Sting: An End-to-End Self-healing System for Defending against Zero-day Worm Attacks on Commodity Software

James Newsome, David Brumley, Dawn Song, Mark R. Pariente
Carnegie Mellon University

Abstract

Complex computer systems are plagued with bugs and vulnerabilities. Worms such as SQL Slammer and hit-list worms exploit vulnerabilities in computer programs and can compromise millions of vulnerable hosts within minutes or even seconds, bringing down vulnerable critical services.

In this paper, we propose an end-to-end self-healing approach to achieve the following goal: for a large class of vulnerabilities and attacks, we can protect a large fraction of critical services and enable them to be highly available even in the case of a zero-day hit-list worm. Moreover, our techniques do not require access to source code and thus work on COTS software. We achieve this goal by designing an end-to-end self-healing approach: (1) programs use light-weight techniques to efficiently self-monitor the execution behavior and reliably detect a large class of errors and exploits, (2) we use sophisticated techniques to self-diagnose the root cause of detected errors and exploits, (3) programs self-harden to be resilient against further attacks on the same vulnerability, and (4) safely and efficiently self-recover to a safe state. Self-hardening does not result in false positives of legitimate traffic, and adds little performance overhead.

Moreover, our approach allows a community of nodes to efficiently share Self-Verifiable Antibody Alerts (SVAAs), which are produced by the self-diagnosis engine. Nodes can verify that SVAAs fix real vulnerabilities without trusting the SVAA senders, and self-harden quickly and efficiently based upon SVAAs. By employing a new approach of combining proactive protection and reactive anti-body defense, we show for the first time that it is possible to protect vulnerable programs and enable critical services to remain undisrupted even under extremely fast worm attacks such as hit-list worms.

1 Introduction

We increasingly rely on highly available systems in all areas of society, from the economy, to military, to the government. For example, some estimates put the cost of downtime for businesses at six million dollars lost per hour down [45]. 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 [15]. These vulnerabilities, when exploited, can cause devastating effects, such as self-propagating worm attacks which can compromise all vulnerable hosts within a matter of minutes or even seconds [35, 53], and cause millions or even billions of dollars of damage [32]. Therefore, we need to develop effective mechanisms to enable critical applications and services to remain highly available and effective even under malicious attacks on previously unknown vulnerabilities, such as a fast Internet-scale worm attacks.

Open Issues. Despite great research efforts done in the area of defending against large scale worm attacks, several central challenges remain unaddressed to sustain high availability of vulnerable critical services during a worm outbreak: (1) Can we protect a large fraction of vulnerable critical services against fast zero-day worm attacks and enable them to be highly available even through continuous attacks? (2) Once a state-ful critical application is compromised, can we quickly recover it to a safe and consistent state to enable it to continue providing services and minimize down time? (3) To effectively defend against fast large scale worm attacks, light-weight detectors such as address randomization [4, 9, 10, 16, 23, 25, 59]

and heavy-weight diagnosis engines such as dynamic taint analysis [17, 40, 20, 55] have been proposed. Can we seamlessly combine the two methods to obtain the benefit of both in a secure and safe way? (4) Reactive anti-body defense such as Vigilante [17] has been proposed where upon detection of an attack, an alert will be generated and quickly disseminated to protect vulnerable hosts from the attack. However, will this approach be fast enough to beat fast worms? Hit-list worms can propagate through the entire Internet within seconds. Can we devise an effective defense strategy against such fast devastating attacks?

System Overview and Contributions. In this paper, by carefully uniting a suite of techniques, we propose a new end-to-end self-healing architecture, called *Sting*, as a first step towards addressing the above questions.

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.

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 (Section 5). (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 [17, 6] (A detailed comparison with related work such as Vigilante is availabe in Section 7). 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 (Section 2.2). This is a property that previous approaches such as [6] do not guarantee.

Moreover, our self-healing approach does not only allow a computer program to self-heal, but also allow community of nodes that run the same program to share their self-healing results quickly and effectively. In particular, once a node self-heals, it can create a 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 [17].

