Vulnerability-Specific Execution Filtering for Exploit Prevention on Commodity Software

James Newsome David Brumley Dawn Song

May 2005
Last modified November 2005

CMU-CS-05-169

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Keywords: Worms, exploits, antibodies, vulnerability-specific execution filtering, VSEF

Abstract

Exploits for new vulnerabilities, especially when incorporated within a fast spreading worm, can compromise nearly all vulnerable hosts within a short amount of time. This problem demonstrates the need for fast defenses which can react to a new vulnerability quickly. In addition, a realistic defense system should (a) not require source code since in practice most vulnerable systems do not have source code access nor is there adequate time to involve the software vendor, (b) be accurate, i.e., have a negligible false positive rate and low false negative rate, and (c) be efficient, i.e., add little overhead to normal program execution.

We propose vulnerability-specific execution-based filtering (VSEF) – a new approach for automatic defense which achieves a lower error rate and wider applicability than input filters and has better performance than full execution monitoring. VSEF is an *execution-based filter* which filters out attacks on a specific vulnerability based on the vulnerable program's execution trace. We present VSEF, along with a system for automatically creating VSEF filters and a hardened program without access to source code. In our system, the time it takes to create the filter and generate the hardened program is negligible. The overhead of the hardened program is only a few percent in most cases. The false positive rate is zero in most cases, and the hardened program is resilient against polymorphic variants of exploits on the same vulnerability. VSEF therefore achieves the required performance, accuracy, and response speed requirements to defend against current fast-spreading exploits.

1 Introduction

The number of new vulnerabilities reported each year continues to grow. According to CERT/CC, in 1995 171 new vulnerabilities were reported, while less than a decade later in 2004 over 3700 new vulnerabilities were discovered [9]. A new exploit for a single vulnerability can readily be turned into worms which compromise hundreds of thousands of machines within only a few minutes [22, 35]. Thus, after a vulnerability is discovered it is important to quickly develop effective mechanisms to protect vulnerable hosts so that (1) they will not be compromised by exploits of the vulnerability, and (2) provide service without disruption.

The speed at which new vulnerabilities are discovered and exploits created necessitates new defenses that meet several goals simultaneously: (1) *Fast defense development and deployment*: there is often very little reaction time, especially when the exploit comes in the form of a fast propagating worm. Thus, we need to be able to develop and deploy defense mechanisms extremely quickly after the detection of a vulnerability. (2) *No requirement for source code*: many vulnerable programs are commodity software for which the source code is proprietary. To respond quickly to new vulnerabilities, we need to be able to develop a defense mechanism without access to source code, so we do not rely on the cooperation of the software vendor. (3) *High accuracy and effectiveness*: the defense mechanism should protect against the vulnerability and should not have any undesirable side effect on normal execution. It should have a low false positive rate (not blocking legitimate requests) and a low false negative rate (even effective against polymorphic attacks). (4) *Low performance overhead*: the defense mechanism should have low performance overhead, so a vulnerable host deploying the defense mechanism can still provide critical services with little performance degradation.

Many defense mechanisms have been proposed to protect a vulnerable host after a vulnerability has been discovered. Previous work has various drawbacks and do not satisfy all the above requirements. One popular approach is to automatically generate network-based *input filters* to filter out known exploits [16, 34, 18, 27, 26]. However, the accuracy and effectiveness of the network-based input filtering approach is fundamentally limited to syntactic properties of the input string and cannot take into account application-specific semantic and context information. In particular, there may be no syntax-based classifier to correctly distinguish between malicious and innocuous traffic for certain applications or vulnerabilities due to polymorphic attacks; and the lack of context information in network-based input filtering can have high false positive rate for certain applications. Input filters also have difficulty recognizing semantically equivalent inputs, such as alternate URL encodings, which leads to false negatives. In the extreme case where an input filter is used on an encrypted protocol, it must somehow be supplied with the decryption key, which is awkward and application-specific. Costa et. al. propose automatically generated *host-based* input filters [11], which has greater accuracy than network-based input filters, and can correctly recognize some semantically equivalent inputs. However, the approach still suffers difficulty when the correct classification rule is complex and needs program state information, or when input is encrypted. Therefore the input filtering approach is not a complete solution.

On the other hand, various host-based approaches have been proposed which are more accurate, but fail to meet the other requirements. For example, previous approaches have focused on: (1) *Patching*: patching a new vulnerability can be a time-consuming task—generating high quality patches often require source code, manual effort, and extensive testing. Applying patches to an existing system also often requires extensive testing to ensure that the new patches do not lead to any undesirable side effects on the whole system. (2) *Binary-based full execution monitoring*: many approaches have been proposed to add protection to a binary program. However, these previous approaches are either inaccurate and only defend against a small classes of attacks [6, 31, 17, 23] or require hardware modification or incur high performance overhead when used to protect the entire program execution [14, 27, 36, 11].

In this paper, we propose a new approach for automatic defense: *vulnerability-specific execution-based filtering* (VSEF). At a high-level, VSEF filters out exploits based on the program's execution, as opposed to filtering based solely upon the input string. However, instead of instrumenting and monitoring the full execution, VSEF only monitors and instruments the part of program execution which is relevant to the specific vulnerability. VSEF therefore takes the best of both input-based filtering and full execution monitoring: it is much more accurate than input-based filtering and much more efficient than full execution monitoring.

We also develop the first system for automatically creating a VSEF filter for a known vulnerability *given only a program binary*, and a sample input that exploits that vulnerability. Our VSEF Filter Generator automatically generates a VSEF filter which encodes the information needed to detect future attacks against the vulnerability. Using the VSEF filter, the vulnerable host can use our VSEF Binary Instrumentation Engine to automatically add instrumentation to the vulnerable binary program to obtain a hardened binary program. The hardened program introduces very little overhead and for normal requests performs just as the original program. On the other hand, the hardened program detects and filters out attacks against the same vulnerability. Thus, VSEF protects vulnerable hosts from attacks and allow the vulnerable hosts to continue providing critical services.

Contributions. The central contribution of this paper is a new approach for automatic defense against known vulnerabilities, called vulnerability-specific execution-based filtering. Using the execution trace of an exploit of a vulnerability, our VSEF automatically generates a hardened program which can defend against further (polymorphic) exploits of the same vulnerability. VSEF achieves three important goals: low performance overhead, fast generation, and a low error rate. Specifically:

- Our VSEF is an extremely fast defense. In general, it takes a few milliseconds for our VSEF to generate the hardened program from an exploit execution trace.
- Our VSEF filtering techniques provide a way of detecting exploits of a vulnerability more accurately than input-based filters and more efficiently than full execution monitoring.
- Our techniques do not require access to source code, and are thus applicable in realistic environments.
- We provide two VSEF filtering mechanisms for detecting overwrite attacks, including buffer overflows, double-free attacks, and format string vulnerabilities. The first mechanism, taint-based VSEF, is the most accurate and requires potentially a longer filter. The second mechanism, destination-based VSEF, is more efficient and is still highly accurate. Both mechanisms have zero false positives in most cases, and are effective against polymorphic variants of the exploit of the vulnerability. Note that our approach is general, and could potentially be applied to other faults such as integer overflow, divide-by-zero, *etc.*
- Our experiments show that the performance overhead of the hardened program is usually only a few percent.

