Context-Aware Malicious Code Detection

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Boxuan Gu, B.E., M.S.

Graduate Program in Computer Science And Engineering

The Ohio State University

2012

Dissertation Committee:

Dong Xuan, Advisor

Ten H. Lai

Feng Qin

Abstract

Malicious codes are one of the biggest threats on the Internet according to the US-CERT vulnerability database [109]. One salient example is Conficker, a malicious code targeting MS Windows that was released in 2009. Before it was discovered, millions of computers on the Internet were infected [60]. Many approaches to malicious code detection have been proposed. However, such approaches have a key weakness: they do not leverage context information from target systems and input data in order to perform detection. Malicious codes can fully utilize context information for attack purposes, thereby evading detection. To address this issue, we propose a methodology that leverages such context information for malicious code detection. Based on this methodology, we design and implement three detection systems for malicious code detection on servers, Web browsers, and smartphones. Our first system takes "snapshots" of a target process's virtual memory space and leverages these snapshots to reveal malicious codes' true behaviors when consuming input data. Based on the first system, we construct the second system, which leverages Web browsers' JavaScript code execution environment to detect malicious JavaScript codes that exploit browsers' memory errors. Our third system uses an information flow tracking mechanism to detect malicious codes that steal sensitive information stored in smartphones. We comprehensively evaluate these detection systems with many real-world

malicious codes. Our experimental results indicate that the context information can be used to greatly improve detection effectiveness with reasonable overhead.

This is dedicated to my family, my teachers and my friends.

Acknowledgments

First and foremost, I thank my advisor, Prof. Dong Xuan. Without his support, I doubt I could finish my Ph.D study. This dissertation would not be possible without his advice and help. He brought me to his research group when I was unsure about my area of study. Prof. Xuan introduced me to the area of system security and he directed me in finding research problems in this area. Over the years, Prof. Xuan has taught me a great deal regarding research, including how to find and formalize problems, think at different levels of abstraction, discover solutions, present ideas, and so on. All these skills helped my Ph.D study as well as my future work.

I thank other professors in the Department of Computer Science and Engineering. I learned much about systems research from Prof. Xiaodong Zhang, especially research significance. Prof. Ten H. Lai taught me much fundamental knowledge related to security theory. He taught me how to prove that a system is secure. Prof. Kenneth J. Supowit taught me much theoretical knowledge of computer science. Many thanks to Prof. Feng Qin, an adept scholar in the area of programming systems. I was fortunate to work with Prof. Qin for five years. His insightful views and knowledge in the security of programming systems always helped me overcome difficulties in my malicious code detection research. I thank Prof. Douglass Schumacher for serving on my Ph.D. defense committee. I greatly appreciate his evaluation of my research work.

I thank my colleagues in Prof. Xuan's and Prof. Qin's research groups. I am very lucky to have known them and had them in my life. They are my brothers and sisters. They gave me much help ranging from study to daily life. I hope we can work together again in the future.

I thank Xiaole Bai for helping me organize my ideas and present them. I thank Adam Champion for helping me in paper writing. I thank Zhimin Yang, Wenbin Zhang, Xinfeng Li, and Gang Li for helping me implement systems for detecting malicious codes in server computers, desktop computers, and smartphones. I thank Zhezhe Chen for his help in detection system experiments. In addition, Boying Zhang and Jin Teng always provided me many useful comments on my papers.

Vita

1996	B.E. Computer Science and Engineering, Southeast University, China
1999	M.E. Computer Science and Engineering, Southeast University, China
2005	M.S. Computer Science, University Of Kentucky, USA
2005-2011	Graduate Research/Teaching Associate, The Ohio State University, USA
2011-present	Software Engineer, A10networks, San Jose, USA

Publications

Research Publications

Boxuan Gu, Wenbin Zhang, Xiaole Bai, Adam Champion, Feng Qin and Dong Xuan. "JSGuard: Shellcode Detection in JavaScript". *Proceeding of International Conference on Security and Privacy in Communication Networks (SecureComm)*, Sep. 2012.

Boxuan Gu, Xiaole Bai, Zhimin Yang, Adam Champion and Dong Xuan. "Malicious Shellcode Detection with a Virtual Memory Snapshot". *Proceeding of IEEE International Conference on Computer Communications (INFOCOM)*, Mar.2010.

Zhimin Yang, Adam C. Champion, Boxuan Gu, Xiaole Bai, and Dong Xuan. "Link-Layer Protection in 802.11i WLANs with Dummy Authentication". *Proceeding of ACM Conference on Wireless Network Security (WiSec)*, Mar.2009

Fields of Study

Major Field: Computer Science And Engineering

Table of Contents

Page

List of Tables

Table **Page**

List of Figures

Chapter 1: Introduction

1.1 Background

Malicious codes are executable codes that can be leveraged to compromise the security of target applications and hosts. According to the US-CERT vulnerability database [109], they constitute one of the biggest threats to the Internet. When a computer is infected by malicious codes, they compromise the confidentiality of data stored thereon. Malicious codes can also compromise the data's integrity. Worse, such codes can leverage victim systems to infect other systems, exploiting them to form a *botnet*—a network of "zombie computers" facilitating large scale attacks via the Internet. A typical example malicious code with such functionality is Conficker.

Conficker [60] targets Microsoft (MS) Windows systems. Before its detection in 2009, millions of computers had been infected [60]. Conficker infects computers and propagates itself by exploiting the MS08-67 vulnerability in the MS Windows Server service [71]. This vulnerability allows attackers to remotely execute arbitrary code when a target system receives a specially crafted remote procedure call (RPC) request. The vulnerability can be used to create exploits with worm functionality [71].

The vulnerability occurs in the `wcscpy_s` function, which copies a string from one memory location to another. The function is called by the macro `_tcscpy_s` that

does not check boundary conditions for the given string. When a very long string is provided, a buffer overflow occurs. This can be exploited to inject malicious codes into a target process and redirect its control flow. In MS Windows systems, the macro `_tcscpy_s` is called by two RPC application programming interfaces (APIs), i.e., the functions `NetPathCompare()` and `NetPathCanonicalize()` [63]. Hence, the vulnerability can be used via RPC to inject malicious codes into a vulnerable system and activate them. When attackers use Conficker to attack vulnerable systems, the first step to inject shellcodes into the target processes. (*Shellcode* is a segment of binary code that is not self-contained.) The detailed infection procedure is as follows [63]:

1. Malicious users craft a packet containing a shellcode, and send it to TCP ports 139 or 445 of a victim system. These two ports are used for file and printer sharing in MS Windows. After the target process receives the packet, the shellcode is injected into the process's address space. Control flow is directed to the shellcode, activating it.

2. After the shellcode is activated, it registers itself as a service, attempts to obtain an IP address from predefined websites, and downloads a Web server program from the website addressed by this IP address. The shellcode enables the Web server after the download completes.

3. Once the Web server is enabled, the shellcode scans for other vulnerable computers. When a target is found, the infected computer's IP address or Uniform

Resource Locator (URL) is sent to the target as the payload. The target computer then downloads the malicious code from the given address and commences infection of other computers.

Conficker has infected millions of computers in the Internet. These compromised computers are interconnected forming a botnet [31]. Such a botnet, controlled by attackers, becomes a platform for further sophisticated attacks in cyberspace.

The Conficker example illustrates the importance of defending against malicious code infection. Our work studies effective malicious code detection.

1.2 State of the Art

In general, malicious code is injected into target systems via input data. Depending on the time when malicious code detection systems are activated, current detection methods can be classified into the following two categories:

- *Detection Before Input Data Consumption (DBC):* The detection systems are activated before the target systems consume the input data. When the detection systems receive these input data, they check if the data contain any malicious code. If malicious code is found, the data are dropped and alerts are raised; otherwise, the target systems consume the data for further processing. Detection methods used in such systems include: (a) signature based methods, which use string patterns to check malicious code [64, 96]; and (b) code analysis based methods, which interpret input data into machine instruction sequences and then check if the instruction sequences exhibit malicious behaviors [55, 66, 82–84, 114].

- *Detection During Input Data Consumption (DDC):* The detection systems are activated when the target systems actually consume the input data. Their execution will be halted if any potential vulnerabilities are exploited. These methods include information flow tracking (IFT) [14, 77, 89], address space randomization (ASR) [10, 80, 116], instruction set randomization (ISR) [8, 62], data space randomization (DSR) [11], and operating system (OS) extension approaches that insert checkpoints in OS kernels or libraries to determine if calls to vulnerable library functions are indeed safe [7, 61].

In general, DDC methods have better detection accuracy because they focus on preventing vulnerability exploitation. When vulnerabilities are exploited, such methods raise alerts and stop execution of target processes. However, they have several disadvantages compared with DBC methods. On the one hand, DDC methods can greatly slow down program execution. They usually need extra codes to instrument execution of target processes. For example, in IFT based methods [14, 77, 89], when the value of one variable is assigned to a second variable, the first variable's origin information needs to be propagated to the second variable. Such propagation is performed by extra code inserted by the DDC method. As a result, the target program's execution is considerably slowed. On the other hand, it is hard to determine the root cause of intrusion using DDC based methods. From an intrusion detection perspective, determining what input data contain malicious code is the most crucial task. After such input data are found, they can be used to generate signatures, which other systems can leverage to check if their input data contain similar malicious codes. This helps stop propagation of malicious codes in a timely manner. In DDC based methods, program vulnerabilities can be located easily, but input data containing

malicious code cannot be identified easily. This occurs because a target program can have many different types of input data that can be modified many times before vulnerability exploitation.

1.3 Motivation

Although current detection methods can be leveraged to help prevent system exploitation, they have two major disadvantages. First, some malicious codes can evade detection by such methods. Second, some benign input data or codes can be incorrectly classified as malicious. These disadvantages arise because current detection methods do not fully leverage context information from target systems or input data while performing detection.

Current DBC based detection methods do not fully leverage such context information while performing malicious code detection. These methods check the input data in an environment isolated from the target process. Malicious codes that intelligently leverage target processes' virtual memory information can thus evade detection. One example is the *swarm attack*, which leverages spatial properties of virtual memory blocks to conduct attacks [23]. In the swarm attack, the entire shellcode is first split into several parts that are sent via different messages to the target process, which places these messages into different virtual memory blocks. After the last message arrives, the shellcode contained in the last message is activated by exploiting a memory error. This shellcode directs control flow to shellcodes received in previous messages and stored in the other blocks. The entire shellcode can be "chained together" and executed. Such attacks cannot be detected by current DBC based detection methods, which can only identify an entire shellcode in one message. When

receiving a message, such methods place it in an isolated environment for checking. This environment does not use the target process's real virtual memory information to chain together shellcodes in different messages.

In addition, current DDC based detection methods do not fully leverage context information from input data in detection. As a result, benign data or codes can be classified as malicious. For example, such misclassification can occur in TaintDroid [41], a DDC detection system that can help detect information leakage in smartphones. In TaintDroid, any data originating from the Internet are labeled as *net*. TaintDroid does not keep the data's source information in the system. When the data are transmitted over the Internet, TaintDroid will raise an alert to indicate that the executed program may be malicious and is leaking sensitive information. Because there is no source information for the data, it is difficult for a smartphone user to determine whether the data transmission is truly illegal. Accordingly, it is difficult for the user to determine if the executed program is truly malicious. For example, AT&T subscribers can use the myAT&T smartphone application to pay their AT&T bills online. Subscribers need to read the bill information from AT&T, process it, and send it to AT&T via the application. In TaintDroid, data related to the bill are labeled *net*. Later on, TaintDroid alerts the user because some labeled data will be transmitted. In this case, without information about the data's exact origin, it is not easy for the user to determine the legality of the billing information flow. As a result, the application can be incorrectly classified as malicious.

From the above discussion, it is obvious that the context information from target systems and input data is very important in malicious code detection. Such context information can help detection systems reduce both their false negative rates and their

false positive rates so that malicious code can be detected more effectively. Therefore, this dissertation focuses on leveraging such context information to detect malicious code.

Despite over two decades of independent, academic, and industrial research, memory errors still undermine the security of computer systems [111]. Typical memory errors are buffer overflows, which are often used to inject malicious code into target processes' address spaces and redirect control flows. For many years, buffer overflows have been ranked among the top 3 of the CWE SANS top 25 most dangerous software errors [93]. Thus, we first explore methods that leverage target processes' context information to detect malicious codes that exploit the processes' memory errors. Although our detection systems are implemented on server and desktop computers, our methods can also be applied to smartphones. In addition, an increasing number of malicious codes target smartphones [45, 108]. Malicious codes that steal sensitive information are extremely dangerous. Hence, we further study detection methods that leverage the context information of target processes and the context information of data stored in smartphones to detect information leakage therein.

1.4 Dissertation Organization

The rest of the dissertation is organized as follows. Chapter 2 presents our system for detecting malicious codes in input data. Chapter 3 introduces our detection system that is used to detect malicious codes that are dynamically generated in JavaScript codes. Chapter 4 presents our system for detecting malicious codes targeting smartphones. Chapter 5 concludes the dissertation.

Chapter 2: Malicious Code Detection In Network Messages

2.1 Overview

According to the US-CERT database [109], buffer overflows are very common and among the most critical software vulnerabilities. Attackers routinely exploit these vulnerabilities to inject malicious shellcodes and gain control of computer systems. Unlike self-contained pieces of malware, malicious shellcodes are segments of binary code disguised as normal input data. They are injected into a target process's virtual memory and hijack the process control flow. It is important to filter out messages that contain such shellcodes before they cause damage.

Recall that the current detection approaches are classified into two categories: *d*etection *d*uring input data *c*onsumption-based approaches (*DDC*-based approaches for short) and *d*etection *b*efore input data *c*onsumption-based approaches (*DBC*-based approaches for short). DDC-based approaches, which conduct detection when the target process *actually* consumes input data, include flow tracking [14, 77, 89], randomization [10, 80, 116], compiler extensions [1, 19, 27, 28, 43, 98], OS extensions [7,61], and hardware modifications [70]. In spite of their strong detection capabilities, these approaches are often heavyweight and they may not detect malicious shellcode in a timely manner [85]. DBC-based approaches, in which input data are checked

before the target process consumes them, can be subdivided into two categories: *static analysis* [21, 22, 65, 107] and *dynamic analysis* [82, 83]. Static analysis uses code-level patterns to detect malicious shellcodes in input data whereas dynamic analysis conducts network-level emulation. Section 2.3 gives a detailed discussion of these analyses.

In this chapter, we focus on DBC-based approaches, which are lightweight. However, such existing approaches have limited detection capability. Attacks exploit knowledge of the target process's runtime information to evade detection by DBC-based approaches via techniques such as polymorphism and metamorphism. We will present two representative examples in Section 2.3 that illustrate these attacks. The first example uses runtime information generated by the target process and the second uses dynamic information generated by the attack on the fly. Existing DBC-based approaches fail to detect these attacks. In general, these approaches conduct detection by scanning input data, but they rarely utilize the target process's runtime information. We aim to remedy this situation by introducing a new detection methodology to detect *malicious shellcodes*. We do *not* focus on detecting self-contained malware programs.

We highlight our contributions in this chapter as follows.

• We propose a new malicious shellcode detection methodology in which we take *snapshots* of the target process's virtual memory immediately before input data are consumed and "feed" these snapshots to a *lightweight* DBC-based malicious code detector that we design and implement.

• We propose using these snapshots to instantiate a runtime environment that emulates the target process's *input data consumption*. This environment facilitates

monitoring shellcodes' behaviors. We use the snapshots to examine *system calls* invoked by (executable) input data and the parameters thereof as well as the process's execution flow to detect malicious shellcodes.

- We implement a prototype system based on the above methodology in Debian Linux with kernel version 2.6.26. Our system is adaptive and extensible. It is designed to be loaded into the target process's address space statically or dynamically. Our system can efficiently fetch and use the target process's virtual memory snapshots for shellcode detection.

- We conduct extensive experiments based on real traces and thousands of malicious shellcode samples. The experimental results illustrate our detection's low runtime overhead and high detection accuracy, which achieves lower overhead and higher accuracy than state-of-the-art DBC-based approaches.

Note that unlike emulators such as QEMU [88] that emulate an entire OS and processes running therein, our system only emulates instructions decoded from input data. In addition, our detection methodology has broad adaptability. Other dynamic analysis techniques can be developed based on virtual memory snapshots and DBC-based static analysis approaches can apply our methodology to augment their shellcode detection capabilities.

Chapter Organization. The rest of this chapter is organized as follows. We introduce related work and its limitations in Section 2.3. We present our system design in Section 2.3. We present our implementation and evaluation thereof in Section 2.4. We conclude in Section 2.5.

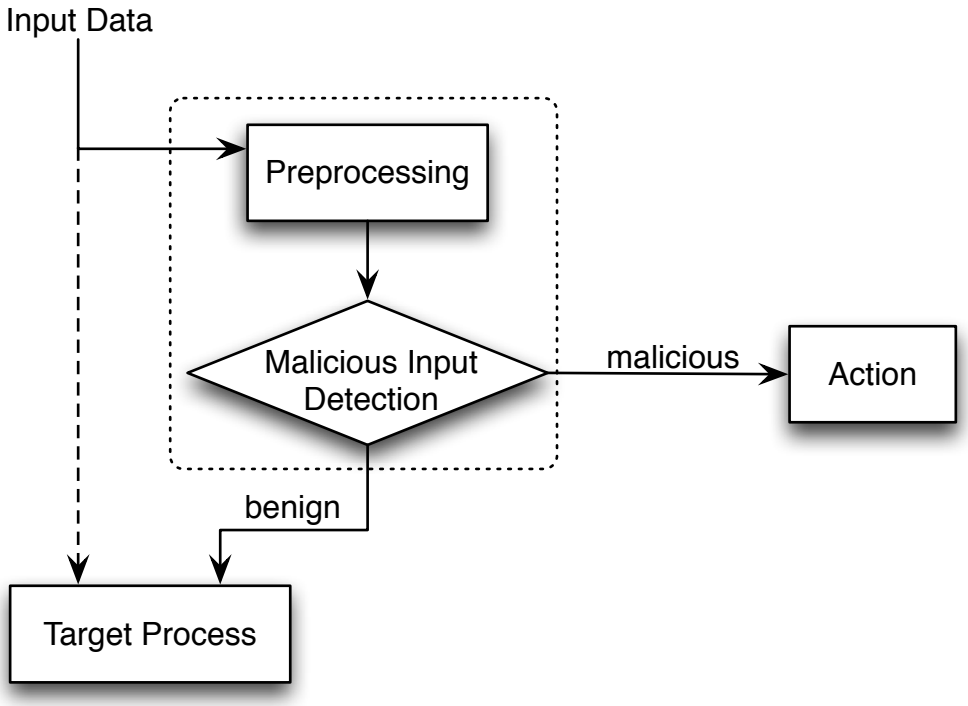


Figure 2.1: In DBC-based intrusion detection approaches, the input data are checked before they are consumed by the target process. Such state-of-the-art approaches do not use the target process's runtime information.

2.2 Limitations of Existing Work

Intrusion detection approaches can be classified as *D*etection *B*efore Input Data *C*onsumption-based (DBC-based) and *D*etection *D*uring Input Data *C*onsumption-based (DDC-based).

2.2.1 DBC-Based Approaches

In this intrusion detection approach category, input data are checked as *executables* before the target process consumes them as shown in Fig. 2.1. We subdivide this category into two subcategories: *static analysis* and *dynamic analysis*.

Static Analysis. In static analysis, input data are first disassembled and then screened via code-level pattern analysis and matching. Patterns can be complicated signatures or simple heuristics obtained from studying known malicious codes. In [107], Toth and Kruegel proposed identifying exploit code by detecting NOP sleds, an approach that the Snort IDS preprocessor also uses [48]. However, attacks can bypass this detection by either *not* including NOP sleds or by using polymorphic techniques [18, 36, 68]. Chritodorescu in [21] and [22] proposed techniques to detect malicious patterns in executables by semantic heuristics. Lakhotia and Eric in [65] used static analysis techniques to detect obfuscated calls in binaries. Chinchani and van den Berg proposed a rule-based scheme in [18]. Wang et al. proposed SigFree [114] that checks if packets contain malicious codes using "push and call" patterns and the number of useful instructions in the longest possible execution chain.

In general, static analysis is efficient. However, it has limitations regarding accuracy and completeness. In [9], Bayer et al. point out that, in general, determining program behavior via static analysis is undecidable, and binary obfuscation often effectively thwarts both the disassembly and code analysis steps.

Dynamic Analysis. Instead of static code-level patterns, dynamic analysis detects malicious input data by exploiting information generated during program execution on those data. The state-of-the-art dynamic analysis approach is network-level emulation, which Polychronakis et al. proposed [82, 83]. Network-level emulation disassembles input data into several possible execution chains and then emulates the execution of each chain. If any chain exhibits malicious behavior during emulation, the input data are classified as malicious.

```
 1  push esp            ; esp points here
 2  pop  eax            ; [eax] <- 0xBFFF0000
 3  sub  eax,0x2d2d2d6c ; [eax] <- 0x92D1D294
 4  sub  eax,0x5858557a ; [eax] <- 0x3A797D1A
 5  sub  eax,0x7a7a7a7a ; [eax] <- 0xBFFF02A0
 6  push eax
 7  pop  esp            ; [esp] <- 0xBFFF02A0
 8  and  eax,0x2321252d ; [eax] <- 0x20012020
 9  and  eax,0x44424242 ; [eax] <- 0x00000000
10  sub  eax,0x2d2d2d2d ; [eax] <- 0xD2D2D2D3
11  sub  eax,0x252d252d ; [eax] <- 0xADA5ADA6
12  sub  eax,0x655e6761 ; [eax] <- 0x48474645
13  push eax            ; M[0xBFFF029c] = "EFGH"
14  sub  eax,0x2d2d2d2d ; [eax] <- 0x1B1A1918
15  sub  eax,0x5e5e5e5e ; [eax] <- 0xBCBBBABA
16  sub  eax,0x78787879 ; [eax] <- 0x44434241
17  push eax            ; M[0xBFFF0298] = "ABCD"
...
```

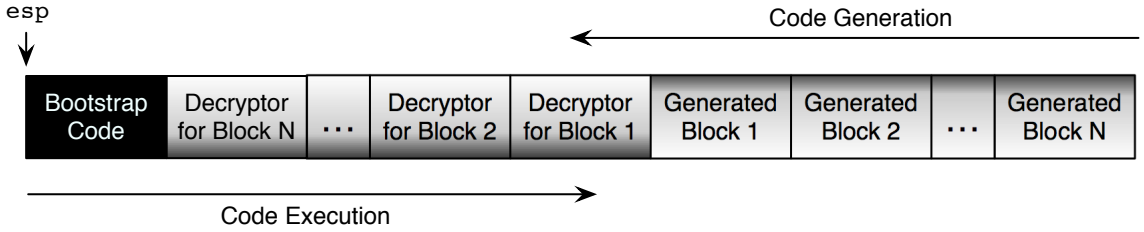


Figure 2.2: Original Shellcode Execution Trace Produced by "Encode" Engine.

