SigFree: A Signature-Free Buffer Overflow Attack Blocker

Xinran Wang, Chi-Chun Pan, Peng Liu, and Sencun Zhu

Abstract—We propose SigFree, an online signature-free out-of-the-box application-layer method for blocking code-injection buffer overflow attack messages targeting at various Internet services such as web service. Motivated by the observation that buffer overflow attacks typically contain executables whereas legitimate client requests never contain executables in most Internet services, SigFree blocks attacks by detecting the presence of code. Unlike the previous code detection algorithms, SigFree uses a new data-flow analysis technique called *code abstraction* that is generic, fast, and hard for exploit code to evade. SigFree is signature free, thus it can block new and unknown buffer overflow attacks; SigFree is also immunized from most attack-side code obfuscation methods. Since SigFree is a transparent deployment to the servers being protected, it is good for economical Internet-wide deployment with very low deployment and maintenance cost. We implemented and tested SigFree; our experimental study shows that the dependency-degree-based SigFree could block all types of code-injection attack packets (above 750) tested in our experiments with very few false positives. Moreover, SigFree causes very small extra latency to normal client requests when some requests contain exploit code.

Index Terms—Intrusion detection, buffer overflow attacks, code-injection attacks.

1 Introduction

Throughout the history of cyber security, buffer overflow is one of the most serious vulnerabilities in computer systems. Buffer overflow vulnerability is a root cause for most of the cyber attacks such as server breaking-in, worms, zombies, and botnets. A buffer overflow occurs during program execution when a fixed-size buffer has had too much data copied into it. This causes the data to overwrite into adjacent memory locations, and depending on what is stored there, the behavior of the program itself might be affected [1]. Although taking a broader viewpoint, buffer overflow attacks do not always carry binary code in the attacking requests (or packets),[1] code-injection buffer overflow attacks such as stack smashing probably count for most of the buffer overflow attacks that have happened in the real world.

Although tons of research has been done to tackle buffer overflow attacks, existing defenses are still quite limited in meeting four highly desired requirements: (R1) *simplicity* in maintenance; (R2) *transparency* to existing (legacy) server OS, application software, and hardware; (R3) *resiliency* to obfuscation; (R4) economical Internet-wide deployment. As a result, although several very secure solutions have been proposed, they are not pervasively deployed, and a considerable number of buffer overflow attacks continue to succeed on a daily basis.

To see how existing defenses are limited in meeting these four requirements, let us break down the existing buffer overflow defenses into six classes, which we will review shortly in Section 2: (1A) Finding bugs in source code. (1B) Compiler extensions. (1C) OS modifications. (1D) Hardware modifications. (1E) Defense-side obfuscation [3], [4]. (1F) Capturing code running symptoms of buffer overflow attacks [5], [6], [7], [8]. (Note that the above list does not include binary-code-analysis-based defenses, which we will address shortly.) We may briefly summarize the limitations of these defenses in terms of the four requirements as follows: 1) Class 1B, 1C, 1D, and 1E defenses may cause substantial changes to existing (legacy) server OSes, application software, and hardware, thus they are not transparent. Moreover, Class 1E defenses generally cause processes to be terminated. As a result, many businesses do not view these changes and the process termination overhead as economical deployment. 2) Class 1F defenses can be very secure, but they either suffer from significant runtime overhead or need special auditing or diagnosis facilities, which are not commonly available in commercial services. As a result, Class 1F defenses have limited transparency and potential for economical deployment. 3) Class 1A defenses need source code, but source code is unavailable to many legacy applications.

Besides buffer overflow defenses, worm signatures can be generated and used to block buffer overflow attack packets [9], [10], [11]. Nevertheless, they are also limited in meeting the four requirements, since they either rely on signatures, which introduce maintenance overhead, or are not very resilient to attack-side obfuscation.

1. A buffer overflow attack may corrupt control flow or data without injecting code such as return-to-libc attacks and data-pointer modification [2]. In this paper, we only focus on code-injection buffer overflow attacks.

- *X. Wang is with the Department of Computer Science and Engineering, Pennsylvania State University, 344 Information Science and Technology Building, State College, PA 16802. E-mail: xinrwang@cse.psu.edu.*
- *C.-C. Pan and P. Liu are with the College of Information Sciences and Technology, Pennsylvania State University, Information Science and Technology Building, State College, PA 16802. E-mail: {cpan, pliu}@ist.psu.edu.*
- *S. Zhu is with the Department of Computer Science and Engineering and the College of Information Sciences and Technology, Pennsylvania State University, 338F Information Science and Technology Building, State College, PA 16802. E-mail: szhu@cse.psu.edu.*

Manuscript received 29 Mar. 2007; revised 25 Jan. 2008; accepted 28 Mar. 2008; published online 8 May 2008.
For information on obtaining reprints of this article, please send e-mail to: tdsc@computer.org, and reference IEEECS Log Number TDSC-2007-03-0046.
Digital Object Identifier no. 10.1109/TDSC.2008.30.

1545-5971/08/$25.00 © 2008 IEEE Published by the IEEE Computer Society

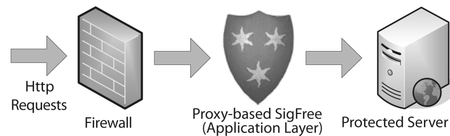

Fig. 1. SigFree is an application layer blocker between the protected server and the corresponding firewall.

To overcome the above limitations, in this paper, we propose SigFree, an online buffer overflow attack blocker, to protect Internet services. The idea of SigFree is motivated by an important observation that "the nature of communication to and from network services is predominantly or exclusively data and not executable code" [12]. In particular, as summarized in [12], 1) on Windows platforms, most web servers (port 80) accept data only; remote access services (ports 111, 137, 138, 139) accept data only; Microsoft SQL Servers (port 1434), which are used to monitor Microsoft SQL Databases, accept data only. 2) On Linux platforms, most Apache web servers (port 80) accept data only; BIND (port 53) accepts data only; SNMP (port 161) accepts data only; most Mail Transport (port 25) accepts data only; Database servers (Oracle, MySQL, PostgreSQL) at ports 1521, 3306, and 5432 accept data only.

Since remote exploits are typically binary executable code, this observation indicates that if we can precisely distinguish (service requesting) messages containing binary code from those containing no binary code, we can protect most Internet services (which accept data only) from code-injection buffer overflow attacks by blocking the messages that contain binary code.

Accordingly, SigFree (Fig. 1) works as follows: SigFree is an application layer blocker that typically stays between a service and the corresponding firewall. When a service requesting message arrives at SigFree, SigFree first uses a new $O(N)$ algorithm, where N is the byte length of the message, to disassemble and distill all possible instruction sequences from the message's payload, where every byte in the payload is considered as a possible starting point of the code embedded (if any). However, in this phase, some data bytes may be mistakenly decoded as instructions. In phase 2, SigFree uses a novel technique called *code abstraction*. Code abstraction first uses data flow anomaly to prune useless instructions in an instruction sequence, then compares the number of useful instructions (Scheme 2) or *dependence degree* (Scheme 3) to a threshold to determine if this instruction sequence (distilled in phase 1) contains code. Unlike the existing code detection algorithms [12], [13], [14] that are based on signatures, rules, or control flow detection, SigFree is generic and hard for exploit code to evade (Section 2 gives a more detailed comparison).

We have implemented a SigFree prototype as a proxy to protect web servers. Our empirical study shows that there exists clean-cut "boundaries" between code-embedded payloads and data payloads when our code-data separation criteria are applied. We have identified the "boundaries" (or thresholds) and been able to detect/block all 50 attack packets generated by Metasploit framework [15], all 700 polymorphic shellcode packets generated by polymorphic shellcode engines Countdown [15], JumpCallAdditive [15], JempiScodes [16], ADMmutate [17] and CLET [18], and worm Slammer, CodeRed and a CodeRed variation, when they are well mixed with various types of data packets. In addition, our experiment results show that the extra processing delay caused by SigFree to client requests is negligible.

The merits of SigFree are summarized as follows: they show that SigFree has taken a main step forward in meeting the four requirements aforementioned:

- *SigFree is signature free, thus it can block new and unknown buffer overflow attacks.*
- Without relying on string matching, *SigFree is immunized from most attack-side obfuscation methods.*
- *SigFree uses generic code-data separation criteria instead of limited rules.* This feature separates SigFree from [12], an independent work that tries to detect code-embedded packets.
- *Transparency.* SigFree is an out-of-the-box solution that requires no server side changes.
- *SigFree is an economical deployment with very low maintenance cost*, which can be well justified by the aforementioned features.

The rest of this paper is organized as follows: In Section 2, we summarize the work related to ours. In Section 3, we give an overview of SigFree. In Sections 4 and 5, we introduce the instruction sequence distiller component and the instruction sequence analyzer component of SigFree, respectively. In Section 6, we show our experimental results. Finally, we discuss some remaining research issues in Section 7 and conclude this paper in Section 8.

2 Related Work

SigFree is mainly related to three bodies of work. [Category 1] Prevention/detection techniques of buffer overflows; [Category 2] worm detection and signature generation; [Category 3] machine code analysis for security purposes. In the following, we first briefly review Category 1 and Category 2, which are less close to SigFree. Then, we will focus on comparing SigFree with Category 3.

2.1 Prevention/Detection of Buffer Overflows

Existing prevention/detection techniques of buffer overflows can be roughly broken down into six classes:

Class 1A: Finding bugs in source code. Buffer overflows are fundamentally due to programming bugs. Accordingly, various bug-finding tools [19], [20], [21] have been developed. The bug-finding techniques used in these tools, which in general belong to static analysis, include but are not limited to model checking and bugs-as-deviant-behavior. Class 1A techniques are designed to handle source code only, and they do not ensure completeness in bug finding. In contrast, SigFree handles machine code embedded in a request (message).

Class 1B: Compiler extensions. "If the source code is available, a developer can add buffer overflow detection automatically to a program by using a modified compiler" [1]. Three such compilers are StackGuard [22], ProPolice [23], and Return Address Defender (RAD) [24]. DIRA [25] is another compiler that can detect control hijacking attacks, identify

the malicious input, and repair the compromised program. Class 1B techniques require the availability of source code. In contrast, SigFree does not need to know any source code.

Class 1C: OS modifications. Modifying some aspects of the operating system may prevent buffer overflows such as Pax [26], LibSafe [27], and e-NeXsh [28]. Class 1C techniques need to modify the OS. In contrast, SigFree does not need any modification of the OS.

Class 1D: Hardware modifications. A main idea of hardware modification is to store all return addresses on the processor [29]. In this way, no input can change any return address.

