Fast and Automated Generation of Attack Signatures: A Basis for Building Self-Protecting Servers*

Zhenkai Liang and R. Sekar
Dept. of Computer Science, Stony Brook University
Stony Brook, NY, USA
{zliang, sekar}@cs.sunysb.edu

ABSTRACT

Large-scale attacks, such as those launched by worms and zombie farms, pose a serious threat to our network-centric society. Existing approaches such as software patches are simply unable to cope with the volume and speed with which new vulnerabilities are being discovered. In this paper, we develop a new approach that can provide effective protection against a vast majority of these attacks that exploit memory errors in C/C++ programs. Our approach, called COVERS, uses a forensic analysis of a victim server's memory to correlate attacks to inputs received over the network, and *automatically* develop a signature that characterizes inputs that carry attacks. The signatures tend to capture characteristics of the underlying vulnerability (e.g., a message field being too long) rather than the characteristics of an attack, which makes them effective against variants of attacks. Our approach introduces low overheads (under 10%), does not require access to source code of the protected server, and has successfully generated signatures for the attacks studied in our experiments, without producing false positives. Since the signatures are generated in tens of milliseconds, they can potentially be distributed quickly over the Internet to filter out (and thus stop) fast-spreading worms. Another interesting aspect of our approach is that it can defeat guessing attacks reported against address-space randomization and instruction set randomization techniques. Finally, it increases the capacity of servers to withstand repeated attacks by a factor of 10 or more.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: Unauthorized access, Invasive software

General Terms

Security, Management

Keywords

Memory error, Buffer overflow, Signature generation, Denial-of-service protection, Worm defense

*This research is supported in part by an ONR grant N000140110967 and an NSF grant CCR-0208877.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CCS'05, November 7–11, 2005, Alexandria, Virginia, USA.
Copyright 2005 ACM 1-59593-226-7/05/0011 ...$5.00.

1. Introduction

The past few years have witnessed an alarming increase in *large-scale attacks:* automated attacks carried out by large numbers of hosts on the Internet. These may be the work of worms, zombies, or large numbers of hackers running attack scripts. Such attacks have the following characteristics: they originate from many different hosts, target a non-negligible fraction of vulnerable hosts on the Internet, and are repetitive. Buffer overflow attacks (or more generally, attacks that exploit memory errors in C/C++ programs) are the most popular choice in these attacks, as they provide substantial control over a victim host. For instance, virtually every worm known to date has been based on this class of attacks.

The best protection available today against such large-scale attacks is the deployment of software patches that correct the underlying software flaw targeted by the attack. The benefit of this approach is that it can filter out attacks without impacting legitimate requests. Moreover, *neither the performance nor the availability* of a patched application is degraded by attacks. Unfortunately, patches are not available for previously unknown vulnerabilities that are targeted by *zero-day attacks.* Even for known vulnerabilities, there is often a long period of time between the identification of a vulnerability and the availability of a patch; and a further delay in installation of these patches due to the need for extensive testing at each site where the software is deployed. Consequently, patches aren't adequate against large-scale, repetitive attacks.

A second line of defense against zero-day attacks is memory-error exploitation detection techniques such as StackGuard [10], address-space randomization [1, 4, 5], and complete memory-error protection [13, 14, 20, 27, 41]. Although these techniques can detect attacks before system resources (e.g., files) are compromised, they cannot protect the victim process itself, whose integrity is compromised by the time of detection. For this reason, the safest approach for recovery is to terminate the victim process. With repetitive attacks, such an approach will cause repeated server restarts, effectively rendering its service unavailable during periods of attack. Even worse, [29, 35] have shown how such repetitive attacks can defeat probabilistic protection techniques such as address-space and instruction set randomization.

In contrast to the above approaches, we present a new approach that can provide effective protection against large-scale, repetitive attacks. Our approach, called COVERS (COntext-based, VulnERability-oriented Signature), is based on *automatic generation* of attack signatures by observing ongoing attacks, and using these signatures to filter out future occurrences of these attacks. By doing so, our approach preserves not only the integrity of the service, but also its availability. The filters can deployed inside the address space of a protected server (by intercepting library calls that read network input) or as an in-line network filter. Our signatures

tend to capture the underlying vulnerability, which means that they would likely block other attacks exploiting the same vulnerability. This makes the approach effective against *polymorphic worms.*

1.1 Overview of Approach and Key Contributions

Our approach relies on the following four steps to generate attack signatures:

- ***Attack detection.*** We can use previously developed techniques for detecting buffer overflow attacks, such as StackGuard [10], address-space randomization (ASR) [1, 4, 5], and instruction-set randomization (ISR) [3, 15].
- ***Correlation to input.*** The correlation step identifies the specific network packet (or flow) involved in an attack, and the bytes within this packet that were responsible. This enables the signature search to be focused on the relevant parts of input, enabling higher-quality signature generation. Previous work [22] has suggested the use of taint analysis to track the input data that led to the attack, but their implementation incurs significant runtime overheads, often slowing down programs by a factor of 10.

  A key contribution of this paper is that of developing *efficient* techniques for correlation. Our approach is based on the observation that all known memory error exploits involve corruption of pointer values, and that this value must be included in the attack input. Based on this observation, we develop a forensic analysis of the victim process memory to identify the memory region surrounding the corrupted pointer value, and match this region with recent inputs. Since the correlation step is performed only when an attack is detected, it does not impact the performance of the server during normal operation.
- ***Identifying input context.*** This step is concerned with identifying the logical input context within which an attack appears. For instance, an attack may appear within a particular field of a specific type of message. Knowing this information, we can restrict signature matching to this field, thereby avoiding some false positives that may otherwise arise. In addition, note that it is likely that the same piece of code processes a particular message field. An attacker that wishes to exploit a vulnerability in this code needs to embed the attack in this message field, thereby making it harder to evade detection. This contrasts with approaches that use byte offsets to identify attack context, since these offsets can be easily changed by modifying the length of some (benign) field preceding the vulnerable field.

  Our approach uses a simple specification of message formats (consisting of a few lines for most services) to guide the input context identification step. The relevant information for these specifications is readily available from network protocol specifications such as the IETF RFCs.

  Development of input-context-aware signatures is another important contribution of this paper. In our experiments, this step was necessary for generating accurate signatures in five out of seven attacks. Previous approaches have largely ignored this step, compensating for this weakness either by requiring a large number of attack instances, which delays signature generation; or by including non-essential details in the attack signature, raising the odds of false negatives.
- ***Signature generation.*** This step exploits the characteristics of the underlying vulnerability, such as excessively long message fields or the presence of binary data within text-valued fields.

1.2 Benefits of Approach

We have experimentally evaluated the effectiveness and performance of our approach, as described in Section 7. Our experiments involved several popular server programs, and exercised different types of buffer overflow attacks such as stack-smashing, heap overflows, and format-string attacks. These results demonstrate the following benefits of our approach.

