Automatic Generation of Buffer Overflow Attack Signatures: An Approach Based on Program Behavior Models*

Zhenkai Liang and R. Sekar
Department of Computer Science,
Stony Brook University, Stony Brook, NY 11794
{zliang, sekar}@cs.sunysb.edu

Abstract

Buffer overflows have become the most common target for network-based attacks. They are also the primary mechanism used by worms and other forms of automated attacks. Although many techniques have been developed to prevent server compromises due to buffer overflows, these defenses still lead to server crashes. When attacks occur repeatedly, as is common with automated attacks, these protection mechanisms lead to repeated restarts of the victim application, rendering its service unavailable. To overcome this problem, we develop a new approach that can learn the characteristics of a particular attack, and filter out future instances of the same attack or its variants. By doing so, our approach significantly increases the availability of servers subjected to repeated attacks. The approach is fully automatic, does not require source code, and has low runtime overheads. In our experiments, it was effective against most attacks, and did not produce any false positives.

1 Introduction

In the past few years, there has been an alarming increase in *automated attacks* that are launched by worms or zombies. A key characteristic of such automated attacks is that they are *repetitive*, i.e., multiple instances of the same attack may be launched against the same victim machine in a quick succession. A vast majority of these automated attacks are due to *buffer overflows*, which account for more than three-quarters of the US CERT advisories in the last few years. Current technology for defending against buffer overflows uses some form of *guarding* [5, 7, 8] or *randomization* [1, 2, 3, 4, 14]. Although these techniques can detect attacks before system resources, such as files, are compromised, they cannot protect the victim process itself, whose integrity is compromised prior to 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 the service unavailable during periods of attack. For instance, at a relatively low rate of 10 attacks per second, services such as DNS and NTP became unavailable in our experiments. In contrast, we present an approach, called ARBOR (Adaptive Response to Buffer OveRflows), that *filters out* attacks before they compromise the integrity of a server, thereby allowing the server to continue to run without interruption. By doing so, ARBOR dramatically increases the capacity of servers to withstand repetitive attacks.

This paper builds on the core idea outlined in [17] of using program behavior models to recognize those inputs that carry buffer overflow attacks, and discarding them. As compared to the earlier technique of automated patch generation [29], as well as subsequent works such as [26, 30, 32], our approach predicts attacks at the earliest possible stage, namely, at the point of network input. This enables reliable recovery in our approach. In contrast, previous approaches recognize buffer overflow attacks close to the point of memory corruption, and cannot always recover. Another important benefit of our approach is that it generates a generalized vulnerability-oriented signature from a single attack instance, and this signature can be deployed at other sites to block attacks exploiting the same vulnerability.

1.1 Overview of Approach

ARBOR is based on the observation that attacks on network services arrive via inputs to server processes. It makes use of an off-the-shelf buffer-overflow exploit prevention technique, specifically, address-space randomization (ASR) [1, 3]. (Other techniques such as StackGuard would work as well.) ARBOR compares the characteristics of benign inputs with those of inputs received around the time of an attack, and synthesizes a signature that matches the attack input but not the benign ones. Once generated, this signature can be deployed within the victim process to filter out future instances of the same attack (or its variants). It may also be distributed to other servers using the same version of software, so that an entire community of cooperating servers may be protected from an attack, based on a single attack sample. The two main steps in our approach, namely, signature generation and recovery after discarding input, are described in more detail below.

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

I. Automatic signature generation proceeds in two steps.

1. Identifying characteristic features of attacks. Buffer overflow attacks are associated with excessively long inputs, and hence *input length* is one obvious criterion in signatures. Moreover, buffer overflow attacks are based on overwriting pointers and/or execution of attacker-provided binary code. Thus, the *presence of binary data* in inputs is a second useful criterion for signature generation.

We *do not* rely on other possible characteristics, such as data or code sequences that repeat across attacks. Although previous work on worm signature generation [15, 16, 22, 31, 33] has often relied on these characteristics, we note that polymorphic worms, as well as intelligent attackers, can easily modify these characteristics. In contrast, the length and binary data characteristics are essential features of buffer overflow attacks.

2. Using program context to improve signature accuracy. Server programs accept inputs with different characteristics in different contexts. For instance, only text data may be acceptable during the authentication phase of a protocol, while binary data may be accepted subsequently. A simple signature that is based on the presence of binary characters in input data will work correctly during authentication phase, but will subsequently cause legitimate inputs to be dropped. To increase the accuracy of signatures, we incorporate the context in which an input is processed into the signature. Without the use of these contexts, ARBOR will produce too many false positives to be useful.

II. Light-weight recovery after discarding input. After discarding input, it is necessary for the server process to take recovery actions, such as releasing resources that were set aside for processing the (attack-bearing) request, and returning control to the point where the program awaits the next service request. Rather than trying to infer the exact set of (application-specific) recovery actions, we observe that networked servers expect and handle transient network errors, which can cause their input operations to fail. ARBOR leverages this error recovery code to perform the necessary clean up actions. Specifically, whenever an input matches an attack signature, this input is dropped, and an error code signifying a network error is reported to the server.

