Sting: An End-to-End Self-Healing System for Defending against Internet Worms

David Brumley[1], James Newsome[2], and Dawn Song[3]

[1] Carnegie Mellon University, Pittsburgh, PA, USA dbrumley@cs.cmu.edu
[2] Carnegie Mellon University, Pittsburgh, PA USA jnewsome@ece.cmu.edu
[3] Carnegie Mellon University, Pittsburgh, PA, USA dawnsong@cmu.edu

1 Introduction

We increasingly rely on highly available systems in all areas of society, from the economy, to military, to the government. Unfortunately, much software, including critical applications, contains vulnerabilities unknown at the time of deployment, with memory-overwrite vulnerabilities (such as buffer overflow and format string vulnerabilities) accounting for more than 60% of total vulnerabilities [12]. These vulnerabilities, when exploited, can cause devastating effects, such as self-propagating worm attacks which can compromise millions of vulnerable hosts within a matter of minutes or even seconds [33, 59], and cause millions of dollars of damage [31]. Therefore, we need to develop effective mechanisms to protect vulnerable hosts from being compromised and allow them to continue providing critical services, even under aggressively spreading attacks on previously unknown vulnerabilities.

We need *automatic* defense techniques because manual response to new vulnerabilities is slow and error prone. Fast reaction is important because previously unknown ("zero-day") or unpatched vulnerabilities can be exploited orders of magnitude faster than a human can respond by worms [9, 59]. Automatic techniques have the potential to be more accurate than manual efforts because vulnerabilities exploited by worms tend to be complex and require intricate knowledge of details such as realizable program paths and corner conditions. Understanding the complexities of a vulnerability has consistently proven very difficult and time consuming for humans at even the source code level [11], let alone COTS software at the assembly level.

Overview and Contributions. By carefully uniting a suite of new techniques, we create a new end-to-end self-healing architecture, called *Sting*, as a first step towards automatically defending against fast Internet-scale worm attacks.

At a high level, the Sting self-healing architecture enables programs to efficiently and automatically (1) self-monitor their own execution behavior to detect a large class of errors and exploit attacks, (2) self-diagnose the root cause of an error or exploit attack, (3) self-harden to be resilient against further attacks, and (4) quickly

self-recover to a safe state after a state corruption. Furthermore, once a Sting host detects and diagnoses an error or attack, it can generate a verifiable *antibody*, which is then distributed to other vulnerable hosts, who verify the correctness of the antibody and use it to self-harden against attacks on that vulnerability. We provide a more detailed overview below.

First, we propose dynamic taint analysis to detect new attacks, and to provide information about discovered attacks which can be used to automatically generate antibodies that protect against further attacks on the corresponding vulnerability. Dynamic taint analysis monitors software execution at the instruction level to track what data was derived from untrusted sources, and detect when untrusted data is used in ways that signify that an attack has taken place. This technique reliably detects a large class of exploit attacks, and does not require access to source code, allowing it to be used on commodity software. This work is described in detail in [44, 45].

Once a new attack is detected, there are several types of antibodies that can be generated, and several methods to generate them. We have investigated automatic methods of generating input-filters by finding common byte-patterns in collected worm samples, even for polymorphic worms. This work is described in detail in [42]. However, we have found that a worm author can severely cripple such methods by including spurious features in samples of the worm [43].

In [10], we propose *vulnerability*-based signatures, in which signatures are created based upon the vulnerability itself. Vulnerability signatures are input signatures which provably have zero-false positives (or false negatives, if desired). Therefore, vulnerability signatures are appropriate even in an adversarial environment where malicious parties may try to mislead the signature creation algorithm.

In some circumstances input-based filters may not be practical. For example, performance requirements may only allow for token-based signatures, but token-based signatures may be too imprecise to be useful. Therefore, we propose an alternative of automatically generating *execution* filters, which are specifications of where the vulnerability lies in the vulnerable program. These are used to automatically insert a small piece of instrumentation into the vulnerable program, which in turn allows the vulnerable program to efficiently and reliably detect when that vulnerability is exploited. This work is described in [40].

Once a new attack has been found, and an antibody generated for that attack, we disseminate that antibody to other vulnerable hosts. These vulnerable hosts can verify both that an attack exists and that the antibody successfully stops it by *replaying* the attack against the antibody-protected software in a confined environment.

Finally, we integrate the above techniques to form Sting, an end-to-end self-healing system for defending against zero-day worm attacks on commodity software. In this system, users use light-weight detectors (such as address randomization [3, 7, 8, 13, 22, 23, 66]) and random sampling to initially detect new attacks with little performance cost. When a potential attack is detected, we then use dynamic taint analysis to perform automatic self-diagnosis, which verifies whether it is truly an attack, and automatically generates an execution filter. That execution filter is used to harden the vulnerable binary, and is distributed to others running the vulnerable software to allow them to also harden their own vulnerable binaries. When

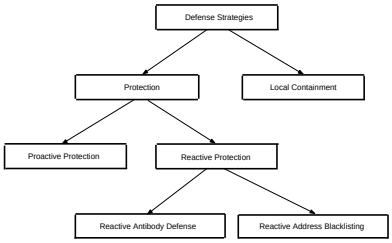


Fig. 1. Worm Defense Strategy Taxonomy

an exploit is detected, the system performs diagnosis-directed self-recovery using process checkpointing and recovery [58, 46].

Organization. In Section 2, we briefly describe the design space for worm defense systems. Our analysis indicates that the best designs incorporate both a proactive protection component and a reactive antibody component. This analysis motivates our Sting architecture. We then describe TaintCheck in Section 3, which is one of the primary mechanism we use to detect new exploits and vulnerabilities. In Section 4, we discuss automatic input-based signature creation. We show many proposed algorithms are fragile and can be mislead by an adversary into creating incorrect signatures. We then describe a new class of signatures called vulnerability signatures which are provably correct, even in an adversarial environment. In Section 5, we describe an alternative to input-based filters called vulnerability-based execution filters (VSEF). Section 6 describes the complete Sting architecture and our experiences creating it. We then present related work, and conclude.

2 Worm Defense Design Space

The design space for worm defense systems is vast. For example, should a worm defense system try to contain infected machines from further propagation of the worm, blacklist known infected hosts, or filter infection attempts? In [9], we propose a taxonomy for worm defense strategies and perform theoretical and experimental evaluation to compare different strategies in the design space. Our analysis shows a hybrid scheme using proactive protection and a reactive antibody defense is the most promising approach. Thus, we adopt this strategy in the Sting architecture.

