

# PASAN: Automatic Patch and Signature Generation for Network Buffer-Overflow Attacks

November 8, 2007

## Abstract

Control-hijacking attacks exploit vulnerabilities in software programs to steal the control of the victim applications and eventually their underlying machines. Although much work had been done on detection and prevention of control-hijacking attacks, most of them did not provide comprehensive post-attack response, which should include an attack signature and a patch generation components. After a control-hijacking attack is detected, the signature generation component provides the front-end firewall with a corresponding filtering rule that could stop the detected attack and hopefully its variants from entering the premise, and the patch creation component furnishes a bug fix that can permanently eliminate the vulnerabilities the detected attack exploits. This paper describes the design, implementation and evaluation of a security-enhancing compiler called *PASAN*, which can transform the source code of an arbitrary C program into a form that can detect a control-hijacking attack and automatically generate a corresponding attack signature and a software patch. The automatically generated attack signatures are abstract in that they can contain regular expressions and a length constraint. The patches that *PASAN* creates is similar to those created manually so that developers can examine and merge them with the original code base with minimal efforts. We have implemented *PASAN* prototype as a GNU C compiler extension on Linux operating system that adds protection from local buffer overflow attacks to the programs that it compiles. We evaluated the effectiveness of this prototype using several network daemon programs with known vulnerabilities and found that the attack signatures *PASAN* produces can indeed stop detected control-hijacking attacks and their variants and that the automatically generated patches can successfully fix the associated buffer overflow vulnerabilities. In addition, these patches are similar in logical structure to those released by vendors in several cases, even though they are generated automatically. An execution state logging mechanism enables automatic attack signature and patch generation. Its run-time performance overhead ranges from 10% to 150% for the programs that we have tested.

## 1 Introduction

Control-hijacking attacks typically compromise some control-sensitive data structures in victim applications, such as a return address or a function pointer, and eventually usurp the control of these applications and potentially their underlying machines. Examples of control-hijacking attacks are buffer overflow attacks, integer overflow attacks, and format string attacks. There have been numerous methods and approaches proposed on how to detect and prevent control-hijacking attacks. Many of these proposals take a *dynamic checking* approach, in which the original program is augmented with additional checks to guard the program against certain anomalies at run time. For example, an array bound-checking compiler [14, 10] ensures that no array accesses in the protected program can exceed their bound; a target address checking system [17] makes sure that the target address of each indirect branch instruction (including the function return instruction) in the protected program fall within the read-only text area; a system call pattern checking system [13] prevents the protected program's run-time system call pattern from deviating from a statically derived policy. Even though these dynamic checking systems could effectively detect and prevent control-hijacking attacks, they share one weakness: lack of post-attack response that could prevent recurrence of the same attack and its variants.

An ideal post-attack response system should be able to generate a characterizing signature for the detected attack that the front-end firewall can use to block the attack and potentially its variants, and to generate a patch that can seal the security hole that the detected attack exploits. Existing attack identification systems either generated signatures [7, 24, 19] that correspond to the last received packet only and thus lead to false positives, or required a large number of attack instances [11, 18, 16] and thus cannot respond quickly enough to fast spreading worms. As for patch development, it takes 54 days on average to release a patch for a new vulnerability, according to Symantec’s recent Internet Security Threat Report [25]. This paper describes the design, implementation, and evaluation of a security-enhancing compiler called *PASAN*, which eliminates the shared weakness of dynamic checking systems by automating the process of detection, signature generation, and patch creation for control-hijacking attacks. Consequently, *PASAN* offers an effective defense against fast spreading polymorphic worms because it reduces the development time of a firewall rule and software patch for an attack to an order of magnitude.

*PASAN* takes a program transformation approach to the problem of automating attack detection, identification (i.e., generating an attack signature) and repair (i.e., generating a fix). More concretely, *PASAN* takes an application’s source code and augments it with additional instructions that check for tampered control-sensitive data structures, and record sufficiently detailed execution state from which to derive an identifying signature for a detected attack and its corresponding patch at repair time. Conceptually, the recorded execution state contains enough information to reconstruct the data and control dependencies that actually take place at run time. When detecting a control-hijacking attack, *PASAN* pinpoints the corresponding target address (for example a return address) that gets tampered by the attack. At repair time, *PASAN* computes a backward slice from the tampered target address through the dynamic data/control dependencies existing in the execution state log back to the input packets, and uses the resulting slice to identify relevant portions of relevant input packets that collectively contribute to the successful manipulation of the tampered target address. The bytes in the input packets thus identified form a matching signature that the firewall can use to filter out the detected attack.

All known control-hijacking attacks overflow a data structure in the victim programs to tamper the target address of a certain indirect branch instruction. For buffer overflow attacks, the data structure being overflowed is an array or a buffer. For format string attacks, it is the input argument list to functions such as `fprintf()` that could take a variable number of input arguments. Therefore, to generate a patch for a control-hijacking attack, *PASAN* first identifies the data structure whose overflowing leads to the tampered target address based on the data/control dependency information in the execution state log, then identifies the size of the overflowed data structure, and finally creates a bound check to stop the data structure from being overflowed. The resulting bound check and where it should be inserted constitute a software patch that permanently fixes the vulnerability.

Although *PASAN* produces attack signatures automatically, these signatures are more accurate in that they minimize the false positive and negative rate, because *PASAN*’s signatures could contain multiple disjoint byte sequences, each of which is characterized by a regular expression and/or a length constraint. To the best of our knowledge, no other systems can achieve this level of signature accuracy. Similarly, in addition to automating patch creation, the patches that *PASAN* delivers are more human-made so that they are more likely to be merged with the original source code tree without additional modifications. Other patch generation systems [21] produce patches that are used as a protection measure in addition to signature-based filtering, and therefore require additional programming effort to merge them with the original code base.

The current *PASAN* prototype is implemented as an extension to GNU C compiler version 3.4.1 and focuses only on buffer overflow attacks that overflow function’s local array/buffer and eventually modify the function’s return address. However, we believe that the same techniques could be applied to other types of buffer overflow attacks and format string attacks. The modified compiler instruments the source code of a program so that the resulting program can detect a buffer overflow attack at run time and record an execution state log, which contains logs of memory updates and control transfers. *PASAN*’s

repair-time component is implemented as a library that is transparently linked with each application being compiled. After detecting a control-hijacking attack the control is transferred to this library which implements the attack signature and patch generation based on the execution state log and the modified target address corresponding to the detected attack.