These properties make VSEF an attractive approach toward building an automatic worm defense system that can react to extremely fast worms.

2 Approach: Vulnerability-Specific Execution-based Filtering

Overview. We propose a new approach for automatically defending against just-discovered attacks, *vulnerability-specific execution-based filtering* (VSEF). VSEF is based on the observation that for a specific vulnerability only the part of the program execution that is relevant to the exploit of the vulnerability need be monitored. VSEF monitoring has full context and semantic information, as opposed to input-based filters which are limited to syntactic properties. Instrumenting the binary to perform the vulnerability-specific execution filtering results in a hardened binary. As a result, VSEF is much more accurate than network-based filtering, and much more efficient than full execution monitoring. The combination of accuracy and low overhead makes the VSEF approach very attractive for automatic deployment schemes.

The main research questions for enabling VSEF include (1) what part of the program should we monitor/instrument, (2) how can we detect and filter out the attack when we only monitor/instrument part of the program, and (3) how can we minimize the overhead of the VSEF defense. In this paper we address these questions. In particular, we propose an architecture that will automatically create VSEF filters and harden the vulnerable program given an exploit execution trace.

VSEF Architecture. Figure 1 shows the overall architecture. Our architecture contains two main components: the VSEF Filter Generator and the VSEF Binary Instrumentation Engine. To enable VSEF, we assume that a sample exploit has been detected by some exploit detector which outputs an exploit execution trace. The exploit execution trace contains the information about the program execution up to the detected exploit of the vulnerability. The exploit execution trace can be a simple instruction trace dump of the program execution or some more intelligent output from the exploit detector. The VSEF Filter Generator uses the exploit execution trace to create a VSEF filter which encodes the information needed for the monitoring to detect future attacks on the vulnerability. The VSEF filter can then be disseminated.

Vulnerable hosts use the VSEF Binary Instrumentation Engine to apply a VSEF filter to a binary. The result is a hardened binary program. The hardened program functions like the original program for normal requests and introduces very little overhead. The hardened program, however, detects and filters out attacks against the same vulnerability. Thus, VSEF protects vulnerable hosts from attacks and allows the vulnerable hosts to continue to provide critical services.

VSEF Requirements. The vulnerability-specific execution filtering architecture should have the following properties:

- Robust VSEF filters. A VSEF filter should be vulnerability-specific but exploit agnostic. For example, it should be able to detect the sample exploit even when a polymorphic engine has been used to encrypt the payload [37]. Note that input filters are particularly vulnerable to polymorphism, as there may not be enough syntactic information in the input to reliably detect polymorphic variants.
- Efficient generation of VSEF filters. Once a vulnerability is discovered, it often takes days or months to prepare a suitable patch. However, fast worms may be able to infect the entire Internet in under a few minutes. We should be able to generate filters quickly enough to allow an effective response to such flash events.

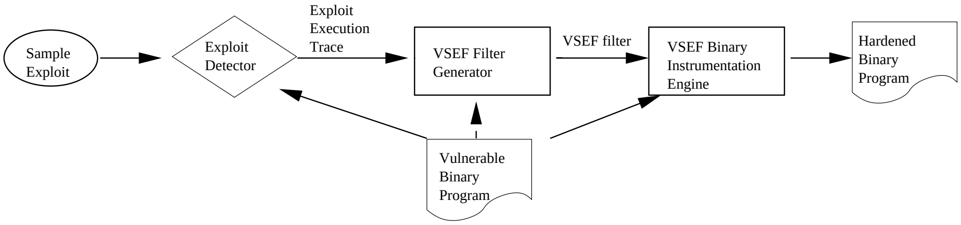

Figure 1: VSEF architecture. Once an exploit is detected, an execution log is produced. The VSEF Filter Generator produces a filter that recognizes execution patterns that exploit the vulnerability. These filters can then be disseminated. The VSEF Filter Generator takes the filter and instruments the binary to recognize execution sequences that exploit the vulnerability, the result being a hardened binary.

- Efficient detection. The vulnerability-specific execution filtering should add as little overhead as possible to program execution.

3 Taint-based and Stack-based VSEF

In this section, we present two concrete examples of our VSEF system: the taint-based VSEF and the destination-based VSEF. The taint-based VSEF is based on dynamic taint analysis and has high accuracy. The destination-based VSEF is an optimistic version of taint-based that usually requires fewer instructions instrumented.

3.1 Taint-based VSEF

3.1.1 Overview

One effective method recently proposed to detect memory-safety based attacks is dynamic taint analysis [14, 27, 36, 11]. Dynamic taint analysis marks data coming from untrusted sources (such as the network) *tainted*, and then keeps track of what data becomes tainted by untrusted input data by inserting *instrumentation* instructions to propagate the taint attribute. For example, it adds instrumentation to each data movement instruction (`mov`, `push`, `pop`, *etc.*), and data arithmetic instruction (`add`, `sub`, `xor`, *etc.*), so that the result of the instruction will be marked tainted if and only if any operand of the instruction is tainted. Dynamic taint analysis also inserts extra instrumentation before every point where data is used in a sensitive way (such as return addresses, function pointers, and format strings) to ensure that the data is not tainted. Dynamic taint analysis has been shown to accurately detect a wide range of exploit attacks including buffer overrun, format string, and double free attacks [14, 27, 36, 11], making it one of the most comprehensive protection mechanisms that does not require access to source code.

However, dynamic taint analysis requires instrumenting many instructions. Every data movement, arithmetic, and control transfer instructions that could potentially touch a tainted memory location must be instrumented in order to accurately propagate the taint attribute and detect when tainted data is misused. Such extensive instrumentation can add significant performance overhead — up to a factor of 30 or more in some cases [27].

We observe that when exploiting a particular vulnerability, only a handful of instructions are involved in propagating the tainted input to the sensitive location that is overwritten. When we know what those instructions are, we can instrument *only those* instructions to propagate the taint attribute, and the instruction that unsafely uses the tainted data, and still successfully detect attacks against that vulnerability.

Thus, in taint-based VSEF, we automatically identify and instrument the instruction positions that need to be instrumented to propagate the taint attribute and to detect the misuse of tainted data to detect exploits of a particular vulnerability. As a result, taint-based VSEF can detect exploits of the same vulnerability much more efficiently than full execution monitoring.

3.1.2 Taint-based VSEF Filter Generation

A taint-based VSEF filter includes two parts: (1) the list of instruction positions that we need to add instrumentation to for taint propagation, and (2) the instruction position to which we need to add instrumentation to detect the misuse of tainted data. Instruction positions can be expressed as absolute addresses, or as the name of a shared library and offset into that library for increased portability.

The instruction position that we need to add instrumentation to to detect the misuse of tainted data is simply the instruction position where tainted data was detected being misused. The list of instruction positions that we need to add instrumentation to for taint propagation is the list of instructions that propagated the taint attribute from the original malicious input to the point where it was detected being misused in the exploit execution trace.