- *Effectiveness against attacks.* Our approach generated effective signatures to stop all the attacks used in our experiment.
- *Fast generation of signatures.* Our approach is typically able to generate signatures with just a single sample, and the overall time for signature generation is of the order of ten milliseconds.
- *Low overheads under normal operation.* Our approach introduces low overheads of under 10% during normal operation.
- *No false positives.* For all the attacks evaluated, our approach does not produce any false positives.
- *Applicable to COTS software, without access to source.* Our approach does not require any modifications to the protected server software, or access to its source code.

These features of the approach make it possible to achieve the following objectives using our approach.

- **Effective protection against denial-of-service effect of repetitive attacks.** Servers protected by our technique were able to withstand repetitive attacks at a rate that is at least 10 times higher than that of unprotected servers.
- **Protection from attacks on randomization.** [29] has developed an attack that defeats the 16-bits of randomness in PaX [1] by using an attack that successively tries out all possible 16-bit values. Our approach can defend against this attack by filtering out all but the first few attack attempts. In a similar manner, our approach can also defend against guessing attacks on instruction set randomization [35].
- **Protection from polymorphic attacks.** Our approach tends to generate signatures that characterize underlying vulnerability, e.g., excessive length of a message field, rather than specific details such as the instruction sequence contained within an attack. As a result, these signatures can stop attack variants that exploit the same vulnerability.
- **Network-wide signature deployment to defeat fast-spreading attacks.** The speed of our signature generation makes it possible to distribute and deploy these signatures in the Internet to stop fast-spreading worms. Moreover, the nature of our signatures permit the receivers of signatures to verify them before deployment — they can compare the signatures against their recent input, and verify that the signature would not cause recently received legitimate inputs to be discarded.

1.3 Organization of the Paper

The rest of the paper is organized as follows. We describe the four steps in our approach in Sections 2–5. Implementation of these steps in described in Section 6. We evaluate our approach in Section 7. Related work is discussed in Section 8, followed by concluding remarks in Section 9.

2. Attack Detection

For attack detection, we can make use of any of the existing techniques for memory-error exploit protection, such as StackGuard [10, 11, 7], address-space randomization (ASR) [1, 4, 5], instruction set randomization (ISR) [3, 15], and complete memory-error protection [13, 14, 20, 27, 41]. Of these techniques, we use ASR in this paper, since it provides broad coverage against memory error exploits, has low overheads, and unlike complete memory error protection techniques, does not pose any backward compatibility problems.

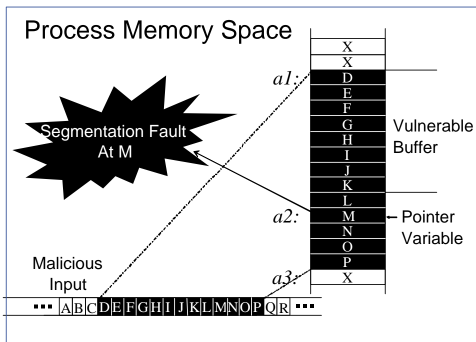

Figure 1: Buffer overflow attack scenario.

All memory error exploits reported so far have been based on pointer corruption. In particular, these exploits use a software vulnerability (such as out-of-bounds array access) to change a security-critical pointer with an attacker-provided value. Code injection attacks overwrite a function pointer value (e.g., return address) with the address of an input buffer that holds attacker-provided code. Existing-code attacks (sometimes called return-to-libc attacks) are based on overwriting code pointers with the locations of functions such as `execve`. Attacks that target security-critical data [6] are usually based on overwriting data pointer values.

ASR defeats pointer corruption attacks by randomizing the location of various data and code objects in the process address space. Thus, the attacker can no longer guess the address of the input buffer containing attacker-provided code, the address of any specific function in the process memory (such as `execve`), or any vulnerable data items (e.g., file names). The only choice for the attacker is to *guess* the locations of these objects. Since most programs use only a small fraction of the address space available to them, the probability p of the attacker guessing a valid address within the process memory is small. For instance, $p = 0.024$ for a process using 100MB of memory on a 32-bit processor. This means that a dereference of the pointer will, with a high probability, cause a memory exception, which leads to a UNIX signal (segmentation fault, bus error or illegal instruction). The receipt of this signal initiates a forensic analysis described in the next section. Before proceeding to describe this analysis, we make two observations:

- There is a non-negligible probability p that an attack does not lead to a memory fault immediately, but crashes after executing several (possibly random) instructions[1]. While a more complex forensic analysis can cope with some cases of delayed detection, in other cases, execution of random code can corrupt process memory to the point of making such analysis impossible. Our approach to the problem of delayed detection is to simply wait for the next instance of the attack. Thus, if p is large, we may need multiple attack instances to generate a signature — for instance, if $p = 0.5$, we need an average of two attack instances before a signature is generated. Since the value of p is much smaller in practice (see above), signatures were generated with just one attack instance in all our experiments.
- The ability to carry out forensic analysis is not closely tied to ASR. Indeed, techniques that provide more prompt error detection, such as complete memory-error protection or ISR (for code-injection attacks) will make forensic analysis simpler.

[1]The likelihood that the attack will actually succeed on the first attempt is very small (of the order of 10^{-4} in the case of PaX). If this risk is unacceptable, then one can rely on complete memory-error protection techniques that can stop all memory-error exploits.

3. Input Correlation

The correlation step identifies the specific network packet (or flow) involved in an attack, and the bytes within this packet that were responsible. This enables the signature search to be focused on the relevant parts of input, enabling higher-quality signature generation. Our correlation approach uses a forensic analysis of victim process memory at the point of attack detection.

Figure 1 shows the input processing cycle in a buffer overflow attack. First, some malicious input is received by a victim process. The manner in which this input is handled will depend on the internal state of the process, as well as the content of the input. If the attack is properly crafted, control-flow will eventually reach a vulnerable fragment of the victim program. Here, some part of the input gets copied into a buffer that is too small, and hence the input overwrites a pointer value that is past the end of the buffer. As discussed previously, the attack is detected when this corrupted pointer is used, and causes a memory exception. Note that in Figure 1, the address that causes memory exception is M, and it can be obtained by querying the OS. At this point, a simple approach for correlation is to search for the occurrence of M within recent inputs. However, due to the small size (i.e., 32-bits) of M, there may be occurrences of M within benign inputs. This likelihood is increased when "offset errors" are present, i.e., the attacker misjudges the distance between the beginning of the buffer and the vulnerable pointer, and as a result, the intended pointer value may be P rather than M. In this case, M may contain a benign, commonly occurring value rather than a pointer value.

