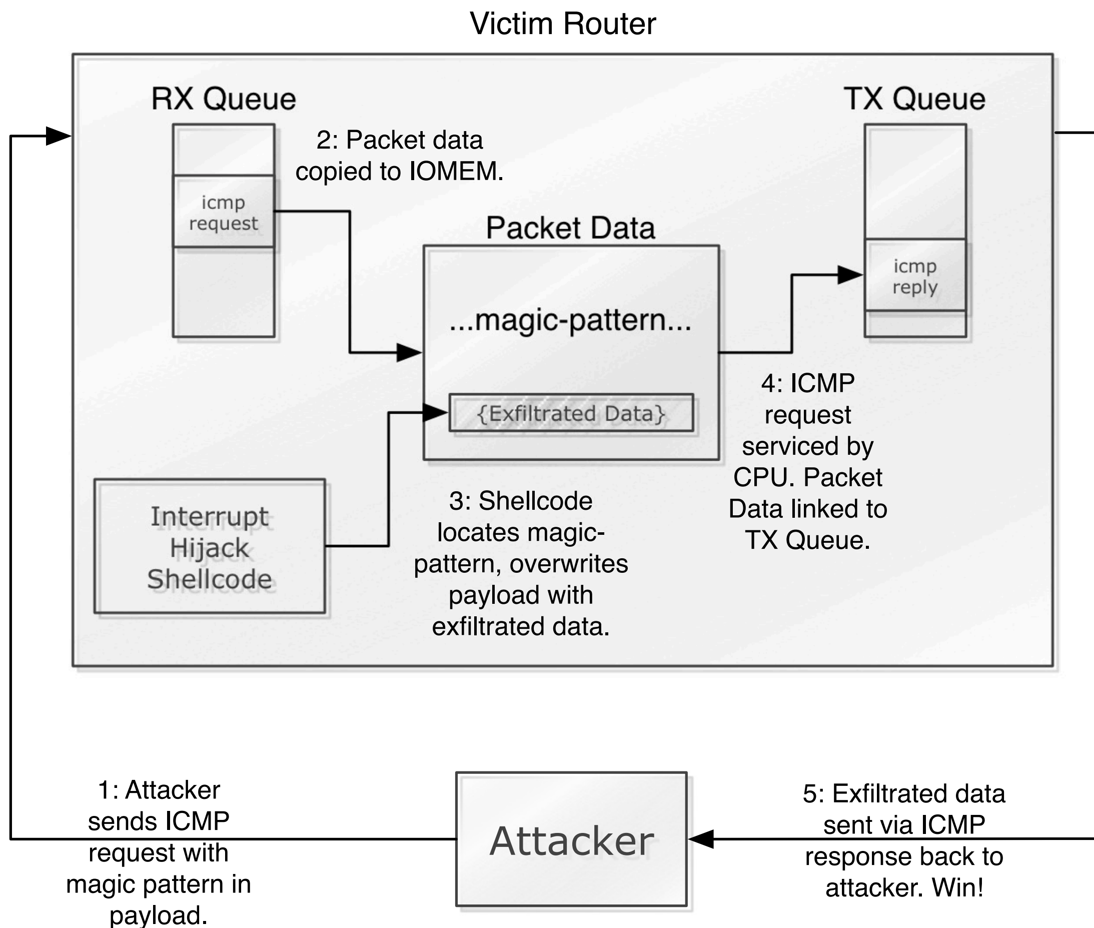

Figure 7.7: Data exfiltration through forwarded packet payload. 1: The attacker crafts a packet with a magic pattern in its payload indicating exfiltration request. 2: Packet payload is copied into a *packet data* structure. 3: Rootkit locates magic pattern, overwrites remaining packet with exfiltrated data. 4: Packet is process-switched. The packet data entry is linked to the TX queue. 5: The requested data is sent back to the attacker inside an ICMP response packet.

	Hardware Platform	Sample Size	Reliability
xref	7200	76	100%
eret	7200	76	100%
xref	3600	52	100%
eret	3600	52	100%
xref	2800	31	0%
eret	2800	31	100%

Table 7.1: Reliability of the disassembling shellcode and interrupt hijack shellcode when tested on 159 IOS images.

This type of manipulation is highly time-sensitive, as the attacker will typically only have a few milliseconds after the packet's arrival to locate and manipulate its payload, before the packet is transmitted. However, since the second-stage rootkit is invoked with *every* interrupt, it can precisely intercept the desired packet before it is placed on the TX queue, allowing the attacker to use the same covert command and control channel for data exfiltration. Section 7.1.7 discusses the performance of this exfiltration method. Due to the timing constraints of the interrupt hijack shellcode and various race conditions related to process-switching and CEF, not all exfiltration requests sent by the attacker will be processed. In practice, approximately 10% of exfiltration requests are answered by the rootkit when tested on an emulated 7204VXR/NPE-400 router.

The video demonstration of this exfiltration method can be found at [30].

7.1.7 Experimental Data

The reliability of the disassembling shellcode, presented in Section 7.1.4 and the interrupt hijack shellcode, presented in Section 7.1.5, are shown in Table 7.1. Three major Cisco router platforms, the 7200, 3600, and 2800 series routers are tested. The two proposed shellcode algorithms are tested against 159 IOS images, ranging from IOS version 12.0 to 15.

The computational overhead of both shellcodes are shown in Figure 7.10 for a typical 7200 IOS 12.4 image. In some instances, the disassembling shellcode does not terminate

	2	4	8	16	32	64
reliability	0%	0.67%	1.29%	4.67%	5.38%	10.10%

Table 7.2: Reliability of exfiltration mechanism when the number of packet-data nodes searched per invocation varies. Searching more than 64 nodes caused the test router to behave erratically.

in time, which triggers a watchdog timer exception to be thrown and logged (See Figure 7.11). The interrupt hijack shellcode consistently completed first-stage execution without triggering any watchdog timer exception.

Table 7.2 shows the reliability of the exfiltration mechanism presented in Section 7.1.6, as the number of packet-data nodes searched during each interrupt-driven invocation. The reliability rate is calculated by counting the number of exfiltration requests the rootkit successfully answered out of 150 ICMP requests. Searching more than 64 nodes at each invocation caused the router to behave erratically, sometimes leading to crashes.

Figure 7.8 and 7.9 shows the distribution of features found by the disassembling shellcode and interrupt hijack shellcode respectively across 159 tested IOS images. Note that while the string reference tends to be more widely distributed, interrupt handler routines are typically found in a much smaller area. While the exact location of interrupt handlers still remain unpredictable, this concentration allows the interrupt hijack first-stage shellcode to search through a relatively small range of memory when compared to the disassembling shellcode.

7.1.8 Defense

In order to categorically mitigate against the offensive techniques described in this chapter, host-based defenses must be introduced into the router's firmware. Since persistent rootkits must modify portions of the router's code, a self-checksumming mechanism can be injected into IOS to detect and prevent unauthorized modification of IOS itself. This can be generalized to all regions of the router, which should remain *static* during normal operation of the router, and can include large portions of the .data, ROMMON, and .text sections.

Such a defensive mechanism, called Software Symbiote, has been proposed by the author

Figure 7.8: Distribution of the location of the password authentication function. This location varies greatly across the IOS .text segment, forcing the disassembling shellcode to search a large region.

to solve this problem. Software Symbiote is discussed in Chapter 8. We have shown that Symbiotes can be injected into Cisco IOS in a version-agnostic manner to provide continuous integrity validation capability to the host router. Our experimental results show that such Symbiotes can detect unauthorized modification to any static region of IOS in approximately *300ms*. Symbiotic defenses of this type is the focus of ongoing research.

7.1.9 Concluding Remarks

We present a two-stage attack strategy against Cisco IOS, as well as two unique multi-stage shellcodes capable of reliable execution within a large collection of IOS images on different hardware platforms. The disassembling shellcode, first proposed by Felix Linder [80] operates by scanning through the router's memory, looking for a string reference, allowing the attacker to disable authentication on the victim router. The interrupt hijack shellcode injects a second-stage shellcode capable of continuously monitoring incoming *punted*

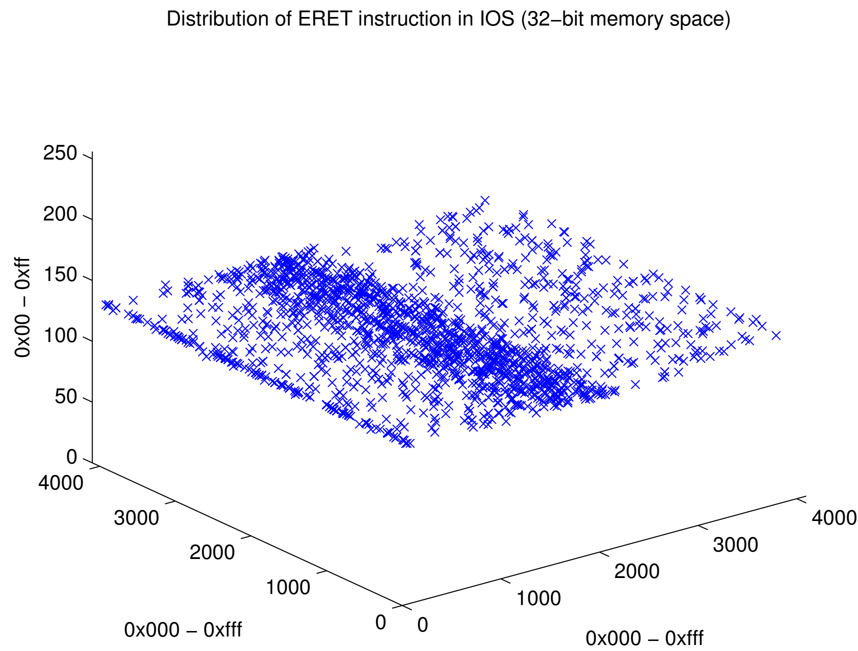

Figure 7.9: Distribution of the location of *eret* instructions over 162 IOS images. These locations mark the end of all interrupt service routines in IOS, and tend to be concentrated within a predictable region of IOS.

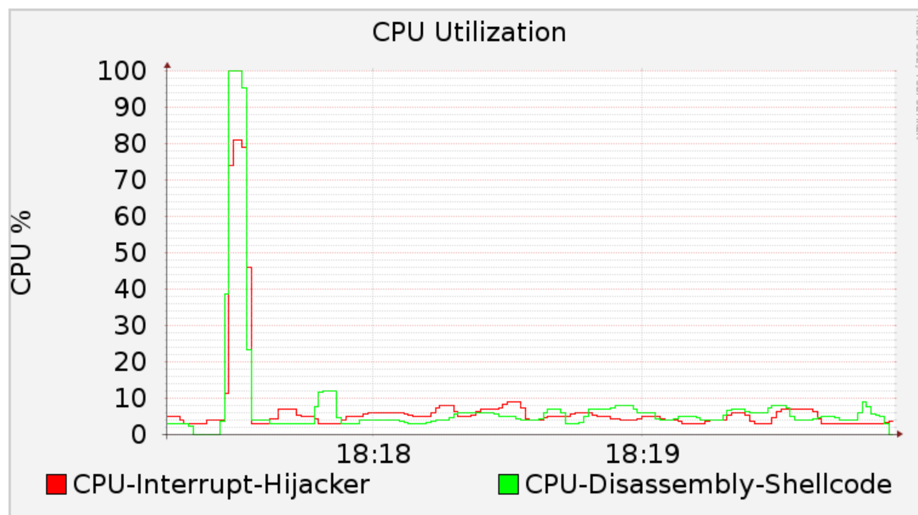

Figure 7.10: CPU utilization of 7204 router during the first-stage execution of both the disassembling and intercept hijack shellcodes. Note that the interrupt hijack shellcode is simpler, requires less CPU and thus avoids watchdog timer exceptions.

```
Router>
*May  1 16:22:56.599: %SYS-3-CPUHOG: Task is running for (2020)msecs,
 more than (2000)msecs (3/2),process = Exec.
-Traceback= 0x62641C3C 0x6068D914 0x606A9BD8 0x6074E780 0x6074E764
*May  1 16:22:58.599: %SYS-3-CPUHOG: Task is running for (4020)msecs,
 more than (2000)msecs (3/2),process = Exec.
-Traceback= 0x62641C3C 0x6068D914 0x606A9BD8 0x6074E780 0x6074E764
*May  1 16:23:00.603: %SYS-3-CPUHOG: Task is running for (6020)msecs,
 more than (2000)msecs (4/2),process = Exec.
-Traceback= 0x62641C3C 0x6068D914 0x606A9BD8 0x6074E780 0x6074E764
*May  1 16:23:02.599: %SYS-3-CPUHOG: Task is running for (8012)msecs,
 more than (2000)msecs (5/2),process = Exec.
-Traceback= 0x62641C3C 0x6068D914 0x606A9BD8 0x6074E780 0x6074E764
*May  1 16:23:03.103: %SYS-3-CPUYLD: Task ran for (8516)msecs, more t
han (2000)msecs (5/2),process = Exec
```

Figure 7.11: CPU intensive shellcodes will be caught by Cisco's watchdog timer, which terminates and logs all long running processes. The disassembling shellcode, although reliably bypasses password verification, consistently triggers the watchdog timer, generating the above logs, which give precise memory location of the shellcode.

packets for specially crafted command and control packets from the attacker. The attacker can use this covert backdoor by sending a wide gamut of packet types, like ICMP and UDP, with specially crafted payloads. In both shellcodes, when the first-stage completes execution, a host fingerprint is computed and exfiltrated back to the attacker. Using this data, the attacker can accurately identify the exact micro-version of IOS running on the host router. Using the second-stage rootkit, the attacker can then upload a version specific rootkit, which can be pre-made *a priori* for *all* IOS images, onto the victim router. This two-stage attack scenario allows the attacker to compromise any vulnerable IOS router as if the specific version of the firmware is known, bypassing the software diversity hurdle which has obstructed the reliable, large-scale rootkit execution within Cisco routers.

7.2 Case-Study Firmware Modification: HP-RFU

7.2.1 Overview

The ability to update firmware is a feature that is found in nearly all modern embedded systems. We demonstrate how this feature can be exploited to allow attackers to inject malicious firmware modifications into vulnerable embedded devices. We discuss techniques

for exploiting such vulnerable functionality and the implementation of a proof of concept printer malware capable of network reconnaissance, data exfiltration and propagation to general purpose computers and other embedded device types. We present a case study of the HP-RFU (Remote Firmware Update) LaserJet printer firmware modification vulnerability, which allows arbitrary injection of malware into the printer's firmware via standard printed documents. We show vulnerable population data gathered by continuously tracking all publicly accessible printers discovered through an exhaustive scan of IPv4 space. To show that firmware update signing is not the panacea of embedded defense, we present an analysis of known vulnerabilities found in third-party libraries in 373 LaserJet firmware images. Prior research has shown that the design flaws and vulnerabilities presented in this chapter are found in other modern embedded systems. Thus, the exploitation techniques presented in this chapter can be generalized to compromise other embedded systems. Modern embedded devices exist in large numbers within our global IT environments and critical communication infrastructures. Embedded systems like routers, switches and firewalls constitute the majority of our global network substrate. Special purpose appliances like printers, wireless access points and IP phones are now commonplace in the modern home and office. These appliances are typically built with general purpose, real-time operating systems using stock components. They are capable of interacting with general purpose computers as general purpose computers themselves.

The diverse and proprietary nature of embedded device hardware and firmware is thought to create a deterrent against effective wide-spread exploitation. While such claims of embedded security fundamentally reduce to security through obscurity, it is nonetheless claimed by embedded device vendors to provide security for their products[56].

To demonstrate that such claims of embedded security are overly optimistic and that emerging embedded exploitation techniques and embedded system malware pose a threat to the security of our existing networks, we present the following four contributions:

General firmware modification attack description: We present firmware modification attacks, a general strategy that is well-suited to the exploitation of embedded devices. This strategy aims to make arbitrary, persistent changes to victim devices' firmware by leveraging design flaws commonly found within embedded software. Firmware modification attacks

can affect entire families of devices adhering to the same system design flaw, transcending operating system versions and instruction set architectures. The HP-RFU vulnerability presented in this chapter affects MIPS- and ARM-based printers alike, regardless of their underlying software implementation. We discuss the general preconditions for and the process of leveraging firmware modification attacks against modern embedded devices.

HP LaserJet firmware modification case study: We use a firmware modification vulnerability recently discovered by the authors in nearly all HP LaserJet printers[60] to present a real-world case study of the development cycle of such attacks against common embedded devices. We present the threat model characterization, vulnerability analysis and threat assessment of HP-RFU and show a full exploit against the vulnerability. The entire process, from discovery to the implementation of the final attack and malware package, took approximately two months, and was carried out using public vendor information readily available on the Internet and required a hardware budget of under $2,000. This attack is effective against the majority of LaserJet printers currently in production and affects a large number of installed devices. While it is difficult to divine the actual size of the vulnerable device population, HP shipped 11.9 million such units in a single quarter of 2010 alone[63].

The design flaws identified in the HP remote firmware update functionality can be seen in other modern embedded systems. Thus, the attack strategy we present can be generalized and applied to other vulnerable embedded device types. We discuss the offensive potential of our proof of concept printer malware and its impact on the efficacy of traditional network defense doctrine.

Vulnerable population / patch propagation analysis: The severity of the HP-RFU attack is further increased due to the ubiquitous nature of the vulnerable population. While firmware fixes have been released by the vendor, mitigation of the vulnerability discussed in this chapter ultimately depends on end-users diligently updating firmware. Applying firmware updates on mission-critical embedded systems can be cumbersome and daunting[9]. It is not surprising that we have found that this diligence is lacking, which favors the attacker.

We present the results of exhaustive scans of IPv4 to show the distribution of all publicly accessible, vulnerable LaserJet printers on the Internet. We have identified over 90,000

unique vulnerable printers inside numerous government organizations, educational institutions and other sensitive environments. We periodically fingerprint the specific firmware version of each tracked device in order to analyze the rate and pattern of firmware patching throughout the world. We believe this data will shed light on the inefficacy of the patch cycle for large populations of embedded devices as compared to patch propagation patterns within general purpose computer populations. Firmware patch propagation data for the first two months following the official release of firmware updates for 53 printer models[59] is presented in this chapter. Initial data indicates a global patch level of approximately 1.08%. Furthermore, 24.8% of all patched printers still have open telnet interfaces with no root password configured (a default setting).

Vulnerable third-party library analysis: Mandatory firmware update signature verification was introduced by the vendor on some vulnerable LaserJet printer models following the disclosure of the HP-RFU vulnerability. This mitigates the specific vulnerability discovered by the authors. However, mandatory firmware signature verification allows known vulnerable code to be signed and verified. It does not remove the actual vulnerabilities within the signed firmware, nor will it detect or mitigate the exploitation of the actual vulnerability.

We present the results of automated analysis of a large collection of LaserJet printer firmwares released over the last decade, including the latest firmwares released in response to the HP-RFU disclosure. We analyzed all publicly available firmware images for 63 models of HP LaserJet printers. By cross-referencing the specific version numbers of third-party libraries like OpenSSL and zlib found within firmware updates with known vulnerabilities for those specific library versions, we conclude that a large number of vendor-issued firmwares are released with multiple known vulnerabilities. In some cases, we identified recently released firmware updates containing vulnerabilities in third-party libraries that have been known for over eight years. We identified third-party libraries with known vulnerabilities in 80.4% of all firmware images analyzed.

The remainder of this section is organized as follows: Section 7.2.2 describes the general firmware modification attack strategy and surveys such existing attacks against embedded devices. Section 7.2.3 discusses the discovery of the HP-RFU vulnerability and the

subsequent proof of concept attack and malware development. Section 7.2.6 discusses the real-world offensive potential of our proof of concept attack. The distribution of publicly accessible vulnerable LaserJet printers and initial firmware patch propagation telemetry is presented in Section 7.2.7. Vulnerable third-party library analysis of 373 vendor-issued firmware updates is presented in Section 7.2.8. Lastly, we survey related works and ongoing work in the area of host-based embedded defense and vulnerability analysis in Section 7.1.2.

7.2.2 Firmware Modification Attack

Firmware modification attacks aim to inject malware into the target embedded device. Predictions of firmware modification attacks against printers are almost a decade old[6]. Firmware modification attacks can be carried out either as standalone attacks or as secondary attacks following initial exploitation using traditional attack vectors.

Standalone firmware modification attacks manipulate firmware update features instead of exploiting flaws in the victim software. For example, the firmware modification case study presented in Section 7.2.3 utilizes the remote firmware update feature within HP LaserJet printers. This attack vector is not unique to the vulnerable devices discussed in this chapter. Other ubiquitous embedded systems like ATM machines, smart battery controllers, keyboards, enterprise routers and PBX equipment are also vulnerable to such attacks. Similar standalone firmware modification attacks [22, 24, 51, 64, 86, 93] have recently been reported.

The standalone firmware modification strategy is well-suited to embedded exploitation in general for the following reasons:

Feasibility: Firmware update is an ubiquitous feature found in modern embedded devices. Previous work[8, 36, 51] shows that a large number of embedded devices have firmware update features that are not sufficiently protected by proper user authentication. Many devices that require authentication before allowing firmware updates are vulnerable to trivial administrative interface bypass attacks[107]. Furthermore, net-booted embedded devices that use insecure protocols like TFTP to retrieve their configurations and firmware are vulnerable to standard OSI Layer 2 attacks.

Fail-Safe: Firmware update mechanisms usually mandate integrity and model verification

prior to execution of the actual firmware modification. Malicious firmware update packages sent to incompatible embedded devices are rejected and ignored. This relaxes the reconnaissance and accuracy requirements for the attacker and reduces the penalty of a misdirected attack. For example, the final malicious binary described in Section 7.2.3 contains a single RFU image targeting a precise printer model. However, if the exact model of the victim printer is unknown, multiple malicious RFU commands covering all potential printer models can be sent sequentially without damaging the printer. Furthermore, each RFU command need not contain a full printer OS image, which is at least several megabytes in size. A bare-bones OS boot loader can be sent instead. Such a loader could be implemented to be at most several hundred kilobytes in size (the development of this offensive technique is outside the scope of this chapter).

Platform Independence: Attacks that manipulate firmware update features within the vulnerable device do not need to depend on specific software vulnerabilities in the victim and will generally work across many models of the same device, even across different machine architectures. For example, the HP-RFU vulnerability manipulates a feature of the LaserJet firmware, which is supported across nearly all printer models and is common among MIPS- and ARM-based devices.

While mandatory firmware signature verification can mitigate standalone firmware modification attacks, this countermeasure is not the panacea of embedded security. Firmware modification attacks can be carried out as a secondary payload following the successful exploitation of the embedded device via traditional vectors like memory modification attacks. Firmware content is typically stored in rewritable, nonvolatile memory like flash. Embedded operating systems generally lack the fine-grain privilege separation and execution isolation found in modern operating systems; even when available in later builds, vendors oftentimes choose to not utilize these memory isolation features. Furthermore, for embedded operating systems with process and memory isolation, vulnerabilities within the kernel or privileged processes can still allow an attacker to make persistent changes to the device. For example, prior research has demonstrated that it is possible to make persistent modifications to the boot ROM portion of enterprise routers using only software operations[35]. Thus, countermeasures like authentication and firmware signature verification cannot fully prevent

firmware modification attacks on embedded systems with vulnerable attack surfaces.

Section 7.2.3 illustrates the development cycle of a typical firmware modification attack and embedded malware. Section 7.2.8 presents vulnerable third-party library analysis for a large corpus of HP LaserJet firmware images.

7.2.3 Case Study: HP LaserJet Exploitation

The HP-RFU firmware modification vulnerability[60] was discovered unintentionally when the authors attempted to inject host-based defenses into network printers. The HP LaserJet family was chosen because of its popularity and commanding market share[63]. The LaserJet P2055DN model was chosen as our initial target device.

Analysis of the HP LaserJet firmware revealed a reliably exploitable design flaw that allows remote attackers to make persistent modifications to the printer's firmware by printing to it.

In order to inject host-based defenses into any target hardware, the original firmware must be unpacked and analyzed. In the case of prior work on Cisco IOS routers, this process was straightforward[4]. However, unpacking and analyzing HP LaserJet firmware images presented several challenges. Figure C.1 of the Appendix shows the hex dump of a RFU file.

The remote firmware update for the P2055DN printer begins with standard PJL (Printer Job Language) but enters into an undocumented language called *ACL*. Approximately 7 MB of binary data follow. Initial static analysis[5] revealed no recognizable filesystem headers and no function preambles for any known machine architecture inside the RFU binary.

Without further analysis, a key design flaw became apparent: the firmware modification mechanism is coupled with the printing subsystem, which must accept incoming requests in an unauthenticated manner as per general specification. As confirmed by vendor documentation[55], the RFU file is *printed* to the target device via the raw-print protocol over standard channels like TCP/9100, LPD and USB. Various other vendors also use the

[4]IOS images are simple ZIP files with slightly non-standard headers.

[5]We used standard industry practices of loading the image into IDA Pro, fixing the memory mapping, and so forth. A detailed discussion of reverse engineering is outside the scope of this chapter.

same update strategy.

When a print job is received by the printer's job-parsing subsystem, a proprietary mechanism is used to determine the presence of a valid firmware update package. If a PJL command containing a valid RFU package is present, the integrity of the RFU payload is verified and decompressed. The payload's unpacked binary data is then written to persistent storage within the target printer, thereby modifying the printer's firmware.

Once the RFU binary structure was obtained through standard hardware and software reverse engineering methods, we discovered that it was possible to pack arbitrary executable code back into a legitimate RFU package in a PJL command. This command can then be embedded into a malicious document or sent directly to the victim printer to arbitrarily and persistently modify its firmware. Such an attack does not affect the printing of the legitimate carrier document and only makes the printer unavailable for approximately 90 seconds. The printer will continue to respond to network requests throughout most of the firmware update process. Thus, the attack will likely go completely unnoticed by users and network monitoring systems.

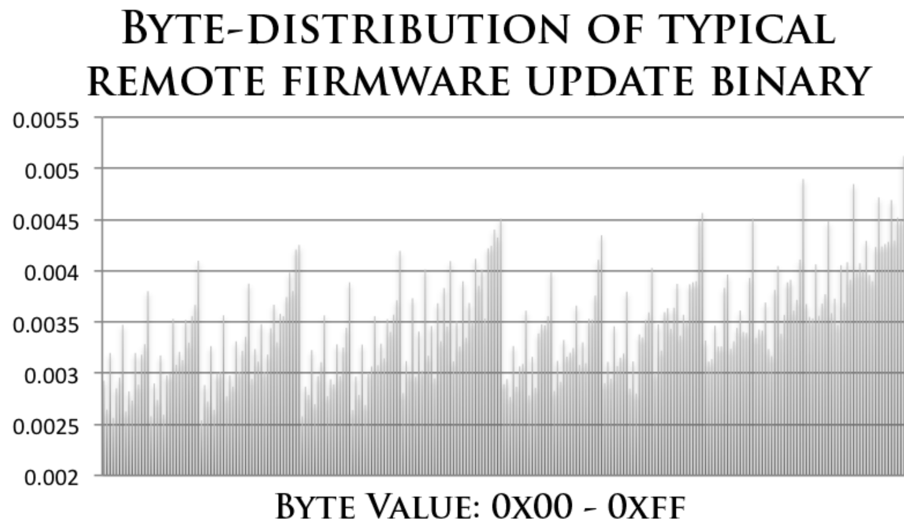

Figure 7.12: Byte value distribution histogram of a typical RFU file. Distribution suggests that the data is compressed and not encrypted.

7.2.4 Discovery Process

Initial static analysis of the original RFU binary revealed no printable strings, no known filesystem headers and no recognizable executable binaries. We concluded that the binary payload was likely either encrypted or compressed. Figure 7.12 shows the byte distribution histogram for a typical RFU binary payload for the P2055DN printer. The histogram suggests that the binary blob is compressed and not encrypted as common encryption algorithms typically generate high-entropy ciphertext, which was not observed.

Manual inspection of the binary revealed a simple package header structure containing a short checksum field followed by multiple entries of the same data structure, containing the compressed and uncompressed size of each firmware component and its target address within the printer's persistent storage address space. This header is shown in Figure C.2 of the Appendix.

Figure 7.13: Formatter board for LaserJet P2055DN. Dump of the onboard SPI flash revealed RFU format and integrity checking algorithm.

The printer's formatter board hardware components were desoldered and reverse engineered. Figure 7.13 shows the actual formatter board inside the target device. Figure 7.14 illustrates the main components found on the P2055DN's primary control (formatter) board. Manual inspection revealed that the system was powered by a Marvell SoC. Aside from the machine architecture (ARM), no other information was publicly available due to the proprietary nature of the chip. However, the SPI flash chip is a stock component with a publicly available datasheet.

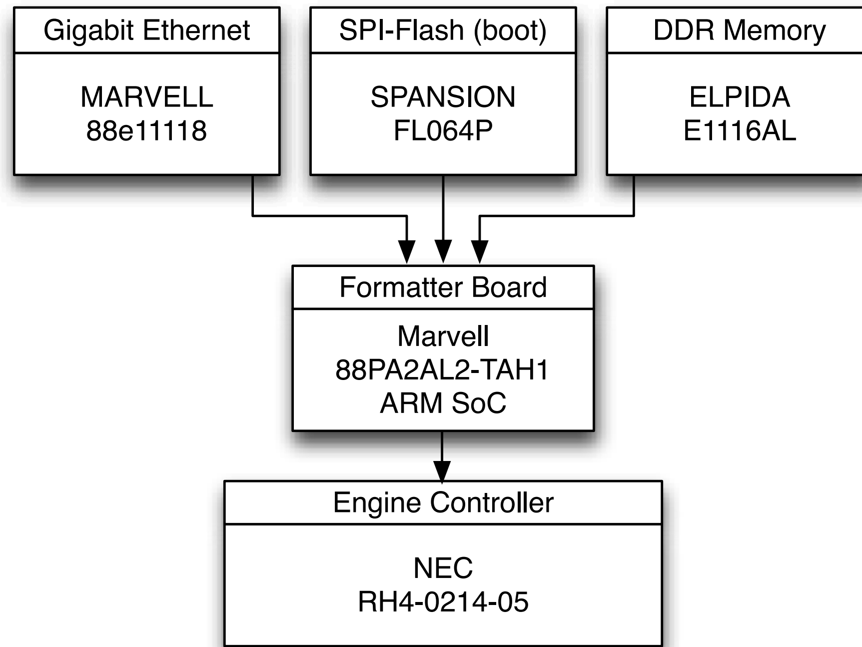

Figure 7.14: Logical block diagram of the major components used on the LaserJet P2055DN formatter board. The Spansion boot flash was key to our reverse engineering effort.

The main SoC on the formatter board uses the Spansion flash chip as a boot device. This chip has 8 MB of storage and communicates with the main Marvell SoC via a Serial Peripheral Interface (SPI) using a simple command protocol defined in its datasheet. In order to extract the contents of the flash chip, a SPI chip dumper was implemented using an Arduino[7] to perform the actual I/O. Figure 7.15 shows the physical hardware setup connecting the SPI boot flash chip to the Arduino board.

Analysis of the boot loader code revealed the binary structure and compression algorithm used in the RFU format. Manual inspection of the flash chip content revealed a boot image layout shown in Figure 7.16.