1.2 Benefits of Our Approach

- *Effectiveness against "real-world" attacks.* We collected 11 remote buffer overflow attacks published by securityfocus.com. Since the development of exploit code is a challenging task, we considered only those attacks for which working exploit code was available on Red Hat Linux (our experimental platform). ARBOR was effective in generating signatures for 10 of these 11 attacks.
- *Preserving service availability.* Our experiments show that the availability of key servers (such as `httpd`, `ntpd` and `named`), when exposed to repeated attacks, is improved by at least an order of magnitude by ARBOR.
- *Applicable to black-box COTS software.* Our approach does not require any modifications to the protected server, or access to its source code.
- *Low runtime overheads.* ARBOR introduces low runtime overheads of under 10%.
- *High-quality signatures* generated from a *single attack sample*. These signatures are:
  - *general enough to capture attack variations that exploit the same underlying vulnerability.* Since our signatures rely on essential characteristics of buffer overflow attacks, attack variations that involve changes to exploit code or other attack details will likely be captured.
  - *specific enough to avoid matches with benign inputs.* Attack inputs were usually many times larger than benign inputs, and hence no false positives were observed in our experiments.

The ability to generate a *general* signature from a single attack sample distinguishes our approach from previous signature generation approaches [15, 16, 22, 31, 33, 40].

ARBOR signatures can be distributed over the Internet to protect other servers running the same copy of software. Such an approach can defend against fast-spreading worms. Moreover, an entire community of servers can be immunized from future instances of an attack, including servers that lack buffer overflow exploit prevention capabilities.

Note that ARBOR signatures *cannot* be deployed on a firewall (or an inline network filter), as they rely on program context information available only within the address-space of a server process. On the positive side, ARBOR is able to handle end-to-end encryption because it can intercept inputs after decryption. For instance, ARBOR can handle SSL encryption by intercepting `SSL_read`, which returns decrypted data, rather than `read`, which would return encrypted data. In contrast, a network layer filtering approach would not be able to access decrypted data.

1.3 Organization of the Paper

The rest of the paper is organized as follows. Section 2 provides a technical description of our approach. An evaluation of our approach is presented in Section 3. Related work is discussed in Section 4, followed by a summary in Section 5.

2 Approach Description

Figure 1 illustrates our approach. It is implemented using *inline* and *off-line* components. Inline components reside within the address space of the process being protected by our approach (protected process), and are optimized for performance, whereas the off-line components perform time-consuming tasks such as signature generation.

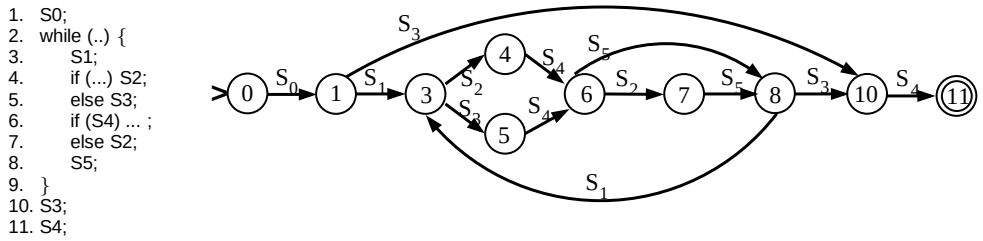

Figure 2. A sample program and its model.

The inline components "hook" themselves into the execution environment of the protected process by library interception. The primary reason for using library interception, as opposed to system call interception, is that it allows interception of a richer class of events. For instance, some server programs use buffered I/O using library functions such as `getc` and `scanf`. In this case, many calls to `getc` and `scanf` do not result in a `read` system call, as the input may be returned from a buffer within the library. An approach that relies on system call interception will consequently miss many of the input operations made by a program. A disadvantage of library interposition is that it can be bypassed after a successful attack. However, ARBOR relies only on the observations made before a successful attack, so this drawback does not impact it.

The *input filter* intercepts all input actions of the protected process. The inputs returned by these actions are then compared with the list of signatures currently deployed in the filter. Inputs matching any of these signatures are discarded, and an error code is returned to the protected process. If the input is associated with a TCP connection, then the input filter breaks the connection so as to preserve the semantics of the TCP protocol.

The *behavior model* is a central component of ARBOR. It enables our approach to leverage knowledge embedded in the program for making filtering decisions, rather than requiring manual encoding of application-specific syntax or semantics of input contents. Library interception is used to learn the behavior model of a protected process. In principle, the model can incorporate all standard C library functions. In practice, we incorporate calls to (a) all input operations, and (b) all system call wrappers.

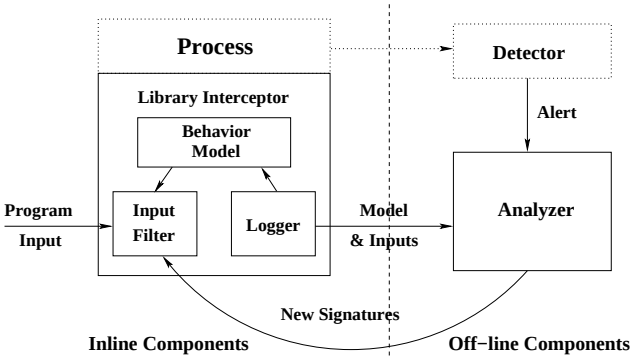

Figure 1. Architecture of ARBOR.