The rest of this paper is organized as follows. We review the related work in the areas of attack detection, signature generation, and patch creation in Section 2. Then in Section 3 we discuss execution state logging, whose output is the basis for signature generation and patch creation. The detailed algorithms for attack detection, signature generation and patch creation are described in Section 4, 5 and 6, respectively. We present the results of an evaluation study on the first *PASAN* prototype in Section 7. Section 8 concludes the paper with a summary of research contributions and directions for future work.

## 2 Related Work

Stackguard [6] is the first tool that used a compiler-based technique to protect the return address. A canary word is added in the stack frame next to the return address in function prolog. It is then checked in function epilog. If the word was modified then an attack is detected. RAD [5] is another compiler-based project for return address protection. The idea of RAD is to duplicate the return address in a protected buffer in function prolog and compare the value on the stack with the value in the protected buffer in function epilog. An attack is detected if the two values are different. Propolice [8], Libsafe, Libverify [2] are other tools for attack detection. BCC [10], a compiler for array bounds checking can detect a buffer-overflow attack also. However, BCC incurs a higher overhead than Stackguard and RAD. Ruwase and Lam [20] proposed an improvement to BCC that perform bounds checking for strings only.

Honeypots are often used to produce attack signatures. Earlier systems [7, 24, 19] used the last attack packet or a part of it as the signature. However, these signatures are prone to both false positives and false negatives. Context-aware signatures augment a single packet with the attack state by providing all packets relevant to a particular attack. Nemean [27] and Polygraph [18] are systems that analyze a large pool of malicious network traffic to derive multi-packet context-aware attack signatures. Autograph [11], EarlyBird [23], and Honeycomb [12] are also based on analyzing network data but they are capable of producing single-packet signatures only. PADS [26] generates signatures as a probabilistic distribution of characters for each byte of attack packets.

ARBOR [15] intercepts standard library calls and uses program’s behavior model to generate a signature when an attack is detected. It does not require access to source code. COVERS [16] improves on ARBOR with a correlation of attack packets with victim host’s memory and an input format specification language. The latter is required to generate context-aware signatures.

Sidiroglou and Keromytis [21] addressed the problem of automatic patch generation. The idea is to re-allocate the compromised local array as a global array and sandwich it in a pair of write-protected pages. The instrumented program installs a special page-fault signal handler that processes write attempts to the protected buffer. STEM [22] uses an x86 emulator called selective transactional emulation to emulate the vulnerable functions in a sand-boxed environment, undo memory updates that the attacker performed, and allow program to execute normally as if attack packets were never received.

## 3 Execution State Log

Execution state logging records the side effects of both memory updates and control transfers that take place at run time. In addition, it contains information about the location and size of static arrays and dynamically allocated buffers that are actually used in program execution. The execution state log is organized as an in-memory circular buffer.

Function class	Libc function
Copying and concatenation	<code>memcpy()</code> , <code>mempcpy()</code> , <code>memmove()</code> , <code>strcpy()</code> , <code>strncpy()</code> , <code>strcat()</code> , <code>strncat()</code> , <code>bcopy()</code> , <code>strdup()</code>
Network I/O	<code>readv()</code> , <code>recv()</code> , <code>recvfrom()</code> , <code>read()</code> , <code>fread()</code> , <code>scanf()</code> , <code>vscanf()</code> , <code>fscanf()</code> , <code>vfscanf()</code> , <code>gets()</code> , <code>fgets()</code>
Socket	<code>socket()</code> , <code>bind()</code> , <code>accept()</code> , <code>dup()</code> , <code>dup2()</code> , <code>fcntl()</code>
Format string	<code>sprintf()</code> , <code>snrptinf()</code> , <code>vsprintf()</code> , <code>vsnprintf()</code>
String/character search	<code>strstr()</code> , <code>strchr()</code> , <code>strspn()</code> , <code>strcspn()</code> , <code>strpbrk()</code> , <code>strcmp()</code> , <code>strcasecmp()</code> , <code>strtok()</code> , <code>isdigit()</code> , <code>isalnum()</code> , <code>isalpha()</code>

Table 1: The set of functions that *PASAN* proxies.

### 3.1 Memory Updates Logging

A buffer overflow attack uses one or multiple network packets to overrun a buffer in the victim program and eventually overwrite the target address of some indirect branch instruction. Therefore, the corrupted target address must be *data- or control-dependent* on attack packets. Memory updates logging is designed to track these control and data dependencies. A *data dependency* between variable *X* and variable *Y* is created when variable *X* is assigned an expression that includes variable *Y*. For example, there are two dependencies in statement  $X=Y+Z$ : *X* is data-dependent on *Y* and *Z*. A *control dependency* is created between variables *X* and *Y* when variable *Y* defines the control flow of the program and lead to the assignment of variable *X* eventually. For example, variable *X* becomes control dependent on variable *Y* when the following statement is executed: `if (Y>0) X=1; else X=0;`

Tracking control dependencies allow *PASAN* to identify the "context" that sets up the final attack packet. For example, the Blackmoon FTP server attack [4] uses a `CWD` command with an excessively long argument. The FTP server requires user authentication before processing user commands. The program saves the authentication state in a variable which we call `is_auth`. This variable is checked when the server receives a new command. If the authentication is not completed the server will not process the command. Once the authentication is completed, variables assigned during the processing of subsequent commands are control-dependent on `is_auth`. However, no data dependencies exist between `is_auth` and subsequent packets. Therefore, using data dependencies alone in attack identification algorithm will not be able to identify the authentication packets when an attack is detected.

Each memory updates log record has the following fields: `read_addr`, `write_addr`, `len`, `data`, `file_id`, and `lineno`. For an assignment statement  $X = Y$ , where *X* and *Y* are directly referenced variables, array references (e.g., `a[i]`) or de-referenced variables (e.g., `*(a+1)`), *PASAN* logs its effects into a log record, where the `read_addr` field contains the address of the right-hand-side variable of the assignment operation, in this case *Y*'s address, the `write_addr` field holds the address of the left-hand-side variable being modified, in this case *X*'s address, the `len` field is the size of the modified variable, size of *X* in this case. The `data` field is not used for this type of statements. The `file_id` and `lineno` fields identify the source code location of the statement for which the log record was generated. If *Y* is a complex expression containing a number of variables or a function call, *PASAN* traverses the expression recursively and identifies all variables in it. A log record with the same write address of the left-hand side variable but a different read address is created for each identified variable. Similarly, a log record is created for each argument of the function call on the right-hand side of the expression.