A factory reset RFU image was found inside the boot flash. This image is immediately preceded by a boot loader containing the code that validates and parses RFU images. IDA Pro[62] disassembled the boot loader binary. The resulting assembly code revealed that the RFU image is validated using a trivial checksum function and compressed using a common algorithm.

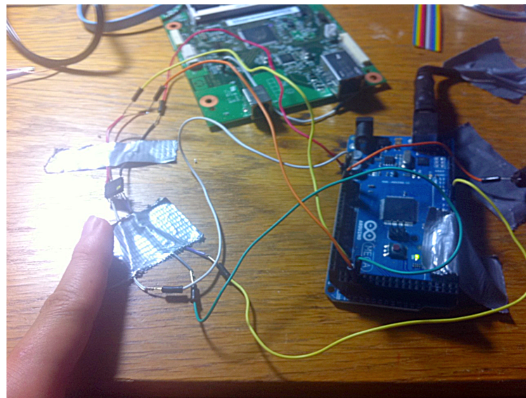

Figure 7.15: The SPI flash chip was physically removed then connected to an Arduino for boot code extraction.

Furthermore, the specific version of the compression library used to process RFU images appear to have several known arbitrary code execution vulnerabilities[15, 16, 17]. Section 7.2.8 presents an analysis of vulnerable third-party libraries found in a large number of firmware images released by the vendor.

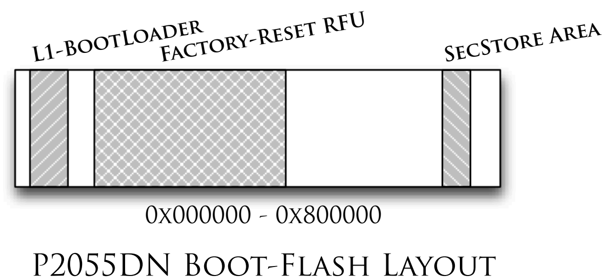

Figure 7.16: Boot image layout on the SPI flash chip. The level-1 boot loader contains code that validates, unpacks and decompresses the factory reset RFU allowing us to reverse engineer the binary RFU format and compression algorithm.

7.2.5 Proof of Concept Printer Malware

Static analysis of the extracted boot flash code revealed the precise RFU binary structure, checksum and compression algorithms used. This information allowed the authors to write HPacker, a tool that takes an uncompressed ARM ELF image as input and returns a valid

compressed PJL update command as output.

The malicious PJL command can be printed directly to the target printer or embedded within various document formats (an example is included in Figure C.3 of the Appendix). Either way, once the PJL command is sent to the victim printer, it will recognize the print job as containing a valid firmware update and allow the attacker to make arbitrary modifications to the victim's firmware storage area.

The unpacked RFU package for the P2055DN contains over a dozen files. The main file of interest is the binary OS image, a single 14 MB ELF image containing the VxWorks operating system and various other vendor-specific additions.

The creation of the proof of concept malware essentially reduced to creating a VxWorks rootkit capable of:

- Command and control via covert channel
- Print job snooping and exfiltration
- Autonomous and remote-controlled reconnaissance
- Multiple device type infection and propagation to the Windows operating system and other embedded devices
- Reverse IP tunnel to penetrate perimeter firewalls
- Self-destruction

A video discussing the technical mechanics of this rootkit and a demonstration of its capabilities is publicly available[32].

The VxWorks OS image found within the RFU binary contains a complete socket library[114] and direct access to the underlying network transceiver hardware. The creation of the proof of concept code was mainly an exercise in identifying and intercepting the proper pieces of binary within the VxWorks image.

No host-based security mechanism exists within the firmware image. Thus, the attacker is free to make arbitrary changes to the victim device. As long as the functionality and general performance of the device is not altered, detection of firmware modification is not possible without careful removal and inspection of the hardware inside the printer.

Several challenges arose during the construction of the proof of concept code. The Vx-

Works image extracted from the RFU package contained no symbol information. Locating the appropriate socketlib, print job processing and raw network I/O binary interfaces within the binary proved non-trivial.

We developed a set of IDA-Python scripts to perform standard control-flow analysis of the target binary around code that we manually identified as network-facing. This effort was expedited by a patch made to the VxWorks kernel, which redirected debug messages destined for the UART to a TCP connection. Using these two mechanisms, a dynamic analysis environment was created to probe network-facing code, which eventually yielded a small set of functions likely to be libraries used by multiple pieces of unrelated code. Function prototype data was taken from available VxWorks documentation and used as a final check to positively identify each library function.

Typically, the malware would be optimized, compressed, packed and broken up to fit within gaps inside the original firmware or placed within dynamically allocated memory. However, since the attacker controls the firmware storage area absolutely, we added a new section within the ELF header marked with *rwx* privileges. This gave us more than sufficient space to implement all the previously mentioned malware functionality. In total, 2,800 lines of assembly were written to create the proof of concept malware.

7.2.6 Threat Model and Assessment

We present the threat model and assessment analysis for the HP-RFU vulnerability presented in Section 7.2.3.

7.2.6.1 Threat Model Characterization

The HP-RFU vulnerability exploits a design flaw in the firmware update mechanism found in nearly all LaserJet printers. In order to achieve persistent firmware modification on the victim device, the attacker must deliver a malicious PJL command to the raw-printing processing subsystem of the target. This can be done by using the following attack types:

Active Attacks require the attacker to directly trigger the firmware update process by actively connecting to the printer and sending it the malicious PJL command over the printer's raw-printing port.

Reflexive Attacks are akin to reflexive cross-site scripting attacks where malicious firmware update commands are embedded in passive data that is passed along to the user of the victim device. For example, the final binary package of the HP-RFU attack can be embedded inside innocuous-looking documents and sent to unwitting users, perhaps in the form of an academic paper or resume. In this reflexive attack scenario, the actual attack is launched when the malicious document is *printed*.

7.2.6.2 Threat Assessment

Figure 7.17 illustrates an advanced persistent attack scenario where a compromised printer is used as a reconnaissance tool and offensive asset. Once the malware package is delivered to the victim printer, it can be used to carry out firmware modification attacks against other embedded devices like other printers, IP phones, and video conferencing units. Compromised embedded devices can be used to establish reverse IP tunnels back out to the Internet, giving the attacker direct access to the secured internal network. These devices can also be used to carry out standard network attacks like ARP cache poisoning and act as offensive assets to further compromise general purpose computers and other embedded devices behind the victim's perimeter defenses.

No host-based security mechanisms exist on the compromised printer. Thus, the presence of malware on this device will most likely go undetected if the functionality of the device is not affected. The compromised printer is an ideally situated stealthy asset that can be used as a fail-safe device allowing the attacker re-entrance into the victim network even if all compromised general purpose machines are neutralized. Contrary to the sensationalized media coverage regarding the HP-RFU vulnerability, it would be unwise for the attacker to destroy a compromised printer physically[6].

The HP-RFU vulnerability disclosure is described in CVE-2011-4161[18]. As Section 7.2.7 shows, there are currently over 90,000 vulnerable LaserJet printers publicly accessible over the IPv4 Internet.

[6] While it has been demonstrated and stated in the initial reports that using the printer's fuser as an ignition source to create fire is not possible, physical destruction of the printer is possible via multiple methods.

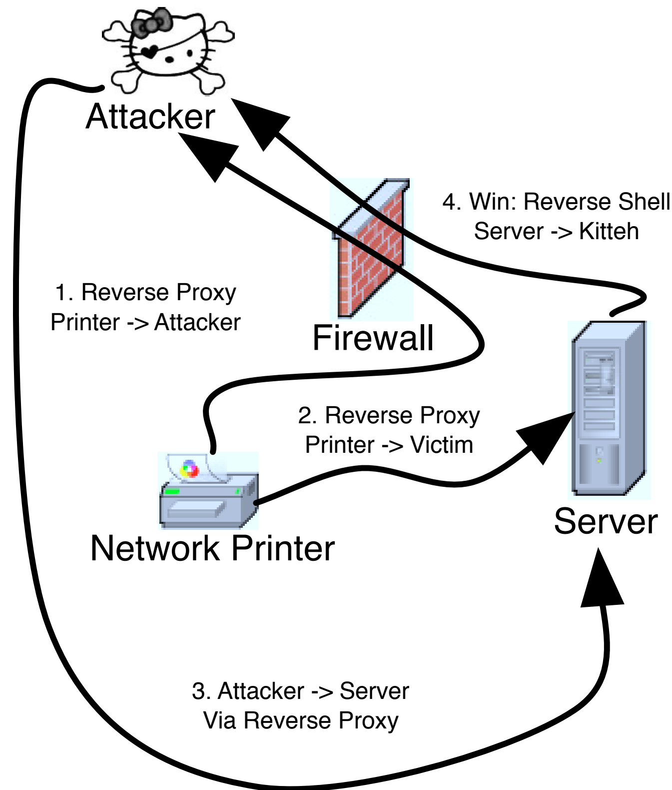

Figure 7.17: Typical advanced persistent threat attack scenario involving compromised printers.

7.2.6.3 Compounding Factors

The following factors compound the severity of the HP-RFU vulnerability:

No authentication prior to firmware update: The PJL/RFU mechanism is coupled with the raw-printing protocol, a cleartext protocol that does not support authentication. Any party who is allowed to use the victim printer can carry out a firmware modification attack against the printer. Therefore, the attacker does not need to have direct IP connectivity to the victim printer even in the active attack scenario because the malicious payload can be relayed by intermediate print servers.

RFU feature enabled by default: The majority of the firmwares we analyzed enable

the remote RFU update feature by default. Network-printing typically requires the printer to be reachable via TCP/9100. Since arbitrary binary traffic is allowed in the raw-printing protocol by specification, it is difficult to detect and stop malicious PJL commands at the network layer. Recent research suggests that it may be possible to use languages like PostScript to compute a valid, malicious PJL command on the victim printer when the malicious document is processed[24]. If so, this will significantly increase the difficulty of detection of this type of attack on the network level or within print servers.

Poor and incomplete configuration interface: The configuration interface of many "advanced" security features does not exist on the printer's HTTP or Telnet administrative interfaces. For example, disabling the remote RFU feature and setting PJL passwords can only be done through a separate enterprise printing management tool called HP Web Jetadmin (WJA)[61]. This is a 315 MB program that requires the installation of Windows-based web and SQL servers and is generally not practical for average users without enterprise IT support.

RFU feature cannot be disabled: Several LaserJet models, including the P2055DN used in our initial experimentation, do not support any way to disable the remote RFU feature, even through Web Jetadmin. As far as the authors are aware, prior to the release of the second version of the security bulletin[58], no combination of available configurable settings could disable the vulnerable feature on these printers. Furthermore, these models were not included in the first release of the security bulletin[57], since security bulletins released by the vendor must contain an acceptable mitigation method. Since no firmware fix was available, the devices most affected by the HP-RFU vulnerability were not listed in the initial vendor disclosure document.

Potential for irreversible, permanent malware injection: The SPI boot flash chip used on the P2055DN formatter board supports a One-Time-Programmable (OTP) feature[104] that allows areas of memory within the chip to be programmed and locked *permanently*. This is an irreversible operation that is typical for similar flash components. If the malicious malware package injected into the boot flash chip of the printer took advantage of this feature, removal of the malware would be impossible without physical removal of the compromised component.

7.2.7 Vulnerable Device Population Analysis

Vendors of general purpose operating systems and popular applications have deployed large-scale distribution networks to automatically update host software with little to no user interaction. However, no such widely deployed distribution exists to push patches and firmware updates to embedded systems.

The results presented in this section indicate that approximately 1.08% of vulnerable HP LaserJet printers have been patched worldwide, despite the public announcement of the HP-RFU vulnerability and the rapid release of firmware updates by the vendor (see Table 7.3).

This highlights the ineffectiveness of simple public release of firmware updates for vulnerable embedded devices. Empirical evidence suggests that vulnerable embedded devices will persist for a long period of time as compared to vulnerable general purpose computers. The threat will persist unless proactive firmware update mechanisms are developed for legacy embedded systems. However, a more proactive firmware update mechanism may also be exploited in firmware modification attacks.

7.2.7.1 Methodology

In order to quantify the number of printers that are vulnerable to the HP-RFU attack, we scanned the IPv4 Internet for publicly accessible HP printer web, telnet, SNMP and raw-print server sockets. Model numbers and firmware datecodes were gathered by employing the following methods:

- "@PJL INFO ID" command over TCP/9100
- "@PJL INFO CONFIG" command over TCP/9100
- "@PJL INFO PRODINFO" command over TCP/9100
- SNMP GET using "public" as the community string
- Model-specific banner scraping over TCP/23,80

7.2.7.2 Findings

In the two months following the official release of firmware updates for the HP-RFU vulnerability, we identified 90,847 unique HP printers that are publicly accessible over the IPv4 Internet. Firmware version data is collected periodically for each device. Table 7.3 shows our findings.

Potentially vulnerable printers	90,847
Printers with identifiable firmware datecode	74,770
Number of patched printers	808
Overall patch rate	1.08%

Table 7.3: Observed population of printers vulnerable to the HP-RFU attack on IPv4.

Patching vulnerable printers to the latest firmware does not necessarily secure the printer. We probed each printer for other well-known vulnerabilities and common misconfigurations that can result in unrestricted root-level access to the printer. Table 7.4 lists the vulnerabilities, including a ChaiVM vulnerability FX exploited in 2003[79] (this talk also discussed the potential for firmware modification).

Vulnerable printers are grouped into five general organizational types: *educational*, *private enterprise*, *military*, *civilian government* and *Internet service providers*. Tables 7.5 and 7.6 show the distributions of the average age of the firmware images currently installed across different organization types and continents, respectively. The firmware age is taken from the datecode in the response from the devices' administrative interfaces. Organizational and geographic data were gathered though the DNS, Internet Routing Registry (IRR) whois or commercial geolocation databases.

The above data is a lower bound on the number of vulnerable LaserJet printers on the Internet since it does not include devices behind firewalls or NATs or in other private networks..

[7] The ChaiVM EZLoader allows unsigned .jar files to be installed[98].

[8] A remote crash vulnerability exists in Virata EmWeb R6.0.1[91].

Unrestricted Telnet	50,500
Unrestricted ChaiVM[7]	27,570
Vulnerable Virata EmWeb[8]	2,740

Table 7.4: Observed population of printers vulnerable to attacks other than HP-RFU on IPv4.

	Count	Avg Age (years)	Oldest Firmware
Education	48,626	4.13	1993-08-20
ISP	4,650	3.70	1994-10-12
Enterprise	2,842	4.02	1992-12-16
Military	201	4.63	1999-10-30
Government	126	4.33	1996-12-20

Table 7.5: Organizational distribution of vulnerable printers.

In the months following the HP-RFU vulnerability disclosure, we observed 808 unique vulnerable printers that have been updated to firmware versions that mitigate the problem. We also observed 211 printers that did not require updated firmware to be invulnerable to the HP-RFU. However, out of these 1,019 devices, 24.8% (253) of them still have open telnet interfaces with no root passwords configured.

Approximately 64% of all vulnerable printers were located in North America. Over 65% of all vulnerable printers were found within the networks of educational institutions world-wide.

We also identified the following populations of vulnerable printers within two notable organizations:

- United States Department of Defense: *201 printers*
- Hewlett-Packard: *6 printers*

	Count	Avg Age (years)	Oldest Firmware
N. America	47,840	4.46	1992-12-16
Europe	14,196	4.16	1993-08-20
Asia	10,353	3.77	1998-09-02
Oceania	1,081	4.79	1998-09-02
S. America	673	3.43	1999-01-27
Africa	60	4.56	2001-04-26

Table 7.6: Geographical distribution of vulnerable printers.

7.2.8 Vulnerable Third-Party Libraries

Mandatory firmware signature verification was introduced by the vendor[59] in response to the disclosure of the HP-RFU vulnerability. While this effectively mitigates the specific attack presented in Section 7.2.3, we believe this response is inadequate for at least two reasons:

Signed firmware ≠ secure firmware: Firmware signature verification guarantees that the binary data to be processed at firmware update time originated from a trusted source within the vendor's organization. Vulnerable code that is signed by the vendor remains vulnerable to exploitation. This mechanism does not prevent firmware or memory modification attacks in general and thus contributes little to the overall security of the embedded device.

Signed firmware prevents independent third-party defense development: Mandatory signature verification that only accepts firmware updates signed by the vendor will categorically prevent all non-vendor issued code from running. This makes the injection of legitimate third-party host-based defenses into vulnerable firmware images impossible.

In order to show that firmware signing as the sole security mechanism is inadequate, we present the results of the automated analysis of the third-party library vulnerabilities in a set of 373 firmware update packages issued by the vendor over the last decade. The dataset includes 358 RFUs released prior to the disclosure of HP-RFU as well as 15 RFUs released as part of SSRT100692 rev.3. The printer models and firmware images analyzed are listed

in Table C.1 of the Appendix.

7.2.8.1 Methodology

All RFU images were unpacked and decompressed. Embedded filesystems (LynxFS) were extracted from the decompressed data. Extracted executables and shared objects were pattern-matched against known ASCII and binary signatures to detect the presence of specific versions of two specific third-party libraries: zlib and OpenSSL.

While this process suggests the presence of specific versions of third-party libraries in the analyzed firmware updates, no analysis was performed to check whether the libraries can be invoked by the attacker, or that the known vulnerabilities are reliably exploitable on the printers' machine architectures. This is the topic of ongoing research.

We present findings for the following third-party library vulnerabilities found in 373 vendor-issued firmware updates:

zlib: *CA-2002-07, CERT-*{*68062, 238678*} Discovered in 2002, zlib ver. 1.1.3 and earlier have a double free bug that allows arbitrary code execution[15]. In 2005 the vendor was notified of a buffer overflow in zlib ver. 1.2.1 and 1.2.2[17]. The vendor was notified of a DOS condition in zlib ver. 1.2.0.x and 1.2.x in 2004[16].

OpenSSL: *CVE-*{*2009-3245, 2006-3738, 2006-4339*} There are over 100 known vulnerabilities in various versions of OpenSSL. We scanned for the above three critical vulnerabilities in our firmware update dataset because they involve features that are likely to be reachable via network attack. The first two vulnerabilities can lead to arbitrary code execution. The last vulnerability can bypass x.509 certificate verification.

7.2.8.2 Findings

Figure 7.18 shows the percentage of vendor released firmware images that use versions of zlib and OpenSSL library containing known vulnerabilities for a subset of LaserJet models. Table 7.7 shows the duration of which known vulnerabilities that have existed for in various models of LaserJet printers.

Overall, we made the following observations:

Mandatory firmware update signature verification is not an adequate defense mechanism

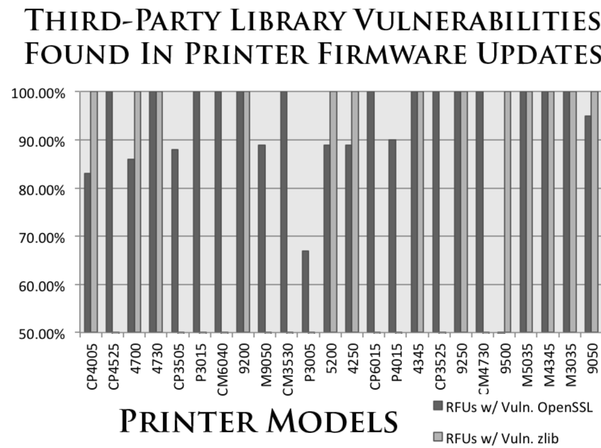

Figure 7.18: Percentages of RFUs for each printer model containing known zlib and OpenSSL vulnerabilities.

Model	Lib	Earliest RFU	Latest RFU
2055	ssl	Unknown	Unknown
	zlib	2009-04-30	Present
4005	ssl	2010-02-11	Present
	zlib	2009-06-05	Present
4250	ssl	2004-09-02	Present
	zlib	2004-09-02	Present
4700	ssl	2009-09-14	Present
	zlib	2009-06-05	Present
9050	ssl	2004-06-30	Present
	zlib	2004-06-30	Present

Table 7.7: Lifespan of vulnerabilities in third-party libraries used by LaserJet firmware.

against vulnerabilities that exist in the codebase of existing printers. Therefore, a large population of network printers is still potentially vulnerable to exploitation, despite the firmware updates released by the vendor.

Printer models analyzed	63
RFU images analyzed	373
All RFUs w/ at least 1 vulnerability	300
Latest RFUs w/ at least 1 vulnerability	41 (65.1%)
Most common zlib version	1.1.4
Most common OpenSSL version	0.9.7b

Table 7.8: Third-party library vulnerability analysis observations.

7.2.9 Recommended Defenses

We present two host-based defense techniques developed by the authors to mitigate the vulnerabilities described in Chapters 8 and 9. The vulnerable firmware update feature found in HP LaserJet printers is rarely used and should be disabled until it is needed. However, we found that disabling this feature was not trivial and at times impossible, as was the case with the LaserJet P2055DN. We propose a technique, which we call Autotomic[10] Binary Structure Randomization (ABSR): it not only disables unnecessary features but also removes the unused binary from the firmware image. This technique simultaneously reduces the attack surface of the embedded device as well as the amount of code and data that can be used as part of any shellcode.

Disabling unused features on the embedded device is helpful, but it does not guard against exploitation via attack vectors within necessary features that cannot be removed. For example, vulnerable third-party libraries like ones identified in Section 7.2.8 may be pivotal to the functionality of the embedded device. We believe techniques like ABSR should be used in conjunction with other host-based defenses to detect and mitigate the consequences of successful exploitation. Software Symbiotes have been demonstrated as a viable dynamic firmware integrity attestation technique on embedded systems such as enterprise routers.

Despite proper software and security engineering practices by vendors, firmwares will continue to be released with bugs and vulnerabilities. ABSR and Symbiotes are aimed at

[10] Autotomy - The spontaneous casting off of parts is a (biologically) viable security mechanism.

securing devices that run such firmware.

7.3 Concluding Remarks

In this section, we have qualified and quantified the nature and scope of the exploitability of embedded devices. We have established a quantitative lower-bound on the number of trivially vulnerable embedded devices in the world, which likely numbers in millions. We have devised several exploitation techniques aimed at overcoming challenges of carrying out reliable shell-code execution across heterogeneous populations of embedded devices. We have surveyed the exploitability of a collection ubiquitous embedded devices, which resulted in the public disclosure of several high impact vulnerabilities effecting hundreds of million devices. In short, we have demonstrated, through quantitative and qualitative analysis, two important findings about the current security posture of embedded devices. First, vulnerable embedded devices exist in large numbers in the world. Second, vulnerable embedded devices can be reliably exploited in similar ways as general-purpose computers.

We conclude this chapter of offensive study of embedded devices by drawing the following conclusions:

1. Exploitation of embedded devices is possible.
2. Reliably large-scale exploitation of embedded devices is feasible.

7.3.1 Poly-species propagation of advanced persistent embedded implants

Consider the embedded exploitation scenario illustrated in Fig. 7.19. By leveraging the vulnerabilities presented in this chapter, an attacker can compromise several classes of ubiquitous embedded devices, such as phones, printers and routers, in an automated fashion. Such an automated attack has been demonstrated in real-time[11].

During exploitation stage 1, the attacker can leverage CVE-2011-4161 to inject persistent malware into common network printers within the secured network. This can be done by sending a specially crafted document to an user within the secured network. The attack is triggered when the document is printed.

[11] aesop.redballoonsecurity.com

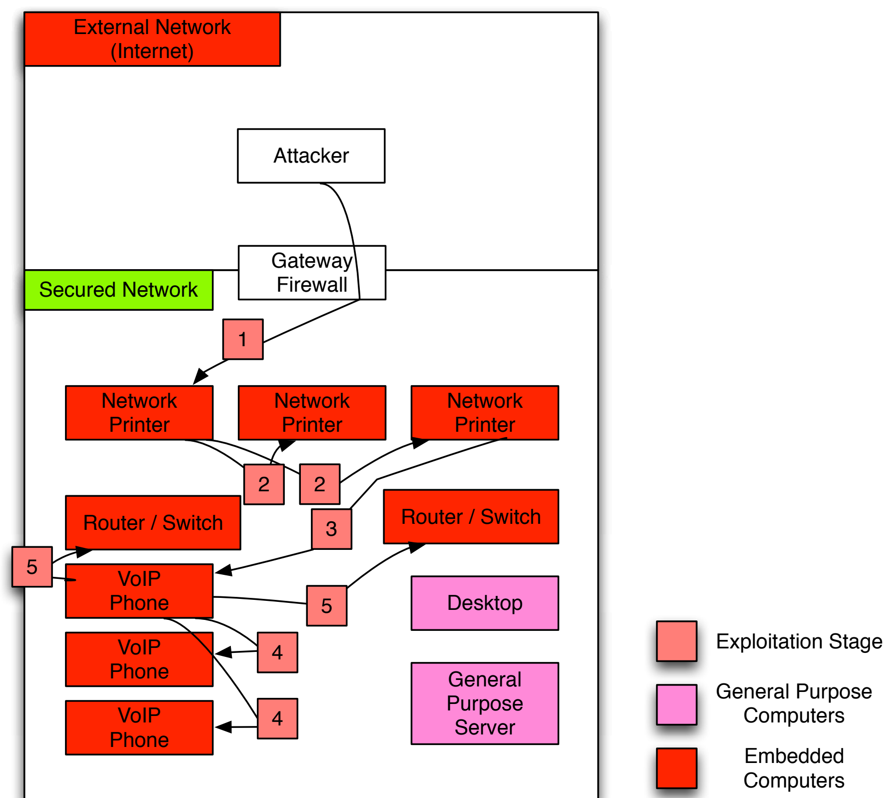

Figure 7.19: Anatomy of a plausible poly-species malcode propagation scenario

Once compromised, the malicious payload running within the network printer creates a reverse-ip tunnel through the gateway firewall back to the attacker. This effectively grants the attacker, located outside the secured network, layer-2 access and persistent presence within the secured network.

Using the compromised printer as an advanced persistent asset, the attacker carries out layer-2 and layer-3 reconnaissance within the secured network in order to identify other vulnerable embedded devices. During exploitation stage 2, the compromised printer identifies and compromises other vulnerable printers within the victim network.

During exploitation stage 3, the compromised printers leverage CVE-2012-5445, CVE-2013-6685 and ASA-2014-099 to exploit vulnerable VoIP endpoints identified through typical network reconnaissance.

During exploitation stage 4, the compromised VoIP endpoints are leveraged to identify and exploit other similarly vulnerable devices.

Lastly, during exploitation stage 5, the routers and switches themselves are exploited.

The pattern of reconnaissance, exploitation and malcode propagation is similar to that which have been observed within populations of general-purpose computers. However, the ability for a functionally cohesive piece of malcode to propagate across and operate stealthily within a heterogeneous collection of embedded devices running many different operating systems and ISA's can give the attacker novel channel of persistent access to the secured network via various compromised embedded devices.

Since individual embedded devices lack host-based defense capabilities, the secured network lacks the ability to detect the presence of embedded malcode as a whole. In the absence of host-based detection capabilities, the presence of malcode propagation can only be detected using methods that rely on phenomenons external to the compromised devices, such as network-based intrusion detection. However, such techniques will likely have limited efficacy in the context of embedded exploitation scenarios such as the one presented in this section.

Consider the fact that several stages of the above malcode propagation scenario can be carried out via direct network connections between devices. In such cases, traditional network-based intrusion detection techniques will be rendered nearly useless, especially if the

routers and switches, which constitute the networking substrate, is themselves compromised.

7.3.2 Large-scale exploitation

We conclude this chapter of offensive study of embedded devices by asserting that host-based defense within each embedded device endpoint is crucial to the defense of modern computer networks. The next chapter will discuss several such defensive mechanisms, and the core contributions of this thesis.

Part III

Embedded Defense

Chapter 8

Symbiotic Embedded Machines

Large numbers of legacy embedded devices on the Internet, such as routers, are ripe for exploitation. However, little to no host-based defensive technology like antivirus and IDS's are available to protect these devices, thus leaving vast portions of the Internet substrate vulnerable to attack. We propose a host-based defense mechanism, which we call Symbiotic Embedded Machines (SEM), that is specifically designed to inject intrusion detection functionality into the firmware of existing embedded devices. A SEM or simply a *Symbiote*, may be injected into deployed legacy embedded systems with no disruption to the operation of the device. A Symbiote is a code structure embedded *in situ* into the firmware of an embedded system. The Symbiote tightly co-exists with its host executable in a mutually defensive arrangement, sharing computational resources with its host while simultaneously protecting the host against exploitation and unauthorized modification. The Symbiote is stealthily embedded in a randomized fashion within an arbitrary body of firmware to protect itself from removal and unauthorized deactivation. We demonstrate the operation of a generic whitelist-based rootkit detector Symbiote injected *in situ* into Cisco IOS with little performance penalty and without impacting the routers functionality. The proposed Symbiote can detect unauthorized modification of IOS memory in approximately 300 ms on a physical Cisco 7121 router. We present the performance overhead of a Symbiote on physical Cisco hardware. A MIPS implementation of the proposed Symbiote is also ported to ARM and injected into a Linux 2.4 kernel, allowing the Symbiote to operate within Android and other mobile devices. The use of Symbiotes represents a practical and effective

protection mechanism for a wide range of devices, especially widely deployed, unprotected, legacy embedded devices.

8.1 Introduction

We propose to use the Symbiote to inject intrusion detection functionality into the firmware of legacy embedded systems. The Symbiote structure should allow an integrity verification payload to detect unauthorized modification of the static regions of memory of the protected device. Symbiote injection may be randomized so that each instance is distinct from all other injected systems in order to thwart attempts by an adversary to disable the injected Symbiote. In general, we aim to create an injectable software construct that provides the following four fundamental security properties:

1. The Symbiote has full visibility into the code and execution state of its host program, and can either passively monitor or actively react to the observed events at runtime.
2. The Symbiote executes along side the firmware or host program. In order for the host to function as before, its injected SEM must execute, and vice versa.
3. The Symbiote's code cannot be modied or disabled by unauthorized parties. Such attempts will either be detected by the Symbiote or will render the host program inoperable.
4. No two instantiations of the same Symbiote is the same. Each time a Symbiote is created, its code is randomized and mutated by a polymorphic engine, rendering signature based detection methods and attacks that require predictable memory and code structures within the Symbiote ineffective.

An immediate application of the system presented in this chapter is the fortification of existing vulnerable network routing devices. Network embedded devices like routers and firewalls are vulnerable to the same attacks as general purpose computers, but generally do not have the facility to execute third-party host-based defenses like anti-virus. Using the Symbiote, we have successfully injected a host-based root-kit detection mechanism into a closed-source proprietary operating system, Cisco IOS. We believe that the techniques discussed in this chapter can be used to fortify existing vulnerable devices within the critical

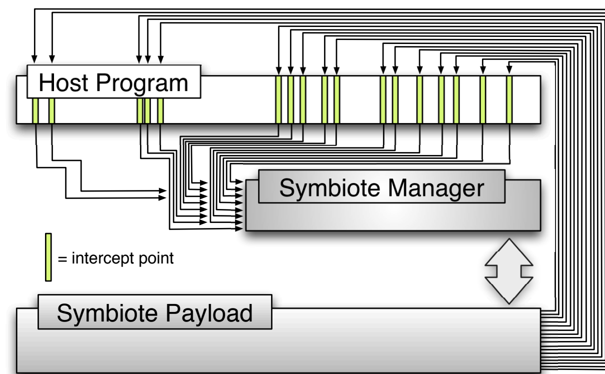

Figure 8.1: Logical overview of SEM injected into embedded device firmware. SEM maintains control of CPU by using large-scale randomized control-flow interception. The SEM payload executes alongside original OS. Figure 6 shows a concrete example of how the SEM payload can be injected into a gap within IOS code.

infrastructure, like smart power meters, machine to machine control systems, as well as everyday embedded devices like VoIP phones, home routers and mobile computers.