We have designed and implemented the whole system. Our evaluation demonstrates that our system has 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 [46], we show that reactive anti-body defense alone (as proposed in [17]) 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-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.

Despite the fact that several components in this Sting architecture have been proposed [40, 38], it takes great care to design and build an end-to-end system to demonstrate the feasibility of enabling stateful critical services to be protected and highly available even during a zero-day hit-list worm attack. This is precisely the contribution of this paper: by carefully uniting a suite of 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. We hope such a holistic system-building approach will encourage more end-to-end systems to be built and evaluated to provide realistic measures against large scale attacks.

2 Self-Healing Architecture: Requirements and Design

In this section we first provide an overview of our self-healing architecture and discuss the properties and challenges of our self-healing approach. We then describe our design of each component in our self-healing architecture.

2.1 Architecture Overview

To protect a vulnerable program against new exploit attacks on previously unknown vulnerabilities and enable critical applications and services to continue providing high-quality service even under these malicious attacks, we propose a new self-healing approach to enable a computer program to efficiently and automatically (1) *self-monitor* its own execution behavior and detect errors or intrusions, (2) *self-diagnose* the root cause of the error/intrusion, (3) *self-harden* to be resilient against further attacks, and (4) *self-recover* to a safe state. Our overall self-healing architecture is given in Figure 1.

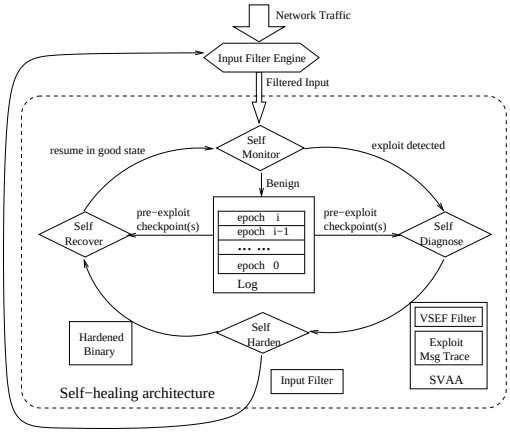


Figure 1: The Sting Self-healing Architecture

There are many requirements and challenges to make this approach practical and effective:

- High efficiency: There should be little overhead on normal program execution. For example, the self-monitoring step should have low performance overhead.
- High accuracy and coverage: The system should be effective against many different classes of exploits and vulnerabilities.
- Work for black-box applications and COTS software: the whole approach should not require access to source code, or require specific configurations or setting changes beyond what is normally required by the program.
- Fast and effective response: the whole approach needs to respond to a new attack extremely fast and effectively, so that the critical applications and services can remain undisrupted under attacks.
- Safe: in particular the approach should not change the semantics of the program which can lead to further attacks and unexpected program behavior. For example, the self-hardening and self-recovery step should not change the semantics of the program.

The Sting self-healing architecture is the first to meet all the above requirements simultaneously, meeting the challenge of a holistic self-healing architecture.

2.2 Self-monitoring

Problem and Challenges. The self-monitoring engine needs to be able to detect new attacks to trigger detailed diagnosis, and record sufficient information to enable post-attack diagnoses. Thus, our self-monitoring engine has two components: a light-weight detection engine which detects attacks and errors, and a self-logging engine which logs relevant information about program execution for post-attack diagnosis and recovery. There are several challenges for self-monitoring: (a) detect attacks with minimal impact on performance, (b) detection should have high coverage, and (c) efficiently log information needed for post-attack diagnosis and recovery.

2.2.1 Checkpointing and Logging

The self-logging engine performs periodic process checkpointing and continuous system-call logging. The logs will be used in the self-diagnosis step to deterministically replay an attack to determine the root cause of an attack and in the self-recovery step to restore the program to a safe state.