To avoid recompilation of library functions, *PASAN* proxies standard library function calls that modify the calling program's memory state such as `memcpy()`. Each call to a library function is replaced with a call to a proxy function from *PASAN* library. The compiler proxies the following classes of *libc* functions: network and file input functions, socket functions, memory copying/concatenation functions, and format string functions. Table 1 enumerates the functions that *PASAN* currently proxies.

Whenever a proxy function is called, it produces a memory updates log record that describes the

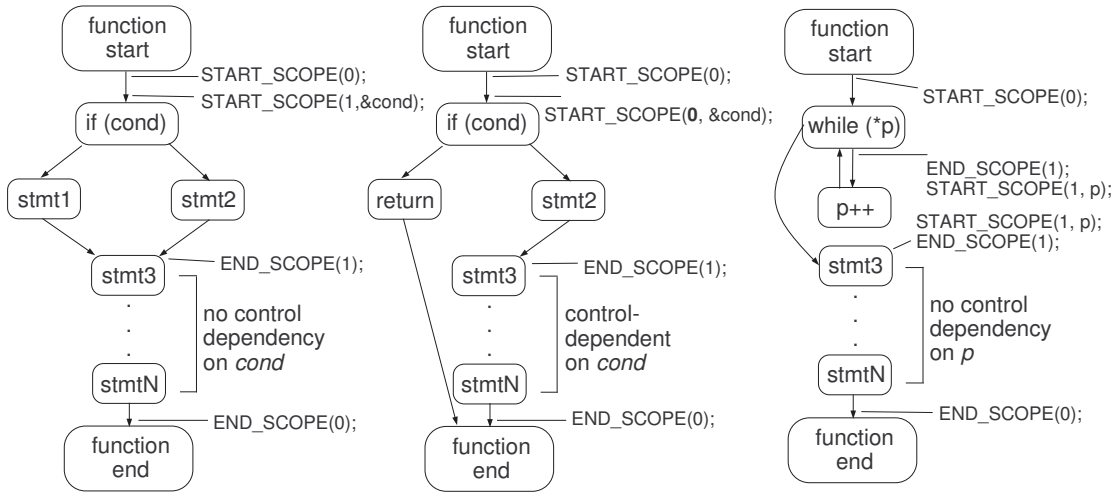


Figure 1: Control dependencies and code instrumentation. `START_SCOPE` and `END_SCOPE` records added after `while` are used in patch generation algorithm.

side effects of the original library function. The fields of the memory updates log entry are set differently for different functions. For a string copying function, for example `strcpy(a, b)` the read address field is set to `b`, the write address is set to `a`, and the length equals `strlen(b)+1`. The data field is not used in this case. For a network read function such as `recv()` the read address is set to tag value `RECV_TAG`, the write address is set to the address of the destination buffer, the length is set to the number of bytes read, and data field stores the data read from the network which is used for attack identification and patch generation. For a format string function a number of log records, one for each input argument, are created. In this case, for each input argument, the read address is set to the address of the argument and the write address is set to the address of the proper location in the destination string.

### 3.2 Control Transfer Logging

The control transfer log also contains information that marks function boundaries and control dependencies. The memory updates log records and control transfer log records are stored in the same physical log.

The scope of each conditional expression, loop-closing expression, and function is assigned a scope ID. For each function, `PASAN` logs a `FUNCTION_START` record and a `FUNCTION_END` record with a specific scope ID to mark its function boundary when the program is executed. A `FUNCTION_START` log record is added to function's prolog and a `FUNCTION_END` log record is added to function's epilog. A `START_SCOPE` record and `END_SCOPE` record with a specific scope ID are created to mark the boundary of the scope of each conditional conditional expression or loop-closing expression. In addition, the `START_SCOPE` record also contains the address of each variable that is used in the conditional expression or loop-closing expression. All variables corresponding to write addresses of the memory updates log records that appear between a `START_SCOPE` record and the corresponding `END_SCOPE` record with the same scope ID are control-dependent on the conditional variables listed in the `START_SCOPE` tag.

Examples of statements that use conditional expressions are `if`, `while`, `for`, `do-while`, and `switch`. All variables on the left-hand side of assignment statements inside a loop's or a if-then-else statement's body become control-dependent on its conditional variables. However, the exact scope of a conditional expression or loop-closing expression could vary depending on the actual instructions used. For example, for an `if` statement, the control flow usually continues with the statement following it no matter which branch is taken. If, however, one of the branches contains a `return` statement then the statements

following **if** never get executed if that branch is taken. Therefore, the variables in the statements following this **if** statement become control-dependent on the conditional variables. More formally, a variable **X** of the left-hand side of statement **S** is control-dependent on a conditional variable **C** used in a conditional expression if certain values of **C** prevent **S** from being executed. Figure 1 illustrates the control dependencies and how *PASAN* handles them. In addition to **return**, **break** and **continue** operators also extend the scope of an **if** statement that is inside a loop until its end. Similarly the scope of a loop-closing expression in **for**, **while**, and **do-while** can be extended past the loop body if the latter contains a **return**. Other control-flow operators such as **break** and **continue** do not affect a loop-closing expression’s scope because the control flow will continue with the statements following the loop when it terminates. Figure 1 gives examples of abstract syntax trees and their instrumentations.

### 3.3 Array Allocation Table

In addition to data and control dependencies, *PASAN*’s execution state log also contains information about the location and size for arrays/buffers, because *PASAN*’s repair-time library needs this information to derive the bound checks that prevent overflows. Each entry in the *array allocation table* contains the following fields: **data** which stores the base address of an array/buffer, **len** which stores the length of the array/buffer, and **name** which stores the source code name of the array/buffer. *PASAN* parses the input program to find array definitions and dynamic memory allocation calls. Then it instruments the program to add the array/buffer definitions into the array allocation table at run time. For each local array, an array allocation table entry is added in the prolog and removed in the epilog.

## 4 Attack Detection and Identification

### 4.1 Attack Detection

*PASAN* uses two methods to detect a buffer overflow attack, one based on return address protection and the other based on bounds-checking for array accesses. Return address protection has minimal performance overhead, but cannot detect attacks that overflows an array but does not modify a return address. The bounds-checking method can detect arbitrary array bounds violations, but incurs a much larger performance overhead except on Intel X86 architecture [14]. *PASAN* uses return address protection when the program is running normally, and uses array bounds checking for testing the correctness of patches it generates after an attack is detected.