Class 1E: Defense-side obfuscation. Address Space Layout Randomization (ASLR) is a main component of PaX [26]. Address-space randomization, in its general form [30], can detect exploitation of all memory errors. Instruction set randomization [3], [4] can detect all code-injection attacks, whereas SigFree cannot guarantee detecting all injected code. Nevertheless, when these approaches detect an attack, the victim process is typically terminated. "Repeated attacks will require repeated and expensive application restarts, effectively rendering the service unavailable" [7].

Class 1F: Capturing code running symptoms of buffer overflow attacks. Fundamentally, buffer overflows are a code running symptom. If such unique symptoms can be precisely captured, all buffer overflows can be detected. Class 1B, Class 1C, and Class 1E techniques can capture some—but not all—of the running symptoms of buffer overflows. For example, accessing nonexecutable stack segments can be captured by OS modifications; compiler modifications can detect return address rewriting; and process crash is a symptom captured by defense-side obfuscation. To achieve 100 percent coverage in capturing buffer overflow symptoms, dynamic data flow/taint analysis/program shepherding techniques were proposed in Vigilante [6], TaintCheck [5], and [31]. They can detect buffer overflows during runtime. However, it may cause significant runtime overhead (e.g., 1,000 percent). To reduce such overhead, another type of Class 1F techniques, namely postcrash symptom diagnosis, has been developed in Covers [7] and [8]. Postcrash symptom diagnosis extracts the "signature" after a buffer overflow attack is detected. A more recent system called ARBOR [32] can automatically generate vulnerability-oriented signatures by identifying characteristic features of attacks and using program context. Moreover, ARBOR automatically invokes the recovery actions. Class 1F techniques can block both the attack requests that contain code and the attack requests that do not contain any code, but they need the signatures to be firstly generated. Moreover, they either suffer from significant runtime overhead or need special auditing or diagnosis facilities, which are not commonly available in commercial services. In contrast, although SigFree could not block the attack requests that do not contain any code, SigFree is signature free and does not need any changes to real-world services. We will investigate the integration of SigFree with Class 1F techniques in our future work.

2.2 Worm Detection and Signature Generation

Because buffer overflow is a key target of worms when they propagate from one host to another, SigFree is related to worm detection. Based on the nature of worm infection symptoms, worm detection techniques can be broken down into three classes: [Class 2A] techniques use such macro symptoms as Internet background radiation (observed by network telescopes) to raise early warnings of Internet-wide worm infection [33]. [Class 2B] techniques use such local traffic symptoms as content invariance, content prevalence, and address dispersion to generate worm signatures and/or block worms. Some examples of Class 2B techniques are Earlybird [9], Autograph [10], Polygraph [11], Hamsa [34], and Packet Vaccine [35]. [Class 2C] techniques use worm code running symptoms to detect worms. It is not surprising that Class 2C techniques are exactly Class 1F techniques. Some examples of Class 2C techniques are Shield [36], Vigilante [6], and COVERS [7]. [Class 2D] techniques use anomaly detection on packet payload to detect worms and generate signature. Wang and Stolfo [37], [38] first proposed Class 2D techniques called PAYL. PAYL is first trained with normal network flow traffic and then uses some byte-level statistical measures to detect exploit code.

Class 2A techniques are not relevant to SigFree. Class 2C techniques have already been discussed. Class 2D techniques could be evaded by statistically simulating normal traffic [39]. Class 2B techniques rely on signatures, while SigFree is signature free. Class 2B techniques focus on identifying the unique bytes that a worm packet must carry, while SigFree focuses on determining if a packet contains code or not. Exploiting the content invariance property, Class 2B techniques are typically not very resilient to obfuscation. In contrast, SigFree is immunized from most attack-side obfuscation methods.

2.3 Machine Code Analysis for Security Purposes

Although source code analysis has been extensively studied (see Class 1A), in many real-world scenarios, source code is not available and the ability to analyze binaries is desired. Machine code analysis has three main security purposes: (P1) malware detection, (P2) to analyze obfuscated binaries, and (P3) to identify and analyze the code contained in buffer overflow attack packets. Along purpose P1, Chritodorescu and Jha [40] proposed static analysis techniques to detect malicious patterns in executables, and Chritodorescu et al. [41] exploited semantic heuristics to detect obfuscated malware. Along purpose P2, Lakhotia and Eric [42] used static analysis techniques to detect obfuscated calls in binaries, and Kruegel et al. [43] investigated disassembly of obfuscated binaries.

SigFree differs from P1 and P2 techniques in design goals. The purpose of SigFree is to see if a message contains code or not, not to determine if a piece of code has malicious intent or not. Hence, SigFree is immunized from most attack-side obfuscation methods. Nevertheless, both the techniques in [43] and SigFree disassemble binary code, although their disassembly procedures are different. As will be seen, disassembly is not the kernel contribution of SigFree.

Fnord [44], the preprocessor of Snort IDS, identifies exploit code by detecting NOP sled. Binary disassembly is also used to find the sequence of execution instructions as an evidence of an NOP sled [13]. However, some attacks such as worm CodeRed do not include NOP sled and, as mentioned in [12], mere binary disassembly is not adequate.

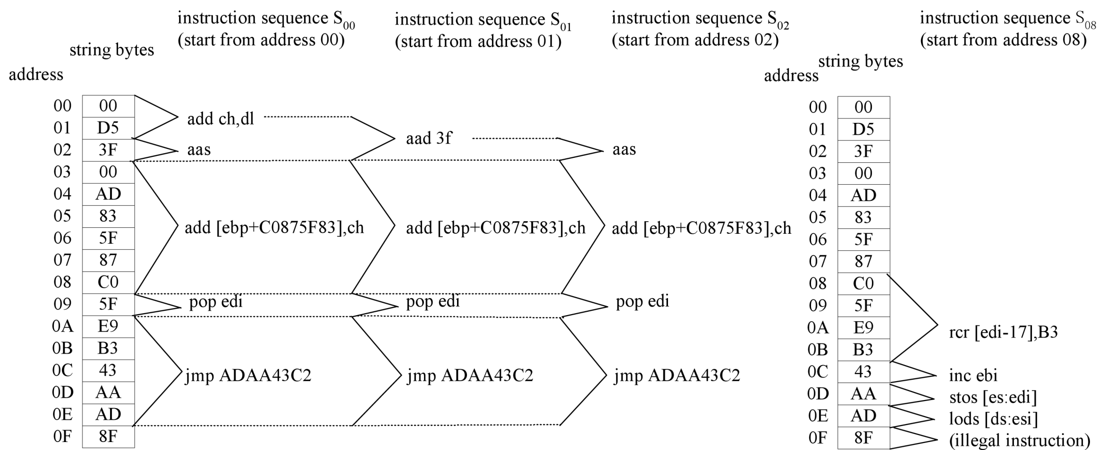

Fig. 2. Instruction sequences distilled from a substring of a GIF file. We assign an address to every byte of the string. Instruction sequences s_{00}, s_{01}, s_{02}, and s_{08} are distilled by disassembling the string from addresses 00, 01, 02, and 08, respectively.

Moreover, polymorphic shellcode [17], [18] can bypass the detection of NOP instructions by introducing fake NOP zone. SigFree does not rely on the detection of NOP sled.

Finally, being generally a P3 technique, SigFree is most relevant to two P3 works [12], [14]. Kruegel et al. [14] innovatively exploited control flow structures to detect polymorphic worms. Unlike string-based signature matching, their techniques identify structural similarities between different worm mutations and use these similarities to detect more polymorphic worms. The implementation of their approach is resilient to a number of code transformation techniques. Although their techniques also handle binary code, they perform offline analysis. In contrast, SigFree is an online attack blocker. As such, their techniques and SigFree are complementary to each other with different purposes. Moreover, unlike SigFree, their techniques [14] may not be suitable to block the code contained in *every* attack packet, because some buffer overflow code is so simple that very little control flow information can be exploited.

Independent of our work, Chinchani and Berg [12] proposed a rule-based scheme to achieve the same goal as that of SigFree, that is, to detect exploit code in network flows. However, there is a fundamental difference between SigFree and their scheme [12]. Their scheme is rule-based, whereas SigFree is a *generic* approach that does not require any preknown patterns. More specifically, their scheme [12] first tries to find certain preknown instructions, instruction patterns, or control flow structures in a packet. Then, it uses the found patterns and a data flow analysis technique called program slicing to analyze the packet's payload to check if the packet really contains code. Four rules (or cases) are discussed in their paper: Case 1 not only assumes the occurrence of the call/jmp instructions but also expects that the push instruction appears before the branch; Case 2 relies on the *interrupt* instruction; Case 3 relies on instruction *ret*; Case 4 exploits hidden branch instructions. Besides, they used a special rule to detect polymorphic exploit code that contains a loop. Although they mentioned that the above rules are initial sets and may require updating over time, it is always possible for attackers to bypass those preknown rules. Moreover, more rules mean more overhead and longer latency in filtering packets. In contrast, SigFree exploits a different data flow analysis technique, which is much harder for exploit code to evade.

3 SigFree Overview

3.1 Basic Definitions and Notations

This section provides the definitions that will be used in the rest of this paper.

Definition 1 (instruction sequence). *An instruction sequence is a sequence of CPU instructions, which has one and only one entry instruction and there exists at least one execution path from the entry instruction to any other instruction in this sequence.*

A fragment of a program in machine language is an instruction sequence, but an instruction sequence is not necessarily a fragment of a program. In fact, we may distill instruction sequences from any binary strings. This poses the fundamental challenge to our research goal. Fig. 2 shows four instruction sequences distilled from a substring of a GIF file. Each instruction sequence is denoted as s_i in Fig. 2, where i is the entry location of the instruction sequence in the string. These four instruction sequences are not fragments of a real program, although they may also be executed in a specific CPU. Below, we call them *random instruction sequences*, whereas use the term *binary executable code* to refer to a fragment of a real program in machine language.

Definition 2 (instruction flow graph). *An instruction flow graph (IFG) is a directed graph $G = (V, E)$ where each node $v \in V$ corresponds to an instruction and each edge $e = (v_i, v_j) \in E$ corresponds to a possible transfer of control from instruction v_i to instruction v_j.*

Unlike traditional control flow graph (CFG), a node of an IFG corresponds to a single instruction rather than a basic block of instructions. To completely model the control flow of an instruction sequence, we further extend the above definition.

Definition 3 (extended IFG). *An extended IFG (EIFG) is a directed graph $G = (V, E)$, which satisfies the following properties: each node $v \in V$ corresponds to an instruction,*

an illegal instruction (an "instruction" that cannot be recognized by CPU), or an external address (a location that is beyond the address scope of all instructions in this graph); each edge $e = (v_i, v_j) \in E$ corresponds to a possible transfer of control from instruction v_i to instruction v_j, to illegal instruction v_j, or to an external address v_j.