The self-logging engine performs process checkpointing by saving a copy of the active memory image. A checkpoint acts like a shadow process, replicating the complete process state of the process at the time of checkpoint. The memory image includes the process's virtual memory, register values, open file handles, signal table, etc. The checkpoint is created periodically either depending on the time elapsed or the number of sessions elapsed. In order to save space and time, each checkpoint is copy-on-write (COW) to minimize the amount of copying between the i^{th} and $i-1^{th}$ checkpoint, as well as between the latest checkpoint and the active process. A checkpoint will be used to rollback the program into a previous process state during self-diagnosis and recovery phases.

The system-call log records the system-call number, arguments, and return values. The system-call log is used during diagnosis and recovery phases to replay interactions with the operating system, i.e., during diagnosis and recovery when a system call is executed we replay the logged return value instead of actually executing the real system call a second time. The system call log enables us to replay the sequence of steps in a past execution faithfully and also without introducing any side-effects (e.g., a packet will not be sent out twice).

2.2.2 Light-weight exploit detection

Light-weight detector vs. heavy-weight detector, The light-weight detection engine detects different classes of exploit attacks and needs to be very efficient and function without access to source code. Different methods have been proposed to detect software exploit attacks without access to source code. They fall into two main categories: light-weight detectors such as address space randomization and system-call based anomaly detection, and heavy-weight detectors such as dynamic taint analysis [17, 40, 20, 55]. The light-weight detectors are efficient and often only adds a few percentage of performance overhead at the most, although they may miss certain attacks and do not provide detailed information about the vulnerability and the attack. The heavy-weight detectors, on the other hand, are accurate and have high coverage and provide detailed information about the vulnerability and the attack, however, they can have significant performance overhead if used to check the entire program execution [40]. Thus, it would be largely beneficial to be able to combine light-weight detectors and heavy-weight detectors in a systematic manner and obtain the benefits of both.

As we will describe in more detail in section 2.3, we devise a novel method that allows us to benefit both from the efficiency of light-weight detectors and from the accuracy, coverage, and detailed analysis offered

by heavy-weight detectors. In our approach, we use only light-weight detectors in the light-weight detection engine so the performance overhead is low; the heavy-weight detectors are only used in the self-diagnosis engine to perform post-mortem analysis. In addition to using light-weight detectors to detect attacks, the light-weight engine also randomly samples sessions to be diagnosed by the diagnosis engine. This random sampling guarantees the worst case probability that an attack will be detected even if an attack detectable by the heavy-weight detectors is missed by the light-weight detectors.

Address Space Randomization as a Light-weight Detector. As a concrete example, we use program address space randomization as a light-weight detector in our light-weight detection engine (and we can easily plug in other light-weight detectors such as system-call based anomaly detection). Because most exploit attacks require specific knowledge of the internal state of a program to succeed, address-randomization techniques can prevent an attacker from knowing the internal state of a program beforehand by randomizing certain internal states of each process. Various address-space randomization techniques have been proposed to efficiently randomize process internal states without access to program source code, including randomizing run-time memory layout [4, 9, 10, 16, 23, 25, 59] and global library entry points [16]. Many of these techniques are already widely deployed in the Linux community such as in the Fedora Core Linux distribution.

In address-space randomization, if the number of possibilities available for randomization is p then each attack attempt succeeds with a probability p^{-1}. For example, Shacham et. al. show that the current address randomization implementations in Linux gives 2^{16} different randomization possibilities for a process, so each attack attempt will succeed with a probability about $2^{-16} = 0.000015\%$ [46].

When an attack attempt fails due to address randomization techniques, it usually causes a crash of the program such as a segmentation fault because it corrupts the process's internal state. Upon detecting the error/crash, the light-weight detection engine will trigger the self-diagnosis step.

Retro-active Random Sampling. Address-randomization techniques in practice defend against a wide-range of attacks. However, certain types of attacks such as information leakage attacks require heavy-weight detectors to detect. To handle these attacks, we propose random selection as another light-weight detection technique. Sessions are sampled randomly with a certain probability. Selected sessions are then served as normal, however, they are then analyzed in the diagnosis engine in the background: they are *retro-actively* "replayed" using the checkpoint and system call log under a heavy-weight detector which can then detect whether there has been a real attack (details of diagnosis are described in the next section).