2.1 Defense Strategy Taxonomy

We analyze a taxonomy of possible solutions in the worm defense design space in [9]. The taxonomy is depicted in Figure 1. At a high level, the four defense strategies are:

Reactive Antibody Defense. This approach reactively generates an *antibody*, which is a protective measure that prevents further infections. The scheme is reactive

because the antibody is created based upon a known worm sample. Many input-based filtering schemes such as in Section 4 and [28, 30, 42] are examples of a reactive antibody defense since the input filters are created from known worm samples. Vulnerability-specific execution filters (Section 5) are another example.

Proactive Protection. A proactive protection scheme is always in place and prevents at least some worm infection attempts from succeeding. Running TaintCheck on all programs, all the time is an example of a proactive protection scheme. However, running TaintCheck all the time is unrealistic due to the potentially high overhead. An example of a *probabilistic* proactive protection is address space randomization [3, 7, 8, 13, 22, 23, 66], in which each infection attempt succeeds with some probability p.

Reactive Address Blacklisting. Blacklisting generates a worm defense based upon the address of an attacking host. For example, filtering any subsequent connections from a known infected host [35].

Local Containment. Local containment is a "good neighbor" strategy in which a site filters outgoing infection attempts to other sites. Scan rate throttling schemes such [62, 65] are an example of this strategy.

2.2 The Sting Architecture

We show in [9] that the most effective strategy in a realistic setting is combining proactive protection with a reactive antibody defense. The intuition is that proactive protection will slow down the initial worm outbreak, which allows time to develop and deploy a permanent antibody.

Sting is designed around the hybrid proactive protection with reactive antibody defense. Sting utilizes TaintCheck, address space randomization, and random sampling as proactive protection mechanisms. The combination of these mechanisms provides efficient probabilistic protection. Sting develops verifiable antibodies which can be distributed and installed. The antibodies provide efficient protection against subsequent infections.

3 Dynamic Taint Analysis for Automatic Detection of New Exploits

Many approaches have been proposed to detect new attacks. These approaches roughly fall into two categories: *coarse-grained detectors*, that detect anomalous behavior, such as scanning or unusual activity at a certain port; and *fine-grained detectors*, that detect attacks on a program's vulnerabilities. While coarse-grained detectors are relatively inexpensive, they can have frequent false positives, and do not provide detailed information about the vulnerability and how it is exploited. Thus, it is desirable to develop fine-grained detectors that produce fewer false positives, and provide detailed information about the vulnerability and exploit.

Several approaches for fine-grained detectors have been proposed that detect when a program is exploited. Most of these previous mechanisms require source code

or special recompilation of the program, such as StackGuard [18], PointGuard [17], full-bounds check [26, 51], LibsafePlus [5], FormatGuard [16], and CCured [37]. Some of them also require recompiling the libraries [26, 51], or modifying the original source code, or are not compatible with some programs [37, 17]. These constraints hinder the deployment and applicability of these methods, especially for commodity software, because source code or specially recompiled binaries are often unavailable, and the additional work required (such as recompiling the libraries and modifying the original source code) makes it inconvenient to apply these methods to a broad range of applications. Note that most of the large-scale worm attacks to date are attacks on commodity software.

Thus, it is important to design fine-grained detectors that work on commodity software, *i.e.*, work on arbitrary binaries without requiring source code or specially recompiled binaries. This goal is difficult to achieve because important information, such as data types, is not generally available in binaries. As a result, existing exploit detection mechanisms that do not use source code or specially compiled binary programs, such as LibSafe [6], LibFormat [50], Program Shepherding [29], and the Nethercote-Fitzhardinge bounds check [38], are typically tailored for narrow types of attacks and fail to detect many important types of common attacks.

We propose a new approach, *dynamic taint analysis*, for the automatic detection of exploits on commodity software. In dynamic taint analysis, we label data originating from or arithmetically derived from untrusted sources such as the network as *tainted*. We keep track of the propagation of tainted data as the program executes (*i.e.*, what data in memory is tainted), and detect when tainted data is used in dangerous ways that could indicate an attack. This approach allows us to detect *overwrite attacks*, attacks that cause a sensitive value (such as return addresses, function pointers, format strings, *etc.*) to be overwritten with the attacker's data. Most commonly occurring exploits fall into this class of attacks. We have developed an automatic tool, *TaintCheck*, to demonstrate our dynamic taint analysis approach.

3.1 Dynamic Taint Analysis

Our technique is based on the observation that in order for an attacker to change the execution of a program illegitimately, he must cause a value that is normally derived from a trusted source to instead be derived from his own input. For example, values such as return addresses, function pointers, and format strings should usually be supplied by the code itself, not from external untrusted inputs. In an *overwrite attack*, an attacker exploits a program by overwriting sensitive values such as these with his own data, allowing him to arbitrarily change the execution of the program.

We refer to data that originates or is derived arithmetically from an untrusted input as being *tainted*. In our dynamic taint analysis, we first mark input data from untrusted sources tainted, then monitor program execution to track how the tainted attribute propagates (*i.e.*, what other data becomes tainted) and to check when tainted data is used in dangerous ways. For example, use of tainted data as a function pointer

or a format string indicates an exploit of a vulnerability such as a buffer overrun or format string vulnerability [4], respectively.

Note that our approach detects attacks at the time of *use*, *i.e.*, when tainted data is used in dangerous ways. This significantly differs from many previous approaches which attempt to detect when a certain part of memory is illegitimately overwritten by an attacker at the time of the *write*. Without source code, it is not always possible at the time of a write to detect whether an illegitimate overwrite is taking place, because it cannot always be statically determined what kind of data is being overwritten, *e.g.* whether the boundary of a buffer has been exceeded. Hence, techniques that detect attacks at the time of write without source code are only applicable to certain type of attacks and/or suffer from limited accuracy. However, at the time that data is *used* in a sensitive way, such as as a function pointer, we know that if that data is tainted, then a previous write was an illegitimate overwrite, and an attack has taken place. By detecting attacks at the time of use instead of the time of write, we reliably detect a broad range of overwrite attacks.

3.2 Design and Implementation Overview

We have designed and implemented TaintCheck, a new tool for performing dynamic taint analysis. TaintCheck performs dynamic taint analysis on a program by running the program in its own emulation environment. This allows TaintCheck to monitor and control the program's execution at a fine-grained level. We have two implementations of TaintCheck: we implemented TaintCheck using Valgrind [39]. Valgrind is an open source x86 emulator that supports extensions, called *skins*, which can instrument a program as it is run.[5] We also have a Windows implementation of TaintCheck that uses DynamoRIO [1], another dynamic binary instrumentation tool. For simplicity of explanation, for the remainder of this section, we refer to the Valgrind implementation unless otherwise specified.

Whenever program control reaches a new basic block, Valgrind first translates the block of x86 instructions into its own RISC-like instruction set, called *UCode*. It then passes the UCode block to TaintCheck, which instruments the UCode block to incorporate its taint analysis code. TaintCheck then passes the rewritten UCode block back to Valgrind, which translates the block back to x86 code so that it may be

[4] Note that the use of tainted data as a format string often indicates a format string *vulnerability*, whether or not there is an actual exploit. That is, the program unsafely uses untrusted data as a format string (`printf(user_input)` instead of `printf(``%s'', user_input)`), though the data provided by a particular user input may be innocuous.