The VSEF Filter Generator can identify this list using 1) any exploit detector that can identify the tainted data that was misused and what instruction misused it, and 2) a log of instructions that have been executed, and the values of dynamically calculated addresses. The latter can be logged in software, or generated efficiently using hardware support [7, 32]. The VSEF Filter Generator examines the trace in a backward manner to determine which instructions propagated tainted data that reached the vulnerability detection point. It begins at the end of the trace, called the

	C source		IA-32 assembly	Taint propagation
1	struct dummy_t {			
2	char buf[16];			
3	void (*fnptr)(void);			
4	};			
5				
6	void vuln(struct dummy_t *dummy)			
7	{			
8	char bigbuf[100];			
9	int i = 0;			
10	int count = 0;			
11	void (*fnptr)(void) = NULL;			
12				
13	fgets(bigbuf, 100, stdin);	**A**	int $0x80	0x3a966010 ← stdin
		B	repz movsb %ds:(%esi),%es:(%edi)	0xafefea80 ← 0x3a966010
14	strcpy(dummy− >buf, bigbuf);	**C**	movzbl (%edx),%eax	al ← 0xafefea80
		D	mov %al,(%ecx,%edx,1)	0x80ad1b0 ← al
15	fnptr = dummy− >fnptr;	**E**	mov 0x10(%eax),%eax	eax ← 0x80ad1b0
		F	mov %eax,0xffffffcc(%ebp)	0xafefea64 ← eax
16	fnptr();	**G**	mov 0xffffffcc(%ebp),%eax	eax ← 0xafefea64
		H	call *%eax	illegal use of tainted eax
17	}			

Table 1: Overwrite example: A piece of vulnerable code, and the instructions that propagated and misused the tainted data when the vulnerability was exploited. Instruction position **D** is the overwrite point, where tainted data overwrites a function pointer. Instruction position **H** is the exploit point, where the tainted data is misused.

exploit point, where the exploit was detected. The source operand to this instruction must have been tainted by some previous instruction in the trace (since this is an overwrite attack using tainted data); and the source operand of that instruction must have been tainted by some other previous instruction, etc.. The VSEF Filter Generator continues performing the analysis recursively until it reaches the initial instructions for reading the original untrusted input in the sample exploit.

By following the chain of tainted operations backwards, the VSEF Filter Generator can identify the list of instructions in the execution trace which were involved in propagating the taint attribute from the original untrusted input to the exploit point. This list of instructions is used in the filter as the list of instructions to be instrumented to propagate the taint attribute. This calculation is an instance of flowback analysis [5], a well studied and efficient procedure [38].

An obvious choice for the exploit detector is a taint-based exploit detector [14, 27, 36, 11]. In particular, TaintCheck [27] already keeps a directed acyclic graph (DAG) of where tainted data was propagated from, and at at what instruction points. That is, each time tainted data is propagated, a node is generated that contains the position of the currently executing instruction, and pointers to nodes corresponding to each tainted operand. In this approach all the information needed to calculate the filter is already on hand. The VSEF Filter Generator simply follows the DAG from the point(s) where tainted data was misused to the point(s) where it was originally input, and records all the instruction positions on that path.

Table 1 shows an example of code that is vulnerable to an overwrite attack, in this case a buffer overflow that overwrites a function pointer. The second column shows the assembly instructions that are involved in propagating tainted data to the point where it is misused. The third column shows the actual propagation, with the data addresses as resolved at run time. In this example, the exploit is detected at instruction **H**, where tainted data in `eax` is misused. The VSEF Filter Generator traces backwards in the execution log (or the DAG if using TaintCheck) and finds that instruction **G** was the last instruction to write to `eax`, and so on, back to instruction **A** which performed a `read` system call. Hence, the taint-based VSEF filter consists of position **H**, where tainted data was misused, and positions **A** through **G**, which propagated the tainted data to that point.

3.1.3 Taint-based VSEF Binary Instrumentation

The Taint-based VSEF Filter Generator instruments each instruction in the taint-based VSEF filter to propagate taint information, and inserts the appropriate safety check at the exploit point. The instrumentation conceptually keeps a list of tainted memory locations. When an instruction listed in the VSEF executes, the added instrumentation checks to see if any source operand is a tainted location. If so, it marks the destination as also tainted. The Taint-based VSEF Filter Generator inserts instrumentation at the exploit point to detect if the sensitive value being used is tainted, signifying an attack, and if so to take appropriate action. Here, we assume the appropriate action is exiting the program. Others have investigated other actions, such as returning an error code and continuing execution [30, 33]. The resulting program with the added instrumentation is the hardened binary.

When the hardened binary is run, the instrumentation propagates the taint attribute throughout the program as would have been done by a full taint-based exploit detector. If the exploit point is reached, and the data being used in a sensitive way has been marked tainted, execution is aborted.

Since the VSEF Filter Generator does not instrument all data movement and arithmetic instructions, tainted locations are not marked untainted when overwritten with untainted data by uninstrumented instructions. This could potentially lead to false positives in some cases. For example, suppose a stack-based buffer marked as tainted is popped off the stack, and is later overwritten with a (legitimate) return address, without being marked untainted.

We address this problem by having the hardened binary record the value that a location takes on when it is marked as tainted. When another instrumented instruction later checks to see if that location is tainted, it also checks to see if it still has the same value. If not, then it has been overwritten by an uninstrumented instruction, and is marked as no longer tainted. This approach adds little overhead, but there is still some potential for false positives. If an uninstrumented instruction overwrites tainted data with *the same value* that was already there, this heuristic will not correctly untaint that location.

An alternative approach is to use existing memory watch-point techniques to monitor tainted locations, and untaint them when other instructions write to them. On the IA-32 architecture the debug registers can be used to monitor up to 4 memory locations (up to 4 bytes each). We can also use page-protection techniques (e.g., setting tainted memory pages to be read-only) to be notified of writes to tainted memory. Moreover, when available, we can also use ECC memory to be notified of writes to tainted memory similar to techniques used in [28]. All of these techniques generate a trap when the watched memory is accessed (or memory near the watched memory), allowing our code to untaint the watched location if it has been rewritten by untainted data. The cost of generating traps when data is untainted can be reduced by reducing the amount of data that gets tainted. One way to achieve this is to modify the instrumentation of each of the data propagation instruction in the VSEF filter, so that it will only taint the destination when executing in the same call-stack context as during the original exploit. This technique comes with a trade-off of false negatives when data is tainted by the same instructions, but in a different context, until the alternate contexts are discovered and added to the VSEF filter. While we are unaware of existing mechanisms to watch for writes to processor registers, we expect that a processor register will not remain tainted for long before being overwritten with untainted data. Hence, when a register becomes tainted, we can switch to monitoring all instructions until it becomes untainted again. We show how to efficiently turn full taint analysis on and off at run time in [25].

