© ACM, 2006. This is the authors' version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version is available at http://doi.acm.org/10.1145/1180405.1180412.

Packet Vaccine: Black-box Exploit Detection and Signature Generation

XiaoFeng Wang, Zhuowei Li
Indiana University
{xw7,zholi}@indiana.edu

Jun Xu
Google Inc. & NCSU
jxu3@unity.ncsu.edu

Michael K. Reiter
Carnegie Mellon University
reiter@cmu.edu

Chongkyung Kil
North Carolina State University
ckil@ncsu.edu

Jong Youl Choi
Indiana University
jychoi@indiana.edu

ABSTRACT

In biology, a *vaccine* is a weakened strain of a virus or bacterium that is intentionally injected into the body for the purpose of stimulating antibody production. Inspired by this idea, we propose a *packet vaccine* mechanism that randomizes address-like strings in packet payloads to carry out fast exploit detection, vulnerability diagnosis and signature generation. An exploit with a randomized jump address behaves like a vaccine: it will likely cause an exception in a vulnerable program's process when attempting to hijack the control flow, and thereby expose itself. Taking that exploit as a template, our signature generator creates a set of new vaccines to probe the program, in an attempt to uncover the necessary conditions for the exploit to happen. A signature is built upon these conditions to shield the underlying vulnerability from further attacks. In this way, packet vaccine detects and filters exploits in a black-box fashion, i.e., avoiding the expense of tracking the program's execution flow. We present the design of the packet vaccine mechanism and an example of its application. We also describe our proof-of-concept implementation and the evaluation of our technique using real exploits.

Categories and Subject Descriptors: K.6.5 [Security and Protection]: Invasive software, Unauthorized access

General Terms: Security

Keywords: Black-Box Defense, Exploit Detection, Signature Generation, Worm, Vaccine Injection

1. INTRODUCTION

In biology, a *vaccine* is a living, weakened strain of a virus or bacterium that is intentionally injected into the body for the purpose of stimulating antibody production. That strain is weakened so as to prevent it from causing infection. Similarly, a "weakened" exploit packet with important elements of its payload scrambled would quickly expose itself through the exception it causes in a vulnerable program. Forensic analysis of the exception could uncover the related program vulnerability and enable the generation of an "immunity", a signature for capturing future exploits on the same vulnerability.

The above intuition can be applied to exploit detection, vulnerability diagnosis and automatic signature generation. Design of such mechanisms has been impeded by the constraints of commodity software, for which access to source or binary recompilation is often prohibited. Existing approaches [23, 7, 5] have suggested tracking the input data as the program executes until the point at which control-flow hijacking happens. We call these approaches *gray-box analysis*, as they do not need source code (as a *white-box* approach would) but do have to monitor a program's execution flow closely (a *black-box* approach would not). Gray-box analysis is accurate and applicable to commodity software. However, it incurs significant runtime overheads, often slowing the system by an order of magnitude.

Inspired by the principle of vaccination, we develop a much faster *black-box* approach. Rather than using expensive dataflow tracking, it detects and analyzes an exploit using the outputs of a vulnerable program. Specifically, we first identify anomalous tokens in packet payloads, e.g., byte strings resembling injected jump addresses in a control-flow hijacking attack, and randomize the contents of these tokens to generate a *vaccine*. If the packets carrying these tokens indeed contain an exploit, the vaccine will likely cause an exception in the vulnerable software. When this happens, our approach will automatically generate a signature to protect the software using the forensic data gleaned from the exception and fault injection techniques [18]. We call this approach *packet vaccine*.

Compared to other techniques, packet vaccine offers some important benefits:

Fast, black-box exploit detection. Packet vaccine detects an exploit attempt by directly injecting vaccine packets into a program. Therefore, it performs as fast as a normal run of that program, and up to an order of magnitude faster than gray-box approaches. In addition, packet vaccine does not use source code or recompiled binaries and thereby works well with commodity software.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CCS'06, October 30–November 3, 2006, Alexandria, Virginia, USA.
Copyright 2006 ACM 1-59593-518-5/06/0010 ...$5.00.

Effective signature generation. Packet vaccine generates signatures using host information, so it is immune to interference from Internet noise [28] and poisoning [25], which can mislead network-based signature generators (e.g., Early Bird [30], Polygraph [22], Nemean [41]) into generating false signatures. Moreover, the resulting signature tends to capture some key properties of a vulnerability such as the size of a vulnerable buffer, which can be used to detect a range of exploit mutations employed by polymorphic worms.

Using a confirmed exploit as a template, packet vaccine can generate a number of vaccines, i.e., variations of that exploit, to gain a better characterization of a software application's vulnerability. For instance, one type of our signatures uses a packet's field length as an attribute to identify a buffer-overflow attack; injection of vaccines with different field lengths allows us to accurately estimate the size of the underlying vulnerable buffer and thereby generate a more accurate signature (Section 2.3). Moreover, our technique can generate a signature without any information about an application or its protocol.

Some gray-box approaches perform static analysis [3, 21] over a vulnerable program's binary code and could generate signatures more accurate than our signatures. However, our black-box approach tends to be faster than those approaches and even works with obfuscated code [37, 19]. For many exploits, our black-box technique can produce signatures close to their signatures in quality, as we report in our experimental study. We argue that a rapidly-generated and reasonably accurate signature could be more useful in practice because such a signature is supposed to serve as a band-aid to a vulnerable application rather than a permanent fix [20], for use before a software manufacturer finishes developing its patch.

Low overhead and easy deployment. Packet vaccine is more lightweight and easier to deploy than many existing techniques. Exploit detection using our approach does not require installing *anything* on the host running vulnerable programs. Vulnerability diagnosis needs only a lightweight collector to gather forensic data from an exception, and even this requirement can be waived for operating systems which already offer error logging and debugging services. For example, Windows XP's event logs contain everything we need, such as corrupted pointer contents.

We present the design of the packet vaccine mechanism (Section 2) and the implementation of this technique in the paper. We evaluate it using real exploits and signatures generated by a gray-box approach (Section 3). Our study shows that packet vaccine can effectively detect exploits, and efficiently generate signatures of high quality. A problem of a vaccine is that it could modify a server's state, and interrupt its service. To apply this technique to protect an online service, we present an architecture which employs test servers to carry out exploit detection, and empirically evaluate its performance with a proof-of-concept implementation (Section 4). We also discuss the limitations of our approach (Section 5) and review related work (Section 6).

2. DESIGN

In this section, we present the design of the packet vaccine mechanism. Figure 1 illustrates the major steps of our approach: vaccine generation, exploit detection, vulnerability diagnosis and signature generation.