*PASAN* uses RAD [5] for detecting buffer-overflow attacks that compromise function’s return address. The modified compiler modifies the prolog of each function to add its return address to a protected buffer called return address repository (RAR), and modifies the epilog where it compares the actual return address on the stack and the one stored in RAR. If the two addresses are different then an attack is detected.

Array bounds checking makes it possible to detect a buffer overflow attack as soon as the first out-of-bound modification occurs no matter whether a target address is compromised or not eventually. Array bound checking compilers such as Jones and Kelly’s [10] represent each dereferenced pointer as **base+offset**, identify the referenced array using **base**, and check if the reference exceeds the bound of the corresponding array. The **base+offset** representation could lead to ambiguities in some cases. For example, if the *i*-th element of array **arr** inside a structure **str** is referenced, one can refer to **str->arr[i]** as **\*(str+offset(arr)+i)**. In this case **base** is **str** and **offset** is **offset(arr)+i**. However, no array with base address **str** exists because **str** is the address of a structure, not of an array, and Jones and Kelly compiler will not detect any out-of-bound references.

To eliminate the limitation of the **base+offset** representation, *PASAN* extends each array with a certain number of bytes which the original program should never access and detects array bound violations by searching the memory updates log at run time. Figure 2 shows how two local arrays on the stack are extended. Instead of checking each pointer when it is de-referenced, *PASAN* checks whether

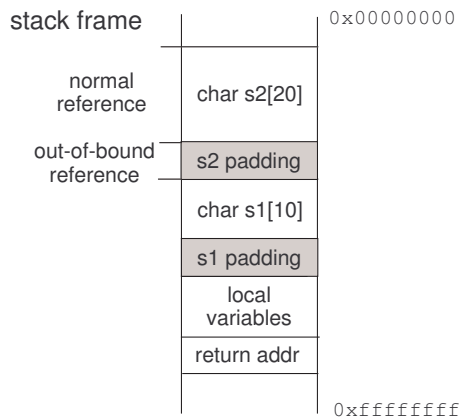


Figure 2: An improved bounds-checking algorithm. Arrays `s1` and `s2` are extended with a fixed number of bytes. The compiler returns the original size of each array as the result of `sizeof()`, but compiles the program with with extended array size.

the extended bytes of each array are accessed at the end of the function in which it appears. The extended bytes are similar to a canary word in Stackguard. To ensure that the original program never touches the extended bytes, *PASAN* re-implements the `sizeof()` function so that it returns the original size of an array. Detect all possible out-of-bounds references requires a certain number of padding bytes. This algorithm is described in Section 6.1.

## 4.2 Attack Identification

*PASAN* detects a buffer overflow attack by detecting a corrupted return address. To identify packets that are responsible for the attack *PASAN* traces back the dynamic data dependency graph constructed from the memory updates log starting from the corrupted return address. This tracing relies on the read address and write address fields of the memory updates log entries. Let us call *MA* (modified address) the address of the corrupted return address, which is tampered due to either an unchecked array-to-array copy loop or a proxied function such as `strcpy()`. Tracing starts with the most recent memory updates log entry whose write address equals *MA*, and uses the read address field of this entry as a key to traverse the memory updates log backward and find the most recent log entry whose write address is equal to it, etc. This process continues iteratively until the algorithm finds a memory updates log entry whose read address is a special tag that indicates a network read function, which means that the data written in this entry comes from the network.

Tracking only data dependencies allows one to identify the attack packet that contains the bytes used to overwrite the corrupted return address, but not necessarily all the essential packets required in the attack. For example, it cannot identify packets steering the victim program into a state vulnerable to the buffer overflow attack. To identify the latter type of packets, *PASAN* needs to track control dependencies as well. More specifically, starting from *MA*, *PASAN* finds all the variables on which *MA* is control-dependent and/or data-dependent from the memory updates log and recursively repeats the same process for each of the variables found. In the end, *PASAN*'s attack identification algorithm can identify all relevant bytes in the input byte stream that contribute to the detected buffer overflow attack.

## 5 Attack Signature Generation

### 5.1 Overview

*PASAN* is able to generate attack signatures in the form of multiple patterns in a byte stream, each of which can be characterized by a regular expression and/or a length constraint. The last pattern in a buffer overflow attack's signature typically includes a length constraint. Because *PASAN*'s signatures can contain multiple non-contiguous patterns, they tend to be more specific and thus less likely to generate false positives. Because *PASAN*'s signatures contain generic regular expressions rather than hard-coded values, they are more general and thus less likely to generate false negatives. This generality is particularly important for thwarting polymorphic worms. In addition, *PASAN* ignores the packet boundary to eliminate false negatives that arise because the senders and receivers fragment packets differently.

For buffer overflow attacks, *PASAN* also attempts to generate a length constraint for particular portions of attack packets. More specifically, a length constraint is characterized by a starting pattern  $S$ , a terminating pattern  $T$ , and the maximum length  $L$ . A length constraint  $\langle S, T, L \rangle$  means that after the pattern  $S$  appears in the byte stream, there could be at most  $T$  bytes after  $S$  before the pattern  $L$  appears; otherwise the byte stream is considered as an attack. For example, `ghhttpd` server attack packet [3] contains an HTTP `GET` request with an overly long URL. The URL is terminated by a new line character `'\n'`. The corresponding length constraint includes a starting pattern, which is a concatenation of `GET` and a space character, a single terminating character `'\n'`, and a length parameter that prevents it from overflowing the array allocated to hold the URL.

### 5.2 Abstracting Input Values into Regular Expressions

To reduce the false negative rate *PASAN*'s signature generation algorithm attempts to produce the most general signature without increasing the false positive rate. Initially, each byte in the input byte stream that leads to an attack is irrelevant. Then *PASAN*'s attack identification algorithm convert all bytes in the input byte stream that contribute to the detected attack as relevant. These relevant bytes are explicitly specified in the resulting signature, whereas each irrelevant byte is represented as a don't-care character, which means it may or may not exist in any attacks that try to exploit the same vulnerability as the detected attack.

Moreover, *PASAN* attempts to abstract a relevant byte as much as possible without over-generalization. For example, if a special function such as `isdigit()` is applied to a relevant byte, then this relevant byte is specified as a digit character in the signature if the check succeeds or as a non-digit character if the check fails. In the former case sequence `'\d'` is written to the signature file, while in the latter sequence `'!\d'` is written into it. *PASAN* does not save the actual value of the character in the memory updates log. Instead, it performs this check at the signature generation time by tracing back the argument of `isdigit()` to the network packet whose content is saved in the memory updates log.