Accordingly, we name the types of nodes in an EIFG *instruction node*, *illegal instruction node*, and *external address node*.

The reason that we define IFG and EIFG is to model two special cases, which CFG cannot model (the difference will be very evident in the following sections). First, in an instruction sequence, control may be transferred from an instruction node to an illegal instruction node. For example, in instruction sequence s_{08} in Fig. 2, the transfer of control is from instruction "lods [ds:esi]" to an illegal instruction at address $0F$. Second, control may be transferred from an instruction node to an external address node. For example, instruction sequence s_{00} in Fig. 2 has an instruction "jmp ADAAC3C2," which jumps to external address ADAAC3C2.

3.2 Assumptions

In this paper, we focus on buffer overflow attacks whose payloads contain executable code in machine language, and we assume normal requests do not contain executable machine code. A normal request may contain any data, parameters, scripts, or even an SQL statement. Note that although SQL statements and scripts are executable in the application level, they cannot be executed directly by a CPU. As such, SQL statements and scripts are not viewed as executable in our model. Application level attacks such as data manipulation and SQL injection are out of the scope.

Though SigFree is a generic technique that can be applied to any instruction set, for concreteness, we assume the web server runs the Intel IA-32 instruction set, the most popular instruction set running inside a web server today.

4 Instruction Sequence Distiller

This section first describes an effective algorithm to distill instruction sequences from requests, followed by several pruning techniques to reduce the processing overhead of instruction sequence analyzer.

4.1 Distilling Instruction Sequences

To distill an instruction sequence, we first assign an address (starting from zero) to every byte of a request, where address is an identifier for each location in the request. Then, we disassemble the request from a certain address until the end of the request is reached or an illegal instruction opcode is encountered. There are two traditional disassembly algorithms: *linear sweep* and *recursive traversal* [45], [46]. The linear sweep algorithm begins disassembly at a certain address and proceeds by decoding each encountered instruction. The recursive traversal algorithm also begins disassembly at a certain address, but it follows the control flow of instructions.

In this paper, we employ the recursive traversal algorithm, because it can obtain the control flow information during the disassembly process. Intuitively, to get all possible instruction sequences from an N-byte request, we simply execute the disassembly algorithm N times and each time we start from a different address in the request. This gives us a set of instruction sequences. The running time of this algorithm is $O(N^2)$.

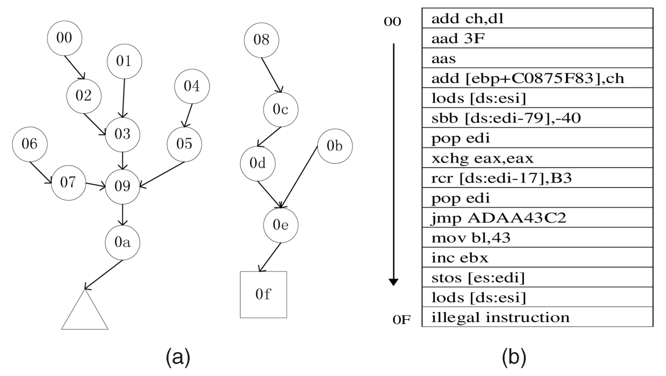

Fig. 3. Data structure for the instruction sequences distilled from the request in Fig. 2. (a) EIFG. Circles represent instruction nodes; triangles represent external addresses; rectangles represent illegal instructions. (b) The array of all possible instructions in the request.

One drawback of the above algorithm is that the same instructions are decoded many times. For example, instruction "pop edi" in Fig. 2 is decoded many times by this algorithm. To reduce the running time, we design a memorization algorithm [47] by using a data structure, which is an EIFG defined earlier, to represent the instruction sequences. To distill all possible instruction sequences from a request is simply to create the EIFG for the request. An EIFG is used to represent all possible transfers of control among these instructions. In addition, we use an instruction array to represent all possible instructions in a request. To traverse an instruction sequence, we simply traverse the EIFG from the entry instruction of the instruction sequence and fetch the corresponding instructions from the instruction array. Fig. 3 shows the data structure for the request shown in Fig. 2. The details of the algorithm for creating the data structure are described in Algorithm 1. Clearly, the running time of this algorithm is $O(N)$, which is optimal as each address is traversed only once.

Algorithm 1 Distill all instruction sequences from a request

```
initialize EISG G and instruction array A to empty
for each address i of the request do
    add instruction node i to G
i ← the start address of the request
while i <= the end address of the request do
    inst ← decode an instruction at i
    if inst is illegal then
        A[i] ← illegal instruction inst
        set type of node i "illegal node" in G
    else
        A[i] ← instruction inst
        if inst is a control transfer instruction then
            for each possible target t of inst do
                if target t is an external address then
                    add external address node t to G
                add edge e(node i, node t) to G
        else
            add edge e(node i, node i + inst.length) to G
    i ← i + 1
```

4.2 Excluding Instruction Sequences

The previous step may output many instruction sequences at different entry points. Next, we exclude some of them based on several heuristics. Here, *excluding an instruction sequence means that the entry of this sequence is not considered as the real entry for the embedded code (if any)*.

The fundamental rule in excluding instruction sequences is not to affect the decision whether a request contains code or not. This rule can be translated into the following technical requirements: if a request contains a fragment of a program, the fragment must be one of the remaining instruction sequences or a subsequence of a remaining instruction sequence, or it differs from a remaining sequence only by few instructions.

Step 1. If instruction sequence s_a is a subsequence of instruction sequence s_b, we exclude s_a. The rationale for excluding s_a is that if s_a satisfies some characteristics of programs, s_b also satisfies these characteristics with a high probability.

This step helps exclude lots of instruction sequences since many distilled instruction sequences are subsequences of the other distilled instruction sequences. For example, in Fig. 3a, instruction sequence s_{02}, which is a subsequence of instruction sequence s_{00}, can be excluded. Note that here we only exclude instruction sequence s_{02} rather than remove node v_{02}. Similarly, instruction sequences s_{03}, s_{05}, s_{07}, s_{09}, s_{0a}, s_{0c}, s_{0d}, and s_{0e} can be excluded.

Step 2. If instruction sequence s_a merges to instruction sequence s_b after a few instructions (e.g., 4 in our experiments) and s_a is no longer than s_b, we exclude s_a. It is reasonable to expect that s_b will preserve s_a's characteristics.

Many distilled instruction sequences are observed to merge to other instruction sequences after a few instructions. This property is called self-repairing [46] in Intel IA-32 architecture. For example, in Fig. 3a, instruction sequence s_{01} merges to instruction sequence s_{00} only after one instruction. Therefore, s_{01} is excluded. Similarly, instruction sequences s_{04}, s_{06}, and s_{0b} can be excluded.

Step 3. For some instruction sequences, when they are executed, whichever execution path is taken, an illegal instruction is *inevitably reached*. We say an instruction is inevitably reached if two conditions hold. One is that there are no cycles (loops) in the EIFG of the instruction sequence; the other is that there are no external address nodes in the EIFG of the instruction sequence.

We exclude the instruction sequences in which illegal instructions are inevitably reached, because causing the server to execute an illegal instruction is not the purpose of a buffer overflow attack (this assumption was also made by others [12], [14], implicitly or explicitly). Note that however the existence of illegal instruction nodes cannot always be used as a criteria to exclude an instruction sequence unless they are inevitably reached; otherwise, attackers may obfuscate their program by adding *nonreachable* illegal instructions.

Based on this heuristic, we can exclude instruction sequence s_{08} in Fig. 3a, since it will eventually execute an illegal instruction v_{0f}.

After these three steps, in Fig. 3a, only instruction sequence s_{00} is left for consideration in the next stage.

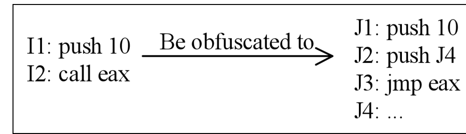

Fig. 4. An obfuscation example. Instruction "call eax" is substituted by "push J4" and "jmp eax."

5 Instruction Sequences Analyzer

A distilled instruction sequence may be a sequence of random instructions or a fragment of a program in machine language. In this section, we propose three schemes to differentiate these two cases. Scheme 1 exploits the operating system characteristics of a program; Scheme 2 and Scheme 3 exploit the data flow characteristics of a program. Scheme 1 is slightly faster than Scheme 2 and Scheme 3, whereas Scheme 2 and Scheme 3 are much more robust to obfuscation.

5.1 Scheme 1

A program in machine language is dedicated to a specific operating system; hence, a program has certain characteristics implying the operating system on which it is running, for example calls to operating system or kernel library. A random instruction sequence does not carry this kind of characteristics. By identifying the call pattern in an instruction sequence, we can effectively differentiate a real program from a random instruction sequence.

More specifically, instructions such as "call" and "int 0x2eh" in Windows and "int 0x80h" in Linux may indicate system calls or function calls. However, since the op-codes of these call instructions are only 1 byte, even normal requests may contain plenty of these byte values. Therefore, using the number of these instructions as a criterion will cause a high false positive rate. To address this issue, we use a pattern composed of several instructions rather than a single instruction. It is observed that before these call instructions there are normally one or several instructions used to transfer parameters. For example, a "push" instruction is used to transfer parameters for a "call" instruction; some instructions that set values to registers al, ah, ax, or eax are used to transfer parameters for "int" instructions. These call patterns are very common in a fragment of a real program. Our experiments in Section 6 show that by selecting the appropriate parameters we can rather accurately tell whether an instruction sequence is an executable code or not.

Scheme 1 is fast since it does not need to fully disassemble a request. For most instructions, we only need to know their types. This saves a lot of time in decoding operands of instructions.

Note that although Scheme 1 is good at detecting most of the known buffer overflow attacks, it is vulnerable to obfuscation. One possible obfuscation is that attackers may use other instructions to replace the "call" and "push" instructions. Fig. 4 shows an example of obfuscation, where "call eax" instruction is substituted by "push J4" and "jmp eax." Although we cannot fully solve this problem, by recording this kind of instruction replacement patterns, we may still be able to detect this type of obfuscation to some extent.

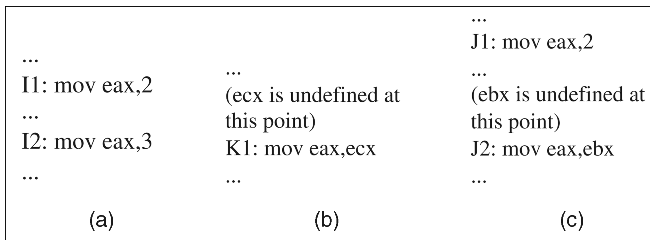

Fig. 5. Data flow anomaly in execution paths. (a) Define-define anomaly. Register eax is defined at I1 and then defined again at I2. (b) Undefine-reference anomaly. Register ecx is undefined before K1 and referenced at K1. (c) Define-undefine anomaly. Register eax is defined at J1 and then undefined at J2.