3.1.4 Analysis and Combining Filters

Performance. By design the taint-based VSEF filter can be created with information already on hand to a Taint-based detector. As a result, filter generation is almost instantaneous. The length of the filter is proportional to the number of instructions that propagate tainted data from the input to the exploit point. Similarly, the execution overhead of the hardened program is proportional to this number of instructions. By design, most programs attempt to minimize unnecessary data copying, so this will intuitively be a small set of instructions. We verify this in our experimental results.

Note that it is likely that one or more of the instructions that propagate tainted data in the attack belong to a commonly used data movement function such as `strcpy` or `memcpy`, and hence the instrumentation will be executed any time that function is called. In our evaluation this was true, though we did not find it to be a performance problem. If it were, we could use the techniques described in Section 3.2 so that the instrumentation is only executed when the function is called in the vulnerable context.

Accuracy. The VSEF-hardened binary has no false positives when memory watchpoint techniques are used to ensure locations are correctly marked untainted when written to by uninstrumented instructions. There is nothing marked as tainted by the instrumentation that was not actually derived from untrusted input, and during detection we already determined that the attacker should not be able to write to the sensitive value being guarded. Note that without using memory watchpoint support, the untaint heuristic will not correctly untaint data if it has been overwritten by untainted data with the same value, which could lead to false positives. However, we have not encountered any in practice and expect them to be rare.

A false negative is when the same vulnerability is exploited without being reported. This can occur if the tainted input is propagated along a different code path than in the sample exploit, or if the overwritten sensitive value is misused at a different location. Note polymorphic variants created by tools such as MetaSploit [3] will be detected from a single filter. The reason is such polymorphic variants differ in the payload, which would be executed strictly after the exploit point. Only an exploit that is polymorphic in the execution path exploited could be missed. Specifically, it would be missed if and only if different instructions propagate the tainted data to the exploit point, or there is a different exploit point. We expect that there is a relatively small number of such possible variants for a particular vulnerability, and that the attacker must identify them manually or by static analysis of the vulnerable binary. Naturally, we can apply the same static analysis techniques to preemptively identify the other paths that should be instrumented. This is discussed further in Section 3.3.

Combining filters. We may want to combine several different taint-based VSEF filters. For example, a single binary may have several vulnerabilities that are not all discovered simultaneously. We want to harden the binary as each new vulnerability is discovered. Another example is vulnerabilities that can be exercised via several different code paths. We want to be able to re-harden the binary as each new code path is discovered by the detector.

We combine taint-based VSEF filters by a simple union: any instruction listed in either of the filters should be instrumented. The simplicity and efficiency of combining filters is a nice property for defense systems using our approach since it means the system does not become complex as new vulnerabilities and attackers are discovered.

3.2 Destination-based VSEF

Overview. We next consider an optimistic filter that focuses on instrumenting the point where sensitive data was illegitimately overwritten, rather than the point where tainted data was illegitimately used. Conceptually, a taint-based VSEF filter consists of a chain of data movement operations, and the instruction at the exploit point, which misuses the tainted data. The taint-based VSEF filter detects when the tainted data is misused, which is a very accurate detection method. However, the actual security violation is the data movement instruction in the chain that wrote to an illegitimate destination, copying the tainted data to the overwrite target. We refer to this instruction as the overwrite point. Therefore, we propose destination-based VSEF, which monitors only the overwrite point, *i.e.*, the specific instruction that illegitimately wrote to a specific destination (such as a specific function pointer). We use the term optimistic because of cases where destination-based VSEF may have false positives. Destination-based VSEF is based on the idea that an overwrite attack results in the instruction at the overwrite point writing to a destination that it would not normally write to. This idea is supported by Zhou *et. al.* [44], who built a system that successfully detects many memory faults (and overwrite attacks) by detecting when an instruction writes to a destination that it hasn't written to during normal execution.

It is not enough to specify the overwrite point only by the position of the instruction that performed the overwrite. For example, suppose that the instruction that performed the overwrite was a `mov` inside `memcpy`. Because of a bug in the way `memcpy` was called, it wrote past the end of a buffer and overwrote a sensitive value, such as a function pointer. However, a different call to `memcpy` in another part of the program may be used to intentionally copy legitimate data to the same location. Therefore, we specify the overwrite point as the position of the instruction that performed the overwrite, plus the *context* in which it was executed, which we call the *vulnerable context*. We specify the context to be the list of return addresses on the stack, which indicates the sequence of function calls that led to the exploit.

Destination-based VSEF Filter Generation. To generate a destination-based VSEF filter, the VSEF Filter Generator needs to determine (1) which data movement instruction illegitimately wrote to a sensitive location (the overwrite point), (2) the vulnerable stack configuration when that data movement takes place (the vulnerable context), and (3) what destination(s) should not be overwritten by that instruction, in that context. The VSEF Filter Generator can extract this information from an execution log of a general purpose detector, or use a specialized detector that makes this information immediately available.

To identify the data movement instruction that performed the illegitimate write, the VSEF Filter Generator first identifies the chain of instructions that propagated the tainted data to the exploit point, in the same manner as to generate a taint-based VSEF filter. The VSEF Filter Generator then identifies which of the instructions in that taint propagation chain is the overwrite point.

When available, the VSEF Filter Generator can use debug information compiled into the program to help identify the overwrite point. Debug information can be used to determine the allocated size of a buffer. Hence, for buffer overflows, the VSEF Filter Generator can identify the overwrite point as a data movement instruction that calculates an address as a base plus an offset, where the offset causes the calculated address to point outside of the buffer that the base pointer points to.

Debug information also provides information about the *type* of each memory object. Hence, the VSEF Filter Generator can use this information to identify the overwrite point as the the data movement instruction that caused a type violation, *e.g.*, a string copied over a function pointer. For programs that have not been compiled with debug information, type information can sometimes be inferred at run time. For example, return addresses can be identified for programs that obey normal stack conventions. It is possible to infer the types of other locations based on how the data is used during normal execution [8].

When neither debug information nor type information is available, the VSEF Filter Generator identifies the overwrite point as the last instruction in the propagation chain that writes to a dynamically calculated memory address. Heuristically this will usually be true, given the assumptions that overwrite attacks are the result of such a memory address taking on an unintended value, and that there are not any other such copies that occur between the overwrite point and the exploit point.

Using our previous example in Table 1, any of these techniques correctly identifies the overwrite point as instruction **D**. Using buffer size information: While the base address used at that point points to `dummy->buf`, the offset causes the calculated address to point to `dummy->fnptr`. Using type information: Instruction **D** is the first instruction in the chain where tainted data is written to a data type that should not be tainted. Using neither: Instruction **D** is the last instruction in the chain to write to a dynamically calculated address. Instructions **E** and **G** write to processor registers. Instruction **F** writes to a hard-coded offset within the current stack frame.

Once the overwrite point has been identified, the vulnerable context in which it was executed can be found by examining the calls and returns up to that point in the exploit execution trace. Alternatively, a specialized detector such as TaintCheck can log the call-stack state along with each tainted data propagation, so that the call-stack is already on-hand when the overwrite point is reached in the backwards trace of the exploit execution trace. In our previous example from Table 1, the stack context at the overwrite point (instruction **D**) is $[main + 47, vuln + 68, strcpy + 25]$. That is, the instruction at offset 47 from the start of `main` called `vuln`, the instruction at offset 68 from the start of `vuln` called `strcpy`, and the instruction at offset 25 from the start of `strcpy` is the `mov` that overwrote the function pointer. This example demonstrates why we need to keep track of the vulnerable context, and not just the overwrite point instruction. Here, as in many cases, there is nothing wrong with the instruction at the overwrite point, or even the function it is in (`strcpy`). The problem is that `vuln` called `strcpy` in an unsafe way.