Vaccine generation is based upon detection of anomalous packet payloads, e.g., a byte sequence resembling a jump address, and randomization of selected contents. A vaccine generated in this way can detect an exploit attempt, since it should now trigger an exception in a vulnerable program. Vulnerability diagnosis correlates the exception with the vaccine to acquire information regarding the exploit, in particular the corrupted pointer content and its location in the exploit packet. Using this information, the signature generation engine creates variations of the original exploit to probe the vulnerable program, in an effort to identify necessary exploit conditions for generation of a signature.

2.1 Vaccine Generation

To generate a vaccine, we need to preserve the exploit semantics—i.e., its behavior that leads to an attempt to hijack control flow—while weakening it enough to prevent a control-flow hijacking from succeeding. Here, we describe a simple way to do that.

A key step in most exploits is to inject a jump address to redirect the control flow of a vulnerable program. Such an address points to somewhere in the stack or heap in a code-injection attack, or to a global library entry in an existing-code attack. Our approach is to check every 4-byte sequence (32-bit system) or 8-byte sequence (64-bit system) in a packet's application payload, and then randomize those which fall in the address range of the potential jump targets in a protected program. The vaccine generated in this way should cause an exception, segmentation fault (SEGV) or illegal instruction fault (ILL), to a vulnerable program's process if an exploit is indeed present in the original packet. A question here is how to determine the address range.

Address Range. A process's virtual memory layout is usually easy to obtain. On Linux and UNIX, the `proc` virtual filesystem maintains a file called `maps` under the directory `/proc/pid/` that offers the runtime memory layout for the process `pid`. From that file, we can obtain the base addresses for the stack (usually from 0xc00000000 downwards) and the entry for function libraries (in segment 0x40000000). The base address for heap is the end of the BSS segment, which can be determined by analyzing the binary executable using tools such as `objdump` or `readelf`. To find out the address range, we also need to know an application's stack and heap sizes. These can be estimated by monitoring stack and heap usage recorded in the `status` file of the application's process for a period of time. Using these data, we determine the address ranges as follows. Let b_s and u_s be the stack's base address and typical maximum usage, respectively. Stack addresses are estimated to range from $b_s - \alpha u_s$ to b_s, where $\alpha \geq 1$ is a ratio for keeping a safe margin. Similarly, the heap range is approximated as b_h to $b_h + \alpha u_h$, where b_h and u_h are the heap's base and typical maximum usage, respectively.[1] Address ranges can also be customized by the user. For example, one could restrict monitoring to the heap on an operating system with a nonexecutable stack.

[1] A process may have multiple heap regions, which can be observed from its memory maps. In this case, we can use the base addresses of these regions plus αu_h to estimate multiple heap address ranges.

Control-flow Hijacking (e.g., worms)
Suspicious Jump Addresses
Vaccine Generation
Vaccine Packets
Exploit Detection
an end host
Exceptions
Forensic Info
Vulnerability Diagnosis
Signature Generation
Immunities
After installing immunities
Correlations

Figure 1: The design of packet vaccine.

We can pinpoint the address range of the global libraries intensively used by exploits, e.g., `msvcrt.dll` or `libc.so`, and even the entry addresses of some "dangerous" functions, such as `system()` and `execve()`. These addresses can be easily acquired on Linux or UNIX using the `maps` file and the command `nm`. A Windows application's memory information can be collected using memory monitoring tools like Memview [16] or debugging tools such as CDB or NTSD [34]. The address range could also cover the global offset table (GOT), though this might not be necessary: an exploit usually changes a function pointer in the GOT to an address in the stack or heap, where the attack code lies. Again, it is at the user's discretion to decide the coverage of the address range. The larger the range becomes, the more packets must be checked and randomized.

Address ranges can also be approximated through an empirical study of known exploits, which could reveal 'hotspots' to which most exploits jump. In our research, we collected around 1000 jump addresses from known exploits and discovered that on Linux, most code-injection attacks use the jump addresses either in the range `0xbfff0000` to `0xbfffffff` for the stack or `0x08040000` to `0x08ffffff` for the heap. This treatment also works for existing-code attacks, as most of these exploits use a small set of libc (Linux or UNIX) or dll (Windows) functions as stepping stones.

Vaccine Generation Algorithm. Now we are ready to present the vaccine generation algorithm, which is formally described as follows.

- Gather data from the application being protected and build a *target address set* $T = [b_s - \alpha u_s, b_s] \cup [b_h, b_h + \alpha u_h] \cup S$, where S is a set containing the address ranges of objects other than the stack and heap, such as the entries for global library functions.
- Aggregate the application payloads of the packets in one session into a dataflow, carry out a proper decoding (e.g., Unicode decoding, URL decoding, etc.) if necessary and scan that dataflow to find all byte sequences $\tau \in T$.
- For every τ, replace its most significant byte with a byte randomly drawn from a scrambler set R to output a new dataflow.
- Construct vaccine packets using the new dataflow as application payloads.

In the above algorithm, the scrambler set R could be set to avoid introducing undesired symbols (such as syntax tokens) which could interrupt a protocol, and ensure a randomized byte sequence falls *outside* a process's memory map. An example of R is {`A` to `Z`, `a` to `z`, `0` to `9`, '+' and '-'}.

For example, the payload of the Code Red II worm is presented in Figure 2. Our vaccine generator identifies multiple occurrences of the byte sequence `0x7801cbd3` from the payload after Unicode decoding. This sequence falls in the address range of `msvcrt.dll`, which is being monitored. Therefore, a vaccine is generated as illustrated in Figure 2,

The Orignal Packet of Code Red:
```
GET /default.ida?NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNN%u9090%u6858%ucbd3%u7801%u909
0%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u7801%u9090%u9090%u8190
%u00c3%u0003%u8b00%u531b%u53ff%u0078%u0000%u00=a HTTP/1.0\r\n
```

A Vaccine Packet for Code Red:
```
GET /default.ida?NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNN%u9090%u6858%ucbd3%ua001%u9090
%u6858%ucbd3%u0401%u9090%u6858%ucbd3%u8c01%u9090%u9090%u8190%
u00c3%u0003%u8b00%u531b%u53ff%u0078%u0000%u00=a HTTP/1.0\r\n
```

Figure 2: A vaccine generated from Code Red II worm.

in which the most significant bytes of the sequence have been scrambled.

Discussion. A central question here is whether the vaccine generated above is effective in detecting an exploit if it is indeed present. Exploits tend to be fragile—a random perturbation could cause them to vanish. For example, randomization of protocol syntax tokens, such as the keyword 'GET' in the above example, renders the vaccine impossible to parse; modification of other exploit tokens can modify the exploit semantics, i.e., interfere with the exploit's attempt to hijack control flow. We address these concerns as follows.