Even though network-level emulation can achieve better detection completeness than static analysis, it is still prone to evasion. The problem is that it has insufficient context information about the target process and thus it must make assumptions during initialization and execution emulation. We will introduce two representative examples that can bypass it.

Before we give the examples, we provide some background information to aid understanding. Our examples are derived from shellcodes that are produced by an adapted version of the "Encode" shellcode engine [97]. The attack mechanism of such shellcode is illustrated in Fig. 2.2. The shellcode exploits a stack buffer overflow to which `esp` points after the shellcode is injected into the target process's address

```
 1  push esp              ; esp points here
 2  pop  eax              ; [eax] <- 0xBFFF0000
 3  sub  eax,0x2d2d2d6c   ; [eax] <- 0x92D1D294
 4  sub  eax,0x5858557a   ; [eax] <- 0x3A797D1A
 5  sub  eax,0x7a7a7a7a   ; [eax] <- 0xBFFF02A0
 6  push eax
 7  pop  esp              ; [esp] <- 0xBFFF02A0
 8  and  eax,0x2321252d   ; [eax] <- 0x20012020
 9  and  eax,0x44424242   ; [eax] <- 0x00000000
10  mov  ebx,0xa0ef(ebp); 0xa0ef is known
11  cmp  ebx,0x252d252d   ; compare [ebx], 0x252d252d
12  jz   +3               ; conditional jump
13  lidt eax              ; privileged instruction
14  sub  eax,ebx          ; [eax] <- 0x48474645
15  push eax              ; M[0xBFFF029c] = "EFGH"
16  add  ebx,0x04350435
17  add  ebx,0x04cb04cb
18  cmp  ebx,0x2d2d3d3d   ; compare [ebx], 0x2d2d2d2d
19  jz   +3               ; conditional jump
20  lidt eax              ; privileged instruction
21  sub  eax,ebx          ; [eax] <- 0x44434241
22  push eax              ; M[0xBFFF0298] = "ABCD"
```

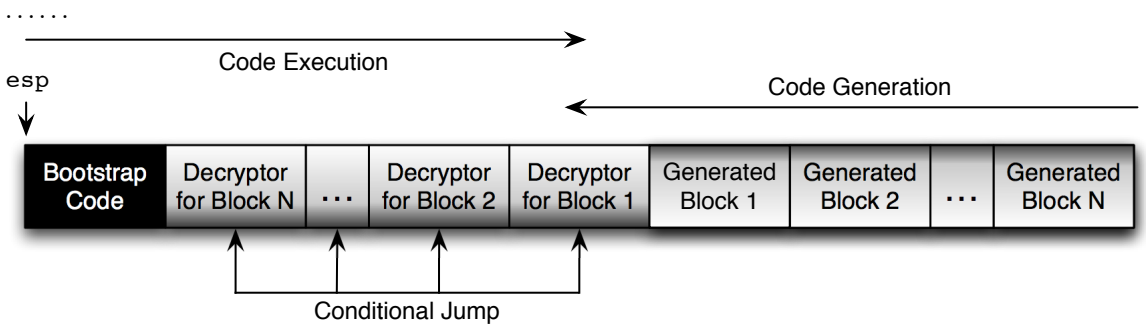


Figure 2.3: Shellcode for Example 1.

space. After activation, this shellcode decrypts a series of encrypted payloads, each of which contains 4 bytes. Due to the stack's LIFO nature, the decrypted payload is generated *backwards*, starting with its *last* four bytes. When the final decrypted block is pushed onto the stack, the shellcode's control flow "meets" the newly built payload and the control flow is redirected to the decrypted instructions. Though is difficult for static analysis to detect such attacks, network-level emulation can detect them [83]. Now we are ready to introduce the examples.

– *Example 1:* Fig. 2.3 shows the IA-32 assembly language code for the first type of attack. Assume the attacker has some knowledge of the target process runtime information. More specifically, he knows that a constant value is stored somewhere in memory that can be used as well as the value in `ebp`, which is true for almost all stack-based buffer overflows [23]. After zeroing `eax` (in instruction 9), the decryption block process begins, again using separate decryption blocks (10–15, 16–21, ...) for each four bytes of the encrypted payloads. However, the decryption blocks in this trace rely on a constant value stored in address `0xa0ef + (ebp)`. This value is put into `ebx` (instruction 10), and then compared with a desired value (instruction 11). If this value differs from the one the attacker wants, decryption stops and a privileged instruction is executed (instruction 13). Otherwise, decryption continues. As network-level emulation does not have *real* runtime information about the target process, all eight (emulated) general-purpose registers are initialized to hold the absolute address of the first instruction of each execution chain. An incorrect `ebp` will result in the emulated execution of instruction 13. Since `lidt` is a privileged operation, which malicious shellcodes should not contain, emulation will stop and consider the execution chain containing this instruction benign.

– *Example 2:* Chung and Mok proposed a new type of attack, the *swarm attack*, which can defeat existing DBC-based approaches that detect malicious shellcodes *one message at a time* [23]. In swarm attacks' methodology, shellcodes can be injected via *several* messages instead of *one* message. The authors point out that it is possible to create the decoder inside the attacked process's address space using multiple instances thereof with each instance of the attack writing a small part of the decoder at the designated location [23]. Fig. 2.4 shows the IA-32 assembly language code of an

attack using this methodology. In this case, the entire shellcode is separated into several blocks, each of which (except the first) is a decryptor and can *only* generate *one* new instruction if it is executed. If the entire shellcode is sent to a target process using *one* message, then network-level emulation will easily detect it [83] because the number of newly generated instructions during emulation of the malicious shellcode will exceed a threshold that is set in advance. According to the swarm methodology, if each block is injected into a target process using one message via one attack instance, the entire malicious shellcode can evade detection by network-level emulation because the threshold for the number of newly generated instructions during the emulation of a shellcode is too large [82, 83].

The above two examples represent two types of attacks that can thwart state-of-the-art DBC approaches. The first type exploits runtime information generated by the *target process*. The second one uses information generated by the *previous attacks*.

2.2.2 DDC-Based Approaches

In this intrusion detection approach category, detection is conducted *while* processes consume input data. If malicious runtime behavior is detected, the process often stops and sounds an alarm. The input data and the log of target process states are stored for further analysis. Flow tracking [14, 77, 89] uses a taint-based method to check whether the input data are malicious. Randomization can effectively defend against attacks exploiting memory errors through address space randomization (ASR) [10, 80, 116], instruction set randomization (ISR) [8, 62] and data space randomization (DSR) [11]. OS extension approaches insert checkpoints in OS kernels or

```
1  push esp            ;
2  pop  eax            ; [eax] <- 0xBFFF0000
3  sub  eax,0x2d2d2d6c ; [eax] <- 0x92D1D294
4  sub  eax,0x5858557a ; [eax] <- 0x3A797D1A
5  sub  eax,0x7a7a7a7a ; [eax] <- 0xBFFF02A0
6  push eax
7  pop  esp            ; [esp] <- 0xBFFF02A0
8  and  eax,0x2321252d ; [eax] <- 0x20012020
9  and  eax,0x44424242 ; [eax] <- 0x00000000
```

Message 1

```
10 sub  eax,0x2d2d2d2d ; [eax] <- 0xD2D2D2D3
11 sub  eax,0x252d252d ; [eax] <- 0xADA5ADA6
12 sub  eax,0x655e6761 ; [eax] <- 0x48474645
13 push eax            ; M[0xBFFF029c] = "EFGH"
```

Message 2

```
14 sub  eax,0x2d2d2d2d ; [eax] <- 0x1B1A1918
15 sub  eax,0x252d252d ; [eax] <- 0xBCBBBABA
16 sub  eax,0x655e6761 ; [eax] <- 0x44434241
17 push eax            ; M[0xBFFF029c] = "ABCD"
```

Message 3

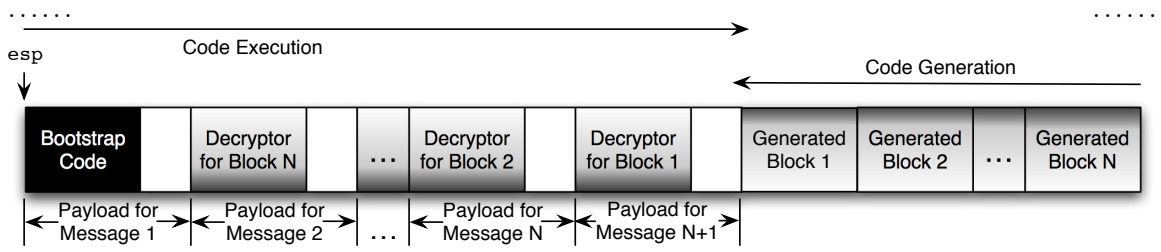


Figure 2.4: Shellcode for Example 2.

libraries to determine if calls to vulnerable library functions are safe [7,61]. In general, DDC-based approaches have good detection completeness due to their extensive use of context information. However, troubleshooting is inefficient [85, 114].

Compared with DDC-based approaches, DBC-based approaches are efficient in troubleshooting but their detection completeness is not as good as DDC-based approaches' detection completeness. This is because DBC-based approaches do not

use the runtime virtual memory information of target processes. We design a new DBC-based detection methodology that overcomes this limitation.

2.3 System Design

2.3.1 Design Rationale

We propose taking snapshots of the target process's virtual memory before input data are consumed and feeding these snapshots to our DBC-based detection system. This idea is illustrated in Fig. 2.5, where the DBC-based detector has a *new* input that is the virtual memory runtime information of the target process.

A virtual memory snapshot records the state of a process's virtual memory at this time, including the target process's address space and register values. In our system, snapshots are used in the following two ways:

– *Instantiating Virtual Execution Environment.* Immediately before the target process consumes data, its control flow is directed to our system. A virtual memory snapshot is taken at this moment and the detection procedure is immediately activated, instantiating a virtual environment where the input data are executed and monitored. Snapshots are used to initialize this environment and provide two benefits. First, snapshots are critical for observing the input data's *real* behaviors, as they illustrate *real* execution flow. To accurately reveal shellcodes' behavior, the environment should be able to mimic the process's consumption of input data. In malicious shellcodes, process state information can be used to redirect the execution flow, e.g, for encryption or decryption as Example 1 illustrates. Without precise virtual memory information about the process, shellcodes' execution flow can be changed or even

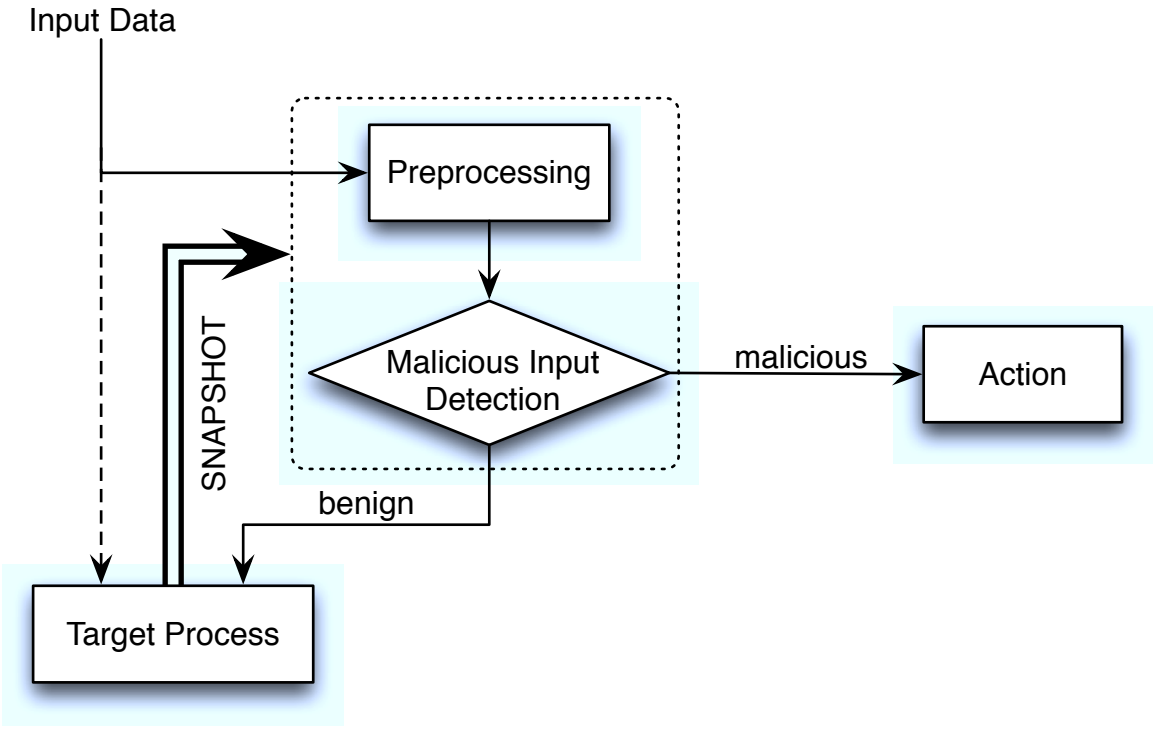


Figure 2.5: We propose a new DBC-based intrusion detection methodology that feeds a virtual memory snapshot of the target process to the detector.

interrupted. Second, since the virtual memory snapshot is compact and easy to obtain, our system's virtual environment is *lightweight*, unlike the environments that *complete* system emulators such as QEMU create [88]. In existing DBC-based approaches, it is hard to observe *real* shellcode behavior in detection systems since the target process's virtual memory information is either assumed or ignored [82, 83, 114].

– *Facilitating System Call-based Detection.* One type of shellcode behavior that virtual memory snapshots facilitate is system call invocation, which is valuable for detection. Malicious shellcode must switch to kernel mode to launch OS-related operations by invoking system calls. No matter how well malicious shellcode disguises

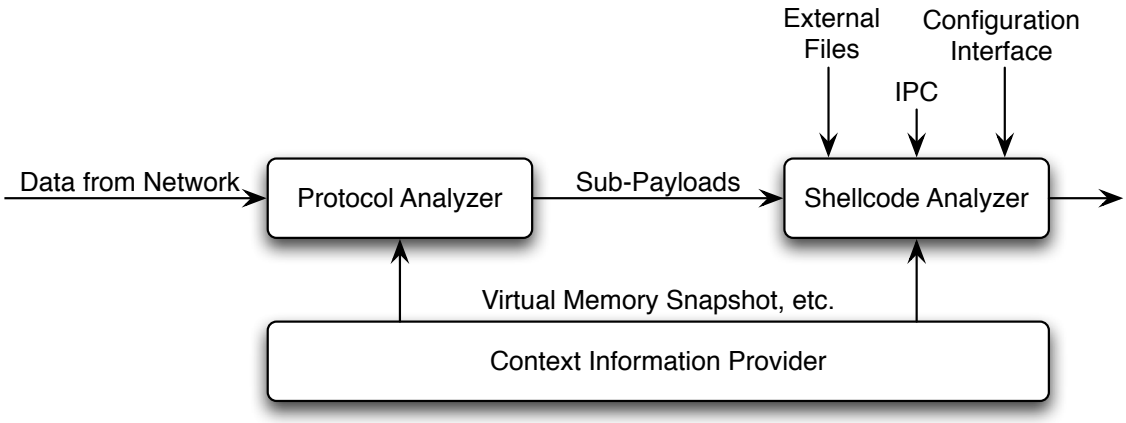


Figure 2.6: Our System Modules.

itself, it will eventually use system calls to launch attacks. Different system call operations are distinguished by system call numbers and parameters, which are normally stored in the registers. For example, in the IA-32 architecture, the system call number is placed into `eax` and its parameters are placed in `ebx`, `ecx`, `edx`, `esi`, `edi` and `ebp` as necessary. Existing DBC-based approaches, including both static analysis and dynamic analysis, cannot use system call invocations as detection criteria as they lack such necessary register information [82,83,113]. As we use virtual memory snapshots, we can accurately distinguish among different system calls to improve detection capability as well as accuracy.

In the following, we first introduce our system's architecture followed by its workflow.

2.3.2 System Architecture

In this section, we first introduce our system's entire architecture and then we present its key module—the shellcode analyzer.

Architecture

There are three modules in our system as shown in Fig. 2.6. The first module is the *Protocol Analyzer*, which extracts headers from input messages and separates the payload into several sub-payloads (fields) as necessary according to upper-layer specifications. The second module is the *Shellcode Analyzer*, which is responsible for detecting malicious shellcodes with virtual memory snapshots. It can receive sub-payloads from the protocol analyzer, an external file, or another process within the same host through local IPC. The third module is the the *Context Information Provider*, which is an interface between the *emulated* execution environment and the *real* one. This component assists the protocol analyzer in producing correct sub-payloads and the shellcode analyzer in conducting detection.

Key Module – Shellcode Analyzer

The architecture of the shellcode analyzer is shown in Fig. 2.7. This module consists of an instruction decoder, an instruction emulator, a malicious behavior detector, an emulated memory system, and a set of emulated registers. The instruction decoder iteratively decodes buffer contents into instructions and sends each one to the emulator. For each instruction the emulator receives, it emulates the execution thereof, for which the emulated memory system and registers provide a runtime virtual environment. This environment is instantiated by the virtual memory snapshot that is taken through the virtual memory snapshot interface. During emulation, virtual memory or register access is directed to the emulated memory system or registers.

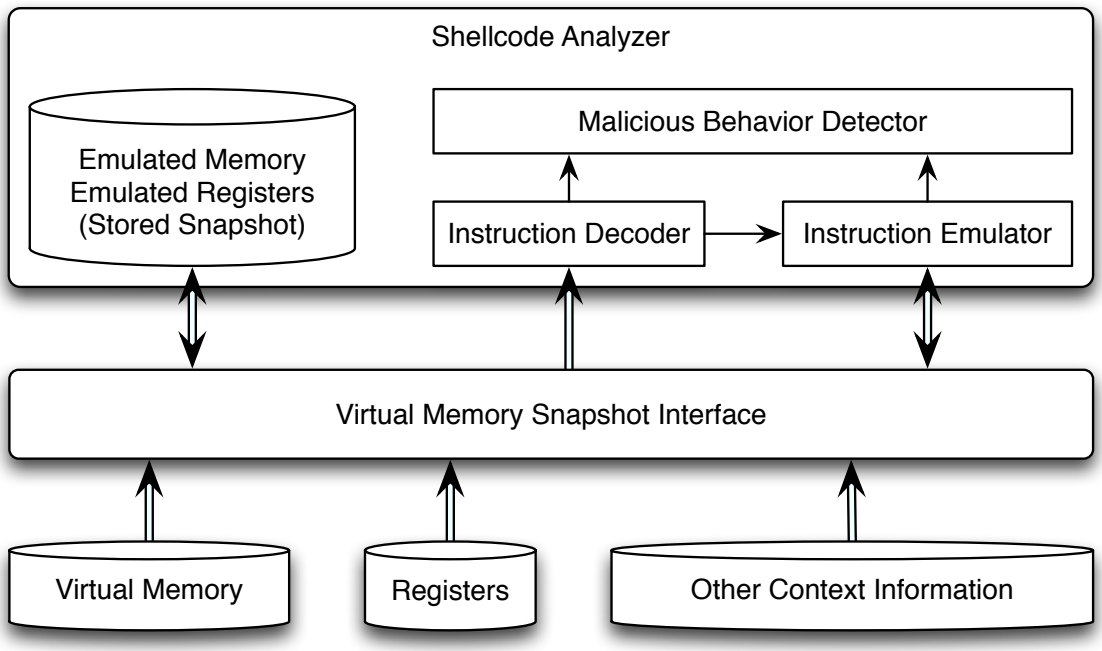


Figure 2.7: The Architecture of the Shellcode Analyzer.

2.3.3 Workflow

In this section, we first introduce the overview of our entire system's workflow, then we present our shellcode analyzer's workflow.

Overview

Recall that our system has three modules: the protocol analyzer, the shellcode analyzer, and the context information provider. After receiving a message from a network, our detection system first uses the protocol analyzer to extract the message header and then separates the payload into several sub-payloads as necessary. Then the protocol analyzer feeds the payload (or sub-payloads) to the shellcode analyzer. The shellcode analyzer's instruction decoder treats the payload (or each sub-payload)

as a byte-indexed sequence of input data. It decodes the data into different instruction sequences according to different start positions and sends them to the emulator. The virtual memory snapshot is taken at this moment and used to instantiate the emulation environment. The emulator executes each instruction sequence to determine whether there is any malicious behavior. If any such behavior is detected, the sequence is considered malicious, and if this occurs, the shellcode analyzer will conclude that the there is malicious shellcode in the payload (or sub-payloads) and the message is considered malicious.

Our detection system is not intended to run as an independent process. After a target process is created, our system can be loaded into its address space either statically or dynamically. When a target process receives a message from a network interface, our system is activated and the control flow is directed to our system.

Shellcode Analyzer Workflow

The shellcode analyzer workflow is shown in Fig. 2.8. From each position of the input data, the shellcode analyzer uses the virtual memory snapshot to emulate the execution of the decoded instruction sequence. Snapshots facilitate observing *real* behaviors of the input data by illustrating the *real* execution flow. There are two input parameters for `ShellcodeAnalyzer()`. The first is `base_address`, the starting address of the input data to be analyzed. The second is `base_size`, the size of the input data.