Figure 8.1 shows how a Symbiote is typically injected into a host program. A large number of control-flow intercepts are distributed randomly throughout the body of the host program, allowing the Symbiote Manager to periodically regain control of the CPU. Once the Symbiote Manager is invoked, it executes a small portion of its defensive payload before saving its execution context and returning control back to the host program. This allows the Symbiote and host program to execute in tandem and in a time-multiplexed manner without altering the original functionality of the host program. The Symbiote injection process provides a probabilistic lower bound on the frequency in which the Symbiote will be invoked at runtime as an adjustable parameter. The Symbiote resides within the same execution environment as the host program. It has the ability to passively monitor or proactively alter the host program's behavior at runtime. Since the Symbiote is deeply intwined with its protected host program, attempts to corrupt or alter the Symbiote binary will either be detected by the Symbiote or cause the host program to crash as described in Section 8.4.

As we see in Section 8.3, Symbiotes can defend any arbitrary executable, even other Symbiotes. Unlike traditional anti-virus and host-based defense mechanisms, which install into and depend heavily on facilities provided by the vulnerable systems they are meant to protect, the Symbiote treats its host program as an external and untrusted entity. Symbiotes do not depend on functionality provided by its host, giving it several advantages.

The Symbiote:

Is agnostic to its operating environment. Since the Symbiote injects itself **into** its host program, it does not need to conform to any executable format. The Symbiote will execute as long as its host program is a valid executable, regardless of operating system type or version.

Can be injected into proprietary black box operating systems. Since Symbiotes are agnostic to the inner workings of its host program and execution environment, deploying Symbiotes on proprietary systems is as easy as deploying them within well known ones.

Is entirely self-contained and does not depend on facilities provided by its host program. The Symbiote treats its host program as an untrusted and foreign entity. It does not use any external code to protect the host program. Therefore, vulnerabilities within the host program will not be shared by its Symbiote.

Is self-protecting and stealthy and thus is difficult to detect and deactivate by an adversary.

Is lightweight. The Symbiote does not require hardware virtualization support. It executes natively on its host program's hardware and does not perform JIT compilation. This efficiency makes the use of Symbiotes in resource-constrained embedded devices feasible.

8.2 Threat Model

We assume the attacker is technically sophisticated and has access to both zero-day vulnerabilities and compatible exploits allowing reliable execution of arbitrary code. We further

assume that the attacker executes the attacks in an online fashion. In other words, the attacker must carry out the attack remotely against a running device without interfering with its function or causing it to crash or reboot. Attacks involving configuration changes or replacement of the entire firmware image (that requires a reboot) are excluded from our model because they can be detected by conventional methods like network-based monitoring and filtering. We also assume that the attacker has access to the original host program image, before any Symbiotes are injected into it.

Online attacks against the Symbiote-protected host program can be divided into two categories: those attacks that attempt to disable or evade the Symbiotes protecting the host program, and those that do not. We first address existing attacks that target the host program and show how Symbiotes can prevent such attacks. Section 8.4 discusses multi-stage attacks that attempt to disable Symbiotes prior to executing their malicious payloads.

With respect to Cisco routers, we focus on rootkit techniques that make persistent changes to the IOS operating system. The SEM mechanism introduced in this chapter is used to detect injected code that changes portions of the device that are otherwise **static** during the life time of the device. The Symbiote payload presented in this chapter is designed to detect unauthorized code modification and cannot defend against exploits that do not make persistent change to the router's code. However, the SEM approach can also be used to detect exploitation in dynamic areas of the target embedded device like the stack and heap. Symbiote control-flow interception methods and payloads that defend against return-to-libc, return oriented and heap related attacks are currently under research.

8.3 Symbiotic Embedded Machines

The Symbiote is a self-contained entity and is not installed onto the host program in the traditional sense. It is injected into its host program's code in a randomized fashion. Anti-virus and other host-based defenses are typically installed onto or into the operating system it protects. This places a heavy dependence on the features and integrity of the operating system. In general, this arrangement requires a strong trust relationship with the very

software (often of unknown integrity) it tries to protect. In contrast, the Symbiote treats its entire host program as an external and untrusted entity, and therefore eliminates the unsound trust on traditional legacy systems.

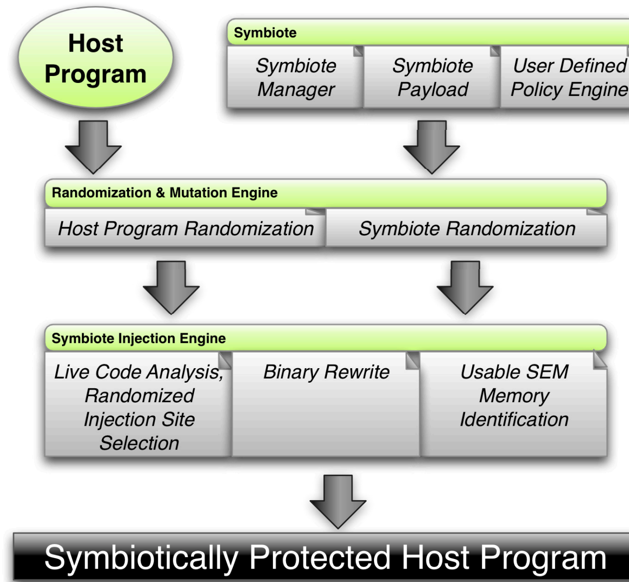

Figure 8.2: Generic end-to-end process of fortifying an arbitrary host program with a Symbiote. Our proof-of-concept Symbiote for Cisco routers, Doppelgänger, is completely implemented in software and can execute on existing commodity systems without any need for specialized hardware.

Figure 8.2 shows the process in which a Symbiote is instantiated and injected into a host program, producing a symbiotically protected host program. First, a Symbiote is instantiated by combining a Symbiote Manager with a Symbiote Payload. The resulting Symbiote binary, along with the host program binary, can be optionally passed to the Randomization and Mutation Engine. The resulting pair of binaries are passed to the Symbiote Injection Engine. Here, the host program is analyzed. Using a combination of automated static and dynamic analysis, a large number of control-flow interception points within the host program is chosen in a randomized fashion. The Symbiote is finally injected into the host program using standard binary rewriting techniques, yielding a new symbiotically protected executable binary that is functionally equivalent to the original host program.

If the Randomization and Mutation Engine is used, each instantiation of a Symbiote is polymorphically mutated and randomized during the injection process. In this case, studying and reverse engineering one instance of a particular Symbiote provides the attacker with little to no useful information about the specifics of any other instantiation of the same Symbiote.

The Symbiotic Embedded Machine structure creates an **independent** execution context from the native operating system at runtime. SEM uses the newly created context to execute its Symbiote payload. Payloads are interchangeable and can be written in any high level language. As the host program executes, its SEM periodically diverts CPU to its Symbiote payload in brief bursts before returning control back to the host program. It is important to note that SEM does not use traditional virtualization techniques. As most network embedded devices do not have hardware hypervisors or virtualization support, the methods we use to achieve execution context separation use only standard CPU instructions.

8.3.1 Doppelgänger: A Symbiote Protecting Cisco IOS

Figure 8.3 shows the rendering of a typical Symbiote when injected into Cisco IOS. Note that the Symbiote payload portion is shown as a contiguous block for clarity. In practice, the Symbiote payload should be randomly distributed across many noncontiguous segments.

For generality, SEM does not rely on firmware specific code features like system calls or standard libraries. The Control-Flow Interceptor component uses inline hooks to intercept a large number of functions within the target firmware. Upon invocation of an intercepted function, control of the CPU is redirected to the Symbiotic Embedded Machine Manager (SEMM), which executes a small portion of the SEM payload. Once invoked, the SEMM manages the execution of injected SEM payload as follows:

1. Store the execution context of the native OS (i.e. IOS).
2. Load the context of the SEM payload.
3. Compute how long the SEM payload can run, based on current native OS system utilization.
4. Execute the SEM payload for a fixed amount of time.

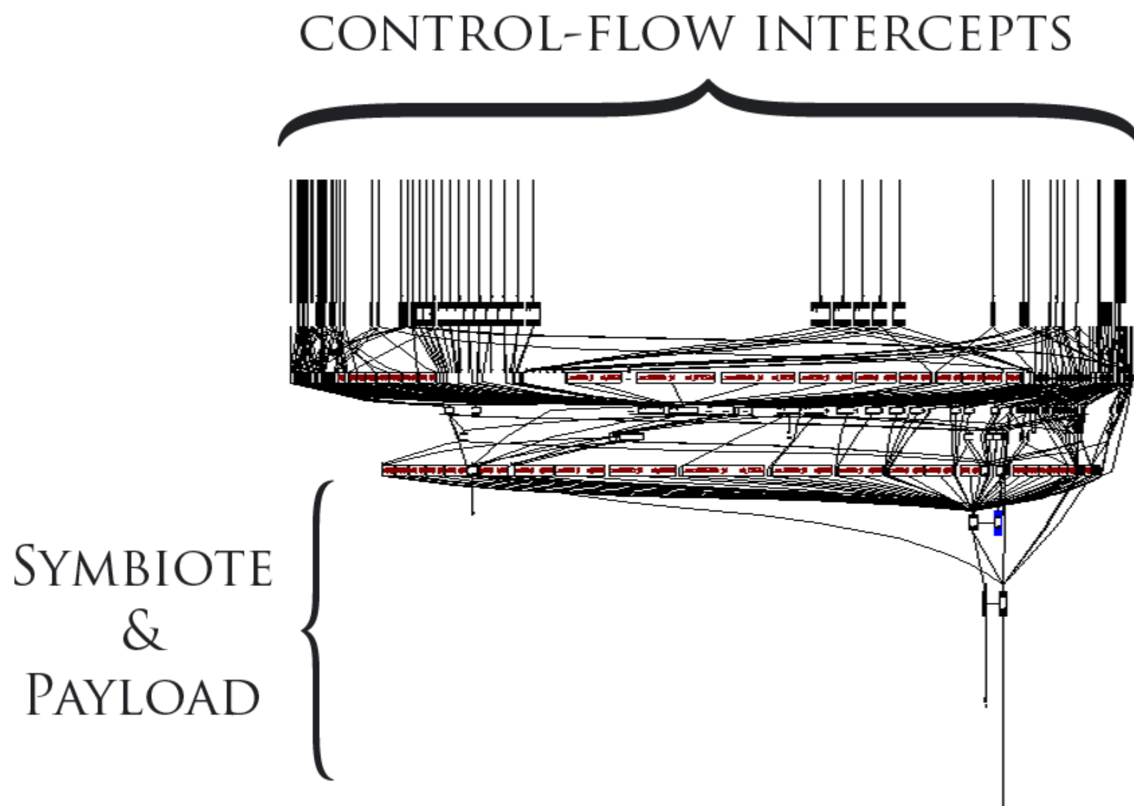

Figure 8.3: Rendering of Symbiote structure inside a typical IOS firmware. The top of the graph shows large numbers of control-flow interceptors diverting the CPU to the SEM manager and payload, which can be seen as the small vertical structure at the bottom of the graph.

5. Store the execution context of the suspended SEM payload.
6. Load the execution context of the native OS at the time the SEMM assumed control.
7. Restore CPU control to the invoked function.

8.3.2 Live Code Interception with Inline Hooks

First, analysis is performed on the original host program in order to determine areas of live code, or code that will be run with high probability at runtime. Next, random intercept points are chosen out of the live code regions found. Lastly, each Symbiote Manager, Symbiote payload, and a large number of control-flow intercepts are injected into the host program binary, yielding a Symbiote protected host program.

Control-flow intercepts are distributed in a randomized fashion throughout the host program's binaries in order to ensure that the Symbiote regains control of the CPU periodically. We would like to ensure that these randomly chosen intercept points are located within regions of code that will be frequently executed at runtime. This problem is difficult to solve with high accuracy in the general case. However, our purposes do not require the classification mechanism to be absolutely accurate. In reality, implementing a sufficient solution for real-world host programs is not too difficult. Section 8.3.5 discusses the methods used in our experiments for live code classification.

Once regions of code within the host program are chosen for control-flow interception, the Symbiote injection process imbeds interceptors as well as the Symbiote binary into the host program. The Symbiote implementation presented in this chapter uses a Detour [115] style inline function hooking mechanism for control-flow interception. While we injected our intercepts within the function preamble in the current Symbiote implementation, this is not a requirement. Control-flow intercepts can be embedded in arbitrary positions within the host program using existing binary instrumentation techniques.

Detour [115] style inline hooking is a well known technique for function interception. However, SEM uses function interception in a different way. Instead of targeting specific functions for interception that requires precise *a priori* knowledge of the code layout of the target device, SEM randomly intercepts a large number of functions as a means to periodically and consistently re-divert a small portion of the device's CPU cycles to execute

the SEM payload. This approach allows SEM to remain agnostic to operating system specifics while executing its payload **alongside** the original host program. The SEM payload has full access to the internals of the original firmware but is not constrained by it. This allows the SEM payload to carry out powerful functionality that are not possible under the original OS. For example, the IOS rootkit detection payload presented in Section 8.3.7 bypasses the process watchdog timer constraint, which terminates any IOS process running for more than several seconds, because the detector operates outside the control of the OS.

Stealth is a powerful byproduct of the SEM structure. In the case of IOS, no diagnostic tool available within the OS (short of a full memory dump) can detect the presence of the SEM payload because it manipulates no OS specific structure and is effectively invisible to the OS. The impact of the SEM payload is further hidden by the fact that CPU utilization of the payload is not reported within any single process under IOS and is distributed randomly across a large number of unrelated processes.

Note that the mechanism of action used by our current Symbiote is designed specifically for resource-constrained real-time embedded devices. The Symbiote approach can be migrated to general purpose computers in standard operating systems running on x86. However, because of the plethora of dynamic instrumentation mechanisms available in these environments, the x86 version of the Symbiote can use a wide variety of control-flow interception mechanisms not available and not suitable for embedded devices. For example, instead of using Detour style hooks, the Symbiote can leverage existing tools like Pin. While each component of the Symbiote may be implemented differently to take advantage of the underlying hardware architecture, the structure and interfaces between each component will remain the same.

8.3.3 SEMM and Execution Context Records

The SEM Manager (SEMM) is responsible for managing all necessary state information to ensure that the SEM payload and the native OS can safely execute in a time multiplexed manner. This information is saved and loaded each time the SEMM switches CPU control between the SEM payload and the native OS. The major challenge of the SEMM is preserving the integrity of the saved execution contexts. Any corruption to this data will very

likely cause unintended behavior or a system crash when the context is loaded into CPU. Since the SEM payload is under our control, it will not accidentally modify the saved context record of the native OS. Therefore, one of the SEMM's critical goals is to prevent the native OS from overwriting the memory chunks used to store the SEM payload's execution context record.

This seems like a difficult task, as the native OS has no knowledge of the existence of the SEM and has legitimate access to all addressable memory on the target device. However, the size of each execution context record is approximately 256 bytes. In practice, finding small, usable chunks of memory which can be safely used by the Symbiote is fairly simple. The next section discusses how usable memory is identified in Cisco IOS to support our proposed Symbiote. Note that many other methods of locating or creating usable Symbiote memory exists. We chose a simple and noninvasive allocation method for the purposes of the Symbiote payload presented in this chapter. However, as the size and memory requirements of payloads increase, we may explore other ways of allocating usable memory for the Symbiote. This is a topic of ongoing research.

8.3.4 SEM Memory Management

In order for SEM to execute safely along the native OS, it must have access to memory that the native OS will not modify. Using SEM's small memory footprint to our advantage, we locate small chunks of usable memory located in areas of the binary which will not be used by the native OS. We have identified several methods of finding such unused memory. The most successful strategy looks for "gaps" within various structures in the firmware. Small gaps between chunks of binary executables, strings, and other **static** structures are often intentionally introduced for optimization and alignment purposes. Given that the native OS uses these gaps only as space holders, SEM can safely use these small chunks of memory. It is important to note that usable memory identified through this method constitutes **unused** address space, rather then **unlikely to be used** address space.

Usable memory can be identified with a single linear sweep of the firmware image. While this approach places an upper bound on the amount of usable space available to the SEM, experimental results have shown that this is more than adequate. For example, we

identified 290Kb and 130Kb of usable space trivially inside the **.data** and **.text** sections of the 12.2(27c) IOS image respectively. This space is more than sufficient for our current SEM memory requirements (Section 8.5, Table 8.1). Furthermore, much larger amounts of usable space exist within dynamic sections of memory like the stack and heap. Safely locating usable memory chunks within these areas is currently under research.

Our proof of concept Symbiote implementation places the SEMM and SEM payload binaries within the large alignment gap at the end of the **.text** section of the firmware. This static placement advertises to an adversary where to look for a Symbiote, so in an actual deployment, the SEMM and payload binaries would be distributed across many non-contiguous segments in memory. In addition, we use a pool of gaps in the .data section for storing the stack and execution context records of the SEM payload

8.3.5 Automatically Locating Control-Flow Intercept Points

Control-flow intercept points are chosen randomly out of candidate *live* code regions within the host program. The way code regions are classified as *live*, and the number of intercepts chosen from each region directly affects the frequency in which the Symbiote will gain control of the CPU. This in tunr directly affects the performance and overhead of the Symbiote.

Both dynamic and static methods of live code classification are considered for our experiments. First, the host program is executed under a profiler in order to observe live code, or code coverage under normal operating conditions[1]. Using code coverage analysis to classify live code is advantageous because it cannot produce false positives, i.e. dead code cannot be classified as live code. However, this dynamic approach cannot classify regions of code that are reachable only through rare or malformed program input. Therefore, we augment our code coverage based live code classifier with static analysis of the control-flow graph of the host program. Figure 8.4 shows the live code regions of a typical IOS router firmware image after our initial analysis. Control-flow intercept points will be chosen randomly out of these code regions (shown in white) to periodically divert CPU control to the injected Symbiote. Note that intercept points can, and should also be placed in the binary outside of the detected live code regions.

[1] In the case of IOS, we profiled the router image using Dynamips under various workloads.

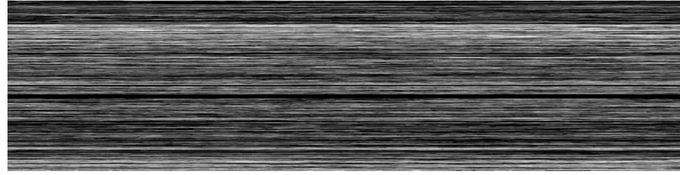

Figure 8.4: Live Code Regions (White) Within IOS 12.4 Firmware (Black). Code Range: 0x80008000-0x82a20000

8.3.6 Injecting Symbiotic Embedded Machines into Firmware

The general injection process for any embedded device may be represented as a transform function from the original binary of the devices software to a new binary safely modified with new functionality:

$$I_{SEM}(B_i, X_{arch}, SEM_{arch}, P_{SEM}) -> B_o$$

Given a binary firmware B_i, which executes on a CPU architecture X_{arch}, a SEM implementation SEM_{SEM} and a chosen payload P_{SEM} written in C, the SEM injection function I_{SEM} outputs a new binary firmware B_o such that it:

- Retains the same size and functionality as B_i.
- Contains both SEM_{arch} and a cross-compiled binary of P_{SEM}.
- Redirects a share of CPU time to the execution of P_{SEM}.

The injection function I_{SEM} does the following:

- Cross-compile P_{SEM} into executable binary for X_{arch}.
- Identify "live" code regions within the host program.
- Identify a sufficiently large set of control-flow intercept points.
- Identify a sufficiently large usable memory chunks within B_i.
- Creates a new image B_o.
- Inject SEM_{arch} and the cross-compiled payload P_{SEM} into suitable memory chunks in B_o.
- Install inline hooks to redirect CPU control to the embedded SEM_{arch}.
- Output B_o.

As the injection process uses only gap memory chunks *in situ*, the lengths of B_i and B_o are identical.

8.3.7 Rootkit Detection Payload

To detect IOS malcode and rootkits described in the previous section, we implement a white-list strategy. Known rootkits operate by hooking into and altering key functions within IOS. To do this, specific binary patches must be made to executable code. Therefore, a continuous integrity check on all **static areas** of Cisco IOS will detect all function hooking and patching attempts made by rootkits and malware. The rootkit detection payload described below is not specific to IOS, and can be used on other embedded operating systems as well. For the white-list strategy to be effective, the protected kernel code must either remain static during legitimate operation, or only be allowed to change in **predictable** ways. For example, while some embedded operating systems support legitimate mechanisms to dynamically update the kernel, the contents of those updates are static and known *a priori*. Therefore, the checksums of approved updates can be calculated and distributed to SEM ahead of time.

Formally, let

$$H_c = F_{hash}(S_c)$$

where $\{S_c\}$ is a set of contiguous code segments we wish to monitor within B, a binary host program. If H_c outputs a cryptographically secure hash function over all monitored code segments, a change in H_c, then, indicates a change within at least one code segment in $\{S_c\}$.

$$H_c = \{x | x \in S_c, F_{hash}(x)\}$$

Further, we can compute and monitor multiple hash values $\{H_{c_i}\}$ over any arbitrary subset of $\{S_c\}$. By doing so, we gain arbitrary resolution on the location of code modification at cost of increased memory and computational overhead.

8.4 Computational Lower Bound of Successful Software-Only Symbiote Bypass

This section discusses multi-stage attack strategies, which attempt to disable the Symbiote prior to executing their malicious payload. We provide an intuitive lower bound of the computational cost of a successful attack against software-only Symbiotes. We also discuss ways of detecting and defending against such multi-stage attacks.

Generally, software-only Symbiotes can be successfully bypassed in two ways:

Attack 1: Remove control-flow intercepts. If the attacker can remove all control-flow intercepts within all live code regions before the Symbiote's detection latency, the attacker can prevent the Symbiote from ever regaining control of the CPU.

Attack 2: Deactivate the SEMM or Payload. If the attacker can locate and patch the Symbiote's manager or payload code, the Symbiote can be completely disabled.

The Symbiote control-flow intercepts are randomly distributed within the live code region, while the Symbiote Manager and Payloads are distributed randomly throughout the entire host program[2].

Both attacks reduce to a general problem of identifying all P out of N bytes, P being the bytes belonging to the Symbiote component under attack, N being the bytes of the host program in which P can exist. In the case of attack 1, the attacker must identify and remove all control-flow intercepts, P injected into all live code regions, N (assuming that this is known). Since the Symbiote binary is polymorphically mutated at injection time, the attacker cannot search for a well-known Symbiote signature through the binary. Instead, the attacker must compare an unmodified copy of the host program with the victim host program during an online attack. This is essentially equivalent to at least a linear operation over the size of all live code regions.

Similarly, since the Symbiote binary is distributed randomly throughout the host program, an attacker must identify all code regions belonging to the Symbiote. There are many

[2]While the Symbiote is distributed randomly throughout the binary of the host program, the injection process ensures that the Symbiote code cannot be inadvertently executed by the host program. In other words, the control-flow intercepts are the only mechanism in which the Symbiote code will be invoked.

ways to do this. However, since no well-known signature exists for the Symbiote code, the attacker must perform dynamic disassembly in order to follow control-flow intercepts to a piece of Symbiote code. Alternatively, the attacker can perform a linear comparison of the entire host program to identify all injected Symbiote code. In the former case, the attacker's problem is reduced to attack 1, because unless all control-flow intercepts are removed, the attacker cannot be sure that all Symbiotes are removed. In the latter case, the attacker must use a linear amount of CPU and network I/O, which again reduces to the problem of identifying P bytes out of N.

To put these attacks into perspective, the average size of the host programs analyzed in our experiments is approximately 35 MB, the size of live code regions considered for control-flow interception is approximately 10 MB. Each host program contains approximately 75,000 functions, all of which can be intercepted. (Note that control-flow interception need not take place only at the function preamble, but can exist anywhere within the function body.) If the attacker attempts to perform a linear comparison, portions of the unmodified host program will have to be transferred over the network during the online attack. The attacker can also attempt to dynamically disassemble the 10 MB of live code. Both attack strategies require a very large amount of network I/O or CPU which raises the bar quite high for the attacker to overcome without being noticed. If an attacker is able to successfully carry out either attacks in an online fashion faster then the Symbiote's detection latency, the self-monitoring Symbiote arrangement will still be able to detect the attack.

To further raise the necessary complexity of a successful attack, multiple Symbiotes can be arranged in a self-monitoring monitor arrangement. As proposed by Stolfo, Greenbaum and Sethumadhavan [105], a network of monitors can be constructed in a way such an alarm will be raised if any subset of monitors are compromised or deactivated, or if any critical condition monitored by the system is violated. Multiple mutually protected Symbiotes can be injected into the same host program. Networks of Symbiote-protected devices, like routers, can also be arranged in this self-protecting manner.

8.5 Symbiote Performance and Computational Overhead

Once a Symbiote is injected into its host program, it is important to bound the frequency and duration which the Symbiote will control the CPU as well as the aggregate computational overhead the Symbiote system imposes on the original host program.

We randomly choose a set of control-flow intercept points within *live* regions of the target host program. The method and parameters used to determine *live* regions, as well as the number of intercept points chosen gives us fine grain control of $p(\alpha_i, \delta, \tau_q)$, and gives us a probabilistic bound on the frequency in which the Symbiote will gain control of the CPU. Section 8.3.5 discusses the methods we used to extract "live" regions from the host program.

Consider the computational cost of an injected SEM during some time period τ_q.

Let $\{\alpha_1 ... \alpha_n\}$ be the set of all functions in binary firmware β.

Let $g(\alpha_i, \tau_q)$ be the cost of SEM per invocation at time period τ_q.

Let $h(\alpha_i)$ be the binary function representing whether function α_i is "intercepted" by the SEM.

Let $p(\alpha_i, \delta, \tau_q)$ be the number of times function α_i will be invoked during time period τ_q, given some probability distribution δ.

Note that the probability distribution δ is derived from the live code analysis performed during the Symbiote injection process. Supposing a control-flow intercept is inserted into a piece of live code which is known to execute with some probability according to the normal execution model of the host program, we can claim that the Symbiote control-flow intercept will also be invoked with at least this probability. Thus, the "live" code analysis gives us a probabilistic lower bound on the frequency in which the Symbiote will regain control of the CPU over any time period τ_q.

Let the SEM cost function $g(\alpha_i, \tau_q)$ be:

$$g(\alpha_i, \tau_q) = O_{SEMM} + O_{payload}(\alpha_i, \tau_q) \tag{8.1}$$

Where O_{SEMM} is the (**constant**) cost of invoking the SEMM and $O_{payload}(\alpha_i, \tau_q)$ is the

amount of the SEM payload to execute (**variable**), given function α_i and time period τ_q.

The **Lower bound on SEM cost** C_q**, over time period** τ_q can be expressed as:

$$C_q = \Sigma_i O_{SEMM} * p(\alpha_i, \delta, \tau_q) \quad (8.2)$$

$$= O_{SEMM} \Sigma_i p(\alpha_i, \delta, \tau_q) \quad (8.3)$$

Intuitively, the lower bound on the SEM cost is simply the overhead of invoking the SEMM multiplied by the expected number of times that the SEMM will be invoked over time period τ_q.

The computational cost of SEM C_q, over time period τ_q is:

$$C_q = \Sigma_i g(\alpha_i, \tau_q) * h(\alpha_i) * p(\alpha_i, \delta, \tau_q) \quad (8.4)$$

The **Upper bound on SEM cost** C_q **over time period** τ_q**.** is a function of the number and distribution of functions intercepted in order to execute the SEMM and the cost of the payload execution the SEMM manages. Let $h(\alpha_i) = 1$ for all functions α), then

$$C_q = \Sigma_i g(\alpha_i, \tau_q) * p(\alpha_i, \delta, \tau_q) \quad (8.5)$$

$$= \Sigma_i (O_{SEMm} + O_{payload}(\alpha_i, \tau_q)) * p(\alpha_i, \delta, \tau_q) \quad (8.6)$$

$$= O_{SEMm} \Sigma_i p(\alpha_i, \delta, \tau_q) + \Sigma_i O_{payload}(\alpha_i, \tau_q) * p(\alpha_i, \delta, \tau_q) \quad (8.7)$$

Observations

- The distribution δ, and therefore, $p(\alpha_i, \delta, \tau_q)$ cannot be changed (without changing the host's original functionality), and varies with respect to different devices, firmware, and input.
- The function $h(\alpha_i)$ can be used to control SEM CPU utilization but is binary and **imprecise**.
- The function $g(\alpha_i, \tau_q)$ can be used to control SEM CPU utilization[3] **precisely**.

[3]In practice, O_{SEMm} is much smaller than $O_{payload}()$; therefore, the second summation in equation 8.7 dominates over the first (Section 8.5.1).

Doppelgänger SEMM Size	Payload Size
1048 Bytes	336 Bytes
Payload Memory Footprint	IOS 12.2 Image
2240 Bytes and 1,936 Inline Hooks	c7100-is-mz.122-27c.bin
IOS 12.2(27c) Size	Router Model
25,922,308 Bytes	Cisco 7120-AT3
Router CPU	Router Memory
MIPS R527x @ 225Mhz	57344K

Table 8.1: Doppelgänger Implementation Stats

We can vary the **number** of control-flow interceptions ($h(\alpha_i)$) and the **amount** of SEM payload that is executed at each invocation ($g(\alpha_i, \tau_q)$) to control precisely the amount of CPU time used by the SEM. We can implement these two mechanisms in the **SEMM** to divert more CPU cycles to the SEM during periods of low CPU utilization and divert less during periods of high CPU utilization. Figure 6 shows actual CPU utilization when Doppelgänger and our rootkit detection payload are installed on a physical Cisco 7120 router with $g(\alpha_i, \tau_q)$ set to several fixed values. This parameter directly affects the portion of the CPU that is diverted to executing the SEM payload. Figure 7 and Table 8.2 shows an inverse relationship between $g(\alpha_i, \tau_q)$ and the amount of time required to detect a modification of IOS, which we call the **detection latency**.

8.5.1 Experimental Results: Doppelgänger, IOS 12.2 and 12.3, Cisco 7121

Doppelgänger, our proof of concept SEM implementation is injected into IOS 12.2(27c) and IOS 12.3(3i) on the a Cisco 7120 router. The rootkit detection payload is implemented in C and calculates a single hash covering the .text memory range **0x60008000** to **0x61662000**. As a proof of concept, we implemented CRC-32 as the hashing function used by the rootkit detection payload.