There are two important benefits to retro-active random sampling: (1) we can employ heavy-weight detectors without affecting the performance of legitimate requests since detection is done in the background on a small fraction of the sessions, and (2) attackers cannot determine when heavy-weight detection is under way. The last point is subtle: if Sting randomly selected live network sessions to monitor under heavy-weight detectors, then an attacker may be able to judge whether its session has been selected to be monitored under heavy-weight detection by observing the response time. If the attacker can detect when its session is selected to be monitored under heavy-weight detectors, he may then choose not to send the attack, and retry the attack later with the expectation of not being chosen by random sampling again. This evasion attack shows that heavy-weight detectors cannot simply use sampling to reduce their performance overhead, and retro-active sampling is necessary to prevent the evasion attack.

2.3 Self-diagnosis

Problem and Challenges The self-diagnosis engine performs in-depth analysis to determine the root cause of an attack and to provide detailed information and generate antibodies. The antibodies and information provided by the self-diagnosis engine can then be used to self-harden the program to prevent future instances of the attack from succeeding. The self-diagnosis mechanism needs to satisfy several requirements: (1) able to perform post-mortem analysis since the self-diagnosis step is triggered after the attack; (2) accurate

detection and able to handle a wide range of vulnerability classes and exploits; (3) no need to access to source code so it works with commodity software; (4) providing detailed information about the vulnerability and the attack: in particular, it should be able to identify which original session has caused the attack and provide sufficient information to enable automatic responses to defend against further attacks; (5) sufficiently fast: the self-diagnosis step is not performed on high throughput requests, so it can be less efficient than the light-weight detectors; however, it does need to be sufficiently fast to enable a fast response to new attacks.

The self-diagnosis engine performs post-mortem analysis by using an *analyzer* to examine the program's previous execution via a rollback and replay approach. Note that we build our analyzer based on previous work on dynamic taint analysis. The new contribution in this self-diagnosis step is that by employing rollback-and-replay, we can faithfully reproduce the attack execution (even for stateful applications) and guarantee no side-effects on the system, and thus provide a safe and faithful environment for post-mortem analysis.

The diagnosis procedure. Once the self-diagnosis step is triggered (e.g., when an attack is detected in self-monitoring), the self-diagnosis engine first rollbacks the program state to a previous checkpoint, then replays the program execution using the system-call replay under the analyzer to perform in-depth analysis. Note that using system-call replay ensures that no side-effects will happen during replay because system-calls are not really executed. Also, rollback and replay ensures that the previous execution is faithfully reproduced. Thus, our system achieves two important properties for post-mortem analysis: (1) no side-effect; (2) faithfully reproduce the attack execution. Note that this rollback-and-replay method handles faithful replay of attack execution even for stateful applications; whereas a network-trace based replay will not ensure a faithful replay of attack execution for stateful applications.

If the analyzer detects the attack which has triggered the diagnosis step, it means that the attack has happened after the last checkpoint. Otherwise, it means that the analyzer needs information from program execution before the last checkpoint to detect the attack, the self-diagnosis engine then rollbacks the program state to a checkpoint before the last checkpoint and replays the program execution using system-call replay under the analyzer again. This process continues until the analyzer detects the attack triggered the diagnosis step.

After detecting the attack during replay, the analyzer performs detailed analysis, generates antibodies which will be used in the self-hardening step, and identifies the original inputs and session which have caused the attack (this information will be used in the self-recovery step). Note that with this approach, the more expensive operation – diagnosis – is only performed when an attack has happened (or when sessions are sampled which is a small percentage of total traffic), and thus the performance on normal program execution is mostly unaffected.