To minimize the likelihood of matching with benign inputs, our approach searches for the entire buffer contents ("$DEF \cdots P$" in Figure 1) within recent inputs. To do this, we first need to locate the address $a2$ at which M is stored – the value of M alone isn't sufficient. Location of $a2$ is described in Section 3.1. Once $a2$ is obtained, our analyzer searches for the longest common substring between recent inputs, and the region of memory surrounding $a2$. In the example of Figure 1, this step will identify the highlighted part of external input as constituting the attack.

The method described so far handles buffer overflows, which are associated with most memory error exploits such as stack-smashing and heap overflows. Although format-string attacks do not involve a buffer overflow, they share the key characteristic used in the analysis described above: namely, the forged pointer (i.e., attacker specified pointer value such as M) occurs in the middle of attacker-provided data residing in a buffer. Consequently, they can be handled by our analysis. This leaves certain forms of integer overflows, where the forged pointer value does not occur contiguously with any other attacker-provided data, as the only class among memory error exploits that cannot be handled by our approach. This limitation is further addressed in Section 3.2.

3.1 Identifying the Location of Vulnerable Pointer

Depending on the type of corrupted pointer, we consider 3 cases.

Return address corruption. In this case, a memory exception occurs immediately after the return, i.e., when the processor attempts to fetch the instruction at the return address. By the semantics of the return instruction in x86, the stack pointer (SP) register will contain the value $a2 + 1$ in this case. We confirm a return address corruption attack by checking for the presence of M, which has the same value as the instruction pointer (IP), at location $SP - 1$, and then take $SP - 1$ as the value of $a2$. In the x86 architecture, a return instruction can take an operand that specifies the number of additional bytes to pop off the stack. To handle this case, we need to search the region near SP for M, since we don't know the exact number of bytes that were supposed to be popped off.

```
PC1: mov    0x10(%ebp),%eax
PC2: mov    0xfffffa90(%ebp),%ecx
PC3: addl   $0x4,0x10(%ebp)
PC4: mov    (%eax),%eax
PC5: mov    %ecx,(%eax)
```

Figure 2: Code involved in a format string attack

Function pointer corruption. In this case, a memory exception occurs immediately after the call. By the semantics of the call instruction on x86, the location of the instruction following the call instruction will be stored at address SP. Using this information, we can decode the call instruction to determine its operand. Note that a call involving a function pointer must be an indirect call, which means that the address will be stored in some location, and this location will be specified as an operand to the call instruction. In the following paragraph we describe how to obtain $a2$ from the operand to this instruction.

Data pointer corruption. In this case, a memory exception arises in the instruction that takes the corrupted pointer value as an operand. At this point, we can obtain both the instruction causing the fault (by querying the OS for the value of IP), as well as the address of invalid memory dereference. Once the instruction is determined, we proceed in the exact same manner as with function pointers. In particular, the operand may be a memory location or a register. In the first case, the location specified in the instruction is the value of $a2$ we are likely looking for. In the second case, we need to trace back further to determine the memory location from which the register operand was loaded. A general solution to this problem requires accurate disassembly and data-flow analysis of binaries, which is a hard problem.

In order to ensure simplicity and practicality of our technique, we developed a simpler implementation technique that relies on the following observations about buffer overflows. First, stack-smashing is the most common exploit, and it does not require this sort of binary analysis at all. Second, most other attacks rely on vulnerabilities in commonly used library functions, e.g., heap overflows rely on a section of code in heap management functions (`malloc` family), while format-string attacks rely on a code section within `vfprintf`. Therefore, we can statically construct a table that identifies the instruction sequences associated with commonly exploited code sections, and specify in advance the source of $a2$. For instance, consider Figure 2, which shows the vulnerable code fragment within `vfprintf` on RedHat Linux 7.3. A memory exception arises when the instruction at PC5 is executed. By examining instructions at PC1 and PC4, it can be seen that the value of EAX register came from the contents of $EBP + 10$. However, this location has been incremented by 4 by the instruction at PC3. Thus, the value of $a2$ is given by $[EBP + 10] - 4$. Our table stores the byte sequence corresponding to the instructions stored at PC1 through PC5, and associates it with the expression $[EBP+10]-4$ for computing $a2$.

Using the above table-based approach, we were able to identify $a2$ in all the attacks used in our evaluation. So far, we have needed to include only two entries in our table, one corresponding to format string attacks and another corresponding to heap overflows. Although we discuss only one instance of each of these types of attacks in our evaluation, we have actually tested our system with other attacks (on lesser known servers) and found that the two table entries worked in those cases as well.

3.2 Limitations and Their Impact

Our input correlation assumes that an exact copy of some part of the attack input would be found around the corrupted pointer. As discussed earlier, this assumption may be violated by some integer overflow attacks, in which case our correlation step can only point out the occurrence of M within recent inputs. Even worse, some form of encoding/decoding (e.g., URL encoding) may be applied to an input before it is copied to a vulnerable buffer. In this case, we cannot even locate M within the input.

In the cases mentioned above, the correlation step may fail to provide information to pinpoint the inputs (or locations within these inputs) that led to attacks. Nevertheless, signature generation can still succeed — we simply need to consider all recent inputs as suspicious, and proceed from there. Indeed, successful signatures can be generated in spite of a failed correlation step for the examples used in our evaluation. This is because of the large difference (in terms of size as well as character distribution) between benign and malicious inputs. If the differences weren't as pronounced, or if the attacker specifically crafts an attack that misdirects the signature generator as to the fields involved in the attack, then signature generation may not succeed, or may produce overly specific signatures.

4. Identifying Input Context

Most network protocols involve many different types of messages and message fields, each of which may have different characteristics. However, attacks typically exploit a specific vulnerability that affects only a single message type, and/or specific fields of a message. As a result, a signature that is matched against all types of messages (and its fields) can lead to false positives. To minimize this likelihood, our approach uses a simple specification of message formats to parse a message and break it up into its components. This enables signature generation (and matching) to focus on meaningful components of the message, rather than quantities such as offsets within network packets. As we show later in this section, very simple input specifications are sufficient for our purposes, and the information needed to write these specifications is readily available from network protocol specifications such as the IETF RFCs.

4.1 Input Format Specification Language

Network servers may use text-based protocols or binary protocols. Regular expressions are an obvious choice for specifying the format of text-based protocols. Thus, our format specification language extends regular expressions to support binary protocols. The syntax of the language is based on the syntax of format strings in C and the regular expression language used by the Lex lexical analyzer generator tool.

A format specification in our language describes input format using a sequence of format definitions. These definitions have the form $name = format$, where $name$ denotes a name for a format specification, and $format$ denotes its value. The $format$ argument may directly describe the format of a message field, or may refer to other format specifiers using their name. A special format-specifier name `message` is used to specify the format of the entire message. Input to the server is expected to be a sequence of strings that match `message`.