Our approach is very unlikely to modify a protocol's syntax tokens, which usually look quite different from a suspicious jump address. We checked the most frequently used syntax tokens in HTTP, FTP and SMTP, and found none of them coincide with a typical Linux stack segment (`0xbfff`) and heap segment (`0x08`). To make the break of protocol syntax even less likely to happen, we can use a *whitelist* to guide vaccine generation. The whitelist contains all syntax tokens of a protocol, which can be either collected from the protocol's RFC or extracted from users' normal traffic. In our research, we were able to extract all important HTTP syntax tokens from one million HTTP traces. When generating vaccines, the generator checks a byte sequence τ against that whitelist. If it contains a syntax token, or it is a substring of such a token, the generator will refrain from scrambling it.

Our approach can also preserve exploit semantics in most cases. Exploits typically provide certain protocol parameters in the payload, in order to drive the target program's state to a "breakpoint" where exploit payload can be injected [3, 7]. Theoretically, it is possible for these parameters to coincide with addresses in T. However, this seems to be rare in practice, especially for protocols with an uneven distribution of byte values (e.g., text-based protocols such as HTTP). The appearance of an address-like string is uncommon for these protocols, as discovered in previous research [24, 39]. Furthermore, although binary protocols such as DNS could have an even distribution of byte values, the set T is usually small, occupying less than 0.1% of the virtual memory address space, and an exploit's parameters

(except the injected code) are usually short, less than tens of bytes as we observed in our experiments. Therefore, it seems that the chance a byte sequence in T coincides with a necessary exploit parameter is small. In our research, we carefully studied 26 exploits, including attacks through binary protocols, and found none of their parameters were tampered with by our approach. In addition, those parameters are mostly dependent on the underlying vulnerability, which could leave an attacker little room to vary them.

Our randomization strategy also helps preserve exploit semantics: instead of scrambling the whole byte sequence, we only modify *one* byte—the most significant byte. We could extend the idea, for example, by generating three vaccines, each of which scrambles one of the three most significant bytes of the sequence. These vaccines can then used to probe an application in parallel. As a result, even if an exploit does use an address-like two-byte parameter (such as `0xbfff`), we can still detect the exploit. Another approach involves a simple network anomaly detector (NAD) which narrows the search for address-like substrings to only part of an anomalous packet's payload. For example, a NAD monitoring the length of packets' application fields may identify an overlong CGI parameter; this allows a vaccine generator to scan only that field, avoiding randomizing other parameters even if they look like addresses. We can also whitelist well-known exploit tokens such as `%n`, and tokens present in normal traffic such as `.ida?`. All of these will then be kept intact during vaccine generation.

2.2 Exploit Detection and Vulnerability Diagnosis

Exploit attempts from vaccine packets are detected from the exceptions they cause in a vulnerable program, such as SEGV and ILL. Such exceptions happen with high probability if exploits' jump addresses have been scrambled.

The objective of vulnerability diagnosis is to reliably correlate an exception with one of the byte sequences being randomized, which identifies the location of the jump address on an exploit packet. This correlation is established by matching these byte sequences to the forensic data gathered from an exception, in which the corrupted pointer is of particular importance. On x86 systems, the corrupted pointer which causes a SEGV exception can be found in register CR2. It may also appear in EIP. Our approach logs the contents of these registers once an exception happens.

Formally, vulnerability diagnosis works as follows. Let $\tau_1, \tau_2, \ldots, \tau_n$ be the byte sequences (tokens) of a vaccine packet that have been scrambled (i.e., the high-order byte randomized) by the vaccine generator. Let p be the forensic string—the corrupted pointer collected from registers. If $p = \tau_i$ for $1 \le i \le n$, we correlate τ_i with the exception. This correlation can be validated using the following test: we randomize *all* bytes of τ_i to produce a new token τ and use it to generate a new vaccine; sending this vaccine to the vulnerable program, we check whether the exception happens again and the corrupted pointer also changes to τ. The validation test can be repeated to increase the confidence in the correlation.

2.3 Signature Generation

After vulnerability diagnosis, we have identified the jump address and its location in an exploit packet. The address alone, however, could be too general to be a signature, especially for binary protocols such as DNS. More information is required to form a high-quality signature. Here, we describe a *signature generation engine* that uses a known exploit as a template to generate vaccines and injects them into a vulnerable program to acquire key attributes of the underlying vulnerability. We call this technique *vaccine-injection* (VI). Our approach can generate signatures with or without application-specific information, as we elaborate below.

Application-independent Signature Generation. We can generate a signature without any knowledge about an application's protocol. Such a signature is in the form of a *token sequence*, which consists of an ordered sequence of byte strings (tokens) [22]. These tokens' locations in the exploit packet's payload could also be included as a part of the signature for a binary application protocol such as DNS. Our idea is to determine the roles played by individual bytes in an exploit by scrambling them to create vaccines and testing them in the vulnerable application, in an effort to identify the inputs necessary for the exploit to occur.

Let L be the byte length of an application-level exploit dataflow, and $B[i]$ be the ith byte on that dataflow, where $1 \le i \le L$. Suppose the scrambled jump address τ with a byte length l starts from the rth byte. The signature generation engine generates $L-l$ vaccines, $\{v_1, v_2, \ldots, v_{r-1}, v_{r+l}, \ldots, v_L\}$, such that v_i $(1 \le i \le L)$ randomizes the ith byte of the exploit payload and also keeps the token τ. Then, it injects all these vaccines into a vulnerable program. If v_i does *not* cause any exception, we record $B[i]$ (and also i for a binary protocol) as a signature token. A signature is formed using these tokens and the target address set T. A dataflow is deemed to match such a signature if it contains all these tokens and at least one byte sequence in T. We refer to this approach as *byte-based vaccine injection* (BVI).

Some servers process requests using multiple processes, such that crashing one does not affect the others. This property allows us to test many vaccines in *parallel*. Many exploits have exploit payload of a modest size, usually below 1kB. Therefore, we believe BVI can offer good performance. We also adopted a 'block-searching' technique to reduce the number of vaccines for generating a signature. We first test a vaccine which randomizes a *block* of contiguous bytes on an exploit packet. If the vaccine still causes the exception, we move on to randomize another byte block; otherwise, we test every byte inside that block to identify signature tokens. However, BVI could still be slow if the payload is large.

An attacker might duplicate an exploit token to several places. For example, the Code Red II worm (Figure 2) has multiple `%u` tokens, any of which is sufficient for the exploit to occur. This prevents the BVI algorithm from detecting that token, as randomization of one of its replicas does not make the exception disappear. We can solve this problem using an improved BVI algorithm described as follows. A vaccine v'_i scrambles the first i bytes on the exploit dataflow except all the signature tokens identified so far. If the vaccine does not cause any exception to the vulnerable program, the signature engine records the ith byte as a new signature token. Otherwise, our approach scrambles that byte before generating the next vaccine v'_{i+1}. This approach can capture one of the duplicated tokens. However, it is not parallelizable. Fortunately, such a duplication trick cannot be played on most tokens (e.g., `.ida` and `GET`) and thus the original BVI algorithm works in many cases.