Two sets of experiments are done to demonstrate both performance characteristics and accurate IOS code modification detection. To measure CPU utilization, the Cisco 7120

SEM Payload Burst Rate			
0xF	0x1F	0xFF	0x7FF
56s	43s	35s	0.3s

Table 8.2: Detection Latency at Different SEM Payload Burst Rates IOS 12.2

router is put through a standard workload script with varying SEM payload execution burst rates. The workload script touches a cross section of standard router attack surface by performing tasks like enabling / disabling routing, generating system status dumps, reconfiguring routing parameters, and advertised routes, etc. The CPU utilization is measured by SNMP polling.

To demonstrate IOS code modification detection, we simulate the installation of a rootkit by modifying a SEM protected IOS firmware with added function hooks and code. We then boot the Cisco router with the altered image and measure the time required for the SEM payload to detect the modification. We configure the payload detector to **halt** the router once the modification is detected. This is also done with varying SEM payload execution burst rates to demonstrate the relationship between SEM payload execution rate and runtime detection latency. Performance evaluation data are included in the Appendix.

8.5.1.1 Experimental Results

Figure 6 demonstrates CPU utilization of the 7120 router when the SEM payload execution burst rate, aka $g(\alpha_i, \tau_q)$, is varied. Table 8.2 is the average detection latency. With a fixed execution burst rate of 0x7FF, the Symbiote payload was able to detect unauthorized code modification in approximately 0.3 seconds.

8.5.1.2 Experimental Findings

- The Cisco router continues to function with Doppelgänger running concurrently, even during periods of near maximum CPU utilization.
- SEM CPU utilization can be controlled by varying the payload execution burst rate within the SEMM.

- Detection Latency is inversely proportional to SEM CPU utilization (and SEM payload execution burst rate).

As Figure D.1 shows, Doppelganger's CPU utilization currently increases proportionately to the CPU utilization of its host program. This behavior may be desirable as it directs more CPU resources to the host's defense mechanisms when the host is heavily utilized. However, this may not always be the best approach. For example, we may want to execute the Symbiote's payload heavily when the host program is idle and throttle down the Symbiote's payload when the host program is busy. This can be accomplished by implementing an alternative SEMM, specifically, one which regulates the Symbiote's CPU utilization by reacting to current host system utilization.

8.5.2 Doppelgänger, Linux 2.4.18, ARM and Qemu

We have completed a preliminary port of Doppelgänger onto the ARM architecture. We chose ARM as the second implementation candidate in order to demonstrate the feasibility of injecting SEM payloads into ARM-based mobile devices running Android and other Linux-based embedded operating systems. We have successfully injected this port of Doppelgänger into a vanilla Linux 2.4 kernel running on Qemu, a popular ARM processor emulator.

While the Symbiote manager required a re-implemention using ARM instructions, the control-flow interception and SEM injection process remained the same for both Cisco IOS and Linux, as did the Symbiote payload, which was simply cross-compiled.

8.6 Concluding Remarks

We presented a Symbiotic Embedded Machine (SEM), a novel software mechanism that provides a means of embedding defensive software into existing embedded devices. Using a specific SEM implementation, which we call Doppelgänger, we automatically inject a rootkit detection payload into a Cisco 7120 router running multiple firmware images across two major IOS versions, 12.2 and 12.3. By injecting under 1400 bytes of code into the IOS firmware, Doppelgänger protects the router from all function hooking and interception at-

tempts. Our white-list based rootkit detection payload does not require *a priori* knowledge of IOS internals, or signatures of known rootkits, and can protect the router against any code modification attempts. Empirical results show that unauthorized code modification attempts are detected in approximately 300 ms. As the SEM structure operates alongside the native OS of the embedded device and not within it, it can inject generic defensive payloads into the target device regardless of itself original hardware or software. Due to the unique nature of network embedded devices, we posit that retrofitting these widely deployed vulnerable devices with defensive SEM's is the best hope of mitigating a significant emerging threat on our global communication infrastructure. SEM is a generic defensive mechanism suitable for general purpose host protection. Our ongoing research aims to demonstrate the advantages of the Defensive Mutualistic paradigm and Symbiotes over traditional AV solutions. Lastly, using the techniques presented in this chapter, we believe it is feasible to transform existing routers into exploitation detection sensors in order to monitor and analyze attacks against the Internet substrate.

Chapter 9

Autotomic Binary Structure Randomization

9.1 Motivation

Attack surface reduction and software diversification have been applied in many ways to improve the security posture of software. We outline the benefits of each defensive technique and propose a novel method of combining the two types of defenses into a hybrid defense that is suitable for the defense of legacy embedded systems.

We present Autotomic Binary Structure Randomization (ABSR), a hybrid defense that applies two techniques in a mutualistic manner to enhance the security posture of legacy embedded systems. An automated attack surface reduction, Autotomic Binary Reduction (ABR), is used in conjunction with a collection of non-localized in-place binary randomization techniques, Binary Structure Randomization (BSR), to simultaneously address multiple weaknesses in legacy embedded systems while side-stepping multiple technical constraints that make devising general and effective defenses difficult.

We present the theoretical operation of ABSR in this section. In Section 10 , we discuss performance, in terms of computational overhead, amount of binary diversity and empirical proof of safety, of ABSR as applied to three real-world devices, a MIPS-based router, a PowerPC-based switch, and an ARM computer running Linux and BusyBox. Lastly, we

present a case-study of ABSR defeating a well-known shell-code on Cisco IOS routers[1].

While the automated removal of code and data from firmware might intuitively seem unsafe at first glance, we demonstrate, in the remainder of this chapter, that the binary reduction algorithms presented in this dissertation will not negatively impact the functionality of the device given the satisfaction of several simple assumptions. This is discussed in depth in Section 9.1.2.

9.1.1 Software Diversification

Attackers often leverage the predictability of various features of software, like precise memory layout of code and data, to reliably exploit vulnerabilities. Attacks leveraging memory corruption, code injection and code reuse can usually be exploited reliably when no software diversification defenses are applied. A large body of work has demonstrated the efficacy of software randomization defenses against these common types of attack[72].

Theoretically, software diversification techniques can be applied to embedded device firmware the same way they are applied to software running on well-known general purpose operating systems. However, the unique constraints of most embedded systems, such as the lack of source-code and lack of operating-system level support, make many classes of software diversification techniques impractical.

Due to the proprietary nature of embedded devices and the general lack of source-code, compile-time diversification techniques sometimes cannot be practically applied to embedded device firmware. Load-time randomization techniques are also sometimes impractical for embedded devise due to the proprietary and non-standard nature of operating systems running within many embedded devices. For Cisco IOS is typically loaded into memory from non-volatile storage once, at boot time. While randomization mechanisms can be implemented in the boot loader, the distinction between operating system, user-land program, and libraries can be blurred within embedded firmware. The monolithic nature of special-purpose embedded software limits what the randomized loader can do, short of treating the

[1]The ABSR and Symbiote defenses have been formally red-teamed by MITLL. I am seeking permission to reproduce appropriate parts of their report.

entire firmware image as a single executable.[2] Furthermore, base-address randomization techniques like *Address Space Layout Randomization* provides minimal protection due to the fact that the vast majority of embedded devices operates within 32-bit address space, in which ASLR can be easily defeated due to the lack of address-space entropy.

Like load-time randomization, system level randomization techniques like *System Call Mapping Randomization* are impractical due to the monolithic and proprietary nature of embedded firmware. Recent work have demonstrated the feasibility of using syscall-based features in anomaly detectors in embedded Linux systems [119]. However, the applicability of such techniques within proprietary and non-standard operating-systems is currently unknown.

Techniques that involve hardware modification *Instruction Set Randomization* are impractical because they cannot be used to secure the large numbers of embedded devices already in deployment.

Localized in-place binary randomization techniques are promising for embedded systems. However, they are constrained to small localized changes because the randomized snippet must be the same size as the original code. Many more code randomization techniques can be applied if the randomized binary can be larger than its replacement. For example, basic blocks can be split and relocated with the introduction of unconditional jump statements. In order to take advantage of more complex binary-rewriting randomization techniques, we must account for the binary-size increase via some other means. This is, in part, the purpose of the proposed Autotomic Binary Reduction algorithm.

9.1.2 Attack Surface Reduction

[97] posits that security vulnerabilities are frequently the consequence of unwanted features in a software system. Such vulnerabilities result from overly bloated software, feature accretion, subsystem reuse and development errors on the part of designers and implementors

[2] The practical limitations of randomization techniques implemented within the bootloader of an embedded device can be seen in Cisco IOS. Versions 12.4 and later of the OS implements a memory randomization technique that shifts the entire IOS binary by a random positive offset from a known base address. This randomization technique can be easily defeated[35].

and vulnerability insertion on the part of attackers. For example, Cisco IOS ships with a full implementation of HTTP and HTTPS servers, hardly of use in most enterprise environments. The reduction of available attack-surface is a commonly used means to increase the security of computer systems. Thus, we propose that the automated reduction of attack surface within firmware binaries through the removal of unused software features can improve the security posture of embedded devices.

9.1.3 A Hybrid Approach

Modern embedded systems such as IP phones, network printers and routers typically ship with all available features compiled into its firmware image. A small subset of these features is activated at any given time on individual devices based on its specific configuration. An even smaller subset of features is actually used, as some unused and insecure features are enabled by default and cannot be disabled. As a result, many embedded devices still contain a significant amounts of code and data that should never be executed or read according to its current configuration. This unnecessary binary is not simply a waste of memory; it contains potentially vulnerable code and data that can be leveraged by an attacker to exploit the system. This unnecessary code provides an ideal attack surface. Automated minimization of this attack surface will significantly improve the security of the device without any impact to its functionality. The two components of our proposed technique are:

Autotomic Binary Reduction (ABR): The automated removal of unnecessary binaries from each embedded device according to its current configuration.

Binary Structure Randomization (BSR): The automated randomization of executable binary through a series of functionality-preserving transforms that alters the binary layout of the executable at the sub-basic-block granularity and up.

Autotomic Binary Structure Randomization (ABSR) combines automatic software attack surface reduction with a non-localized in-place binary randomization technique.

First, in the Autotomic Binary Reduction (ABR) phase, the firmware and configuration of the target device is analyzed together. The configuration of the device is analyzed for features that have been disabled by the operator. Such features are mapped to regions of code and data within the firmware binary using a hybrid control-flow and data-flow analysis

algorithm.

The regions of code and data identified ABR is removed from the original firmware image using our proposed Autotomy algorithm. Meta-data describing the virtual-memory and file offset ranges of code and data removed by the Autotomy algorithm is stored for use in the next phase of ABSR.

The Binary Structure Randomization (BSR) phase of ABSR is a non-local, in-place binary randomization technique that operates by introducing a large sequence of simple, localized, and functionality preserving transforms to the code and data content of the program it protects. ABSR does not depend on any special hardware capabilities or operating-system level constructions and does not increase the overall size of the protected binary. Thus, ABSR is particularly suitable for securing legacy embedded systems. While ABSR is applicable to all computing systems in general, we will focus specifically on its application in securing embedded devices in this chapter.

Theoretically, there are several alternative paths to engineering the security properties offered by ABSR in embedded devices. For example, the hardware of existing embedded devices can be redesigned to support features like secure boot by introducing TPM hardware and true ASLR by introducing an MMU that supports 64-bit address space. Similarly, their software stack can be rewritten to correctly leverage the newly introduced hardware.

Realistically, approaches that require hardware modification or software redesign are impractical. It is estimated by some commercial trade reports that there will be over 25 billion embedded devices in the world by 2015. Thus, systematic redesign of all such hardware and software is prohibitively expensive and will remain an academic exercise with little chance to make material impact to improve the security posture of existing embedded devices.

The ABSR process generally works as follows. First, the target firmware image is unpacked. Autotomic Binary Reduction (ABR) is then applied to identify regions of binary code and data that can be removed given a set of high-level features of the firmware determined to be disabled. The Autotomy algorithm replaces removed code and data with null-bytes. It also modifies the control-flow graph by inserting return statements in appropriate places to ensure that all removed code regions become dead-code. All removed

regions are given to a randomized free-pool space allocator and used by Binary Structure Randomization (BSR). After the application of BSR, Execution Detection Pads (XD_{pad}) are optionally injected into selected regions of the binary. Finally, the modified unpacked firmware is repacked into its original format and loaded onto the target embedded device. FRAK[33], the Firmware Reverse Analysis Konsole, is a generalized firmware unpacking, analysis, modification and repacking framework. We implemented the three components presented in this chapter, ABR, BSR and XD_{pad} as modifier components within the FRAK framework, allowing us to apply ABSR to arbitrary firmware images in an automated fashion.

We first present Autotomic Binary Reduction in Section 9.3 and Binary Structure Randomization in Section 9.5. We then present ABSR, a mutualistically defensive application of both techniques, in Section 9.6. We discuss the use of XD_{pad} to detect unauthorized code-execution in Section 9.5.3.

9.2 Threat Model

We assume the following remote exploitation threat model.

- Remote exploitation. Attacker knows the specific model of hardware and version of firmware of the target device but must remotely exploit the vulnerable device over the network.
- The attacker is able to obtain the original firmware binary that has not been processed by ABSR and analyze it statically and dynamically.
- The attacker possesses a typical remotely exploitable vulnerability that allows the attacker to modify memory and control IP register values.

Furthermore, we assume that the target device is a typical embedded device. The defender may analyze and modify any portion of the contents of the device's firmware. However, the defender does not have access to the source-code of the embedded device. We assume that the target device is a special purpose device. As such, its configuration and set of enabled features changes rarely.

9.3 Autotomic Binary Reduction

Autotomic Binary Reduction aims to remove regions of code within a body of software that can be considered *unreachable*, given a specific configuration for that firmware. Similar techniques are commonly used by modern compilers for the removal of dead-code and unreachable-code [39]. Generally, dead-code elimination aims to identify computation that is unnecessary, *i.e.* computation yielding data that is never used. In contrast, unreachable-code elimination aims to identify portions of code that will never be executed. Broadly speaking, dead-code elimination is performed inherently on the *data-flow* graph of a program, and aims to improve the efficiency of software by pruning unnecessary computation, whereas unreachable-code elimination is performed on the *control-flow* graph of a program, and aims to reduce the size of software. The proposed ABR algorithm described in this section takes a hybrid approach that leverages both the control-flow and the data-flow graph of a program in order to identify unreachable-code regions given a *specific configuration input* to the program.

ABR can be divided into two sub-problems: Feature Entry-Point Map ($F_{et}EM$) extraction and generalized Autotomy. The $F_{et}EM$ extraction sub-problem is intractable for the general case. However, the ABR algorithm presented in this chapter aims to solve only a

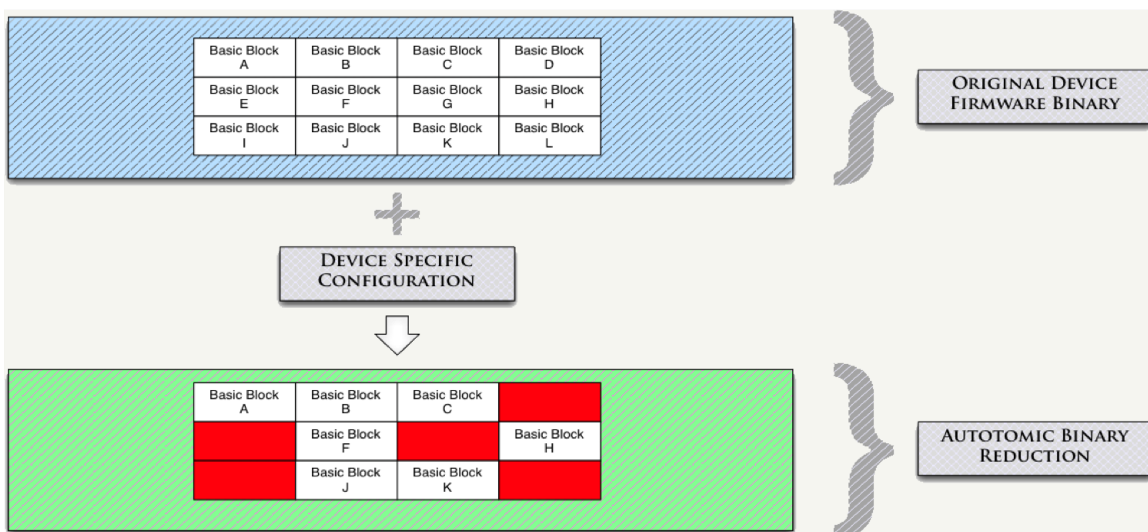

Figure 9.1: Autotomic Binary Reduction

subset of the general problem space. Several common constructs of real-world embedded device firmware make the Autotomy algorithm presented in this chapter feasible. The proposed Autotomy algorithm identifies and removes binary regions conservatively only when several constraints are satisfied. These constraints leverage several generalized heuristics describing common control and data flow patterns and ensure that code and data is removed only when the execution patterns comport to identifiable execution patterns.

The $F_{et}EM$ is a mapping between high-level descriptions of the firmware's features to the low-level code and data that implements those features. Specifically, this mapping is used to associate each feature of the firmware, such as the HTTP or SSH server, to a set of functions that are top-level entry-points of the code used to implement each feature. When a specific feature is selected for ABR, the specific $F_{et}EM$ entry for the feature in question is given as input to the generalized Autotomy algorithm, which in-turn returns a collection of code and data regions that is **exclusively** used by the high-level feature in question.

The $F_{et}EM$ is unique to each specific version of the embedded firmware binary. Experimental data suggests that this mapping can usually be created by analyzing the configuration file of the target embedded device along with its firmware. Straight forward application of static analysis of the corresponding firmware is typically sufficient to identify the initial entry point within the firmware executable for each feature identified in the configuration file. The result of static analysis can be further validated by standard dynamic analysis techniques.

Once the $F_{et}EM$ is identified for a particular firmware image, a specific device configuration file is analyzed to produce a list of features, which are administratively disabled. The $F_{et}EM$ and the list of disable features are given to the Autotomy algorithm to produce a unique firmware instance with the code and data associated with the disabled features removed from the firmware binary.

9.3.1 Generalized $F_{et}EM$ Extraction

For the general case, $F_{et}EM$ extraction of an executable program can be reduced to the halting problem and is undecidable. However, when the relevant portions of a particular program's control-flow can be represented as a *finite* graph, the $F_{et}EM$ extraction problem

can be reduced to the finite graph reachability problem, which can be solved in linear time [108]. Furthermore, the reachability problem for many classes of programs whose computational states can only be represented by infinite graphs have been proven to be decidable [26]. This suggests that the $F_{et}EM$ extraction problem can be practically solvable for categories of executable programs that are less well-formed and containing less meta-data than the programs that are considered in the context of this current work.

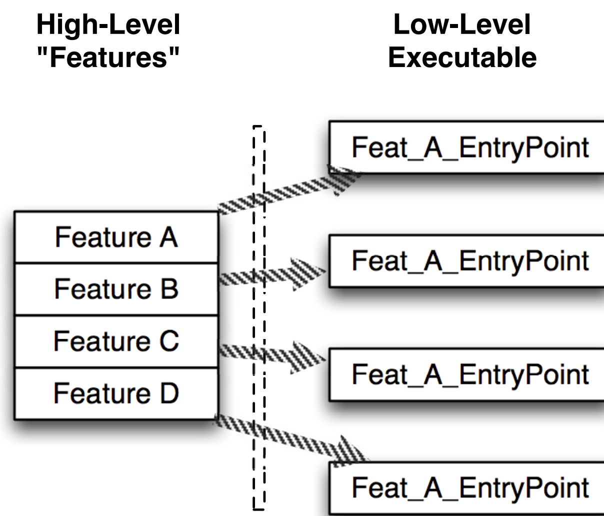

Figure 9.2: Generalized code structure useful for identifying Feature Entry-Point Maps in embedded firmware.

In the context of this document, we consider only programs whose Feature Entry Map logic can be represented as finite control-flow graphs. In practice, such graphs can be reliably extracted from most non-obfuscated code adhering to common software design conventions. To demonstrate feasibility, the $F_{et}EM$ extraction problem was solved three times, once for MIPS, ARM and PowerPC using three unique code-bases. The implementation of the three $F_{et}EM$ extractors are presented as case-studies in Section 9.6. The automation of the $F_{et}EM$ extraction problem is a subject of future research.

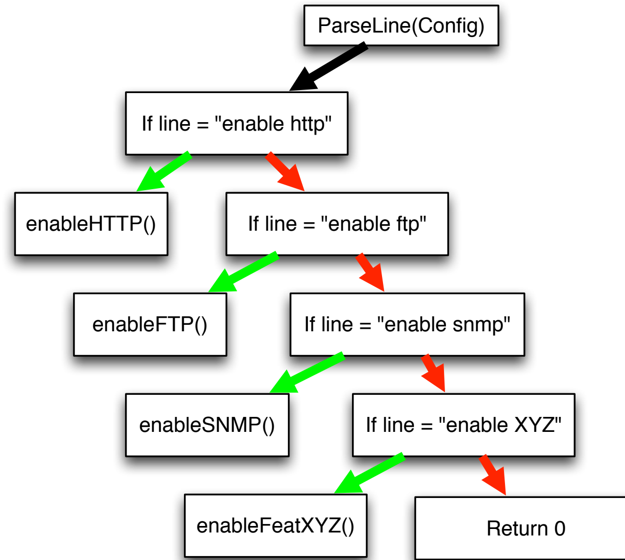

Figure 9.3: Typical finite graph representation of Feature Entry Map logic

9.3.2 General Autotomy Algorithm

We first describe a general Autotomy algorithm in this section. We then present an efficiently computable approximation to this general Algorithm in Section 9.4.2.

Given the $F_{et}EM$ for a specific executable, a set of features to be removed $\{f\}$, a control-flow graph (CFG) g_c and a data-flow graph (DFG) g_d, the Autotomy algorithm does the following:

1. Resolve set of executable entry points, $\{\epsilon\}$, using $F_{et}EM$ and $\{f\}$.
2. Identify set of code regions λ that is exclusively control-flow dependent on $\{\epsilon\}$ in g_c.
3. Identify set of data regions $\{\delta\}$ that is exclusively data-flow dependent to $\{\lambda\}$ in g_d.
4. Replace the beginning of each region in $\{\lambda\}$ and the remainder with either nop instructions or XD_{pad} instructions.
5. Replace regions within $\{\delta\}$ with null bytes.
6. Inject return hook at each member of $\{\epsilon\}$ using the appropriate return value.

Step 1 requires a simple lookup of the $F_{et}EM$ table. As we show in the next section, steps 2 and 3 can be reduced to the **Inverse Dominator** (DOM^{-1}) set problem and can

be solved in polynomial-time[73]. Step 4 and 5 involve in-place binary replacement of code and data. We discuss step 6 in Section 9.4.1.

Steps 2 and 3 aim to identify code and data regions (λ, δ) that are required if and only if code regions within f are executed. This can be reduced to the problem of computing the set (DOM^{-1}) of λ and δ given control-flow and data-flow graphs g_c and g_d. While polynomial-time solutions to DOM^{-1} exist, the practical application of the Autotomy algorithm relies on the ability to efficiently and accurately extract the control-flow and data-flow graphs from embedded device firmware, as well as the ability to make approximating compactions to the flow-graphs to a level of granularity that makes computation of the Autotomy algorithm feasible for large numbers of firmware images.

For Autotomy to be practical, the representative flow-graphs should be at least *finite*. Furthermore, it is desirable to reduce the size of both the representative control and data flow-graphs without discarding information that may impact the safety of the Autotomy algorithm. Intuitively, we can apply several types of approximations without violating the safety of the Autotomy algorithm.

9.3.2.1 Reachability can be over-estimated

The goal of the Autotomy algorithm is to identify portions of the flow-graph that are exclusively reachable through a specific set of nodes. Nodes within the flow-graph that are labeled as having an Immediate Dominator $(iDOM)$ node that outside the set of nodes removed by the Autotomy algorithm cannot be removed. Given a CFG x and an approximate graph $x\prime$ whose reachability between nodes are strictly greater than x, $iDOM(f, x\prime) \subset iDOM(f, x)$ must hold for any node in $\{f\}$. If we assume that the result of $Autotomy(f, x)$ is safe, an over-estimation of reachability will cause $Autotomy(f, x\prime)$ to return a subset of results. Thus, over-estimation of reachability cannot negatively impact the safety of the Autotomy algorithm. Note that in the extreme case, an over-estimation of the reachability of computed branch instructions in the firmware binary can cause all nodes to be marked as reachable from that instruction. This will cause the Autotomy algorithm to return a null-set. While undesirable, this does not negatively impact the safety of the resultant binary.

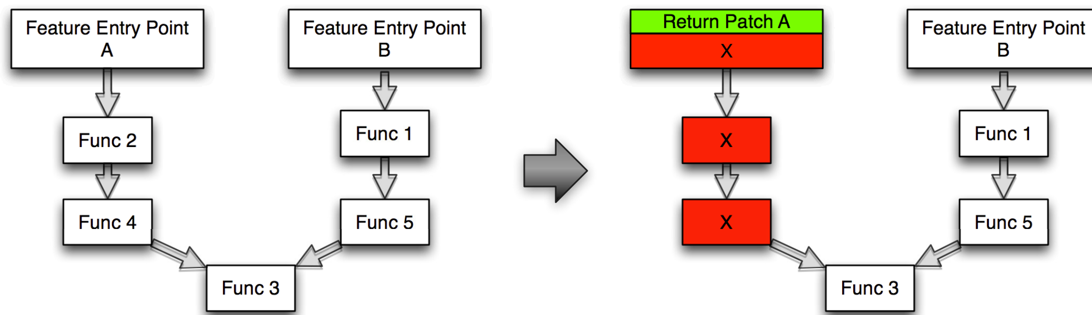

Figure 9.4: Autotomy of Feature Entry Point A. The red areas denote code regions which can be safely removed.

9.3.2.2 Control-flow graph can be approximated by Function Call Graph

We can approximate the control-flow graph of a program using its Function Call Graph (CG). While the creation of the exact static call graphs from software is undecidable in the general case [94], modern disassemblers like IDAPro can efficiently produce accurate finite call-graphs by opportunistically approximating code reachability. Note that tools like IDAPro can under-estimate code reachability. This is typically caused by errors in the disassembling algorithm and computed jumps and calls in the code. Such errors will cause the control-flow and data-flow graphs to be under-estimated, which will in turn negatively impact the safety of the Autotomy algorithm. Thus, ABR depends on the accuracy of the representative flow-graphs to guarantee safety of its operations.

Since over-estimation of this nature does not effect the safety of the Autotomy algorithm, we propose to apply the call-graph of the program to the Autotomy algorithm. This allows the Autotomy algorithm to remove code regions at the function level. We discuss an even more aggressive approximation to this approach in Section 9.4.2.

We first discuss the implementation of binary Autotomy for code (step 2). We then discuss the extension of this algorithm to remove both code and data (step 3).

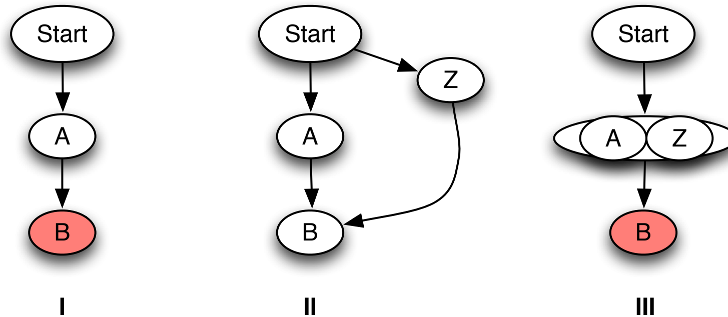

Figure 9.5: Three example flow-graphs to illustrate the **Dominator** function

9.3.2.3 Dominators and Inverse Dominators

For reference, we first present the **Dominator** and **Inverse Dominator** functions in the context of control-flow and data-flow graphs. [82] originally demonstrated that the Dominator function can be solved in $O(N^4)$. [73] has since presented nearly linear-time solutions to the same problem.

Given a flow-graph g and nodes a, b within that graph, a dominates b, a DOM b iff all paths from the root of g to b includes a.

Let the **Dominator** function, $DOM(b) = \{$set of all nodes which dominates $b\}$.

Let the **Inverse Dominator** function $DOM^{-1}(b)$ $=\{$set of all nodes which b dominates$\}$, $\{x \mid b \; DOM \; x\}$.

Figure 9.3.2.3 illustrates the Dominance relationship using three simple flow-graphs. In graph **I**, we can say that node A Dominates node B because all possible paths from the Start node to node B must go through node A. In graph **II**, an extra node, Z, is added. In II, neither node A nor Z can be said to Dominate B. In graph **III**, the nodes A and Z are collapsed into a single node, AZ. In this case, we can say that node B is Dominated by AZ.

Figure 9.3.2.3 illustrates the Inverse Dominator (DOM^{-1}) function using two simple flow-graphs. Intuitively, $DOM^{-1}(x)$ is set of nodes in the flow-graph that x dominates. In graph **I**, we can say that nodes {1,2,3,4,5} are all dominated by node X. Thus, in graph **I**, $DOM^{-1}(x)$ = {1,2,3,4,5}. In graph **II**, due to the addition of node Z, X no longer

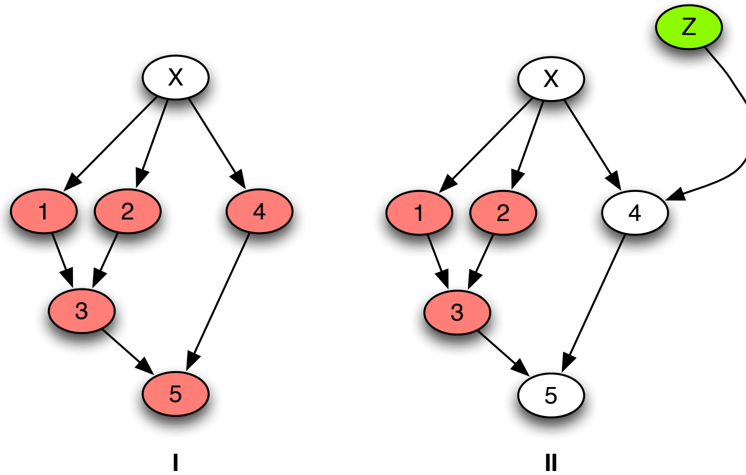

Figure 9.6: Three example flow-graphs to illustrate the **Inverse Dominator** function

dominates nodes {4,5}. Thus, in graph **II**, $DOM^{-1}(x) = \{1,2,3\}$

9.4 Code Autotomy Algorithm

The Autotomy algorithm requires the following as input; a control-flow graph g and a set of nodes f representing entry-points of features to be removed.

Algorithm 1 Code Autotomy Algorithm

Require: CFG g, code entry-point set f

1: **function** CodeAutotomy(g, f)

2: **return** $f \cup DOM^{-1}(f, g)$

9.4.0.4 Data Autotomy Algorithm

Similar computation can be performed on the data-flow graph of the firmware binary to identify removable data regions.

The Data Autotomy algorithm requires the following as input; a data-flow graph g_d, a control-flow graph g, and a set of nodes f representing entry-points of features to be removed.