5.2 Scheme 2

Next, we propose Scheme 2 to detect the aforementioned obfuscated buffer overflow attacks. Scheme 2 exploits the data flow characteristics of a program. Normally, a random instruction sequence is full of data flow anomalies, whereas a real program has few or no data flow anomalies. However, the number of data flow anomalies cannot be directly used to distinguish a program from a random instruction sequence because an attacker may easily obfuscate his program by introducing enough data flow anomalies.

In this paper, we use the detection of data flow anomaly in a different way called *code abstraction*. We observe that when there are data flow anomalies in an execution path of an instruction sequence, some instructions are useless, whereas in a real program at least one execution path has a certain number of useful instructions. Therefore, if the number of useful instructions in an execution path exceeds a threshold, we conclude the instruction sequence is a segment of a program.

Data flow anomaly. The term data flow anomaly was originally used to analyze programs written in higher level languages in the software reliability and testing field [48], [49]. In this paper, we borrow this term and several other ones to analyze instruction sequences.

During a program execution, an instruction may impact a variable (register, memory location, or stack) on three different ways: *define*, *reference*, and *undefine*. A variable is defined when it is set a value; it is referenced when its value is referred to; it is undefined when its value is not set or set by another undefined variable. Note that here the definition of undefined is different from that in a high level language. For example, in a C program, a local variable of a block becomes undefined when control leaves the block.

A data flow anomaly is caused by an improper sequence of actions performed on a variable. There are three data flow anomalies: *define-define*, *define-undefine*, and *undefine-reference* [49]. The define-define anomaly means that a variable was defined and is defined again, but it has never been referenced between these two actions. The undefine-reference anomaly indicates that a variable that was undefined receives a reference action. The define-undefine anomaly means that a variable was defined, and before it is used it is undefined. Fig. 5 shows an example.

Detection of data flow anomalies. There are static [48] or dynamic [49] methods to detect data flow anomalies in the software reliability and testing field. Static methods are

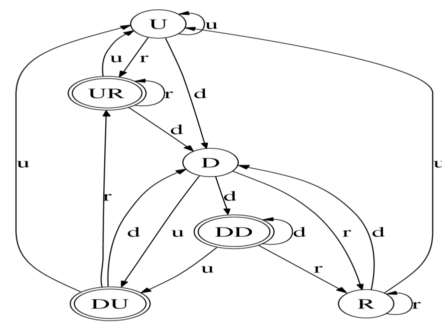

Fig. 6. State diagram of a variable. State U: undefined, state D: defined but not referenced, state R: defined and referenced, state DD: abnormal state define-define, state UR: abnormal state undefine-reference, and state DU: abnormal state define-undefine.

not suitable in our case due to its slow speed; dynamic methods are not suitable either due to the need for real execution of a program with some inputs. As such, we propose a new method called code abstraction, which does not require real execution of code. As a result of the code abstraction of an instruction, a variable could be in one of the six possible states. The six possible states are state U: undefined; state D: defined but not referenced; state R: defined and referenced; state DD: abnormal state define-define; state UR: abnormal state undefine-reference; and state DU: abnormal state define-undefine. Fig. 6 depicts the state diagram of these states. Each edge in this state diagram is associated with d, r, or u, which represents "define," "reference," and "undefine," respectively.

We assume that a variable is in "undefined" state at the beginning of an execution path. Now, we start to traverse this execution path. If the entry instruction of the execution path defines this variable, it will enter the state "defined." Then, it will enter another state according to the next instruction, as shown in Fig. 6. Once the variable enters an abnormal state, a data flow anomaly is detected. We continue this traversal to the end of the execution path. This process enables us to find all the data flow anomalies in this execution path.

Pruning useless instructions. Next, we leverage the detected data flow anomalies to remove useless instructions. A *useless* instruction of an execution path is an instruction that does not affect the results of the execution path; otherwise, it is called *useful* instructions. We may find a useless instruction from a data flow anomaly. When there is an undefine-reference anomaly in an execution path, the instruction that causes the "reference" is a useless instruction. For instance, the instruction $K1$ in Fig. 5, which causes undefine-reference anomaly, is a useless instruction. When there is a define-define or define-undefine anomaly, the instruction that caused the former "define" is also considered as a useless instruction. For instance, the instructions $I1$ and $J1$ in Fig. 5 are useless instructions because they caused the former "define" in either the define-define or the define-undefine anomaly.

After pruning the useless instructions from an execution path, we will get a set of useful instructions. If the number of useful instructions in an execution path exceeds a threshold, we will conclude the instruction sequence is a

segment of a program. Algorithm 2 shows our algorithm to check if the number of useful instructions in an execution path exceeds a threshold. The algorithm involves a search over an EISG in which the nodes are visited in a specific order derived from a depth first search. The algorithm assumes that an EISG G and the entry instruction of the instruction sequence are given, and a push down stack is available for storage. During the search process, the visited node (instruction) is abstractly executed to update the states of variables, find data flow anomaly, and prune useless instructions in an execution path.

Algorithm 2 check if the number of useful instructions in an execution path exceeds a threshold
Input: *entry* instruction of an instruction sequence, EISG G

$total \leftarrow 0$; $useless \leftarrow 0$; $stack \leftarrow$ empty
initialize the $states$ of all variables to "undefined"
push the entry $instruction$, $states$, $total$, and $useless$ to $stack$
while $stack$ is not empty **do**
  pop the top item of $stack$ to i, $states$, $total$, and $useless$
  if $total - useless$ greater than a threshold **then**
    return true
  if i is visited **then**
    continue (passes control to the next iteration of the WHILE loop)
  mark i visited
  $total \leftarrow total + 1$
  Abstractly execute instruction i (change the $states$ of variables according to instruction i)
  if there is a define-define or define-undefine anomaly **then**
    $useless \leftarrow useless + 1$
  if there is an undefine-reference anomaly **then**
    $useless \leftarrow useless + 1$
  for each instruction j directly following i in the G **do**
    push j, $states$, $total$, and $useless$ to $stack$
return false

Handling special cases. Next, we discuss several special cases in the implementation of Scheme 2.

General purpose instruction. The instructions in the IA-32 instruction set can be roughly divided into four groups: general purpose instructions, floating point unit instructions, extension instructions, and system instructions. General purpose instructions perform basic data movement, arithmetic, logic, program flow, and string operation, which are commonly used by programmers to write applications and system software that run on IA-32 processors [50]. General purpose instructions are also the most often used instructions in malicious code. We believe that malicious codes must contain a certain number of general purpose instructions to achieve the attacking goals. Other types of instructions may be leveraged by an attacker to obfuscate his real-purpose code, e.g., used as garbage in garbage insertion. As such, we consider other groups of instructions as useless instructions.

Initial state of registers. For registers, we set their initial states to "undefined" at the beginning of an execution path.

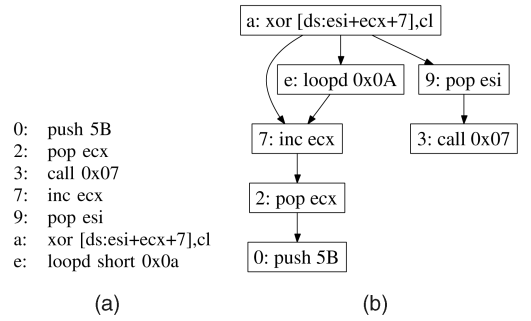

Fig. 7. (a) A decryption routine that only has seven useful instructions. (b) A def-use graph of the decryption routine. Instruction a can reach all other instructions through paths $a \rightarrow e \rightarrow 7 \rightarrow 2 \rightarrow 0$ and $a \rightarrow 9 \rightarrow 3$, and therefore, the dependence degree of instruction a is 6.

The register "esp," however, is an exception since it is used to hold the stack pointer. Thus, we set register esp "defined" at the beginning of an execution path.

Indirect address. An indirect address is an address that serves as a reference point instead of an address to the direct memory location. For example, in the instruction "move eax,[ebx+01e8]," register "ebx" may contain the actual address of the operand. However, it is difficult to know the runtime value of register "ebx." Thus, we always treat a memory location to which an indirect address points as state "defined" and hence no data flow anomaly will be generated. Indeed, this treatment successfully prevents an attacker from obfuscating his code using indirect addresses.

Useless control transfer instructions (CTIs). A CTI is useless if the corresponding control condition is undefined or the control transfer target is undefined. The status flags of the EFLAGS register are used as control condition in the IA-32 architecture. These status flags indicate the results of arithmetic and shift instructions, such as the ADD, SUB, and SHL instructions. Condition instructions Jcc (jump on condition code cc) and LOOPcc use one or more of the status flags as condition codes and test them for branch or end-loop conditions. During a program execution at runtime, an instruction may affect a status flag on three different ways: *set*, *unset*, or *undefine* [50]. We consider both *set* and *unset* are defined in code abstraction. As a result, a status flag could be in one of the two possible states—defined or undefined in code abstraction. To detect and prune useless CTIs, we also assume that the initial states of all status flags are undefined. The states of status flags are updated during the process of code abstraction.

The objective in selecting the threshold is to achieve both low false positives and low false negatives. Our experimental results in Section 6 show that we can achieve this objective over our test data sets. However, it is possible that attackers evade the detection by using specially crafted code, if they know our threshold. For example, Fig. 7a shows that a decryption routine has only seven useful instructions, which is less than our threshold (15 to 17 in our experiments) of Scheme 2.

5.3 Scheme 3

We propose Scheme 3 for detecting the aforementioned specially crafted code. Scheme 3 also exploits *code abstraction* to prune useless instructions in an instruction sequence.