Below, we describe the syntax and semantics of format specifiers. The semantics is given in terms of the set of strings that match a given pattern. There are three types of basic format specifiers.

- *binary specifier*: `%B` matches any single byte, while `%b` matches any single bit.
- *text specifier*: a regular expression can be used to specify text formats, and its semantics is as usual.
- *named specifier:* `%`$name$ matches the same set of strings as format specifier x, where $name$ has been previously defined using a definition of the form $name = x$.

The following two constructs enable the value matching a pattern to be remembered and used subsequently.

- *binding:* The format specifier $name = format$ matches the same string as $format$, but has the additional effect of assigning the string that matches $format$ to the variable $name$. For efficiency of matching, bindings are restricted so that they are deterministic. This enables bindings to be handled without backtracking.
- *conditional pattern:* The format specifier <*cond*>$format$ matches the exact same strings as $format$, provided *cond* is true. Otherwise, it doesn't match any string. The condition component may refer to variables that have previously been bound using the previous construct. They may also use simple arithmetic and string operations.

The basic specifiers can be composed by the following operators.

- *Concatenation:* A string s_1s_2 matches a format specifier F_1F_2 iff s_1 matches F_1 and s_2 matches F_2.
- *Repetition:* A format specifier $F*$ matches s if $s = s_1s_2 \cdots s_n$, where $n \geq 0$, and each s_i matches F. A tighter bound on the value of n can be specified using bounded repetition: $F\{l, m\}$ has the same meaning as $F*$, provided $l \leq n \leq m$. $F\{m, m\}$ is abbreviated as $F\{m\}$. $F+$ is a short form of $F\{1, \infty\}$.
- *Choice:* A string s matches a format specifier $F_1| \cdots |F_n$ iff s matches any of F_1 through F_n.

4.2 Illustrative Examples

In this section, we illustrate the format specification language with a few examples, starting with the FTP protocol.

```
WORD = [^ \t\n]*
LINE = [^\n]*\n
message = ([ \t]*)(cmd=%WORD)(params=%LINE)
```

This specification states that each message is on a line by itself. The first word in the line, which matches the regular expression named WORD, specifies the name of an FTP command. The rest of the line describes the parameters to this command. The above specification does not attempt to capture the format of these parameters in detail, but simply states that, for the purposes of signature generation, every thing until the end of line should be considered as one field. This ability to leave out detailed specifications (for some parts of the protocol) is the reason for the simplicity of our format specifications.

In the context of generating signatures, bindings have an additional effect: only those parts of an input that are matched and bound to variables are used in signature generation. This provides a way for programmers to exert some control over the message fields that are used in signature generation.

We now illustrate the specification of a more complex protocol, namely, SMTP:

```
CMD = [^ \t\n]*
WORD = [^ \t\n]+
LINE = [^\n]*\n
HEADER = ([ \t]*)(hdrtype=%WORD)(%LINE)
BODYLINE = (\n)|(\.[^\n]+\n)|([^\.][^\n]*\n)
EMAILBODY = (%BODYLINE*)(\.\n)
EMAILMSG = (%HEADER+)(%EMAILBODY)
EXTRA = <type == "DATA">%EMAILMSG
message = ([ \t]*)(cmd=%CMD)(%LINE)(%EXTRA|ϵ)
```

At the top level are SMTP protocol messages such as HELO. The actual email message appears immediately following the protocol command DATA. Following this line, there are a list of email headers, followed by an email body. The boundary between the header and message body is given by a blank line. The end of email message body is identified by a line consisting of just one "." character. In spite of the fact that SMTP is a complex protocol, our input specification is only several lines long.

Next, we illustrate the language for the SMB file-sharing protocol that uses binary data.

```
message = (%b{15})(len=%b{17})(%B{4})
          (type=%B)(%B{len-5})
```

In SMB specification, the first four bytes correspond to the session header, in which the last 17 bits specify the length of the message. Following the header is the message body, of which the fifth byte specifies the message type. This specification ignores the details of the first 4 bytes of message (captured by the format string %B{4}), and then stores the fifth byte in a variable named type. Finally, the remaining sequence of bytes must have a length of five bytes less than the length specified in the session header.

5. Signature Generation

After the three preceding steps have identified the potentially malicious field and malicious data within it, our signature generation algorithm generates a signature by comparing this malicious data with data appearing in the same field in benign messages. The signature generation algorithm needs to satisfy the following requirements:

- *Virtually zero false positive rate.* It is critical that automatically generated filter doesn't discard legitimate inputs.
- *Protection from variations of previously encountered attacks.* Any sophisticated attack, including those launched by polymorphic worms, is characterized by the fact that the exact details of the attack vary from one attack instance to the next.
- *High performance.* In order to meet our goal of sub-second deployment of filtering rules, the filter generation must be very fast.

General-purpose classifiers such as RIPPER [8] provide powerful rule generation capabilities, but do not satisfy all of the above requirements. Therefore, we have developed a simple, light-weight rule generation algorithm that exploits unique features of buffer overflows to satisfy the above requirements. Our algorithm makes use of all available benign input samples, which can include *all* of the inputs ever seen by a server. It ensures that *none of these inputs will be filtered out,* thereby minimizing the likelihood of false positives. In order to provide high performance, it operates in an incremental fashion: it maintains certain summary statistics about all benign inputs, which can be quickly compared with those of attack inputs when they are encountered.

Our algorithm uses two basic characteristics of inputs: (a) length of input fields, and (b) distribution of characters in a field. A generated signature may use (a), (b), or both. Buffer overflows are usually characterized by excessively long input fields, and moreover, contain binary data such as addresses and executable code that significantly alters the statistical characteristics of input. By using rules based on these characteristics, we increase the likelihood that the signature captures properties of the underlying vulnerability, over which the attacker has no control, as opposed to the specifics of the attack string that he/she can control.

We illustrate signature generation with a few examples. The signatures are internally represented in our system using a binary format. We present them using a textual representation for readability. The first example is a size based signature, which is generated for the OpenSSL heap overflow vulnerability (discussed in Section 7).

$$\{type = 2, data.size > 420\}$$

Here, type and data refer to variable names defined in the input format specification for OpenSSL. This signature says that the system needs to drop an input if one of its field's type is 2, and the

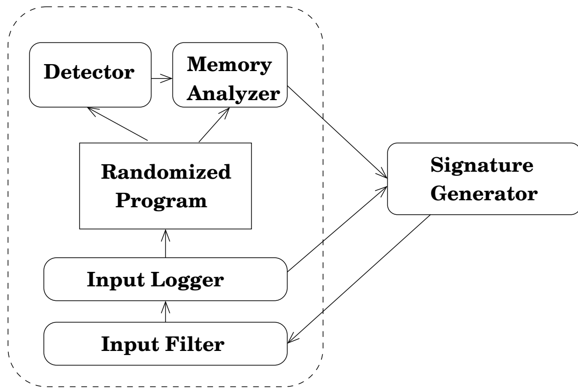

Figure 3: Architecture of our implementation.

field's size is greater than 420. The next example is uses character distribution based signature for FTP.

$$\{cmd = \texttt{"SITE"}, params.size > 452 \ \&\& \ \text{non-ASCII}(params)\}$$

This signature specifies that an attack has size greater than 452 and contains non-ASCII characters.