Using Protocol Information. If an application's protocol specifications are available, in some cases we can generate a very accurate signature, close to a vulnerability-based signature. Such a signature makes use of the characteristics of buffer-overflow exploits and format-string exploits to describe a vulnerability. The algorithm for generating these signatures is also built upon the VI technique, and so we call the approach *application-based vaccine injection* (AVI).

Buffer-overflow exploits usually employ anomalously long fields [14]. Thus, a signature of the form (*application*, *command*, *field.name*, *max.field.size*) offers a good description of the vulnerability being exploited. Our signature generation engine first identifies the application field that includes the jump address, and then makes a quick estimate of that field's length using the number of the bytes prior to the address. This gives a coarse signature. To refine that signature, our approach iteratively alters the field size to generate new vaccines, and injects them into the vulnerable program. If a vaccine makes the exception disappear, we infer that the field is too short and then increase it. Otherwise, we shrink that field. Using a binary search, we can quickly determine the minimal length for the exploit to happen. The signature generated in this way can be pretty close to the size of a vulnerable buffer: for example, our experiment over ATP httpd (see Section 3.3) produced a signature only 23 bytes longer than the real size of the program's vulnerable buffer.

Format-string exploits usually contain the special symbol `%n`. In addition, the address token usually appears prior to this symbol. Therefore, a simple representation of the signature could be as follows: (*application*, *command*, *field.name*, `%n`). The accuracy of this signature can be verified by removing the `%n` from a vaccine to test the vulnerable program.

3. EVALUATION

We evaluated packet vaccine using a proof-of-concept implementation. In this section, we first describe this implementation and then present our experimental results and analysis on vaccine effectiveness and signature quality.

Our experiments were carried out on two Linux workstations: one with Redhat 7.3 operating system, Intel Pentium 4 1.5GHz CPU and 256MB memory, and the other with Redhat 6.2, Pentium 3 1GHz CPU and 256MB memory. We used the Redhat 7.3 system for all experiments except those involving the Bind TSIG exploit, which requires Redhat 6.2.

We also used several network traces to evaluate the quality of the signatures generated by our approach. Our dataset includes a trace of one million HTTP flows and one million DNS flows in and out of Indiana university.

3.1 Prototype Implementation

We implemented packet vaccine on Linux. The target address set T is extracted from an application's process `proc` files, including `maps` and `status`, and sent to a vaccine generation module. This module scans the dataflow of a recorded session for the byte sequences inside T, scrambles their most significant bytes, creates a socket to convert the new dataflow into vaccine packets and transports them to the application. On the systems running the application, we installed a process monitor developed using `ptrace`, which serves as a collector to gather the contents of important registers should an exception happen to the process being monitored. Registers important to vulnerability diagnosis are CR2 and EIP. However, CR2 can be accessed only in kernel mode. In our research, we developed a kernel patch for Linux 2.4.18 to read its content.

The signature generation engine has two components, a *prober* and a *verifier*. The prober tests an application using vaccines to identify signature tokens. It can work remotely. The verifier monitors processes for exception signals, and restarts the application if necessary. In our implementation, the verifier was embedded in the `ptrace`-based monitor. On starting signature generation, the prober first makes a persistent connection with the verifier, and then sends a vaccine packet to the application. If the application's process crashes, the verifier intercepts the exception signal and notifies the prober through the connection. Otherwise, the verifier waits for a period of time (longer than the maximum crash time) before signaling that no exception has occurred. Our implementation supports both the BVI and AVI algorithms and can generate token-sequence and application-level signatures. We implemented only sequential vaccine injection in our prototype system, which unfortunately introduced performance penalties. In our experiments, we found that some applications could take tens of milliseconds to crash. The delay caused by awaiting the crashes of multiple processes could be greatly reduced by a parallel approach.

3.2 Vaccine Effectiveness

A paramount question for packet vaccine is a vaccine's ability to detect an exploit. We address this question through an empirical evaluation reported in this section. We carried out experiments on real exploits of seven vulnerable applications obtained from SecurityFocus.[2] They have also been widely used for evaluating other techniques (e.g., [14, 40, 7]). In our research, we made sure that all these exploits were successful in the vulnerable applications by spawning a remote shell before testing them with our technique.

Packet vaccine successfully detected these exploits, and additionally diagnosed the related vulnerabilities to generate precise signatures. The details of exploits and detection results are listed in Table 1. While we implemented our proof-of-concept system only on Linux, we also analyzed another 19 exploits which include Windows-based exploits such as Code Red II. We found none of their semantics would be damaged by our approach. This implies that packet vaccine should also detect them.

Detecting a heap-based overflow turned out to be a little trickier. In the experiment on openssl, the value of the byte sequence we got from CR2 was larger than that of the randomized token by 12. We explain this as follows. The exploit took advantage of the `free()` function to overwrite a function's return address. The location of that address was faked as the content of a linking pointer in a bogus idle memory segment's heap management data structure. On the exploit's payload, the address of that segment's header was provided. That address was supposed to be lower than the linking pointer's address by 12. The exception happened when the heap management system attempted to access that linking pointer using the header's address which was randomized by our approach.

[2]Technical details of these exploits can be found by searching their Bugtraq ID from `http://www.securityfocus.com`.

Exploits	Bugtraq ID	Vulnerability Type	Exploit Packet Length	Detected	Number of Address-like Tokens
BIND tsig	2402	stack-based buffer overflow	510	Yes	3
Light httpd	6162	stack -based buffer overflow	231	Yes	13
ATP httpd	8709	stack-based buffer overflow	820	Yes	90
Samba	7294	stack-based buffer overflow	3097	Yes	26
OpenSSL v2	5363	heap-based buffer overflow	474	Yes	4
wu-ftpd	1378	format string attack	435	Yes	1
rpc.statd	1480	format string attack	1076	Yes	8

Table 1: Exploit Detection.