The *logger* records inputs for offline analysis. It also saves the entire behavior model periodically (say, every 5 minutes) to the disk, so that the model does not have to be rebuilt from scratch on process restarts. Any behavior model that is saved very close to the time of an attack is not reused. This ensures that actions associated with a successful attack do not compromise the behavior model.

The off-line components include a detector and an analyzer. The *detector* is responsible for attack detection. It promptly notifies the *analyzer*, which begins the process of generating an attack signature. The generated signature is then deployed in the input filter. This enables future instances of the attacks to be dropped before they compromise the integrity or availability of the protected process.

2.1 Behavior Model

Our approach is based on inferring program context that can be used in making filtering decisions. We employ a program behavior model to guide the search for useful program context. Many of the recent approaches for extracting automata models of programs [9, 10, 28, 35] can potentially be used for this purpose. We have used the finite-state automaton (FSA) technique of [28] due to its simplicity.

Figure 2 illustrates the FSA approach. The FSA model is very similar to a control-flow graph of a program. However, the FSA only captures security-sensitive operations (`S1` through `S5` in the figure) made by the program, while leaving out the details of its internal computation. The states in the FSA correspond to program locations (i.e., memory addresses) from which these operations are invoked, while the edges are labeled with operation names. (For readability, line numbers are used in place of memory addresses in Figure 2.) There is an edge from a state L_1 to state L_2 in the FSA labeled with the call e whenever the program invokes e from location L_2, and the previous call made by the program was from location L_1. We point out that such an FSA model can be constructed from the sequence of library calls intercepted by our system, without any access to source code. (Further details about the learning technique can be found in [28].)

2.2 Logger

The logger records information regarding intercepted operations for subsequent use by the analyzer. Following information is logged in our current implementation: the calling context for the operation, which includes the set of all callers on the runtime stack at the point of call; return code from the operation; and the values of integer-type arguments. For input operations, the fraction of binary (i.e., non-ASCII) characters in the input is also logged.

Since the logger operates within the process space of the protected server, a server crash can lead to loss or corruption of buffered log data. To protect against this possibility, the logger flushes the buffer after each input operation.

2.3 Input Filter

Signatures generated by ARBOR are deployed within the input filter. Any input that matches a deployed signature will be dropped, and an error code of -1 returned to the process. The external variable `errno` is set to `EIO` to indicate an input/output error. Since servers are built to expect network errors, they invoke appropriate recovery actions to quickly (and fully) recover from the error and proceed to process the next request.

If a server uses TCP, reporting an error to the server without notifying the client may lead to inconsistencies caused by violation of reliable message delivery semantics of TCP. To avoid this problem, the input filter closes the TCP connection on which the bad input was received. (ARBOR can determine whether a file descriptor is associated with a network connection using `fstat` and `getsockopt` calls.)

2.4 Detector

The detector monitors the execution status of the protected process. On an intrusion attempt, it raises an alert and terminates the process. Our approach uses an existing technique, address space randomization (ASR) [3], to implement the detector. With ASR, the addresses of all program objects (including code and data objects) are randomized. All buffer overflow attacks reported so far have been based on overwriting pointer values, e.g., the return address on the stack. Due to ASR, the attacker does not know the value to be used for the overwrite, as she does not know the location of any of the objects (e.g., the code injected by the attacker) in memory. As a result, attacks cause programs to crash due to invalid memory access.

Note that ASR itself needs to be deployed within the protected process. The detector component shown in Figure 1 does not denote ASR, but an external process that intercepts signals received by the protected process. In our implementation, it uses the `ptrace` mechanism in Linux. When the detector intercepts a memory access related signal (`SIGBUS`, `SIGSEGV` and `SIGILL`), it reports an attack.

Note that ASR interacts with the FSA behavior models in some ways. In particular, since the base addresses of various code segments are randomized, the absolute memory locations associated with the FSA states will change from one execution of the server to the next. To compensate for this, the FSA technique needs to decompose each memory address into a pair $(name, offset)$, where $name$ identifies the segment (e.g., the name of an executable or a shared library) and $offset$ denotes the relative distance from the base of this segment. By the nature of ASR described in [1, 3], this quantity remains invariant across all executions of an ASR-protected server.

2.5 Analyzer

The analyzer generates signatures to distinguish attack-bearing inputs from benign ones. The two main aspects of signature generation in ARBOR are discussed below.

2.5.1 Obtaining Context Information

ARBOR relies on two types of contexts: *current context* and *historical context*. The current context for an input operation captures the calling context for that operation. It helps distinguish among different input operations used by a program. For example, in Figure 2, even if `S4` and `S5` are both `read` operations, their purpose may be different, as they are invoked from different parts of the program. In our implementation, current context is defined by the program location from which the input operation is performed (which is the same as the state of the FSA model), and a sequence of return addresses (up to 20 in our implementation) on the top of the program's stack. Moreover, instead of explicitly remembering the list of all callers, we compute and use a single 32-bit hash-value from them. (Recall that in order to cope with ASR, all absolute addresses are decomposed into $(segment, offset)$ pairs before they are used.)