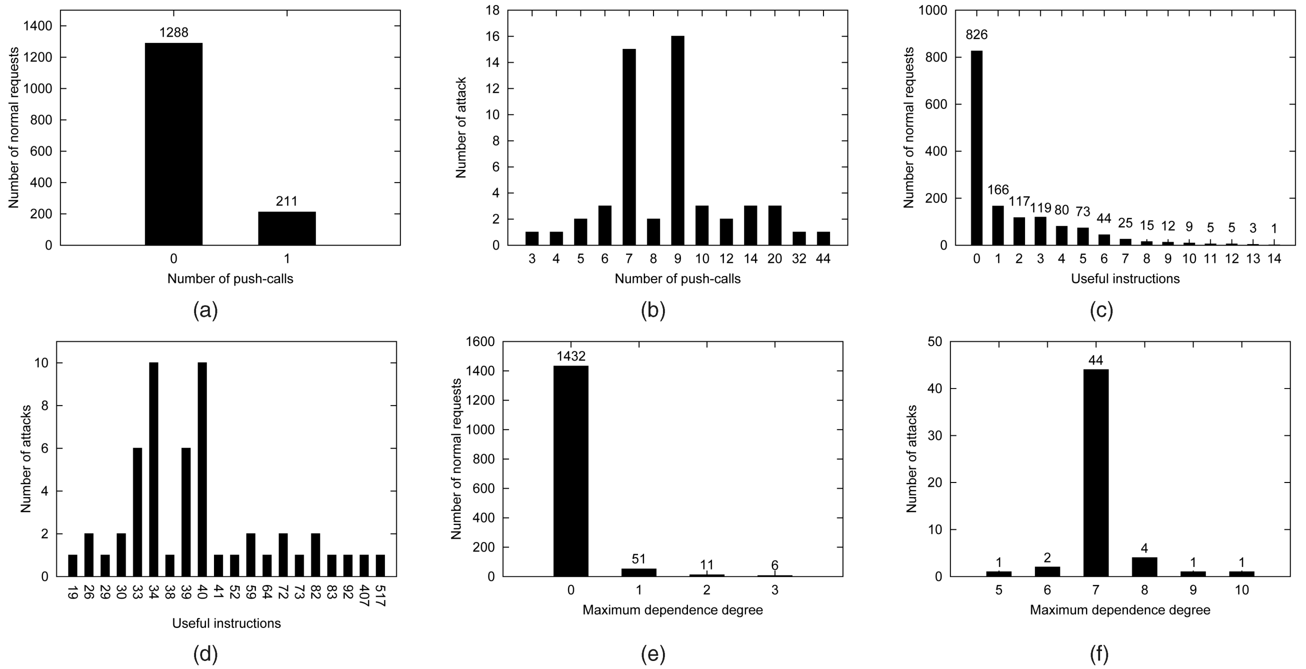

Fig. 8. (a) The number of push-calls in normal requests. (b) The number of push-calls in attack requests. (c) The number of useful instructions in normal requests. (d) The number of useful instructions in attack requests. (e) Maximum dependence degree in normal requests. (f) Maximum dependence degree in attack requests in a request.

Unlike Scheme 2, which compares the number of useful instructions with a threshold, Scheme 3 first calculates the *dependence degree* of every instruction in the instruction sequence. If the dependence degree of any useful instructions in an instruction sequence exceeds a threshold, we conclude that the instruction sequence is a segment of a program.

Dependency is a binary relation over instructions in an instruction sequence. We say instruction j *depends* on instruction i if instruction i produces a result directly or indirectly used by instruction j. Dependency relation is transitive, that is, if i depends on j and j depends on k, then i depends on k. For example, instruction 2 directly depends on instruction 0 in Fig. 7a and instruction 7 directly depends on instruction 2, and by transitive property instruction 7 depends on instruction 0. We call the number of instructions, which an instruction depends on, the *dependence degree* of the instruction.

To calculate the dependence degree of an instruction, we construct a *def-use graph*. A *def-use graph* is a directed graph $G = (V, E)$ where each node $v \in V$ corresponds to an instruction and each edge $e = (v_i, v_j) \in E$ indicates that instruction v_j produces a result directly used by instruction v_i. Obviously, the number of instructions that an instruction can reach through any path in the def-use graph is the dependence degree of the instruction. For example, instruction a can reach all other instructions through paths (a, e, 7, 2, 0) and (a, 9, 3) in the def-use graph of Fig. 7b; therefore, the dependence degree of instruction a is 6.

We observe that even though the attackers may use only a few useful instructions, at least one useful instruction depends on a certain number of other useful instructions in a program, whereas all useful instructions in a random instruction depend on few other useful instructions. Therefore, if the number of useful instructions a useful instruction depends on exceeds a threshold in an instruction sequence, we conclude that there are real codes embedded in the request. We will show that Scheme 3 provides a higher detection rate than Scheme 2.

6 Experiments

In this section, we first tune the parameter (detection threshold) for each SigFree scheme based on some training data, then evaluate and compare the performance of these schemes in checking messages collected from various sources. Performance overhead of SigFree is also evaluated when deploying it to protect a web server.

6.1 Parameter Tuning

All three schemes use a threshold value to determine if a request contains code or not. Clearly, it is critical to set the threshold values appropriately so as to minimize both detection false positive rate and false negative rate. To find out the appropriate thresholds, we tested these schemes against 50 unencrypted attack requests generated by Metasploit framework, worm Slammer, CodeRed (CodeRed.a) and a CodeRed variation (CodeRed.c), and 1,500 binary HTTP replies (52 encrypted data, 23 audio, 195 jpeg, 32 png, 1,153 gif, and 45 flash) intercepted on our department network. Here, we choose HTTP replies rather than requests as normal data for parameter tuning, because HTTP replies contain more binaries (our test over real traces of web requests is reported in Section 6.3). This allows us to obtain a better threshold, which can be used to protect not only web servers but also other Internet services. Note that although worm Slammer attacks Microsoft SQL servers instead of web servers, it also exploits buffer overflow vulnerabilities.

Threshold of push-calls for Scheme 1. Fig. 8a shows that all instruction sequences distilled from a normal

request contain at most one push-call code pattern. Fig. 8b shows that for all the 53 buffer overflow attacks we tested, every attack request contains more than two push-calls in one of its instruction sequences. Therefore, by setting the threshold number of push-calls to 2, Scheme 1 can detect all the attacks used in our experiment.

Threshold of useful instructions for Scheme 2. Fig. 8c shows that no normal requests contain an instruction sequence that has more than 14 useful instructions. Fig. 8d shows that an attack request contains over 18 useful instructions in one of its instruction sequences. Therefore, by setting the threshold to a number between 15 and 17, Scheme 2 can detect all the attacks used in our test. The three attacks, which have the largest numbers of instructions (92, 407, and 517), are worm Slammer, CodeRed.a, and CodeRed.c, respectively. This motivates us to investigate in our future work whether an exceptional large number of useful instructions indicate the occurrence of a worm.

Threshold of dependence degree for Scheme 3. Fig. 8e shows that no normal requests contain an instruction sequence whose dependence degree is more than 3. Fig. 8f shows that for each attack there exists an instruction sequence whose dependence degree is more than 4. Therefore, by setting the threshold to 4 or 5, Scheme 3 can detect all the attacks used in our experiment.

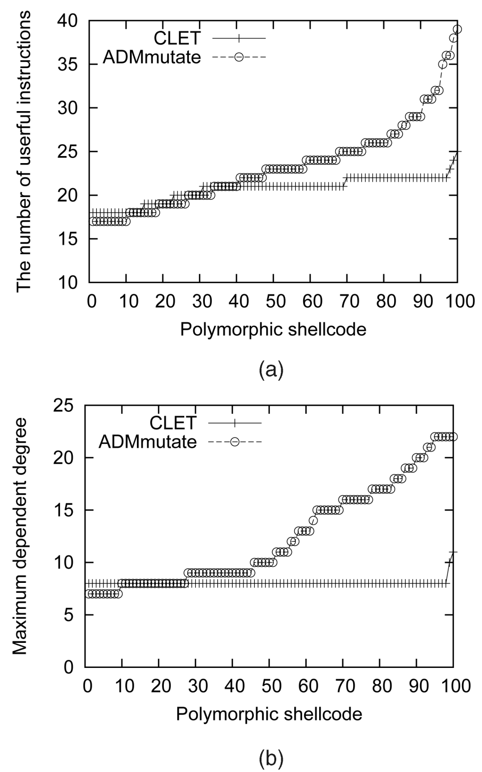

Fig. 9. The number of useful instructions and maximum dependence degree in all 200 polymorphic shellcodes. (a) The number of useful instructions. (b) Maximum dependence degree.

6.2 Detection of Polymorphic Shellcode

We also tested polymorphic attack messages from five publicly available polymorphic engines: Countdown [15], JumpCallAdditive [15], JempiScodes [16], CLET v1.0 [18], and ADMmutate v0.84 [17]. Polymorphic engines encrypt original exploit code and decrypt them during execution. Countdown uses a decrementing byte as the key for decryption; JumpCallAdditive is an XOR decoder; JempiScodes contains three different decryption methods. CLET and ADMmutate are advanced polymorphic engines, which also obfuscate the decryption routine by metamorphism such as instruction replacement and garbage insertion. CLET also uses spectrum analysis to defeat data mining methods.

Because there is no push-call pattern in the code, Scheme 1 cannot detect this type of attacks. However, Scheme 2 and Scheme 3 are still very robust to these obfuscation techniques. This is because although the original shellcode contains more useful instructions than the decryption routine does and it is also encrypted, Scheme 2 may still find enough number of useful instructions and Scheme 3 may still find enough maximum dependence degree in the decryption routines.

We generated 100 different attack messages per each of Countdown, JumpCallAdditive, ADMmutate, and CLET. For JempiScodes, we generated 300 different attack messages, 100 per each of its three decryption approaches.

Detection by Scheme 2. We used Scheme 2 to detect the useful instructions in the above 700 attack messages. Fig. 9a shows the (sorted) numbers of useful instructions in 200 polymorphic shellcodes from ADMmutate and CLET. We observed that the least number of useful instructions in these ADMmutate polymorphic shellcodes is 17, whereas the maximum number is 39; the least number of useful instructions in the CLET polymorphic shellcodes is 18, whereas the maximum number is 25. Therefore, using the same threshold value as before (i.e., between 15 and 17), we can detect all the 200 polymorphic shellcodes generated by ADMmutate and CLET. However, for each of the 100 Countdown polymorphic shellcode attack instances, Scheme 2 only reports seven useful instructions, and for each of the remaining polymorphic attack instances, it outputs 11 useful instructions Therefore, if we still use the same threshold (i.e., between 15 and 17), we will not be able to detect these attacks using Scheme 2.

Detection by Scheme 3. We also used Scheme 3 to calculate the dependence degree in the above 700 attack messages. Fig. 9b shows the (sorted) maximum dependence degree in 200 polymorphic shellcodes from ADMmutate and CLET. We observed that the least number of dependence degrees in these ADMmutate polymorphic shellcodes is 7, whereas the maximum number is 22; the least number of dependence degree in the CLET polymorphic shellcodes is 8, whereas the maximum number is 11. The dependence degree of 100 Countdown polymorphic shellcodes is 6. The dependence degree of 100 JumpCallAdditive polymorphic shellcodes is 7; the dependence degree of 100 JempiScodes polymorphic shellcodes is 5. Therefore, by using the same threshold value as before (i.e., 4 or 5), we can detect all the 700 polymorphic shellcodes.