### 5.3 Generating a Length Constraint

Because a buffer overflow attack overflows an array/buffer in the victim program, the associated attack signature should ideally include a length constraint on the portion of the input byte stream that corresponds to the overflowed array/buffer. Because consecutive fields in the input byte stream are separated by special delimiter characters, these delimiter characters are required for length constraint generation.

From a corrupted return address, which must come from a network packet, *PASAN* first determines the maximal byte range surrounding it that also comes from the same packet. It applies the data dependency trace-back algorithm to the bytes above and below the corrupted return address as much as it can, and stops when reaching bytes that cannot be traced to the same network packet that contributes to the corrupted return address. Eventually this algorithm finds a byte range surrounding the corrupted return address (*destination range*) and its corresponding byte range in an input packet (*source range*).

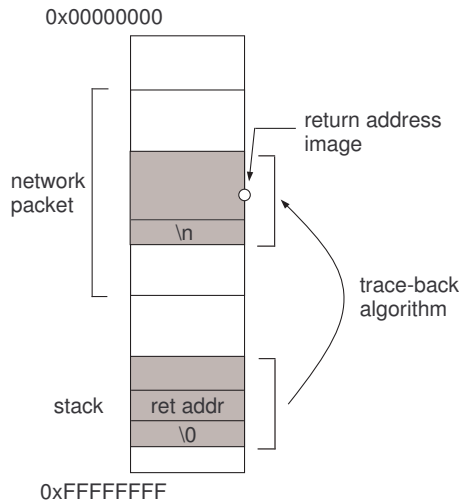


Figure 3: Generating signature’s length constraint.

The last byte of the source range in the network packet is the delimiter character. For the example shown in Figure 3 the last character in the destination range is the null character, possibly because the victim program copies data in the packet using the `strcpy()` function. However, the delimiter character in the attack packet in this case is actually the new line character ‘\n’.

Sometimes it is not possible to trace from the terminating character in the destination range back to the corresponding delimiter character in the network packet. For example, if the terminating character is written outside of a loop as `p[i]='\0'`; then then neither data nor control dependency exists between the null character in the destination range and the new line character in the source range, although logically the mapping between the two should exist. To solve this problem, *PASAN* uses the following heuristic. Let  $L$  correspond to the stack address that stores the corrupted return address. *PASAN* first identifies from the memory updates log the two addresses immediately preceding  $L$ ,  $L - 1$  and  $L - 2$ , then traces them back to the packet bytes  $P_1$  and  $P_2$  respectively, and eventually maps  $L$  to  $P_1 + (P_1 - P_2)$  using linear interpolation. The content of the address  $P_1 + (P_1 - P_2)$  holds the delimiter character. This algorithm correctly establishes a mapping between the terminating character ‘\0’ and its delimiter character ‘\n’ in Figure 3.

To determine the actual constraint on the length of an input field, *PASAN* searches the array allocation table using the destination range to locate the corresponding array’s size. Combining length parameter and delimiter character, the length constraint for the example in Figure 3 specifies that the delimiter character ‘\n’ must appear between the beginning of the destination range and the beginning of the destination range plus the overflowed array’s size in the input stream.

If the corrupted return address was overwritten by a `memcpy()` call rather than by a `strcpy()` call and the length argument of the `memcpy()` call comes from a network packet, the above length constraint generation algorithm cannot produce a valid length constraint. Indeed, there is no explicit delimiting character that stops `memcpy` from copying more data. Instead it uses the length that was read from the network packet. In this case the length constraint should include the offset of the length data field in the packet, its length, and its maximum possible value. *PASAN* handles this case by saving the address of the length argument of `memcpy()` in the memory updates log. At the signature generation time the trace-back algorithm checks whether any of these variables came from the packet. If so, it uses this value as the length constraint instead.

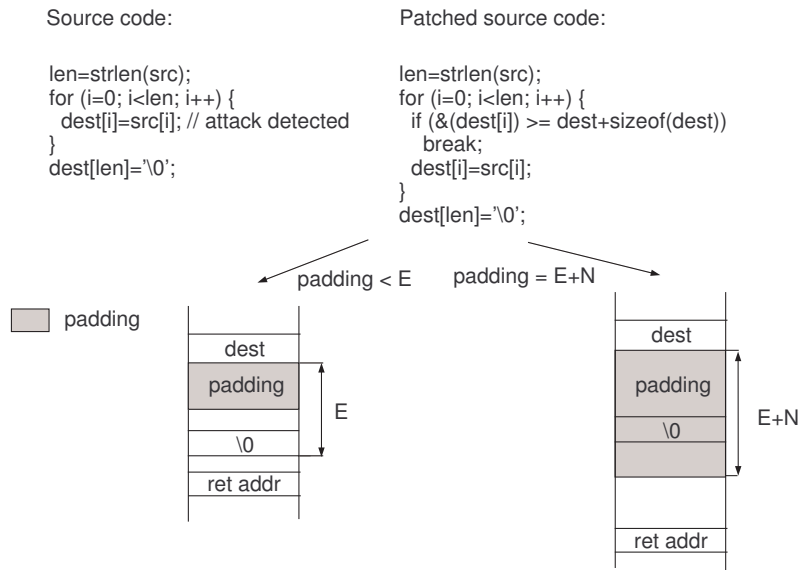


Figure 4: Finding the number of padding bytes.

## 6 Patch Generation and Testing

From the corrupted return address associated with a buffer overflow attack, *PASAN* first produces a patch and then tests the patch against the original attack packets to ensure that the patch fixes the original vulnerability as well as related vulnerabilities.

### 6.1 Patch Generation

The current *PASAN* prototype can handle the following three types of buffer overflow vulnerabilities: (1) buffer overflow because of an unsafe *libc* function such as `strcpy()`; (2) buffer overflow because of an array copying loop that eventually corrupts the return address; (3) buffer overflow that does not corrupt the return address. Only array bounds checking can detect type (3) vulnerability.