Historical context takes into account the FSA states that precede an input operation. The rationale for using historical context is as follows. Often, network protocols involve a sequence of steps. An attack may be based on sending an unexpected sequence of messages, where each message, in isolation, is indistinguishable (to ARBOR) from legitimate messages previously seen. Historical context enables us to utilize program context information across these steps, and hence recognize unexpected sequences of messages.

In addition to providing the ability to handle truly multi-step attacks, historical context also helps ARBOR handle some cases where the attack is really delivered in the last step, while all previous steps are legitimate. Typically, this happens due to the fact that a server program performs all its input actions from a single location, regardless of the type of request being read. This can happen with a server that uses "wait-read-process" loop structure, where the server waits for the availability of any input, and then uses a `read` call to read the entire input in one step into an internal buffer, and then uses internal code to parse the contents of this buffer

and carry out the request. Since the current context remains the same for all input operations made by such servers, all types of messages will be lumped together into a single category, thereby decreasing the likelihood of deriving a length or character distribution based signature. This problem can be mitigated using historical context. Specifically, note that even though all input actions occur from the same program location, the processing of these requests is almost sure to be carried out by different functions, or more generally, different sections of code. It is also quite likely that the processing step will involve one or more function calls that are intercepted by ARBOR, thereby allowing it to distinguish between different types of messages. Now, consider a server protocol where a message M_1 is always followed by a message of type M_2 or M_3. Although ARBOR cannot tell whether it is M_2 or M_3 at the time of reading the message, historical context seen during the processing of request M_1 enables it to avoid confusing these two types of messages with other message types. This factor, in turn, can enable signature discovery.

2.5.2 Synthesizing Signatures

Inputs received closest to the time of detection are the ones most likely to be attack-bearing. For this reason, the signature generation algorithm searches for a suspect input in the reverse temporal order among recent inputs. (ASR typically detects attacks within a millisecond timeframe, so the search can be limited to the previous 10ms for most servers.) This search is carried out in two stages. The first stage uses current context. If this fails, a historical context is used in the second stage.

In the first stage, the analyzer first identifies the current context for each recent input, and compares the input length and binary character percentage for this context with all the past inputs received in the same context. To speed up this process, the FSA model already stores the maximum input size and maximum fraction of binary characters seen among all previous benign input actions in the same context. As a result, ARBOR generates current context based signatures within 10ms.

Unlike current context, an input operation can have multiple historical contexts. Part of signature discovery is to identify the particular historical context that yields the best signature. In general, a historical context can represent a path in the FSA, but for simplicity, we have limited our current implementation to refer to just a single context that precedes an input operation by k steps, for some $k > 1$. Our technique starts with $k = 1$, and keeps incrementing k until a historical context that can distinguish benign inputs from attack input is identified, or k exceeds a certain threshold (20 in our implementation). Note that this search requires an examination of the information about previous benign inputs that was recorder by the input logger.

After an input I is identified as malicious under a context C, if its size a is significantly larger than the maximum size b_{max} of benign inputs seen so far in C, then a size-based signature is generated. Initially, the signature may specify a size threshold of $a - 1$ in order to minimize the likelihood of false positives. However, such an approach can be exploited by an attacker to send a series of attacks of successively smaller size, requiring our system to generate many signatures. To tackle this problem, the approach can be made more adaptive, e.g., by setting a threshold of $max(a - 2^k, b_{max} + 1)$ after k attack attempts. Signature generation based on percentage of binary characters is done in a similar way. The format of signatures is as follows:

```
At <function_name>@(name, offset, hash)
[Distance <dist> <function_name>@(name,
offset, hash)]
[Size <filtering_size>] [Bin% <bin_pct>]
```

"At" and "Distance" specify the program context;"Size" and "Bin%" specify the conditions characterizing an attack. We illustrate signature formats with two examples.

- **At** `read@(S1,0xBFE0,0x3A4561FE) Bin% 0`
  Meaning: if a `read` operation is invoked from the `S1` segment of the program at offset 0xBFE0 (from the base of this segment), and the set of return addresses on the stack hash to the value 0x3A4561FE, and the fraction of non-ASCII characters in the input returned by this `read` is non-zero, it needs to be dropped.
- **At** `read@(S2,0xB2FE,0xF3928621)`
  Distance 5 `time@(S2,0x2CD0,0x9823A53B)` **Size** 500
  Meaning: if a `read` operation is invoked at offset 0xB2FE in `S2` segment of the program, and the set of return addresses on the stack hash to the value 0xF3928621, and if `time` function was called from offset 0x2CD0 of the same segment five steps earlier, and the return addresses on the stack hash to the value 0x9823A53B, an input larger than 500 bytes must be dropped.

3 Evaluation

In this section, we experimentally evaluate the effectiveness of ARBOR, its runtime overheads and availability. All experiments were carried out on Red Hat Linux 7.3, except those on `lshd` which used Red Hat Linux 8.0. Finally we discuss false positives and false negatives.