Exploits	Application Signature	Time(s)	Byte Sequence Signature	Time(s)
BIND tsig	—	—	4-12 (00, 01, 00, 00, 00, 00, 00, 01, 3c), 73 (3c), 134 (0c), 147 (31), 197 (0c), 210 (3e), 273 (3e), 336 (1e), 367 (10), 384 (3e), 447 (34), 500 (00), 505-507 (00, 00, fa)	4.881
Light httpd	(., 'GET', filename, 178)	0.345	0-3 (47, 45, 54, 20), 229-230 (0a, 0a)	1.360
ATP httpd	(., 'GET', filename, 703)	0.274	0-4 (47, 45, 54, 20, 2f), 818 (0a)	2.708
Samba	(., 'TRANS2_OPEN2', filename, 2000)	0.622	0-2 (00, 04, 08), 4-8 (ff, 53, 4d, 42, 32), 28-29 (01, 00), 32-33 (64, 00), 37-40 (d0, 07, 0c, 00), 55-56 (d0, 07), 58-60 (00, 0c, 00), 63-66 (01, 00, 00, 00)	7.636
OpenSSL v2	(., 'Master Key', arguments, 298)	0.358	0-11 (81, d8, 02, 01, 00, 80, 00, 00, 00, 80, 01, 4e)	5.012
wu-ftpd	(., 'SITE', 'EXEC', %n)	0.130	0-9 (53, 49, 54, 45, 20, 45, 58, 45, 43, 20), 431-432 (25, 6e)	4.228
rpc.statd	(., 'STAT', name, %n)	0.116	4-31 (00, 00, 00, 00, 00, 00, 00, 02, 00, 01, 86, b8, 00, 00, 00, 01, 00, 00, 00, 01, 00, 00, 00, 01, 00, 00, 00, 20), 36-39 (00, 00, 00, 00, 09), 60-63 (00, 00, 00, 00), 68-74(00, 00, 00, 00, 00, 00, 03), 164-165 (25, 6e)	5.780

Table 2: Signatures Generated. A token in a byte sequence signature is represented as $i - j(B_i, \ldots, B_j)$ $(i \leq j)$, where i and j are the positions of the individual bytes on the token and B_i is a byte's hexadecimal value. For example, 229-230(0a,0a) indicates that the token `0x0a0a` lies between the 229th and the 230th bytes in the payload. The position information is *optional* and not useful for text-based protocols such as HTTP.

3.3 Signature Quality and Performance

A summary of results of our experiments on signature generation can be found in Table 2. To evaluate the quality of our signatures, we compared them with signatures reported in recent literature [3]. A vulnerability-based signature can prevent all possible exploits on a vulnerability [7]. Recently, Brumley et al. have proposed a gray-box approach to generate such a signature on the basis of static analysis of a vulnerable program's binary code [3]. Their technique intensively utilizes application information.

Brumley et al. describe in their paper two *monomorphic-execution-path* (MEP) signatures, one for Bind TSIG and the other for ATP httpd. MEP signatures computed from a single exploit are usually not vulnerability-based. Nevertheless, with the information extracted from the vulnerable application, they are still very accurate. Here, we analyze our signatures using these signatures.

Quality of the Token-Sequence Signature: Bind-TSIG. Bind is a very popular DNS server. It supports a secret-key transaction authentication in which messages bear transaction signatures (TSIG). There is a buffer-overflow vulnerability in Bind 8.2.x which allows an attacker to gain control of a system running Bind. This vulnerability can be exploited through both UDP and TCP queries. Our experiments were on UDP-based exploits and Bind 8.2.2. Figure 3 presents the MEP signature (the first row) and our token-sequence signature (the second row) computed using the BVI algorithm.[3]

Both signatures include bytes 6 to 10 which are zero and bytes 505 to 507 which are `0x0000fa` (a zero-length Qname followed by the field type TSIG). From Bind's source code, we found that these bytes are the most important tokens for a successful exploit. Besides these tokens, our signature also contains some other bytes. Bytes 4 to 5 are the number of queries inside the packet. Byte 4 must be zero for the UDP-base exploit due to the size limit of a UDP-based packet. However, byte 5's content is unnecessarily specific because an exploit using more than one query could also succeed. On the other hand, byte 5 must be nonzero, which has not been pointed out by the MEP signature. Bytes 10-11 are the 'ARcount' field, which indicates the number of resource records in the additional records part. It must be nonzero to accommodate the TSIG field, but our signature is unnecessarily specific in fixing its value. Byte 12 appears in both signatures, but ours specifies its content. Ten bytes in the interval 73 to 447 in our signature are also unnecessarily specific. These ten bytes serve as the length octets in the 'Qname' field of a query, which are important for the successful parsing of a DNS query. However, an attacker may change the structure of the exploit packet to avoid these bytes. This problem is hard to avoid with only a single instance of the exploit and no application information at all.

The MEP signature also has some problems. It misses bytes 4 and 11, and also contains unnecessarily specific tokens, such as bytes 268 and 500. Byte 500 is also present in our signature. Both bytes signal the end of a query in a particular exploit. However, the attacker can avoid them by changing an exploit packet's structure, such as the number of questions and their sizes. For example, byte 268 has a nonzero value in the exploit used in our research.

A more accurate signature could be generated by our technique given more than one exploit instance. In our research, we compared another exploit of the Bind-TSIG vulnerability with the above one. These two exploit packets share 19 bytes at the same locations of their application payloads. Based on these 19 bytes, the BVI algorithm generated another signature (the third row in Figure 3) with 10 bytes.

[3]Our signature may also include the target address set T, which we believe does not make the signature too specific for a control-flow hijacking attack. This is because that set includes all possible jump targets, not a specific address.

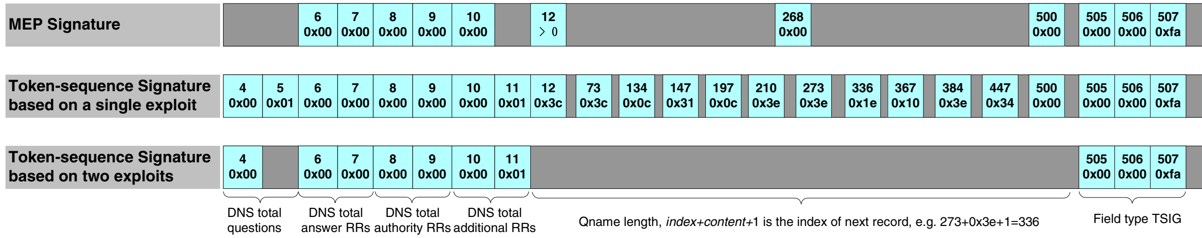

Figure 3: Signatures for Bind TSIG.

Only one of them, byte 11, is unnecessarily specific. This signature is comparable to the MEP signature in quality and capable of fending off many attacks on the vulnerability.

Using the block-searching technique, a sequential BVI algorithm took 4.881 seconds to generate the first token-sequence signature for Bind. We believe an optimized implementation and introduction of parallelization could improve that performance. The second signature was generated within 0.2 seconds.

Quality of the Application-level Signature: ATP-httpd. We also compared our application-level signature for ATP-httpd with the MEP signature in [3]. ATP-httpd contains a vulnerable buffer which will be overrun by a requested filename longer than 680 bytes. Built upon the analysis of the program's binary code, the MEP signature contains richer information than ours. It points out the HTTP command which leads to the vulnerability could be either 'GET' or 'HEAD', while our signature only identifies 'GET' from a single exploit instance. However, the MEP signature contains two specific tokens, '//' and '/', which actually are parts of the shell code. In addition, the total field length required by their signature is 812 bytes, which is not necessary for an exploit. Our signature offers a better estimate of the vulnerable buffer size. The AVI algorithm determined the maximal length of the field 'filename' as 703, 23 bytes longer than the vulnerable buffer. These 23 bytes turned out to be the local variables between the buffer and the pointer overwritten by the exploit. Our approach took 0.274 seconds to generate the signature. By comparison, the algorithm in [3] spent more than a second to complete a single step of signature generation which converts the results from static analysis into a signature.