From the corrupted return address associated with a detected buffer overflow attack, *PASAN* first identifies the array that is being overflowed using the array allocation table and then the earliest memory updates log record that steps out of array's bound. Next it uses the source code file id and line number associated with this memory updates log record to find out the type of vulnerability. If the source code line contains a *libc* function name and does not contain an assignment statement, then it is type (1) overflow vulnerability. Otherwise, *PASAN* continues traversing the memory updates log backward until either of the following is true: (a) the beginning of the current function is found, which corresponds to type (3) overflow vulnerability; (b) a `START_SCOPE` record is found for which no previous `END_SCOPE` was found and the source code line specified in the `START_SCOPE` record contains a loop keyword such as **for**, **while**, etc; this corresponds to a type (2) overflow vulnerability. Each overflow vulnerability type requires a distinct patch:

**Type (1).** The idea is to replace an unsafe *libc* function with its safe version. For example, `strcpy()` is replaced with `strncpy()`, `memcpy()` is replaced with `memncpy()`, etc. The name and size of the overflowed array can be found using the array allocation table.

**Type (2).** For an array overflow that happens within a loop, the patch is to add a check in the beginning of loop's body to ensure the corresponding array bound is never exceeded. To do so, *PASAN* first identifies the array being overflowed and its size using the corrupted return address and the array allocation table. Then *PASAN* generates an if-statement that checks each reference to the overflowed

```

while (*p!='\0')           while (1) {                 while (*p!='\0')
    *dst++=*p++;           if (*p=='\0') {           *dst++=*p++;
*dst=*p;                   *dst='\0';                 *dst='\0';
                           break;
                           }
                           *dst++=*p++;
                           }

```

Figure 5: Program examples with terminating characters (left and middle) and with no terminating characters (right). The code on the left has a terminating character because of a data dependency, but the code in the middle has a control-dependent terminating character.

array inside the loop against array’s bound, as shown in Figure 4. If the destination array in the copying loop needs a terminating character, the bound used in the check should be decremented by 1. An array’s terminating character such as `'\0'` is usually written outside a loop. A character written to an array is called a *terminating character* if it is data- or control-dependent on the values of conditional expression variables of the last loop iteration and those after loop termination. *PASAN*’s patch generation algorithm finds the copying loop’s conditional variables used in the last iteration, and checks if there are any dependencies between bytes written into the array on these conditional variables. If a dependency is found then the array was terminated with a terminating character. In Figure 5, the above algorithm can correctly identify a terminating character for the left and middle case. The null character in the right case is not a terminating character because no control or data dependency exists. This array reference is a type (3) vulnerability.

**Type (3).** If from the memory updates log the overflowed array’s bound is exceeded by  $N$  bytes, *PASAN* extends the overflowed array with  $N + 4$  bytes, inserts an array bound check, recompiles the program, applies the same packets to the new version, and checks if these extended bytes are modified. If these extended bytes are updated, *PASAN* identifies the source file id and line number for these memory updates, and modifies them into a form using explicitly the destination array’s size. For the example in Figure 4, the array access outside the loop (`dest[len]='\0';`) does not depend on any variables in the loop-closing expression, therefore decreasing the number of loop iterations cannot prevent the overflowing due to this outside array access. In this case, the address checking patch will replace `dest[len]` with `dest[(len>=dest+sizeof(dest) ? dest+sizeof(dest)-1:len]`.

## 6.2 Patch Testing

After *PASAN* creates a patch, it applies the patch to the vulnerable program, recompiles the program with bounds-checking detection instead of return address detection, and exercises the same attack packets against the new version. If during this testing *PASAN* detects an attack, it generates another patch and repeats the same testing cycle. If no attack is detected, the patched program is not vulnerable to the original attack packets and the patch generation is completed. *PASAN*’s patch testing stage checks whether the produced patch indeed stops the original attack and whether the original attack exploits any other vulnerabilities.

## 7 Evaluation of *PASAN*

We used several network daemons with known vulnerabilities to evaluate *PASAN*’s attack signature generation and patch creation algorithms. The following programs were used in our experiments: `ghttpd` 1.4 — a web server, `named` 8.1 — a DNS server, `drcatd` 0.5.0 — a remote cat program, `ntlm_auth` — an

Program	Type	Vulnerability	Client request	No. of times
ghttpd	HTTP server	5960	get a 10KB HTML page	100
drcatd	remote cat	10608	get a 10KB file	100
named	DNS server	Advisory 252	lookup a domain name	1,000
passlogd	syslog sniffer	7261	send a log message	1500
ntlm_auth	web caching daemon	10500	authenticate	500

Table 2: Programs in the test suite, their Bugtraq vulnerability IDs, and the test used in measuring their performance overhead.

Program	Compilation time, s			Binary size, bytes			Requests per second		
	GCC	PASAN	Diff, %	GCC	PASAN	Diff, %	GCC	PASAN	Diff, %
ghttpd	0.516	3.35	549	39234	154901	294	219	200	10
drcatd	1.176	1.416	20	37548	135789	261	187	152	23
named	7.721	31.0	302	776664	1955742	151	1512	1325	14
passlogd	1.72	2.05	19	21828	206637	847	300	120	150
ntlm_auth	1.86	2.14	15	99347	314383	216	385	342	13

Table 3: Compilation time, binary size, and run-time overheads of *PASAN* for the test programs.

authentication program from `squid 2.5-stable1`, a web caching daemon, and `passlogd 0.1c` — a remote logging program. We used the exploit code from Bugtraq [1] and Securiteam [3]. Table 2 describes the programs in the test suite and the vulnerabilities that were exploited.

The performance evaluation study has two goals: (1) to quantify the increase in compile time, increase in binary size, and the run-time overhead that *PASAN* introduced to programs it instruments, and (2) to assess the quality of automatically generated signatures and patches as compared with vendor-released patches. We ran the experiments on an Intel Pentium4-HT 3.2 GHz-based machine with 256 MB of memory under Fedora Core 4 operating system.

## 7.1 *PASAN* Overheads

We measured the compilation time of the *PASAN* compiler and the size of the binaries it produces as compared with the standard GNU C compiler. All program in the test suite were compiled with `-g -O` flags. To evaluate the run-time overhead due to *PASAN*’s instrumentation, we sent a series of requests to the instrumented programs and measured the total elapsed time it takes to process these requests. Results of these tests for each test program are presented in Table 3. The average run-time overhead of the instrumented programs is between 10% and 150%, which makes it possible to use the instrumented programs on production-mode servers as well as on honeypots. The reason of the 150% overhead of the logging daemon is because of its simplicity. The program receives a packet from the network card, scans it in a loop once, and prints it to the system log. The instrumentation of loops adds several `START_SCOPE` and `END_SCOPE` tags. Saving these records in the memory updates log on each iteration takes longer than processing a byte of the packet. This is reason of the high overhead.

## 7.2 Effectiveness of Attack Signature Generation

*PASAN* was able to generate a signature for each program in the test suite automatically. Figure 6 lists the signatures *PASAN* generates for three programs in the test suite.