[5] Note that while Memcheck, a commonly used Valgrind extension, is able to assist in debugging memory errors, it is not designed to detect attacks. It can detect some conditions relevant to vulnerabilities and attacks, such as when unallocated memory is used, when memory is freed twice, and when a memory write passes the boundary of a `malloc`-allocated block. However, it does not detect other attacks, such as overflows within an area allocated by one `malloc` call (such as a buffer field of a struct), format string attacks, or stack-allocated buffer overruns.

executed. Once a block has been instrumented, it is kept in Valgrind's cache so that it does not need to be re-instrumented every time it is executed.

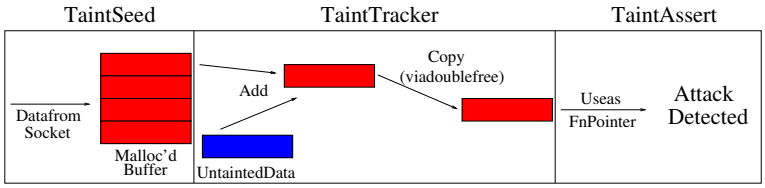


Fig. 2. TaintCheck detection of an attack. (Exploit Analyzer not shown).

To use dynamic taint analysis for attack detection, we need to answer three questions: (1) What inputs should be tainted? (2) How should the taint attribute propagate? (3) What usage of tainted data should raise an alarm as an attack? To make TaintCheck flexible and extensible, we have designed three components: *TaintSeed, TaintTracker,* and *TaintAssert* to address each of these three questions in turn. Figure 2 shows how these three components work together to track the flow of tainted data and detect an attack. Each component has a default policy and can easily incorporate user-defined policies as well. In addition, each component can be configured to log information about taint propagation, which can be used by the fourth component we have designed, the *Exploit Analyzer*. When an attack is detected, the Exploit Analyzer performs post-analysis to provide information about the attack, including identifying the input that led to the attack, and semantic information about the attack payload. This information can be used to automatically generate antibodies against the attack, including input-based filters (Section 4) and execution filters (Section 5).

4 Automatic Generation of Input-based Filters

We first describe previous attempts at automatically generating signatures by syntax pattern-extraction techniques. These techniques find and create signatures based on syntactic differences between exploits and benign inputs. Our experience shows these methods are fragile, and thus not suitable in an adversarial environment where an adversary may try to mislead the signature generation algorithm. We then introduce *vulnerability signatures*, which produce signatures with zero false positives (even in an adversarial setting). In addition, vulnerability signatures are generally of a higher quality (i.e., more accurate and less fragile) than signatures generated by syntax pattern-extraction techniques.

4.1 Limitations of Pattern-Extraction based techniques

First generation worms: identical byte strings. Motivated by the slow pace of manual signature generation, researchers have recently given attention to *automating*

the generation of signatures used by IDSes to match worm traffic. Systems such as Honeycomb [30], Autograph [28], and EarlyBird [56] monitor network traffic to identify novel Internet worms, and produce signatures for them using *pattern-based* analysis, *i.e.*, by extracting common byte patterns across different suspicious flows.

These systems all generate signatures consisting of a *single, contiguous substring* of a worm's payload, of *sufficient length* to match only the worm, and not innocuous traffic. The shorter the byte string, the greater the probability it will appear in some flow's payload, regardless of whether the flow is a worm or innocuous. These syntax pattern-extraction signature generation systems all make the same underlying assumption: that there exists a single payload substring that will remain *invariant* across worm connections, and will be sufficiently unique to the worm such that it can be used as a signature without causing false positives.

Second generation worms: polymorphism. Regrettably, the above payload invariance assumption is naïve, and gives rise to a critical weakness in these previously proposed signature generation systems. A worm author may craft a worm that substantially changes its payload on every successive connection, and thus evades matching by any single substring signature that does not also occur in innocuous traffic. *Polymorphism* techniques[6], through which a program may encode and re-encode itself into successive, different byte strings, enable production of changing worm payloads. It is pure serendipity that worm authors thus far have not chosen to render worms polymorphic; virus authors do so routinely [36, 61]. The effort required to do so is trivial, given that libraries to render code polymorphic are readily available [2, 20].

In Polygraph [42], we showed that for many vulnerabilities, there are several invariant byte strings that must be present to exploit that vulnerability. While using a single one of these strings would not be specific enough to generate an accurate signature, they can be combined to create an accurate *conjunction* signature, *subsequence* signature, or *Bayes signature*. We proposed algorithms that automatically generate accurate signatures of these types, for *maximally varying* polymorphic worms. That is, we assumed the worm minimized commonality between each instance, such that only the invariant byte strings necessary to trigger the vulnerability were present.

Third generation worms: Attacks on learning. The maximal variation model of a polymorphic worm's content bears further scrutiny. If one seeks to understand whether a worm can vary its content so widely that a particular signature type, *e.g.,* one comprised of multiple disjoint substrings, cannot sufficiently discriminate worm instances from innocuous traffic, this model is appropriate, as it represents a worst case, in which as many of a worm's bytes vary randomly as possible. But the maximally varying model is one of many choices a worm author may adopt. Once a worm author knows the signature generation algorithm in use, he may adopt payload variation strategies chosen specifically in an attempt to defeat that algorithm or class of algorithm. Thus, maximal variation is a distraction when assessing the robustness of

[6] We refer to both polymorphism and metamorphism as polymorphism, in the interest of brevity.

a signature generation algorithm in an adversarial environment; some other strategy may be far more effective in causing poor signatures (*i.e.,* those that cause many false negatives and/or false positives) to be generated .

In *Paragraph* [43], we demonstrated several attacks that make the problem of automatic signature generation via pattern-extraction significantly more difficult. The approach taken by pattern-extraction based signature generation systems such as Polygraph is to find common byte patterns in samples of a worm, and then apply some type of *learning* algorithm to generate a classifier, or signature. Most research in machine learning algorithms is in the context in which the content of samples is determined randomly, or even by a *helpful teacher,* who constructs examples in an effort to assist learning.

However, learning algorithms, when applied to polymorphic worm signature generation, attempt to function with examples provided by a *malicious teacher*. That is, a clever worm author may manipulate the particular features found in the worm samples, innocuous samples, or both—not to produce maximal variation in payload, but to *thwart learning itself.*

We demonstrate this concept in Paragraph [43] by constructing attacks against the signature generation algorithms in Polygraph [42]. We have shown that these attacks are practical to perform, and that they prevent an accurate signature from being generated quickly enough to prevent wide-spread infection. From our analysis, we conclude that generating worm signatures purely by syntax pattern-extraction techniques seems limited in robustness against a determined adversary.