6. Implementation

We have prototyped COVERS on Red Hat Linux 7.3. The architecture of our prototype system is shown in Figure 3. The components are divided into two groups: internal components and external components. Shown in the dashed box, internal components hook into the protected program by means of library interposition. They implement the functionalities of attack detection, attack input correlation, and input context identification. The external component runs in a separate process, which implements the functionality of signature generation.

We chose library interposition, rather than system call interposition, to implement the internal components. This is because the former has less overhead and can intercept operations at higher layers, e.g., `scanf` function, as opposed to `read`. The main weakness of library interception is that it can be bypassed by a successful attack. However, in our case, we are interested in examining inputs *before* an attack compromises the protected application, and hence this weakness does not pose a problem for us. Moreover, ASR stops the attack before it can cause any damage, and hence bypassing itself is a non-issue.

Library interposition is implemented by providing wrapper functions for the library functions we are interested in, and using the LD_PRELOAD mechanism to redirect calls to these wrappers.

6.1 Input Filter and Logger

The input filter intercepts all inputs consumed by the protected application. Inputs from different sessions are kept separate, while inputs corresponding to a single session are aggregated together into a single string[2]. This string is broken into fields, based on the input specifications. The result of this matching is a series of bindings to variables used in signatures. With these bindings in place, the active signatures are evaluated. The input filter drops any input that matches a signature. For TCP-based services, the associated connection is severed, and an error-code signifying a broken connection is returned as the return code for the most recently intercepted input call. As network servers expect and handle these types of network errors, they invoke appropriate error recovery code on such an error return, and bring the server back to a normal state without the need for a restart.

Unmatched inputs are forwarded to the protected program, as well as logged by the input logger to maintain a history of inputs to the program. These logs are buffered in memory[3]. Those inputs that do not trigger the detector are labeled as benign inputs. Periodically, the main-memory buffer is examined, and the key characteristics of each benign input (such as the length of each recognized field, its character distribution, etc.) are written to the disk for subsequent use in signature generation.

The most complex part of the input filter is that of parsing them using format specifications described in Section 4. This parsing code is generated by first translating our format specifications into Lex specifications, and then using Flex to compile them down to C-code. The details of this translation aren't very important for the purposes of this paper, and hence to conserve space, they are not provided here. Our compiler relies on the capability of Lex specifications to describe regular-expression based as well as FSA based lexical specifications. (The concept of *start states* provides support for FSA-based lexical specifications in Lex.) The regular expression capability is useful for matching text-based inputs, while an FSA based specification is useful for handling binary data.

The parser needs operate in an incremental fashion, parsing the input as it is read, and making the results available to the signature generator and matcher. However, the program generated by Flex expects to read all its input all at once, and provides no way to "suspend" itself while awaiting more input. In order to make this parser incremental, we take the following steps. First, we use a separate stack to run the parsing program. Next, when the parser attempts to read more input when there is no more available, it is effectively suspended by switching back to the original stack, and continuing. Essentially, the parsing code and the rest of the application (including all the rest of our defense code) run as co-routines.

6.2 Detector and Memory Analyzer

The detector monitors the execution of the protected program, and activates memory analyzer to correlate a crash to its corresponding input event when an attack is observed. Both components are implemented as signal handlers.

In order to detect an attack and analyze the process memory, we replace the signal handler for SIGBUS (bus error), SIGSEGV (segmentation violation) and SIGILL (illegal instruction) signals with our own. This is done when our interposing library initializes. When the protected server attempts to install a new signal handler or change it, our wrapper functions intercept these calls, and ensure that our library will continue to be the one to get these signals first.

Our signal handler uses a separate stack, so that it can function even if the stack pointer contents are invalid at the time of signal delivery. Moreover, this approach helps preserve the process memory image at the point of memory exception, thereby permitting a more accurate forensic analysis. The Linux signal mechanism provides a way to access the program's context when the signal occurs, such as the value of processor registers, which is used in our analysis.

[2] Session aggregation is simplified in our approach since we rely on library interposition. Specifically, a series of input actions involving a single file descriptor are assumed to belong to a session until the descriptor is closed.

[3] In-memory buffering may seem to be vulnerable to corruption by an attack, but since we rely on ASR to detect attacks before any damaging code can be executed, this is not a problem.

6.3 Signature Generator

The signature generator uses the input identified by the memory analyzer to generate a signature as described in Section 5, which is then deployed in the input filter to make the system immune to future attacks.

7. Evaluation

In this section, we present an experimental evaluation of our approach. Our experiments were conducted on Red Hat Linux 7.3 operating system running over VMWare Workstation 4.5. Runtime measurements were conducted *without* using VMWare on a dual Intel Xeon 1.70GHz CPUs and 2GB main memory. All servers programs used in our experiments were protected using address-space randomization.

7.1 Experience

In this section we describe our experience with three types of attacks, stack overflow, heap overflow, and format string. We illustrate how our system works using examples from each category.

OpenSSL heap overflow attack. OpenSSL 0.9.6d and older versions have a remotely exploitable vulnerability in handling client master keys. This was the vulnerability exploited by the Linux Slapper worm. Our evaluation was performed on an Apache web server with the vulnerable SSL module. Normal requests were generated using a Mozilla web browser, which requests web pages using a version 2 SSL protocol. The attack was then launched, which caused the vulnerable server to crash with a memory error. Our system detected the SEGV signal. Using our table-based analysis, our system identified the address in the vulnerable buffer. Our approach then searched recent inputs, and found that the second most recent input contained the longest matching substring (of length 418 bytes) with the vulnerable buffer. Using input format specification, the attack was linked to the "Client Master Key" message (represented using a integer), whose length was found to be 421 bytes. The signature generation module then retrieved the statistics on all "Client Master Key" messages seen during normal execution, and found that the maximum message length was 204 bytes. Therefore, a signature was generated to filter out all "Client Master Key" messages larger than 420 bytes. After the signature was deployed, the attack did not affect the server, while normal server operation was unaffected.

Samba stack overflow attack. Samba server versions 2.2.8 and earlier have a vulnerability in processing a "transaction 2" open request. The server fails to perform bounds-checking on a buffer that holds the name of the file to be opened. A long name in the request will overwrite the return addresses on the stack. In our evaluation, the server crashed when the overflowed return address was used. Our system successfully identified the vulnerable buffer near the stack top. The correlation step matched the vulnerable buffer with a recent input, with the length of the match being 900 bytes. Using input format specification, this message was identified as a "transaction 2" open. This request had not been observed under normal operation, i.e., maximum benign size was 0, while the attack size was 2080. A signature based on this length was successfully generated.