The analyzer. The analyzer needs to be able to detect and analyze a wide range of exploit attacks accurately and without access to source code. As a concrete example, we employ dynamic taint analysis in the analyzer. Dynamic taint analysis has been recently proposed to monitor program execution and detect software exploit attacks [20, 40, 55, 17]. It is based on the observation that in order for an attacker to change the execution of a program illegitimately, he must perform some form of *overwrite attack*. That is, he needs to illegitimately overwrite a security-sensitive value using values derived from his own input. For example, values such as return addresses and function pointers should usually be supplied by the code itself, not from external untrusted inputs. However, an attacker may attempt to exploit a program by overwriting these values with his own data.

Dynamic taint analysis has been implemented as a tool called TaintCheck, and shown to accurately detect a wide range of exploit attacks including buffer overrun, format string, and double free attacks [40]. TaintCheck performs dynamic taint analysis on binary programs by using program emulation, so it can detect exploit attacks without the access to program source code, allowing it to be used on commodity software, and without any recompilation. TaintCheck has been implemented for Linux using the Valgrind run-time monitoring tool [37], and for Windows using the DynamoRIO run-time monitoring tool [1].

TaintCheck keeps track of what data is read from untrusted sources by intercepting system calls, such as `read`, and tracking what memory locations they write to. TaintCheck then keeps track of what data becomes tainted by this untrusted data by inserting *instrumentation* instructions to propagate the taint attribute after certain instructions. For example, for data movement instructions (LOAD, STORE, MOVE, *etc.*) and data arithmetic instructions (ADD, SUB, XOR, *etc.*), the result will be tainted if and only if any operand is tainted. TaintCheck also inserts extra instrumentation before every point where data is used in a sensitive way to check whether the data is not tainted. For example, it checks that addresses used as return addresses and function pointers are not tainted before using them. If it is tainted, TaintCheck signals that an attack has taken place. For more details, please see [40].

Diagnosis output Once the analyzer detects the attack, it performs backtracing analysis which traverses the chain of operations which operated on tainted data backwards starting from the detection point. By performing this backtracing analysis, the analyzer identifies the list of instructions that operated on tainted data which propagated until the detection point. From this information, the analyzer generates a *VSEF* filter which includes: (1) the list of instruction addresses that contributed to the taint propagation to the detection point, and (2) the instruction address that detected the misuse of tainted data and thus detected the attack. We describe in section 2.4 more details about VSEF and how the VSEF filter will be used to generate a hardened binary in the self-hardening step.

From the backtracing analysis, the analyzer also identifies which original input in which session has caused the attack. This information will then be used in the self-recovery step which restores the program into a safe state. More details are described in section 2.5.

Finally, the self-diagnosis phase outputs a SVAA, which includes the VSEF filter, the original exploit message trace, and information about the program exploited such as the name and version of the vulnerable program. An example SVAA alert is shown in Figure 2.

Name:	ATPHttpd 0.4b
Time:	2005-10-17 23:59:59 PST
VSEF:	0x80b34a 0x80b40b ... 0x80cdcd
Exploit:	GET /aaaaa.....

Figure 2: Example SVAA for ATPhttpd

2.4 Self-harden

Problem and Challenges The self-diagnosis step generates an SVAA which contains the VSEF filter along with the exploit message trace. In this section we describe how hosts self-harden based upon self-generated SVAA (where only the VSEF portion is used). In the next section we discuss verification of remotely generated SVAA's. The self-hardening mechanism needs to satisfy several properties: (1) does not change normal program execution behavior; (2) protects the program from further attacks on the same vulnerability as analyzed in the self-diagnosis step; (3) low performance overhead.