4.2 Automatic Vulnerability Signature Generation

A realistic signature generation mechanism should succeed in an adversarial environment without requiring assumptions about the amount of polymorphism an unknown vulnerability may have. Thus, to be effective, the signature should be constructed based on the property of the vulnerability, instead of an exploit (this observation has been made by others as well [64]).

We show that signatures with zero false positives, even in an adversarial setting, can be created by analyzing the vulnerability itself. We call these signatures *vulnerability signatures* [10]. Vulnerability signatures are provably correct with respect to the goal of the administrator: they are constructed with zero false positives or zero false negatives *regardless of how the attacker may try and deceive the generation algorithm*.

Requirements for Vulnerability Signature Generation

We motivate our work and approach to vulnerability signatures in the following setting: a new exploit is just released for an unknown vulnerability. A site has detected the exploit through some means such as dynamic taint analysis (Section 3), and wishes to create a signature that recognizes any further exploits. The site can furnish our analysis with the tuple $\{\mathcal{P}, T, x, c\}$ where $\mathcal{P}$ is the program, x is the exploit string, c is a vulnerability condition, and T is the execution trace of $\mathcal{P}$ on x. Since

our experiments are at the assembly level, we assume $\mathcal{P}$ is a binary program and T is an instruction trace, though our techniques also work at the source-code level. Our goal is to create a vulnerability signature which will *match* future malicious inputs x' by examining them without running $\mathcal{P}$.

Vulnerability Signature Definition

A *vulnerability* is 2-tuple $(\mathcal{P}, c)$, where $\mathcal{P}$ is a program (which is a sequence of instructions $\langle i_1, \cdots, i_k \rangle$), and c is a vulnerability condition (defined formally below). The execution trace obtained by executing a program $\mathcal{P}$ on input x is denoted by $T(\mathcal{P}, x)$. An execution trace is simply a sequence of actual instructions that are executed. A vulnerability condition c is evaluated on an execution trace T. If T satisfies the vulnerability condition c, we denote it by $T \models c$. The *language* of a vulnerability $L_{\mathcal{P},c}$ consists of the set of all inputs x to a program $\mathcal{P}$ such that the resulting execution trace satisfies c. Let Σ^* be the domain of inputs to $\mathcal{P}$. Formally, $L_{\mathcal{P},c}$ is the language defined by:

$$L_{\mathcal{P},c} = \{x \in \Sigma^* \mid T(\mathcal{P}, x) \models c\}$$

An *exploit* for a vulnerability $(\mathcal{P}, c)$ is simply an input $x \in L_{\mathcal{P},c}$, i.e., executing $\mathcal{P}$ on input x results in a trace that satisfies the vulnerability condition c. A *vulnerability signature* is a matching function MATCH which for an input x returns either EXPLOIT or BENIGN without running $\mathcal{P}$. A *perfect* vulnerability signature satisfies the following property:

$$\text{MATCH}(x) = \begin{cases} \text{EXPLOIT when } x \in L_{\mathcal{P},c} \\ \text{BENIGN when } x \notin L_{\mathcal{P},c} \end{cases}$$

As we show in Section 4.2, the language $L_{\mathcal{P},c}$ can be represented in many different ways ranging from Turing machines which are precise, i.e., accept exactly $L_{\mathcal{P},c}$, to regular expressions which may not be precise, i.e., have an error rate.

Soundness and completeness for signatures.. We define completeness for a vulnerability signature MATCH to be $\forall x : x \in L_{\mathcal{P},c} \Rightarrow$MATCH$(x) =$ EXPLOIT, i.e., MATCH accepts everything $L_{\mathcal{P},c}$ does. Incomplete solutions will have false negatives. We define soundness as $\forall x : x \notin L_{\mathcal{P},c} \Rightarrow$ MATCH$(x) =$ BENIGN, i.e., MATCH does not accept anything extra not in $L_{\mathcal{P},c}$.[7] Unsound solutions will have false positives. A consequence of Rice's theorem [24] is that no signature representation other than a Turing machine can be both sound and complete, and therefore for other representations we must pick one or the other. In our setting, we focus on soundness, i.e., we tolerate false negatives but not false positives. Vulnerability signature creation algorithms can easily be adapted to generate complete by unsound signatures [10].

[7] Normally soundness is $\forall x : x \in S \Rightarrow x \in L_{\mathcal{P},c}$. Here we are stating the equivalent contra-positive.

Vulnerability Signature Representation Classes

We explore the space of different language classes that can be used to represent $L_{\mathcal{P},c}$ as a vulnerability signature. Which signature representation we pick determines the precision and matching efficiency. We investigate three concrete signature representations which reflect the intrinsic trade-offs between accuracy and matching efficiency: *Turing machine* signatures, *symbolic constraint* signatures, and *regular expression* signatures. A Turing machine signature can be precise, i.e., no false positives or negatives. However, matching a Turing machine signature may take an unbounded amount of time because of loops and thus is not applicable in all scenarios. Symbolic constraint signatures guarantee that matching will terminate because they have no loops, but must approximate certain constructs in the program such as looping and memory aliasing, which may lead to imprecision in the signature. Regular expression signatures are the other extreme point in the design space because matching is efficient but many elementary constructions such as counting must be approximated, and thus the least accurate of the three representations.

Turing machine signatures. A Turing machine (TM) signature is a program T consisting of those instructions which lead to the vulnerability point with the vulnerability condition algorithm in-lined. Paths that do not lead to the vulnerability point will return BENIGN, while paths that lead to the vulnerability point and satisfy the vulnerability condition return EXPLOIT. [8] TM signatures can be precise, e.g., a trivial TM signature with no error rate is emulating the full program.

Symbolic constraint signatures. A symbolic constraint signature is a set of boolean formulas which approximate a Turing machine signature. Unlike Turing machine signatures which have loops, matching (evaluating) a symbolic constraint signature on an input x will always terminate because there are no loops. Symbolic constraint signatures only approximate constructs such as loops and memory updates statically. As a result, symbolic constraint signatures may not be as precise as the Turing machine signature.

Regular expression signatures. Regular expressions are the least powerful signature representation of the three, and may have a considerable false positive rate in some circumstances. For example, a well-known limitation is regular expressions cannot count [24], and therefore cannot succinctly express conditions such as checking a message has a proper checksum or even simple inequalities such as $x[i] < x[j]$. However, regular expression signatures are widely used in practice.

Vulnerability Signature Generation

At a high level, our algorithm for computing a vulnerability signature for program $\mathcal{P}$, vulnerability condition c, a sample exploit x, and the corresponding instruction trace T is depicted in Figure 3. Our algorithm for computing vulnerability signatures is:

[8] A path in a program is a path in the program's control flow graph.