Precise virtual memory information provided by the snapshot ensures the shellcode's execution flow is not interrupted. If an instruction sequence is malicious shellcode, all of the instructions it uses are eventually reached during emulation via the execution flow, including the shellcode's, those generated by previous messages, and

```
1 #define MALICIOUS    1
2 #define BENIGN       0
3 #define MALICIOUS_SEQUENCE    1
4 #define BENIGN_SEQUENCE       0
5
6 int ShellcodeAnalyzer(base_addr, base_size)
7 {
8     for (i = 0; i< base_size; i++)
9          if (MaliciousInstructionSeq(base_addr + i))
10             return MALICIOUS;
11    return BENIGN;
12 }
13
14 int MaliciousInstructionSeq(addr)
15 {
16    InitializeEmulationEnvironment();
17    instruction = InstructionDecoder(addr);
18    if (End(instruction)) return BENIGN_SEQUENCE;
19    instruction.exe_depth = 1;
20    while (instruction)
21    {
22    if (MaliciousSystemCall(instruction))
23           if(instruction.exe_depth > exe_depth_threshold)
24                    return MALICIOUS_SEQUENCE;
25       InstructionEmulator(instruction);
26       UpdateEmulationEnvironment();
27       target = ComputeTarget(instruction);
28       prev_instruction = instruction;
29       instruction = InstructionDecoder(target);
30       if (End(instruction)) break;
31       SetExecutionDepth(instruction, prevInstruction);
32    }
33    return BENIGN_SEQUENCE;
34 }
```

Figure 2.8: Shellcode Analyzer Workflow.

those from the libraries loaded into the target process's address space. Thus emulation within the shellcode analyzer with a virtual memory snapshot can be used to accurately observe the behaviors of polymorphic and metamorphic shellcodes as well as shellcodes used in swarm attacks.

The key function of the shellcode analyzer is `MaliciousInstructionSeq()`, which detects a malicious instruction sequence. The workflow of `MaliciousInstructionSeq()` is shown in lines 14–34 in Fig. 2.8. At the beginning of `MaliciousInstructionSeq()`, the emulation environment is instantiated by taking a virtual memory snapshot. The `while` loop from line 20 to line 32 in Fig. 2.8 emulates a sequence of

instructions, which continues until one of the following occurs: (1) a malicious behavior is detected; (2) a privileged or invalid instruction is encountered;[1] (3) an illegal memory access occurs; (4) the number of executed instructions exceeds a threshold.

`MaliciousInstructionSeq()` returns `BENIGN_SEQUENCE` for conditions (2)–(4) and `MALICIOUS_SEQUENCE` for condition (1). For this condition, a *malicious behavior* is defined in our system by a *malicious* system call invocation. In Linux and MS Windows systems, not all system calls can compromise the target host's security. This depends on a system call's number and its parameters, which are stored in registers before a system call instruction is executed. Because of the snapshot, the system call number and its parameters can be accurately obtained to determine if the system call invocation is intended to compromise host security. For example, in Linux, system call number `11` corresponds to the system function `execve`, which executes a program. During emulation of an instruction, if it is a system call instruction and the value of the emulated `eax` is `11`, then the system call number is `11`. After checking its parameters stored in other emulated registers, if its first parameter is `/bin/sh`, then we can conclude that the instruction tries to open a root shell that can be used to compromise the host's security. In this case, the system call instruction will be considered to be a *malicious* system call.

Note that malicious shellcode normally uses several instructions to initialize system call parameters. We also use the `exe_depth` of an instruction that invokes a system call to decrease false positives, which is shown in line 23 of Fig. 2.8. An instruction's `exe_depth` is defined as the number of instructions from the starting point to it during emulation of an instruction sequence. For example, suppose that

[1]Privileged instructions can only be executed in kernel mode while a shellcode normally runs in user mode. If a shellcode contains a privileged instruction, it leads to an exception.

a statement S in a `for` loop is executed 100 times. Then the execution depth of S is `2` (the `for` statement and S). We further discuss the execution depth threshold in Section IV.

2.3.4 Detection Example

In this section, we use the example shown in Fig. 2.3 to illustrate our system's entire detection procedure. We assume that this malicious shellcode is inserted into the body of an HTTP request message and that it aims to open a root shell in an HTTP server with an exploitable memory error. When this message is received, the server's control flow is directed to our system. The protocol analyzer extracts the payload from the request body and calls the function `ShellcodeAnalyzer()` to analyze it. The instruction decoder will decode it into several instruction sequences, and feed each one to the function `MaliciousInstructionSeq()`. Immediately before emulating the execution, a virtual memory snapshot is taken. At this moment, suppose that the value of the memory unit pointed by `0xa0ef(ebp)` is `0x252d252d`.

Suppose that the function `MaliciousInstructionSeq()` is emulating an instruction sequence that contains malicious shellcode, and the instruction being emulated is `mov ebx,0xa0ef(ebp)` as shown in line 10 of Fig. 2.3. After emulating the instruction, the value of the emulated `ebx` is `0x252d252d`. After emulating the next two instructions, because the value of the emulated `ebx` is `0x252d252d`, the control flow will reach the instruction on line 15 of Fig. 2.3, namely, `push eax`. After emulating it, the first decrypted instruction will be written into the emulated memory system. Similarly, all of the encrypted instructions are decrypted and stored in the emulated memory system.

When the control flow later reaches the first decrypted instruction, all the decrypted instructions are read from the emulated memory system. Finally, the control flow reaches a system call instruction, and then the function `MaliciousSystemCall()` is called. By checking the emulated registers' values, we discover that the system call instruction tries to open a root shell. In this case, this system call instruction will be considered to be a malicious system call instruction. In addition, its *execution depth* exceeds the preset threshold. Thus the emulated instruction sequence is considered malicious as are the analyzed payload and the request message. After the control flow returns from our detection system, the server will discard the message.

2.4 Implementation and Experimental Evaluation

2.4.1 Implementation

Our detection system is implemented in Debian Linux with kernel version 2.6.26. We build our prototype system in C using `gcc` 4.3.2. The prototype system focuses on the IA-32 architecture and Linux operating systems. The core component of our prototype system is the instruction emulator. In the following, we present its implementation.

Our instruction emulator can *interpret* all IA-32 instructions and *emulates* a subset thereof, including all general-purpose instructions and the FPU instructions that are used to obtain injected shellcodes' absolute addresses [59,82]. In addition, we also emulate some system instructions such as `rtdsc`. This subset contains all instructions used by known malicious shellcodes in useful computation of malicious attacks [82]. We consider our implemented subset sufficient, though extensions are feasible. When an unimplemented instruction is encountered in emulation, if it is not a privileged

instruction, the control flow skips it and goes to the next instruction; otherwise, emulation stops. When encountering an system call instruction, namely, `sysenter` and `int 0x80`, we check if it is one of 36 system calls that can be used to compromise the Linux system security [73]. Besides these "malicious" system calls, we also use the `exe_depth` threshold to determine if the instruction truly tries to compromise the host's security; we set the threshold to 14 since most unencrypted malicious shellcodes have at least 14 instructions [83, 114]. Recall that an emulation termination condition is that the number of the executed instructions reaches a threshold. According to current research, it suffices to set the threshold to `7000` in order to detect malicious shellcodes in the input stream [82, 83]. In our system, we also use this threshold.

Besides this core component, we also implement an emulated memory system and a virtual memory snapshot interface. The snapshot interface contains procedures to access the host's virtual memory snapshot and context information. Moreover, we use the Bastard project's `libdisasm`, version 0.23-pre [2] to construct our instruction decoder. In addition, during emulation of instructions, our detection system also collects the data that could be leveraged to implement the detection rules that are proposed by other papers [82, 83].

We have integrated our system into `glibc` and we also modify some read functions thereof. When these functions are called, they call our detection system to detect input data before returning to the calling procedure. If an application is statically linked to `glibc`, it must be rebuilt. However, when an application is dynamically linked to `glibc`, rebuilding is unnecessary, as our system will be dynamically loaded into the target process's address space. Users can also directly use our detection system in their source codes to check network data before they are consumed. Thus

Data Set	# Requests	# Responses	Size (MB)
1	56,002	56,002	628.5
2	64,916	64,916	840.5
3	31,229	31,229	341.2
4	52,003	51,965	721.3
Total	204,150	204,112	2,531.5

Figure 2.9: Collected Data Sets.

our detection system accurately obtains the virtual memory snapshot that malicious shellcodes utilize and then the detection accuracy of our detection system can be further increased.

2.4.2 Experiments

In this section, we use malicious shellcodes and HTTP traces to conduct our experiments, which are divided into two parts. The first part tests our detection system's effectiveness. The second part measures its overhead.

Effectiveness

In this part of the experiments, we first describe the collection of the data sets that are used to measure the false negatives and false positives of our detection system, and then we evaluate the results of our experiments.

We collect 51 unencrypted malicious shellcodes from the Internet that target Linux systems. We use these shellcodes in conjunction with the following encryption tools to

generate 5000 encrypted malicious shellcodes: the Metasploit project's JumpCallAdditive, Pex, PexFnstenvMov, PexFnstenvSub, and ShikataGaNai [104] as well as ADMmutate [68] and TAPiON [6]. These 5000 encrypted malicious shellcodes and the 51 unencrypted malicious shellcodes are used to test our system's false negatives.

Since most Internet traffic is HTTP traffic, we use HTTP traces to test our system's false positive of our detection system. Because it is very difficult to obtain real traces of HTTP messages that contain the entire contents thereof, we enlist four volunteers who collect HTTP messages for six weeks and place them into data sets 1–4. The properties of these data sets are shown in Fig. 2.9, where *# Requests* and *# Responses* denote the number of request messages and response messages, respectively. The overall size of the traces are about 2.5 GB. The traffic is captured using Fiddler [46]. After obtaining the data sets, we feed our detection system these four data sets to test the false positives thereof.

In both of these two experiments, we run our detection system on a computer and then we feed the data into our system for testing. In the following, we present the evaluation of the experiments of this part.

Fig. 2.10 presents the false positives on request messages and response messages, respectively. *# Malicious* is the number of messages that our system considers malicious.

All request messages in these data sets are classified as benign messages with very few false positives. Response messages have few false positives. After checking the response messages that are classified as malicious, we find that some of them contain MS Windows executables. Others have large binary objects that have execution chains with malicious system call instructions and the execution depths of

Data Set	Request Messages		Response Messages	
	# Malicious	False Positives	# Malicious	False Positives
1	0	0	41	0.000732
2	0	0	43	0.000662
3	0	0	27	0.000865
4	0	0	45	0.000866

Figure 2.10: False Positives for the HTTP Traces.

these instructions exceed the preset execution depth threshold. If a user receives a response message that is classified as malicious, he can choose to accept it, decline it, or perform a further check based on his knowledge of desirable response messages.

After sending the 5000 encrypted malicious shellcodes to our detection system, we find that it can detect all of them. This result is the same as other DBC-based approaches [82, 83, 113]. Although they have the same number of false negatives as our detection system, our system's false positives are lower than those of other DBC-based approaches.

Current approaches based on network-level emulation use the *WX threshold* to detect malicious shellcodes, where the `WX` instructions are those that are executed and are dynamically generated during shellcode emulation [83]. When we set this threshold to `8` as suggested by [83] and use it to detect the 51 unencrypted malicious shellcodes, we find that 31 of them cannot be detected. After studying the traces of these shellcodes, we find that they have no sequences of `WX` instructions of length `8` or larger. We feed these 51 unencrypted malicious shellcodes to our detection system and we find that it can detect all of them. It can detect them because it uses malicious

Data Set	Request Payload Size (bytes)				Overhead (ms)			
	Min	Max	Average	Median	Min	Max	Average	Median
1	154	3412	799.28	736	2.002	125.144997	13.0219431	12.272
2	132	3864	826.93	827	3.032	120.025002	14.12347399	13.484
3	152	3822	921.02	890	2.107	150.350998	13.80020388	12.386
4	110	3845	840.89	777	1.916	138.990997	12.43439796	11.696

Figure 2.11: Net Overhead Incurred by Our System.

system calls and emulation with a virtual memory snapshot to check whether an message contains a malicious shellcode, and all malicious shellcodes eventually invoke system calls to launch attacks.

Furthermore, our detection system can detect the examples presented in Figs. 2.3 and 2.4 but existing DBC-based approaches cannot do so. The reason is that they do not use the target process's context information to detect malicious shellcode.

Overhead

In this section, we evaluate the performance of our system. Our tests consists of two experiments. The purpose of the first experiment is to evaluate the *net* overhead incurred by our detection system. The purpose of the second experiment is to investigate how our system affects the performance of a real HTTP server.

– *Experiment 1.* The first experiment is conducted on a Dell Dimension 5150 machine with an Intel Pentium 4 2.8 GHz processor and 1 GB RAM. In this experiment, we only use the HTTP request messages from the above four data sets to conduct the experiment since the main purpose of our detection system is to protect a server. To evaluate the net overhead incurred by our detection system, we first send a request

message to the detection system and record the timestamps before our detection system is invoked and after it returns. The difference between the two timestamps is considered net overhead.

Fig. 2.11 presents our evaluation results. We believe the results are acceptable for most Internet application servers, though they are usually much more powerful than our test machine.

– *Experiment 2.* In the second experiment, we use two computers to test the performance of our system. One is a Dell Dimension 5150 Web server with an Intel Pentium 4 2.8 GHz processor and 1 GB RAM. The other is an IBM ThinkPad T60 Web client. Both of them are running Debian Linux with kernel version 2.6.26. They are connected by a 100 Mbps Ethernet switch. In the following, we first present the methods and data used in two experiments, and then we evaluate the results of these two experiments.

In the second experiment, we use the modified `glibc` with our detection system to rebuild a real Web server and evaluate our system's influence on the server's performance. We compare the performance of a Web server using our detection system with that of a server not using it. We use `thttpd`, a single-threaded open source Web server from ACME Laboratories that is designed for simplicity, a small execution footprint, and speed [106]. For the server using our system, when it calls `read()` or `recv()` to read from a socket, each function uses our system to analyze the data therefrom. Each function returns `-1` if the data are identified as malicious; otherwise, returns `0` and the data are consumed normally.

The client traffic was generated by Jef Poskanzer's HTTP load program [58] from the IBM machine. Clients send requests from a predefined URL list from which the

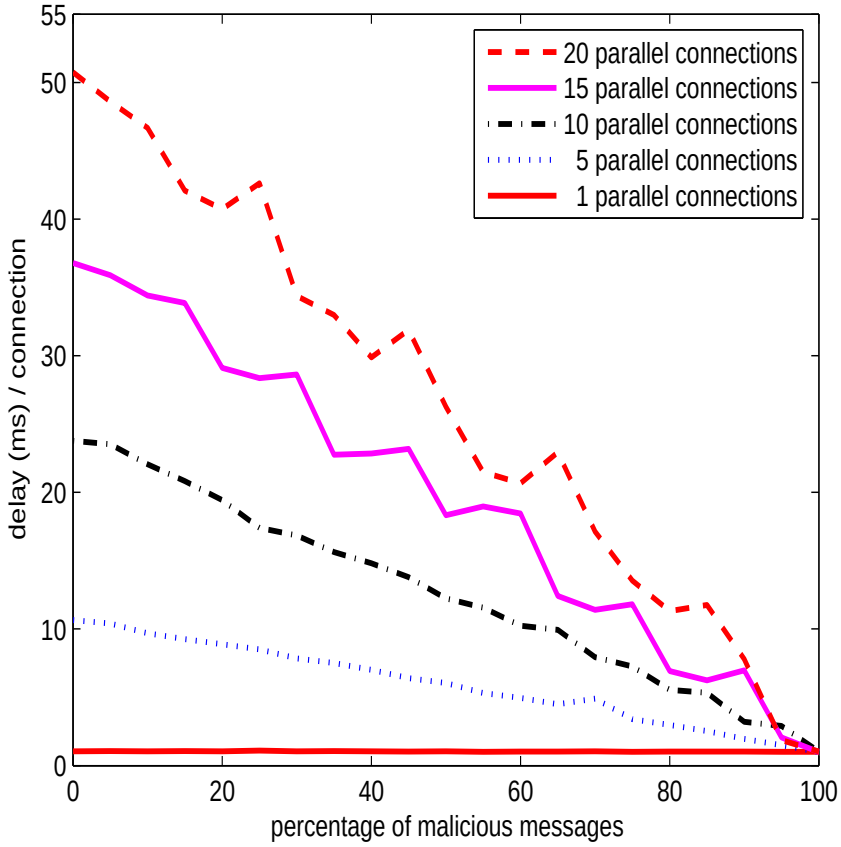


Figure 2.12: Server Performance with Detection System.

referenced documents are stored in the server. We modified the original HTTP load program so the client can inject malicious shellcodes in the requests. As attacks that embed malicious payloads in the HTTP Request-Header have not yet been observed, the original `thttpd` Web server without using our detection system ignores malicious code and returns the requested documents. The server using our detection system returns HTTP status code `400` (bad request) if there is malicious code in the request as the read function returns `-1`.

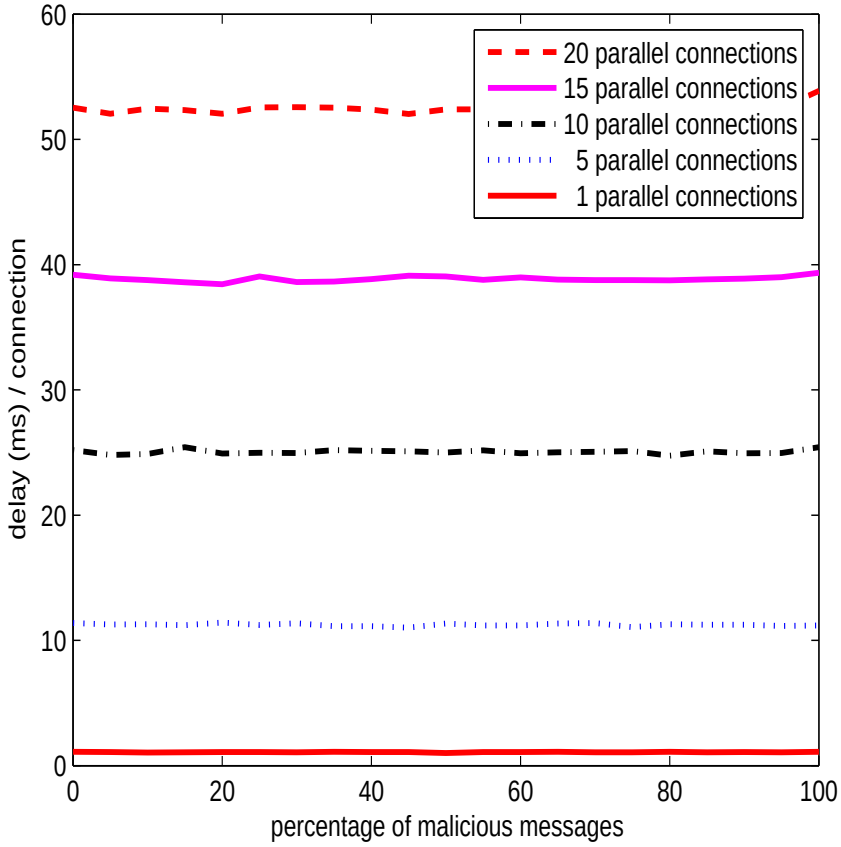


Figure 2.13: Server Performance Without Detection System.

We measured the average connection response latency by running HTTP load for 1000 fetches. Figs. 2.12 and 2.13 show the average connection latency as a function of the percentage of traffic attacking the Web server with and without our detection system, respectively. We randomly choose a shellcode and inject it into requests. We observe that the average latency is stable for the original `thttpd` web server, as malicious code is just ignored. The average latency of the server *with* our system decreases as the percentage of malicious attacks increases, since request errors are returned for connections with shellcodes. Comparing Fig. 2.12 with Fig. 2.13, we observe that the

average response time in the system with detection is only slightly higher than in the system without detection when there are no malicious shellcodes. When there are malicious shellcodes, our prototype system actually reduces the average response time, which means attack requests have little impact on normal requests. For malicious messages, we observe from Fig. 2.8 that if any instruction sequence decoded from input data is malicious, then the processing thereof terminates. We also observe that during emulation of a malicious instruction sequence, it will be discovered before the number of executed instructions reach the threshold. Our prototype's experimental results show that our approach has reasonably low performance overhead.

2.5 Summary

In this chapter, we proposed a new detection methodology in which we fed the target process's virtual memory snapshot into a DBC-based detector. These snapshots were used to instantiate a runtime environment that emulated the process's input data consumption. This environment helped monitoring shellcodes' behaviors. To detect malicious shellcodes, we used the snapshots to examine system calls invoked by shellcodes masquerading as input data, the system call parameters, and the process's execution flow. We implemented a prototype system in Debian Linux. Our experiments with real traces showed our system's strong runtime performance with few false negatives and few false positives.

Although our system had few false negatives, one kind of attack may still evade detection. Our system can detect malicious shellcode that starts from some position in a message. If the attack only uses addresses of `glibc` functions to overwrite the current stack frame's return address [95], then the hijacked control flow will be

directed to the function, not to a position in the message. This kind of attack can evade detection by our system. However, such attacks are difficult to implement as malicious users need to have *complete* knowledge of the target process and use very advanced techniques.

It is possible to further reduce our system's overhead by: (1) using static analysis techniques to analyze input data before or during emulation; and (2) designing a faster instruction decoder. We will address these directions in our future work.

The virtual memory snapshot is very useful at quickly and efficiently detecting malicious shellcodes in the input stream. We believe it can assist other detection approaches. Using this methodology in static analysis is another interesting direction for our future work.

Chapter 3: Malicious Code Detection In JavaScript

3.1 Overview

JavaScript (JS) is a scripting language that is widely used to enrich the functionality of client-side applications, e.g., Web browsers and Adobe Reader. Unfortunately, User experience improvement brought by JS is often accompanied by security risks since JS codes can programmatically access these applications' computational objects. There are several types of JS based attacks against client-side applications [49,86,87], the most dangerous of which exploits target processes' memory errors using shellcodes. Shellcodes are segments of executable codes that are injected into vulnerable processes' address spaces. When they are injected, attackers can execute arbitrary code in the target hosts that can steal sensitive information, furtively download and activate malware, and carry out other nefarious tasks.