6.3 Testing on Real Traces

We next apply SigFree over real traces for false positives.

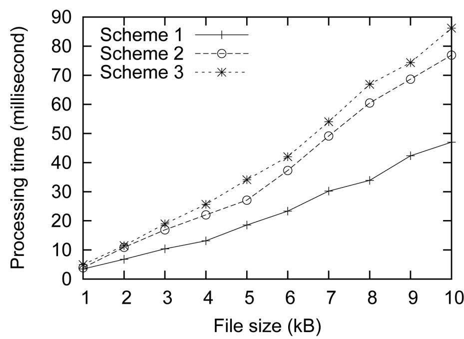

Fig. 10. Processing time of SigFree over image files of various sizes.

HTTP requests in the local network. Due to privacy concerns, we were unable to deploy SigFree in a public web server to examine real-time web requests. To make our test as realistic as possible, we deployed a client-side proxy underneath a web browser. The proxy recorded a normal user's HTTP requests during his/her daily Internet surfing. During a one-week period, more than 10 of our laboratory members installed the proxy and helped collect totally 18,569 HTTP requests. The requests include manually typed URLs, clicks through various web sites, searchings from search engines such as Google and Yahoo, secure logins to email servers and bank servers, and HTTPs requests. In this way, we believe our data set is diverse enough, not worse than that we might have got if we install SigFree in a single web server that provides only limited Internet services.

Our test based on the above real traces did not yield an alarm. This output is of no surprise because our normal web requests do not contain code.

HTTP replies in the local network. To test SigFree over more diverse data types, we also collected a total of 17,904 HTTP replies during the above one-week period. The data types of the replies include plaintext, octet-stream, pdf, javascript, xml, shockwave, jpeg, gif, png, x-icon, audio, and video. We use each of the three schemes to test over the above replies. Scheme 1 raised warnings on five HTTP replies; Scheme 2 raised warnings on four HTTP replies; Scheme 3 raised warnings on 10 HTTP replies. By manually checking these warnings, we find they are indeed false positives. The result shows that the three schemes have a few false positives. It also shows although Scheme 3 is more robust to obfuscation than Schemes 1 and 2, it has relatively higher false positives.

CiteSeer requests. Finally, we tested SigFree over one-month (September 2005) 397,895 web requests collected by the scientific and academic search engine Citeseer [51]. Our test based on the Citeseer requests did not yield an alarm.

6.4 Performance Evaluation

Stand-alone SigFree. We implemented a stand-alone SigFree prototype using the C programming language in the Win32 environment. The stand-alone prototype was compiled with Borland C++ version 5.5.1 at optimization level O2. The experiments were performed in a Windows 2003 server with Intel Pentium 4, 3.2-GHz CPU, and 1-Gbyte memory. We measured the processing time of the stand-alone prototype over all (2,910 totally) 0-10 Kbyte images collected from the above real traces. We set the upper limit to 10 Kbytes because the size of a normal web request is rarely over that if it accepts binary inputs. The types of the images include jpeg, gif, png, and x-icon. Fig. 10 shows that the average processing time of the three schemes increases linearly when the sizes of the image files increase. It also shows that Scheme 1 is the fastest among the three schemes and Scheme 3 is a little bit slower than Scheme 2. In all three schemes, the processing time over a binary file of 10 Kbytes is no more than 85 ms.

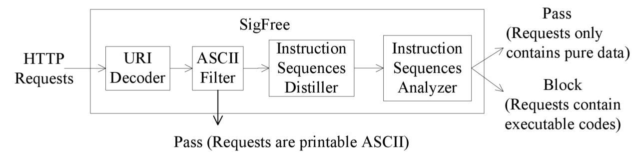

Fig. 11. The architecture of SigFree.

Proxy-based SigFree. To evaluate the performance impact of SigFree to web servers, we also implemented a proxy-based SigFree prototype. Fig. 11 depicts the implementation architecture. It is comprised of the following modules.

URI decoder. The specification for URLs [52] limits the allowed characters in a Request-URI to only a subset of the ASCII character set. This means that the query parameters of a request-URI beyond this subset should be encoded [52]. Because a malicious payload may be embedded in the request-URI as a request parameter, the first step of SigFree is to decode the request-URI.

ASCII filter. Malicious executable codes are normally binary strings. In order to guarantee the throughput and response time of the protected web system, if a request is printable ASCII ranging from 20 to 7E in hex, SigFree allows the request to pass. Note that ASCII filter does not prevent the service from receiving non-ASCII strings. All non-ASCII strings will analyzed by ISD and ISA. In Section 7.2, we will discuss a special type of executable codes called alphanumeric shellcodes [53] that actually use printable ASCII.

The proxy-based prototype was also compiled with Borland C++ version 5.5.1 at optimization level O2. The proxy-based prototype implementation was hosted in the Windows 2003 server with Intel Pentium 4, 3.2-GHz CPU, and 1-Gbyte memory.

The proxy-based SigFree prototype accepts and analyzes all incoming requests from clients. The client testing traffics were generated by Jef Poskanzer's http_load program [54] from a Linux desktop PC with Intel Pentium 4 and 2.5-GHz CPU connected to the Windows server via a 100-Mbps LAN switch. We modified the original http_load program so that clients can send code-injected data requests.

For the requests that SigFree identifies as normal, SigFree forwards them to the web server, Apache HTTP Server 2.0.54, hosted in a Linux server with dual Intel Xeon 1.8-Gbyte CPUs. Clients send requests from a predefined URL list. The documents referred in the URL list are stored in the web server. In addition, the prototype implementation uses a time-to-live-based cache to reduce redundant HTTP connections and data transfers.

Rather than testing the absolute performance overhead of SigFree, we consider it more meaningful measuring the impact of SigFree on the normal web services. Hence, we

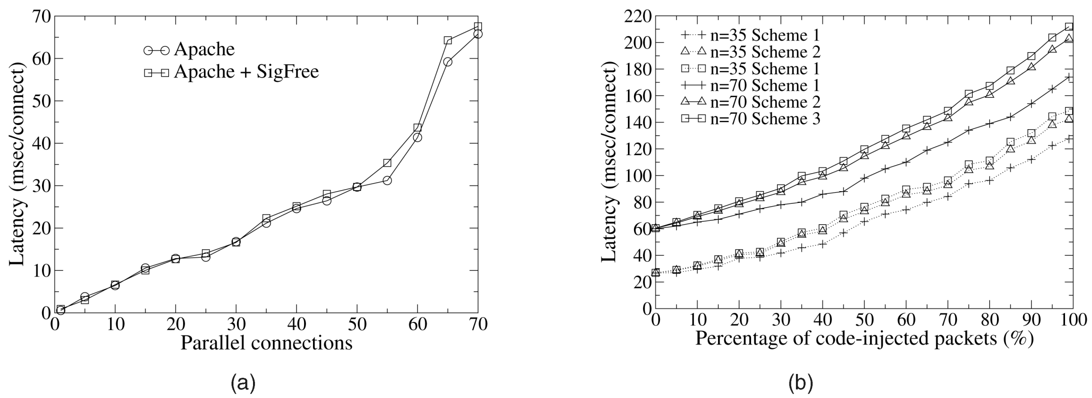

Fig. 12. Performance impact of SigFree on Apache HTTP server.

measured the *average response latency* (which is also an indication of *throughput* although we did not directly measure throughput) of the connections by running http_load for 1,000 fetches. Fig. 12a shows that when there are no buffer overflow attacks, the average response time in the system with SigFree is only slightly higher than the system without SigFree. This indicates that, despite the connection and ASCII checking overheads, the proxy-based implementation does not affect the overall latency significantly.

Fig. 12b shows the average latency of connections as a function of the percentage of attacking traffic. We used CodeRed as the attacking data. Only successful connections were used to calculate the average latency; that is, the latencies of attacking connections were not counted. This is because what we care is the impact of attack requests on normal requests. We observe that the average latency increases slightly worse than linear when the percentage of malicious attacks increases. Generally, Scheme 1 is about 20 percent faster than Scheme 2 and Scheme 3 is slightly slower than Scheme 2.

Overall, our experimental results from the prototype implementation show that SigFree has reasonably low performance overhead to web servers. Especially when the fraction of attack messages is small (say < 10 percent), the additional latency caused by SigFree to web servers is almost negligible.

7 Discussion

7.1 Robustness to Obfuscation

Most malware detection schemes include a two-stage analysis. The first stage is disassembling binary code, and the second stage is analyzing the disassembly results. There are obfuscation techniques to attack each stage [46], [55] and attackers may use them to evade detection. Table 1 shows that SigFree is robust to most of these obfuscation techniques.

Obfuscation in the first stage. *Junk byte insertion* is one of the simplest obfuscation against disassembly. Here, junk bytes are inserted at locations that are not reachable at runtime. This insertion however can mislead a linear sweep algorithm but cannot mislead a recursive traversal algorithm [43], on which our algorithm bases.

Opaque predicates are used to transform unconditional jumps into conditional branches. Opaque predicates are predicates that are always evaluated to either true or false regardless of the inputs. This allows an obfuscator to insert junk bytes either at the jump target or in the place of the fall-through instruction. We note that opaque predicates may make SigFree mistakenly interpret junk byte as executable codes. However, this mistake will not cause SigFree to miss any real malicious instructions. Therefore, SigFree is also immune to obfuscation based on opaque predicates.

Obfuscation in the second stage. Second-stage obfuscation techniques can easily obfuscate the rule-based scheme by Chinchani and Berg [12]. For example, Rule 1 expects that push instructions appear before the branch. This rule can be confused by obfuscation technique instruction replacement (e.g., "push eax" replaced by "sub esp 4; move [esp] eax"). Most of the second-stage obfuscation techniques obfuscate the behaviors of a program; however, the obfuscated programs still bear characteristics of programs. Since the purpose of SigFree is to differentiate executable codes and random binaries rather than benign and malicious executable codes, most of these obfuscation techniques are ineffective to SigFree. Obfuscation techniques such as instruction reordering, register renaming, garbage insertion, and reordered memory accesses do not affect the number of calls or useful instructions that our schemes are based on. By exploiting instruction replacement and equivalent functionality, attacks may evade the detection of Scheme 1 but cannot evade the detection of Scheme 2 and Scheme 3.