3.1 Effectiveness in Signature Generation

In this evaluation, our focus was on real-world attacks. Since developing exploit programs involves significant amount of effort, we limited our selection to attacks with working exploit code available on our OS platform, Red Hat Linux. We selected eleven such programs shown in Figure 3. Six of them were chosen because they were

Program	Vulnerability	Effective?	Attack Length	Max Benign Input size			Attack to Benign Size Ratio	Attack to Benign BIN% Ratio
				All Contexts	Current Context	Historical Context		
wu-ftpd	CVE-2000-0573	Yes	473	8192	55	N/A	8.6	∞
apache ssl	CAN-2002-0656	Yes	419	815	0	N/A	∞	1.0
ntpd	CVE-2001-0414	Yes	500	1024	48	N/A	10.4	1.0
ircd	CAN-2003-0864	Yes	490	8191	258	N/A	1.9	∞
lshd	CAN-2003-0826	Yes	5025	1024	376	N/A	13.4	1.0
gtkftpd	BugTraq ID 8486	Yes	260	4096	195	N/A	1.3	∞
samba	CAN-2003-0201	Yes	2080	4144	4144	0	∞	1.0
epic4	CAN-2003-0328	Yes	1024	3477	3477	0	∞	∞
cvs	CAN-2004-0396	Yes	1024	1024	1024	1024	1	∞
passlogd	BugTraq ID 7261	Yes	916	1049	1049	1049	0.9	4.0
oops	CAN-2001-0029	No	1392	2048	2048	2048	0.7	1.0

Figure 3. Effectiveness of our approach in signature generation.

widely used programs, and as a result, would have had obvious bugs fixed, thereby providing us with more sophisticated attacks. These include the `wu-ftpd` FTP server, `apache` web server, `ntpd` network time protocol server, `ircd` Internet relay chat server, `samba` server that supports Windows-compatible file and print sharing, and `CVS` server used for source-code versioning. Of the remaining programs, `passlogd` (a passive `syslog` capture daemon) was chosen because it had a message subfield overflow that did not increase overall message length, thereby posing a problem for length-based signature detection. `oops` (a freeware web proxy server) was chosen because it represents perhaps the hardest example for ARBOR, providing no useful current or historical context information. Other examples were moderately popular programs, including `gtkftpd`, a FTP server with a Gtk-based GUI, `lshd`, the GNU secure shell server, and `epic4`, a popular Internet relay chat client.

The examples were also chosen to exercise different types of memory errors, including stack overflow, heap overflow, and format string bugs.

Figure 3 shows the results obtained with these programs, organized into four groups according to the nature of signatures generated. In the first group, current context was enough to generate effective length-based signatures. Although some of the programs receive inputs larger than the attack-bearing input, the corresponding contexts were different. The second group consists of `samba` and `epic4`, both of which read their inputs from a single location. This means that the current context remains the same for all message types. Since some of the messages, by their nature, are very long, ARBOR could not generate a length-based signature. However, since both attacks use a sequence of messages, signatures can be generated using historical context.

In the third group, both current context and historical context did not help to synthesize a length-based signature. In the case of `passlogd`, there was only one message type, so historical context was not applicable. Moreover, the attack involved an overflow in a subfield of the message, so the overall length was still within the size of benign requests. A similar situation applied in the case of `CVS` as well. However, both these attacks were characterized by a large fraction of non-ASCII characters, whereas benign inputs consisted of mostly ASCII characters. Hence signatures based on character distributions were generated.

The last group consists of `oops` which is a proxy web server. By its nature, it simply passes on its requests to an external web server. As a result, it reads its input requests from the same program location. Moreover, its input requests are independent of each other. As a result, no useful current or historical context was available. As a result, ARBOR failed to generate a signature.

From these results, we can see that program context is very important for generating accurate signatures. Without context information, length-based signatures can be generated for less than 10% of the attacks. This increases to 55% and 72% with current and historical context. Using both contexts and length as well as character distribution criteria, successful signatures are generated for 91% of attacks.

3.2 Evaluation of Runtime Overhead

Since analysis is an offline process, we have not tuned the signature generator for performance. For this reason, we did not study its performance in our experiments.

The runtime overhead due to inline components was 7% for a CPU-intensive benchmark (compilation of Openssh version 3.8.1p1), and 10% for an Apache server.

A 7% to 10% overhead is modest, and it can be further

Program	Partial Logging	Full Logging
Compilation	$< 5\%$	7%
httpd	$< 5\%$	10%

Figure 4. Performance overheads.