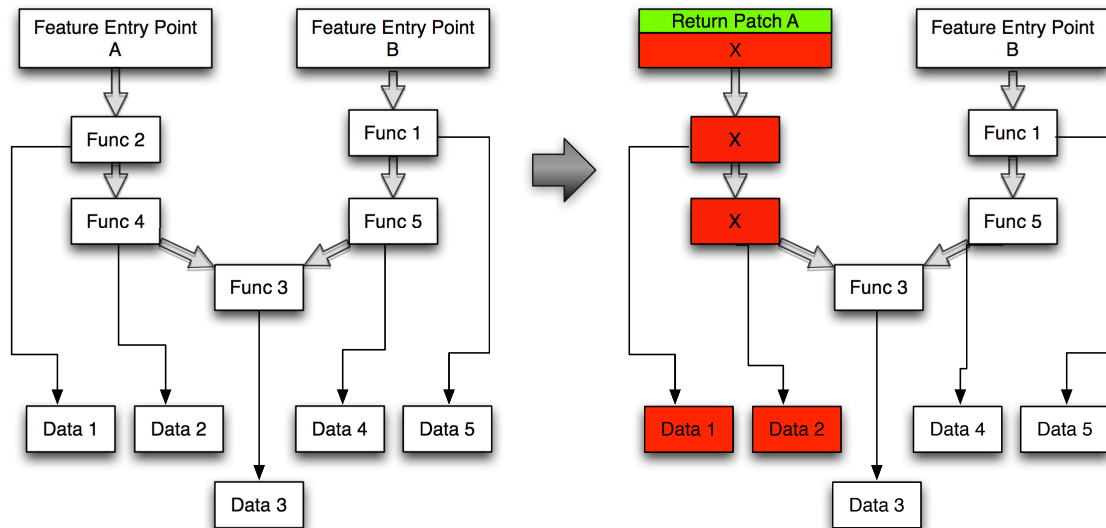

Figure 9.7: Autotomy of both code and data of Feature Entry Point A. The red areas denote code regions that can be safely removed.

Algorithm 2 Data Autotomy Algorithm

Require: CFG g, DFG g_d, code entry-point set f

1: **function** DATAAUTOTOMY(g_d, g, f)

2:     **return** $DOM^{-1}(CodeAutotomy(g, f), g_d)$

9.4.1 Return instruction injection

Given a set of Feature Entry-Points $\{\epsilon\}$, the Autotomy algorithm identifies all functions $\{\lambda\}$ that are exclusively reachable from only functions within $\{\epsilon\}$. Thus, if no functions within $\{\epsilon\}$ is called, we conclude that no code regions in $\{\lambda\}$ can be executed. Since $\{\lambda\}$ is never executed, it can simply be replaced by no-op instructions. However, the final removal of functions within $\{\epsilon\}$ requires the final step of injecting the appropriate return instruction sequence at the beginning of each member function within $\{\epsilon\}$. In practice, this step involves the construction and injection of a function return instruction, as defined by the specific function calling convention used by the target program. The function return instruction should return an appropriate value to indicate a failure condition. While the problem of identifying the proper return value to use can be reduced to the halting problem and is undecidable in the general case, commonly used software-engineering conventions make this problem solvable for many types of executables in practice. We present 3 such examples in Section 9.6. The methods used to identify proper return values in this chapter are similar to those proposed by [103] to virtualize error conditions to create self-healing software.

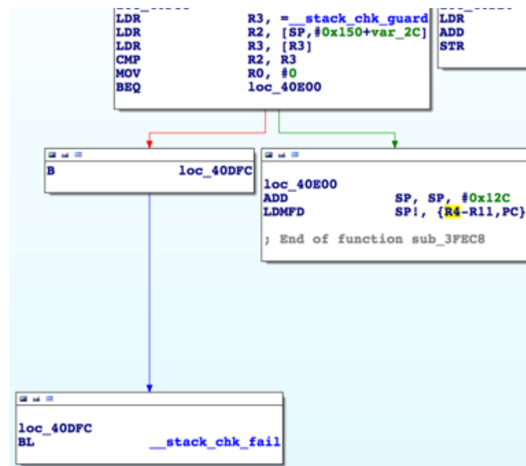

Figure 9.8: Feature Entry-Point Return Value Identification

Figure 9.8 illustrates the return code of a typical Feature Entry-Point in ARM. Simple static analysis determined that the return value (stored in register R3) is either 0 or some integer that adheres to standard Linux exit-code conventions.

9.4.1.1 Randomized Free-Pool Allocator

Regions of code and data removed by the Autotomy algorithm are tracked and managed by a free-pool allocator. This allocator is responsible for dispensing, in a randomized manner, newly freed regions of code and data to the Binary Structure Randomization algorithm described in Section 9.5.

9.4.2 Fast Code Autotomy Algorithm

We present **FastCodeAutotomy**, an efficiently computable approximation of the Autotomy algorithm presented in Section 9.3.2. The key to a fast Autotomy algorithm implementation is the reduction of the flow-graph considered and proper constraints on the algorithm's search-space. For example, the Cisco 2821 IOS firmware image we discuss in Section 10 contains 10,222 functions. A typical firmware binary of this type will potentially contain millions of basic-blocks. FastCodeAutotomy is designed to be usable as an "on-demand" computation. As we show in Section 10, a Python implementation of FastCodeAutotomy running within IDAPro on standard commodity hardware generally completes computation within seconds to minutes.

Instead of computing the **Inverse Dominator Set** by considering the entire flow-graph, FastCodeAutotomy returns a subset of what the Autotomy function returns by iteratively reducing an initial set of nodes $\{\beta\}$.

Given a flow-graph g and a set of nodes $\{f\}$ and an integer d, FastCodeAutotomy builds the initial node set, $\{\beta\}$, by enumerating all reachable nodes via breadth-first traversal, up to depth d. FastAutotomy then iteratively reduces $\{\beta\}$ until all members of $\{\beta\}$ is dominated by a member of $\{f\}$.

Note that the depth value d determines how deeply the FastCodeAutotomy function will look for potential removable code regions, whereas the set f will determine how many possible Dominator nodes the algorithm will consider. As we show in Section 10, given a fixed set f, increasing the value of d will cause FastCodeAutotomy to increase computation time and will potentially return more code and data regions. Given the same d, FastCodeAutotomy will potentially return more code and data regions if the functions in f are *functionally* related.

Algorithm 3 Fast Autotomy Algorithm

Require: CFG g, code entry-point set f, depth d

1: **function** BFSENUMERATE(g, f, d)

  return Set of nodes as enumerated by breadth-first search, starting at node f, to a depth of d.

2: **function** FASTCODEAUTOTOMY(g, f, d)

3:   $\beta \leftarrow$ BFSENUMERATE($g, f, depth = d$))

4:   **while** $True$ **do**

5:     $culled \leftarrow \{\}$

6:     **for** i in β **do**

7:       **if** XREFSTO(i, g) $\not\subseteq \beta$ **then**

8:         $culled \leftarrow culled \cup i$

9:     $\beta \leftarrow \beta \setminus culled$

10:     **if** $culled == \varnothing$ **then return** $\beta \cup f$

9.5 Binary Structure Randomization

Figure 9.10 shows the general spectrum of software randomization techniques. Given the constraints of proprietary embedded firmware, in-place binary randomization techniques are most feasible. However, in-place randomization techniques are restricted to making localized

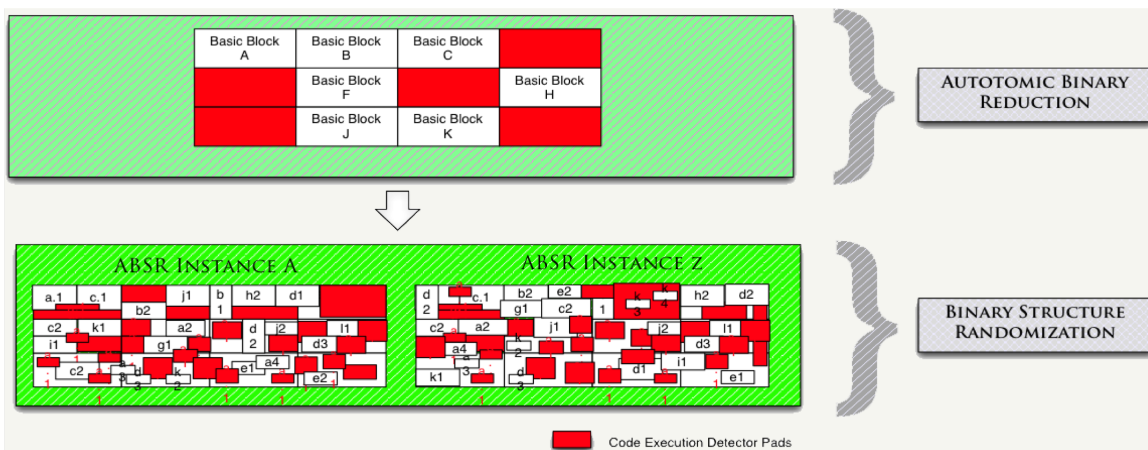

Figure 9.9: Binary Structure Randomization

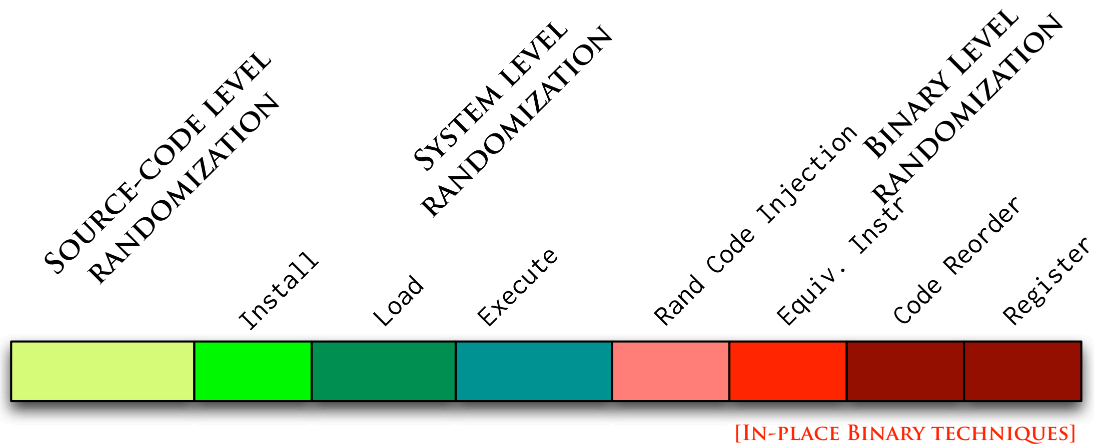

Figure 9.10: General

randomizations. For example, individual basic-blocks can be randomized using register-reassignment or equivalent-instruction replacement. While effective against certain classes of ROP attacks, the structure and binary layout of the software still remains unchanged. We aim to develop a non-localized randomization scheme that can change the layout of software at the binary level.

BSR can be considered as a non-localized, in-place binary randomization technique. Given a firmware image and a set of free-space regions within the firmware, BSR will generate a functionally equivalent firmware that is diversified and structurally randomized at a basic block level. Randomization is done by applying a large sequence of localized functionality-preserving transforms at the basic-block level. Using code and data regions freed by Autotomic Binary Reduction, basic blocks can be split and relocated into randomly selected positions within free-space regions given to the BSR algorithm by the ABR space allocator (Section 9.4.1.1).

For the purpose of this document, we define functional equivalence of software at an abstract ISA level. In other words, given the same input, register contents and execution state, the functionality of two pieces of code is equivalent if and only if the resultant register contents and execution states are identical after the execution of both pieces of code. Note, that micro-architectural perturbations, such as changes to the branch-predictors, instruction

and data caches are out of the scope of our current functionality preservation claim.

In general, BSR applies a series of functionality preserving transformations to a body of executable binary to yield a functionally equivalent executable that has significantly different binary layout. The BSR algorithm repeatedly applies such transforms in a randomized fashion until the desired binary randomization level is achieved.

We first present a collection of simple basic-block-level transforms or primitive BSR transforms. Next, we describe the creation of several complex BSR transforms by applying primitive BSR transforms in specific patterns.

9.5.1 Primitive Transforms

Any binary transform that preserves the functionality of the original executable is compatible with BSR. We first present a collection of BSR transforms, which operates on single basic blocks. We then discuss the functionality preserving property of each primitive transform as well as techniques for combining multiple primitive BSR transforms into composite BSR transforms. Unlike *localized* in-place code randomization techniques, BSR transforms are not bound by the same space and locality constraints. Due to the prior application of ABR, BSR can leverage functionality-preserving transforms that increase the size of the replacement binary and place such replacements in any suitable free-space region within the firmware image.

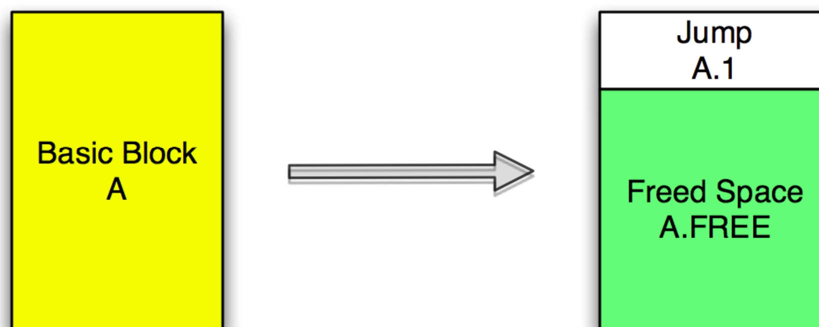

Figure 9.11: Binary Structure Randomization: Primitive Transform

9.5.1.1 Basic-block relocation

A basic-block of code can be defined as a series of instructions that must be executed in sequence, starting with the first instruction in the basic block and ending in the last instruction. Thus, a basic-block can have only a single entry-point and a single exit-point.

The relocation of a basic block β to new location ϵ involves the following modifications:

- Insert a non-conditional branch instruction to address ϵ at the entry-point address of β.
- Copy the contents of β to address ϵ. Let this new basic block be known as $\hat{\beta}$.
- Alter the exit-point branch instruction of $\hat{\beta}$ to branch to β.
- Insert the code-region occupied by β, less newly inserted branch instruction, into the free-space pool.

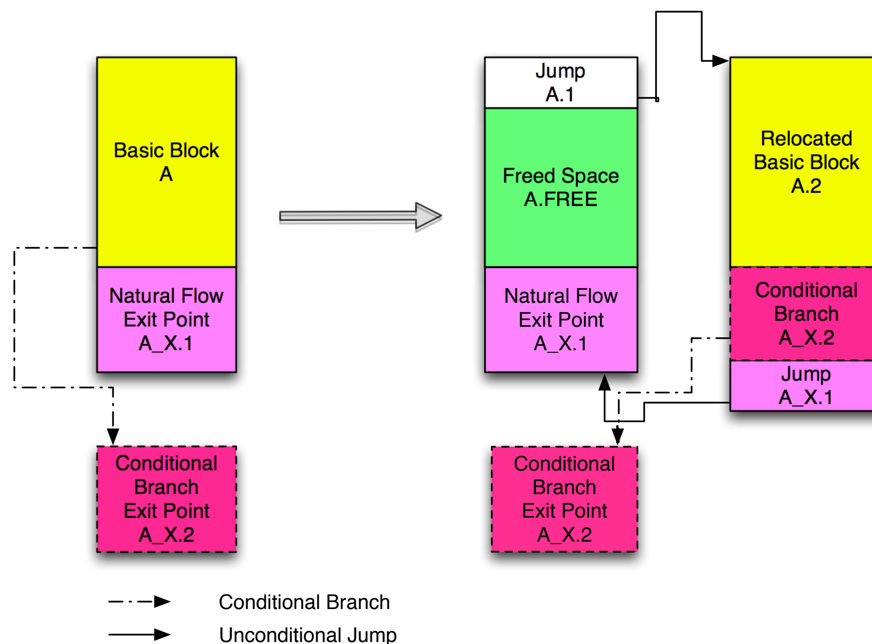

Figure 9.12: Binary Structure Randomization: Basic block Relocation

9.5.1.2 Basic-block splitting

Individual basic blocks can be split into two or more basic blocks with the addition of branches. As with relocation, care must be taken to find free regions of memory for not only the split basic block but also the addition of the branch instruction. A example of basic-block splitting is shown in Figure 3.

The split of a basic-block β to two new locations (ϵ_1, ϵ_2) involves the following modifications:

- Insert a non-conditional branch instruction to address ϵ_1 at the entry-point address of β.
- Split β into two sub-basic-blocks, β_1, β_2.
- Copy the contents of β_1 to address ϵ_1. Let this new basic block be known as $\hat{\beta}_1$.
- Copy the contents of β_2 to address ϵ_2. Let this new basic block be known as $\hat{\beta}_2$.
- Insert a non-conditional branch instruction at the end of $\hat{\beta}_1$ to branch to the first instruction of $\hat{\beta}_2$.
- Alter the exit-point branch instruction of $\hat{\beta}_2$ to branch to β.
- Insert the code-region occupied by β, less newly inserted branch instruction, into the free-space pool.

9.5.2 Complex BSR Transforms

The above two primitive BSR transforms, basic-block relocate and basic-block split, can be applied sequentially to form numerous **complex** BSR transforms.

9.5.2.1 Basic-block swap

Given two basic-blocks β_1 and β_2, a sequence of primitive BSR transforms can effectively swap the location of the two basic-blocks.

Note, since the basic-block relocate and split transforms both increase the size of the basic-blocks under transformation, purely in-place basic-block swap is not possible. As a result, each basic-block is first split into two sub-basic blocks according to the available size

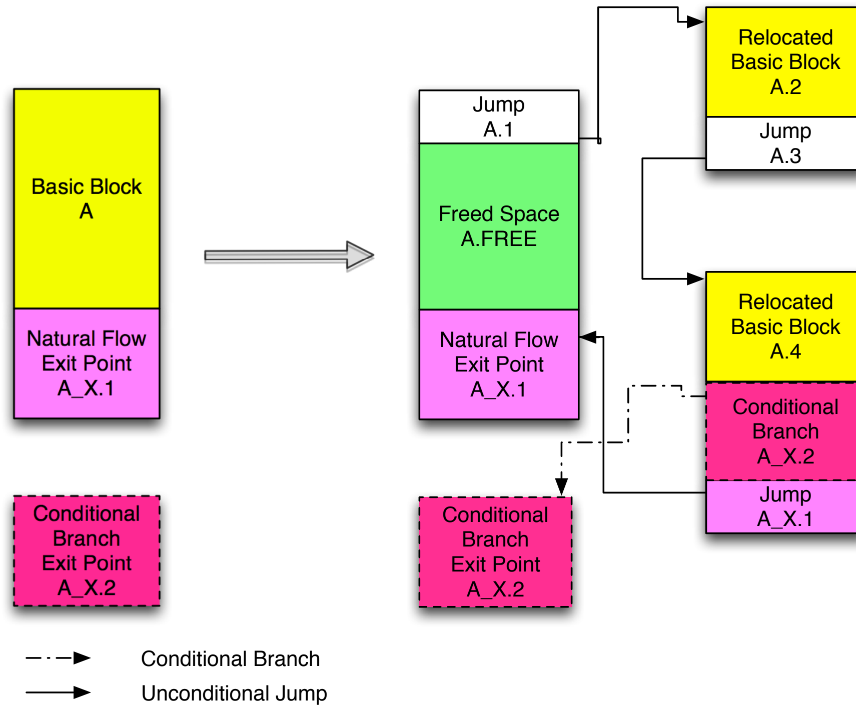

Figure 9.13: Binary Structure Randomization: Basic block splitting

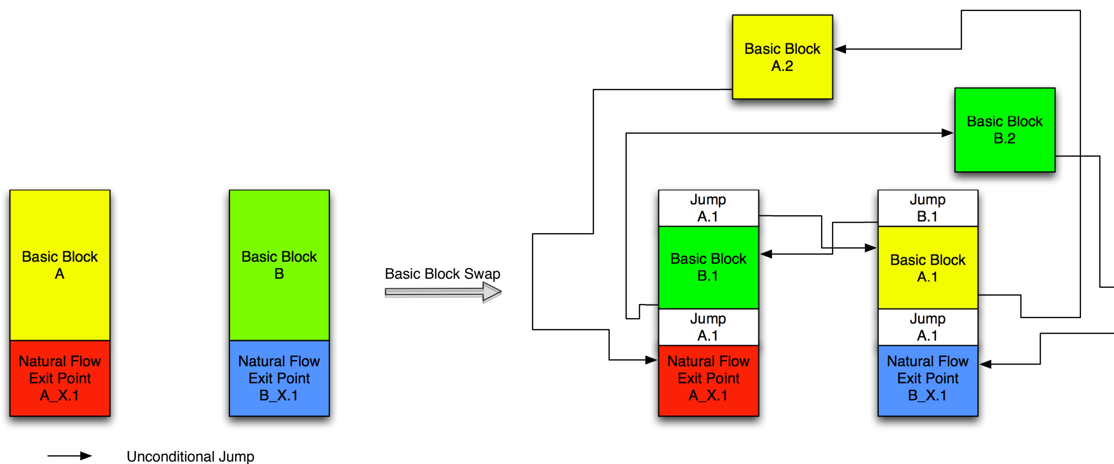

Figure 9.14: Binary Structure Randomization: Basic block swapping

of the other basic-block to be swapped. The first portion of each basic-block is swapped, while the second portion of each basic-block is randomly located to some other region as dictated by the ABSR free-space allocator.

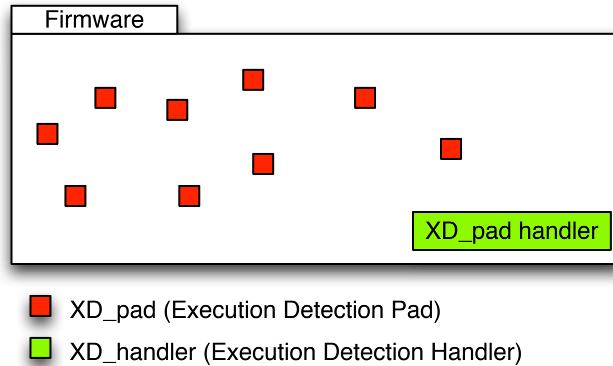

Figure 9.15: Code Execution Detection Detector Pads

9.5.2.2 Basic-block rotate

Similarly, given a list of n basic-blocks $(\beta_1, ...\beta_n)$, a circular application of basic-block **swap** can rotate the location of all basic-blocks, such that β_1 is relocated to β_2, β_2 is relocated to β_3, ..., β_{n-1} is relocated to β_n, and β_n is relocated to β_1.

9.5.2.3 Complex-block transforms

Let a **complex block** be defined as a collection of all basic blocks, which constitutes a subgraph of the control-flow graph of the target firmware executable such that there is only a single entry point and a single exit point for the control flow subgraph. Intuitively, we can see that all functions can be seen as complex blocks. However, complex blocks can also include control flow subgraphs within a function or control flow subgraphs comprising of multiple functions.

Complex-blocks can be relocated, split, swapped, and rotated by the sequential application of primitive BSR transforms.

9.5.3 ABSR: Code Execution Detection (XD_{pad})

Execution detector pads (XD_{pad}) can be injected into regions of free space generated by Autotomic Binary Reduction. Such detector pads can be used to detect unauthorized code execution. Any sequence of executable code that generates an observable side-effect

can be used as a detector pad. Using a sequence of BSR transforms, namely basic-block relocation and splitting, detector pads can be placed in virtually any location within the target firmware binary. Detector pads can be used as a trip-wire mechanism to detect execution of virtual-memory regions, which should not be executed. Detector pads can be used to detect any code-reuse attacks, which use payloads that rely on return-to-lib-c or ROP techniques.

9.5.4 Functional Preservation

Algorithm 4 Unconditional branch in MIPS

1: $PC \leftarrow nPC; nPC \leftarrow (PC\&0xf0000000)|(target << 2);$

BSR transforms are strictly additive and insert only unconditional branch instructions into the basic-blocks under transformation. Thus, demonstration of the functionality-preserving property of all primitive BSR transforms through the demonstration of the functionality-preserving property of unconditional branch instructions. Since unconditional-branches are possible in MIPS, ARM, and PowerPC without generating any additional side-effects in any CPU register or memory location, we claim that primitive BSR transforms are functionality-preserving in MIPS, ARM and PowerPC instruction sets. Furthermore, since complex BSR transforms are simply sequential applications of primitive BSR transforms, we further claim that all complex BSR transforms are also functionality-preserving in MIPS, ARM and PowerPC.

Note, the scope of our functionality-preservation claim is limited to the abstract ISA level. Micro-architectural perturbations, such as changes to the branch-predictors, instruction and data caches are out of the scope of our current claim.

9.6 Applied ABSR

We demonstrate two applications of ABSR within real-world devices running MIPS and ARM. To demonstrate the range of applicability of our technique, two bodies of code running on two different ISAs were selected. We first describe the application of ABSR within a

monolithic Busybox image compiled for ARM/Linux. We then describe the application of ABSR within Cisco IOS in MIPS.

While Busybox is an open-source project, our analysis of the Busybox executable is performed exclusively on a monolithic binary with debugging symbols stripped. This initial target was chosen to aid in the testing and evaluation of our techniques. Since the original source-code is available, the accuracy of our implementation of the Autotomy algorithm can be validated accurately. We then applied the same implementation of ABSR to Cisco IOS, a black-box operating system, on a different ISA. As our experimental data shows in this section, both applications of ABSR yielded stable executables with expected reduction in available attack surface and increase in binary layout randomization.

The following subsections describe implementation level details of two case-study applications of ABSR, in MIPS and ARM. Section 10 discusses the empirical performance overhead analysis of ABSR for each case-study and discusses the experimental methodology for validating correctness in each case-study.

9.6.1 ABSR in ARM BusyBox

We describe the specific implementation details use to apply ABSR in a monolithic Busybox image compiled for ARM/Linux. BusyBox can be described as "The Swiss Army Knife of Embedded Linux" and is a small executable that implements many common UNIX utilities. While Busybox does not have a configuration file that allows the user to administratively deactivate specific utilities (short of recompilation), we can use ABSR to generate randomized variants of BusyBox with any combinations of utilities disabled.

A BusyBox 2.21.0 binary image is analyzed by FRAK [33] and IDAPro 6.5. The MyNav 1.1.2 IDA plugin was used to assist the auto-analysis of the sample binary.

9.6.1.1 $F_{et}EM$ Extraction

We define each BusyBox utility (passwd, chmod, *etc.*) as a feature. Identification of each feature's main control-flow entry-point was achieved through static analysis. Specifically, the names and entry-points of all supported utilities can be enumerated by traversing a single lookup table. This lookup table is easily identifiable through a number of means.

```
loc_129FC
MOV          R1, R7
LDR          R3, =off_CFDEC
MOV          R0, R6
MOV          LR, PC
LDR          PC, [R3,R5,LSL#2]

BL           exit

; End of function sub_12810
```

Figure 9.16: Feature selection control-flow structure for Busybox

```
.rodata:000CFDEC off_CFDEC     DCD func_test_main       ; DATA XREF: sub_12810+1F0↑o
.rodata:000CFDEC                                        ; .text:off_12A48↑o
.rodata:000CFDF0               DCD func_test_main
.rodata:000CFDF4               DCD func_acpid_main
.rodata:000CFDF8               DCD func_remove_shell_main
.rodata:000CFDFC               DCD func_addgroup_main
.rodata:000CFE00               DCD func_adduser_main
.rodata:000CFE04               DCD func_adjtimex_main
.rodata:000CFE08               DCD func_ar_main
.rodata:000CFE0C               DCD func_arp_main
.rodata:000CFE10               DCD func_arping_main
.rodata:000CFE14               DCD func_sh_main
.rodata:000CFE18               DCD func_awk_main
.rodata:000CFE1C               DCD func_base64_main
.rodata:000CFE20               DCD func_basename_main
.rodata:000CFE24               DCD func_beep_main
.rodata:000CFE28               DCD func_blkid_main
.rodata:000CFE2C               DCD func_blockdev_main
.rodata:000CFE30               DCD func_bootchartd_main
```

Figure 9.17: Data associated with feature selection control-flow structure for Busybox

Figure 9.18: BusyBox $F_{et}EM$ Enumeration

We used a data reference to a string containing a well-known string to reliably identify the location of the lookup table. Figure 9.18 shows a disassembled and statically analyzed BusyBox image containing the $F_{et}EPM$ table.

9.6.1.2 Autotomic Binary Reduction

Figure 9.19 is a rendering of ABR applied to the BusyBox binary when all entries of the $F_{et}EM$ was removed except **unzip** and **sha512**. All non-black regions represent code removed by the ABR algorithm. 51.3% of original code was removed.

ABR was applied using each $F_{et}EPM$ entry iteratively, using input feature sets , $\{f\}$, of size varying sizes.

Figure 9.27 shows the output of ABR with a $\{f\}$ of size 353, containing all identifiable features. The executable text section of BusyBox used in our analysis is 795,768 bytes in size. At $d = 9$, the FastCodeAutotomy algorithm was able to remove 408,766 bytes or 51.3% of code.

9.6.1.3 Binary Structure Randomization

9.6.1.4 Performance and Overhead

We present the user cpu-time required to compute two BusyBox commands: *unzip* and *sha512*, on two sets of binaries; one random, one null of the following sizes: 1MB, 10MB,

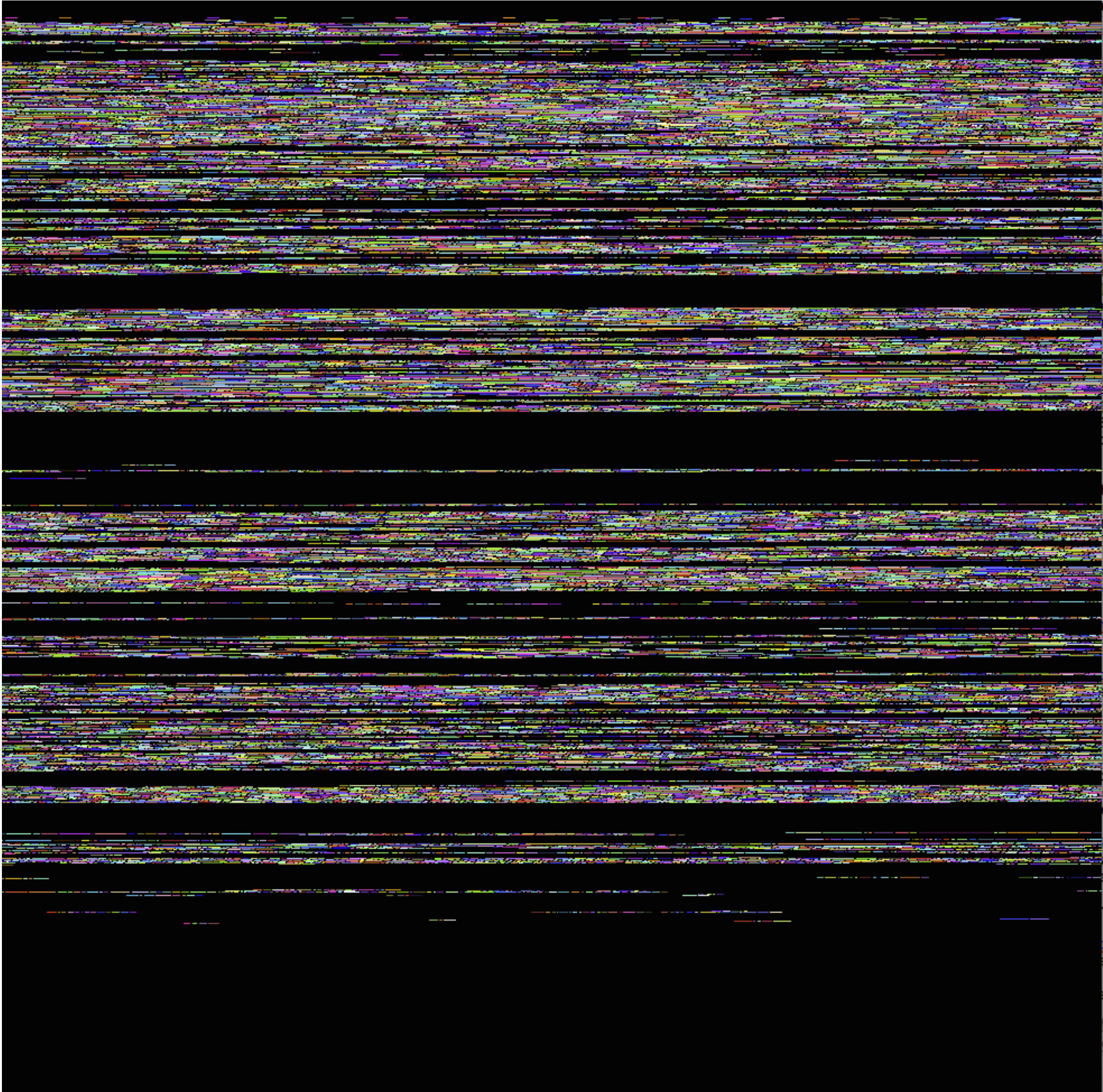

Figure 9.19: ABR applied to busybox. Non-black regions represent removed code regions.