A typical example of JS based shellcode injection attacks is to exploit Microsoft Internet Explorer's (IE's) *HTML object memory corruption vulnerability* [112] by an HTML document with a specially crafted JS code embedded. After IE loads the HTML document, the crafted JS code will be parsed, compiled, and then executed, creating many large objects in IE's heap using *heap spraying* [100]. These objects

usually contain shellcodes that will be activated when IE's control flow is hijacked and redirected to them.

Recently, such JS based shellcode injection attacks are growing increasingly severe [26, 86]. This stems from two facts: (1) users do not update their Web browsers in a timely manner yet spend more and more time surfing the Internet [51]; and (2) that numerous browser plug-ins have been released, many of which have vulnerabilities [94]. The deteriorating situation is also witnessed by the recent popularity of "drive-by download" attacks [87] in which users are duped into downloading JS codes that dynamically generate shellcodes and activate them via client-side vulnerabilities.

Unfortunately, existing solutions that detect JS based shellcode injection attacks are insufficient. Some approaches can miss shellcodes whose malicious features are only exhibited during execution while its *continuous* execution requires supporting target process runtime memory information. Some approaches cannot effectively handle attacks in which shellcode is divided into several parts that are interconnected using instructions that redirect control flows. We will present two representative examples in Section 3.2 that illustrate the limitations of existing detection approaches. We provide a review of existing solutions in Section 3.6.

In this chapter, we focus on detecting malicious JS codes that inject shellcode into target applications. We propose a detection system that is called JSGuard and effectively overcomes the problems of existing solutions. Similar to existing work, we assume that the JS interpreter does not have exploitable memory errors and that such errors exist in the application that runs the JS interpreter, plug-ins, or extension modules. We also assume that the application and its plug-ins and extensions are not malware. Therefore, we target malicious codes coming from external untrustable

sources. Although we use a Web browser as an exemplary client-side target application in the rest of this paper and our prototype system is also built within a Web browser, our system can be used to protect other client-side applications such as Adobe Reader.

To the best of our knowledge, our system is the first that creates an emulation environment within the target application process's address space that shadows the address space information during emulation to detect malicious shellcodes in JS codes. We perform such shadowing only when necessary. Our system accurately and comprehensively captures customized application information and real-time runtime memory information in a lightweight manner; stand-alone machine simulators cannot easily obtain this information. Because of our new design, we are now able to fully use JS code stack frame information to optimize performance. From extensive experiments, we find that JSGuard yields very few false negatives and false positives. These results illustrate the promise of our detection methodology. In particular, we make the following contributions:

- We propose a new methodology that can achieve excellent detection completeness. We propose leveraging the JS code execution environment information to instantiate a lightweight emulation environment that reveals and monitors shellcodes' real behaviors. Our emulation environment also enables examination of invoked system calls and their parameters as well as the execution flow to detect malicious shellcodes.
- We propose a multiple-level redundancy reduction technique to reduce detection redundancy. We fully utilize JS code execution environment information to reduce the number of times the detection system is activated and a JS string is checked. This

information includes native methods, stack frames, and properties of each individual JS object.

- We implement JSGuard, a prototype system using the above methodology in Debian Linux with kernel version 2.6.26. We integrate JSGuard into the Firefox 4 Web browser. Our system is adaptive and extensible. It is designed to run in the target process's address space. JSGuard can efficiently fetch and use JS code execution environment information for shellcode detection.

- We conduct extensive experiments based on real traces and thousands of malicious shellcode samples. The experimental results illustrate our malicious JS code detector has high detection accuracy with acceptable overhead.

Chapter Organization. The rest of this chapter is organized as follows. Section 3.2 provides background information and motivating examples. Section 3.3 presents our system design and implementation and Section 3.4 presents detection examples. Section 3.5 evaluates JSGuard's performance. Section 3.6 reviews related works. Section 3.7 concludes.

3.2 Limitations of Existing Work

In this section, we provide a brief background on detecting shellcode in JS objects. Then we use two examples to illustrate the limitations of existing approaches.

3.2.1 Background: Detecting Shellcode in JS Objects

Malicious JS code usually places shellcode into objects generated at runtime and then activates it by exploiting vulnerable applications' memory errors. Therefore,

detecting shellcode in JS objects is critical. Existing detection approaches can be classified into two categories: *content analysis* and *hijack prevention*.

Content Analysis

The approaches in this category are based on scanning JS objects' contents to determine if they contain malicious shellcode. It can be further divided into two subcategories: *static analysis* and *dynamic analysis*. In static analysis, input data are first disassembled and then screened via code-level pattern analysis and matching. Patterns can be complicated signatures or simple heuristics that are obtained from studying known malicious codes. A representative work is Nozzle [90]. Static analysis detection is fast, but it is known that determining program behavior via static analysis is generally undecidable and, often, it can be effectively thwarted by obfuscation techniques [9].

Dynamic analysis based methods detect malicious shellcode by exploiting information generated during shellcode execution. A representive work is [39] that uses `libemu` [66] to detect shellcode in JS strings. The state of the art of dynamic analysis is network-level emulation, which decodes input data into instruction sequences and then emulates their execution [66, 82–84]. If any of them exhibits malicious behavior during emulation, the input data are classified as malicious. Even though network-level emulation can achieve better detection completeness than static analysis, it is still prone to evasion. This is because it assumes that the working shellcodes either are self-contained or use specific memory access behaviors, i.e., their executions are independent of the dynamics of the JS code execution environment. Without knowledge of execution environments, these approaches can be fooled by shellcode whose execution takes advantage of the virtual memory information in the target process.

```
<SCRIPT language"text/javascript">
var object = preBlock;
while(object.length < CHUNK_SIZE){
 object +=preBlock;
}
sprayArray = new Array();
for (i=0; i<OBJECT_NUMBER;i++){
sprayArray[i]=object+shelcode;
}
</SCRIPT>
```

Figure 3.1: Heap Spraying Framework

Hijack Prevention

As suggested by the name, hijack prevention approaches focus on preventing shellcode from being fully executed. This is often achieved by inserting special characters into the shellcode. A representative example is Bubble [52]. In Bubble, a JS string object is divided into multiple units, each 25 bytes long. In each unit, Bubble inserts `0xCC` (i.e., `int 3`) into a randomly selected position. If a JS string object contains shellcode and the shellcode is executed, an interrupt handler will be activated when the control flow reaches the insertion point. However, existing hijack prevention approaches cannot effectively detect shellcodes split into parts that are "interconnected" at runtime via instructions that alter control flow, e.g., `jmp` and `call`.

In the following, we present two examples that can evade content analysis and hijack prevention approaches, respectively. Before presenting these two examples, we first introduce heap spraying, a key technique they use.

3.2.2 Heap Spraying

Heap spraying is an attack technique to thwart address space layout randomization (ASLR) [10, 80], a memory protection mechanism where objects' positions are

randomly arranged in a process's address space. ASLR intents to prevent attackers from easily predicting target object addresses. However, the memory space that can be randomized is often limited, especially in 32-bit operating systems. If we allocate many large objects in the heap, then new objects will likely be placed in a contiguous memory area after a number of allocations, making their positions predictable. This technique is called heap spraying [37, 100].

For example, in MS Windows Vista 32-bit systems, heap randomization shifts the beginning of the heap by up to 2 MB, but all allocations after that are contiguous; hence their addresses can be predicted. If we allocate 100 objects, each of which is 1 MB, then most of them will end up in the same memory range even if ASLR is used [101].

Fig.3.1 shows a JavaScript framework to spray big objects in heaps. CHUNK_SIZE means the size of an object and OBJECT_NUMBER indicates the number of objects to be created. Shellcode is used to launch an attack and preBlock can be a NOP sledge, which is composed of NOP-like instructions, or other kinds of bytes. In the above example, CHUNK_SIZE could be 1MB and OBJECT_NUMBER could be 100. After using the framework to allocate heap space for 100 objects, we can get the objects with predictable positions in heap space, and then we can choose one of them and use its address to overwrite return addresses or function pointers of vulnerable programs with memory errors. And then the vulnerable programs' control flows can be directed to the ojbect's content, and further to the shellcode contained in it.

```
 1  0000  6a7f      push $0x7f
 2  0002  59        pop  %ecx
 3  0003  6a08      push $eaddr;eaddr = 0x08
 4  0005  5e        pop  %esi
 5  0006  46        inc  %esi
 6  0007  4e        dec  %esi
 7  0008  fec1      incb %cl
 8  000a  80460ae2  addb $0xe2, 0xa(%esi)
 9  000e  304c0e0b  xorb %cl, 0xb(%esi, %ecx)
10  0012  00fa      addb %bh, %dl
11  0014
12  .........<encrypted payload>.........
13  0093
```

Figure 3.2: A self-modifying shellcode example. The second column indicates the address of each instruction, the third column indicates the instruction binary code, and the fourth column is the IA-32 assembly code. The shellcode is mapped to address `0x0000`.

```
 1  0000  6a7f      push $0x7f
 2  0002  59        pop  %ecx
 3  0003  6a08      push $eaddr
 4  0005  5e        pop  %esi
 5  0006  46        inc  %esi
 6  0007  4e        dec  %esi
 7  0008  fec1      incb %cl
 8  000a  80460ae2  addb $0xe2,0xa(%esi)
 9  000e  304c0e0b  xorb %cl,0xb(%esi,%ecx)
10  0012  e2fa      loop 0xe
11  000e  304c0e0b  xorb %cl,0xb(%esi,%ecx)
12  0012  e2fa      loop 0xe
13  ....  ........  ......................
```

Figure 3.3: Execution trace of the self-modifying shellcode shown in Fig. 3.2.

3.2.3 Motivating Example 1: Thwart Content Analysis Approaches

Fig. 3.2 shows a shellcode that is modified from an example illustrated in [82]. In the shellcode, `eaddr` is used to calculate the addresses at which the encrypted payload can be accessed. Since heap spraying can make the positions of some heap objects predictable, a skilled attacker can write JS code that first sprays target processes'

heaps, and then inserts the shellcode into the objects whose addresses can be predicted and determined. Because the heap objects' addresses can be predicted, the shellcode addresses can also be predicted, and it is not difficult to assign a value to `eaddr` from which the encrypted part can be accessed. In this example, we assume that the starting address of the shellcode is `0x0000` and `eaddr` is `0x0008`.

This shellcode modifies its instructions at runtime. From address `0x0014` to address `0x0093`, there is an encrypted payload, which often appears to be a meaningless or invalid instruction sequence. When the control flow reaches address `0x000a`, the instruction `addb $0xe2, 0xa(%esi)` will be executed. This instruction modifies the contents of memory at address `0x0012`. After it is executed, the instruction at address `0x0012` will be modified to `loop 0xe`, which forms a backward loop to decrypt instructions from `0x0093` to `0x0014`. The loop is controlled by register `ecx`, which decreases by 1 upon each execution of `loop 0xe`. Within the loop, the instruction at address `0x000e`, `xorb %cl, 0xb(%esi,%ecx)`, is for decryption. It decrypts one byte in each iteration. When `ecx` becomes 0, the loop terminates, the content stored from `0x0093` to `0x0014` is fully decrypted, and the control flow continues to the instruction at address `0x0014`, which is the last decrypted instruction. We can see this from the shellcode execution trace that is shown in Fig. 3.3.

As there is no information that is dynamically generated during shellcode execution, e.g., register values at runtime, static analysis based detection approaches cannot effectively handle the decrypting procedure after the shellcode is interpreted as an instruction sequence; these approaches only see the encrypted payload as a meaningless or invalid instruction sequence. Malicious behaviors that are only exhibited during execution are thus effectively concealed.

The shellcode shown in Fig. 3.2 can also be used to evade detection by current dynamic analysis based tools [39, 66, 82–84]. Given an input stream containing the shellcode shown in Fig. 3.2, network-level emulation based approaches will copy the input stream into a memory space that performs this emulation, and all read/write operations will be performed in the emulated memory space. The real contents of virtual memory units at the addresses calculated from `eaddr` are difficult to obtain. Then the shellcode's encrypted payload cannot be correctly decoded and emulated. In addition, these approaches do not use information stored in other objects to detect shellcode in the current object, which precludes shellcode detection. Since using heap spraying can enable prediction of objects' positions in a heap, it is not difficult for attackers to design shellcode in JS code that makes use of information stored in different objects. For example, if two JS objects have predictable heap positions, an attaker can store shellcode in one and critical information for decryption in the other.

We also notice that some tools based on network-level emulation use heuristics based on the GetPC code [59, 82] in shellcode detection, e.g., [39] uses `libemu` [66]. Besides the aforementioned evasion methods, a skilled attacker can evade detection by these tools by writing shellcode without *call group* instruction or `fstenv` instruction opcodes, e.g., using purely alphanumeric shellcode [69]. Note that the shellcode shown in Fig.3.2 contains no bytes that can be decoded as the GetPC code.

3.2.4 Motivating Example 2: Thwart Hijack Prevention Detections

In this subsection, we discuss designing shellcode that can evade hijack prevention detection.

```
 1  be20000505      movl $addr, %esi          ;%esi<-addr
 2  8976f8          movl %esi, -0x8(%esi)     ;[%esi-8]<-addr
 3  836ef810        subl $0x10, -0x8(%esi)    ;[%esi-8]<-addr-16
 4  31c0            xor %eax, %eax            ;%eax<-0
 5  8846f7          movb %al, -0x9(%esi)      ;[%esi-9]<-0
 6  8946fc          movl %eax, -0x4(%esi)     ;[%esi-4]<-0
 7  b00b            mov $0x0b, %al            ;%eax<-11
 8  8b5ef8          movl -0x8(%esi), %ebx
 9  8d4ef8          leal -0x8(%esi), %ecx
10  8d56fc          leal -0x4(%esi), %edx
11  cd80            int $0x80                 ;call systemcall
12  31db            xor %ebx, %ebx
13  89d8            mov %ebx, %eax
14  40              inc %eax
15  cd80            int $0x80
```

Figure 3.4: Shellcode example that tries to create a shell.

sub-shellcode1

```
1  be20010505  movl $Saddr,%esi
2  8976f8      movl %esi,-0x8(%esi)
3  836ef810    subl $0x10,-0x8(%esi)
4  31c0        xor %eax,%eax
5  eb09        jmp Offset1
```

sub-shellcode2

```
1  8846f7  movb %al,-0x9(%esi)
2  8946fc  movl %eax,-0x4(%esi)
3  b00b    mov  $0x0b,%al
4  8b5ef8  movl -0x8(%esi),%ebx
5  8d4ef8  leal -0x8(%esi),%ecx
6  8d56fc  leal -0x4(%esi),%edx
7  cd80    int  $0x80
8  eb04    jmp  Offset2
```

sub-shellcode3

```
1  31db  xor %ebx,%ebx
2  89d8  mov %ebx,%eax
3  40    inc %eax
4  cd80  int $0x80
```

Figure 3.5: A shellcode can be divided into multiple parts (3 parts here). Each part, denoted by *sub-shellcode*, can be connected to another part by using a `jmp` instruction.

Fig. 3.4 shows a shellcode that opens a root shell in Linux systems. `addr` is a memory address and the memory units pointed by `(addr-16)` are used to store an ASCII sequence `/bin/sh`, which is used by the system call opening a root shell. In this example, we assume that `addr` is `0x05050020`. This shellcode can be divided into three parts as shown in Fig. 3.5. The first part, denoted *sub-shellcode1*, is 16 bytes long. The second part, *sub-shellcode2*, is 21 bytes long. The third part, *sub-shellcode3*, is 7 bytes long. In sub-shellcode1, `Saddr` is `0x05050120` pointing to some part of an object. The memory at address `(Saddr-16)` stores arguments of the system

call used to open a root shell. These include an ASCII sequence `/bin/sh`. At the end of sub-shellcode1, there is an instruction `jmp Offset1`, where `Offset1` is the offset between sub-shellcode1 and sub-shellcode2. This instruction diverts control flow from sub-shellcode1 to sub-shellcode2. In sub-shellcode2, `Offset2` is the offset between sub-shellcode2 and sub-shellcode3. At the end of sub-shellcode2, instruction `jmp Offset2` diverts control flow from sub-shellcode2 to sub-shellcode3.

Using heap spraying, the arguments and the sub-shellcodes can be placed into two different objects whose positions can be predicted. Let the arguments be placed in *object1* and sub-shellcode1, sub-shellcode2, and sub-shellcode3 be placed in *object2*. Because the data structures of *object1* and *object2* are known to the attacker, it is not difficult to arrange and predict the addresses of the arguments and the above three sub-shellcodes in memory.

Consider a Web browser with a certain memory vulnerability that can be exploited to overwrite a function pointer and thus execute arbitrary code. The attacker can use sub-shellcode1's address to overwrite the function pointer. After the web browser's control flow is directed to sub-shellcode1 and the instruction `jmp Offset1` is executed, the control flow can be directed to sub-shellcode2, and then to sub-shellcode3 through the instruction `jmp Offset2`. In this way, the entire shellcode can be executed and a root shell is opened eventually.

Existing hijack prevention approaches fail to detecting such shellcode with certain probabilities. For example, if the three sub-shellcodes are placed at the beginning of three 25-byte blocks in *object2*, the probability that the entire shellcode can evade detection by Bubble [52] is $(25-16)/25 \times (25-21)/25 \times (25-7)/25 = 4.1\%$. This

probability, which implies on average more than 4 attacks can succeed per 100 trials, is quite high.

We comment that this example also illustrates the importance of JS code execution environment information to launch an attack. The arguments of the system calls used by the shellcode embedded in *object2* rely on information stored in *object1*.

These examples clearly demonstrate that it is critical to fully leverage JS code execution environment information to detect shellcode in JS objects. In addition, to guarantee detection completeness, it is also necessary to check all possible instruction sequences that can be decoded.

3.3 System Design And Implementation

In this section, we present the design methodology of JSGuard, its architecture and key components, and implementation. The detailed workflow is further illustrated by examples in Section 3.4.

3.3.1 Design Rationale

Fundamentally, the limitations of existing approaches arise because they do not fully use JS code execution environment information in detection. This motivates our proposal of a new detection approach that overcomes the limitations by efficiently and fully exploiting this information, which includes: (1) the virtual memory contents of the target application that runs the JS interpreter; (2) the context information of the host system, e.g., system call information; and (3) the JS code semantics, which includes stack frames, native method information, properties of JS objects, etc.

This environment information is used at the core of JSGuard in the following two ways:

- *Creating A Virtual Execution Environment for Detection.* When our detection system is activated, the real environment information at that moment is used to instantiate a virtual environment where potentially malicious JS strings are executed and monitored. Such real environment information is critical for observing the real behaviors of possible shellcodes, as they exhibit real execution flow. In malicious shellcodes, process state information can be used to redirect the execution flow, e.g., for encryption or decryption as illustrated in Example 1, or it can be leveraged to compute arguments for system calls to perform malicious actions, as illustrated in Example 2. Besides, some binary code existing in the virtual memory could also be utilized to launch attacks. For example, a libc function, *read()*, which can be used to read sensitive files from a vulnerable system. A shellcode can compute arguments for *read()* first and then use the instruction *jmp* to direct control flow to *read()*. In this case, the shellcode itself does not contain any instructions to call system calls to read sensitive files so that its malicious characteristics can be hiden. Without precise virtual memory information, the shellcode's execution flow or characteristics can be changed and its malicious behavior may not be captured.

In addition, using the real environment information also enable leveraging a target system's binary code to emulate system calls appearing in a decoded instruction sequence, especially those that do not change processes' states but can be used to take part in computation of shellcode. This kind of emulation can help us observe more possible shellcode behaviors.

- *Facilitating Multiple-level Redundancy Reduction.* We propose reducing detection overhead from the following three levels. First, we should activate our detection system as infrequently as possible. Once our detection system is activated, then the number of JS objects to be checked should be minimal. Finally, if one JS object is to be checked, checking should only happen when necessary (e.g., after mutable objects have changed).

At the core of JSGuard, we achieve this multiple-level redundancy reduction by using the following execution environment information: native methods, stack frames, and properties of JS objects. Native methods are used by JS code to call native system functions written in C/C++. They are "gateways" between the JS interpreter and the external components. In JS code, any calls to external components pass through native methods. After the JS code calls an external component, the control flow is redirected from the JS interpreter to the external component. Our system is activated right before the control flow enters the external components, since their vulnerabilities are targets for malicious code. Native method information is used to distinguish built-in JS native methods that are secure (as we assume the JS interpreter is secure) from external ones that are written by the users and used to call their external components. To reduce activation time, we only activate our detection system before the external native methods are called. We further use information stored in current stack frames to reduce the number of objects to be checked. The JS interpreter maintains a stack frame for each JS function being interpreted including its origin information. By searching the current stack frames, we can determine if JS functions are internal functions or from trustable sites. If not, the objects generated in the JS functions are checked.

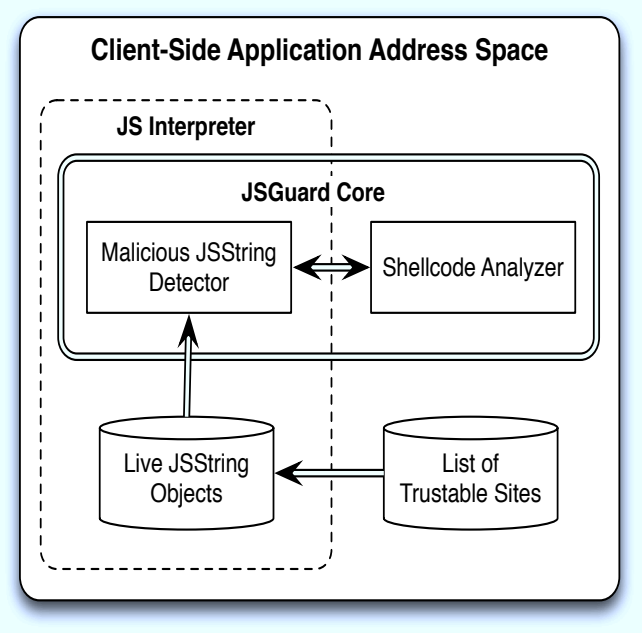


Figure 3.6: The overall architecture of JSGuard.