WU-ftpd format string attack. There is a format string vulnerability in wu-ftpd server version 2.6.0 and earlier. The vulnerability is in "SITE EXEC" command, in which user-provided data can be used as a format string to `printf`-family functions. When the exploit program specified an address to be changed via the format string, it resulted in a memory error. As in the above two cases, our system successfully identified the vulnerable buffer, and matched the attack-bearing input message. The input message is 453 bytes, while normal message is around 50 bytes. Moreover, normal commands are all ASCII characters, while the attack contains binary characters. Based on these facts, a signature that uses both the length and character distribution information was generated.

Importance of input specification. We can see the importance of input specification through the above examples. For example, in the OpenSSL heap overflow vulnerability, the server accepted other types of messages larger than the malicious "Client Master Key" message. If the input format specification was not used, we could not generate a length-based signature. Moreover, since OpenSSL is a binary protocol, it is unlikely that a robust signature based on character distributions can be derived,

For Samba, once again the protocol uses a binary format. In addition, some of the messages, especially those involving file data transfer, can be much larger than the transaction2 open request. Thus, it is unlikely that a successful signature can be generated without input format specification.

For WU-ftpd, signature generation would likely succeed without input format specification, as the attack-bearing input contains binary data, instead of pure ASCII data in normal ftp sessions. However, if the attacker's goal is to cause denial-of-service, he/she can use pure ASCII as the attack payload to evade detection.

7.2 Performance

Effectiveness in signature generation. In our evaluation, we are interested in the result of our approach on "real-world" attacks. The attacks used in our evaluation were selected from the website `securityfocus.com`. In selecting attacks, our first criterion is the vulnerable program's popularity, as popular programs' vulnerability have more real-world impact. Also, it is less likely for these programs to contain obvious bugs, and thus the attacks on them tend to be more sophisticated. Another criterion is the availability of exploit code. Since developing exploit code is a non-trivial effort, we limited our selection to the attacks that have working exploit on Linux. The attacks evaluated are shown in Table 1, and our approach successfully generated signatures for all the attacks.

Speed of signature generation. In all the evaluated examples, our approach generated signatures within 10 milliseconds after the attack is detected.

Performance overhead. We measured the performance overhead of our system on an Apache web server, in which the server's CPU time is used as the metric. (The server throughput and latency metrics are likely to show much lower overheads because the workloads tend to be limited by the 100Mbps network bandwidth.) The workload was produced by a script that downloads a set of files with size ranging from 500 bytes to 5M bytes. The script was repeated for 100 times in each test.

The regular Apache server took a CPU time of 2.52 seconds to complete the task. The Apache server protected by our approach took 2.70 seconds to finish the task, a moderate 7.1% overhead. In this test, no filters were deployed. To test the performance impact caused by filtering inputs, we loaded 100 character distribution filters into our system, and the CPU time increased to 2.74, a total 8.7% overhead. As the character distribution filter is more expensive than length-based filters, in common case, the performance overhead caused by deploying filters is rather small.

Protecting service availability. We also measured the availability of three key servers under repeated attack. The servers are Apache, BIND, and NTPD. Our results shows that COVERS enables the

Program Name	Attack Type	Bugtraq ID	Vulnerable Message Type	Ratio of Attack/Benign Size
ntpd	Stack Overflow	#2540	Read variables	234%
samba	Stack Overflow	#7294	Trans2open	4333%
passlogd	Stack Overflow	#7261	Log Type	18000%
epic4	Stack Overflow	#8999	CTCP nickname	7780%
gtkftpd	Stack Overflow	#8486	MKDIR command	1130%
wu-ftpd	Format String	#1387	SITE command	860%
apache mod_ssl	Heap Overflow	#5363	Client master key	207%

Table 1: Attacks used in effectiveness evaluation and the ratio of attack input size to benign input size.

protected servers to tolerate at least 10 times attacks per second at a given server availability. For example, 50 attacks/second can reduce an unprotected Apache server's availability to 70%, but to achieve the same effect on a server protected by our approach, the attack rate need to be 500 attacks/second. On NTPD and BIND, the availability gains were even more significant.

7.3 Quality of Signatures

False positives. In order to evaluate the quality of signatures, we manually verified the signatures by analyzing the source code of the vulnerable programs. For some of the programs, such as `passlogd`, `ntpd`, and `gtkftpd`, the generated signature won't have false positives: Any input matching the signature will definitely cause memory error in the program. For the rest of the programs, as we can see from the last column of Table 1, the difference between the size of attack input and that of benign input is very large. Therefore, the probability of a false positive is low.

Polymorphic attacks. Polymorphic attacks are based on changing the attack payload frequently. But they cannot change the fact that the vulnerable buffer is overflowed, and the fact that binary data such as return addresses must be included in the attack input. Because the signatures generated by our approach tends to capture the characteristics of the underlying vulnerability, and not the characteristics of a specific instance of an attack, polymorphic attacks will likely be defeated by our signatures.

7.4 Protection from Guessing Attacks on Randomization

Randomization-based approaches such as ASR and ISR are vulnerable to brute-force and guessing attacks. [29] describes an attack on PaX address-space randomization that can successfully guess the value used in randomization in about 10^4 attempts. [35] describes an attack on instruction set randomization that succeeds, once again, using thousands of attack attempts.

With our approach protecting a vulnerable server, an effective signature will be generated within the first few unsuccessful attack attempts. After this point, all attacks will be dropped even before they reach the server, thereby ensuring that these attacks don't compromise it.

[29] suggests that perhaps the ASR used in PaX is "unfixable." They claim that (a) there is no effective response that an automated system can take when faced with such attacks, and (b) shutting down servers is unacceptably expensive. The results presented in this paper counters their claim, and provides an effective protection against attacks that require multiple attempts before succeeding.

Of course, it is possible that an attack against randomization may succeed on the first attempt, in which case, no attack is ever detected and hence no signatures can be generated. However, the probability of succeeding on the first attempt is very small with the technique of [29]. The technique of [35] requires a large random mask (several 32-bit words) to be broken step-by-step, one byte at a time. Thus, the probability of mounting a successful attack in one attempt is negligibly small. Moreover, when considering a large population of hosts, it is clear that even if some machines are successfully attacked, most other attempts will fail, and these systems will then become immune to the attack. Moreover, the immunized systems can distribute the signature to other machines, thereby providing even better protection for the population.