1. Pre-process the program before any exploit is received by:
   a) Disassembling the program $\mathcal{P}$. Disassemblers are available for all modern architectures and OS's.
   b) Converting the assembly into an intermediate representation (IR). The IR disambiguates any machine-level instructions. For example, an assembly statement `add a, b` may perform $a + b$ but also set a hardware overflow flag. The IR captures both operations.
2. Compute a chop with respect to the execution trace T of a sample exploit. The chop includes all paths to the vulnerability point including that taken by the sample exploit [25, 48]. Intuitively, the chop contains all and only those program paths any exploit of the vulnerability may take.
3. Compute the signature:
   a) Compute the Turing machine signature. Stop if this is the final representation.
   b) Compute the symbolic constraint signature from the TM signature. Stop if this is the final representation.
   c) Solve the regular expression signature from the symbolic constraint signature.

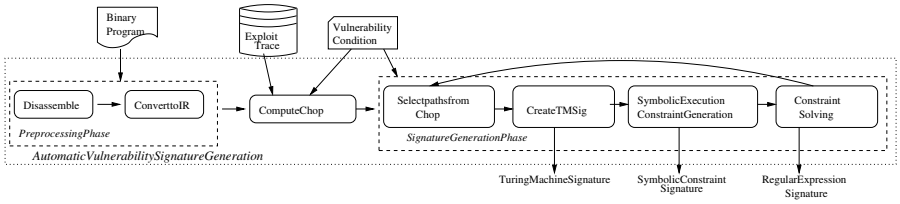


Fig. 3. A high level view of the steps to compute a vulnerability signature.

At a high level, the resulting signature is provably correct since only input strings that can be proved to exploit the vulnerability are included, i.e., a TM signature by construction accepts an input iff the input would exploit the original program; the symbolic constraints are satisfiable iff the TM signature would accept the input; and the regular expression contains only those strings that satisfy the symbolic constraints.

Vulnerability Signature Results

We show in [10] that our automatically generated vulnerability signatures are of much higher quality than those generated with syntax pattern-extraction techniques. The higher quality is because given only a single exploit sample, our vulnerability signature creation algorithm will deduce properties of other unseen exploits. For example, in the atphttpd webserver vulnerability the `get` HTTP request method is case-insensitive [47], and in the DNS TSIG vulnerability that there must be multiple

DNS "questions" (which is a field in DNS protocol messages) present for any exploit to work [63].

5 Automatic Generation of Vulnerability-Specific Execution Filters

In some situations input-based filters are not an appropriate solution. For some vulnerabilities, it is not possible to generate an input-based filter that is accurate, efficient, and of reasonable size. In addition, while one of the desirable properties of input-based filters is that they can be evaluated off the host (*e.g.*, by a network intrusion detection system), this advantage is largely negated in cases where it is impossible to perform accurate filtering without knowledge of state that is on the vulnerable host, such as what encryption key is being used for a particular connection. On the other hand, various host-based approaches have been proposed which are more accurate, but have other drawbacks. 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. Patching is far too slow to respond effectively to a rapidly spreading worm. (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, 50, 29, 38] or require hardware modification or incur high performance overhead when used to protect the entire program execution [19, 44, 60, 15].

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.

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 the following desirable properties:

- 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.
- Our experiments show that the performance overhead of the hardened program is usually only a few percent.
- Our approach is general, and could potentially be applied to other faults such as integer overflow, divide-by-zero, *etc.*

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

6 Sting Self-healing Architecture and Experience

We integrate the aforementioned new techniques with each-other and with existing techniques to form a new end-to-end self-healing architecture, called *Sting* [41], as a first step towards automatically defending against fast Internet-scale worm attacks.

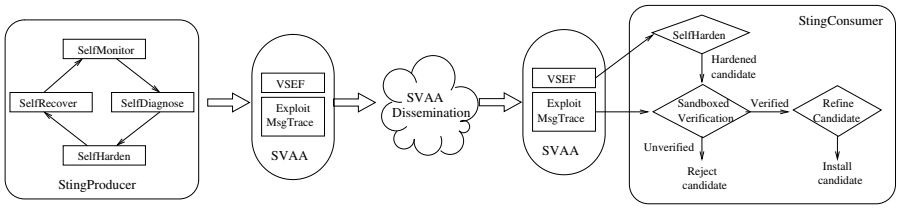


Fig. 4. Sting distributed architecture

Figure 4 illustrates Sting's distributed architecture. At a high level, the Sting self-healing architecture enables programs to efficiently and automatically (1) self-monitor their own execution behavior to detect a large class of errors and exploit attacks, (2) self-diagnose the root cause of an error or exploit attack, (3) self-harden to be resilient against further attacks, and (4) quickly self-recover to a safe state after a state corruption. Further, once a Sting host detects and diagnoses an error or attack, it generates a Self-Verifiable Antibody Alert (SVAA), to be distributed to other vulnerable hosts, who verify the correctness of the antibody and use it to self-harden against attacks against that vulnerability.

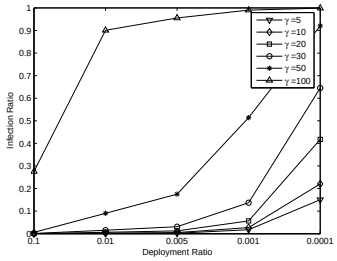
Our Sting self-healing architecture achieves the following properties: Our techniques are accurate, apply to a large class of vulnerabilities and attacks, and enable

critical applications and services to continue providing high-quality services even under new attacks on previously unknown vulnerabilities. Moreover, our techniques work on black-box applications and commodity software since we do not require access to source code. Furthermore, such a system integration allows us to achieve a set of salient new features that were not possible in previous systems: (1) By integrating checkpointing and system call logging with diagnosis-directed replay, we can quickly recover a compromised program to a safe and consistent state for a large class of applications. In fact, our self-recovery procedure does not require program restart for a large class of applications, and our experiments demonstrate that our self-recovery can be orders of magnitude faster than program restart. (2) By integrating faithful and zero side-effect system replay with in-depth diagnosis, we can seamlessly combine light-weight detectors and heavy-weight diagnosis to obtain the benefit of both: the system is efficient due to the low overhead of the light-weight detectors; and the system is able to faithfully replay the attack with no side effect for in-depth diagnosis once the light-weight detectors have detected an attack, which are important properties lacking in previous work [14, 4]. Such seamless integration is also particularly important for *retro-active random sampling*, where randomly selected requests can be later examined by in-depth diagnosis without the attacker being able to tell which request has been sampled. This is a property that previous approaches such as [4] do not guarantee.