**ghttpd attack:** This attack uses an excessively long URL. *PASAN* identifies the attack packet and finds out that `ghttpd` looks at the first four characters “GET “ only but ignores the rest of the packet

1	# number of packets	2		3	# number of packets
198	# packet length	4		4	# 1st packet size
GET ? ... ?	# packet regexp	59 52 0a 00		user	
1 4	# length constraint	481		6	# 2nd packet size
	# size and start offset	4B 4B 20 54 6C 52 4D 54 56 4E 54 55 41 41 ? ... ?		passwd	
0A	# length constraint pattern	? ...	? 41 41 4c 36 79 65 50 44 69	398	# 3rd packet size
		.....		? ... ?	
	ghttpd signature		ntlm_auth signature	1 0	# length constraint size # and start offset
				0	# length constraint pattern
					drcatd signature

Figure 6: The attack signatures that *PASAN* generates automatically for three test programs after detecting an attack.

which represents the actual URL. The attack overwrites the return address using `strcpy()` function call. The length constraint generation algorithm correctly identifies the terminating character `'\n'` and the maximum size allowed for the URL.

**named attack:** This attack exploits an inverse DNS query vulnerability. In a DNS query, the type of the DNS query along with other DNS-specific parameters are specified at specific offsets of the query packet's payload. `named` checks these fields when it starts processing a DNS query packet. *PASAN* is able to find each field that `named` checks. The attack overwrites a return address with the queried host's domain name using a `memcpy()` call. *PASAN* successfully calculates that the length argument of `memcpy()` that comes from a network packet and builds the length constraint which includes the offset (byte 21) and the length represented in 2 bytes of the field that contains the queried host name.

**drcatd attack.** `Drcatd` is a daemon that allows authenticated users to view files located on a remote machine. It uses standard Unix user name and password as the authentication mechanism. Thus, each query requires sending three packets: the user name, the password, and the name of the requested file. The exploit program uses an excessively long file name after completing authentication with `drcatd`. All three attack packets are identified. *PASAN* decides that the user name and the password packets are relevant because they are used in authentication function `crypt()` and string comparison functions that are proxied. The attack overwrites the return address of a `strcpy()` function call when copying the requested file name. The terminating character `'\0'` used in `strcpy()` is identified correctly, as well as the maximum length allowed for the file name.

**squid ntlm\_auth attack.** This attack exploits a vulnerability in `ntlm_auth` Samba authentication module of `squid` and consists of two packets: a negotiation packet that creates a connection with an SMB server and a login packet. When the first packet is received, `ntlm_auth` creates a connection to Samba and stores it in variable `handle`. The function processing the next packet checks `handle` and continues only if `handle` is not NULL. The second packet is therefore control-dependent on the first packet. It is not possible to identify the first negotiation packet if data dependencies only are used. The program receives a base64-encoded message and decodes any 4 characters of the packet into 3 bytes. The decoded message is then processed. *PASAN* signature generalization algorithm uses memory updates log and identifies relevant bytes in the encoded packet that correspond to bytes of the de-coded message that the program analyzes. Protocol magic word and message type are among the bytes identified as relevant.

**passlogd attack.** One attack packet is used in this attack. The packet processing function parses the packet using `while()` loops without checking the boundary of the destination array. The source code contains two vulnerable while loops. The first loop terminates when `'>'` is found in the network packet. The second loop terminates when a new-line character is found in the packet. The destination buffer in the first loop is overwritten as a result of attack. The signature generation algorithm identifies `'>'` and `'\n'` as relevant characters and generates a length constraint specifying that terminating `'>'` character

Program	Attack packets	Patch type	# of iterations
ghttpd	1	vsprintf, sprintf	3
drcatd	3	sprintf	2
named	1	memcpy	2
passlogd	1	two while loops	3
squid	2	memcpy	4

Table 4: Number of identified attack packets, patch type and number of iterations of patch generation algorithm.

should occur within a specified number of bytes from packet’s beginning.

### 7.3 Evaluation of Patch Generation

For each program in the test suite *PASAN* was able to detect an attack, identify the corresponding attack packet(s) and generate a patch. In this experiment, we measured the number of iterations used in patch generation, the patch generation time, and the type of patches generated. These results are presented in Table 4. We describe our experiences with each of these test programs below.

**ghttpd attack.** At its first iteration the patch generation algorithm patches the `vsprintf()` function in function `Log()`. The patch testing algorithm then detects an off-by-N bug in the `sprintf()` function. Only BCC can detect this off-by-N bug because it does not overwrite the return address.

**drcatd attack.** A call to `sprintf()` inside the logging function `logIt()` overruns a local buffer. In this case, patch generation takes only one iteration which replaces `sprintf()` with `snprintf()`. The second iteration makes sure that the program is not vulnerable to the same attack.

**named attack.** The `memcpy()` function in the query processing function `req_iquery()` overwrites a local buffer using the size of the source array as the length argument of `memcpy()`. *PASAN* replaces the size argument with the size of the destination buffer.

**squid ntlm\_auth attack.** This vulnerability is similar to the previous one. *PASAN*’s patch generation algorithm replaces the size argument of the vulnerable `memcpy()` call with the size of the destination buffer. Patch testing also finds an off-by-one error when a NULL character is written into array `pass`: `pass[25]='\0'`; and the array is declared as `char pass[25]`. *PASAN* replaced the index with `sizeof(pass)-1`. This off-by-one error does not exist in the latest release of `squid`.

**passlogd attack.** The original attack does not exploit the vulnerability in the second `while()` loop. Increasing attack packet’s length enables the resulting attack to exploit both vulnerabilities at the same time. To fix the vulnerabilities, it takes two iterations of the patch generation algorithm to add a length check in front of each while loop. On the third iteration an off-by-one error was found similar to the error found in `squid`. This error was found in the latest version of `passlogd`. We sent the description of the off-by-one error to developers who confirmed it.

The exercise of using *PASAN* to generate patches for the test programs results in the following findings:

- Patch testing is a valuable addition to patch generation because it can help identify the inadequacy of the first patch and other vulnerabilities that the original attack does not exploit. *PASAN* found two off-by-one bugs in the evaluation study, one of which was already fixed and the other was brand new.
- To our surprise, the study also revealed that programmers make errors in `memcpy()` that has a length argument and is therefore similar to `strncpy()`, a safe version of `strcpy()`. *PASAN* found two `memcpy()` errors in addition to errors in string processing functions such as `strcpy()` and `sprintf()`.