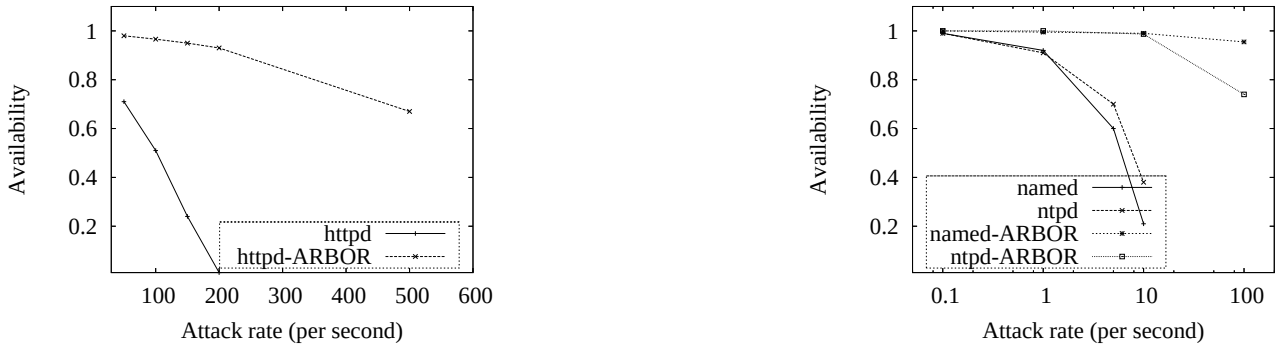

Figure 5. Availability Degradation under Repetitive Attacks

improved by logging only a fraction of the operations under normal conditions, and switching to full logging during periods of attacks. For instance, if only 10% of the program operations were logged during normal operation, this brings the overheads to below 5%. With partial logging, logging is turned on for a period of time (say 100 milliseconds) and then turned off for a period (say, 900 milliseconds). The potential downside to partial logging is that when the first attack occurs, the associated input data may not have been logged. But this can be corrected right away, as the logger can be reconfigured to perform full logging after the first attack. Thus, the only effect will be that of a slight delay in signature generation. Note that the behavior model is always updated, so partial logging has no effect on the model.

3.3 Improvement in Server Availability

Figure 5 compares the availability of three key servers in the face of repetitive buffer overflow attacks: the Apache web server (`httpd`), the domain name server (`named`), and the network time server (`ntpd`). The availability at a given attack rate was measured as the ratio of server throughput at that attack rate, expressed as a fraction of the server throughput under no attacks. In all experiments, attacks were carried out by one or more clients, while the server was accessed in a legitimate fashion by another client. For servers protected by our approach, the input filter dropped requests and reported an error to the server. For an unprotected server, the server would crash after processing input from an attacker. The server was restarted automatically after a crash. In the case of `httpd`, normal request accesses were simulated using WebStone. For other servers, we wrote scripts to make repeated requests to the server.

In the absence of our protection, `ntpd` and `named` need to be restarted after each attack, which is quite expensive. As a result, our approach achieved about a factor of 10 to 100 improvement in their ability to withstand repetitive attacks, i.e., for a given value of server availability, protected servers can withstand attacks at rates that are about 10 to 100 times higher than that of unprotected servers. In the case of `httpd`, the Apache web server uses multiple processes to serve requests, and attacks cause one of the "worker processes" to die, not the main server. This means that attacks do not require a server restart, but only that a new process be created to replace the process that crashed due to the attack. So the normal recovery process is more efficient than `ntpd` and `named`. As a result, the availability improvement due to ARBOR was closer to 10 than 100.

3.4 False Positives

We did not encounter false positives in our experiments, as our approach generates signatures only when the attack input size exceeds all previously encountered benign input sizes in a given context. The column "Attack to Benign Size Ratio" in Figure 3 shows that there is a significant difference between benign and attack input sizes, thus providing a safety factor against false positives. It can also be seen that for many programs, the BIN% ratio is ∞, once again providing a margin of safety from false alarms. To further reduce the possibility of false positives, we can combine length and character distribution into a single signature.

For `samba` and `epic4`, the maximum size of 0 indicates that the corresponding historical context was never witnessed in the presence of benign requests. Similarly, for `apache`, the context corresponding to the attack was never witnessed with benign requests. This is not reassuring from a false positive stand-point, as there is a possibility that this is due to insufficient diversity among the clients we used. Further analysis on `apache` revealed that the contexts corresponding to the legitimate and attack inputs were almost the same — in fact, the difference was in a calling function that appeared 15 frames higher in the call stack. If we redefined "context" to use only the top 15 return addresses, then the maximum benign request size increases to 138, which gives us more confidence with respect to false positives.

We are currently investigating two ways to provide increased assurances regarding false positives. The first way is to use an adaptive definition of current context that varies the number of return addresses used. The second way is to derive a confidence metric for the signature based on the number of benign samples seen in any given context.

3.5 False Negatives

In this section, we analyze several scenarios where signature generation may be expected to fail.

Attacks delivered through multiple packets. If an attack is fragmented into multiple packets, then it may be necessary for a server to perform multiple input operations to read the attack input. Each input operation may return a small amount of data, and hence fall below any size threshold used in an attack signature. To address this limitation, we observe that typically, a server will perform such read operations in a loop until the complete request is received. As a result, all these input operations are made from the same calling context, and there are no other input operations in between. Our approach currently concatenates the results of such a sequence of input operations, and is hence able to deal with such fragmented attacks. However, it is possible that some servers may read fragmented requests from different parts of the program. In this case a more sophisticated approach for assembling inputs will be needed.