The sensitive value overwritten is the destination operand of the data movement instruction at the overwrite point. We express this location in a robust way in our filter. For example, this can be done by denoting as an offset from an activation record for stack-based locations, or as an offset from a buffer allocated in a certain stack-context for heap-based locations. In the example from Table 1, the location is offset 16 in $dummy$. This is expressed as offset 16 from the buffer allocated at context $[main + 14]$.

In the case of buffer overruns, we would ideally like to specify that the write does not continue past the end of the buffer, so that future exploits against the vulnerability are not able to overwrite data in between the end of the buffer and the data that was detected as being misused. The VSEF Filter Generator can do this if the binary was compiled with debug information (hence the length of the buffer is known). When this information is not available, the VSEF Filter Generator can still sometimes create a tighter bound for what area should not be overwritten. For example, it recognizes when the value overwritten was the return address. Instead of only protecting the return address, it also protects the saved `ebp`, which is adjacent to the return address, and could be overwritten without overwriting the return address.

Destination-based VSEF Binary Instrumentation. We instrument the binary program to check that the data movement instruction at the overwrite point does not write to the sensitive destination when it is in the vulnerable stack context. Our experiments in Section 4.2 show that this can be done by instrumenting a small number of instructions- the data movement instruction, and the call instruction corresponding to each activation record in the vulnerable context. We also show how this could be reduced to only instrumenting the data movement instruction by making copies of each function in the vulnerable context.

Accuracy. When the program is run with the sample exploit, it will again reach the overwrite point, in the vulnerable stack context. At that point, the instrumentation detects that the destination address is illegitimate, signalling an attack.

As with taint-based VSEF filters, exploits that automatically alter their content while using the same attack vector will still be caught. However, it is possible that an attacker could alter the exploit so that the vulnerability is exploited in a different vulnerable context (*i.e.* there may be multiple functions that call the vulnerable function), or so that it overwrites a different sensitive value. There are unlikely to be many such possible variations, and we may be able to find some of them automatically using static analysis. For example, manual analysis of the vulnerable ATPhttpd shows that there are only two contexts in which the vulnerable function is called in an exploitable way.

We expect that most destination-based VSEF filters will have zero false positives. There are a few cases where a destination-based VSEF filter may have false positives, all of which we expect to be very rare. A destination-based VSEF could have false positives if 1) The VSEF Filter Generator identified the wrong instruction as the overwrite point, and hence the write to that address occurs in normal usage. This problem should be straight-forward to detect and fix after using the filter. 2) The instruction at the overwrite point can *legitimately* write to the monitored location in the vulnerable context. This can be true if the source is sometimes a legitimate (non-tainted) value, or if the destination isn't always used in a sensitive way (*e.g.*, a C `union` that could be a function pointer or a string buffer). In this case a low-false-positive destination-based VSEF filter for that vulnerability is not possible, and a taint-based VSEF filter should be used instead.

Combining Filters. It is straightforward to instrument a program with multiple destination-based VSEF filters. The instrumentation for each filter can be added independently of the other instrumentation. In some cases multiple filters will instrument the same instruction. Each filter can add its own instrumentation independently, without interfering with the other.

Performance. Destination-based VSEF allows the filter to be created almost instantaneously. The length of the filter (as well as the total number of instructions instrumented), however, is bound by the depth of the call stack at the overwrite point of sample exploit, plus the address of the overwrite point, plus the identifier of the sensitive data to be guarded. In Section 4.2 we describe how we can instrument even fewer instructions, further improving performance.

3.3 Static analysis extensions

Our adversarial model requires filters be generated quickly, and requires them to be small enough to distribute rapidly. As a result, filter creation for both schemes relies only on information already on-hand when the exploit is detected. However, if we relax the speed requirement we may be able to generate more accurate filters by performing more analysis.

Backward slicing. The filter we create recognizes the sample exploit along with variants polymorphic in the exploit payload. However, an exploit may be polymorphic in the execution path followed. For example, the ATPhttpd webserver vulnerability we investigate can be exploited along two different code paths: one where the requested file is found but not readable and one if the file is not found at all. The destination-based VSEF filter generated from one will not detect the other, because the overwrite occurs in a different vulnerable context. In this case, the taint-based VSEF filter for one *will* detect the other because the same instructions are involved in copying the tainted data in either case. However, if ATPhttpd had been implemented to use `memcpy` to copy the tainted data on one path, and `strcpy` to copy the tainted data on the other path, then the taint-based VSEF filter generated from one path would not detect the other.

One can perform static analysis to recognize these alternate code paths, and identify the additional instructions that would need to be instrumented to detect the corresponding attacks. That is, alternate data propagation paths can be identified and instrumented in taint-based VSEF filters, and alternate vulnerable contexts can be identified and instrumented in destination-based VSEF filters. Note the static analysis is sound but imprecise, so it is possible that more instructions will be instrumented than necessary. However, including instrumentation for potential alternate exploit paths, will result in a filter that detects future exploits polymorphic both in the path taken and in the exploit payload.

4 Implementation & Evaluation

In this section we present our implementation and experimental evaluation of the taint-based and destination-based VSEF Filter Generators and VSEF Binary Instrumentation Engines. In our experiments we use TaintCheck [27] as the Exploit

Detector, and to record the exploit execution trace.

4.1 Taint-based VSEF

4.1.1 Implementation

As discussed in Section 2, TaintCheck already records the information needed to produce a taint-based VSEF filter. As the monitored program is executing it keeps a directed acyclic graph (DAG) that represents how tainted data was propagated, and what instructions propagated it. When an exploit is detected, part of the output is the part of the DAG showing how the misused tainted data was derived. We implemented the taint-based VSEF Filter Generator by modifying TaintCheck to save the set of instruction addresses from that part of the DAG into a separate file, along with the instruction address where the tainted data was misused. This file is the taint-based VSEF filter.

We also implemented the taint-based VSEF Binary Instrumentation Engine as an extension to TaintCheck. Normally TaintCheck adds taint-propagation instrumentation to every instruction that propagates data, which is most instructions. It also adds taint-assertions to every instruction that could potentially misuse tainted data. In our extension, TaintCheck accepts a taint-based VSEF filter as input, and then only adds taint-propagation to the propagation instructions listed in the VSEF filter, and taint-assertion instrumentation to the misuse instruction listed in the VSEF filter.