100MB, 1000MB. In order to measure the computational overhead introduced by BSR, we identified and cumulatively relocated all basic-blocks associated with each feature. The basic-blocks are identified using a 3 level breadth-first traversal of the call-graph of each feature's entry-point.

Figure 9.30 shows the user cpu-time required to run the *unzip* command on a **1MB** binary composed of random data. Each of the 226 basic-blocks identified were relocated

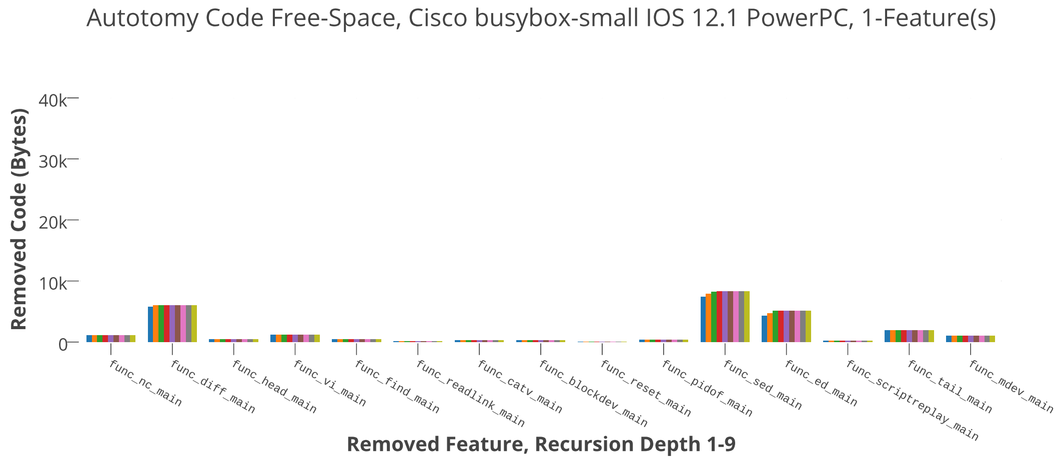

Figure 9.20: ABR applied to busybox, feature-set size=1

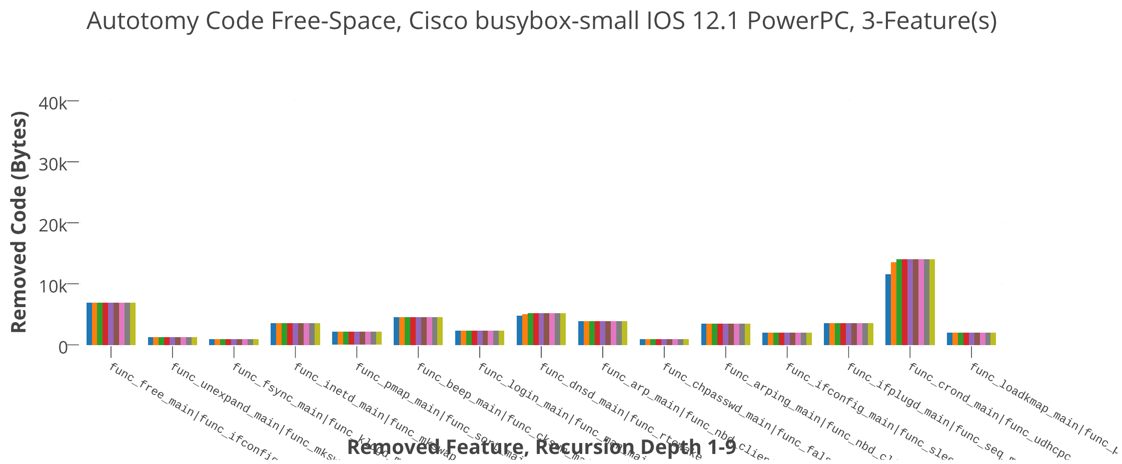

Figure 9.21: ABR applied to busybox, feature-set size=3

sequentially and cumulatively and executed 15 times over the same input data. Entry 0 represents the baseline measurement, where no modifications were made to the busybox binary.

Figure 9.30 also plots the average runtime of the same dataset. While the variance of the runtime data makes exact computational overhead difficult to determine, this figure suggests that the relocation of certain basic-blocks introduces significantly more performance overhead than others. This aligns with our theoretical model of BSR performance over-

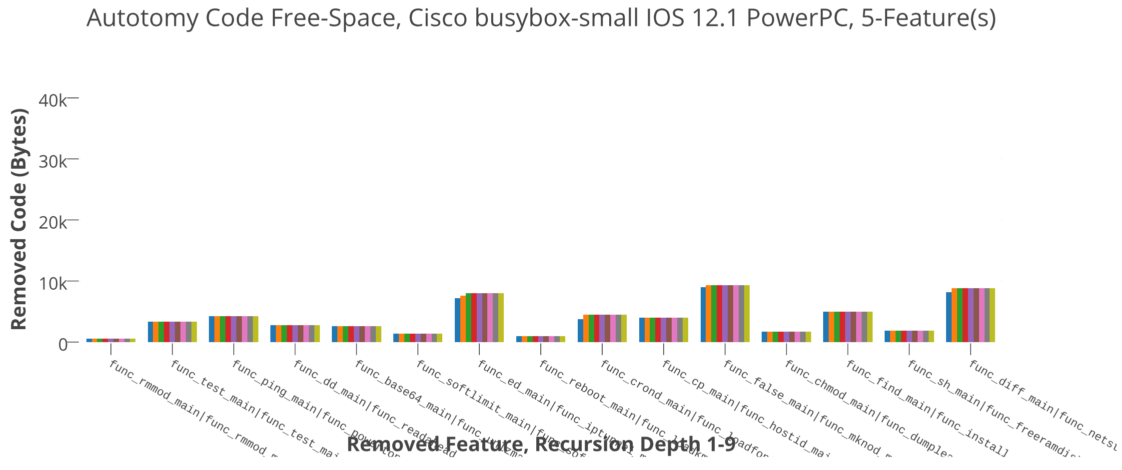

Figure 9.22: ABR applied to busybox, feature-set size=5

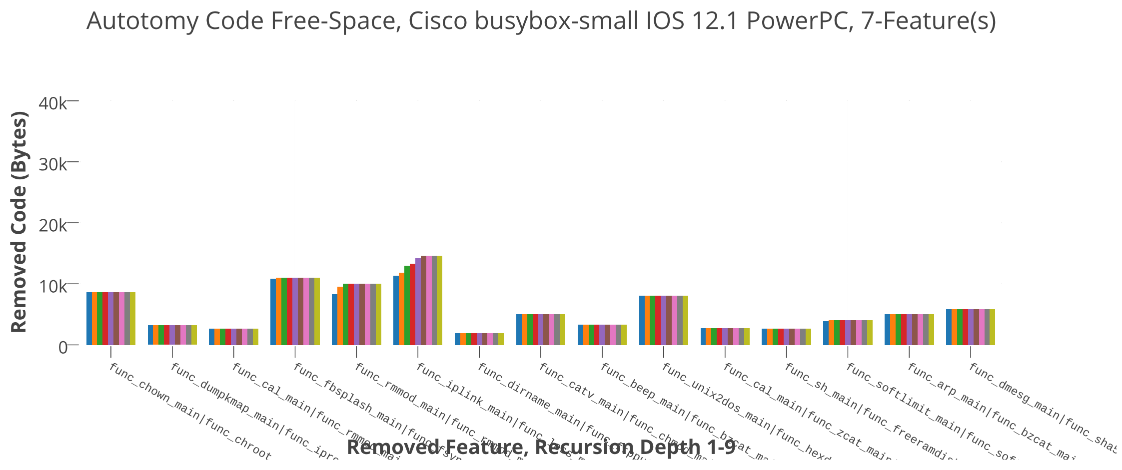

Figure 9.23: ABR applied to busybox, feature-set size=7

head as the total computational overhead is directly related to both the number of BSR transforms applied, and the frequency at which the transformed code is executed. This code invocation frequency is directly related to the code coverage probability distribution function and is input specific.

As we increase the size of data processed, the computational overhead across BSR transforms becomes more apparent. Figure 9.31 shows the performance data when a **10MB** random binary file is used as input of the unzip utility.

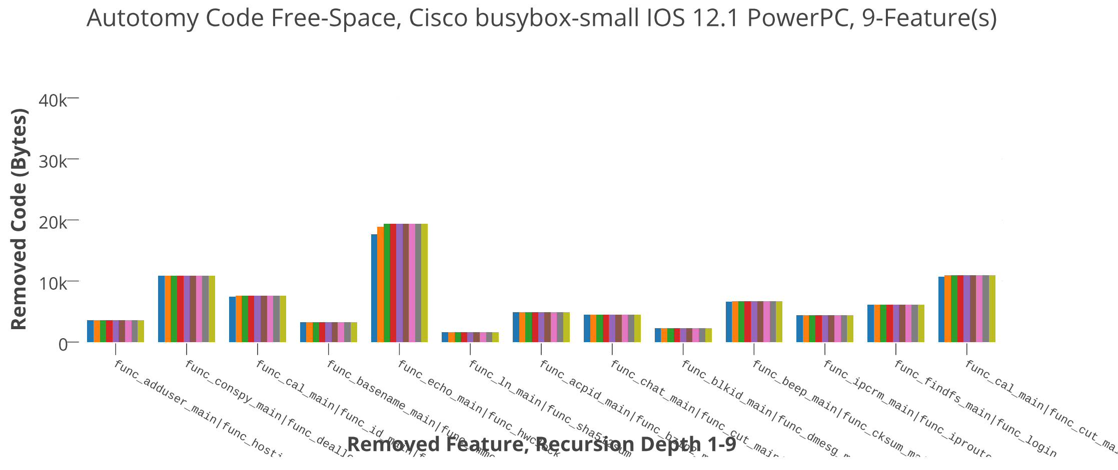

Figure 9.24: ABR applied to busybox, feature-set size=9

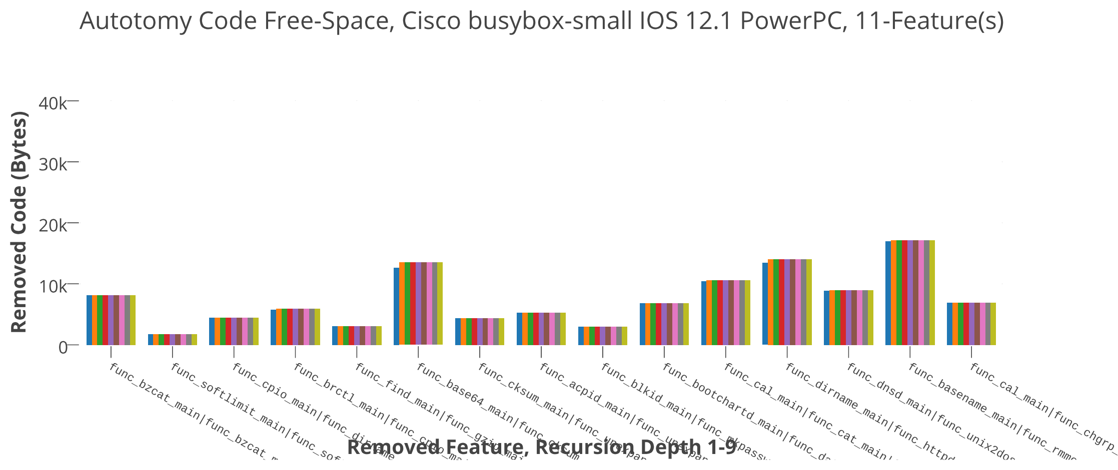

Figure 9.25: ABR applied to busybox, feature-set size=11

Figure 9.32 shows the user cpu-time required to run the *sha512* command on a **1MB** binary composed of random data. Each of the 149 basic-blocks identified were relocated sequentially and cumulatively and executed 5 times over the same input data. Entry 0 represents the baseline measurement, where no modifications were made to the busybox binary.

Figure 9.33 and 9.34 shows the completion times of the sha512 utility over 10MB and 100MB of random data. Unlike the unzip utility, we see that the number of BSR transforms

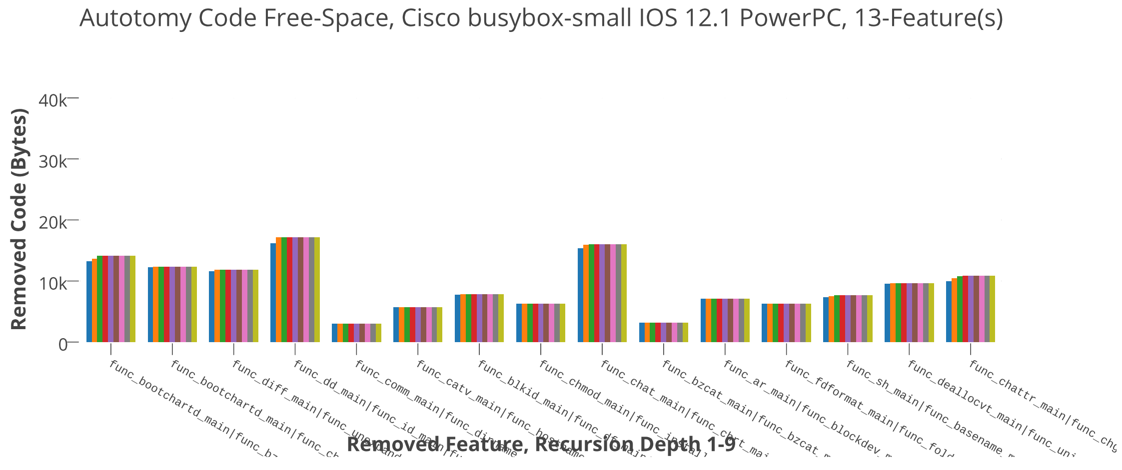

Figure 9.26: ABR applied to busybox, feature-set size=13

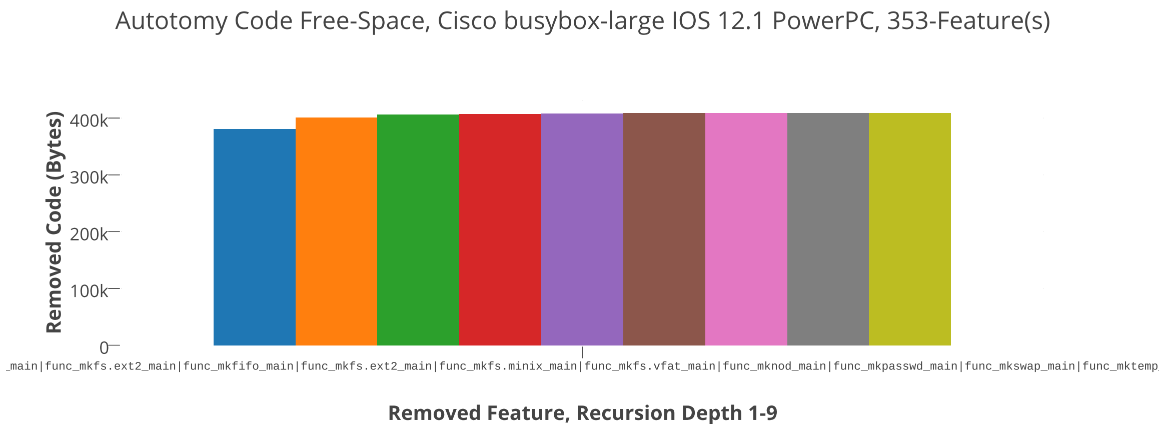

Figure 9.27: ABR applied to busybox, feature-set size=353

has no statistically significant impact on the completion time of the utility over baseline.

The data collected in this subsection also validates the safety of the ABSR algorithm in BusyBox. The computational output of every binary variant is tested against the expected output. The utility returned the correct output for 100% of test-cases.

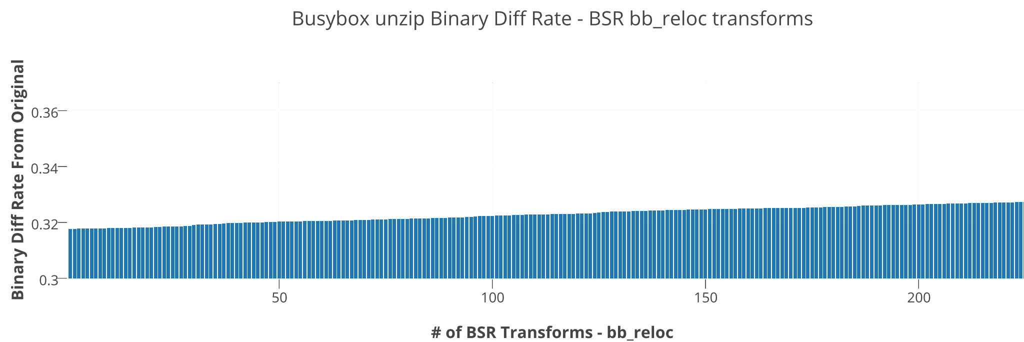

Figure 9.28: Binary Diff Rate vs Original Busybox Binary

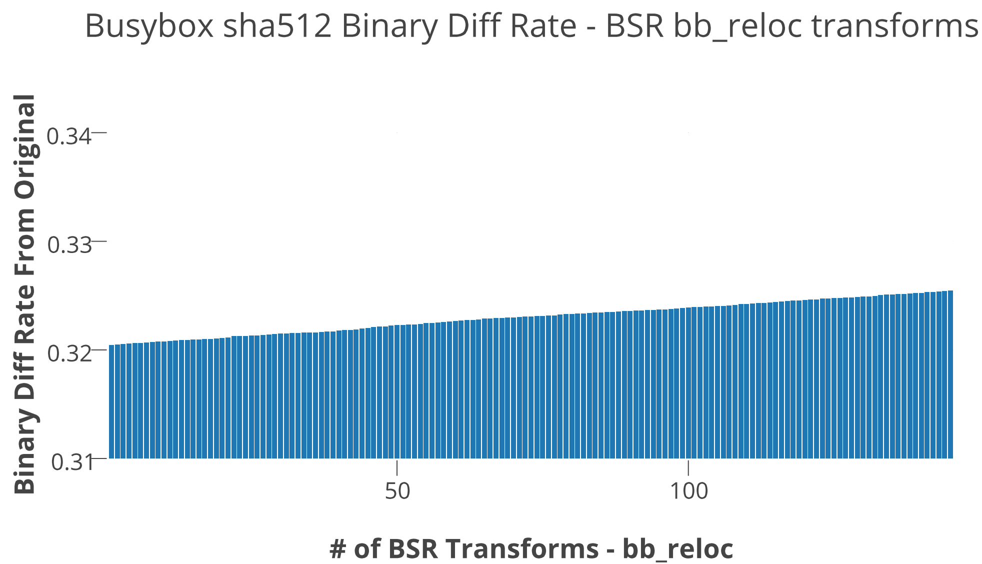

Figure 9.29: Binary Diff Rate vs Original Busybox Binary

9.6.2 ABSR in PowerPC Cisco IOS

We describe the application of ABSR on Cisco IOS 12.1 firmware running on the 3750 layer-3 switch.

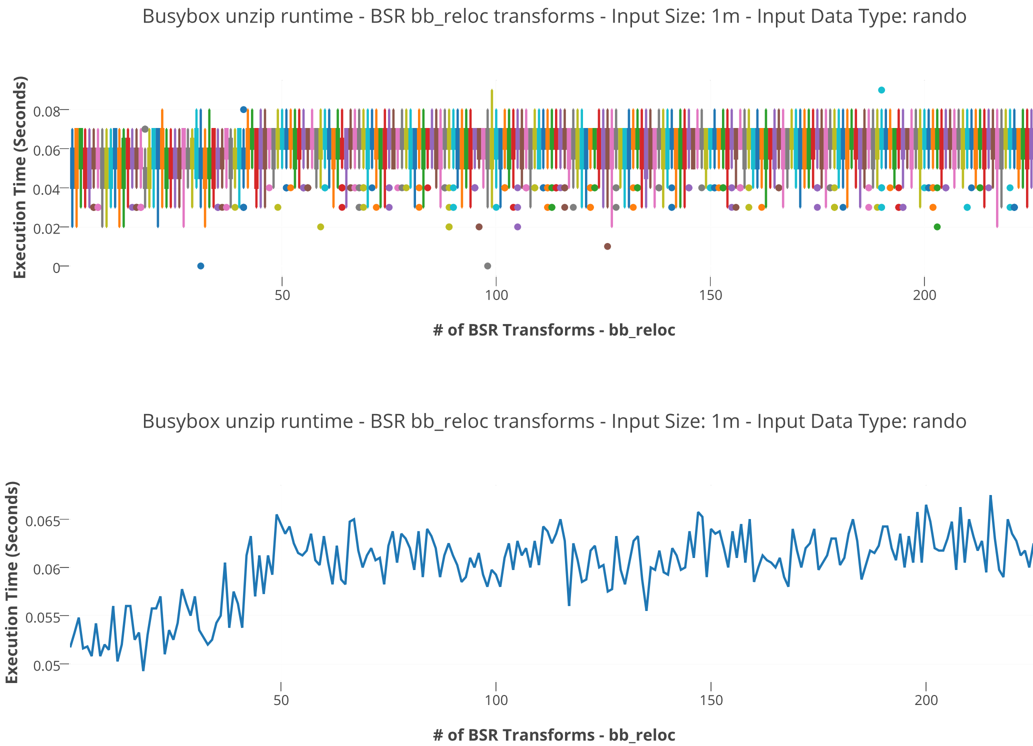

Figure 9.30: BusyBox **unzip** utility runtime over 1MB random data

9.6.2.1 $F_{et}EM$ Extraction

The IOS $F_{et}EM$ extraction algorithm leverages the call-graph associated with the process-creation process. For readability, we have re-named this function *ios_startProcess*. A mapping between the name of each IOS process and its main entry-point is extracted by enumerating all invocations of *ios_startProcess*. Figure 9.35 illustrates a specific invocation of *ios_startProcess* which starts the HTTP server.

The $F_{et}EM$ extraction algorithm identified 248 processes within the specific IOS image processed. Figure 9.36 shows a high-level rendering of a call-graph representing all invocations of the *ios_startProcess* function.

Figure 9.31: BusyBox **unzip** utility runtime over 1MB random data

9.6.2.2 Autotomic Binary Reduction

ABR was applied using each $F_{et}EPM$ entry iteratively and using input feature sets of increasing sizes.

Figure 9.40 shows the output of ABR with a $\{f\}$ of size 248, containing all identifiable features. The executable text section of Cisco IOS 12.1 used in our analysis is 11,864,844 bytes in size. At $d = 9$, the FastCodeAutotomy algorithm was able to remove 759,820 bytes or 6.4% of code.

Figure 9.32: BusyBox **sha512** utility runtime over 1MB random data

Figure 9.33: BusyBox **sha512** utility runtime over 10MB random data

Figure 9.34: BusyBox **sha512** utility runtime over 100MB random data

```
ROM:000B6EA8        lis     r3, sub_B681C@h
ROM:000B6EAC        addi    r3, r3, sub_B681C@l
ROM:000B6EB0        lis     r4, aHttpServer@ha # "HTTP Server"
ROM:000B6EB4        addi    r4, r4, aHttpServer@l # "HTTP Server"
ROM:000B6EB8        li      r5, 0x2EE0
ROM:000B6EBC        li      r6, 3
ROM:000B6EC0        bl      ios_startProcess
```

Figure 9.35: IOS Create Process function

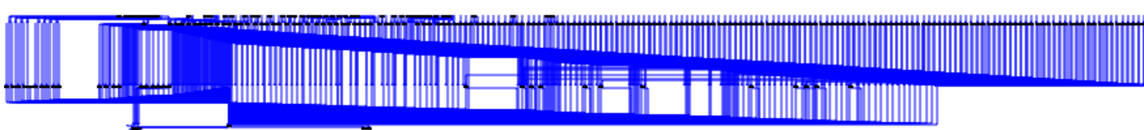

Figure 9.36: IOS Create Process function call-graph

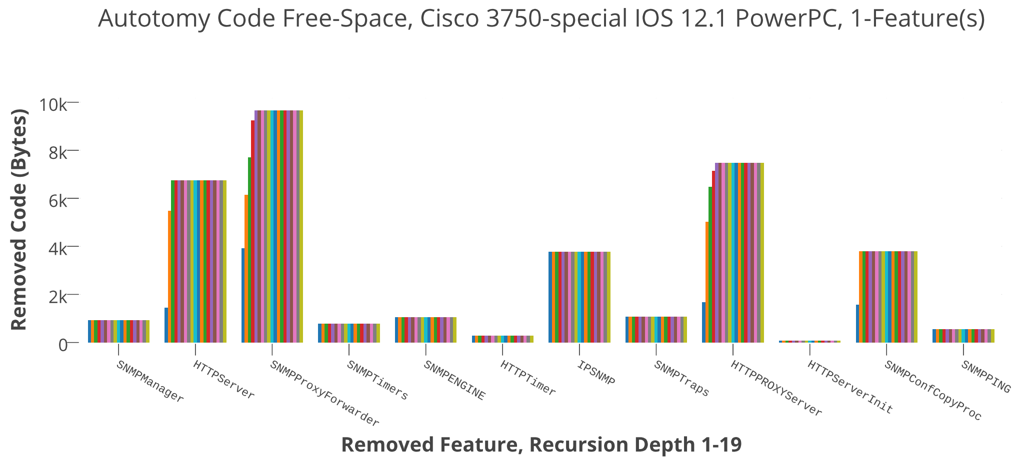

Figure 9.37: ABR applied to Cisco 3750 IOS 12.1, feature-set size = 1

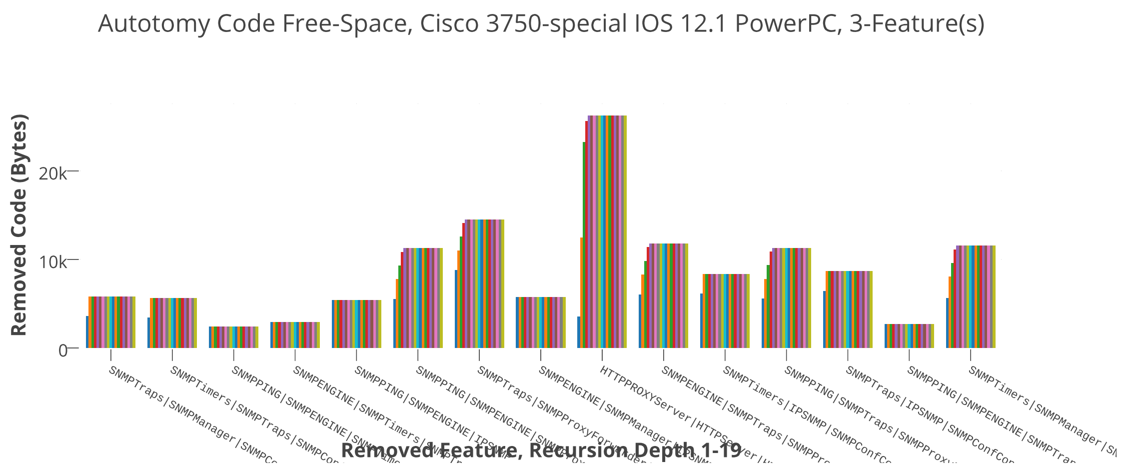

Figure 9.38: ABR applied to Cisco 3750 IOS 12.1, feature-set size = 3

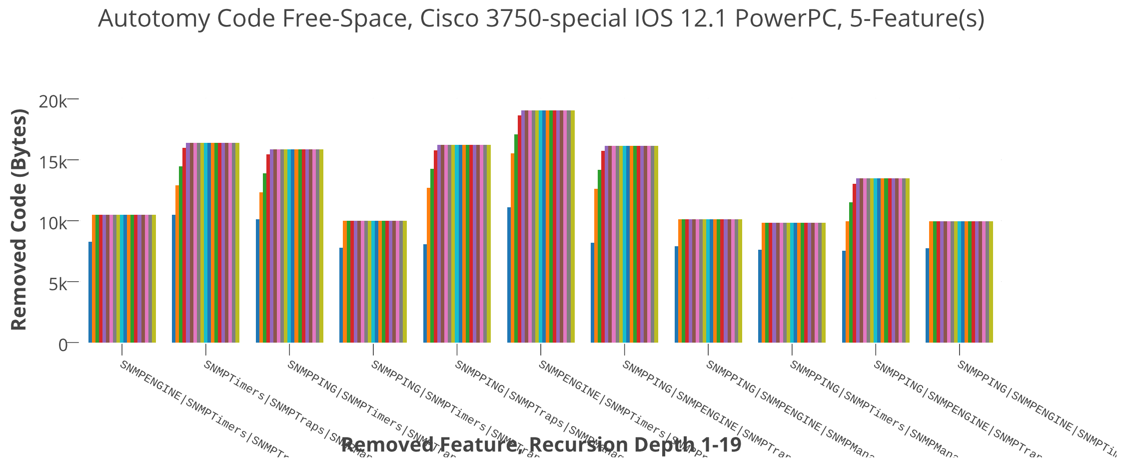

Figure 9.39: ABR applied to Cisco 3750 IOS 12.1, feature-set size = 5

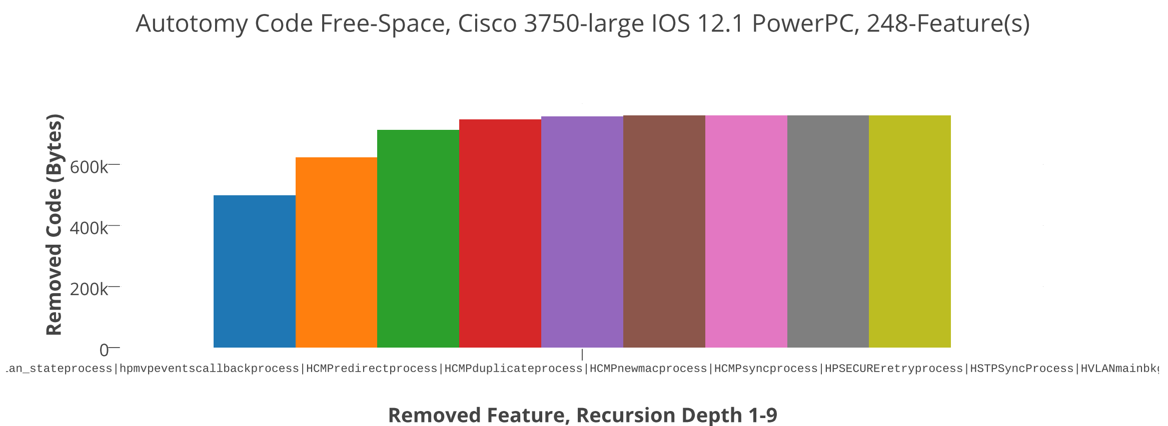

Figure 9.40: ABR applied to Cisco 3750 IOS 12.1, feature-set size = 248

9.6.3 ABSR in MIPS Cisco IOS

We describe the application of ABSR on Cisco IOS 12.3 firmware running on the 2821 router.