Concurrent Servers. With concurrent servers, it is possible that operations associated with processing different requests may be confused, which can be expected to make it difficult to synthesize accurate signatures. However, we observe that ARBOR already incorporates a search for identifying the attack-bearing inputs from recent inputs. Concurrency simply increases the number of recent requests that need to be considered in the search, and hence does not unduly increase false negatives. Indeed, many of the attacks in our experiments involved concurrent servers.

Message field overflows. Some attacks are characterized by the fact that the input message is well within the maximum limits, but subfields of the message are not. Such attacks can pose problems in some cases, but not in others. If a server reads different message fields from different program locations, then a signature can still be generated. This behavior is common in text-based protocols that make use of hand-written parsing code. For instance, `sendmail` uses repeated calls to `getc` to read its input, and uses conditionals and loops for parsing. Other servers may perform a block read into a buffer, and then subsequently process the data contained in the buffer. In such cases, a signature may still be generated based on the presence of non-ASCII characters, as was done in the case of `passlogd`. However, if the protocol involved is a binary protocol, then this approach would fail as well.

DoS attacks aimed at evading character distribution signatures. A typical buffer overflow attack contains binary characters to represent pointer values and executable code. An attacker can replace these characters with ASCII characters chosen to preserve the character distribution of benign inputs. In this case, a character distribution based signature would fail. The attack would not have the effect of injected code execution, but will still cause the victim process to crash. Thus, if the attacker's goal is simply DoS, then such a strategy would successfully evade our signatures. For this reason, we prefer length-based signatures in ARBOR.

Addressing limitations. Motivated by the above difficulties faced by ARBOR, we have recently developed COVERS [19], a complementary approach for signature generation. To address the fragmentation problem, it aggregates inputs read from multiple program locations into a single session. To address the concurrency problem, it uses a technique to correlate the effects of attacks back to specific inputs. Finally, to handle message field overflows, it relies on a manual specification of message formats. The principal drawback of COVERS is this need for manual involvement. In contrast, ARBOR accepts false negatives in some cases to achieve fully automatic signature generation.

4 Related Work

The key ideas behind this paper were first sketched in [17]. Preliminary experimental results, together with a high level exposition of the approach, were presented in [18]. Due to length limitations, [18] does not provide a technical description of the approach, or a detailed experimental evaluation, both of which are included in this full-length paper.

Detection of Memory Errors and/or Exploits [5, 7, 8] describe techniques for preventing stack-smashing attacks. Techniques such as address-space randomization [1, 3, 4] provide broader protection from memory error exploits. Instruction set randomization [2, 14] (and OS features such as non-executable data segments) prevents foreign code injection attacks. Techniques such as [12, 13, 21, 27, 39] provide comprehensive detection of all memory errors, whether or not they are used in an attack . With all these approaches, a victim process is terminated when a memory error (or its exploitation) is detected, thereby leading to loss of server availability during periods of intense attacks.

Approaches for Recovering from Memory Errors *Automatic patch generation* (APG) [29] proposed an interesting approach that uses source-code instrumentation to diagnose a memory error, and automatically generate a patch to correct it. STEM [30] improved on APG by eliminating the need for source code access, and instead using machine-code emulation. Both approach force an error return on the current function when an attack is detected. 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 contrast, our approach forces error returns for input functions, where server applications expect and handle errors. Therefore, recovery is more reliable in our approach.

Failure-oblivious computing [26] uses CRED [27] to detect all memory errors at runtime. When an out-of-bounds write is detected, the corresponding data is stored in a separate section of memory. A subsequent out-of-bound read will return this data. This approach makes attacks harm-

less, and allows for recovery as well. The main drawback of this approach is that it typically slows down programs by a factor of 2 or more.

DIRA [32] uses a source-code transformation for runtime logging of memory updates. When an attack is detected, all the updates made since the last network input operation are undone, and the process restarted at this point. However, their approach limits logging to global variable updates for performance reasons. This limits light-weight recovery, requiring a total application restart in some cases.

Xu et al. [38] developed an approach for diagnosing memory error exploits and signature generation. Their approach uses a post-crash forensic analysis of address-space randomized programs. Their signature consists of the first three bytes of jump address included in a buffer overflow attack. To minimize false positives, they suggest the use of program contexts (specifically, current context), an idea we had described in [18].

As compared to the above approaches, ARBOR has the benefit that it generates *vulnerability-oriented* signatures, as opposed to exploit-specific signatures that can miss attack variants that exploit the same vulnerability. Moreover, it is fully automatic, works on black-box COTS software, has low runtime overheads, and recovers quickly and reliably from attacks.

COVERS [19] presents a technique that complements ARBOR — it can generate robust signatures that can be deployed in the network, and can deal with message subfield overflows in a more robust fashion. However, this is achieved at the cost of requiring manual effort in specifying message formats, whereas ARBOR is fully automatic.

Network-level Detection of Buffer Overflows Buttercup [24] and [11] detect buffer-overflow attacks in network packets by recognizing jump addresses within network packets. Buttercup requires these addresses to be externally specified, while [11] detects them automatically, by leveraging the nature of stack-smashing attacks and the memory layout used in Linux. [34] suggested a more robust approach for detecting buffer overflow attacks using abstract execution of the attack payload. PayL [37] 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 [36] uses manually generated signatures to filter out buffer overflows as well as other attacks.