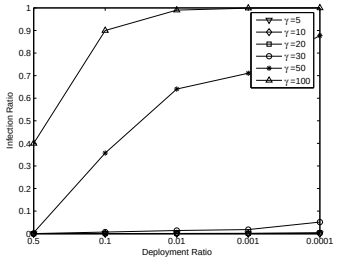
Moreover, our self-healing approach not only allows a computer program to self-heal, but also allows a community of nodes that run the same program to share automatically generated antibodies quickly and effectively. In particular, once a node self-heals, it generates an Self-Verifiable Antibody Alerts containing an antibody that other nodes can use to self-harden before being attacked. The antibody is a response generated in reaction to a new exploit and can be used to prevent future exploits of the underlying vulnerability. Moreover, the disseminated alerts containing the antibody are *self-verifiable*, so recipients of alerts need not trust each other. We call this type of defense *reactive anti-body defense*, similar to Vigilante [14].

Our evaluation demonstrates that our system has an extremely fast response time to an attack: it takes under one second to diagnose, recover from, and harden against a new attack. And it takes about one second to generate and verify a Self-Verifiable Antibody Alerts. Furthermore, our evaluation demonstrates that with reasonably low deployment ratio of nodes creating antibodies (Sting producers), our approach will protect most of the vulnerable nodes which can receive and deploy antibodies (Sting consumers) from very fast worm attacks such as the Slammer worm attack.

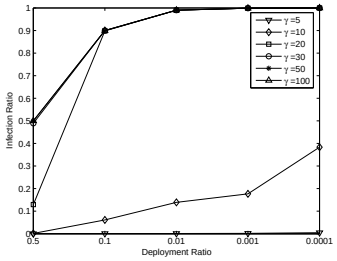
Finally, despite earlier work showing that proactive protection mechanisms such as address randomization are not effective as defense mechanisms [52], we show that reactive anti-body defense alone (as proposed in [14]) is insufficient to defend against extremely fast worms such as hit-list worms. By combining proactive protection and reactive anti-body defense, we demonstrate for the first time that it is possible to defend against even hit-list worms. We demonstrate that if the Sting consumers also deploy address space randomization techniques, then our system will also be able to protect most of the Sting consumers from extremely fast worm attacks such as hit-



(a) Reactive Anti-body Defense against Slammer($\beta = 0.1$)



(b) Hybrid Defense against Hit-list($\beta = 1000$)



(c) Hybrid Defense against Hit-list($\beta = 4000$)

Fig. 5. Effectiveness of Community Defense

list worms. To the best of our knowledge, we are the first to demonstrate a practical end-to-end approach which can defend against hit-list worms.

By developing and carefully uniting a suite of new techniques, we design and build the first end-to-end system that has reasonable performance overhead, yet can respond to worm attacks quickly and accurately, and enable safe self-recovery faster than program restart. The system also achieves properties not possible in previous work as described above. Furthermore, by proposing a hybrid defense strategy, a combination of reactive anti-body defense and proactive protection, we show for the first time that it is possible to defend against hit-list worms.

7 Evaluation

7.1 Reactive Anti-body Defense Evaluation

In this section, we evaluate the effectiveness of our reactive anti-body defense against fast worm outbreaks, using the Slammer Worm and a hit-list worm as concrete examples. In particular, given a worm's contact rate β (the number of vulnerable hosts

an infected host contacts within a unit of time), the effectiveness of our reactive anti-body defense depends on two factors: the deployment ratio of Sting producers α (the fraction of the vulnerable hosts which are Sting producers) and the response time r (the time it takes from a producer receiving an infection attempt to all the vulnerable hosts receiving the SVAA generated by the producer). We illustrate below the total infection ratio (the fraction of vulnerable hosts infected throughout the worm break) under our collaborative community defense vs. α given different β and r.

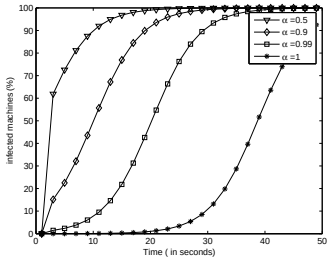
Defense against Slammer worm. Figure 5(a) shows the overall infection ratio vs. the producer deployment ratio α for a Slammer worm outbreak (where $\beta = 0.1$ [34]) with different response time r. For example, the figure indicates that given $\alpha = 0.0001$ and $r = 5$ seconds, the overall infection ratio is only 15%; and for $\alpha = 0.001$ and $r = 20$ seconds, the overall infection ratio is only about 5%. This analysis shows that our reactive anti-body defense can be very effective against fast worms such as Slammer. Next we investigate the effectiveness of this defense against hit-list worms.

Defense against Hit-list worm. Figure 6(c) shows the result of a hit-list worm for $\beta = 1000$ and $\beta = 4000$, and $n = 100,000$[9]. From the figure we see that (ignoring network delay) a hit-list worm can infect the entire vulnerable population (Sting consumers) in a fraction of a second. This is similar to earlier estimates [33, 59] which shows that a hit-list worm can propagate through the entire Internet within a fraction of a second. Thus, our reactive anti-body defense alone will be insufficient to defend against such fast worms because the anti-bodies will not be generated and disseminated fast enough to protect the Sting consumers.

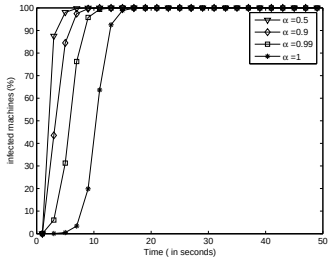
7.2 Proactive Protection against Hit-list Worm

Another defense strategy is a proactive one instead reactive. For example, for a large class of attacks, address space randomization can provide proactive protection, albeit a probabilistic one. The attack, with high probability, will crash the program instead of successfully compromise it. This probabilistic protection is an instant defense, which does not need to wait for the anti-body to be generated and distributed. However, because the protection is only probabilistic, repeated or brute-force attacks may succeed. Figure 6(a) and 6(b) show the effectiveness of such proactive protection against hit-list worms when a certain fraction α of the total vulnerable hosts deploy the proactive protection mechanism, where $p = 1/2^{12}$ (the probability of an attack trial succeeding), and $\beta = 1000$ and $\beta = 4000$ respectively. As shown in the figure, for $\beta = 1000$, when $\alpha = 0.5$ 50% of the vulnerable hosts deploy the proactive protection defense, it will take about 10 seconds for the worm to infect 90% of the vulnerable population; whereas if 100% of the vulnerable hosts deploy the proactive protection defense, it only slows down the worm to about 45 seconds to infect 90% of the vulnerable population. When $\beta = 4000$, the worm propagates even faster as shown in Figure 6(b).