3.3.2 JSGuard Architecture and Key Components

JSGuard aims to detect whether JS codes embedded in web pages generate malicious shellcode. If a JS code generates malicious shellcode at runtime, it will be considered malicious. Similar to other works [39, 52], JSGuard focuses on detecting shellcode in JS string objects since it is difficult to insert shellcode in other types of objects.

As illustrated in Fig. 3.6, JSGuard resides in the address space of the target process. Besides the *JSGuard core* that is the core functionality block that performs detection, JSGuard also involves the JS interpreter and a list of trustable sites. The JS interpreter determines the origins of JS functions being interpreted; only those from external untrusted sites are further checked by the JSGuard core. The list can

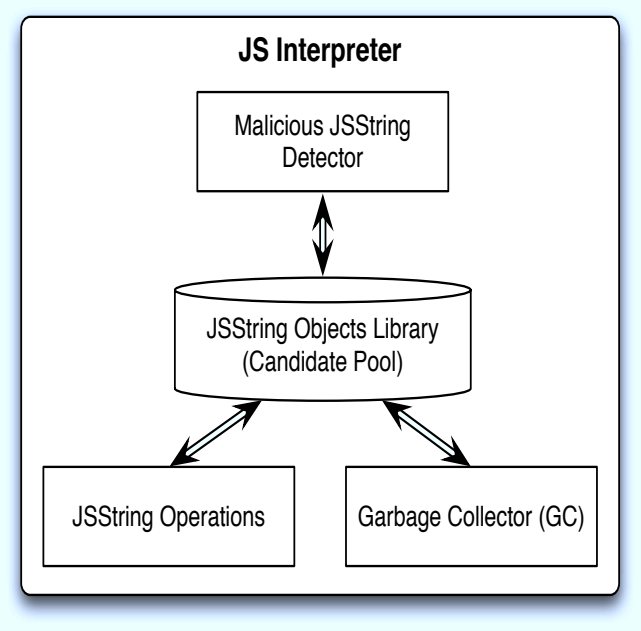


Figure 3.7: Malicious JS string detector takes JS strings from a pool maintained by string-related operations and the JS interpreter's GC.

```
#define   BENIGN      0
#define   MALICIOUS   1

struct JSString {
    size_t          length;
    jschar          *chars;
};

int maliciousJSStringDetector(checkinglist) {
  JSString *string;
  check = checkinglist;
  while(check != NULL) {
    string = check->string;
    if(ShellcodeAnalyzer(string->chars) == MALICIOUS)
        return MALICIOUS;
    check=check->next;
  }
  checkinglist = NULL;
  return BENIGN;
}
```

Figure 3.8: Workflow of malicious JS string detector.

be maintained manually or automatically. New sites can be added into it according to the detection results of JSGuard on them as well as the user's knowledge. These sites can be those that are often visited by the user, e.g., the site of the company he or she is working for. They can be also those maintained by reputable companies or organizations, such as Google, Microsoft, CNN, etc. If users are concerned about a trustable site, they can always force JSGuard to check it. The entries of the list can be host names or domain names of trustable companies and organizations.

As shown in Fig. 3.6, JSGuard core has two key components, namely, the malicious JS string detector and the shellcode analyzer. The malicious JS string detector runs in the JS interpreter. It prepares JS strings to be checked at runtime and then feeds them to the shellcode analyzer. The shellcode analyzer checks if an input object's content contains malicious shellcode or a part thereof and reports the results back to malicious JS string detector. If a malicious JS string is found, interpretation stops; otherwise, it continues. In the following, we detail the malicious JS string detector and the shellcode analyzer.

Malicious JavaScript String Detector

As shown in Fig. 3.7, the detector retrieves and checks JS strings from a checking list, which contains all JS strings that might have malicious shellcode. The checking list is maintained by instrumenting string-related operations and the JS interpreter's garbage collector (GC). For example, when a new JS string is created, it is inserted into the checking list; when the GC reclaims a JS string, the string will be removed from the checking list after its content is zeroed.

The basic workflow of the malicious JavaScript string detector is shown in Fig. 3.8. The function `maliciousJSStringDetector()` has one input `checkinglist`. When

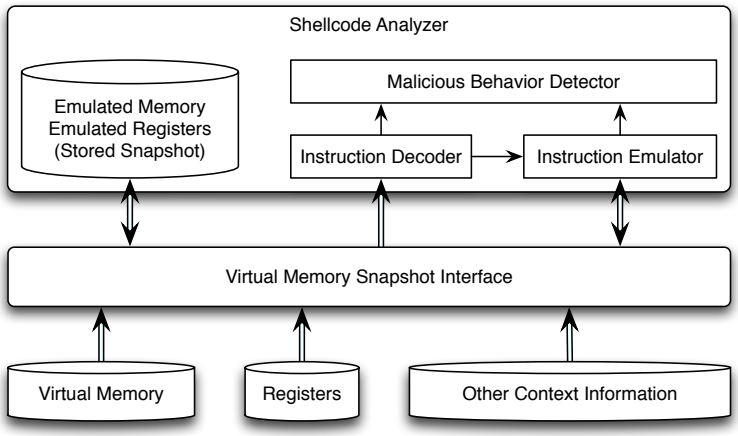


Figure 3.9: Shellcode analyzer architecture.

called, it scans all strings in `checkinglist` and feeds them to `shellcodeAnalyzer()`, which detects malicious shellcode in JS string contents. If `shellcodeAnalyzer()` finds a JS string containing malicious shellcode, it returns `MALICIOUS` to `maliciousJSStringDetector()`. Then `maliciousJSStringDetector()` stops checking the remaining JS strings in `checkinglist` and returns `MALICIOUS` to the JS interpreter. The interpreter in turn stops interpreting the JS code. If no JS string is found to be malicious, then `maliciousJSStringDetector()` returns `BENIGN` to the interpreter. The interpreter continues interpreting the JS code.

`checkinglist` contains the JS strings to be checked. Every time a JS string is generated, all current stack frames are checked. If there are any JS functions from external untrusted sites, then we add the JS string to `checkinglist`. We do so because only JS codes from external untrusted sites attempt to generate shellcode that exploits target applications' vulnerabilities. As JS strings are immutable objects,

```
#define MALICIOUS 1
#define BENIGN    0
#define MALICIOUS_SEQUENCE 1
#define BENIGN_SEQUENCE    0

int ShellcodeAnalyzer(base_addr, base_size) {
  for (i = 0; i< base_size; i++)
    if (MaliciousInstructionSeq(base_addr + i))
      return MALICIOUS;
  return BENIGN;
}

int MaliciousInstructionSeq(addr){
  InitializeEmulationEnvironment();
  instruction = InstructionDecoder(addr);
  if (End(instruction)) return BENIGN_SEQUENCE;
  instruction.exe_depth = 1;
  while (instruction) {
    if (MaliciousSystemCall(instruction))
       if (instruction.exe_depth > exe_depth_threshold)
         return MALICIOUS_SEQUENCE;
       InstructionEmulator(instruction);
       UpdateEmulationEnvironment();
       target = ComputeTarget(instruction);
       prev_instruction = instruction;
       instruction = InstructionDecoder(target);
       if (End(instruction)) break;
       SetExecutionDepth(instruction, prevInstruction);
  }
  return BENIGN_SEQUENCE;
}
```

Figure 3.10: Workflow of shellcode analyzer.

we can safely remove the strings from `checkinglist` after they have been checked once [52].

In the JSGuard core, `maliciousJSStringDetector()` is called immediately before JS code calls an external component. Recall that the malicious JS code is used to exploit memory errors in applications using the JS interpreter or their plugins. If the JS code does not call any external components provided by the applications or the plugins, then any JS strings will not be used to hijack control flows of target applications, even they have malicious shellcode. So, it is not necessary to activate the malicious JS strings detector when there is no calling to external components. Because the execution of the malicious JS string detector also needs to take time

to do checking, it helps reducing extra overhead to activate the malicious JS string detector immediately before the JS code calls an external component.

Shellcode Analyzer

The shellcode analyzer architecture is shown in Fig. 3.9. This module consists of an instruction decoder, an instruction emulator, a malicious behavior detector, an emulated memory system, and emulated registers.

Given a position in a JS string content, the instruction decoder decodes instructions starting at that position and sends each decoded instruction to the emulator. For each instruction the emulator receives, it emulates the execution thereof, for which the emulated memory system and registers provide a virtual runtime environment. The JS code execution environment information provided to the shellcode analyzer includes the target process's address space, current registers, and other context information as necessary. The emulator executes each instruction sequence and the malicious behavior detector determines whether there is any malicious behavior. If any such behavior is detected, then the instruction sequence is considered malicious. As a result, the shellcode analyzer concludes there is malicious shellcode in the content buffer. Hence the JS string object is considered malicious.

During instruction sequence emulation, if there is an instruction that reads memory, the memory values are first fetched from the real memory units in the target process's address space. Next, these values are stored in the emulated memory system. Future read operations to the same memory units will be directed to the emulated memory system. If there is a write memory operation, it will be directed to the emulated memory system. The write operation is never performed on the corresponding

real memory units in the target process's address space to avoid disturbing "normal" JS code execution.

The shellcode analyzer workflow is shown in Fig. 3.10. From each position of the input data, the shellcode analyzer uses the target process's virtual memory information to emulate the execution of the decoded instruction sequence. There are two input parameters for `ShellcodeAnalyzer()`: (1) `base_address`, the starting address of the input data to be analyzed; and (2) `base_size`, the size of the input data.

Precise virtual memory information ensures the shellcode's execution flow is not interrupted. If an instruction sequence is malicious shellcode, all of the instructions it uses are eventually reached during emulation via the execution flow, including the shellcode's, those generated by previous messages, and those from the libraries loaded into the target process's address space. Thus emulation within the shellcode analyzer with a virtual memory information can be used to accurately observe the behaviors of polymorphic and metamorphic shellcodes as well as shellcodes distributed over multiple objects.

The key function of the shellcode analyzer is `MaliciousInstructionSeq()`, which detects a malicious instruction sequence. The workflow of `MaliciousInstruction-Seq()` is shown in lines 13–31 in Fig. 3.10. The `while` loop from line 18 to line 29 in Fig. 3.10 emulates a sequence of instructions, which continues until one of the following occurs: (1) a malicious behavior is detected; (2) a privileged or invalid instruction is encountered;[2] (3) an illegal memory access occurs; or (4) the number of executed instructions exceeds a threshold.

[2]Privileged instructions can only be executed in kernel mode while shellcodes normally runs in user mode. If a shellcode contains a privileged instruction, an exception occurs.

In our system, a *malicious behavior* is defined as a *malicious* system call invocation. In Linux and Microsoft Windows systems, not all system calls can compromise the target host's security. This depends on system call numbers and parameters, which are stored in registers before system call instructions are executed. Through the JS code execution environment information interface, the system call number and its parameters can be accurately obtained to determine if the system call invocation is intended to compromise the host's security. For example, in Linux, the system call number `11` corresponds to the system function `execve`, which executes a program. During instruction emulation, if the instruction is a system call instruction and the value of the emulated `eax` is `11`, then the system call number is `11`. After checking parameters stored in other emulated registers and the emulated memory system, if its first parameter is `/bin/sh`, then we can conclude that the instruction tries to open a root shell. In this case, the system call instruction will be considered malicious.

Note that shellcodes normally need several instructions to initialize system call parameters. Therefore, we also use the `exe_depth` of an instruction that invokes a system call to decrease false positives. An instruction's `exe_depth` is defined as the number of instructions from the starting point to it during emulation of an instruction sequence. For example, suppose that a statement S in a `for` loop is executed 100 times. Then the execution depth of S is `2` (the `for` statement and S).

Our detection system can also leverage heuristics used in current network-level emulation tools [39, 66, 82–84] to detect shellcode in JS strings during emulation. However, these heuristics are confined to detect particular types of shellcode that

exhibit self-decrypting behavior [39, 66, 82, 83] or match specific memory access patterns [84]. In addition, as we said in section 3.2, they are not effective in detecting shellcode that takes full advantage of JS code execution environment information.

3.3.3 Implementation

The JSGuard prototype system is implemented in Debian Linux with kernel version 2.6.26 using C and C++ with gcc 4.3.2. The key component is the JSGuard core, which comprises two major parts. The first part is a modified JS interpreter integrated with the malicious JS string detector. This part is based on the SpiderMonkey JS interpreter [102], which is used in various Mozilla products including Firefox. The second part is the shellcode analyzer module. We implement it as a C library in Debian Linux system. When it is called by the malicious JS string detector, the shellcode analyzer module will be loaded into the address space of the application that runs the JS interpreter. We also implement a Firefox extension that maintains the list of trustable sites, which is loaded into Firefox address space when Firefox is executed.

In the following, we first present the implementation details of the malicious JS string detector, and then the shellcode analyzer module.

Modified JS Interpreter

In our modified JS interpreter, we implement a malicious JS string detector, which scans JS string objects from a `checkinglist` and then calls the shellcode analyzer to determine if they have malicious content. The `checkinglist` is maintained by the code that we add into all functions related to JS string operations. First, we instrument all functions related to JS string object creation. In this way, we can track

all JS string objects generated during execution of the external JS code. Populating the `checkinglist` with all strings fundamentally guarantees the completeness of our detection. Second, before adding a JS string to `checkinglist`, we also use the list of trustable sites and current stack frames to decide if the JS string should be added to `checkinglist`. If all JS functions being interpreted are from trustable sites or internal JS functions, the string will not be added to `checkinglist`; otherwise, it will.

After analyzing the source code of the SpiderMonkey JS interpreter, we find all call points that invoke native methods, and insert calls to the malicious JS string detector at these points. Since the JS interpreter also uses native methods to implement some built-in JS class methods, we check if a native call is calling a JS built-in method at native call points. If this is the case, we do not activate the malicious JS string detector; otherwise, we activate it. This is due to our assumption that the JS interpreter has no exploitable memory errors. The native methods for JS built-in class methods are parts of the JS interpreter, so they do not have exploitable memory errors. However, when control flow leaves the JS interpreter to external functions, the malicious JS string detector will be activated to check all JS strings in the `checkinglist`.

We also modify the JS interpreter's garbage collector to maintain the `checkinglist`, and integrate the modified JS interpreter into the Firefox 4 Web browser.

Shellcode Analyzer

The shellcode analyzer prototype focuses on the IA-32 architecture and Linux operating systems. Its core component is an instruction emulator. In the following, we present its implementation.

Our instruction emulator can *interpret* all IA-32 instructions and *emulate* a subset thereof, including all general-purpose instructions and the FPU instructions that are used to obtain injected shellcodes' absolute addresses [59, 82]. In addition, we emulate some system instructions such as `rtdsc`. This subset contains all instructions used by known malicious shellcodes in useful computation of malicious attacks [82]. We consider our implemented subset sufficient, though extensions are feasible. When an unimplemented instruction is encountered in emulation, if it is not a privileged instruction, the control flow skips it and moves to the next instruction; otherwise, emulation stops. When encountering a system call instruction (`sysenter` or `int 0x80`), the shellcode analyzer will determine, with the parameters stored in the emulated memory/register system, whether it is one of 36 system calls that can be used to compromise the Linux system [73]. Besides these "malicious" system calls, we also use the `exe_depth` threshold to determine if the instruction truly tries to compromise the host's security; we set the threshold to 10 since most unencrypted malicious shellcodes have at least 10 instructions [83, 114]. To avoid an endless loop during instruction sequence emulation decoded from a position of a JS string's content, we set the threshold to 8000 for the number of executed instructions. According to current research, such threshold is sufficient in order to detect malicious shellcodes [82, 83].

We also implement an emulated memory system, an instruction decoder built on the Bastard project's `libdisasm` with version 0.23-pre [2], and a JS code execution environment information interface that is used to fetch the application's virtual memory information and the host context information.

3.4 Detection Example

In this section, we illustrate effectiveness of our detection system by presenting the detection procedure of Example 2 in Section 2.3. We note that Example 1 in Section 2.2 can similarly be detected by our system.

Assume that the attacker tries to exploit a Firefox external component in Linux using a malicious JS code. He first uses heap spraying to allocate many large JS objects, then inserts the arguments and the three sub-shellcodes, as shown in Fig. 3.5, into two objects. We denote these two objects by *object1* and *object2*. The objects are allocated by the attacker in two contiguous memory areas and their addresses are predictable. Let their addresses be `0x05250020` and `0x05350020`, respectively. The JS code places the arguments in *object1* with `Saddr` set to `0x05250084` and places sub-shellcode1, sub-shellcode2 and sub-shellcode3 into *object2* with their addresses set to `0x05350084`, `0x0535009D` and `0x053500B6` respectively. Then the offset between sub-shellcode1 and sub-shellcode2 is 9 and the offset between sub-shellcode2 and sub-shellcode3 is 4. Hence, in Fig. 3.5, `Saddr` is `0x05250084`, `Offset1` is 9, and `Offset2` is 4.

The attack starts when the three sub-shellcodes are ready in the heap. The JS code calls the vulnerable component. Before control flow is diverted from the JS interpreter to the external component, the JS interpreter with JSGuard invokes `maliciousJSStringDetector()` to check whether there are malicious JS strings arranged in the heap. `maliciousJSStringDetector()` will scan JS strings in `checkinglist` and send them one by one to the shellcode analyzer. At a certain moment, the shellcode analyzer receives the content of *object2*.

The shellcode analyzer decodes every possible instruction sequence starting from each byte position of the content, and then executes it. Each instruction in the instruction sequence starting from the address `0x05350084` will be decoded and then executed. When the instruction `jmp Offset1`, i.e., jumping to `0x0535009D`, is decoded and executed, the shellcode analyzer will follow the control flow and begin to decode instructions starting from `0x0535009D` and execute them. Note `0x0535009D` is the starting address of the sub-shellcode2 instruction sequence. In this way, the instruction sequence of sub-shellcode2 is discovered and executed. When system call instruction `int $0x80` is executed, we can obtain its parameters since the contents of the emulated registers/memory system precisely reflect the run-time changes during the emulation. The shellcode analyzer discovers that this system call instruction tries to open a root shell. Meanwhile, this instruction's `exe_depth` exceeds the threshold. Thus this system call instruction will be considered malicious. As a result, the entire emulated instruction sequence is considered malicious. The shellcode analyzer concludes that *object2* content contains malicious shellcode and returns to `maliciousJSStringDetector()`. When the malicious JS string detector receives `MALICIOUS` from the shellcode analyzer, it in turn concludes that *object2* is a malicious JS string and the JS code being interpreted is malicious. It raises an exception and stops interpreting JS code.

3.5 Evaluation

We conduct extensive experiments to evaluate JSGuard, particularly its detection effectiveness and runtime overhead. We do so on a HP Pavilion a815n with one Intel Pentium 4 3.06 GHz CPU and 1 GB RAM. The computer is connected to a

university campus network through 100 Mbps Ethernet; it runs Debian Linux with kernel version 2.6.26.

3.5.1 Effectiveness

Detection effectiveness is measured by false positives and false negatives.

• *False Positive: 0 out of 2000.* We implement a Firefox extension that automatically fetches websites listed in a file. We set the time interval between two fetches to be 50 s, which is generally sufficient for JS codes embedded in a webpage to be fully executed. Every 50 s, the extension iteratively reads a URL from the file and then loads the web page in a browser window. We construct a benign URL list containing 2000 URLs taken from the Alexa ranking of global top sites [3]. These are real websites with various content and Web applications. JSGuard classifies all of them as benign.

• *False Negative: 0 out of 5051.* We collect 8 real world malicious webpages containing JS code that generate shellcode to launch attacks and we also collect 51 plain malicious shellcodes from the Internet. All of them target Linux systems. Based on the 51 plain shellcodes, we use the following tools to generate 5000 polymorphic or/and metamorphic malicious shellcodes: the Metasploit project's JumpCallAdditive, Pex, PexFnstenvMov, PexFnstenvSub, and ShikataGaNai [104] as well as ADMmutate [68] and TAPiON [6], which are also used in other shellcode detection tools [82, 83, 113, 114] to test their effectiveness. We then create 5051 JS codes that generate these malicious shellcodes at runtime and invoke native methods that are not built in to the JS interpreter. For example, the JS method `document.write()` eventually calls a native method. Finally we craft 5051 malicious web pages with

Firefox Version	Total Time (s)	Time/Page (s)	Overhead/Page(s)
Original Version	491.953	1.63984	N/A
Trustable List Only	492.254	1.64085	0.00101

Table 3.1: The overhead of checking trustable sites only. "Original version" is Firefox without our system, and "Trustable List Only" is Firefox with our detection system enabled (JSGuard core disabled).

these malicious JS codes. We put these 5051 malicious web pages and the 8 real world malicious web pages on our internal web server and we visit them using Firefox with JSGuard on a client computer. JSGuard classifies all of them as malicious. We also write two heap spraying JS codes, dynamically generate the two shellcode examples presented in Section 2.2 and 2.3, and feed them to JSGuard. It correctly classifies them as malicious.

3.5.2 Overhead

To measure JSGuard's overhead, we use two versions of Firefox 4: one integrated with JSGuard and an "original" version without JSGuard. We use the top 100 most popular web sites as described by Alexa [3] as the testing dataset. In our experiments, we visit each web site three times using both versions of Firefox, respectively. The time we measured, *rendering time*, includes the times for downloading of a webpage over the Internet, parsing and rendering of the page, and executing all JS codes in the webpage.

We performed three types of experiments to measure overhead incurred: (1) by only checking trustable sites; (2) by only using the JSGuard core functionality block; and (3) by using entire JSGuard system.

In the first experiment, we disable JSGuard and measure the overhead purely incurred by checking trustable sites. When a user accesses a webpage from a trustable site, ideally, all of JS codes contained in the webpage and all of JS functions called during runtime should be from trustable sites or internal JS functions, and then all of JS strings generated during runtime will not be put into `checkinglist`, and thus JSGuard will never be activated. Therefore, the only overhead incurred by our detection system is the time used to checking trustable sites. To conduct the first experiment, We use the top 10,000 most popular web sites from Alexa [3] to form a list of trustable sites. And then we use it to check if a site is trustable or not. We think the list of such size is sufficiently large to common users. The experiment results are shown in Table 3.1, which shows that this overhead is very low. Thus our detection system has little impact on the rendering time when all JS functions called during runtime are internal ones or from trustable sites.