9.6.3.1 $F_{et}EM$ Creation

```
addiu   $a0, $a0, sub_81655C90
lui     $a1, 0x4462
addiu   $a1, $a1, aHttpProxyServe   # "HTTP PROXY Server"
addiu   $a2, $zero, 0x2EE0
jal     ios_startProcess
addiu   $a3, $zero, 3
j       loc_826AF430
addu    $s0, $v0, $zero
```

Figure 9.41: IOS Create Process function

The IOS $F_{et}EM$ extraction algorithm leverages the call-graph associated with the process-creation process. For readability, we have re-named this function *ios_startProcess*. A mapping between the name of each IOS process and its main entry-point is extracted by enumerating all invocations of *ios_startProcess*. Figure 9.41 illustrates a specific invocation of *ios_startProcess*, which starts the HTTP server.

The $F_{et}EM$ extraction algorithm identified 669 processes within the specific IOS image processed.

9.6.3.2 Autotomic Binary Reduction

ABR was applied using each $F_{et}EPM$ entry iteratively and using input feature sets of increasing sizes.

Figure 9.45 shows the output of ABR with a $\{f\}$ of size 669, containing all identifiable features. The executable text section of Cisco IOS 12.3 used in our analysis is 45,987,892 bytes in size. At $d = 9$, the FastCodeAutotomy algorithm was able to remove 559,448 bytes, or 1.2% of code.

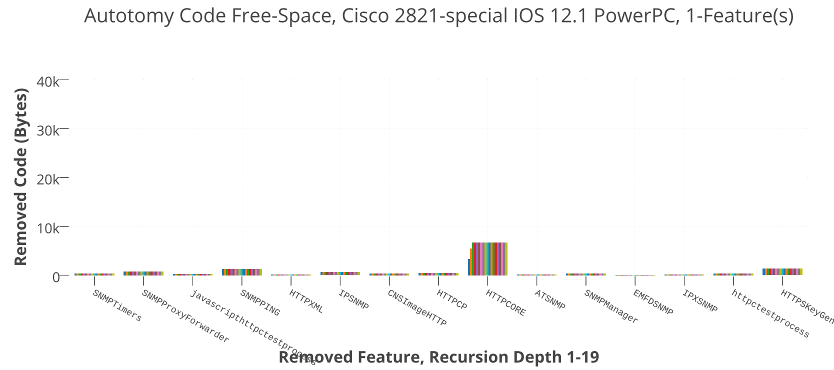

Figure 9.42: ABR applied to Cisco 2821 IOS 12.3, feature-set size = 1

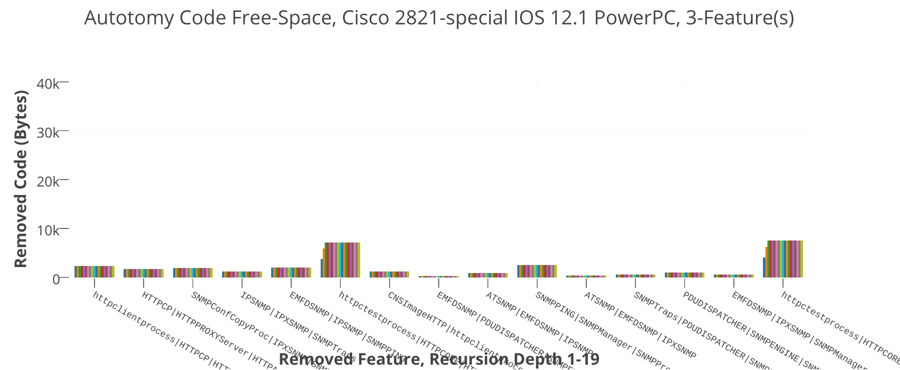

Figure 9.43: ABR applied to Cisco 2821 IOS 12.3, feature-set size = 3

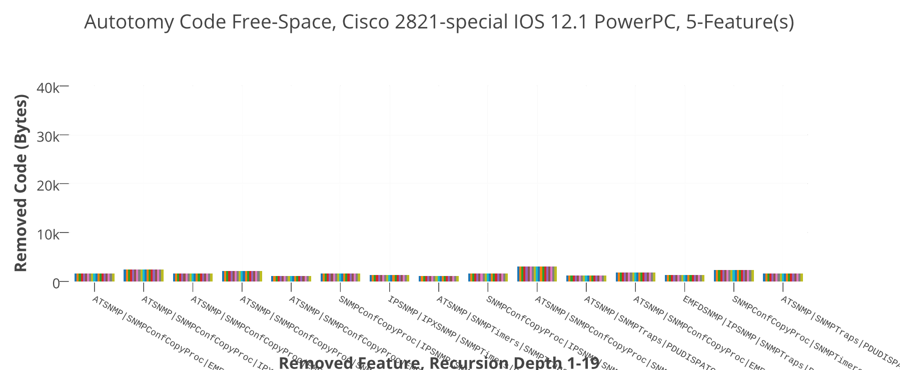

Figure 9.44: ABR applied to Cisco 2821 IOS 12.3, feature-set size = 3

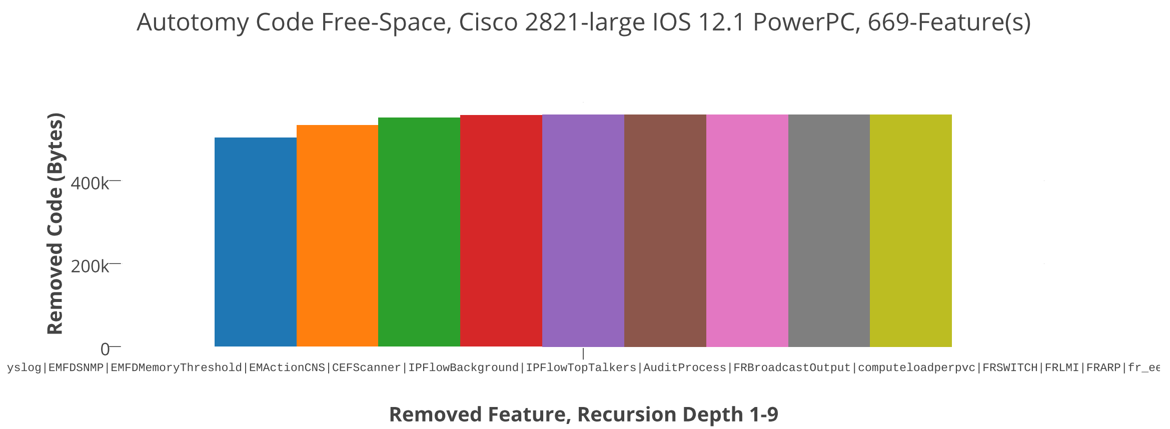

Figure 9.45: ABR applied to Cisco 2821 IOS 12.3, feature-set size=669

Chapter 10

Case-Study: Symbiote and ABSR Defense

10.1 Case-Study: Symbiote and Cisco Routers

10.1.1 Symbiote Performance and Overhead

We measure the performance and overhead of our Symbiote-based exploit detector using two quantitative metrics: *computational overhead* and *detection latency*. Figure 10.1 illustrates the testing and verification environment used to obtain the performance data presented in this section. The Symbiote-protected router is an emulated Cisco 7200 series router running IOS 12.3. Two neighbor routers are used to verify that the Symbiote-protected router's original functionality is unchanged. One neighbor router is an emulated 7200 series router running standard IOS 12.3. The other neighbor router is a physical Cisco 2921 router running IOS 12.5. Each router is configured to expose a cross-section of functionality typically seen on production routers. Specifically, a large number of local loopback interfaces are configured on each router. OSPF routing is enabled on all three routers, along with a suite of standard services like IP-SLA, SNMP, HTTP{S}, and SSH.

A stress-test script automatically generates network traffic throughout the test environment and periodically accesses services on all the test routers. All routers are continuously monitored to ensure that all services operate correctly throughout testing. The workload

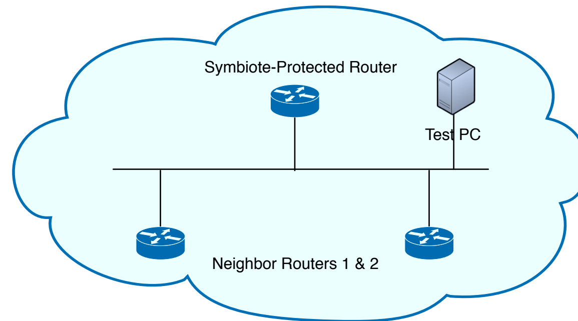

Figure 10.1: Symbiote-based Cisco IOS Detector Testing and Verification Environment

script also periodically forces route-table re-calculations by perturbing the various OSPF routers on the network. In effect, the stress-test script simulates a typical use profile for the IOS routers in the test environment. The same stress-test script is run against several variants of the Symbiote-injected IOS firmware in order to illustrate key performance features of our system.

The computational overhead and performance of our system is a configurable parameter. As the figures in this section shows, the scheduling algorithm used within the Symbiote Manager directly impacts the resource consumption of the Symbiote payload and, thus, the overall utilization of the host device as well as the detection latency. Two scheduling algorithms are discussed in this section: *fixed burst-rate* and *inverse-adaptive*.

As the name suggests, the fixed burst-rate scheduling algorithm instructs the Symbiote payload to execute for a fixed burst-rate each time the Symbiote Manager is invoked through a randomly placed execution intercept. On the other hand, the inverse-adaptive scheduling algorithm calculates the payload burst-rate based on the elapsed time since the Symbiote Manager was last invoked; the longer the elapsed time, the longer the burst-rate.

Intuitively, we can expect the fixed burst-rate scheduling algorithm to execute the Symbiote payload *more frequently* as the host system becomes more utilized. This simple algorithm executes the Symbiote payload more frequently when the Cisco router is heavily utilized and less frequently when the router is idle. In contrast, the inverse-adaptive scheduling algorithm increases Symbiote payload burst-rate when the system is under-utilized and throttles back the Symbiote payload when the router is under high load.

We analyze the performance of 15 Symbiote-injected IOS images under the same stress-test: 7 variants using the fixed burst-rate Symbiote scheduler and 8 variants using the inverse-adaptive Symbiote scheduler. As the next three subsections show, the fixed burst-rate Symbiote scheduler aggressively executes the Symbiote payload and achieves the least detection latency (approximately 400 ms). However, this aggressive scheduler tends to amplify CPU utilization of the protected router, causing very high control-plane latency when the router is under load. Although the higher fixed burst-rate values like 0x7FF and 0xFFF detected IOS modification very quickly, it also caused the router's control-plane to be less responsive.

In contrast, the inverse-adaptive Symbiote scheduler produced slightly longer detection latencies (approximately 450 ms) but was able to significantly reduce the control-plane latency of the host router, even under high load.

10.1.2 Computational Overhead

The same stress-test script is run against various versions of the Symbiote-injected IOS image in order to show how the Symbiote Manager's scheduling algorithm affects the CPU utilization of the router. Two major scheduling algorithms are measured: fixed burst-rate (Figure 10.2) and inverse-adaptive (Figure 10.3).

Figure 10.2 shows the CPU utilization of 7 variants of the *fixed burst-rate* Symbiote scheduler, which unconditionally executes the Symbiote payload for a constant number of CPU cycles each time the Symbiote is invoked via its many control-flow intercepts. The units used, burst-rate, is the number of iterations of the checksum Symbiote payload that is executed each time the Symbiote Manager is invoked.

This Symbiote scheduler disregards the current CPU utilization of the host device. At higher burst-rate values like 0x7FF and 0xFFF, the router's CPU utilization tends to remain above 95% under heavy load, causing large spikes in control-plane latency. (See Figure 10.6)

Figure 10.3 shows the CPU utilization of 8 variants of the *inverse-adaptive* Symbiote scheduler, compared with the baseline CPU utilization of the unmodified IOS image under the same stress-test. The inverse-adaptive scheduler is configured with *maximum* burst-rates from 0xFF to 0x3FFFF. Unlike the fixed burst-rate Symbiote scheduler, the inverse-adaptive scheduler throttles how much the CPU is diverted to the Symbiote based on current host device utilization. As a result, Symbiotes with inverse-adaptive schedulers can achieve comparable detection latencies while significantly reducing its impact on the host router's control-plane latency. (Compare Figure 10.6 and Figure 10.7).

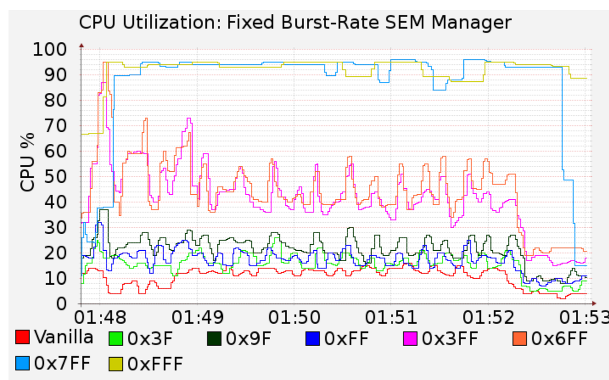

Figure 10.2: CPU Utilization : Fixed Burst-Rate SEM Manager

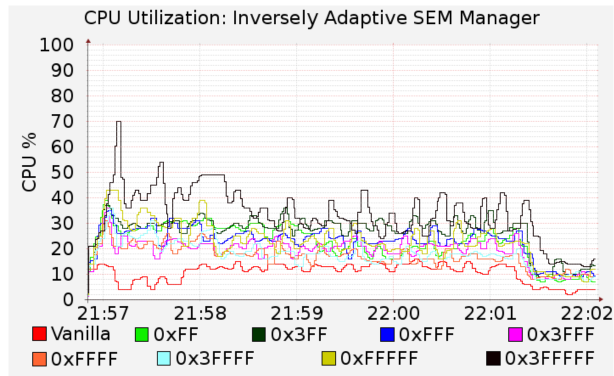

Figure 10.3: CPU Utilization : Inverse-Adaptive SEM Manager

10.1.3 Detection Performance

In order to measure the detection latency of our exploitation detection Symbiote, a simple vulnerability that allows arbitrary memory modification is artificially introduced into the Symbiote-injected IOS image. This vulnerability is triggered using an automated script and modifies a random byte within monitored memory regions. A timer is simultaneously started in order to measure the time it takes the Symbiote payload to detect the event.

Figure 10.4 shows a roughly linear relationship between the Symbiote's fixed burst-rate value and the Symbiote's detection latency. As expected, the Symbiote detection latency decreases as the Symbiote payload's execution burst-rate increases. However, as Figure 10.6 shows, the fixed burst-rate Symbiote scheduler causes significant increases in the router's control-plane latency.

Figure 10.5 shows the Symbiote's impact on the router's control-plane is significantly reduced.

10.1.4 Control-Plane Latency

Control-plane latency is an indicator of how responsive the router is. High control-plane latency can cause a router to drop routing adjacencies and break various time-sensitive network protocols. Note, however, this measurement will not significantly affect the latency of traffic passing through the router, as most modern routers have hardware-accelerated forwarding engines, which are decoupled from the control-plane.

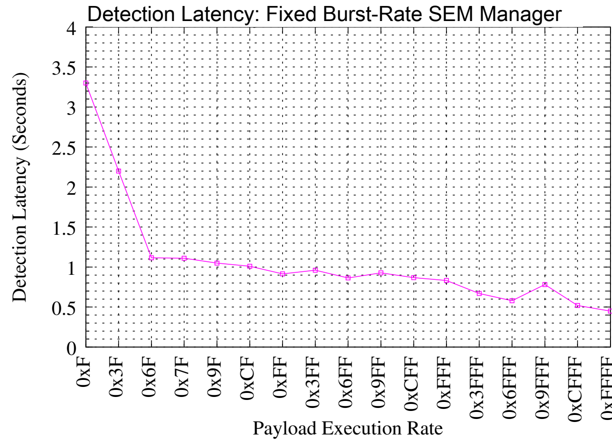

Figure 10.4: Detection Latency : Fixed Burst-Rate SEM Manager

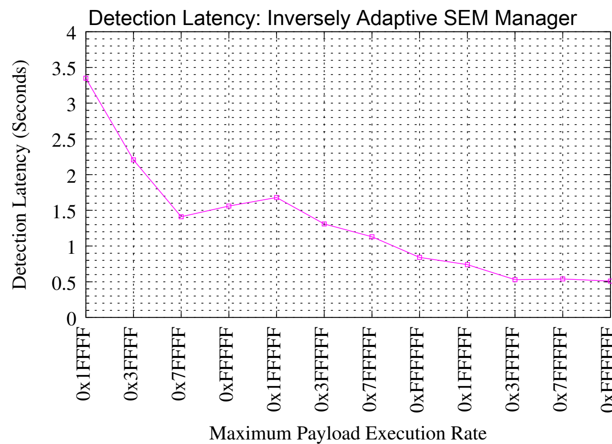

Figure 10.5: Detection Latency : Inverse-Adaptive SEM Manager

Control-plane latency is measured by sending ICMP-echo messages from the test PC to the router's local loopback interface. The round-trip-time is collected and shown in Figure 10.6 for Symbiotes using fixed burst-rate scheduler variants and in Figure 10.7 for Symbiotes using inverse-adaptive scheduler variants. Clearly, the inverse-adaptive Symbiote scheduler significantly reduces the Symbiote's impact on the host router's control-plane latency while achieving comparable detection latency values as fixed burst-rate Symbiotes.

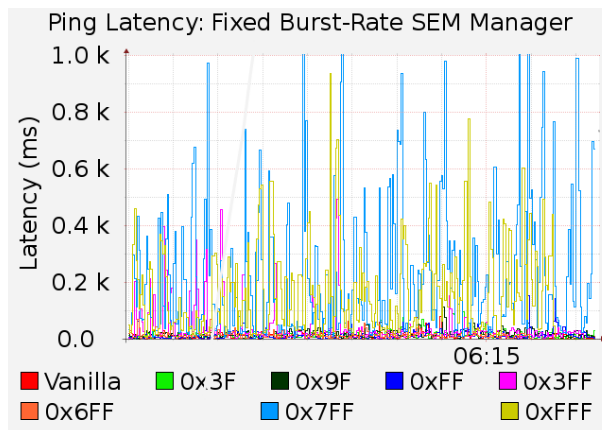

Figure 10.6: Ping Latency : Fixed Burst-Rate SEM Manager

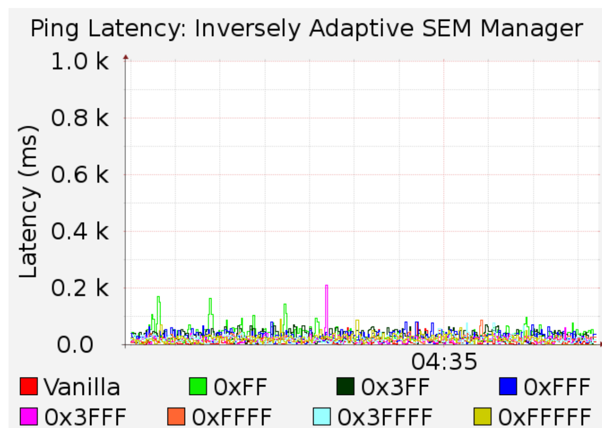

Figure 10.7: Ping Latency : Inverse-Adaptive SEM Manager

10.1.5 Discussion

Preliminary performance results shown in this section suggests that high performance exploitation detection is possible in Cisco IOS. Furthermore, an optimized Symbiote scheduling algorithm can greatly improve performance of the overall sensor system by reducing both detection latency and the Symbiote's impact on the router's control-plane latency. Optimization of the detection latency and the induced control-plane latency is an area of active research.

Part IV

Conclusion

Chapter 11

Future Work

The work presented in this thesis arguably raises more questions than it answers. We present a survey of possible topics for future research in the areas of embedded vulnerability quantification and qualification, the extension of Symbiote capabilities and the further study of Autotomic Binary Reduction, and non-localized binary randomization techniques like Binary Structure Randomization.

11.1 Qualification and Quantification

More large-scale studies are needed to better understand the nature and scope of vulnerabilities in embedded firmware. One such study on a body of firmwares [25] found a total vulnerability rate of 2.14% in a large corpus of diverse firmware binaries while [31] reported a vulnerability rate of over 65% in a different body of widely used firmware images. Vulnerabilities that can plausibly exploitable on both general-purpose computer systems and embedded devices like [20, 95] will likely have significant impact on the vulnerability rates reported by both studies. Thus, large-scale, automated, deep analysis into the content of firmware images can make an important impact on the timely identification and mitigation of embedded vulnerabilities.

11.2 Symbiote

The Symbiote-based sensor presented in this thesis is a first step towards demonstrating the feasibility and novel capability of Symbiotic defense systems. The Symbiote structure allows complex payloads to be injected into legacy embedded devices, allowing the payload to safely execute alongside the original firmware without altering the embedded device's functionality. The firmware integrity attestation payload injected into Cisco IOS can be replaced with a wide range of defensive payloads. Below are several new Symbiote payloads currently under development. Sophisticated Symbiote payloads can alter the behavior of the target machine more aggressively.

11.2.0.1 SEM and Reverse Engineering

Symbiotic Embedded Machines can be used to implement powerful reverse engineering tools for proprietary embedded devices. Using similar inline hook techniques as presented here, Harbour et al recently demonstrated APIThief [53], a sophisticated API tracing tool for the Windows environment. By installing inline hooks into the target code, APIThief can dynamically monitor and alter the behavior of code without succumbing to common anti-reversing techniques designed to defeat traditional debuggers. Similarly, Symbiotic Embedded Machines can be used to control and monitor the behavior of proprietary network embedded devices. As SEM does not require a priori knowledge of the target system's implementation and can take complete control of the hardware on which it resides, it is an ideal platform for deploying reverse engineering tools. For example, SEM can be used to inject a GDB compatible stub onto the target device. This will, in theory, allow remote debugging of many proprietary network embedded devices using the GDB debugger over a standard protocol.

11.2.0.2 SEM and Directed Fuzzing

Symbiotic Embedded Machines can be used to inject monitoring payloads for directed fuzzing systems. For example, Ganesh *et al* have demonstrated the use of dynamic taint tracing to improve the performance of automated fuzzers [113]. Directed fuzzing techniques

generally require realtime visibility into the target software and OS environment and are relatively simple to implement on general purpose computers. Similar techniques are significantly more difficult to implement on network embedded devices due to its closed and proprietary nature. However, since Symbiotic Embedded Machines overcome the challenges of installing complex code onto network embedded devices, it dramatically lowers the difficulty of augmenting existing devices with functionality and visibility required by directed fuzzing systems.

11.2.0.3 SEM and Covert Channels

Symbiotic Embedded Machines and their payloads are designed to maximize generality and portability. Since network embedded devices use a diverse set of I/O hardware, direct network communication must interact with the specific hardware of the target device. Therefore, using standard protocols like IP for communication has at least two drawbacks. First, the SEM payload must contain device specific code to interact with I/O hardware like ethernet controllers, which reduces the generality and portability of the entire system. Second, direct communication channels over the network can be detected, monitored, blocked and intercepted. Communication using covert channels eliminates both drawbacks. Instead of directly communicating with I/O hardware, the SEM can use many methods to alter reliably the state of the device in some measurable way for the purpose of transmitting data. For example, since the SEMM directly controls the CPU usage of the SEM payload, it can influence both the CPU utilization and power consumption of the device. As recently demonstrated by Kiamilev *et al* [69], reliable bi-directional communication can be achieved through the power supply of general purpose computers. Therefore, it is feasible to apply similar techniques to network embedded devices using the SEMM to cause measurable fluctuations of power consumption.

11.2.1 Embedded Self-Healing

The firmware integrity attestation Symbiote payload discussed in Chapter 8 can be extended to **reverse** unauthorized modification of memory after it is detected. A self-healing Symbiote payload can be used to identify and restore regions of memory, which have been

maliciously modified.

11.2.2 Embedded Anomaly Detector

Symbiote payloads can implement existing anomaly detection algorithms. For example, behavior modeling strategies that monitor resource utilization, control, and data flow patterns can be injected into embedded devices via Symbiote payloads.

11.2.3 Large-Scale Embedded Sensor Grid

The exploitation detection sensor described in this chapter can be injected into large numbers of embedded devices like Cisco routers in order to monitor and analyze 0-day exploitation of embedded devices. We believe the use of Symbiote-based exploitation sensors in the wild is a feasible and effective way of monitoring and analyzing exploits levied against the internet substrate. A large-scale Symbiote-based sensor grid can potentially give us real-time visibility into embedded device exploitation on a global scale.

Furthermore, Symbiotes can be used to transform embedded devices into other kinds of sensor grids as well. Symbiotes can allow us to use hardware components of embedded devices in novel ways not intended by its original design. For example, many power-consuming, EM emitting components can be transformed into covert communication channels. Existing sensors on embedded devices, combined with such covert channels can transform a wide gamut of innocuous embedded devices into a web of remotely controlled mobile sensors.

11.3 Autotomy Binary Structure Randomization

The general correctness of language and compiler-level software compaction algorithms have been established. The correctness and safety of binary-level attack-surface reduction algorithms like Autotomic Binary Reduction have only been demonstrated existentially for specific cases. The solvability of the $F_{et}EM$ extraction problem is central to qualifying the scope of applicability of the ABSR algorithm. Further research is necessary to establish the real-world applicability of ABSR in embedded device firmware as well as general-purpose software.

The BSR transform related performance overhead documented in Section 9.5 poses an obvious optimization question. One way to measure the utility of BSR is through binary difference. Each BSR transform will contribute to increasing the binary difference of the resulting binary linearly. However, as we demonstrated, not all BSR transforms will have the same impact on computational overhead of the resultant program. Thus, multi-variable optimization should be considered in order to automate the optimization of both BSR binary difference and overall BSR-induced computational overhead. Since the latter is dependent on the code-coverage probability distribution function of the program, which varies with input data, program profiling, symbolic execution and dynamic analysis can be applied to this problem to make BSR more efficient and effective.

Chapter 12

Concluding Remarks

12.1 Conclusion

We began our research into the security of embedded devices by asking several simple questions. Is it possible, in our increasingly connected and automated world, that software vulnerabilities in black-box embedded devices can be used to cause disruption and destruction in the physical world? Can embedded systems be exploited? If so, do vulnerable devices exist in significant quantity or perform significantly critical functions, such that the exploitation of such devices should be considered a real threat? Can we trust that the embedded devices we depend on today have not already been compromised? Can we reliably detect the consequence of exploitation after the fact? Lastly, what can be done to improve the security of embedded systems, given their unique and constrained nature?

To address the question of embedded defense, we presented the following hypothesis:

Embedded devices are vulnerable to large-scale exploitation. The use of software defensive techniques that take into consideration the hardware and software constraints imposed by such systems can provide effective and efficient detection of and defense against the exploitation of several classes of software vulnerabilities, as well as the injection of persistent software implants in legacy embedded devices. Such software-based defensive techniques can be automatically realized by making modifications to the firmware that do not alter the original functionality of the firmware, but which introduce various security capabilities to the embedded device at a cost of an acceptable level of resource overhead. Most importantly, such

software-based defenses should be realizable at the binary level, without requiring access or modification to source-code, and should not require any hardware modification.

In this thesis, we presented a body of scientific study, which made significant contributions towards answering these questions and established the validity of our hypothesis. In short, embedded systems can be, and have been, exploited. They exist in vast numbers and perform critical functions in the world. The exploitation of embedded systems should be considered a real and present threat. Most importantly, we have proposed several techniques in this thesis that can be applied to real-world embedded systems to make them more secure against cyber attack. These techniques were applied to a range of real-world devices like network routers, network-based printers and IP phones and have been shown to be safe to use and effective against several types of common attacks.

In order to better understand the nature of embedded security, we first presented quantitative and qualitative evidence of the existence of vulnerable embedded devices in the world. Specifically, a internet-wide scan was carried out in order to study the make-up and measure the lower-bound on the quantity of vulnerable embedded devices accessible over the public internet. We discovered that approximately 20% of all embedded devices on the internet was configured with a well-known default root credential, making them trivially vulnerable to attack. Furthermore, we predicted, through quantitative measurements and qualitative analysis, the size of an embedded device botnet would most likely be around 540,000 devices. This prediction was validated two years after the initial publication of our work by the public announcement of the Carna botnet ([5]), which compromised over 480,000 devices.

Next, we presented a series of case-studies of exploitation against ubiquitous embedded devices to gain greater insight into the nature of common embedded vulnerabilities. These case-studies resulted in the public disclosures 4 vulnerabilities that affected millions of devices in the world; CVE-2012-5445, CVE-2013-6685, ASA-2014-099, CVE-2011-4161. While the exact vulnerabilities varied from device to device, several common traits can be extracted about their nature. These vulnerabilities have existed for many years, the complexity of the vulnerabilities are low, and successful exploitation can be done using offensive techniques that can be considered obsolete by modern exploitation standards.

Lastly, we proposed and discussed two software-based defensive techniques, Software Symbiote and Autotomic Binary Structure Randomization, that are aimed to improve the security posture of embedded devices. We demonstrated the efficacy of these two defensive techniques by applying them to real-world embedded devices that are known to be vulnerable. We presented experimental data confirming the safety of our prosed defenses as well as their efficacy against real-world exploitation.

In conclusion, so long, and thanks for all the fish.

Part V

Bibliography

Bibliography

[1] Injunction Against Michael Lynn. http://www.infowarrior.org/users/rforno/lynn-cisco.pdf.

[2] New worm can infect home modem/routers. APCMAG.com, 2009. http://apcmag.com/Content.aspx?id=3687.

[3] Psyb0t' worm infects linksys, netgear home routers, modems. ZDNET, 2009. http://blogs.zdnet.com/BTL/?p=15197.