In summary, it comes as little surprise that the MEP signatures are more accurate than our signatures in general. However, their quality advantages diminish somewhat with the availability of multiple exploit instances and application information. Furthermore, our black-box approach can perform significantly faster in some cases, and even works with obfuscated binaries which static analysis might not manage well.

Exploits	False + (Application Signature)	False + (Byte-Sequence Signature)
BIND tsig	—	w/ T, 0%, w/o T, 0%
Light httpd	0.602%	w/ T, 0%, w/o T, 0.0006%
ATP httpd	0.0077%	w/ T, 0%, w/o T, 0.142%

Table 3: False Positives. T refers to the target address set of the vulnerable application.

False Positives. We tested our signatures for Bind-TSIG, ATP-httpd and light-httpd using the aforementioned DNS and HTTP traces (Table 3). Surprisingly, most false positives come from application-level signatures, which are supposed to be very accurate! Further analysis offers the explanation: these signatures are application-dependent, only working for specific httpd servers, and supposed to be installed on the firewalls connecting to these servers. However, the HTTP traces were collected from edge routers, containing the traffic of other HTTP software that could accommodate a longer field.

4. EXAMPLE APPLICATION: PROTECTING INTERNET SERVERS

In the section, we present an architecture which applies packet vaccine to protect Internet servers from remote control-flow hijacking attacks. This architecture serves as an example to demonstrate the potential application of our technique. We also prototyped the architecture under Linux and empirically evaluated its performance.

4.1 Architecture

Figure 4 illustrates the architecture we propose. A service request is first intercepted and cached by a service proxy and parsed by a parser. The parser is optional here and only useful when we use application-level signatures. Then, the request is screened by a filter which identifies and drops known exploits using exploit signatures. Behind the filter, a *detector* examines the request and labels it as either *normal* or *suspicious*. The detector could simply be part of our packet vaccine mechanism, which classifies packets with regard to the appearance of address-like tokens in their payloads. Alternatively, we could employ other simple detection techniques, such as one which identifies packets with overlong fields. After classification, a *normal* request is forwarded to a server farm directly, while a *suspicious* request triggers the packet vaccine mechanism which acts as discussed in Section 2. If that request is determined to contain an exploit, packet vaccine generates a new signature and adds it to the filter. Otherwise, the proxy forwards the original request to the server farm.

The packet vaccine mechanism makes use of a small set of *test servers* in the server farm to test vaccine packets. A test server has a collector on it, which serves to glean information from registers' contents should an exception happen. In the case that the service being provided is stateful, the test server also needs a checkpoint/rollback (CR) mechanism to recover the state before each test. Such a rollback mechanism could be extremely lightweight (e.g., [8, 31]). Signature generation can also happen on a test server.

4.2 Performance Study

To implement a prototype system for HTTP service, we developed a service proxy and a filter (including an HTTP parser), and combined them with our implementation of

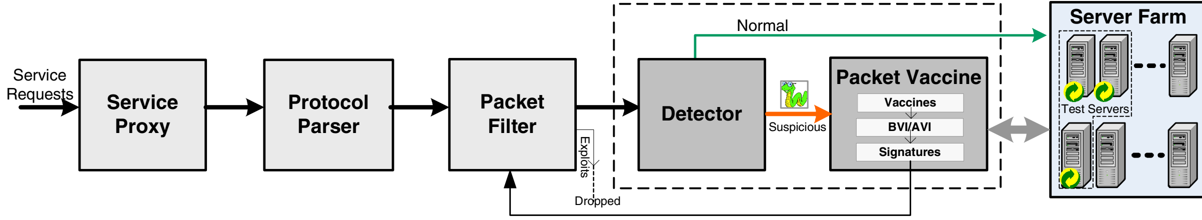

Figure 4: An architecture to protect Internet servers using packet vaccine.

packet vaccine (Section 3.1) which contains a detector. Since HTTP is a stateless service, we did not implement the process-level CR in this prototype.

Over the prototype system, we carried out a performance test. Two hosts were used in our experiment, one for both the proxy and the test server and the other for the web server. Both were equipped with 2.53GHz Intel Pentium 4 Processor and 1 GB RAM, and running Redhat Enterprise 2.6.9-22.0.1.EL. They were interconnected through a 100MB switch. We utilized an Apache 2.0.55 to provide web service. In our experiment, we evaluated the performance of our implementation from the following perspectives: (1) *Server overheads*, where we compared the workload capacity of our implementation with that of an unprotected Apache server; (2) *Client-side delay*, where we studied the average delay a client experiences under different test rates.

Server overheads. We tested the workload capacity using ApacheBench (ab) 2.0.41-dev, which comes bundled with the Apache source distribution. ApacheBench is a tool for benchmarking the Apache web server. In our experiment, we measured the workload capability in terms of requests processed per second (requests/second) under the following five server configurations: (**0**) 'Apache only', (**D0**) 'Apache and the proxy on different hosts', (**S0**) 'Apache and the proxy on the same host', (**D1**) 'Apache on one host, and the proxy and packet vaccine on another', (**S1**) 'Apache, proxy and packet-vaccine all on the same host'.

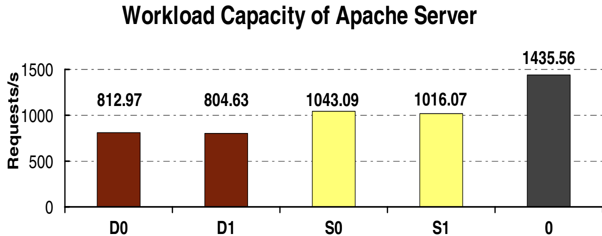

Figure 5: The workload capacities in five different server settings.

Figure 5 illustrates the experiment results. At a first glance, it seems that our implementation brought down the Apache's performance by about 44% in the setting (**D1**) and about 29% in the setting (**S1**), which is quite unpleasant. A close look at the results, however, reveals that the major performance penalty came from the service proxy. The homegrown proxy used in our proof-of-concept implementation could not keep up with the high-performance of Apache and therefore dragged down the performance of the whole system. Simply adding the proxy into the system introduced about 43% performance penalty in (**D0**) and 27% in (**S0**). On the other hand, the packet vaccine components worked pretty fast. They only affected the performance by 1% to 2%. Therefore, we tend to believe that a high-performance HTTP proxy could greatly improve the workload capability.