8. Related Work

Detection of Memory Errors and/or Exploits. Several techniques have been developed to detect attacks that exploit memory errors in C/C++ programs. Initial efforts were targeted at stack-smashing attacks [7, 10, 11]. Broader protection is provided by approaches such as address-space randomization [1, 4, 5], which, in its general form [5], can detect exploitation of all memory errors; and instruction set randomization [3, 15] (and OS features such as non-executable data segments) that can detect all code injection attacks. There have also been more comprehensive techniques for detecting all memory errors, regardless of whether they are being used in an attack [13, 14, 20, 27, 41]. When these approaches detect an attack, the victim process is generally terminated. Repeated attacks (such as those due to worms) will require repeated and expensive application restarts, effectively rendering the service unavailable.

Approaches for Recovering from Memory Errors. *Automatic patch generation* (APG) [30] proposed an interesting approach that uses source-code instrumentation to diagnose a memory error, and automatically generate a patch to correct it. STEM [31] improved on APG by eliminating the need for source code access. It uses binary emulation instead, which can be very expensive. By limiting emulation to a small section of code preceding the vulnerability, they have shown that the overall performance overheads can be kept surprisingly low (about 30%). STEM relies on code instrumentation to detect faults before there is substantial corruption of memory, so that recovery can be attempted. At this point, STEM uses a memory update log to restore the memory changes performed within the faulting function, and forces an error return from this function. A similar error-recovery strategy was used in APG as well. The difficulty with this strategy is that the application may be unprepared to handle the error-code, and as a result, may not recover. In their experiments, this strategy causes an application crash in about 10% of cases. Even when the application continues to run, it isn't clear that it will always work correctly. In contrast, our approach forces error returns for input functions. Since server applications handle such errors, and implement application-specific logic to recover from them, recovery is more reliable in our approach.

Failure-oblivious computing [26] uses CRED [27] to instrument program source-code to detect all memory errors at runtime. When it detects an out-of-bounds write, it stores the data in a separate section of memory, and returns this data when a corresponding read

operation is issued. This approach makes attacks harmless, and allows for recovery as well. The main drawback of this approach is that it often slows down programs by a factor of 2 or more.

DIRA [33] uses a source-code transformation for runtime logging of memory updates. When an attack is detected, DIRA uses this log to revert program state to a point before the last network input, and restarts execution from this point. It achieves good runtime performance by logging only global variable updates, and by supporting recovery at the granularity of function calls. This choice (to limit logging) leads to total restarts for some applications, thereby losing the benefits of light-weight recovery.

ARBOR [18] is similar to the approach presented in this paper in terms of its reliance on an application's built-in error handling capabilities for recovery. It differs from the approach described in this paper in its use of program behavior models [28] (rather than input format specifications) to identify input context. For some programs, the context information provided by the behavior model isn't sufficient to generate accurate signatures. Other differences include the absence of correlation step in ARBOR, and its inability to generate network level attack signatures.

Network-level Detection of Buffer Overflows. Buttercup [23] and [12] detect buffer-overflow attacks in network packets by recognizing jump addresses within network packets. Buttercup requires these addresses to be externally specified, while [12] detects them automatically, by leveraging the nature of stack-smashing attacks and the memory layout used in Linux. [37] suggested a more robust approach for detecting buffer overflow attacks using abstract execution of the attack payload. PayL [39] develops a new technique for anomaly detection on packet payloads that can detect a wider range of attacks. However, the technique has a higher false positive rate than the above techniques. Shield [38] uses vulnerability-specific (but exploit-generic) signatures to filter out attacks, but these signatures need to be specified manually.

Network Signature Generation. Earlybird[32] and Autograph[16], two of the earliest approaches for worm detection, relied on characteristics of worms to classify network packets as benign or attack-bearing. Honeycomb [17] avoids the classification step by using a honeynet, which only receives attack traffic.

From the packets classified as worm-related, these techniques compute a signature that consists of a single contiguous byte sequence that repeats across them. Unfortunately, polymorphic worms can modify themselves as they propagate from one host to another, thereby confusing signature generation. To mitigate this problem, Polygraph [21] can generate multiple (shorter) byte-sequences as signatures. They argue that there must be some invariants even in network flows produced by polymorphic worms, and their technique is tuned to identify these invariants. PADS [36] develops a technique called position-aware distribution signatures to detect certain kinds of polymorphic attacks.

Compared with our approach, all of the above approaches operate with rather coarse information about location and context of attacks within network flows. They don't have the benefit of either our correlation step (which identifies the bytes within network flows that are related to attacks) or the input context identification step (which identifies the location of these attack bytes within the structure of an input message). As a result, there is no way for these techniques to distinguish between relevant parts of attack-bearing flows from irrelevant parts. To compensate for this, they need sufficient number of attack samples such that only the "relevant" parts remain invariant across these samples. In contrast, using correlation and input context identification, our approach is able to generate signatures from single attack instances. Moreover, these signatures only capture message types and/or fields relevant to the attack, thereby reducing false negatives and false positives.

Nemean [42] improves on the above approaches by incorporating protocol semantics into the signature generation algorithm. By doing so, they are able to handle a broader class of attacks than previous signature generation approaches that were primarily focused on worms. Like other network-based approaches, Nemean does not have the benefit of a correlation step to pinpoint attack-containing bytes, but its knowledge of service semantics provides capabilities similar to our input context identification. However, our approach requires only simplified message format specifications, whereas their approach seems to require more detailed knowledge of service semantics that needs to be incorporated into their implementation. Another significant difference is that our technique does not require expert knowledge about which message fields are more likely to contain attacks.

Hybrid Approaches for Signature Generation. The *HACQIT* project [25] uses software diversity for attack detection. A rule-based algorithm is then used to learn characteristics of suspect inputs. The approach generates an effective signature for Code Red.

TaintCheck [22] and Vigilante [9] track the flow of information from network inputs to data used in attacks, e.g., a jump address used in a code-injection attack. Vigilante also develops the notion of self-certifying alerts (SCAs) that can be shared over the network without requiring recipients to trust each other. The signatures generated by the two approaches are somewhat simplistic — Taintcheck uses the 3 leading bytes of a jump address as a signature, whereas Vigilante relies on absolute offsets of the jump address from the beginning of input. In our terminology, these approaches don't employ context identification step, but do provide a more robust implementation of correlation. For instance, accurate correlation can be achieved even when input is transformed (e.g., using URL encoding) before overwriting a pointer. In our approach, we have traded this level of accuracy in favor of significantly better performance — zero overheads under normal operation, as compared to 10x slowdown in the case of Taintcheck.

FLIPS [19] uses PayL [39] to detect anomalous inputs. If the anomaly is confirmed by an accurate attack detector (which, in their implementation, was based on instruction set randomization), a content-based signature (using longest-common substring) is generated after several attack instances. In our terminology, their approach consists of a detection and a signature generation step, but doesn't use the correlation and input context identification steps.