[4] Evolution in attacks against cisco ios software platforms. Cisco PSIRT Alert 40411, 2015. http://tools.cisco.com/security/center/viewAlert.x?alertId=40411.

[5] Anonymous. Internet Census 2012 port scanning /0 using insecure embedded devices, 2012.

[6] I. Arce. The Rise of the Gadgets. *IEEE Security and Privacy*, 1(5):78–81, 2003.

[7] Arduino. `http://arduino.cc/`.

[8] Adam J. Aviv, Pavol Cerný, Sandy Clark, Eric Cronin, Gaurav Shah, Micah Sherr, and Matt Blaze. Security evaluation of es&s voting machines and election management system. In David L. Dill and Tadayoshi Kohno, editors, *EVT*. USENIX Association, 2008.

[9] A. Bellissimo, J. Burgess, and K. Fu. Secure Software Updates: Disappointments and New Challenges. *Proceedings of USENIX Hot Topics in Security (HotSec)*, 2006.

[10] James T. Bennet and J. Gomez. The Shellshock Aftershock for NAS Administrators, 2014.

[11] Vijay Bollapragada, Curtis Murphy, and Russ White. Inside cisco ios software architecture. Cisco Press, 2000. Demonstration of Hardware Trojans.

[12] Derek Bruening, Qin Zhao, and Saman P. Amarasinghe. Transparent dynamic instrumentation. In Steven Hand and Dilma Da Silva, editors, *VEE*, pages 133–144. ACM, 2012.

[13] Bushing, Marcan, Segher, and Sven. Console hacking 2010, 2010.

[14] Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio Soriente. On the difficulty of software-based attestation of embedded devices. In Ehab Al-Shaer, Somesh Jha, and Angelos D. Keromytis, editors, *ACM Conference on Computer and Communications Security*, pages 400–409. ACM, 2009.

[15] CERT. CERT Advisory CA-2002-07 Double Free Bug in zlib Compression Library. `http://www.cert.org/advisories/CA-2002-07.html`, 2002.

[16] CERT. The zlib compression library is vulnerable to a denial-of-service condition. `http://www.kb.cert.org/vuls/id/238678`, 2004.

[17] CERT. zlib inflate() routine vulnerable to buffer overflow. `http://www.kb.cert.org/vuls/id/680620`, 2005.

[18] CERT. CVE-2011-4161. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-4161`, 2011.

[19] Hoi Chang and Mikhail J. Atallah. Protecting software code by guards. In *Digital Rights Management Workshop*, pages 160–175, 2001.

[20] Stephane Chazelas. CVE-2014-6271 shell shock vulnerability, 2014.

[21] K. Chen. Reversing and exploiting an apple firmware update, 2009.

[22] K. Chen. Reversing and Exploiting an Apple Firmware Update. In *Black Hat USA*, 2009.

[23] Shane S. Clark and Kevin Fu. Recent results in computer security for medical devices. In *International ICST Conference on Wireless Mobile Communication and Healthcare (MobiHealth), Special Session on Advances in Wireless Implanted Devices*, October 2011.

[24] Andrei Costin. Hacking MFPs. In *The 28th Chaos Communication Congress*, 2011.

[25] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A large-scale analysis of the security of embedded firmwares. In Kevin Fu and Jaeyeon Jung, editors, *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 95–110. USENIX Association, 2014.

[26] BarbaraF. Csima and Bakhadyr Khoussainov. When is reachability intrinsically decidable? In Masami Ito and Masafumi Toyama, editors, *Developments in Language Theory*, volume 5257 of *Lecture Notes in Computer Science*, pages 216–227. Springer Berlin Heidelberg, 2008.

[27] Ang Cui.

[28] Ang Cui.

[29] Ang Cui.

[30] Ang Cui. http://www.hacktory.cs.columbia.edu/ios-rootkit.

[31] Ang Cui. Print me if you dare: Firmware modification attacks and the rise of printer malware, 2011.

[32] Ang Cui. Print Me If You Dare: Firmware Modification Attacks and the Rise of Printer Malware. In *The 28th Chaos Communication Congress*, 2011.

[33] Ang Cui. Embedded Device Firmware Vulnerability Hunting Using FRAK. In *Black Hat USA*, 2012.

[34] Ang Cui, Jatin Kataria, and Salvatore J. Stolfo. Killing the myth of cisco ios diversity: Recent advances in reliable shellcode design. In David Brumley and Michal Zalewski, editors, *WOOT*, pages 19–27. USENIX Association, 2011.

[35] Ang Cui, Jatin Kataria, and Salvatore J. Stolfo. Killing the Myth of Cisco IOS Diversity: Recent Advances in Reliable Shellcode Design. In *Proceedings of the 5th USENIX conference on Offensive technologies*, pages 3–3. USENIX Association, 2011.

[36] Ang Cui and Salvatore J. Stolfo. A Quantitative Analysis of the Insecurity of Embedded Network Devices: Results of a Wide-Area Scan. In Carrie Gates, Michael Franz, and John P. McDermott, editors, *ACSAC*, pages 97–106. ACM, 2010.

[37] Ang Cui and Salvatore J. Stolfo. A quantitative analysis of the insecurity of embedded network devices: Results of a wide-area scan. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 97–106, New York, NY, USA, 2010. ACM.

[38] Andy Davis. Cisco ios ftp server remote exploit. In *http://www.securityfocus.com/archive/1/494868*, 2007.

[39] Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. Compiler techniques for code compaction. *ACM Trans. Program. Lang. Syst.*, 22(2):378–415, March 2000.

[40] Guillaume Delugre. Closer to metal: Reverse engineering the Broadcom NetExtremes firmware, 2010. HACK.LU.

[41] Tamara Denning, Kevin Fu, and Tadayoshi Kohno. Absence makes the heart grow fonder: New directions for implantable medical device security. In Niels Provos, editor, *HotSec*. USENIX Association, 2008.

[42] Dronebl. Network Bluepill. `http://www.dronebl.org/blog/8`, 2008.

[43] Loïc Duflot, Yves-Alexis Perez, and Benjamin Morin. What if you can't trust your network card? In Robin Sommer, Davide Balzarotti, and Gregor Maier, editors, *RAID*, volume 6961 of *Lecture Notes in Computer Science*, pages 378–397. Springer, 2011.

[44] DynamoRIO. Dynamic Instrumentation Tool Platform. `http://dynamorio.org/`.

[45] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. Xfi: Software guards for system address spaces. In *OSDI*, pages 75–88. USENIX Association, 2006.

[46] Eleazar Eskin, Wenke Lee, and Salvatore J Stolfo. Modeling system calls for intrusion detection with dynamic window sizes. In *DARPA Information Survivability Conference & Exposition II, 2001. DISCEX'01. Proceedings*, volume 1, pages 165–175. IEEE, 2001.

[47] Federico Fazzi. LightAidra, 2012.

[48] Peter Feiner, Angela Demke Brown, and Ashvin Goel. Comprehensive kernel instrumentation via dynamic binary translation. In Tim Harris and Michael L. Scott, editors, *ASPLOS*, pages 135–146. ACM, 2012.

[49] Michael Franz. E unibus pluram: Massive-scale software diversity as a defense mechanism. In *Proceedings of the 2010 Workshop on New Security Paradigms*, NSPW '10, pages 7–16, New York, NY, USA, 2010. ACM.

[50] Sergey Gordeychik. Scada strange love, 2011.

[51] S. Hanna, R. Rolles, A. Molina-Markham, P. Poosankam, K. Fu, and D. Song. Take Two Software Updates and See Me in the Morning: The Case for Software Security Evaluations of Medical Devices. In *Proceedings of the 2nd USENIX conference on Health security and privacy*, page 6. USENIX Association, 2011.

[52] Steve Hanna, Rolf Rolles, Andres Molina-Markham, Pongsin Poosankam, Kevin Fu, and Dawn Song. Take two software updates and see me in the morning: The case for software security evaluations of medical devices. In *Proceedings of 2nd USENIX Workshop on Health Security and Privacy (HealthSec)*, August 2011.

[53] Nick Harbour. Win at Reversing: API Tracing and Sandboxing Through Inline Hooking, 2009. Black Hat USA.

[54] Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. Librando: transparent code randomization for just-in-time compilers. In *Proceedings of the 2013 ACM*

SIGSAC conference on Computer & communications security, CCS '13, pages 993–1004, New York, NY, USA, 2013. ACM.

[55] HP. Hewlett-Packard LaserJet 4200/4300 Series Printers - Firmware Update/Download Release/Installation Notes. `http://ftp.hp.com/pub/printers/software/lj4200lbreadmefw.txt`.

[56] HP. HP Security Solutions FAQ. `http://h30046.www3.hp.com/large/solutions/hp_secsolutions.pdf`, 2006.

[57] HP. SSRT100692 rev.1- Certain HP Printers and HP Digital Senders, Remote Firmware Update Enabled by Default. `http://seclists.org/bugtraq/2011/Dec/3`, 2011.

[58] HP. SSRT100692 rev.2 - Certain HP Printers and HP Digital Senders, Remote Firmware Update Enabled by Default. `http://seclists.org/bugtraq/2011/Dec/175`, 2011.

[59] HP. SSRT100692 rev.3 - Certain HP Printers and HP Digital Senders, Remote Firmware Update Enabled by Default. `http://seclists.org/bugtraq/2012/Jan/49`, 2012.

[60] HP. SSRT100692 rev.6 - Certain HP Printers and HP Digital Senders, Remote Firmware Update Enabled by Default. `http://h20000.www2.hp.com/bizsupport/TechSupport/Document.jsp?objectID=c03102449`, 2012.

[61] HP WebJet Admin. `http://tinyurl.com/ch3g72f`.

[62] IDA. The IDA Pro Disassembler and Debugger. `http://www.hex-rays.com/idapro`.

[63] IDC. Worldwide Hardcopy Peripherals Market Recorded Double-Digit Year-Over-Year Growth in the Second Quarter of 2010, According to IDC. `http://www.idc.com/about/viewpressrelease.jsp?containerId=prUS22476810§ionId=null&elementId=null&pageType=SYNOPSIS`, 2010.

[64] Barnaby Jack. Jackpotting Automated Teller Machines Redux. In *Black Hat USA*, 2010.

[65] Barnaby Jack. IMPLANTABLE MEDICAL DEVICES: HACKING HUMANS, 2013. Blackhat USA.

[66] Todd Jackson, Andrei Homescu, Stephen Crane, Per Larsen, Stefan Brunthaler, and Michael Franz. Diversifying the software stack using randomized nop insertion. In Sushil Jajodia, Anup K. Ghosh, V.S. Subrahmanian, Vipin Swarup, Cliff Wang, and X. Sean Wang, editors, *Moving Target Defense II*, volume 100 of *Advances in Information Security*, pages 151–173. Springer New York, 2013.

[67] Nick L. Petroni Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symposium*, pages 179–194. USENIX, 2004.

[68] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 272–280, New York, NY, USA, 2003. ACM.

[69] Fouad Kiamilev and Ryan Hoover. Defcon 16, 2008. Demonstration of Hardware Trojans.

[70] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In Dan Boneh, editor, *USENIX Security Symposium*, pages 191–206. USENIX, 2002.

[71] Christopher Krügel, William K. Robertson, and Giovanni Vigna. Detecting Kernel-Level Rootkits Through Binary Analysis. In *ACSAC*, pages 91–100. IEEE Computer Society, 2004.

[72] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. Sok: Automated software diversity. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 276–291, May 2014.

[73] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, January 1979.

[74] Yanlin Li, Jonathan M. McCune, and Adrian Perrig. SBAP: Software-Based Attestation for Peripherals. In Alessandro Acquisti, Sean W. Smith, and Ahmad-Reza Sadeghi, editors, *Trust and Trustworthy Computing, Third International Conference, TRUST 2010, Berlin, Germany, June 21-23, 2010. Proceedings*, volume 6101 of *Lecture Notes in Computer Science*, pages 16–29. Springer, 2010.

[75] Yanlin Li, Jonathan M. McCune, and Adrian Perrig. Sbap: Software-based attestation for peripherals. In Alessandro Acquisti, Sean W. Smith, and Ahmad-Reza Sadeghi, editors, *TRUST*, volume 6101 of *Lecture Notes in Computer Science*, pages 16–29. Springer, 2010.

[76] Yanlin Li, Jonathan M. McCune, and Adrian Perrig. Viper: verifying the integrity of peripherals' firmware. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM Conference on Computer and Communications Security*, pages 3–16. ACM, 2011.

[77] Felix Lindner. Burning the bridge: Cisco ios exploits, 2001.

[78] Felix Lindner. Cisco Vulnerabilities. In *BlackHat USA*, 2003.

[79] Felix Lindner. Design Issues and Software Vulnerabilities in Embedded Systems. In *Black Hat Windows Security*, 2003.

[80] Felix Lindner. Cisco IOS Router Exploitation. In *Black Hat USA*, 2009.

[81] Richard Lippmann, Engin Kirda, and Ari Trachtenberg, editors. *Recent Advances in Intrusion Detection, 11th International Symposium, RAID 2008, Cambridge, MA, USA, September 15-17, 2008. Proceedings*, volume 5230 of *Lecture Notes in Computer Science*. Springer, 2008.

[82] Edward S. Lowry and C. W. Medlock. Object code optimization. *Commun. ACM*, 12(1):13–22, January 1969.

[83] PratyusaK. Manadhata. Game theoretic approaches to attack surface shifting. In Sushil Jajodia, Anup K. Ghosh, V.S. Subrahmanian, Vipin Swarup, Cliff Wang, and

X. Sean Wang, editors, *Moving Target Defense II*, volume 100 of *Advances in Information Security*, pages 1–13. Springer New York, 2013.

[84] Andrea M Matwyshyn, Ang Cui, Angelos D Keromytis, and Salvatore J Stolfo. Ethics in security vulnerability research. *Security & Privacy, IEEE*, 8(2):67–72, 2010.

[85] Michael Lynn. Cisco IOS Shellcode, 2005. In BlackHat USA.

[86] Charile Miller. Battery Firmware Hacking. In *Black Hat USA*, 2011.

[87] Sebastian Muniz. Killing the myth of Cisco IOS rootkits: DIK, 2008. In EUSecWest.

[88] Sebastian Muniz. Killing the myth of Cisco IOS rootkits: DIK, 2008. In EUSecWest.

[89] Sebastian Muniz and Alfredo Ortega. Fuzzing and Debugging Cisco IOS, 2011. Black Hat Europe.

[90] Teague Newman, Tiffany Rad, and John Strauchs. Scada & plc vulnerabilities in correctional facilities white paper, 2011.

[91] Offensive Security. Virata EmWeb R6.0.1 Remote Crash Vulnerability. `http://www.exploit-db.com/exploits/12095/`, 2010.

[92] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 601–615, Washington, DC, USA, 2012. IEEE Computer Society.

[93] pt. Ooops I hacked My PBX. In *The 28th Chaos Communication Congress*, 2011.

[94] Thomas Reps. Undecidability of context-sensitive data-dependence analysis. *ACM Trans. Program. Lang. Syst.*, 22(1):162–186, January 2000.

[95] Riku, Antti, and Matti. CVE-2014-0160 openssl security bug - heartbleed, 2013.

[96] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. In Lippmann et al. [81], pages 1–20.

[97] Martin C. Rinard. Manipulating Program Functionality to Eliminate Security Vulnerabilities. In Sushil Jajodia, Anup K. Ghosh, Vipin Swarup, Cliff Wang, and Xiaoyang Sean Wang, editors, *Moving Target Defense*, volume 54 of *Advances in Information Security*, pages 109–115. Springer, 2011.

[98] SecurityFocus. Sec. Vulenrability in ChaiVM EZloader. `http://www.securityfocus.com/advisories/4317`, 2002.

[99] Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert van Doom, and Pradeep K. Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In Mihai Christodorescu, Somesh Jha, Douglas Maughan, Dawn Song, and Cliff Wang, editors, *Malware Detection*, volume 27 of *Advances in Information Security*, pages 253–289. Springer, 2007.

[100] Arvind Seshadri, Adrian Perrig, and Leendert Doorn. Using software-based attestation for verifying embedded systems in cars, 2004.

[101] Arvind Seshadri, Adrian Perrig, Leendert van Doorn, and Pradeep K. Khosla. Swatt: Software-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy*, pages 272–. IEEE Computer Society, 2004.

[102] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.

[103] Stelios Sidiroglou, Oren Laadan, Carlos Perez, Nicolas Viennot, Jason Nieh, and Angelos D. Keromytis. Assure: Automatic software self-healing using rescue points. *SIGPLAN Not.*, 44(3):37–48, March 2009.

[104] Spansion. SPANSION S25FL064P Data Sheet. `http://www.spansion.com/Support/Datasheets/S25FL064P.pdf`, 2011.

[105] Salvatore J. Stolfo, Issac Greenbaum, and Simha Sethumadhavan. Self-monitoring

monitors. Technical Report cucs-026-09, Columbia University Computer Science Department, April 2009.

[106] Frederic Stumpf, Omid Tafreschi, Patrick Rder, and Claudia Eckert. A robust integrity reporting protocol for remote attestation. In *Second Workshop on Advances in Trusted Computing (WATC '06 Fall)*, Tokyo, Japan, November 2006.

[107] Michael Sutton. Corporate Espionage for Dummies: The Hidden Threat of Embedded Web Servers. In *Black Hat USA*, 2011.

[108] Robert Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1972.

[109] Arrigo Triulzi. The jedi packet trick takes over the deathstar, 2010.

[110] Varun Uppal. Cisco IOS Bind shellcode v1.0. In *http://www.exploit-db.com/exploits/13292/*, 2007.

[111] Chris Valasek and Charlie Miller. Adventures in automotive networks andcontrolunits. IOActive White Paper, 2013.

[112] Vikas R. Vasisht and Hsien-Hsin S. Lee. Shark: Architectural support for autonomic protection against stealth by rootkit exploits. In *MICRO*, pages 106–116. IEEE Computer Society, 2008.

[113] Martin Rinard Vijay Ganesh, Tim Leek. Taint-based directed whitebox fuzzing. IEEE 31st International Conference on Software Engineering, 2009.

[114] VxWorks socklib. `http://www-kryo.desy.de/documents/vxWorks/V5.5/vxworks/ref/sockLib.html`.

[115] Redmond Wa, Galen Hunt, Galen Hunt, Doug Brubacher, and Doug Brubacher. Detours: Binary interception of win32 functions. In *In Proceedings of the 3rd USENIX Windows NT Symposium*, pages 135–143, 1998.

[116] Jiang Wang, Angelos Stavrou, and Anup K. Ghosh. Hypercheck: A hardware-assisted integrity monitor. In Somesh Jha, Robin Sommer, and Christian Kreibich, editors,

RAID, volume 6307 of *Lecture Notes in Computer Science*, pages 158–177. Springer, 2010.

[117] Zhi Wang, Xuxian Jiang, Weidong Cui, and Xinyuan Wang. Countering Persistent Kernel Rootkits Through Systematic Hook Discovery. In Lippmann et al. [81], pages 21–38.

[118] Ralf Philipp Weinmann. All your basebands are belong to us, 2010.

[119] Man-Ki Yoon, Sibin Mohan, Jaesik Choi, Mihai Christodorescu, and Lui Sha. Intrusion Detection Using Execution Contexts Learned from System Call Distributions of Real-Time Embedded Systems. *arXiv*, 2015.

[120] Jonas Zaddach, Anil Kurmus, Davide Balzarotti, Erik-Oliver Blass, Aurélien Francillon, Travis Goodspeed, Moitrayee Gupta, and Ioannis Koltsidas. Implementation and implications of a stealth hard-drive backdoor. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, pages 279–288, New York, NY, USA, 2013. ACM.

Part VI

Appendices

Appendix A

Appendix A

A.1 Appendix: List of Embedded Device Profiles Supported by Default Credential Scanner

This section of the appendix details data specific to the work presented in Section 6.

Camera/Survellience			
D-Link Web Cams	Linksys Web Cams	Axis Network Cameras	AVTech Web Cams
Enterprise Networking			
Huawei SmartAX	Huawei NE40E	D-Link DES Switches	NetSys NV-200
Huawei IAD	ZXA10 C220	NetScreen-5GT	D-Link DHS-3224
Huawei Routers	ALCATEL SR 7750	Huawei A8010	H3C Network Devices
Huawei Quidway AR19 Series	CheckPoint Firewall	Cisco Network Devices	
TopSec Network Devices	Ubiquoss Routers and Switches		
Home Networking			
Siemens DSL Modems	Netcomm BCM96348	Huawei Homegateway	D-Link Access Points
D-Link DSL, DIR, DGL, DWA Series	Linksys WRT	HandLink HotSpot	
Belkin Routers	NetGear 3G MBR624GU	MicroTik WebBox	
Vendor Issued Equipment			
Netwave MNG-5000	D-Link DSL 500G	D-Link DSL 500B	D-Link DSL 524B
Arris Modem TM402P	Globespan Virata GS8100	ADTRAN Total Access 604	Zyxel 645
ZXR10 T1200, T600	ZTE ZXDSL ADSL Modem	SpeedStream 5100, 5200, 5400, 5500 Series	Ambit U10C022
WaveCast MW-2010R	Ubiquoss R3004A	WaveCast MW-1700AP, MW-1200AP	SKBroadband SI314T
Video Conferencing Devices			
Kedacom KDV8000C , TS3610	Hikvision Net Video Server	Crestron MP2E	Tandberg VC Units
Polycom VC Units			
Office Appliances			
HP JetDirect Print Server	D-Link Print Server	Lantronix MSS, MPS Series	D-Link DES, DGS, DHS Series
VoIP Appliances			
Linksys PAP2	Linksys SPA	GaoKe MG6000	GrandStream GXP Series
Sipura SPA			
Home Brew		**Power Management**	
DD-WRT	BusyBox	Sentry PDU	APC PDU

Appendix B

Appendix B

B.1 Cisco IOS Rookit

This section of the appendix details various details of the Disassembling and Interrupt-Hijacking Cisco IOS rootkit described in Section 7.1. Source code is available to reputable researchers upon formal request.

B.1.1 Disassembling Shellcode

Target Platform	Tested IOS versions	Size
All MIPS	(12.0 - 12.4)	200 bytes

Table B.1: MIPS-based disassembling rootkit statistics.

B.1.2 Interrupt Hijacking Shellcode

Target Platform	Tested IOS versions	Size
All MIPS	(12.0 - 12.4)	420 bytes

Table B.2: MIPS-based interrupt hijack rootkit statistics.

Appendix C

Appendix C

C.1 CVE-2011-4161: HP LaserJet Firmware Modification Vulnerability

This section of the appendix presents data associated with work presented in Section 7.2.1.


```
000000  40 50 4A 4C 20 43 4F 4D 4D 45 4E 54 20 4D 4F 44 45 4C 3D 48  @PJL COMMENT MODEL=H
000014  50 20 4C 61 73 65 72 4A 65 74 20 50 32 30 35 35 64 6E 0A 40  P LaserJet P2055dn@
000028  50 4A 4C 20 43 4F 4D 4D 45 4E 54 20 56 45 52 53 49 4F 4E 3D  PJL COMMENT VERSION=
00003C  38 33 35 30 34 0A 40 50 4A 4C 20 43 4F 4D 4D 45 4E 54 20 44  83504@PJL COMMENT D
000050  41 54 45 43 4F 44 45 3D 32 30 31 30 30 33 30 38 0A 40 50 4A  ATECODE=20100308@PJ
000064  4C 20 55 50 47 52 41 44 45 20 53 49 5A 45 3D 37 39 32 39 39  L UPGRADE SIZE=79299
000078  30 36 0A 1B 25 2D 31 32 33 34 35 58 40 50 4A 4C 20 45 4E 54  06%-12345X@PJL ENT
00008C  45 52 20 4C 41 4E 47 55 41 47 45 3D 41 43 4C 0D 0A 00 AC 00
0000A0  0F 00 03 62 2D 00 00 00 00 00 79 00 00 AA 55 41 54 00 00 01
0000B4  20 00 67 FB E9 00 E2 17 03 00 00 00 00 00 67 FD 09 00 00 20
0000C8  E0 00 00 4D 3C 00 68 1D E9 00 00 21 86 00 00 50 91 00 68 3F
0000DC  6F 00 00 20 28 00 00 4D AA 00 68 5F 97 00 00 20 BC 00 00 50
0000F0  0C 00 68 80 53 00 00 20 CB 00 00 4C C4 00 68 A1 1E 00 00 20
000104  83 00 00 4D BF 00 68 C1 A1 00 00 20 23 00 00 4B 2A 00 68 E1
000118  C4 00 00 1F E1 00 00 4B D8 00 69 01 A5 00 00 20 84 00 00 4D
00012C  5A 00 69 22 29 00 00 21 1D 00 00 4E 12 00 69 43 46 00 00 21
000140  42 00 00 50 24 00 69 64 88 00 00 24 0D 00 00 54 2D 00 69 88
000154  95 00 00 24 35 00 00 54 C1 00 69 AC CA 00 00 23 84 00 00 50
000168  E7 00 69 D0 4E 00 00 28 24 00 00 7A 8E 00 69 F8 72 00 00 22
00017C  CD 00 00 50 D6 00 6A 1B 3F 00 00 21 3E 00 00 52 CF 00 6A 3C
000190  7D 00 00 1F F3 00 00 4B C0 00 6A 5C 70 00 00 22 11 00 00 51
```


Figure C.1: Hex dump of a typical HP-RFU. For P2055DN, using the undocumented PJL/ACL language.

Figure C.2: "UAT" table structure. Contains a checksum value, followed by a directory manifest describing various compressed components of the binary update package.

Figure C.3: RFU binary embedded inside a typical PostScript file. This illustrates the most straightforward reflexive attack.

Model	RFUs (qty.)	Earliest RFU	Latest RFU
2300	2	2004-05-12	2004-12-03
2400	4	2004-09-02	2009-06-24
3000	2	2004-01-06	2008-04-09
3500	3	2004-01-19	2007-02-20
3550	2	2004-09-22	2005-03-07
3600	2	2006-08-07	2006-08-28
3700	3	2004-03-31	2006-12-06
3800	1	2008-04-08	2008-04-08
4100	2	2004-10-08	2005-12-21
4200	2	2004-10-07	2005-06-02
4250	9	2004-09-02	2011-04-06
4300	2	2004-10-07	2005-06-02
4345	10	2005-01-25	2011-04-29
4600	2	2004-10-12	2006-10-10
4650	3	2004-08-27	2007-04-19
4700	7	2009-06-05	2011-05-11
4730	8	2009-06-04	2011-04-29
5100	1	2004-01-15	2004-01-15
5200	9	2009-06-04	2011-12-14
5500	3	2004-10-07	2005-06-02
5550	10	2004-07-29	2011-04-06
8000	1	2010-10-28	2010-10-28
8150	2	2004-01-14	2004-10-14
8500	6	2010-10-25	2011-06-29
9000	3	2004-08-09	2005-12-21
9050	21	2004-06-30	2011-12-13
9055	1	2008-02-20	2008-02-20
9065	5	2004-09-10	2008-02-20
9200	8	2005-01-25	2011-04-19
9250	10	2009-06-04	2011-12-19
9500	12	2004-10-24	2011-05-24
CM1312	5	2010-06-16	2011-12-09
CM1415	6	2010-07-21	2011-12-15
CM3530	9	2009-06-04	2011-12-13
CM4730	11	2009-06-04	2011-12-12
CM6040	8	2009-09-09	2011-12-12
CM80	5	2008-10-28	2010-08-05
CP1518	3	2010-06-16	2011-12-10
CP1525	6	2010-07-21	2011-12-15
CP2024	3	2010-05-12	2011-12-08
CP3505	8	2009-06-04	2011-04-06
CP3525	10	2008-12-04	2011-12-12
CP4005	6	2009-06-05	2011-05-11
CP4525	7	2010-01-20	2011-12-13
CP5225	5	2011-12-20	2011-12-20
CP6015	9	2009-06-04	2011-12-12
M1522	2	2011-03-19	2011-12-12
M1536	5	2010-07-21	2011-12-15
M2727	3	2010-09-02	2011-12-12
M3035	17	2009-06-05	2011-12-12
M4345	15	2009-06-05	2011-12-12
M5035	15	2009-06-05	2011-12-12
M9050	9	2009-06-05	2011-12-12
P2035	4	2011-03-30	2011-12-13
P2055	9	2009-04-30	2011-12-14
P3005	9	2009-06-15	2011-04-06
P3015	8	2009-09-10	2011-12-13
P4015	10	2009-06-04	2011-12-14
Pro 100	1	2011-10-21	2011-10-21
T1200	2	2010-08-31	2011-07-06
T2300	2	2010-08-31	2010-10-28
T7100	4	2011-09-06	2011-11-05
Z6200	1	2011-11-05	2011-11-05

Table C.1: Printer models and firmware images analyzed for vulnerable libraries.

Appendix D

Appendix D

D.1 Symbiote Performance

This section of the appendix presents data associated with the work presented in Chapter 8.

We present the performance metrics of the rootkit detection SEM payload as measured on a physical Cisco 7271 router running IOS versions 12.2 and 12.3].

Figure D.1: CPU Utilization on Cisco 7121 Router Using Different SEM Payload Execution Bursts Rates ($g(\alpha_i, \tau_q)$) for IOS 12.2 and 12.3. Note the Direct Relationship Between $g(\alpha_i, \tau_q)$, SEM Payload Execution Time and Total CPU Utilization. Terms Low, Med, High, and Really High Utilization Corresponds to Varying SEM Payload Burst Rates, $g(\alpha_i, \tau_q)$.

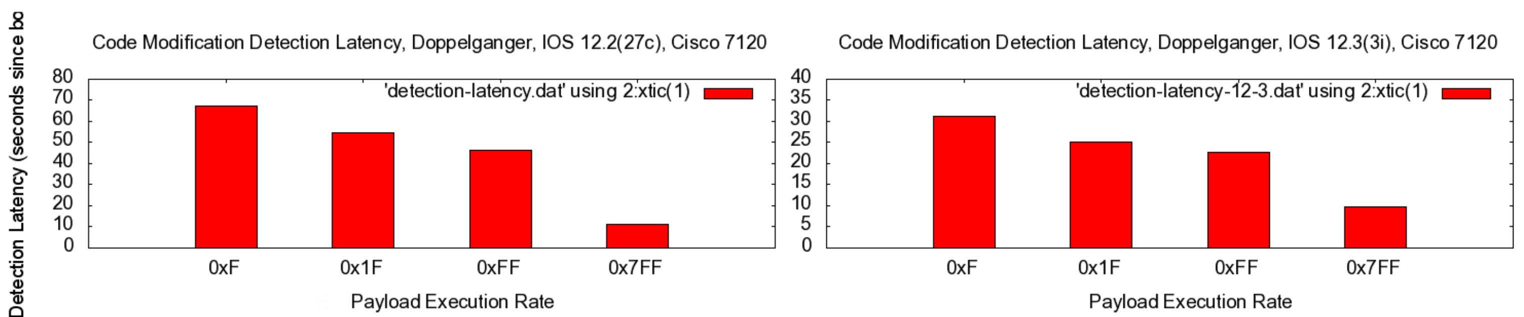

Figure D.2: Inverse Relationship between SEM Payload Burst Rate ($g(\alpha_i, \tau_q)$) and Detection Latency.