Note that our current implementation of the taint-based VSEF Binary Instrumentation Engine is intended only as a prototype to show the relative difference between monitoring nearly every instruction, and monitoring only the instructions in the taint-based VSEF filter. However, TaintCheck is currently implemented on Valgrind [24] (for Linux), and DynamoRIO [1] (for Windows). Both of these tools are well suited for when the entire program needs to be monitored, but they each add substantial overhead even when no instrumentation is added. A more efficient implementation could be done using a tool such as Dyninst [2], which is better suited for adding instrumentation to specific points of a program. (We use Dyninst to implement the destination-based VSEF Binary Instrumentation Engine).

4.1.2 Evaluation

We evaluate the quality and efficiency of our taint-based VSEF using real world exploits. We have tested the effectiveness of our taint-based VSEF approach on Windows against the SQL Slammer attack [22], and on Linux against the ATPhttpd exploit [29].

	Avg Time (s)	Overhead
Native	121.4	-
DynamoRIO	135.05	11%
+ Taint-based VSEF filter	138.35	14%

Table 2: SQL taint-based VSEF benchmark.

	Latency (ms)	Overhead
Native	.566	-
Valgrind	1.279	126%
+ Taint-based VSEF filter	1.360	140%
Full TaintCheck	9.797	1631%
Destination-based VSEF	.585	3%

Table 3: ATPhttpd taint-based VSEF and destination-based VSEF benchmark. (1 KB pages)

Taint-based VSEF Filter Size. The filter generated for the ATPhttpd exploit contains only 10 instructions that must be instrumented. The filter for the vulnerability exploited by the SQL Slammer worm contains 200 instructions that must be instrumented. Note that our Windows implementation of taint-based VSEF Filter Generator, which is based on the less mature DynamoRIO implementation of TaintCheck, currently adds *every* instruction that operates on the misused tainted data to the VSEF filter, rather than refining it to only the instructions that actually propagate the tainted data to the point where it is misused. This refinement is straight-forward to implement, and should reduce the filter size by an order of magnitude. For comparison, the ATPhttpd VSEF filter contains 83 instructions without this refinement.

Taint-based VSEF Performance. The time to generate a VSEF and use it to harden a binary is very small. For ATPhttpd it was 186 microseconds to generate a VSEF from TaintCheck's DAG, and 195 ms to use the VSEF to harden the ATPhttpd binary. Here, we measure the performance of the hardened Microsoft SQL Server and the hardened ATPhttpd server. For both tests, we issue queries to the server process from the same machine so as to not introduce network latency.

We subjected the Microsoft SQL server to the benchmark query described in [19]. We measured performance when the server was run natively, and when it was run under DynamoRIO with and without the taint-based VSEF instrumentation. Table 2 shows the results. The instrumentation added by the taint-based VSEF causes the server to run only 14% slower than native, and only 2% slower than running under DynamoRIO alone. Again, implementing the filter refinement step for the Windows version of TaintCheck would reduce the number of instructions instrumented, and further reduce the taint-based VSEF overhead.

We used the Apache Flood tool [39] to measure the performance of the hardened ATPhttpd server when serving 1 KB files. Results are shown in Table 3. Our results show that the hardened server runs only 6% slower than when running under Valgrind alone. We also ran the same benchmark using Valgrind to count how often the instrumented instructions are executed. We found that the 10 instructions instrumented by the taint-based VSEF accounted for only 32,649 of 746,419,783 instructions executed (.00437%). This suggests that implementing the VSEF Binary Instrumentation Engine with more efficient instrumentation techniques (such as DynamoRIO or Dyninst) should result in the taint-based VSEF having very little performance overhead.

Taint-based VSEF Accuracy. We verified that the hardened ATPhttpd and Microsoft SQL server were able to successfully defend against the original exploit. For ATPhttpd, we also created synthetic polymorphic variants of the exploit by replacing the code in the request with randomly generated bytes. We verified that the hardened ATPhttpd successfully detected these modified versions of the exploit, thus demonstrating that our taint-based VSEF approach is effective against polymorphic variants of the sample exploit.

During our benchmarks, neither hardened server had false positives. We also sent the ATPhttpd server several anomalous requests that exercise similar code paths as the exploit, without actually exploiting the server. The hardened ATPhttpd correctly did not identify these as attacks.

4.2 Destination-based VSEF

4.2.1 Implementation

We implemented the destination-based VSEF Binary Instrumentation Engine using Dyninst [2], a binary instrumentation tool. Unlike Valgrind and DynamoRIO, Dyninst performs static rewriting of the target binary. Instructions are instrumented by overwriting them with `jump`s to trampoline functions that call our instrumentation code, and then execute the overwritten instruction before returning. This approach was chosen to avoid the run-time overhead of dynamic binary rewriting. Dyninst and our destination-based VSEF Binary Instrumentation Engine run on both Linux and Windows.

The destination-based VSEF filter consists of the address of the overwrite point, the activation records on the stack when the overwrite point was executed in the original exploit, and the normalized address of the data that was overwritten.

Given the exploit execution trace generated by TaintCheck, the destination-based VSEF filter is generated using the algorithm from Section 3.2 to identify which instruction is the overwrite point, and pulling the rest of the information from the exploit execution trace in a straight-forward manner. We assume the most difficult scenario, in which no debug or type information is available to help identify the overwrite point.

We observe that the overwrite instruction is usually a `mov` instruction, which is usually too small to be overwritten by a `jump` instruction by Dyninst. Dyninst handles this case by instead overwriting it with a 1 byte instruction to generate a trap, which causes the operating system to deliver a signal to the process, and the instrumentation code to be executed by the signal handler.[1] This is undesirable, since this is a relatively expensive process. We observe that in many cases, the instrumented `mov` is called frequently (*i.e.*, it may be in `strcpy`), but usually not in the vulnerable context. Therefore we address this problem by only having the instrumentation be used when the function is called in the vulnerable context. The most efficient way to do this is by copying the functions that make up the vulnerable context, and rewriting the corresponding `call` instructions so that the instrumented `mov` is only used in the vulnerable context. In cases where this is infeasible, we can dynamically enable or disable the `mov` instrumentation when the vulnerable context is entered or left.

We currently implement the latter approach. We implemented the VSEF Binary Instrumentation Engine to instrument the `call` instruction corresponding to each activation record in the vulnerable context. This instrumentation incrementally tracks which of the activation records of the vulnerable context are currently on the stack. The instrumentation for the last `call` of the vulnerable context dynamically adds or removes the instrumentation at the overwrite point when the vulnerable context is entered or left. Note that if we instrumented only this `call` instead of each `call` in the vulnerable context, the instrumentation would need to walk the stack every time that `call` was executed to see if it was in the vulnerable context, which would result in a higher performance cost.

The instrumentation at the overwrite point checks whether the instruction is about to write to the protected location. If so, an attack is detected.

[1] Dyninst version 5, which is currently under development, uses a different method to insert instrumentation which should mostly eliminate the need to use traps. Unfortunately, we were not able to test this version at the time of writing.