The second experiment measures the overhead purely incurred by running JSGuard core *without* checking trustable sites. This reflects an extreme case in which every site the user visits is assumed to be malicious, i.e., every JS string is put into `checkinglist` so long as all interpreted JS functions are from external sites. As shown in Table 3.2, the average overhead incurred by JSGuard core is 3.865 s. Note that this result reflects performance in the worst-case scenario with a low-end machine. Indeed, studies have shown that overall user frustration increases when page load times exceed 8–10 s [12,76]. Therefore, performance is acceptable in this extreme case.

The third experiment measures the overhead incurred by the entire JSGuard system. We construct a random list of 50 trustable sites from our testing dataset. The

Firefox Version	Total Time (s)	Time/Page (s)	Overhead/Page (s)
Original Version	491.953	1.63984	N/A
JSGuard Core Only	1651.451	5.50483	3.86499

Table 3.2: The overhead purely incurred by the JSGuard core block. "Original Version" is Firefox without our system, and "JSGuard Core Only" is Firefox with our system enabled (checking trustable sites disabled).

Firefox Version	Total Time (s)	Time/Page (s)	Overhead/Page (s)
Original Version	491.953	1.63984	N/A
With JSGuard	753.059	2.51019	0.87035

Table 3.3: The overhead incurred by JSGuard. "Original Version" is Firefox without our system, and the version with JSGuard is Firefox with our entire JSGuard system enabled.

remaining 50 sites in our testing dataset are thus considered untrustable. The experiment results are shown in Table 3.3. The average time to render one web page is $\sim$2.51019 s.

3.6 Related Work

Detecting shellcode in JS objects is essential to protect vulnerable applications from JS based shellcode injection attacks. As pointed out in Section 3.2.1, most existing shellcode detection approaches fall into two categories: *content analysis* and *hijack prevention*.

Content analysis is particularly popular in detecting shellcode from network messages. In [107], Toth and Kruegel proposed identifying exploit code by detecting NOP sleds. However, attacks can bypass this detection technique by either *not* including

NOP sleds or by using polymorphic techniques [18, 36, 68]. Chritodorescu and colleagues [21, 22] proposed techniques to detect malicious patterns in executables using semantic heuristics. Lakhotia and Eric in [65] used content analysis techniques to detect obfuscated calls in binaries. Chinchani and van den Berg proposed a rule-based scheme in [18]. Wang et al. proposed SigFree [114] that checks if network packets contain malicious codes using "push and call" patterns and the number of useful instructions in the longest possible execution chain. These methods are based on static analysis. Although they are efficient in detecting shellcode, they still can be thwarted by using binary obfuscation [9]. To improve detection completeness, Polychronakis et al. proposed a new network-level emulation approach [82, 83] to detect polymorphic shellcode. Gene [84] used network-level emulation with specific memory access pattern heuristics to detect shellcode for MS-Windows systems. Gu et al. proposed the virtual memory snapshot based emulation approach in end systems to detect shellcode in network messages before they are processed by network server programs [55].ShellOS provides a framework leveraging hardware visualization to detect shellcode [99]. It requires users to dump the entire target process's states and load them into ShellOS in order to construct an emulation environment. A powerful shellcode analyzer named "Shellzer" is proposed in [50]. It conducts analysis by instrumenting each instruction, which may incur undesirable overhead for online detection.

All these approaches are useful for detecting shellcode in network messages, but they are not directly applicable to detecting shellcode in JS strings, as such shellcode is not transmitted in its binary form. Instead, each byte of the shellcode is transmitted using its ASCII representation. In general, ASCII character sequences cannot

be successfully decoded into the corresponding shellcode instruction sequences [39], though this is sometimes possible [78]. Nozzle is a well known JS shellcode attack detection tool. It scans a heap object, interprets the object content to build a control flow graph (CFG), and then use the CFG to check weather the content contains shellcode [90]. Egele et al. propose an approach that uses `libemu` [66] to check if the content of a JS string contains a sufficiently long valid instruction sequence using network-level emulation and GetPC code based heuristics. Hijack prevention based approaches can be used before or during shellcode execution. Such approaches include randomization [8, 10, 11, 62, 80], OS extension [7, 61] and flow tracking techniques [77, 89]. In general, these approaches have good detection completeness due to their extensive use of context information. However, their troubleshooting to find out the root cause is inefficient [114], which often requires heavy playback or log analysis. Recently, Gadaleta et al. propose Bubble [52], a lightweight approach that encumbers complete execution of injected shellcode.

Recently, several machine learning based systems were proposed to detect malicious JS code. Zozzle applies Bayesian classification to hierarchical features of the JavaScript abstract syntax tree to identify syntax elements that strongly predict malware [30]. Jsand [26] emulates JS code in a virtual browser environment using machine learning methods to capture malicious features. Prophiler [15] constructs a filter that can quickly discard benign pages and forward potentially malicious pages to heavyweight analysis tools. JSGuard can complement these systems by providing malicious code training samples.

We note that some works like Cujo [91] and Blade [67] can also prevent drive-by-download attacks. However, their focus differs from ours, which is malicious shellcode

detection in JS code. These works cannot prevent in-memory execution of injected shellcode. We are also aware that tools like [16, 44] have been proposed to audit JS activities, but they are not malicious shellcode detection systems.

3.7 Summary

In this chapter, we have proposed a new methodology to detect JS shellcode that fully uses JS code execution environment information in an efficient manner. Following the methodology, we implemented JSGuard and a prototype malicious JS code detection system based on it in Debian Linux. Extensive experiments with real traces and thousands of malicious shellcodes illustrate our detection system's performance with acceptable overhead and very few false negatives or false positives. Our very low false negative and false positive rates validated our methodology's promise for this purpose.

Chapter 4: Malicious Code Detection In Smartphones

4.1 Overview

Smartphones are becoming increasingly popular. According to Nielsen data from July 2012, 54.9% of U.S. mobile users own smartphones and two out of three new handset buyers bought a smartphone in the past three months [105]. This is mainly due to smartphones' all-in-one features combining communication and computing functions, enabling a wide variety of applications (also referred to as *apps*).

As smartphones become more widespread, their users' privacy and security become critical issues. For example, a *Wall Street Journal* study of iOS and Android apps revealed that 46–55% of smartphone apps transmit users' private information such as location and device ID over networks without users' awareness or consent [92]. Worse, many users are enticed to download and run smartphone apps without carefully understanding the consequences of accepting permissions prompted before installation. This can easily lead to installation of malicious apps. In fact, Trend Micro reports that over 25,000 Android malware samples were found in June 2012 alone [108].

Private (or sensitive) information on smartphones comes from various sources, including sources originating from smartphones themselves and sources received from the Internet. Fig. 4.1 illustrates this sensitive information and its leakage. On one

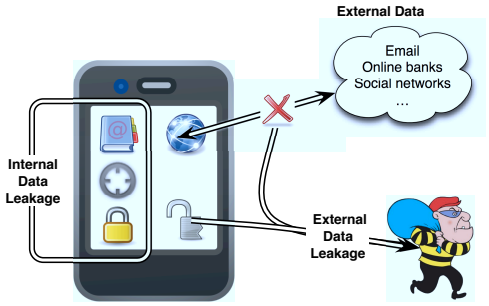


Figure 4.1: Information leakage in smartphones

hand, smartphones themselves generate sensitive information such as photos, GPS locations, and device identifiers (IMEIs/EIDs). On the other hand, smartphones can receive sensitive information from a plethora of possible sources over the Internet. For example, users may check their bank accounts via a browser or a bank-provided app. Similarly, smartphones are often used for checking email contents from servers such as Gmail or Yahoo! Mail. Privacy can be easily invaded if sensitive data from one source were sent to another irrelevant destination, let alone an attacker-controlled one.

Prior work such as TaintDroid [41] has demonstrated that Information Flow Tracking (IFT) mechanisms can be leveraged to detect leakage of sensitive information. Specifically, TaintDroid extended the Dalvik virtual machine (VM) to tag smartphone data using 32 possible types based on its origin, to propagate the tags during program execution, and to raise alerts upon detection of sending out sensitive data. While effective at tracking a limited number of sensitive data sources, TaintDroid's scheme cannot scale to handle sensitive data received from many possible external sources such as Citibank, Bank of America, and Gmail. For example, after being normally used for accessing a bank website, a malicious browser may send the bank

information to some predefined attacker's server. In this scenario, TaintDroid tags the account information data as "network", indicating it is from the network, and may present a notice to the user before the data were sent over to another site. However, with the general tag "network", rather than the precise source of the data, it is difficult for users to determine whether it is legitimate to send the data.

Therefore, it is imperative to design an information flow tracking system for tracking sensitive data from a large number of possible internal and external sources on smartphones and informing users with alerts of relevant source and destination information before data are sent out.

However, our key challenge is tracking a vast number of information sources given limited resources. Smartphone data can originate from many sources such as online banks, social networking websites, etc. Any smartphone based IFT system needs to track all these sources in each data tag. Yet tag capacity is limited, e.g., 32 bits. A naïve approach to solve this challenge is using one bit to track each source. However, this requires many bits to track all sources, far more bits than the tag length. This approach leads to enormous tag overhead. Also, tracking data during program execution is not lightweight since every statement requires tag propagation. The runtime overhead may be even higher if a compressed tag system (for saving space overhead) is used. The bottom line is that our tracking system cannot be so slow as to exceed users' expectation on response time.

This chapter proposes D2Taint, a novel IFT tagging strategy using differentiated and dynamic tracking. We partition information sources into disjoint *classes* that correspond to different information sensitivities. We design a flexible tag structure that stores these classes and their information sources in fixed-length tags. Our tag

structure updates itself on-the-fly based on time-varying received information sources. Our tagging strategy enables us to track at runtime numerous information sources in multiple classes and rapidly detect information leakage from any of these sources.

In summary, our contributions are as follows:

• We propose a novel IFT strategy using differentiated and dynamic tagging. With its flexibility, our tag scheme can handle different numbers of information sources, ranging from a few to thousands;

• We leverage our tagging strategy to design and implement D2Taint, an IFT system using differentiated and dynamic tagging on Nexus One smartphones running Android 2.2;

• We experimentally evaluate D2Taint's effectiveness with 84 real-world apps downloaded from Google Play [54]. D2Taint reports that 71 out of the 84 evaluated apps leak either internal or external information to third-party destinations and 12 out of these 71 apps leak highly sensitive internal information. D2Taint also detects considerable external information leakage in 33 out of these 71 apps and provides detailed information about multiple external sources, which is much more than TaintDroid can provide. Furthermore, we evaluate the performance of D2Taint and our dynamic tag system. Our results show that D2Taint is effective at handling a large number of sources and it can dynamically adjust its tag scheme based on users' behavior during program execution, with moderate space and runtime overheads. For instance, under the CaffeineMark benchmark, D2Taint's space overhead is 4.0%, which is slightly less than TaintDroid's. D2Taint's runtime overhead is 25.9% while TaintDroid's is 14%. This is expected, as D2Taint needs more operations for the tag system and location information table operations.

The rest of the chapter is organized as follows. Section 4.2 provides background information on the Android system and IFT. Section 4.3 presents our differentiated and dynamic tagging strategy. Section 4.4 presents the design and implementation of D2Taint, respectively, followed by the evaluation methodology in Section 4.5. Section 4.6 discusses our experimental results. Section 4.7 provides related work. Section 4.8 concludes.

4.2 Mobile Operating Systems And Information Flow Tracking

4.2.1 Android System

Android [4] is an open, Linux-based mobile operating system for which applications (apps) are written in Java. The Android system has several "layers." The bottom layer is the Linux kernel. Above that are middleware libraries written in C/C++ such as WebKit [115] and OpenGL. The next layer is the Dalvik virtual machine, which lies above these libraries and interprets applications (apps) written in Java. (Android's Java is based on Apache Harmony [5].) Above that, Android provides system components written in Java that are interpreted by the Dalvik VM. The top layer consists of (third-party) apps written in Java using these components. These apps are compiled to Java bytecode, which is then translated to custom Dalvik EXecutable (DEX) bytecode for the Dalvik VM. Each app is isolated from other apps via a sandbox mechanism that assigns it a unique `uid` and `gid`. Apps can communicate with each other using the Binder IPC mechanism, which enables message passing via parcels. Android exposes mobile phone functionalities such as the camera, Internet access, etc. to apps via a capability mechanism ("permissions"). Each app requests a set of permissions that are displayed to users at install time. When Android runs

the app, it verifies that the app has the necessary permissions to perform its tasks. If not, the system throws an exception and halts the app.

Dalvik VM: Dalvik is a register-based virtual machine (VM) that interprets Android applications' DEX bytecode. (By contrast, the Java VM is stack-based.) DEX bytecode is designed for greater compactness than Java bytecode to help save space on resource constrained mobile devices. The Dalvik VM maintains so-called "virtual" registers (*registers* for short) on which all operations are based. As [41] notes, Dalvik also maintains an internal execution stack and the current method's registers are on the top stack frame. Register contents roughly correspond to local variables; registers store primitive types and object references. DEX uses class fields for long-term storage. DEX instructions only work with registers, necessitating load and store operations on fields before and after data are used [41].

4.2.2 Information Flow Tracking Basics

Information flow tracking (IFT) is a promising and effective technique for detecting leakage of sensitive data [74, 75, 110] and system-compromising security attacks [25, 77, 103]. It can be implemented in three different ways, including compiler analysis on programs written in special type-safe programming languages [33, 34, 56, 74, 75], software instrumentation at the source code, bytecode, or binary level [41, 77, 117], and architectural support for IFT [29, 103, 110].

To detect leakage of sensitive data, IFT techniques generally tag (label) the source data using a pre-defined structure, e.g., sensitivity level or source IDs. The source data can come from I/O devices such as disks, keyboards, and cameras. For example, data in the password file can be tagged with the highest sensitivity or the owner id

(`root`) of the file. During program execution, data tags are propagated based on certain policies. For example, $a = b + c$ means a's tag derives b and c's tags. One propagation policy could be $a.tag = max(b.tag, c.tag)$ if we use the sensitivity level as the tag. Finally, the technique checks the data tag against security rules whenever certain data are sent over channels such as networks. If a security rule is violated, an information leakage alarm is raised. An example of such rules could be "The most sensitive data (e.g., passwords) may not be sent over the network."

4.3 Differentiated and Dynamic Tagging

4.3.1 Design Rationale

Recall from Section 4.1 that our key challenge is tracking a vast number of information sources. Smartphone data can originate from many sources such as online banks, social networking websites, etc. Any smartphone based IFT system needs to track all these sources in each data tag. Yet tag capacity is limited, e.g., 32 bits. A naïve approach to solve this challenge is using 1 bit to track each source. However, this requires many bits to track all sources, far more bits than the tag length. This approach leads to enormous tag overhead. On the other hand, we aim to achieve two contradictory goals: *source completeness* and *accuracy of source information*. Source completeness refers to how many information sources we capture. Clearly, we want to capture as many sources as possible. Accuracy of source information refers to the accuracy with which we map information recorded in the tag to the specific information source that was recorded. This information recorded in the tag should *uniquely* identify this information source. However, due to the limited tag length, this may be infeasible. In summary, a better solution must be sought.

In this chapter, we propose a so-called differentiated and dynamic tagging strategy to overcome the above challenge while balancing these two contradictory goals. Our strategy is based on the following observations on at three different levels: information sources, applications, and user behaviors.

Source Level: Different information sources may have different sensitivities in terms of security. For example, information from online bank websites is far more sensitive than information from news websites. As such, bank information merits a higher level of security than news information.

Application Level: The patterns by which applications access information sources differ for different applications. That is, these patterns follow a *non-uniform* distribution over the set of all applications. Some applications such as online banking applications access only a few sources whereas others such as news aggregators access many sources. Additionally, applications have variables that have different correlations with each other. For instance, if we execute the statements $a := c+d+\cdots+z$ and $b := 1$, a is correlated with variables $c, \cdots, z$ whereas b is correlated with no other variables. Clearly, we need to capture these heterogeneous variable correlations. Thus, we need different amounts of storage space (bits) to capture heterogeneous sources and correlations.

User Level: We observe that smartphone users' behavior patterns vary over time. Consider the following real-world usage scenario. Suppose a user often reads many news websites like `cnn.com`. Once or twice, the user logs in a social networking website like Facebook to check for messages from friends and checks a bank account. From this scenario, we can see that users access different information sources over time. Thus, IFT needs to adapt to changing information source access patterns.

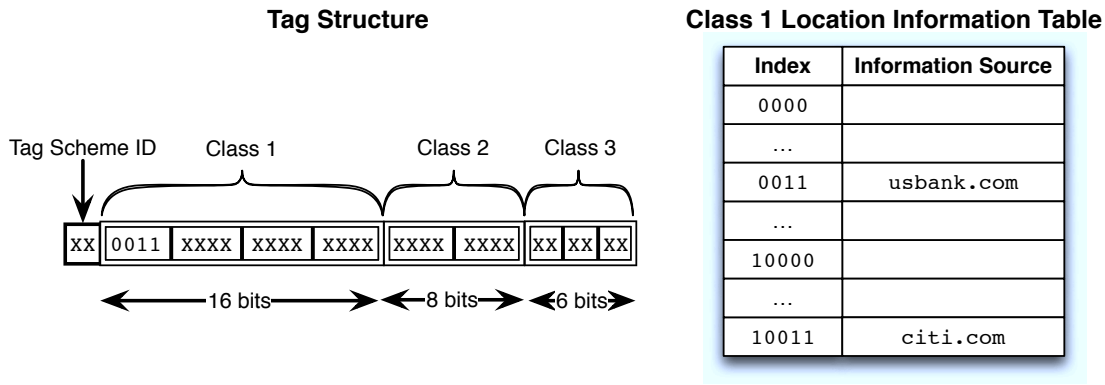


Figure 4.2: Tag structure

We develop the differentiated and dynamic strategy based on these levels. By examining the source level and application level, we develop *differentiated classes*, which classify information sources based on considerations such as their information sensitivities. Differentiated classes inform us about information sources' sensitivity. This information provides guidance for designing the tag scheme given limited tag length. By examining the user level, we develop tag *dynamics*, in which the tag scheme is updated on-the-fly based on properties of received information sources and users' access thereof. Differentiated classes and tag dynamics enable us to track many information sources on smartphones in real-time. We discuss differentiated classes in Section 4.3.2 and tag dynamics in Section 4.3.3.

4.3.2 Differentiated Tag Structure

Our tag structure is illustrated in Fig. 4.2. Each tag has the same fixed length. There may be multiple tag schemes in our tag system when dynamics are considered (we discuss dynamics shortly). We assign each tag a tag scheme ID, which is a fixed length bit string at the beginning of the tag. We partition the remainder of the tag into segments. Each segment corresponds to a distinct class. Each segment contains

a certain number of information sources. Intuitively, there are fundamental tradeoffs among the number of classes in a tag, the number of information sources that can be stored in a class, and the number of bits used to represent an information source in that class. In fact, the bits representing a source map to the indices of a location information table. Each class has its own table. For a particular class, each entry in that class's table "describes" an information source in that class. Specifically, each table entry maps the bits representing an information source to a text string describing that source. As a concrete example, consider the tag in Fig. 4.2. We see the tag is 32 bits long with a 2-bit tag scheme ID. We notice the remainder of the tag is partitioned into three segments of lengths 16 bits, 8 bits, and 6 bits. Each segment corresponds to a distinct class. In the first class, each information source is represented as a 4-bit string. In particular, the bit string `0011` maps to `usbank.com` in the location information table. Yet as more information sources arrive, the table grows and this string can map to either usbank.com or `citi.com`. In the second class, each information source is represented as a 4-bit string, and in the third class, each information source is represented as a 2-bit string. (These classes' tables are omitted for brevity.) Notice that the source representations can have various numbers of bits in different classes.

Examples: The following examples illustrate our differentiated tag structure.

- *32 bits, 1 class for each bit:* This example assumes there is only a single tag, so there is no tag scheme ID field. In this example, each bit in a 32-bit tag can represent one information source. This is exactly the approach TaintDroid [41] uses. A tag system using this approach can support 32 distinct sources and keep 32 live records in the tag.

- *32 bits, 2-bit tag scheme ID, 3 classes, 16/8/6 bits per class, 4/4/2 bits per source:* This example shows the tag structure in Fig. 4.2. We have three classes: "highly sensitive", "moderately sensitive", and "insensitive". We allocate 16 bits for "highly sensitive", 8 bits for "moderately sensitive", and 6 bits for "insensitive".

- *32 bits, 2-bit tag scheme ID, 2 classes, 24/6 bits per class, 3/2 bits per source:* In this example, we allocate 24 bits for the "highly sensitive" class and 6 bits for the "insensitive" class. We can store 8 sources in the "highly sensitive" class and three sources in the "insensitive" class. This example works for the case where there are more sensitive sources.

Tag Parameter Settings: We can realize differentiated classes based on information sensitivities. We can change the number of classes so long as each class stores at least one information source representation and the tag length is fixed. Clearly, we face tradeoffs among the class length, number of information sources represented in each class, and the number of bits for each source representation in the tag. Our proposed design incurs the following tradeoffs among the number of classes, number of information sources, and bits per source.

- If we record too few sources in the tag, some sources will be absent from the tag. If we use too few bits to represent a source, collisions can occur among sources. Then there will be less accurate source information within a class.

- There are two special cases within a class: (1) Many sources with very few bits per source. This works well when an application accesses few information sources but its individual variables are correlated with many other variables; (2) Very few sources with many bits per source. This works well when an application accesses

many different information sources but its individual variables are only correlated with a few variables.

- We cannot arbitrarily set the number of classes, number of sources, and bits per source. The total number of bits is bounded by the tag space. Also, the number of sources in a class is at most 2^n, where n is the number of bits per source.

4.3.3 Tag Dynamics

We realize tag dynamics as follows. Each class can have a different length at different times. As new information sources arrive, we classify them based on sensitivity, add them to the respective location information table, and place their (truncated) indices in the tag. Based on information source knowledge, we can adjust the class size for each class. An intuitive way is to "pre-specify" some classes and change the tag structure once certain conditions are met, e.g., most tags have less than 50% space usage. Another way is to perform "on-demand" machine learning (ML) based on statistical properties of tag space usage and location information tables' recent hash values. With this way, we create an ML process, which collects tag information from a normal process, and calculates a new tag structure for the normal process. After a new tag structure is specified, the ML process sends the new tag structure to the normal process, which can adjust its tag structure accordingly.