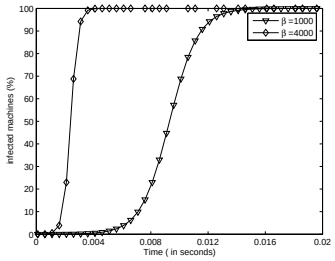
[9] This is basically the same parameters as the Slammer worm, except that instead of a random scanning worm, the worm is a hit-list.



(a) Proactive Protection against Hit-list($\beta = 1000$)



(b) Proactive Protection against Hit-list($\beta = 4000$)



(c) Reactive Anti-body Defense against Hit-list

Fig. 6. Defense Effectiveness Evaluation

Thus, proactive protection alone can slow down the worm propagation to a certain extent, but is clearly not a completely effective defense.

7.3 Hybrid Defense against Hit-list Worm: Combining Proactive Protection and Reactive Anti-body Defense

As explained above, our reactive anti-body defense alone is not fast enough to defend against hit-list worms. Thus, we propose a hybrid defense mechanism where the Sting consumers deploy proactive protection mechanisms such as address space randomization in addition to receiving SVAA using the reactive anti-body defense. In both cases, we assume the probability that an infection attempt succeeds against the proactive protection mechanism (e.g., guessing the correct program internal state with address space randomization) is again 2^{-12}.

Figure 5(b) and Figure 5(c) show the effectiveness of this hybrid defense approach, i.e., the overall infection ratio vs. the producer deployment ratio α, with different response time r, under two different Hit-list worm outbreaks (where $\beta = 1000$ and $\beta = 4000$ respectively). For example, the figures indicate that given $\alpha = 0.0001$ and $r = 10$ seconds, the overall infection ratio is only 5%; for $\beta = 1000$ and 40%

for $\beta = 4000$; and for $\alpha = 0.0001$ and $r = 5$ seconds, the overall infection ratio is negligible (less than 1%) for both cases.

Our simulations show a total end-to-end time (self-detection, self-diagnosis, dissemination, and self-hardening) of about 5 seconds will stop a hit-list worm. Note that our experiments show that self-detection and self-hardening are almost instantaneous, and the total time it takes for a producer to self-diagnose to create a SVAA and for a consumer to verify a SVAA is under 2 seconds. Vigilante shows that the dissemination of an alert could take less than 3 seconds [14]. Thus our system achieves an $r = 2 + 3 = 5$, demonstrating that our system is the first to effectively defend against even hit-list worms.

8 Related Work

Antibody Generation Systems. Vigilante has independently proposed a distributed architecture, where dynamic taint analysis is used to detect new attacks and automatically generate verifiable antibodies [14]. It was a very nice piece of work. There are several important technical differences between Vigilante and Sting. Unlike Sting, Vigilante does not provide self-recovery, and also does not allow the seamless combination of light-weight detectors and heavy-weight detectors. Vigilante automatically generates a specific type of input-based filters, where Sting automatically produces a suite of different antibodies including a wider range of input-based filters and execution-based filters which could provide higher accuracy.

Automatically generating patches when source code is available is explored by Sidiroglou et. al. [53, 54].

Anagnostakis et. al. propose shadow honeypots to enable a suspicious request to be examined by a more expensive detector [4]. However, their approach requires source code access and manual identification of beginning and end of transactions and thus does not work on commodity software. In addition, because they only reverse memory states but do not perform system call logging and replay, their approach can cause side effects. Moreover, because the suspicious request is handled directly by the more expensive detector instead of the background analysis as in our approach, the attacker could potentially detect when its attack request is being monitored by a more expensive detector and thus end the attack prematurely and retry later, whereas our retro-active random sampling addresses this issue.

Liang and Sekar [32] and Xu et. al. [67] independently propose different approaches to use address space randomization as a protection mechanism and automatically generate a signature by analyzing the corrupted memory state after a crash.

Recovery. Our diagnosis-directed self-recovery provides a different point in the design space compared to previous work. For example, Rinard et. al. has proposed an interesting line of research, failure-oblivious computing in which invalid memory operations are discarded and manufactured values are returned [49]. Instead of rolling back execution to a known safe point, Sidiroglou et al have explored aborting the active function when an error is detected [55]. While interesting, these approaches

do not provide semantic correctness, and is thus unsuitable for automatic deployment on critical services. DIRA is another approach that modifies the source code so that overwrites of control data structures can be rolled back and undone [57]. All of these approaches require source code access, and thus cannot be used on commodity software.

There is a considerable body of research on rollback schemes: see [46] for a more detailed discussion. We choose to use FlashBack [58], a kernel-level approach for transactional rollback that does not require access to source code and deterministically replays execution. Another approach is to use virtual machines (VM) for rollback [21, 27]. This approach is more heavy-weight but has advantages such as it is secure against kernel attacks. We plan to explore this direction in the future.

Rx proposes environmental changes to defend against failures, using execution rollback and environment perturbation [46]. However, their approach does not support detailed self-diagnosis and self-hardening, and simply retries execution with different environmental changes until the failure is successfully avoided.

Dynamic Taint Analysis. We use TaintCheck [44, 45] to perform dynamic taint analysis on the binary for self-diagnosis. Others have implemented similar tools [14] which can also be used. Hardware-assisted taint analysis has also been proposed [60, 19]. Unfortunately, such hardware does not yet exist, though we can take advantage of any developments in this area.

9 Conclusion

We presented a self-healing architecture for software systems where programs (1) self-monitor and detect exploits, (2) self-diagnose the root cause of the vulnerability, (3) self-harden against future attacks, and (4) self-recover from attacks. We develop the first architecture, called Sting, that realizes this four step self-healing architecture for commodity software. Moreover, our approach allows a community to share antibodies through Self-Verifiable Antibody Alerts, which eliminate the need for trust among nodes. We validate our design through (1) experiments which shows our system can react quickly and efficiently and (2) deployment models which show Sting can defend against hit-list worms. To the best of our knowledge, we are the first to design and develop a complete architecture capable of defending against hit-list worms.

We are one of the first to realize a self-healing architecture that protects software with light-weight techniques, and enables more sophisticated techniques to perform accurate post-analysis. We are also the first to provide semantically correct recovery of a process after an attack without access to its source code, and our experiments demonstrate that our self-recovery can be orders of magnitude faster than program restart which significantly reduces the down time of critical services under continuous attacks.