Network Signature Generation Earlybird [31] and Autograph [15], two of the earliest approaches for worm detection, relied on characteristics of worms to classify network packets as benign or attack-bearing. Honeycomb [16] avoids the classification step by using a honeynet, which only receives attack traffic. The signatures generated by all three techniques rely on the longest byte sequence that repeats across all attack packets. A polymorphic (and metamorphic) attack can change its code as it propagates, which can cause these signature generation techniques to fail. To mitigate this problem, Polygraph [22] can generate multiple (shorter) byte-sequences as signatures. Nemean [40] improves on the above approaches by incorporating protocol semantics into the signature generation algorithm. By doing so, it is able to handle a broader class of attacks than previous signature generation approaches that were primarily focused on worms.

The above techniques operate at the network level, while our approach works at the host level. This means that our approach is able to exploit the internal state of server processes (e.g., current or historical context) to generate more robust signatures. More importantly, our approach is able to generate a general vulnerability-oriented signature from a *single attack sample*, whereas previous approaches require multiple attack samples to synthesize a generalized signature. Indeed, the generality of the signature provided by previous approaches is largely determined by the attack samples available.

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, but its effectiveness for a broader class of attacks was not evaluated.

TaintCheck [23] and Vigilante [6] track the flow of information from network inputs to data used in attacks, e.g., a jump address used in a code-injection attack. The signatures generated by TaintCheck are somewhat simplistic — it uses the 3 leading bytes of a jump address as a signature, which can lead to false positives, especially with binary protocols. Vigilante's signatures consist of machine code derived from the victim program's code. These signatures do not produce false positives, but can be large and overly specific. They suggested some heuristics for generalizing them, but these heuristics were not well evaluated.

FLIPS [20] uses PayL [37] 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 is generated using techniques similar to network signature generation techniques.

An advantage of ARBOR is our use of a relatively simple infrastructure that is based on library interposition. In contrast, TaintCheck, Vigilante and FLIPS rely on relatively complex infrastructures for runtime instruction emulation or binary transformations.

5 Summary

Our approach solves two key problems encountered in automatic filtering of attacks. First, it automatically discovers the signatures that distinguish attack-bearing data from

normal data. These signatures are synthesized by carefully observing both the input data and the internal behavior of a protected process. Second, it automatically invokes the necessary recovery actions. Instead of simply discarding data, a transient network error is simulated so that the application's own recovery code can be utilized to safely recover from a foiled attack attempt. Our approach can work with COTS software without access to source code.

ARBOR was effective in generating a signature for 10 of the 11 "real world" attacks used in our experiments, thus demonstrating its effectiveness in blocking most buffer overflow attacks. Moreover, false positives were not observed in these experiments.

Although ARBOR is currently a stand-alone system, it can be extended with the ability to communicate with other systems, allowing it to send generated attack signatures and attack payloads to system administrators and other systems protected by our approach, so that these systems can block out recurrences of the same attack without ever having witnessed even a single attack instance.

We believe that the central idea of using program context information to refine input classification has applicability beyond the class of buffer overflow attacks, and is a topic of our ongoing research.

References

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

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

[3] 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.

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

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

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

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

[8] H. Etoh and K. Yoda. Protecting from stack-smashing attacks. Published on World-Wide Web at URL http://www.trl.ibm.com/projects/security/ssp, 2000.

[9] H. Feng et al. Anomaly detection using call stack information. In *IEEE S&P*, 2003.

[10] J. Giffin, S. Jha, and B. Miller. Efficient context-sensitive intrusion detection. In *NDSS*, 2004.

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

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

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

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

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

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

[17] Z. Liang, R. Sekar, and D. DuVarney. Immunizing servers from buffer-overflow attacks. Presentation in *ARCS Workshop*, 2004.

[18] Z. Liang, R. Sekar, and D. DuVarney. Automatic synthesis of filters to discard buffer overflow attacks: A step towards realizing self-healing systems. In *USENIX Annual Technical Conference*, (*Short Paper)* 2005.

[19] Z. Liang and R. Sekar. Fast and automated generation of attack signatures: A basis for building self-protecting servers. In *CCS*, 2005.

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

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

[22] J. Newsome et al. Polygraph: Automatically generating signatures for polymorphic worms. In *IEEE S&P*, 2005.

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

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

[25] J. Reynolds et al. On-line intrusion detection and attack prevention using diversity, generate-and-test, and generalization. *Hawaii Intl. Conference on System Sciences*, 2003.

[26] M. Rinard et al. 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 et al. A fast automaton-based method for detecting anomalous program behaviors. In *IEEE S&P*, 2001.

[29] S. Sidiroglou and A. Keromytis. A network worm vaccine architecture. In *WETICE*, 2003.

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

[31] S. Singh et al. Automated worm fingerprinting. In *OSDI*, 2004.

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

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

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

[35] D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE S&P*, 2001.

[36] H. Wang et al. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *SIGCOMM*, 2004.

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

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

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

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