In designing tag dynamics, we need to consider how the tag structures should be changed. Initially, the tag should record as many sources per class as possible subject to the constraints described above. If most variables do not use their entire class space during tag scheme system execution, we can assign more bits per source

and adjust class length accordingly. For such an adjustment, we need to consider the following two issues:

- *Tag Scheme Switching:* Switching among different tag schemes is crucial for our dynamic tag system design. There are two problems: (1) determining tag scheme configurations; and (2) determining when to switch tag schemes. We have two main approaches: (1) *pre-configured*; and (2) *on-the-fly*. The pre-configured approach lets users configure their own tag schemes based on their Internet usage behaviors. Recall the last two examples in Section 4.3.2. The tag structure can switch between these when users access many highly sensitive information sources. For example, they can list important websites they often visit and classify them into different classes based on information sensitivities. All other websites and external sources are classified as "unknown." When users access more sensitive websites than unknown sites, the source lengths in the sensitive class can be increased and the source lengths in the unknown class can be shortened. Criteria for switching and the switching process can be configured in advance. The on-the-fly approach can adjust the tag scheme based on tag space utilization and newly arriving information sources. It is suitable for determining when to switch tag schemes as user and application behaviors are unpredictable. For example, if class 2 sources arrive while class 1's space is not full, the system allocates more space for class 2 dynamically.

- *Tag merging:* Tag merging is necessary for tag propagation. When multiple tags "meet" based on variable operations, we need to generate a new tag. If both merging tags have different tag schemes due to tag switching, we first need to convert old tags to new tags. In a simple case, both tag schemes' representations have the same length and we can simply merge them in the new tag scheme. If we cannot

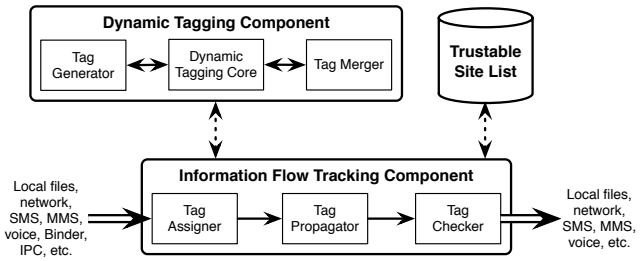


Figure 4.3: D2Taint system architecture

fit all information sources in the new class, we can select source representations for replacement. But their lengths may differ. If the length decreases in the new tag system, some most significant bits need to be removed from the old representations. If the length increases, we retrieve "old" indices from the table and place them into the new class segments.

4.4 IFT with Dynamic and Differentiated Tagging

4.4.1 System Overview

Fig. 4.3 shows our D2Taint system architecture. The system has two main components: (1) the *dynamic tagging* component; and (2) the *information flow tracking* component.

The dynamic tagging component handles tag management and determines when to switch tag schemes. It has three parts: (1) the *dynamic tagging core*; (2) the *tag generator*; and (3) the *tag merger*. The dynamic tagging core handles configuration of the dynamic tag system and decides which tag scheme can be used. The tag generator fetches or generates a tag for incoming data. The tag merger merges two tags and generates a new tag.

The information flow tracking component tracks a data flow from its sources until the related data are sent out or written to files. It has three parts: (1) the *tag assigner*; (2) the *tag propagator*; and (3) the *tag checker*. The tag assigner intercepts incoming data through I/O channels such as networks and assigns the initial tag to the data based on the results from the dynamic tagging component. The tag propagator propagates data tags for each operation during program execution. The tag checker checks the data tags for compliance with security policies when data are going to be sent out over I/O channels. Our system also maintains a trustable site list containing a list of websites to which data can be sent without raising any alerts. Users can modify the list based on the IFT results.

A typical D2Taint workflow is as follows. When an application starts, the dynamic tagging component first loads two configuration files: one stores tag structure definitions and the other stores user-defined classes and known data sources in each class. Then the dynamic tagging component performs two tasks. First, this component checks the data source list for each incoming data source passed by the tag assigner. If found, the tag is retrieved and returned to the tag assigner. Otherwise, a new entry for the data source is created and the new tag is returned. Second, the dynamic tagging component tracks incoming sources' statistics and determines whether it should switch D2Taint to a different tag scheme.

The tag assigner intercepts I/O channels to capture incoming data. For each incoming data source, the tag assigner consults the dynamic tagging component to get the tag and then assigns the tag to the new data.

For each instruction being executed, the tag propagator is in charge of propagating tags for a data flow. If it needs to merge multiple tags to generate a new

tag, it gets a new tag from the dynamic tagging component. The dynamic tagging component merges the tags from different source data and assigns the merged tag to the destination data. For example, a binary operation $a = b + c$ triggers tag propagation $a.tag = b.tag \oplus c.tag$, where $\oplus$ is the merge operation. The dynamic tagging component handles tag merging differently for two cases: (1) where the tag scheme is not switched; and (2) where $b.tag$ and $c.tag$ use different tag schemes.

When data are going to be sent out over I/O channels, the tag checker is activated. It uses the list of trustable sites to check if the data are allowed to be sent to their destinations. If this data sending event is not allowed, the tag checker raises an exception to the user, who is asked if the data can be sent.

We implement a prototype of D2Taint on Nexus One smartphones running Android 2.2. However, we see no particular difficulty applying our ideas to other smartphone architectures. To reduce overhead incurred by tag storage and propagation, our D2Taint system tracks data flows for Java code at the variable level, as TaintDroid [41] does. By instrumenting the Dalvik VM during interpretation of Java bytecode, we can fully utilize Java objects' semantics to store taint tags and propagate them among objects. This helps reduce IFT's overhead.

In the following, we present the two components in detail. First, we introduce the dynamic tagging component, then we illustrate the information flow tracking component.

4.4.2 Dynamic Tagging Component

The dynamic tagging component realizes management of tag schemes, tag assignment, and tag merging. These functions are performed by three sub-components: the

dynamic tagging core, the tag assigner, and the tag merger, respectively. We discuss them in the following.

Dynamic Tagging Core

The dynamic tagging core maintains two configuration files. The first configuration file stores the tag system settings, which are read and parsed into memory during D2Taint's initialization phase. In memory, we maintain a global array that stores settings for each tag scheme. Each tag scheme setting includes the scheme number, the number of bits per tag, the number of classes, and a pointer to the class list. In a class structure, we record the number of classes in the tag system, the number of bits per hashcode, the number of reserved slots for the class, and a text description of the class. In our current implementation, we use 32 bits for a tag and 5 bits for a hashcode by default. The first 2 bits are used to indicate the number of the tag scheme. There are 6 available hashcode slots in a tag.

The second configuration file stores user-defined classes and each class's known data sources. After reading the data from the configuration file, we use a global location information table list to record all source information. Each information table corresponds to one class. Each source has an entry in one particular table. Note that we only store the domain name for a source, e.g., `google.com`, `nsf.gov`, etc. If an IP address has no corresponding domain name or hostname, we store the first 16 bits of the address, e.g., `192.168.0.0`. This helps decrease the total number of entries in the location information tables. A domain name or an IP address suffices for a user to determine the information source. To save space, each table is dynamically allocated as its number of entries increase.

The dynamic tagging core also collects statistics for incoming sources and determines whether the tag scheme should be switched. In particular, after a certain number of new sources (i.e., 50) are added into an location information table, D2Taint decides whether to switch the tag scheme based on these new sources. D2Taint counts the source distributions for each class, finds the best matched scheme with this distribution, and updates the current tag scheme number. This implementation incurs a low overhead as it only makes the "switch" decision periodically after enough new sources arrive.

Tag Generator

The tag generator uses the location information list to respond to the tag assigner's query when new data arrive. The tag generator checks the location information list for each new data source. If the source is in the list, the tag is retrieved and returned to the tag assigner. Otherwise, a new entry for the data source is created and a new tag is returned.

Tag Merger

The tag merger performs tag merging, which is necessary for tag propagation. When multiple tags "meet" in a corresponding bytecode instruction, a new tag has to be generated based on the source data tags. For example, $a = b + c$ triggers $a.tag = b.tag \oplus c.tag$. To merge tags $b.tag$ and $c.tag$, we need to handle two cases: (1) when these two tags use the same tag scheme; and (2) when they use different tag schemes. Merging multiple tags can be seen as multiple instances of merging two tags.

For the first case, a new tag can be quickly formed by collecting the hashcodes in the corresponding class segments when each class has enough room to host all hashcodes from *b.tag* and *c.tag*. Sometimes, the classes in *b.tag* and *c.tag* contain more sources that one class segment in *a.tag* can hold. To handle this case, we can either randomly drop sources or select source tags based on their access recency or frequency.

For the second case, we have to first convert an old tag to a new tag based on the current tag scheme. In a simple case, the old tag scheme has the same hashcode length as the current tag scheme. If so, we just put the hashcode into its class segment in the new tag. When there are more hashcodes than available slots for a class, we can either randomly select some hashcodes or keep the latest ones (i.e., those with larger values). But the hashcode lengths may differ among different tag schemes. If the length decreases in the current tag scheme, certain significant bits need to be truncated from the old hashcodes. If the length increases, we first retrieve the hashcode indices in the location information table, hash these indices into new hashcodes, and finally fit the hashcodes into the class segments.

4.4.3 Information Flow Tracking Component

The information flow tracking component tracks information flows from sources to destinations in an Android application at runtime. It includes three sub-components: (1) the *tag assigner*; (2) the *tag propagator*; and (3) the *tag checker*, which perform tag assignment, tag propagation, and tag checking, respectively. We first discuss how D2Taint stores tags for different data, then we discuss each sub-component. In the following, we call the memory block that is used to store tags a *taint map*.

Taint Map

In Dalvik VM, five types of variables need taint maps to store their tags: method local variables, method arguments, class instance fields, class static fields, and arrays. Among these data types, method local variables and method arguments are stored in methods' stack frames. We store tags of class static fields and arrays into their representative objects. TaintDroid [41] does likewise. However, for the other three variable types, our taint maps differ from TaintDroid's. We do not store variables' tags adjacent to them in memory. Instead, we use specific taint maps for these variable types, as our system's tag lengths tend to change. Further details follow:

Method local variables and method arguments. We use a stack taint map to store tags for a method's local variables and arguments. A stack taint map differs from a method's stack frame. When Dalvik VM allocates a stack frame for a method, our system allocates a stack taint map for it. The last element of the stack taint map is for the method's return value.

Class instance fields. Tags for class instance fields are stored in objects' taint maps. An object's taint map is stored in the memory area immediately after that allocated for the object.

Tag Assigner

The tag assigner labels data tags according to their origins. While the data are read, the tag assigner tries to determine the data's origin and uses such information to query the dynamic tagging component. After it receives the tag, the tag assigner labels the data with the tag. If the origin information contains multiple sources, then the tag can be used to locate multiple sources. To taint data effectively, we insert our

tag assigner logic into file I/O, network I/O, sensor, and other library functions that read private information, e.g., device identifiers, call histories, etc. TaintDroid [41] also does so.

Tag Propagator

As an IFT system, D2Taint needs to instrument program execution to track data flows. Based on this, we use the same propagation logic as TaintDroid to propagate tags in interpreted code and native code [41]. Our system also propagates tags from one process to another via Binder IPC, and writes the data's source information into the file system if the data are written to local files. The biggest difference is TaintDroid's use of bitwise OR to merge two or more tags, whereas we use the method in Section 4.4.2 to merge two tags. Also, when a message is sent via Binder IPC, our system extracts source information from the related tags and sends it with the message via IPC. After the receiver gets the message from IPC, it extracts the message's source information and uses it to get a tag for tainting the received data.

Tag Checker

The tag checker is activated when data are going to be sent out via networks. First, the tag checker leverages the list of trustable sites to determine the destination's trustworthiness. If the destination is trustable, the data can be sent without raising any alerts. If the destination is not in the trustable list, our system extracts source information from the data's tag and then delivers it to the user, who decides if the data can be sent to the destination. If the user does not want the data to be sent to the destination, the tag checker blocks data sending and stops program execution; otherwise, the data is sent out and the execution continues.

4.5 Evaluation Methodology

We evaluate our D2Taint system in three ways: (1) real-world application study; (2) system performance evaluation; and (3) dynamic tagging system evaluation.

In the real-world application study, we select 84 "top free" apps from Google Play [54] in July 2012. We believe these apps represent a cross-section of those in widespread use on Android smartphones. Since many apps are ad-supported, we believe they may potentially leak sensitive information to third parties, as prior work suggests [41, 42]. We download these apps, install them on Nexus One smartphones running our D2Taint system, and exercise app functionalities. We monitor app installation and execution to check if these apps leak information. We collect system logs, IPC messages, and network traces from the phones using `adb logcat`. We verify the results using `tcpdump` on the Nexus One's WiFi interface. No Nexus One had a SIM card and Bluetooth was disabled; all network traffic went through WiFi. When apps read data from the Internet, we record the data's sources via tags as well as the destinations to which the data are sent. We inspect the relevant hostnames and corresponding IP addresses to remove false positives, e.g., two different IP addresses belonging to the same organization. For comparison purposes, we perform the same experiments on smartphones running TaintDroid 2.3.

In the system performance evaluation, we demonstrate that D2Taint's overhead is reasonable. We use an unmodified Android ROM as the base of performance comparison. First, we test the impact of D2Taint on the user experience, especially the execution time. We also test other common smartphone operations, including system and networking operations. Second, we use a standard benchmark tool, CaffeineMark [81], to measure D2Taint's overhead. CaffeineMark reports scores of various

features based on the Java execution time. Memory overhead is measured based on CaffeineMark's increased memory footprint on the D2Taint system.

Lastly, we evaluate the performance of dynamic tag system design and show the benefit of such a design. We develop a test app to emulate two different usage patterns: sequential and random. We measure the number of sources recorded in the tags when the data are sent out via network sockets, which shows a tag's effective space utilization. We demonstrate the performance improvement of our dynamic tag design in comparison to a static tag system.

4.6 Experimental Results

In this section, we present the experimental results following the above evaluation methodology.

4.6.1 Real-world Application Study

D2Taint finds that 71 out of the 84 apps leak information to third-party destinations. D2Taint reveals the paths by which the information is leaked, whereas TaintDroid only reveals the final leakage destinations. In our experiments, we found 33 apps that transmit data among myriad various external sources, especially cloud computing services (e.g., Amazon Web Services). To reduce false positives, D2Taint uses the following rule: information flows whose sources and destinations are the same are treated as legal. In addition, D2Taint provides detailed information about multiple sources when reporting to the user. By contrast, TaintDroid cannot record any source information for external data since it only uses 1 bit to tag the data. There are two consequences: (1) it triggers false positives whenever data flows from an external source to that same source, which we observed frequently during experiments; and (2)

	Stock Android	D2Taint
App Load	53.19 ms	58.12 ms
Download (32.3 MB)	35.73 s	38.34 s
Web Load (`google.com`)	735 ms	856 ms
Web Load (`nytimes.com`)	1081 ms	1116 ms
HttpGet	658 ms	746 ms
Write File	7.96 ms	8.02 ms
Read File	1.21 ms	1.51 ms
Socket Send	5.24 ms	6.37 ms

Table 4.1: Macrobenchmarks

it cannot keep track of data from multiple sources at once. Our experiments validate the real-world problem of external data leakage and show that D2Taint can be used to detect information leakage related to many external sources.

In addition, D2Taint detects that some apps send highly sensitive internal data such as IMEIs/EIDs to third parties, particularly ad and market research companies (e.g., `admob.com` and `flurry.com`). More specifically, D2Taint finds 12 apps leaking devices' IMEIs/EIDs: The Weather Channel, ESPN ScoreCenter, NavFree GPS, SWAT Army, Bible, Fruit Ninja Free, Coin Dozer, Yellow Pages, Scramble with Friends, Words with Friends, Funny Facts Free, and IQ Test. From this aspect, TaintDroid also reports these apps leak sensitive information.

4.6.2 System Performance Evaluation

Macrobenchmarks

Macrobenchmark results are shown in Table 4.1. Each value is averaged over 30 runs.

Application load time: We measure the time needed to load a new Android app and display the UI. D2Taint's overhead with respect to stock Android is 9%.

Download time: We measure the time needed to download a 32.3 MB file from `google.com`. D2Taint's overhead is 7.3%.

Webpage load time: We measure webpage load time using a toy "Web view" app. Specifically, the time between a UI button press and the webpage completely loading is measured. Two types of webpages are tested: light text (`google.com`) and heavy text (`nytimes.com`). Table 4.1 shows D2Taint's overheads are 16% for `google.com` and 3% for `nytimes.com`. This can be explained as follows. `google.com` automatically redirects to a "mobile-friendly" webpage, leading to more webpage data caching operations, as recorded in D2Taint logs. Since D2Taint also outputs tags into the file when writing data, `google.com`'s overhead is larger than `nytimes.com`'s.

Input and output: Besides basic system and networking operations, we develop an app that reads data from the "top 100" websites hosted in the U.S. [3], writes the data to a file, reads 1,000 bytes from this file, and transmits the 1,000 bytes to a remote machine via a socket connection. Table 4.1 shows the results. Each value is averaged over the top 100 websites' data with 10 runs.

The networking input and output overheads are 13% (HttpGet) and 21% (socket transmission), respectively. The input overhead stems from the location information table query to assign a new tag to the input data. For the output overhead, D2Taint needs to access the location information tables to lookup source information as well as the trustable list to determine if data transmission is allowed. The filesystem I/O overhead is negligible: 0.5 ms for reads and writes.

Java Microbenchmark

The CaffeineMark scores are shown in Fig. 4.4. The scores roughly correspond to the number of Java instructions executed per second. D2Taint and unmodified

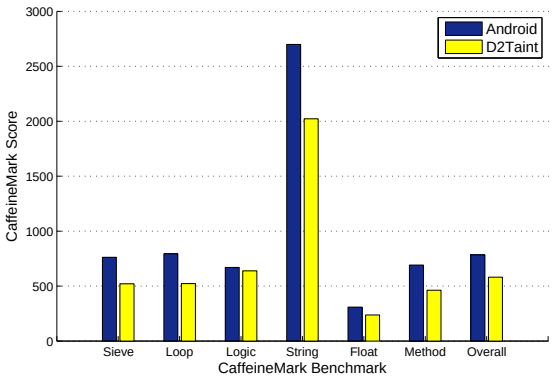
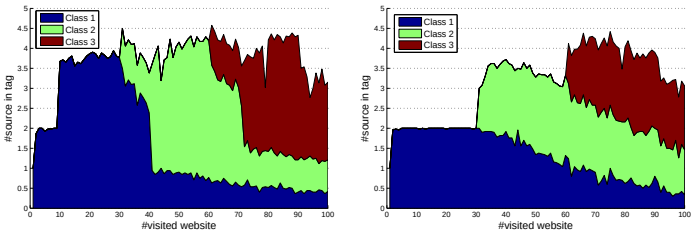


Figure 4.4: Microbenchmark: Java overhead

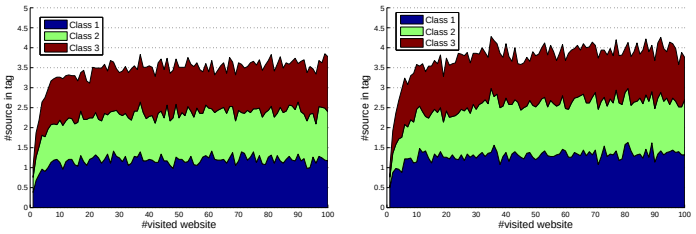
Android have scores 581 and 784, respectively, so D2Taint's overhead is 25.9%. In contrast, TaintDroid and unmodified Android have scores 967 and 1121, respectively, and TaintDroid's overhead is 14%. Though D2Taint's overhead is higher than TaintDroid's, D2Taint's absolute impaired score (203) does not significantly differ from that of TaintDroid (154). D2Taint's extra overhead is expected since D2Taint needs more operations for the tag system and location information table operations. In contrast, TaintDroid uses only the "`OR`" operation to merge tags since its sources are fixed and hard-coded.

We also measured CaffeineMark's memory footprint to determine D2Taint's space overhead. Since the memory footprint value varies with the time after CaffeineMark is launched, we obtained the value immediately after rebooting a system. CaffeineMark consumes 21664 KB and 22528 KB in a unmodified Android system and our D2Taint system, respectively: a 4.0% overhead. This overhead is slightly lower than TaintDroid's, which is 4.4% [41]. Both D2Taint and TaintDroid use the same tag length, 32 bits. The other primary memory used by D2Taint is for the location information table. Note the location information table dynamically increases as more



(a) Dynamic (b) Static

Figure 4.5: Sequential websites



(a) Dynamic (b) Static

Figure 4.6: Random websites

information sources arrive. This overhead is ignored here as CaffeineMark does not access the Internet; hence no entries would be added into the tables.

4.6.3 Dynamic Tag System Performance

We evaluate the performance of a dynamic tag system under different situations. To do so, we measure the number of sources recorded in a tag.

In the experiments, we pre-configure four tag schemes for three classes: (1) 2/2/2 hashcode slots for classes 1/2/3 (default); (2) 4/1/1 hashcode slots for classes 1/2/3; (3) 1/4/1 hashcode slots for classes 1/2/3; (4) 1/1/4 hashcode slots for classes 1/2/3. Each hashcode's length is fixed at 5 bits. The tag scheme switching is triggered after 10 new entries are added into the location information tables. We select the "best matched" tag scheme based on the class distributions among these entries.

We write an app to visit the top 100 websites hosted in the U.S. Websites 1–30 are classified into class 1, 31–60 are classified into class 2, 61–90 are classified into class 3, and the remaining 10 are unclassified. The app visits websites 1–100 sequentially or randomly. To emulate tag propagation, we combine several previously downloaded webpages into the final "stolen" data for socket transmission. Thus, the most recent webpage has a higher probability to be selected for combination. We run the experiments 50 times.