TABLE 1
SigFree Is Robust to Most Obfuscation

Disassembly stage	Obfuscation	SigFree	
	Junk byte insertion	Yes	
	Opaque predict	Yes	
	Branch function	partial	
Analysis stage	Obfuscation	Scheme 1	Schem 2&3
	Instruction reordering	Yes	Yes
	Register renaming	Yes	Yes
	Garbage insertion	Yes	Yes
	Instruction replacement	No	Yes
	Equivalent funcationality	No	Yes
	Reordered memory accesses	Yes	Yes

7.2 Limitations

SigFree also has several limitations. First, SigFree cannot fully handle the branch-function-based obfuscation, as indicated in Table 1. Branch function is a function $f(x)$ that, whenever called from x, causes control to be transferred to the corresponding location $f(x)$. By replacing unconditional branches in a program with calls to the branch function, attackers can obscure the flow of control in the program. We note that there are no general solutions for handling branch function at the present state of the art.

With respect to SigFree, due to the obscurity of the flow of control, branch function may cause SigFree to break the executable codes into multiple instruction sequences. Nevertheless, it is still possible for SigFree to find this type of buffer overflow attacks as long as SigFree can still find enough number of useful instructions or dependence degree in one of the distilled instruction sequences.

Second, SigFree cannot fully handle self-modifying code. Self-modifying code is a piece of code that dynamically modifies itself at runtime and could make SigFree mistakenly exclude all its instruction sequences. It is crafted in a way such that static analysis will reach illegal instructions in all its instruction sequences, but these instructions become legal during execution. To address this attack, we may remove Step 3 in excluding instruction sequences; that is, we do not use inevitably reachable illegal instructions as a criterion for pruning instruction sequences. This however will increase the computational overhead as more instruction sequences will be analyzed. Self-modifying code can also reduce the number of *useful* instructions, because some instructions are considered *useless*. We note that there are also no general static solutions for handling self-modifying code at the present state of the art. Nevertheless, it is still possible for SigFree to detect self-modifying code, because self-modifying code itself is a piece of code that may have enough number of useful instructions or dependence degree. Our future work will explore this area.

Third, the executable shellcodes could be written in alphanumeric form [53]. Such shellcodes will be treated as printable ASCII data and thus bypass our analyzer. By turning off the ASCII filter, Scheme 2 and Scheme 3 can successfully detect alphanumeric shellcodes; however, it will increase computational overhead. It therefore requires a slight tradeoff between tight security and system performance.

Fourth, SigFree does not detect attacks such as return-to-libc attacks that just corrupt control flow or data without injecting code. However, these attacks can be handled by some simple methods. For example, return-to-libc attacks can be defeated by mapping (through mmap()) the addresses of shared libraries so that the addresses contain null bytes[2] [56].

Finally, it is still possible that attackers evade the detection of Scheme 3 by using specially crafted code, once they know the threshold of Scheme 3. However, we believe it is fairly hard, as the bar has been raised. For example, it is difficult, if not impossible, to reduce the dependence degree to 3 or fewer in all the instructions of Countdown decryption routine shown in Fig. 7a. Decryption instruction a depends on all other instructions in Fig. 7b. These instruction can be classified into three groups: instructions 0, 2, and 7 are used to initialize loop counter register ecx; instruction e is used as a loop instruction; instructions 3 and 9 are used to get program counter (PC). To reduce the dependency degree of instruction a to 3, attackers have to use one instruction to initialize the loop counter and one instruction to get PC. However, attackers cannot use one instruction to initialize loop counter (e.g., mov ecx, 0x0000005B), which contains null bytes. It is also hard to find a single instruction to get PC because IA-32 architecture does not provide any instruction to directly read PC.

2. Null bytes, which are C string terminators, cause the termination of attacks before they overflow the entire buffer.

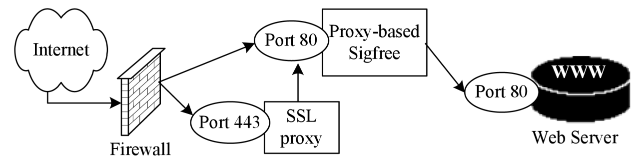

Fig. 13. SigFree with an SSL proxy.

7.3 Application-Specific Encryption Handling

The proxy-based SigFree could not handle encrypted or encoded data directly. A particular example is SSL-enabled web server. Enhancing security between web clients and web servers by encrypting HTTP messages, SSL also causes the difficulty for out-of-box malicious code detectors.

To support SSL functionality, an SSL proxy such as Stunnel [57] (Fig. 13) may be deployed to securely tunnel the traffic between clients and web servers. In this case, we may simply install SigFree in the machine where the SSL proxy is located. It handles the web requests in cleartext that have been decrypted by the SSL proxy. On the other hand, in some web server applications, SSL is implemented as a server module (e.g., mod_ssl in Apache). In this case, SigFree will need to be implemented as a server module (though not shown in Fig. 13), located between the SSL module and the WWW server. We notice that most popular web servers allow us to write a server module to process requests and specify the order of server modules. Detailed study will be reported in our future work.

7.4 Applicability

So far, we only discussed using SigFree to protect web servers. It is worth mentioning that our tool is also widely applicable to many programs that are vulnerable to buffer overflow attacks. For example, the proxy-based SigFree can be used to protect all internet services that do not permit executable binaries to be carried in requests. SigFree should not directly be used to protect some Internet services that do accept binary code such as FTP servers; otherwise, SigFree will generate many false positives. To apply SigFree for protecting these Internet services, other mechanisms such as whitelisting need to be used.

In addition to protecting servers, SigFree can also provide file system real-time protection. Buffer overflow vulnerabilities have been found in some famous applications such as Adobe Acrobat and Adobe Reader [58], Microsoft JPEG Processing (GDI+) [59], and WinAmp [60]. This means that attackers may embed their malicious code in PDF, JPEG, or

MP3-list files to launch buffer overflow attacks. In fact, a virus called Hesive [61] was disguised as a Microsoft Access file to exploit buffer overflow vulnerability of Microsoft's Jet Database Engine. Once opened in Access, infected .mdb files take advantage of the buffer overflow vulnerability to seize control of vulnerable machines. If mass-mailing worms exploit these kinds of vulnerabilities, they will become more fraudulent than before, because they may appear as pure data-file attachments. SigFree can be used to alleviate these problems by checking those files and email attachments that should not include any code. If the buffer being overflowed is inside a JPEG or GIF system, ASN.1 or base64 encoder, SigFree cannot be directly applied. Although SigFree can decode the protected file according to the protocols or applications it protects, more details need to be studied in the future.

Although SigFree is implemented in the Intel IA-32 architecture, it is a generic technique, which can be ported to other platforms such as PowerPC. The mechanism of code abstraction technique and its robustness to obfuscation are not related to any hardware platform. Therefore, we believe that detection capabilities and resilience to obfuscation will be preserved after porting. Of course, some implementation details such as handling special cases in Scheme 2 need to be changed. We will study this portability issue in our future work.

Finally, as a generic technique, SigFree can also block other types of attacks as long as the attacks perform binary code injection. For example, it can block code-injection format string attacks.

8 Conclusion

We have proposed SigFree, an online signature-free out-of-the-box blocker that can filter code-injection buffer overflow attack messages, one of the most serious cyber security threats. SigFree does not require any signatures, thus it can block new unknown attacks. SigFree is immunized from most attack-side code obfuscation methods and good for economical Internet-wide deployment with little maintenance cost and low performance overhead.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions. The authors would also like to thank Yoon-Chan Jhi for helpful suggestions and the members of Penn State Cyber Security Laboratory for collecting real traces. The works of Xinran Wang and Sencun Zhu were supported in part by the Army Research Office (W911NF-05-1-0270) and the National Science Foundation (CNS-0524156). The works of Chi-Chun Pan and Peng Liu were supported in part by US NSF grant number CT-3352241.

References

[1] B.A. Kuperman, C.E. Brodley, H. Ozdoganoglu, T.N. Vijaykumar, and A. Jalote, "Detecting and Prevention of Stack Buffer Overflow Attacks," *Comm. ACM*, vol. 48, no. 11, 2005.
[2] J. Pincus and B. Baker, "Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns," *IEEE Security and Privacy*, vol. 2, no. 4, 2004.
[3] G. Kc, A. Keromytis, and V. Prevelakis, "Countering Code-Injection Attacks with Instruction-Set Randomization," *Proc. 10th ACM Conf. Computer and Comm. Security (CCS '03)*, Oct. 2003.
[4] E. Barrantes, D. Ackley, T. Palmer, D. Stefanovic, and D. Zovi, "Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks," *Proc. 10th ACM Conf. Computer and Comm. Security (CCS '03)*, Oct. 2003.
[5] J. Newsome and D. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software," *Proc. 12th Ann. Network and Distributed System Security Symp. (NDSS)*, 2005.
[6] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham, "Vigilante: End-to-End Containment of Internet Worms," *Proc. 20th ACM Symp. Operating Systems Principles (SOSP)*, 2005.
[7] Z. Liang and R. Sekar, "Fast and Automated Generation of Attack Signatures: A Basis for Building Self-Protecting Servers," *Proc. 12th ACM Conf. Computer and Comm. Security (CCS)*, 2005.
[8] J. Xu, P. Ning, C. Kil, Y. Zhai, and C. Bookholt, "Automatic Diagnosis and Response to Memory Corruption Vulnerabilities," *Proc. 12th ACM Conf. Computer and Comm. Security (CCS)*, 2005.
[9] S. Singh, C. Estan, G. Varghese, and S. Savage, "The Earlybird System for Real-Time Detection of Unknown Worms," technical report, Univ. of California, San Diego, 2003.
[10] H.-A. Kim and B. Karp, "Autograph: Toward Automated, Distributed Worm Signature Detection," *Proc. 13th USENIX Security Symp. (Security)*, 2004.
[11] J. Newsome, B. Karp, and D. Song, "Polygraph: Automatic Signature Generation for Polymorphic Worms," *Proc. IEEE Symp. Security and Privacy (S&P)*, 2005.
[12] R. Chinchani and E.V.D. Berg, "A Fast Static Analysis Approach to Detect Exploit Code inside Network Flows," *Proc. Eighth Int'l Symp. Recent Advances in Intrusion Detection (RAID)*, 2005.
[13] T. Toth and C. Kruegel, "Accurate Buffer Overflow Detection via Abstract Payload Execution," *Proc. Fifth Int'l Symp. Recent Advances in Intrusion Detection (RAID)*, 2002.
[14] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Polymorphic Worm Detection Using Structural Information of Executables," *Proc. Eighth Int'l Symp. Recent Advances in Intrusion Detection (RAID)*, 2005.
[15] *The Metasploit Project*, http://www.metasploit.com, 2007.
[16] *Jempiscodes—A Polymorphic Shellcode Generator*, http://www.shellcode.com.ar/en/proyectos.html, 2007.
[17] S. Macaulay, *Admmutate: Polymorphic Shellcode Engine*, http://www.ktwo.ca/security.html, 2007.
[18] T. Detristan, T. Ulenspiegel, Y. Malcom, and M.S.V. Underduk, *Polymorphic Shellcode Engine Using Spectrum Analysis*, http://www.phrack.org/show.php?p=61&a=9, 2007.
[19] D. Wagner, J.S. Foster, E.A. Brewer, and A. Aiken, "A First Step towards Automated Detection of Buffer Overrun Vulnerabilities," *Proc. Seventh Ann. Network and Distributed System Security Symp. (NDSS '00)*, Feb. 2000.
[20] D. Evans and D. Larochelle, "Improving Security Using Extensible Lightweight Static Analysis," *IEEE Software*, vol. 19, no. 1, 2002.
[21] H. Chen, D. Dean, and D. Wagner, "Model Checking One Million Lines of C Code," *Proc. 11th Ann. Network and Distributed System Security Symp. (NDSS)*, 2004.
[22] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "Stackguard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," *Proc. Seventh USENIX Security Symp. (Security '98)*, Jan. 1998.
[23] *GCC Extension for Protecting Applications from Stack-Smashing Attacks*, http://www.research.ibm.com/trl/projects/security/ssp, 2007.
[24] T. cker Chiueh and F.-H. Hsu, "Rad: A Compile-Time Solution to Buffer Overflow Attacks," *Proc. 21st Int'l Conf. Distributed Computing Systems (ICDCS)*, 2001.
[25] A. Smirnov and T. cker Chiueh, "Dira: Automatic Detection, Identification, and Repair of Control-Hijacking Attacks," *Proc. 12th Ann. Network and Distributed System Security Symp. (NDSS)*, 2005.
[26] *Pax Documentation*, http://pax.grsecurity.net/docs/pax.txt, Nov. 2003.
[27] A. Baratloo, N. Singh, and T. Tsai, "Transparent Run-Time Defense against Stack Smashing Attacks," *Proc. USENIX Ann. Technical Conf. (USENIX '00)*, June 2000.