The self-hardening engine employs two methods to harden the program against further attacks on the same vulnerability: (1) it instruments the program to generate a hardened binary which can detect further attacks; (2) it generates input-based filters which can be used to match the inputs before they are passed into the program to see whether the inputs are malicious, and malicious inputs will then be dropped. It is in general extremely difficult to generate accurate input-based filters, (although partially-accurate input-based filters which has zero false positives can be used as optimizations for the self-hardening problem by filtering out part of the malicious inputs), and much easier to devise accurate hardened binary to effectively defend against further attacks. Thus, due to space limitations, we will focus on the approach of generating a hardened binary as our self-hardening mechanism, in particular, a method called *vulnerability-specific execution-based filtering* (VSEF) [38]. Note that we include a brief overview of the self-harden step here merely for completeness. Please refer to [38] and [24] for details.

VSEF We use a method called *vulnerability-specific execution-based filtering* (VSEF) to generate a hardened binary by instrumenting the original vulnerable binary using information from the VSEF filter in the

SVAA. In particular, the hardened binary can be viewed as the original binary with a light-weight dynamic taint analysis—instead of the full dynamic taint analysis where the entire program is instrumented to propagate the taint attribute and detect misuse of tainted data, we observe that only a few relevant instructions need to be instrumented to detect attacks against a specific vulnerability.

The self-diagnosis step already identified the list of instruction addresses that need to be instrumented for dynamic taint analysis: (1) the list of instruction addresses that contributed to the taint propagation to the detection point, and (2) the instruction address that detected the misuse of tainted data and thus detected the attack. These are the instructions that need to be instrumented for dynamic taint analysis to detect similar exploit attacks later.

We use a binary instrumentation engine to add instrumentation to instruction addresses that appear in the VSEF filter. In particular, we add instrumentation to each instruction whose address appears in the VSEF filter to propagate taint information, and inserts the appropriate safety check at the detection point given in the VSEF filter (the instruction address that detected the misuse of tainted data). The result of this binary instrumentation is a hardened binary.

VSEF-hardened binaries are able to reliably detect attacks against the same vulnerability, even when they have been modified by a polymorphic or metamorphic engine. They do not have false positives other than any that would also be present in full dynamic taint analysis, which is generally very few [40]. The performance overhead of a VSEF-hardened binary is quite small, because only a few instructions need to be instrumented. Additionally, VSEF filters can easily be combined to create hardened binaries that are resilient to attacks against each of the corresponding vulnerabilities. For a more detailed discussion of VSEF's properties, see Appendix B.

Input-based filters Input-based filters are commonly employed to filter out known exploits or suspicious packets at the network edge [30, 50, 31, 40, 39]. Our self-diagnosis engine identifies the execution path leading to the exploit, and how tainted data was propagated from the point of input to the point where it was misused. This information can be analyzed to produce input-based filters. These input-based filters can be used in conjunction with the hardened binary to help improve performance. In particular, it is possible to generate filters that are guaranteed to have no false positives, but may have false negatives. Dropping inputs that such a filter matches poses no risk of dropping legitimate requests, and saves Sting from having to perform self-recovery. For more information on how to generate input-based filters from the self-diagnosis engine, see [24].

2.5 Self-Recovery

Problem and Challenges Once an attack is detected, the vulnerable program may already be in an unsafe state. For example, its memory may already be corrupted. Thus, we need to devise mechanisms to bring the program back into a safe state. The most straightforward approach is to simply kill the process and restart. However, this approach has several limitations. First, restarting a program may be slow (several seconds or longer) and thus render the service unavailable during this time [57]; for servers that require significant caching of state in main memory, it requires a long period of time to warm up the cache for full service capacity [11, 5]. Micro-rebooting [12] alleviates this problem by only rebooting the failed component, but it requires substantial modifications to legacy software, while we would like to have an approach that works on black-box applications. Restarting the process also leads to loss of current state, which can range from annoying (dropped HTTP connections) to fatal (restarted in the middle of modifying a file, leaving it in an inconsistent state).

Our Approach: Diagnosis-directed Self-recovery We propose a new method for *self-recovery*. Upon detecting an attack, the self-diagnosis step will identify which original session has caused the attack. Assuming the original session that caused the attack (i.e., the malicious session) started in epoch i, and let e_{i-1}

represent the checkpoint at the end of epoch $i-1$. Our self-recovery will then rollback the system state to checkpoint e_{i-1}. This step ensures that we've restored the program to a non-corrupted internal state.