The results are shown in Figs. 4.5 and 4.6. For sequential websites, a static tag system can record at most two sources before the class 2 websites are visited. The peak appears after class 3 are visited as all six available hashcode slots can be used. In our D2Taint, we found tag scheme switching happened at websites 11, 41, and 71. There are some troughs among the class transition period as the tag system has not adjusted yet. But shortly thereafter, the average source number increases to about four sources per tag. The final trough is caused by the unclassified websites as they do not appear in the tag. In general, a dynamic tag system's performance is much better than a static tag system's performance as the former fits the currently visited website classes well.

For random websites, tag scheme switching occurs about five times (on average) with D2Taint. The number of sources remains stable with the number for the static tag system. The dynamic tag system's performance is a little worse than the static tag system's, as the default tag system is "best" for random websites. Thus, the dynamic tag system is not a good candidate for a totally random situation. But we argue that patterns tend to arise as users visit websites and run apps.

4.7 Related Work

Information leakage in smartphone systems has attracted considerable attention. In numerous instances, third-party applications have leaked personal information to remote servers [32,72,92]. Most recently, Carrier IQ [13] has gathered voluminous data from smartphones, apparently without their owners' knowledge. Smartphones' sensors can also leak information such as workers' activity [47] to remote servers. Many systems have been proposed to combat information leakage. TaintDroid [41] tracks information flow from *single* third-party applications. Vision [53] extends TaintDroid to detect implicit information flows. AppFence [57] augments TaintDroid with privacy enforcement mechanisms. Kirin [40] and Saint [79] provide rule-based security mechanisms for Android that restrict application access to sensitive information. SxC [35] adds provable security contracts to Windows Mobile for the same purpose. Other systems [34, 38, 42, 74] leverage static analysis to discern information leaking in Android and iOS applications. D2Taint has two key differences from existing IFT systems for smartphones. First, D2Taint accommodates a large number of sources, including multiple applications, and classifies them into multiple groups. Second, D2Taint's tag structure accommodates varying *types* of sources with dynamic granularity. If many untrusted sources enter the system, each source's hashcode occupies less space, and vice versa.

Most IFT systems store *static* taint tags using shadow memory [41, 89, 118] or tag maps [119]. Usually, each data byte or word corresponds to a byte or word in shadow memory. TaintDroid [41] propagates a static taint tag through the entire Android system; this tag's value does not change. By contrast, D2Taint's tag structure is partitioned into classes, providing finer granularity than static tags.

More generally, dynamic taint analysis [24, 77, 89, 118, 119] (also known as "taint tracking") is an approach for information leakage detection. Some dynamic taint analysis approaches are based on whole-system analysis using emulation environments, [20, 118], hardware extensions [29, 103, 110], and per-process tracking with dynamic binary translation [17, 24, 89, 119]. However, these kinds of whole-system analysis are far too heavyweight for resource-constrained smartphones.

4.8 Summary

We proposed a novel IFT tagging strategy using differentiated and dynamic tagging. Our strategy partitioned information sources into differentiated classes and stored class and source information in IFT tags. Our strategy enabled dynamic tag structure adaptation in real-time based on received information sources. We designed and implemented D2Taint, an IFT system using our strategy, on real-world smartphones. Our experimental evaluation illustrated D2Taint's potential to detect information leakage with moderate time and space overhead.

Chapter 5: Conclusion And Future Work

In this dissertation, we propose a methodology leveraging context information from target processes and data for malicious code detection. Based on this methodology, we propose and implement three detection systems that can be used to detect malicious code in client, server, and smartphone applications. We also conduct comprehensive experiments to test our detection systems in server computers, client computers, and smartphones. Our experimental results illustrate that the context information can be used to greatly improve detection effectiveness. In addition, the results show that our detection systems do not incur much extra overhead when they are activated.

Although our detection systems have very low false negative rates, some attacks could still evade our detection. First, our shellcode detection system can detect malicious shellcode starting from certain positions in a message or buffer. If the attack only uses addresses of `glibc` functions to overwrite the current stack frame's return address [95], then the hijacked control flow will be directed to the function, not to a position in the message or buffer. In this case, we cannot correctly decode an instruction sequence for checking because the bytes in the message or in the buffer are *addresses*, not instruction codes. This issue could be solved by utilizing the address information of `glibc` in a target process. Given a message, we can check if there is

a byte sequence that looks like an address of an `glibc` function. If it is the case, we could decode an intruction sequence from the address and emulate its execution. If it exhibits malicious behavior, we consider the message malicious.

Secondly, we will further study IFT mechanisms that can be leveraged to track information flows for native binary codes in smartphones. Currently, we only track information flows for Java codes running in the Dalvik VM. When a Java application calls a native function written in C/C++, we only use some heuristics to propagate tag information from its parameters to its return value. This could impact false positive rates and some benign programs might be considered malicious. We should track information flows in native binary codes in order to further reduce false positive rates. However, tracking information flows in native binary codes is not trivial and incurs much extra overhead. Since smartphones with multi-core CPUs are becoming increasingly common, we could leverage multiple cores to design a new IFT system for smartphones for reducing the extra overhead incurred by IFT.

Bibliography

[1] http://www.angelfire.com/sk/stackshield.

[2] http://bastard.sourceforge.net.

[3] Alexa Top Sites. http://www.alexa.com/topsites.

[4] Android. `http://android.com`.

[5] Apache Harmony. `http://harmony.apache.org`.

[6] Piotr Bania. TAPiON, 2005. http://pb.specialised.info/all/tapion/.

[7] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent Run-Time Defense Against Stack Smashing Attacks. In *Proc. USENIX Annual Technical Conf.*, 2000.

[8] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, Trek S. Palmer, Darko Stefanović, and Dino Dai Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In *Proc. ACM CCS*, 2003.

[9] Ulrich Bayer, Andreas Moser, Christopher Kruegel, and Engin Kirda. Dynamic Analysis of Malicious Code. *Journal of Computer Virology*, 2006.

[10] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proc. USENIX Security*, 2003.

[11] Sandeep Bhatkar and R. Sekar. Data Space Randomization. In *Proc. DIMVA*, 2008.

[12] Anna Bouch, Allan Kuchinsky, and Nina Bhatti. Quality is in the Eye of the Beholder: Meeting Users' Requirements for Internet Quality of Service. In *Proc. SIGCHI Conf. on Human Factors in Computing Systems (CHI)*, 2000.

[13] Jon Brodkin. Carrier IQ hit with privacy lawsuits as more security researchers weigh in, 2 December 2011. http://arstechnica.com/tech-policy/news/2011/12/carrier-iq-hit-with-privacy-lawsuits-as-more-security-researchers-weigh-in.ars.

[14] David Brumley, James Newsome, and Dawn Song. Vulnerability-Specific Execution Filtering for Exploit Prevention on Commodity Software. In *Proc. NDSS*, 2006.

[15] Davide Canali, Marco Cova, Christopher Kruegel, and Giovanni Vigna. Prophiler: A Fast Filter for the Large-Scale Detection of Malicious Web Pages. In *Proc. Int'l. World Wide Web Conf. (WWW)*, March 2011.

[16] Stephan Chenette. Toorconx the ultimate deobfuscator(2008). http://www.toorcon.org/tcx/26_Chenette.pdf.

[17] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. TaintTrace: Efficient Flow Tracing with Dynamic Binary Rewriting. In *ISCC*, 2006.

[18] Ramkumar Chinchani and Eric van den Berg. A Fast Static Analysis Approach to Detect Exploit Code Inside Network Flows. In *Proc. RAID*, 2005.

[19] Tzicker Chiueh and Fu-Hau Hsu. RAD: A Compile-Time Solution to Buffer Overflow Attacks. In *Proc. IEEE ICDCS*, 2001.

[20] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *USENIX Security*, 2004.

[21] Mihai Christodorescu and Somesh Jha. Static Analysis of Executables to Detect Malicious Patterns. In *Proc. USENIX Security*, 2003.

[22] Mihai Christodorescu, Somesh Jha, Sanjit Seshia, Dawn Song, and Randal E. Bryant. Semantics-Aware Malware Detection. In *Proc. IEEE S&P*, 2005.

[23] Simon P. Chung and Aloysius K. Mok. Swarm Attacks against Network-Level Emulation/Analysis. In *Proc. RAID*, 2008.

[24] J. Clause, W. Li, and A. Orso. Dytan: A Generic Dynamic Taint Analysis Framework. In *Int'l. Symp. on Softw. Test. and Anal.*, 2007.

[25] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-End Containment of Internet Worms. In *SOSP*, 2005.

[26] Marco Cova, Christopher Kruegel, and Giovanni Vigna. Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code. In *Proc. 19th Int'l. Conf. on World Wide Web (WWW)*, 2010.

[27] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proc. USENIX Security*, 1998.

[28] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuard™: Protecting Pointers from Buffer Overflow Vulnerabilities. In *Proc. USENIX Security*, 2003.

[29] J. R. Crandall and F. T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *MICRO*, 2004.

[30] Charlie Curtsinger, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. Zozzle: Fast and Precise In-Browser JavaScript Malware Detection. In *Proc. 20th USENIX Security Symp.*, 2011.

[31] David Dagon, Guofei Gu, Chris Lee, and Wenke Lee. A taxonomy of botnet structures. In *Proceedings of the 23 Annual Computer Security Applications Conference (ACSAC'07)*, December 2007.

[32] Chris Davies. iPhone spyware debated as app library "phones home", 17 August 2009. `http://www.slashgear.com/iphone-spyware-debated-as-app-library-phones-home-1752491/`.

[33] D. E. Denning. A Lattice Model of Secure Information Flow. *Commun. ACM*, 19(5):236–243, 1976.

[34] D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *Commun. ACM*, 20(7), 1977.

[35] Lieven Desmet, Wouter Joosen, Fabio Massacci, Pieter Philippaerts, Frank Piessens, Ida Siahaan, and Dries Vanoverberghe. Security-by-contract on the .NET platform. *Inf. Security Tech. Rep.*, 13(1):25–32, 2008.

[36] Theo Detristan, Tyll Ulenspiegel, Yann Malcom, and Mynheer Superbus van Underduk. Polymorphic Shellcode Engine Using Spectrum Analysis. *Phrack*, 2003. http://www.phrack.org.

[37] Yu Ding, Tao Wei, Tielei Wang, Zhenkai Liang, and Wei Zou. Heap Taichi: Exploiting Memory Allocation Granularity in Heap-Spraying Attacks. In *Proc. 26th Annual Computer Security Applications Conf. (ACSAC '10)*, 2010.

[38] M. Egele, C. Kruegel, E. Kirda, , and G. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *NDSS*, 2011.

[39] Manuel Egele, Peter Wurzinger, Christopher Kruegel, and Engin Kirda. Defending Browsers against Drive-by Downloads: Mitigating Heap-spraying Code Injection Attacks. In *Proc. DIMVA*, June 2009.

[40] W. Enck, M. Ongtang, and P. McDaniel. On Lightweight Mobile Phone Application Certification. In *CCS*, 2009.

[41] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: an information-flow tracking system for realtime privacymonitoring on smartphones. In *OSDI*, October 2010.

[42] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A Study of Android Application Security. In *USENIX Security*, 2011.

[43] Hiroaki Etoh and Kunikazu Yoda. Protecting from Stack-Smashing Attacks. http://www.trl.ibm.com/projects/security/ssp/main.html.

[44] Ben Feinstein and Daniel Peck. Caffeine Monkey. http://www.secureworks.com/research/blog/wp-content/uploads/CaffeineMonkey_DEFCON15.pdf.

[45] Johannes Hoffmann Thorsten Holz Sebastian Uellenbeck Christopher Wolf Felix Freiling, Michael Becher. Mobile security catching up? revealing the nuts and bolts of the security of mobile devices. In *IEEE Symposium on Security and Privacy*, May 2011.

[46] Fiddler. http://fiddler2.com/fiddler2.

[47] Michael Fitzpatrick. Mobile that allows bosses to snoop on staff developed. BBC News, 10 March 2010. http://news.bbc.co.uk/2/hi/technology/8559683.stm.

[48] Fnord. http://www.cansecwest.com/spp_fnord.c.

[49] Seth Fogie, Jeremiah Grossman, Robert Hansen, and Anton Rager. *XSS Attacks: Cross Site Scripting Exploits and Defense*. Syngress, May 2007.

[50] Y. Fratantonio, C. Kruegel, and G. Vigna. Shellzer: a tool for the dynamic analysis of malicious shellcode. In *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)*, S. Francisco, CA, 2011.

[51] Stefan Frei, Thomas Duebendorfer, Gunter Ollmann, and Martin May. Understanding the web browser threat, 2008. In *DefCon 16 2008*, August 2008.

[52] Francesco Gadaleta, Yves Younan, and Wouter Joosen. BuBBLe: A JavaScript Engine Level Countermeasure Against Heap-Spraying Attacks. In *ESSoS*, February 2010.

[53] P. Gilbert, B. G. Chun, L. P. Cox, and J. Jung. Vision: Automated Security Validation of Mobile Apps at App Markets. In *MCS*, 2011.

[54] Google Play. `http://play.google.com/apps`.

[55] Boxuan Gu, Xiaole Bai, Zhimin Yang, Adam C. Champion, and Dong Xuan. Malicious Shellcode Detection with Virtual Memory Snapshots. In *INFOCOM*, pages 974–982, 2010.

[56] N. Heintze and J. G. Riecke. The SLam Calculus: Programming with Secrecy and Integrity. In *POPL*, pages 365–377, 1998.

[57] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. "These Aren't the Droids You're Looking For": Retrofitting Android to Protect Data from Imperious Applications. In *CCS*, 2011.

[58] httpload. http://www.acme.com/software/http_load/.

[59] Costin Ionescu. GetPC code. http://securityfocus.com/archive/82/327348/2006-01-03/1.

[60] John Markoff. `http://www.nytimes.com/2009/01/23/technology/internet/23worm.html`.

[61] Gaurav S. Kc and Angelos D. Keromytis. e-nexsh: Achieving an Effectively Non-Executable Stack and Heap via System-Call Policing. In *Proc. ACSAC*, 2005.

[62] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering Code-Injection Attacks with Instruction-Set Randomization. In *Proc. ACM CCS*, 2003.

[63] Kelly Burton. The Conficker Worm. `http://www.sans.org/security-resources/malwarefaq/conficker-worm.php`.

[64] Hyang-Ah Kim and Brad Karp. Autograph: Toward automated, distributed worm signature detection. In *the 13th conference on USENIX Security Symposium*, 2004.

[65] Arun Lakhotia and Uday Eric. Stack Shape Analysis to Detect Obfuscated Calls in Binaries. In *Proc. IEEE Int'l. Conf. on Source Code Analysis and Manipulation*, 2004.

[66] libemu. http://libemu.carnivore.it/.

[67] Long Lu, Vinod Yegneswaran, Phillip Porras, and Wenke Lee. BLADE: An Attack-Agnostic Approach for Preventing Drive-By Malware Infections. In *Proc. 17th ACM Conf. on Computer and Communications Security (CCS 2010)*, 2010.

[68] S. Macaulay. ADMMutate: Polymorphic Shellcode Engine. http://www.ktwo.ca/security.html.

[69] Joshua Mason, Sam Small, Fabian Monrose, and Greg MacManus. English Shellcode. In *Proc. 16th ACM Conf. on Computer and communications security*, 2009.

[70] John P. McGregor, David K. Karig, Zhijie Shi, , and Ruby B. Lee. A Processor Architecture Defense against Buffer Overflow Attacks. In *Proc. ITRE*, pages 243–250, 2003.

[71] Microsoft Security Bulletin MS08-067. `http://technet.microsoft.com/en-us/security/bulletin/ms08-067`.

[72] Dan Moren. Retrievable iPhone numbers mean potential privacy issues, 29 September 2009. `http://www.macworld.com/article/143047/2009/09/phone_hole.html`.

[73] Darren Mutz, William Robertson, Giovanni Vigna, and Richard Kemmerer. Exploiting Execution Contexts for the Detection of Anomalous System Calls. In *Proc. RAID*, 2007.

[74] A. C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *POPL*, 1999.

[75] A. C. Myers and B. Liskov. Protecting Privacy Using the Decentralized Label Model. *ACM Trans. on Softw. Eng. and Methodology*, 9(4), October 2000.

[76] Fiona Fui-Hoon Nah. A Study on Tolerable Waiting Time: How Long are Web Users Willing to Wait? *Behaviour & IT*, 23(3):153–163, 2004.

[77] James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proc. NDSS*, 2005.

[78] Obscou. Building IA32 'Unicode-Proof' Shellcodes. Phrack, 8 2003. http://www.phrack.org/.

[79] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically Rich Application-Centric Security in Android. In *ACSAC*, 2009.

[80] PaX. http://pax.grsecurity.net/docs/aslr.txt.

[81] Pendragon Software Corp. CaffeineMark 3.0. `http://www.benchmarkhq.ru/cm30/`.

[82] Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. Network-level Polymorphic Shellcode Detection Using Emulation. In *Proc. DIMVA*, 2006.

[83] Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. Emulation-Based Detection of Non-Self-Contained Polymorphic Shellcode. In *Proc. RAID*, 2007.

[84] Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. Comprehensive shellcode detection using runtime heuristics. In *Proc. 26th Annual Computer Security Applications Conf. (ACSAC '10)*, December 2010.

[85] Georgios Portokalidis and Herbert Bos. Eudaemon: Involuntary and on-demand emulation against zero-day exploits. In *Proc. ACM EuroSys*, 2008.

[86] Niels Provos, Panayiotis Mavrommatis, Moheeb Abu Rajab, and Fabian Monrose. All Your iFRAMEs Point to Us. In *Proc. Usenix Security Symp.*, 2008.

[87] Niels Provos, Dean McNamee, Panayiotis Mavrommatis, Ke Wang, and Nagendra Modadugu. The Ghost In the Browser: Analysis of Web-based Malware. In *Proc. of the 1st Workshop on Hot Topics in Understanding Botnets*, 2007.

[88] QEMU. http://www.nongnu.org/qemu/.

[89] Feng Qin, Cheng Wang, Zhenmin Li, Ho-Seop Kim, Yuanyuan Zhou, and Youfeng Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *Proc. IEEE/ACM Int'l. Symp. on Microarchitecture (MICRO'06)*, 2006.

[90] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin Zorn. NOZZLE: A Defense Against Heap-spraying Code Injection Attacks. In *Proc. 18th USENIX Security Symp.*, 2009.

[91] Konrad Rieck, Tammo Krueger, and Andreas Dewald. Cujo: Efficient Detection and Prevention of Drive-by-Download Attacks. In *Proc. of 26th Annual Computer Security Applications Conf. (ACSAC)*, December 2010.

[92] S. Thurm and Kane, Y. I. iPhone and Android Apps Breach Privacy, 17 December 2010. http://online.wsj.com/article/SB10001424052748704694004576020083703574602.html.

[93] SANS. 2011 CWE/SANS Top 25 Most Dangerous Software Errors. `http://cwe.mitre.org/top25/`.

[94] Secunia. Secunia PSI study: 28% of all detected applications are insecure, 2007. http://secunia.com/blog/11.

[95] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proc. ACM CCS*, 2007.

[96] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated worm fingerprinting. In *OSDI*, pages 45–60, 2004.

[97] Skape. Shellcode text encoding utility for 7bit shellcode. http://www.hick.org/code/skape/nologin/encode/encode.c.

[98] Alexey Smirnov and Tzicker Chiueh. DIRA: Automatic Detection, Identification, and Repair of Control Hijacking Attacks. In *Proc. NDSS*, 2005.

[99] Kevin Snow, Srinivas Krishnan, Fabian Monrose, and Niels Provos. Shellos: Enabling fast detection and forensic analysis of code injection attacks. In *USENIX Security Symposium*, 2011.

[100] A. Sotirov. Heap Feng Shui in JavaScript. In *BlackHat Europe*, 2007.

[101] Alexander Sotirov and Mark Dowd. Bypassing Browser Memory Protections. In *In Proceedings of BlackHat*, 2008.

[102] SpiderMonkey JavaScript engine. http://www.mozilla.org/js/spidermonkey/.

[103] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *ASPLOS*, 2004.

[104] The Metasploit Project. http://www.metasploit.com.

[105] The Nielsen Co. Two Thirds of New Mobile Buyers Now Opting For Smartphones, 12 July 2012. `http://blog.nielsen.com/nielsenwire/online_mobile/two-thirds-of-new-mobile-buyers-now-opting-for-smartphones/`.

[106] thttpd. http://www.acme.com/software/thttpd/.

[107] Thomas Toth and Christopher Kruegel. Accurate Buffer Overflow Detection via Abstract Payload Execution. In *Proc. RAID*, 2002.

[108] Trend Micro. Android Malware: How Worried Should You Be?, 16 July 2012. http://blog.trendmicro.com/android-malware-how-worried-should-you-be/.

[109] US-CERT Vulnerability Notes Database. http://www.kb.cert.org/vuls.

[110] Neil Vachharajani, Matthew J. Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A. Blome, George A. Reis, Manish Vachharajani, and David August. RIFLE: An architectural framework for user-centric information-flow security. In *MICRO*, Dec 2004.

[111] Victor van der Veen, Nitish dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. Memory Errors: The Past, the Present, and the Future. In *In the Proceedings of the 15th International Symposium on Research in Attacks Intrusions and Defenses (RAID)*, September 2012.

[112] Vulnerability Note VU#492515:Microsoft Internet Explorer HTML object memory corruption vulnerability. http://www.kb.cert.org/vuls/id/492515.

[113] Xinran Wang, Yoon-Chan Jhi, Sencun Zhu, and Peng Liu. STILL: Exploit Code Detection via Static Taint and Initialization Analyses. In *Proc. ACSAC*, 2008.

[114] Xinran Wang, Chi-Chun Pan, Peng Liu, and Sencun Zhu. SigFree: A Signature-Free Buffer Overflow Attack Blocker. In *Proc. USENIX Security*, 2006.

[115] WebKit. http://webkit.org.

[116] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent Runtime Randomization for Security. In *Proc. Int'l. Symp. on Reliable Distributed Systems*, 2003.

[117] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security*, Aug 2006.

[118] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *ACM CCS*, 2007.

[119] D. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. Privacy Scope: A Precise Information Flow Tracking System for Finding Application Leaks. Technical report, Dept. of Computer Science, UC Berkeley, 2009.