4.2.2 Evaluation

We evaluate the quality and efficiency of our destination-based VSEF using the ATPhttpd exploit.[2]

Destination-based VSEF Filter Size. The filter generated for the ATPhttpd vulnerability consists of the addresses of 12 instructions (the `mov` that causes the overwrite, and the 11 *call* instructions corresponding to the vulnerable context), and a range of offsets from the vulnerable stack frame to protect. The ATPhttpd exploit overwrote the return address, so in this case we are protecting the return address, which is located at offsets 4 to 7 in the vulnerable stack frame. (In our implementation, we recognize this case and extend the range to 0 to 7 to also protect the frame pointer). To clarify, if we were protecting data inside the stack frame (such as a local variable storing a function pointer), this offset would be negative.

Destination-based VSEF Performance. It takes a negligible amount of time to create a destination-based VSEF filter from TaintCheck's log, and to use the destination-based VSEF Binary Instrumentation Engine to harden the vulnerable binary. Here, we measure the performance of the hardened ATPhttpd server.

As in Section 4.1, we evaluate the performance of the hardened ATPhttpd server using the Apache Flood tool to measure the time to serve requests for 1 KB files. Our results are shown in Table 3. Our results show that the server runs only 3% slower than when the server is run without instrumentation.

We also used Valgrind to count how often the instrumented instructions are executed during the benchmark. The 12 instrumented instructions accounted for 6,070 of 746,465,052 instructions executed(.000813%).

Destination-based VSEF Accuracy. We verified that the hardened ATPhttpd server was able to successfully defend against the original exploit. As in the taint-based VSEF experiment, we also verified that server was able to defend against polymorphic variations of the exploit, and that it correctly did not identify similar but non-exploiting requests as attacks.

5 Deployment and Applications

Vulnerability-specific execution filtering meets three important goals: fast filter generation, accurate detection, and low performance overhead. These requirements address the most relevant threat to today's Internet infrastructure: fast spreading worms. Worms that exploit known vulnerabilities can cause millions of dollars

[2] At the time of writing, the Windows implementation of TaintCheck does not log the correct information to create a destination-based VSEF, so we were unable to evaluate our destination-based VSEF for the Microsoft SQL server exploit. However, doing so would be straight-forward.

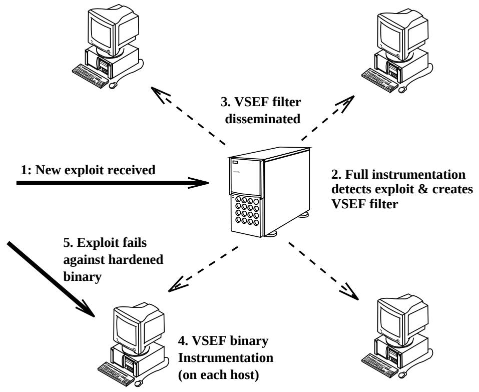
3. VSEF filter disseminated
1: New exploit received
2. Full instrumentation detects exploit & creates VSEF filter
5. Exploit fails against hardened binary
4. VSEF binary Instrumentation (on each host)

Figure 2: The deployment scenario for vulnerability-specific execution filtering. Upon (1) receiving an exploit of a new vulnerability, the (2) full instrumentation engine detects it and creates an appropriate filter. The filter is (3) disseminated to all hosts, which then (4) use the filter to instrument and produce a hardened binary. The hardened binary cannot be then exploited (5). Note that the exploit in step 5 may be a polymorphic variant of step 1.

of damage. A worm exploiting an unknown vulnerability could be much more devastating.

Figure 2 shows our envisioned architecture for defending against worms. Various full instrumentation detectors are placed across the Internet, for example on honeypots or over-provisioned sites. When a new worm is released, the full instrumentation version detects the exploit and uses the VSEF Filter Generator to create an execution filter. The filter is then distributed to other vulnerable hosts across the Internet (that have the same vulnerable binary, similar shared libraries, etc.), which use VSEF Binary Instrumentation Engine to harden their binaries against subsequent infection. This hardening can be done without restarting the server for destination-based VSEF, because Dyninst is able to attach to an already running program and instrument it without restarting. Taint-based VSEF could also be implemented using Dyninst, which would also allow it to harden the program binary without needing to restart the program.

Our architecture provides for completely automatic response and containment, and therefore can respond to a rapid worm outbreak. Our system also works for previously unknown vulnerabilities where the hardened binary can be used until a proper patch can be installed. We note that sites may be unmotivated to install automatically generated network filters with suspect accuracy. The accuracy of our filters make automatic installation much more attractive.

Our techniques and architecture also apply to other adversarial models. Host-based privilege escalation attacks are a serious threat that previous automatic defense systems have mostly ignored. Our scheme can be used to harden known vulnerable programs against such attacks until the proper patch can be applied. Note this is especially important for legacy systems where source code for the running applications may no longer exist or be accessible and thus a permanent patch may never be created.

We present a distributed architecture for efficiently and securely generating, using, and sharing VSEF filters in [25].

6 Related work

Sidiroglou et. al. proposed selective emulation as part of a reactive approach for handling software failure [33]. Their selective emulation is similar in some aspects to our work. Like us, they note that partial instrumentation can reduce total monitoring overhead. However their approach for defending against buffer overflow attacks requires source code to instrument the binary, since it is based on a canary as in StackGuard [12]. In addition, their instrumentation is at function call granularity, and they use heuristics to find out what function calls need to be instrumented. They leave as an open problem how to determine more precise instrumentation, which we solve by using taint-based analysis.

Rinard et. al. has proposed using compiler extensions to deal with writes to unallocated memory. The approach allows a program to execute even in the presence of buffer overflow attacks[30]. These techniques are aimed at increasing availability for services and are not necessarily safe and thus inappropriate as a defense mechanism.

Shield [40] provides vulnerability-specific exploit generic protection. However, it uses manually generated signatures.

Costa et. al. propose a concurrent work to automatically generated *host-based* input filters [11], which has greater accuracy than network-based input filters, and can correctly recognize some semantically equivalent inputs. However, the approach still suffers difficulty when the correct classification rule is complex or needs application state, or when input is encrypted.

IntroVirt [15] uses vulnerability-specific predicates to detect when a vulnerability has been exploited. However, these predicates are manually generated.

DAKODA [13] provides a quantitative analysis for a number of exploit vectors. Their results show that network-based filters are not specific enough for exploits against many vulnerabilities, and that there are a number of vulnerabilities where the attack vector is encrypted, making host-based input filters impractical. The paper also noted that return addresses are not suited to be used as signatures for polymorphic worms which were used in several existing automatic signature generation methods [27, 20, 42].

We benefit directly from the active research for increasing the efficiency of emulation [21, 41, 2]. For example, we use Valgrind and DynamoRIO for taint-based instrumentation (on Linux and Windows, respectively), while Pin reports emulation speeds 3.3x faster than Valgrind and 2x faster than DynamoRIO [21].

We use TaintCheck [27] to initially discover unknown vulnerabilities. Other fine-grained dynamic bug detection tools could be used during initial filter creation, such as program shepherding [17], libsafe [4, 6], or Nethercote-Fitzhardinge bounds checking [23]. We chose TaintCheck because the taint-based approach detects the widest variety of attacks and is easy to augment to produce the taint log needed for taint-based VSEF.

Slicing techniques [38, 43] can be used to help create or refine the VSEF filters, as discussed in Section 3.3. We plan to investigate this approach in the future.