Next, we need to ensure that the program's internal state is consistent with all other external state. For example, suppose the process was handling a legitimate request concurrently with the malicious request. If there was communication with the legitimate user after checkpoint e_{i-1}, we need to make sure the program's internal state still reflects this.

Note that all changes to external state are made by the program making system calls. As the program is deterministically executing from the checkpoint, it will make the same sequence of system calls. Each time the process makes a system call, we could *replay* the return value of that system call without actually making the system call. This would cause the program to reach the same final state. However, we do not want to replay the system calls corresponding to the attacker's request, since that would cause us to reach the unsafe state again. Therefore, we must return something different for system calls corresponding to the attack input.

Before replaying execution from the checkpoint, we give each system call in the system call log one of three labels, which will affect how we treat that system call during replay.

- **Type 1:** System calls that do not modify external state. Rather, they are used to query external state. These include `gettimeofday`, `stat`, and `read` from local files. During replay, if the program makes these system calls again, we return the original return values to keep the replay as close to deterministic as possible.
- **Type 2:** System calls corresponding to malicious input. Our diagnosis identifies these system calls. Examples are `read`, `recv`, and `recvfrom`. We change the return values of these in such a way that the program state does not become corrupted again, while minimizing how how much the current execution state diverges from the logged execution.
- **Type 3:** System calls that modify external state, but do not correspond to receiving malicious input. Examples are system calls that communicate with legitimate users, such as `send` and `recv`, system calls that change the state of other processes, such as `read` from fifos, `fork`, `kill`, `etc`, and system calls that modify the file system such as `write`, `unlink`, *etc.* We need to get the program back into the state of having made these calls so that, for example, it doesn't write something to a socket that it already wrote during the original execution. Therefore, we return the original return values of these calls during recovery.

Before replay, we place a cursor in the system call log corresponding to where checkpoint e_{i-1} was taken. As the program is executing, each time it makes a system call, we match it with the next system call of that type after the cursor that was called with the same parameters. If such a system call is found, the corresponding action is taken as described above, and the cursor is advanced to the matching system call in the log. If there is no matching system call in our log, we may not be able to guarantee consistency between the program's internal state and external state. In such cases, we can fall back on restarting the vulnerable program. Once all the Type 3 system calls have been replayed, the program's internal state is consistent with external state. That is, execution can be safely resumed without breaking the program's normal semantics.

There are two cases which can cause us to be unable to bring the program back to a consistent state. The first case is when there are dependencies between requests processed by the program. For example, suppose that the program keeps a counter based on the number of requests served, and uses that counter as a session ID. When we remove the malicious request during replay, this may cause the counter to not be incremented, and hence generate a different session ID for the next innocuous request in the log. We detect this condition, because when the session ID is sent to that legitimate user, we see that the data sent in that `send` system call does not match the data sent originally. While it may be possible to detect and repair some scenarios such as this one, we default to the safe action of restarting the process.

The second case is when the malicious request resulted in permanent effects, such as writes to a file. That is, there were Type 3 system calls made as a result of the malicious request. When this is the case, these

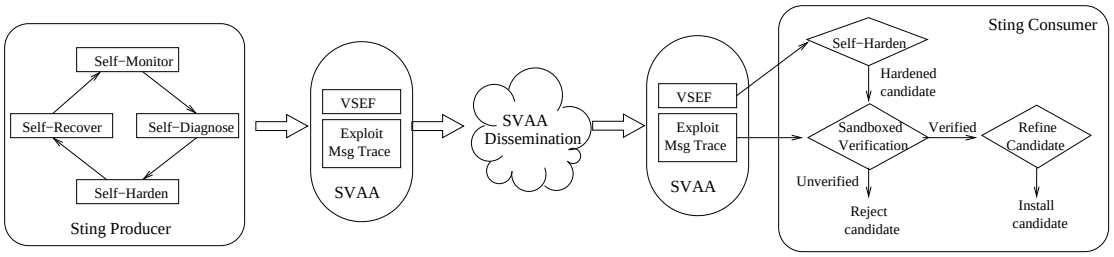


Figure 3: SVAA alert distribution architecture.