[28] G.S. Kc and A.D. Keromytis, "E-NEXSH: Achieving an Effectively Non-Executable Stack and Heap via System-Call Policing," *Proc. 21st Ann. Computer Security Applications Conf. (ACSAC)*, 2005.

[29] J. McGregor, D. Karig, Z. Shi, and R. Lee, "A Processor Architecture Defense against Buffer Overflow Attacks," *Proc. Int'l Conf. Information Technology: Research and Education (ITRE '03)*, pp. 243-250, 2003.

[30] S. Bhatkar, R. Sekar, and D.C. DuVarney, "Efficient Techniques for Comprehensive Protection from Memory Error Exploits," *Proc. 14th USENIX Security Symp. (Security)*, 2005.

[31] V. Kiriansky, D. Bruening, and S. Amarasinghe, "Secure Execution via Program Shepherding," *Proc. 11th USENIX Security Symp. (Security)*, 2002.

[32] Z. Liang and R. Sekar, "Automatic Generation of Buffer Overflow Attack Signatures: An Approach Based on Program Behavior Models," *Proc. 21st Ann. Computer Security Applications Conf. (ACSAC)*, 2005.

[33] R. Pang, V. Yegneswaran, P. Barford, V. Paxson, and L. Peterson, "Characteristics of Internet Background Radiation," *Proc. ACM Internet Measurement Conf. (IMC)*, 2004.

[34] Z. Li, M. Sanghi, Y. Chen, M.Y. Kao, and B. Chavez, "Hamsa: Fast Signature Generation for Zero-Day Polymorphic Worms with Provable Attack Resilience," *Proc. IEEE Symp. Security and Privacy (S&P '06)*, May 2006.

[35] X.F. Wang, Z. Li, J. Xu, M.K. Reiter, C. Kil, and J.Y. Choi, "Packet Vaccine: Black-Box Exploit Detection and Signature Generation," *Proc. 13th ACM Conf. Computer and Comm. Security (CCS)*, 2006.

[36] H.J. Wang, C. Guo, D.R. Simon, and A. Zugenmaier, "Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits," *Proc. ACM SIGCOMM '04*, Aug. 2004.

[37] K. Wang and S.J. Stolfo, "Anomalous Payload-Based Network Intrusion Detection," *Proc. Seventh Int'l Symp. Recent Advances in Intrusion Detection (RAID)*, 2004.

[38] K. Wang, G. Cretu, and S.J. Stolfo, "Anomalous Payload-Based Worm Detection and Signature Generation," *Proc. Eighth Int'l Symp. Recent Advances in Intrusion Detection (RAID)*, 2005.

[39] O. Kolesnikov, D. Dagon, and W. Lee, "Advanced Polymorphic Worms: Evading IDS by Blending in with Normal Traffic," Technical Report GIT-CC-04-13, College of Computing, Georgia Tech, 2004.

[40] M. Christodorescu and S. Jha, "Static Analysis of Executables to Detect Malicious Patterns," *Proc. 12th USENIX Security Symp. (Security '03)*, Aug. 2003.

[41] M. Christodorescu, S. Jha, S.A. Seshia, D. Song, and R.E. Bryant, "Semantics-Aware Malware Detection," *Proc. IEEE Symp. Security and Privacy (S&P)*, 2005.

[42] A. Lakhotia and U. Eric, "Abstract Stack Graph to Detect Obfuscated Calls in Binaries," *Proc. Fourth IEEE Int'l Workshop Source Code Analysis and Manipulation (SCAM '04)*, Sept. 2004.

[43] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static Disassembly of Obfuscated Binaries," *Proc. 13th USENIX Security Symp. (Security)*, 2004.

[44] *Fnord Snort Preprocessor*, http://www.cansecwest.com/spp_fnord.c, 2007.

[45] B. Schwarz, S.K. Debray, and G.R. Andrews, "Disassembly of Executable Code Revisited," *Proc. Ninth IEEE Working Conf. Reverse Eng. (WCRE)*, 2002.

[46] C. Linn and S. Debray, "Obfuscation of Executable Code to Improve Resistance to Static Disassembly," *Proc. 10th ACM Conf. Computer and Comm. Security (CCS '03)*, Oct. 2003.

[47] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms.* MIT Press/McGraw-Hill, 1990.

[48] L.D. Fosdick and L. Osterweil, "Data Flow Analysis in Software Reliability," *ACM Computing Surveys*, vol. 8, Sept. 1976.

[49] J. Huang, "Detection of Data Flow Anomaly through Program Instrumentation," *IEEE Trans. Software Eng.*, vol. 5, no. 3, May 1979.

[50] *Intel IA-32 Architecture Software Developer's Manual Volume 1: Basic Architecture.* Intel, http://developer.intel.com/design/pentium4/manuals/253665.htm, 2007.

[51] *Citeseer: Scientific Literature Digital Library*, http://citeseer.ist.psu.edu, 2007.

[52] T. Berners-Lee, L. Masinter, and M. McCahill, *Uniform Resource Locators (URL)*, RFC 1738 (Proposed Standard), updated by RFCs 1808, 2368, 2396, 3986, http://www.ietf.org/rfc/rfc1738.txt, 2007.

[53] *Writing IA32 Alphanumeric Shellcodes*, rix, http://www.phrack.org/show.php?p=57&a=15, 2001.

[54] *Http Load: Multiprocessing Http Test Client*, http://www.acme.com/software/http_load, 2007.

[55] C. Collberg, C. Thomborson, and D. Low, "A Taxonomy of Obfuscating Transformations," Technical Report 148, Dept. Computer Science, Univ. of Auckland, 1997.

[56] S. Designer, *Getting around Non-Executable Stack (and Fix)*, http://seclists.org/bugtraq/1997/Aug/0063.html, 1997.

[57] *Stunnel—Universal SSL Wrapper*, http://www.stunnel.org, 2007.

[58] *Security Advisory: Acrobat and Adobe Reader Plug-In Buffer Overflow*, http://www.adobe.com/support/techdocs/321644.html, 2007.

[59] *Buffer Overrun in JPEG Processing (GDI+) Could Allow Code Execution (833987)*, http://www.microsoft.com/technet/security/bulletin/MS04-028.mspx, 2007.

[60] *Winamp3 Buffer Overflow*, http://www.securityspace.com/smysecure/catid.html?id=11530.

[61] *Symantec Security Response—Backdoor.Hesive*, http://securityresponse.symantec.com/avcenter/venc/data/backdoor.hesive.html, 2007.

Xinran Wang received the BE and MS degrees in electrical engineering from Tsinghua University, Beijing, in 1998 and 2001, respectively. He is currently a PhD candidate in the Department of Computer Science and Engineering, Pennsylvania State University (PSU). Prior to joining PSU, he was a software engineer in the China Software Development Laboratory, IBM from 2003 to 2004 and Asiainfo from 2001 to 2003. His current research interests include network and systems security with a focus on sensor network security and malware defenses.

Chi-Chun Pan received the BS and MS degrees in computer and information sciences from the National Chiao Tung University, Hsinchu, Taiwan. He is currently a PhD candidate of industrial engineering in the College of Information Sciences and Technology, Pennsylvania State University. He is also a research assistant in the North-East Visualization and Analytics Center and a member of the Laboratory for Intelligent Systems and Quality. His research interests include information security, machine learning, and text mining.

Peng Liu is an associate professor of information sciences and technology in the College of Information Sciences and Technology, Pennsylvania State University. He is the research director of the Penn State Center for Information Assurance and the director of the Cyber Security Laboratory. His research interests are in all areas of computer and network security. He has published a book and more than 100 refereed technical papers. His research has been sponsored by DARPA, US NSF, AFOSR, US DOE, US DHS, ARO, NSA, CISCO, HP, Japan JSPS, and Penn State. He is a recipient of the US Department of Energy Early CAREER PI Award.

Sencun Zhu received the BS degree in precision instruments from Tsinghua University, Beijing, in 1996, the MS degree in signal processing from the University of Science and Technology of China, Graduate School at Beijing in 1999, and the PhD degree in information technology from George Mason University in 2004. He is currently with the Department of Computer Science and Engineering and College of Information Sciences and Technology, Pennsylvania State University. His research interests include network and systems security with a focus on ad-hoc and sensor network security, P2P security, and malware defenses. He was a recipient of the US NSF CAREER Award in 2007. He cochaired the Fourth ACM Workshop on Security of Ad Hoc and Sensor Networks (SASN 2006) and served in the TPC of many international conferences including ACM Conference on Computer and Communications Security (CCS), IEEE INFOCOM, and so forth. His publications can be found in http://www.cse.psu.edu/szhu.