Independent of our work, Xu et al. [40] developed an approach for signature generation that uses a post-crash forensic analysis of address-space randomized programs, similar to our technique. In addition to analyzing program memory, they use attack replay techniques to re-execute the vulnerable sections of code and pinpoint the source of the vulnerability. However, since their approach lacks the correlation and input context identification steps, their signatures cannot get to the root cause of a vulnerability, but rather rely on jump addresses used in an attack. Such an approach can suffer from false positives, especially with protocols that use binary data.

9. Conclusion

In this paper we presented a new approach, called COVERS, for protecting servers from repetitive buffer overflow attacks, such as those due to worms and zombies. Our approach combines off-the-shelf attack detection techniques with a forensic analysis of the victim server memory to correlate attacks to inputs received from the network, and automatically generates signatures to filter out future attack instances. This improves the ability of victim applications to withstand attacks by one to two orders of magnitude.

As compared to previous techniques, COVERS introduces minimal overheads of under 10% during normal operation. In our evaluation, we showed that COVERS signatures capture the characteristics of underlying vulnerabilities, thereby providing protection against attack variants that exploit the same vulnerability. In contrast with previous approaches, which required many attack samples to produce signatures that are sufficiently general to stop polymorphic attacks, our approach is able to generate signatures from a just a single attack instance. This is because of the use of a 4-step approach that allows COVERS to localize the source of attacks to a small part of an input message. We believe that other signature generation techniques, which often employ more powerful algorithms at the signature generation step, can derive significant benefit by incorporating our correlation and input context identification steps.

The signatures generated by COVERS can be distributed and deployed at other sites over the Internet. This deployment can take the form of an inline filter at the network layer. As a proof of this concept, we recently implemented a Snort [34] plug-in that filters out network flows that match the signatures generated by COVERS.

Acknowledgement

We thank Mohammed Mehkri and Karthik Sreenivasa Murthy for the help in collecting attack examples and in the implementation of input format specification. We thank the anonymous reviewers for their comments, which are very helpful in preparing the final version of this paper.

10. REFERENCES

[1] The PaX team. http://pax.grsecurity.net.

[2] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In *USENIX Annual Technical Conference*, 2000.

[3] E. Barrantes et al. Randomized instruction set emulation to disrupt binary code injection attacks. In *ACM CCS*, 2003.

[4] S. Bhatkar, D. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *USENIX Security*, 2003.

[5] S. Bhatkar, R. Sekar, and D. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security*, 2005.

[6] S. Chen, J. Xu, E. Sezer, P. Gauriar, and R. Iyer. Non-control-hijacking attacks are realistic threats. In USENIX Security, 2005.

[7] T. Chiueh and F. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *ICDCS*, 2001.

[8] W. Cohen. Fast effective rule induction. In *International Conference on Machine Learning*, 1995.

[9] M. Costa et al. Vigilante: End-to-end containment of Internet worms. In *SOSP*, 2005.

[10] C. Cowan et al. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security*, 1998.

[11] H. Etoh and K. Yoda. Protecting from stack-smashing attacks. http://www.trl.ibm.com/projects/security/ssp/main.html, 2000.

[12] F. Hsu and T. Chiueh. CTCP: A centralized TCP/IP architecture for networking security. In *ACSAC*, 2004.

[13] T. Jim et al. Cyclone: a safe dialect of C. In *USENIX Annual Technical Conference*, 2002.

[14] R. Jones and P. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Intl. Workshop on Automated Debugging*, 1997.

[15] G. Kc, A. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *ACM CCS*, 2003.

[16] H. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *USENIX Security*, 2004.

[17] C. Kreibich and J. Crowcroft. Honeycomb – creating intrusion detection signatures using honeypots. In *HotNets-II*, 2003.

[18] Z. Liang and R. Sekar. Automatic generation of buffer overflow attack signatures: An approach based on program behavior models. In *ACSAC*, 2005.

[19] M. Locasto, K. Wang, A. Keromytis, and S. Stolfo. FLIPS: Hybrid adaptive intrusion prevention. In *RAID*, 2005.

[20] G. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *POPL*, 2002.

[21] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *IEEE S&P*, 2005.

[22] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.

[23] A. Pasupulati et al. Buttercup: On network-based detection of polymorphic buffer overflow vulnerabilities. In *IEEE/IFIP Network Operation and Management Symposium*, 2004.

[24] H. Patil and C. Fischer. Efficient run-time monitoring using shadow processing. *International Workshop on Automated and Algorithmic Debugging*, 1995.

[25] J. Reynolds, J. Just, L. Clough, and R. Maglich. On-line intrusion detection and attack prevention using diversity, generate-and-test, and generalization. In *Hawaii International Conference on System Sciences*, 2003.

[26] M. Rinard, C. Cadar, D. Roy, and D. Dumitran. A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors). In *ACSAC*, 2004.

[27] O. Ruwase and M. Lam. A practical dynamic buffer overflow detector. In *NDSS*, 2004.

[28] R. Sekar, M. Bendre, P. Bollineni, and D. Dhurjati. A fast automaton-based method for detecting anomalous program behaviors. In *IEEE S&P*, 2001.

[29] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *ACM CCS*, 2004.

[30] S. Sidiroglou and A. Keromytis. Countering network worms through automatic patch generation. *IEEE Security & Privacy*, 2005.

[31] S. Sidiroglou, M. Locasto, S. Boyd, and A. Keromytis. Building a reactive immune system for software services. In *USENIX Annual Technical Conference*, 2005.

[32] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *OSDI*, 2004.

[33] A. Smirnov and T. Chiueh. DIRA: Automatic detection, identification and repair of control-hijacking attacks. In *NDSS*, 2005.

[34] Snort. Open source network intrusion detection system. http://www.snort.org.

[35] A. Sovarel, D. Evans, and N. Paul. Where's the FEEB?: The effectiveness of instruction set randomization. In *USENIX Security*, 2005.

[36] Y. Tang and S. Chen. Defending against Internet worms: A signature-based approach. In *INFOCOM*, 2005.

[37] T. Toth and C. Kruegel. Accurate buffer overflow detection via abstract payload execution. In *RAID*, 2002.

[38] H. Wang, C. Guo, D. Simon, and A. Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *SIGCOMM*, 2004.

[39] K. Wang and S. Stolfo. Anomalous payload-based network intrusion detection. In *RAID*, 2004.

[40] J. Xu, P. Ning, C. Kil, Y. Zhai, and C. Bookholt. Automatic diagnosis and response to memory corruption vulnerabilities. In *ACM CCS*, 2005.

[41] W. Xu, D. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *FSE*, 2004.

[42] V. Yegneswaran, J. Giffin, P. Barford, and S. Jha. An architecture for generating semantics-aware signatures. In *USENIX Security*, 2005.