7 Conclusion

We propose vulnerability-specific execution filtering (VSEF), a new type of filter that recognizes and filters out execution patterns of an exploit exercising a known vulnerability. VSEF is more accurate than input filtering, and significantly faster than full execution monitoring. We give two types of VSEF filters: taint-based VSEF and destination-based VSEF. The former is more accurate while the latter may require less instrumentation. We show how to automatically create both filters using a VSEF Filter Generator. The filters can then be used to automatically harden a binary against the vulnerability via the VSEF Binary Instrumentation Engine. We provide an implementation for both components under Windows and Linux, and run experiments that confirm the accuracy, performance, and generation speed. In most cases the overhead of VSEF binary hardening is only a few percent.

8 Acknowledgments

We would like to thank the following people: Jad Chamcham, for implementing TaintCheck on DynamoRIO [10]; Xeno Kovah, for help running experiments; Drew Bernat, for feedback and assistance with using Dyninst; Timothy Wong; Emery Berger; and the anonymous reviewers for their insightful feedback.

References

[1] Dynamorio. `http://www.cag.lcs.mit.edu/dynamorio/`.

[2] Dyninst. `www.dyninst.org`.

[3] Metasploit. `http://www.metasploit.org`.

[4] Kumar Avijit, Prateek Gupta, and Deepak Gupta. Tied, libsafeplus: Tools for runtime buffer overflow protection. In *USENIX Security Symposium*, August 2004.

[5] R.M. Balzer. EXDAMS - extendable debugging and monitoring system. *Proceedings of the AFIPS SJCC*, 34:567–586, 1969.

[6] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *USENIX Annual Technical Conference 2000*, 2000.

[7] P Bosch, A Carloganu, and D Etiemble. Complete x86 instruction trace generation from hardware bus collect. In *23rd IEEE EUROMICRO Conference*, 1997.

[8] Michael Burrows, Stephen N. Freund, and Janet L. Wiener. Run-time type checking for binary programs. In *International Conference on Compiler Construction*, April 2003.

[9] CERT/CC. CERT/CC statistics 1988-2005. `http://www.cert.org/stats/cert_stats.html`.

[10] Jad Chamcham. Dynamic taint analysis: Protecting Windows against worms and zero-day attacks. Master's Thesis, Carnegie Mellon University, 2005.

[11] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-end containment of internet worms. In *Proceedings of the twentieth ACM symposium on Operating systems principles (SOSP)*, October 2005.

[12] Crispin Cowan, Calton Pu, Dave Maier, Jonathon Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, January 1998.

[13] Jedidiah Crandall, Zhendong Su, S. Felix Wu, and Frederic Chong. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *Proc. 12th ACM Conference on Computer and Communications Security (CCS)*, 2005.

[14] Jedidiah R. Crandall and Fred Chong. Minos: Architectural support for software security through control data integrity. In *To appear in International Symposium on Microarchitecture*, December 2004.

[15] Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the twentieth ACM symposium on Operating systems principles (SOSP)*, October 2005.

[16] Hyang-Ah Kim and Brad Karp. Autograph: toward automated, distributed worm signature detection. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[17] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.

[18] Christian Kreibich and Jon Crowcroft. Honeycomb - creating intrusion detection signatures using honeypots. In *Proceedings of the Second Workshop on Hot Topics in Networks (HotNets-II)*, November 2003.

[19] Gregory Larsen. Benchmarking performance of a query - part 1 elapsed time. `http://www.databasejournal.com/features/mssql/article.php/3298411`, 2004.

[20] Zhenkai Liang and R. Sekar. Fast and automated generation of attack signatures: A basis for building self-protecting servers. In *Proc. of the 12th ACM Conference on Computer and Communications Security (CCS)*, 2005.

[21] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation (PLDI)*, 2005.

[22] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the slammer worm. In *IEEE Security and Privacy*, volume 1, 2003.

[23] Nicholas Nethercote and Jeremy Fitzhardinge. Bounds-checking entire programs without recompiling. In *Proceedings of the Second Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE 2004)*, Venice, Italy, January 2004. (Proceedings not formally published.).

[24] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. In *Proceedings of the Third Workshop on Runtime Verification (RV'03)*, Boulder, Colorado, USA, July 2003.

[25] James Newsome, David Brumley, Dawn Song, Mark R. Pariente, and Theocharis Kampouris. Efficient and effective self-healing for defending against exploit attacks on commodity software. Technical Report CMU-CS-05-191, Department of Computer Science, Carnegie Mellon University, May 2005.

[26] James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2005.

[27] 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 (NDSS)*, February 2005.

[28] Feng Qin, Shan Lu, and Yuanyuan Zhou. SafeMem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, 2005.

[29] Yann Ramin. ATPhttpd. http://www.redshift.com/∼yramin/atp/atphttpd/.

[30] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel Roy, Tudor Leu, and William Beebee Jr. Enhancing server availability and security through failure-oblivious computing. In *Operating System Design & Implementation (OSDI)*, 2004.

[31] Tim J Robbins. libformat. `http://www.securityfocus.com/tools/1818`, 2001.

[32] P. A. Sandon, Y.C. Liao, T.E. Cook, D.M. Schultz, and P Martin de Nicolas. Nstrace: A bus-driven instruction trace tool for powerpc microprocessors. *IBM Journal of Research and Development*, 41(3), 1997.

[33] 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*, 2005.

[34] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. The EarlyBird system for real-time detection of unknown worms. Technical Report CS2003-0761, University of California, San Diego, August 2003.

[35] Stuart Staniford, Vern Paxson, and Nicholas Weaver. How to 0wn the Internet in your spare time. In *11th USENIX Security Symposium*, 2002.

[36] G. Edward Suh, Jaewook Lee, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of ASPLOS*, 2004.

[37] Peter Szor. Hunting for metamorphic. In *Virus Bulletin Conference*, 2001.

[38] Frank Tip. A survey of program slicing techniques. *Journal of programming languages,*, 3, September 1995.

[39] Apache Flood Tool. `http://httpd.apache.org/test/flood`.

[40] Helen J Wang, Chuanxiong Guo, Daniel Simon, and Alf Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *ACM SIGCOMM*, August 2004.

[41] Chadd Williams and Jeffrey Hollingsworth. Interactive binary instrumentation. In *Second International Workshop on Remote Analysis and Measurement of Software Systems (RAMSS)*, 2004.

[42] Jun Xu, Peng Ning, Chongkyung Kil, Yan Zhai, and Chris Bookholt. Automatic diagnosis and response to memory corruption vulnerabilities, 2005.

[43] Xiangyu Zhang and Rajiv Gupta. Cost effective dynamic program slicing. In *2004 Programming Language Design and Implementation (PLDI) conference*, 2004.

[44] Pin Zhou, Wei Liu, Fei Long, Shan Lu, Feng Qin, Yuanyuan Zhou, Sam Midkiff, and Josep Torrellas. AccMon: Automatically detecting memory-related bugs via program counter-based invariants. In *The Proceedings of 37th Annual IEEE/ACM International Symposium on Micro-architecture (Micro'04)*, December 2004.