- Comparison between automatically generated patches and vendor-released patches showed that they are quite similar in structure. The main difference is that vendor patches are more complete in that they perform clean-up and exit the function with an error code when bounds checking finds an illegal modification, whereas patches that *PASAN* generates enable program's normal execution.
- Patch generation and testing was fully automated for all programs in the test suite except `passlogd`, which required attack packets to be sent from a remote computer. Therefore we believe that it is possible to use *PASAN* to generate patches for previously unknown attacks as soon as they are detected without human participation.

## 8 Conclusions

Control-hijacking attacks, and buffer overflow attacks in particular, are arguably the most pervasive and thus dangerous attack method used today. Although there have been numerous efforts on thwarting control-hijacking attacks, most of them focused mainly on detection and prevention, but failed to provide an adequate post-attack response. This paper presents the design, implementation, and evaluation of a system called *PASAN* that can automatically detect a buffer overflow attack, generate an abstract attack signature to filter out the detected attack and possibly its variants, and generate a bounds checking patch to permanently seal the security vulnerability the attack exploits. The basic idea of *PASAN* is to record an execution state log at run time so that the repair-time component can track the dynamic control and data dependencies to derive the corresponding attack signatures and patches. *PASAN* not only can automate the signature and patch generation processes, it can do so transparently to application developers and users. Moreover, the automatically generated signatures typically have a lower false positive and negative rate than those created manually because they take advantage of the victim program's internal workings and could contain regular expressions and length constraints. Finally, the automatically generated patches are similar in logical structure to vendor-released patches, and thus are more likely to be integrated into the original source code base. To the best of our knowledge, *PASAN* is one of the most, if not the most comprehensive and effective automated post-attack response system for control-hijacking attacks.

In addition to local buffer overflow vulnerability, attackers often exploit heap overflow, format string, and integer overflow vulnerabilities. We plan to extend *PASAN* so that it can handle these types of vulnerabilities as well. Bounds checking is a technique used in many patches, and we are planning to extend it to other types of bounds check than array bounds check. Memory extending patches are better than bounds-checking patches because they keep all the information that a client sent instead of truncating it. Patching a large program such as Microsoft Windows typically requires testing many programs instead of just one in which a vulnerability was found because patching a dynamic library can affect multiple programs. We are going to use a virtual machine such as FVM [9] for patch testing.

## References

- [1] bugtraq. <http://www.securityfocus.com>.
- [2] A. Baratloo, T. Tsai, and N. Singh. Transparent run-time defense against stack smashing attacks. In *Proceedings of USENIX Annual Technical Conference*, June 2000.
- [3] Beyond Security's SecuriTeam. <http://www.securiteam.com>.
- [4] Bugtraq. Blackmoon ftp server buffer overflow vulnerability. <http://www.securityfocus.com/bid/3884/info>.

- [5] T-C. Chiueh and F-H. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *Proc. of 21st Intl. Conf. on Distributed Computing Systems*, 2001.
- [6] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Conference*, January 1998.
- [7] D. Dagon, X. Qin, G. Gu, W. Lee, J. Grizzard, J. Levine, and H. Owen. HoneyStat: Local worm detection using honeypots. In *Proc. of the 7th International Symposium on Recent Advances in Intrusion Detection*, 2004.
- [8] H. Etoh. GCC extensions for protecting applications from stack-smashing attacks. <http://www.tr1.ibm.com/projects/security/ssp>, June 2000.
- [9] F. Guo, Y. Yu, and T-C. Chiueh. Automated and safe vulnerability assessment. In *Proc. of Annual Computer Security Applications Conference*, 2005.
- [10] R. Jones and P. Kelly. Bounds checking for C. <http://www-ala.doc.ic.ac.uk/~phjk/BoundsChecking.html>, July 1995.
- [11] H.-A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of USENIX Security Symposium*, 2004.
- [12] C. Kreibich and J. Crowcroft. Honeycomb — creating intrusion detection signatures using honeypots. In *Proc. of the Second Workshop on Hot Topics in Networks (Hotnets II)*, 2003.
- [13] L-C. Lam and T-C. Chiueh. Automatic extraction of accurate application-specific sandboxing policy. In *Proc. of Seventh International Symposium on Recent Advances in Intrusion Detection*, 2004.
- [14] L.-C. Lam and T.-C. Chiueh. Checking array bound violation using segmentation hardware. In *Proc. of the International Conference on Dependable Systems and Networks*, 2005.
- [15] Z. Liang and R. Sekar. Automatic generation of buffer overflow attack signatures: an approach based on program behavior models. In *Proc. of Annual Computer Security Applications Conference*, 2005.
- [16] Z. Liang and R. Sekar. Fast and automated generation of attack signatures: a basis for building self-protecting systems. In *Proc. of the ACM conference on Computer and communications security*, 2005.
- [17] S. Nanda and T-C. Chiueh. Foreign code detection for Windows/X86 binaries. ECSL Technical report TR-190, Computer Science Department, Stony Brook University, 2005.
- [18] J. Newsome, B. Karp, and D. Song. Polygraph: automatically generating signatures for polymorphic worms. In *Proc. of the IEEE Symposium on Security and Privacy*, 2005.
- [19] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. of the 12th Annual Network and Distributed System Security Symposium (NDSS'05)*, 2005.
- [20] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Network and Distributed System Security Symposium*, February 2004.
- [21] S. Sidiroglou and A. D. Keromytis. Countering network worms through automatic patch generation. *IEEE Security and Privacy*, 3(6), 2005.

- [22] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a reactive immune system for software services. In *Proc. of 2005 USENIX Annual Technical Conference*, 2005.
- [23] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *Proc. of the 6th ACM/USENIX Symposium on Operating System Design and Implementation*, 2004.
- [24] A. Smirnov and T-C. Chiueh. DIRA: Automatic detection, identification, and repair of control-hijacking attacks. In *Proc. of the 12th Annual Network and Distributed System Security Symposium (NDSS'05)*, 2005.
- [25] Symantec. Symantec internet security threat report. <http://www.symantec.com/>, September 2005.
- [26] Y. Tang and S. Chen. Defending against internet worms: a signature-based approach. In *Proc. of INFOCOM 2005*, 2005.
- [27] V. Yegneswaran, J. Giffin, P. Barford, and S. Jha. An architecture for generating semantics-aware signatures. In *Proc. of the USENIX Security Symposium*, 2005.