Client-side delay. Once the detector identifies a suspicious request, a round of exploit detection will be triggered to test that request. This introduces delay to a legitimate client if the request turns out to be innocent. Here, we call the ratio of service requests being tested (i.e., the fraction deemed suspicious) the *test rate*. If the test rate increases, the average delay experienced by a legitimate client will also increase. In our experiment, we studied the change of the client-side delay against different test rates. We carried out both a local experiment within IU's campus network and a cross-campus experiment between IU and NCSU. The experimental results are presented in Figure 6.

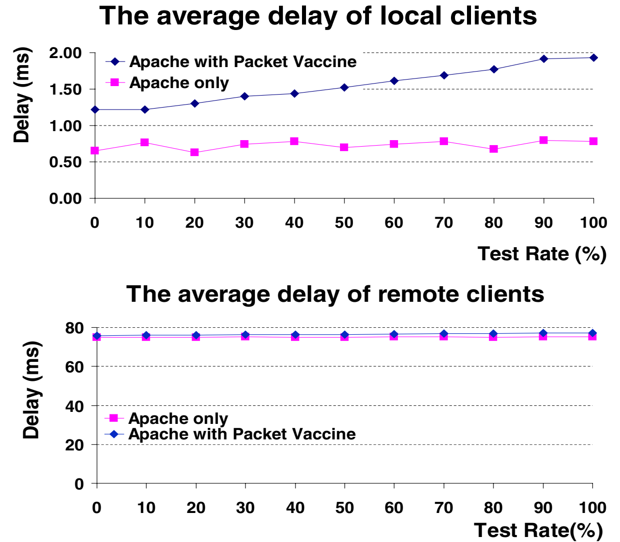

Figure 6: The average delay experienced by a local or remote client.

As we expected, the average delay for a local client increased almost linearly with the test rate. However, this result could be misleading, as the local client experienced much smaller round trip delay (RTD) than an average Internet user: the RTD in a campus we measured is around $300\mu s$, while the average RTD on the Internet is much larger. Therefore, an Internet client's perception of the presence of packet vaccine could be completely overshadowed by the RTD. This was confirmed in the cross-campus experiment: as presented in Figure 6, the $75ms$ RTD between the two campuses dominated the client-side delay, making the $1ms$ overhead of our protection mechanism negligible.

In summary, packet vaccine does introduce performance penalties to the server, but we believe this penalty is acceptable if weighed against the security enhancements it offers.

On the other hand, the client-side overhead is almost negligible, being dwarfed by the RTD an average Internet client experiences.

5. LIMITATIONS

Packet vaccine may have false negatives in exploit detection. For example, there is a possibility that the randomizations performed by our approach destroy the exploit's semantics. This seems more likely to occur for applications using binary protocols, though so far we have not found an example "in the wild". In general, our approach is more reliable in protecting applications using text-based protocols. Several ways to reduce the likelihood of this problem were discussed in Section 2.3. A simple approach is to generate multiple vaccines, each randomizing one byte of an address-like token. In this way, if the exploit semantics survives any of these randomizations, our approach will detect the exploit.

Our approach cannot work directly on packets with encrypted payload or checksums. In this case, we need an application-level proxy to decode these packets and construct new packets for vaccine generation.

Both types of signatures we use in our research are limited in their capabilities to represent necessary exploit conditions. For example, null-httpd contains a vulnerability that allows one to specify a smaller buffer while supplying a longer payload. An ideal signature is to check whether the real payload size matches the specified size. However, none of our signatures can describe this condition. We leave it to future work to examine how to use our black-box techniques to acquire information for more expressive signatures [38, 3].

6. RELATED WORK

Network anomaly detection (NAD) has been widely used to detect exploit attempts from network traffic [41, 39, 35, 12]. A typical network signature generator extracts common substrings from attack dataflow as an exploit signature. Examples include Earlybird [30], Honeycomb [11], Autograph [10], SweetBait [26], Polygraph [22], Hamsa [13] and PADS [32]. Signature generation solely relying on network information can be misled into generating an incorrect signature by carefully crafted attack packets, which helps a worm to evade detection [25] or causes legitimate packets to be dropped.

Host-based approaches make use of host information to detect anomalies and generate signatures. As exploits actually happen on a host, these approaches can be more accurate than network-based approaches. TaintCheck [23], VSEF [21], Minos [6], Vigilante [5] and DACODA [7] track dataflow through a process from the receipt of a network packet (or modification thereof [23]) to the point where an anomaly happens, e.g., jumping to an address offered by the input data. These approaches can slow the running process significantly, however, by an order of magnitude or more. In contrast, our vaccine mechanism tracks suspicious dataflow in a black-box fashion, which is significantly faster than these gray-box approaches and still preserves much of their accuracy in cases we have explored. Some host-based approaches apply static analysis [3] to identify a program's vulnerabilities. Such an approach no longer works over well-obfuscated binaries.

Liang et al. and Xu et al. proposed two approaches [40, 14] that use memory address-space randomization (ASR) to foil exploit attempts, and then automatically generate signatures through forensic analysis of the related exceptions. In particular, COVERS [14] was the first to propose a novel construction of application-level signature which uses field length to characterize a buffer overflow vulnerability. Although we also use this signature, our AVI technique augments their approach by making an accurate estimate of the field length. Our technique also offers a more reliable way to correlate exceptions with the exploit packets.

In an attempt to find a balance between performance and accuracy, several hybrid approaches combining network-based and host-based techniques have been developed [1, 15, 29]. However, many of them are based on instrumenting a vulnerable program's source code, and are therefore less suitable for protecting commodity software. HACQIT [27] invokes a test process after an exploit crashes a protected program, and replays suspicious packets to a sandbox running the same program to monitor whether the same exception happens again. However, this approach does not offer a reliable means to establish a correlation between the exception and the exploit inputs.

The vaccine technique can trace its root to software robustness testing, especially software-implemented fault injection (SWIFI) [18]. SWIFI is a software testing and evaluation method which involves inserting random faults into a system to determine its response to these faults. Some important SWIFI systems include the Crashme program [4], the Fuzz project [17], the FIAT system [2], the FERRARI system [9], the FTAPE system [36], and Ballista [33]. Our proposal differs fundamentally from these approaches in two respects. First, we rely on anomalous packets to guide vaccine generation, making our vaccines more likely to reveal a program's vulnerabilities than the random faults used in a typical SWIFI approach. Second, we aim at exploit prevention and will generate exploit signatures to shield the software vulnerabilities discovered.

7. CONCLUSIONS

In this paper, we presented packet vaccine, a fast, black-box technique for exploit detection, vulnerability diagnosis and signature generation. We described its design and examples for its application. We also implemented a proof-of-concept prototype, and evaluated our technique using it. Our experimental results demonstrate the effectiveness of our technique, which successfully captures real exploits and generates effective signatures, and its efficiency, which improves over gray-box approaches in many cases.