References

1. Dynamorio. http://www.cag.lcs.mit.edu/dynamorio/.
2. K2, admmutate. http://www.ktwo.ca/c/ADMmutate-0.8.4.tar.gz.
3. PaX. http://pax.grsecurity.net/.
4. K. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. Keromytis. Detecting targeted attacks using shadow honeypots. In *Proceedings in USENIX Security Symposium*, 2005.
5. Kumar Avijit, Prateek Gupta, and Deepak Gupta. Tied, libsafeplus: Tools for runtime buffer overflow protection. In *USENIX Security Symposium*, August 2004.
6. Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *USENIX Annual Technical Conference 2000*, 2000.
7. Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of 12th USENIX Security Symposium*, 2003.
8. Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium*, 2005.
9. David Brumley, Li-Hao Liu, Pongsin Poosank, and Dawn Song. Design space and analysis of worm defense systems. In *Proc of the 2006 ACM Symposium on Information, Computer, and Communication Security (ASIACCS)*, 2006. CMU TR CMU-CS-05-156.
10. David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Towards automatic generation of vulnerability-based signatures. 2006.
11. Cesar Cerrudo. Story of a dumb patch. http://argeniss.com/research/MSBugPaper.pdf, 2005.
12. CERT/CC. CERT/CC statistics 1988-2005. http://www.cert.org/stats/cert_stats.html.
13. Monica Chew and Dawn Song. Mitigating buffer overflows by operating system randomization. Technical report, Carnegie Mellon University, 2002.
14. Manuel Cost, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-end containment of internet worms. In 20^{th} *ACM Symposium on Operating System Principles (SOSP 2005)*, 2005.
15. 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.
16. Crispin Cowan, Matt Barringer, Steve Beattie, and Greg Kroah-Hartman. FormatGuard: automatic protection from printf format string vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, August 2001.
17. Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *12th USENIX Security Symposium*, 2003.
18. 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.
19. Jedidiah R. Crandall and Fred Chong. Minos: Architectural support for software security through control data integrity. In *International Symposium on Microarchitecture*, December 2004.

20. T. Detristan, T. Ulenspiegel, Y. Malcom, and M. Von Underduk. Polymorphic shellcode engine using spectrum analysis. `http://www.phrack.org/show.php?p=61&a=9`.
21. George Dunlap, Samuel King, Sukru Cinar, Murtaza Basrai, and Peter Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 2002 Symposium on Operating System Design and Implementation (OSDI)*, 2002.
22. Daniel C. DuVarney, R. Sekar, and Yow-Jian Lin. Benign software mutations: A novel approach to protect against large-scale network attacks. Center for Cybersecurity White Paper, October 2002.
23. Stephanie Forrest, Anil Somayaji, and David H. Ackley. Building diverse computer systems. In *Proceedings of 6th workshop on Hot Topics in Operating Systems*, 1997.
24. John Hopcroft, Rajeev Motwani, and Jeffrey Ullman. *Introduction to automata theory, langauges, and computation*. Addison-Wesley, 2001.
25. D. Jackson and E.J. Rollins. Chopping: A generalization of slicing. In *Proc. of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1994.
26. Richard Jones and Paul Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the Third International Workshop on Automated Debugging*, 1995.
27. 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 2005 Symposium on Operating Systems Principles (SOSP)*, 2005.
28. Hyang-Ah Kim and Brad Karp. Autograph: toward automated, distributed worm signature detection. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.
29. Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.
30. 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.
31. Robert Lemos. Counting the cost of the slammer worm. `http://news.com.com/2100-1001-982955.html`, 2003.
32. 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.
33. 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.
34. 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.
35. David Moore, Colleen Shannon, Geoffrey Voelker, and Stefan Savage. Internet quarantine: Requirements for containing self-propagating code. In *2003 IEEE Infocom Conference*, 2003.
36. Carey Nachanberg. Computer virus-antivirus coevolution. *Communications of The ACM*, 1997.
37. George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *Proceedings of the Symposium on Principles of Programming Languages*, 2002.

38. 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.).
39. 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.
40. James Newsome, David Brumley, and Dawn Song. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *Proceedings of the* 13^{th} *Annual Network and Distributed System Security Symposium (NDSS)*, 2006.
41. James Newsome, David Brumley, Dawn Song, and Mark R. Pariente. Sting: An end-to-end self-healing system for defending against zero-day worm attacks on commodity software. Technical Report CMU-CS-05-191, Carnegie Mellon University, February 2006.
42. 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.
43. James Newsome, Brad Karp, and Dawn Song. Paragraph: Thwarting signature learning by training maliciously. Technical Report CMU-CS-05-149, Carnegie Mellon University, February 2006.
44. 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.
45. James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. Technical Report CMU-CS-04-140, Carnegie Mellon University, May 2005.
46. Feng Qin, Joe Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating bugs as allergies—a safe method to survive software failures. In 20^{th} *ACM Symposium on Operating System Principles (SOSP)*.
47. r code. ATPhttpd exploit. http://www.cotse.com/mailing-lists/todays/att-0003/01-atphttp0x06.c.
48. T. Reps and G. Rosay. Precise interprocedural chopping. In *Proc. of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1995.
49. 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.
50. Tim J Robbins. libformat. `http://www.securityfocus.com/tools/1818`, 2001.
51. Olatunji Ruwase and Monica Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, February 2004.
52. Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, October 2004.
53. Stelios Sidiroglou and Angelos D. Keromytis. A network worm vaccine architecture. In *Proceedings of the IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), Workshop on Enterprise Security*, pages 220–225, June 2003.
54. Stelios Sidiroglou and Angelos D. Keromytis. Countering network worms through automatic patch generation. *IEEE Security and Privacy*, 2005.

55. 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.
56. Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated worm fingerprinting. In *Proceedings of the 6th ACM/USENIX Symposium on Operating System Design and Implementation (OSDI)*, December 2004.
57. Alexey Smirnov and Tzi cker Chiueh. DIRA: Automatic detection, identification, and repair of control-hijacking attacks. In *Proceedings of the* 12^{th} *annual Network and Distributed System Security Symposium (NDSS)*, 2005.
58. Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the 2004 USENIX Technical Conference*, 2004.
59. Stuart Staniford, Vern Paxson, and Nicholas Weaver. How to 0wn the Internet in your spare time. In *11th USENIX Security Symposium*, 2002.
60. G. Edward Suh, Jaewook Lee, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of ASPLOS*, 2004.
61. Peter Szor. Hunting for metamorphic. In *Proceedings of the Virus Bulletin Conference*, 2001.
62. Jamie Twycross and Matthew M. Williamson. Implementing and testing a virus throttle. In *Proceedings of 12th USENIX Security Symposium*, August 2003.
63. US-CERT. Vulnerability note vu#196945 - isc bind 8 contains buffer overflow in transaction signature (tsig) handling code. `http://www.kb.cert.org/vuls/id/196945`.
64. 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.
65. Matthew M. Williamson. Throttling viruses: Restricting propagation to defeat malicious mobile code. In *Proceedings of the 18th Annual Computer Security Applications Conf erence*, 2002.
66. Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent runtime randomization for security. Technical report, Center for Reliable and Higher Performance Computing, University of Illinois at Urbana-Champaign, May 2003.
67. Jun Xu, Peng Ning, Chongkyung Kil, Yan Zhai, and Chris Bookholt. Automatic diagnosis and response to memory corruption vulnerabilities, 2005.