8. REFERENCES

[1] K. G. Anagnostakis, S. Siridoglou, P. Akritidis, K. Xinidis, E. Markatos, and A. Keromytis. Detecting targeted attacks using shadow honeypots. In *Proceedings of USENIX Security Symposium 2005*, August 2005.

[2] J. H. Barton, E. W. Czeck, Z. Z. Segall, and D. P. Siewiorek. Fault injection experiments using FIAT. *IEEE Trans. Comput.*, 39(4):575–582, 1990.

[3] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006.

[4] George J. Carrette. CRASHME: Random input testing. `http://people.delphiforums.com/gjc/crashme.html`, as of March, 2006.

[5] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony I. T. Rowstron, Lidong Zhou, Lintao Zhang, and Paul T. Barham. Vigilante: end-to-end containment of internet worms. In *SOSP*, pages 133–147, 2005.

[6] Jedidiah R. Crandall and Frederic T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *MICRO*, pages 221–232, 2004.

[7] Jedidiah R. Crandall, Zhendong Su, and S. Felix Wu. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 235–248, 2005.

[8] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of OSDI*, 2002.

[9] Ghani A. Kanawati, Nasser A. Kanawati, and Jacob A. Abraham. FERRARI: A flexible software-based fault and error injection system. *IEEE Trans. Comput.*, 44(2):248–260, 1995.

[10] Hyang-Ah Kim and Brad Karp. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of 13th USENIX Security Symposium*, pages 271–286, San Diego, CA, USA, August 2004.

[11] Christian Kreibich and Jon Crowcroft. Honeycomb: creating intrusion detection signatures using honeypots. *SIGCOMM Computer Communication Review*, 34(1):51–56, 2004.

[12] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *Proceedings of RAID'05*, pages 207–226, September 2005.

[13] Zhichun Li, Manan Sanghi, Yan Chen, Ming-Yang Kao, and Brian Chavez. Hamsa: Fast signature generation for zero-day polymorphicworms with provable attack resilience. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 32–47, 2006.

[14] Zhenkai Liang and R. Sekar. Fast and automated generation of attack signatures: a basis for building self-protecting servers. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 213–222, 2005.

[15] Michael E. Locasto, Ke Wang, Angelos D. Keromytis, and Salvatore J. Stolfo. Flips: Hybrid adaptive intrusion prevention. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2005.

[16] MemView. `http://www2.biglobe.ne.jp/~sota/memview-e.html`, as of May, 2006.

[17] Barton Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical report, 1995.

[18] J.D. Musa, G. Fuoco, N. Irving, B. Juhlin, and D. Kropfl. *Handbook of Software Reliability Engineering*, chapter The Operational Profile, pages 167–216. McGraw-Hill, 1996.

[19] Gleb Naumovich and Nasir D. Memon. Preventing piracy, reverse engineering, and tampering. *IEEE Computer*, 36(7):64–71, 2003.

[20] Associate Press News. Microsoft warns against outside fixes. `http://biz.yahoo.com/ap/060331/microsoft_s_security_snags.html?.v=4`, March 31, 2006.

[21] James Newsome, David Brumley, and Dawn Song. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *Proceedings of the* 13th *Annual Network and Distributed Systems Security Symposium*, 2005.

[22] James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 226– 241, Okaland, CA, USA, May 2005.

[23] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, San Diego, CA, USA, Feburary 2005.

[24] A. Pasupulati, J. Coit, K. Levitt, S.F. Wu, S.H. Li, R.C. Kuo, and K.P. Fan. Buttercup: On network-based detection of polymorphic buffer overflow vulnerabilities. In *Proceedings of the9th IEEE/IFIP Network Operation and Management Symposium (NOMS'2004)*, May 2004.

[25] Roberto Perdisci, David Dagon, Wenke Lee, Prahlad Fogla, and Monirul Sharif. Misleading worm signature generators using deliberate noise injection. In *IEEE Symposium on Security and Privacy*, page to appear, May 2006.

[26] Georgios Portokalidis and Herbert Bos. SweetBait: Zero-hour worm detection and containment using honeypots. Technical Report IR-CS-015, Vrije Universiteit Amsterdam, May 2005.

[27] James C. Reynolds, James Just, Larry Clough, and Ryan Maglich. On-line intrusion detection and attack prevention using diversity, generate-and-test, and generalization. In *HICSS '03: Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03) - Track 9*, page 335.2, 2003.

[28] David W. Richardson, Steven D. Gribble, and Edward D. Lazowska. The limits of global scanning worm detectors in the presence of background noise. In *WORM '05: Proceedings of the 2005 ACM workshop on Rapid malcode*, pages 60–70. ACM Press, 2005.

[29] Stelios Sidiroglou, Michael E. Locasto, Stephen W. Boyd, and Angelos D. Keromytis. Building a reactive immune system for software services. In *USENIX Annual Technical Conference*, pages 149 – 161, April, 2005.

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

[31] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX Annual Technical Conference, General Track*, pages 29–44, 2004.

[32] Yong Tang and Shigang Chen. Defending against internet worms: A signature-based approach. In *Proceedings of IEEE INFOCOM05*, Miami, Florida, USA, May 2005.

[33] The Ballista© Project: COTS Software Robustness Testing. `http://www.ece.cmu.edu/~koopman/ballista/`, as of January, 2006.

[34] Microsoft Debuging Tools. `http://www.microsoft.com/whdc/devtools/debugging/default.mspx`, as of May, 2006.

[35] Thomas Toth and Christopher Krügel. Accurate buffer overflow detection via abstract payload execution. In *Proceedings of RAID*, pages 274–291, 2002.

[36] Timothy K. Tsai and Ravishankar K. Iyer. Measuring fault tolerance with the ftape fault injection tool. In *MMB '95: Proceedings of the 8th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 26–40. Springer-Verlag, 1995.

[37] Paul C. van Oorschot. Revisiting software protection. In *Proceedings of ISC*, pages 1–13, 2003.

[38] Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier. Shield: vulnerability-driven network filters for preventing known vulnerability exploits. In *SIGCOMM*, pages 193–204, 2004.

[39] Ke Wang and Salvatore J. Stolfo. Anomalous payload-based network intrusion detection. In *Proceedings of RAID Symposium 2004*, pages 203–222, 2004.

[40] Jun Xu, Peng Ning, Chongkyung Kil, Yan Zhai, and Chris Bookholt. Automatic diagnosis and response to memory corruption vulnerabilities. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 223–234, 2005.

[41] Vinod Yegneswaran, Jonathon T. Giffin, Paul Barford, and Somesh Jha. An architecture for generating semantics-aware signatures. In *Proceedings of USENIX Security Symposium 2005*, August 2005.