system calls may not occur again during replay, since we modified the malicious request. In many cases this is harmless- such as writes to a system log or a temporary file. In other cases, it is possible to roll back the corresponding external state, such as writes to a local file that no other process has read yet. However, it is nontrivial to reliably recognize and correctly deal with such cases, and a mistake could lead to incorrect behavior of the program, which can be catastrophic in some scenarios. We therefore again default to the safe action of restarting the process if we are unable to replay all the Type 3 system calls in the log.

We expect that in many scenarios, neither of these cases will occur, and we will be able to efficiently put the program back into a safe and consistent state, allowing it to continue execution while guaranteeing that its normal semantics are not broken.[1] We reliably detect when recovery is potentially unsafe: when the program does not make the same Type 3 system calls during replay, with the same parameters, in the same order. We then default to the safe action of restarting the program.

3 Reactive Anti-body Defense

Hosts that deploy the full Sting self-healing architecture can detect attacks and generate antibodies to harden themselves, and thus be resilient against attacks. We call such hosts *Sting producers*. Some hosts may choose not to run the full Sting self-healing architecture but would like to benefit from the alert and antibody generated by the Sting producers; we call these hosts *Sting consumers*. Sting producers and Sting consumers collectively form a community where the alert and antibody generated by a Sting producer can then be disseminated through a Sting alert and antibody dissemination mechanism to other Sting producers and consumers to enable them to harden before any infection attempts. We call this type of defense *Reactive Anti-body Defense*. The overall picture for the antibody dissemination architecture is given in Figure 3.

An effective antibody dissemination system needs to enable fast and robust antibody delivery, and also should not require mutual trust among the participants. In particular, recipient's should not blindly trust SVAA's, as attackers or incompetent participants may attempt to cause the recipient to accept a false alert and antibody which blocks legitimate traffic. For example, an attacker should not be able to convince a recipient that a VSEF catches only exploits but when deployed matches legitimate executions as well.

Requirements and Challenges. A robust and secure alert and antibody should allow a recipient to:

- Verify the alert corresponds to a real vulnerability.
- Verify the proposed antibody catches exploits of the vulnerability.
- Verify the proposed antibody does not create false positives, i.e., will not block legitimate traffic.
- The verification procedure should be fast, so it allows the recipient to deploy the antibody before any infection attempt.

[1] An exception is systems that provide real-time performance guarantees, which are outside the scope of this work.

We propose Self-Verifiable Antibody Alerts (SVAA) which achieve all the above properties. We use SVAA in the Sting alert and antibody dissemination architecture which allows for SVAA to be rapidly created and disseminated by SVAA creators and securely verified by SVAA consumers.

3.1 Generating Self-Verifiable Antibody Alerts

A Self-Verifiable Antibody Alerts contains the name and version of the vulnerable application, a VSEF filter, and an exploit message trace that exercises the vulnerability, as shown in Figure 2. As described in section 2.3, the SVAA is generated by a Sting producer after it detects an attack and performs the self-diagnosis step. The generation of a SVAA is fast, taking usually less than a second, as shown in section 5.

3.2 Verifying Self-Verifiable Antibody Alerts

At a high level, recipients verify new SVAA's in a sand-boxed environment. If the SVAA verifies, the included VSEF filter will be accepted and also refined (if necessary) and the node will deploy the new antibody; otherwise, the SVAA will be rejected.

The Verification Procedure. Upon receiving a new SVAA, the recipient first checks to see if he is running the vulnerable program. If yes, it will use the VSEF filter in the SVAA to build a hardened binary as described in the self-harden step in Section 2.4. Then the recipient verifies the SVAA by replaying the exploit against the hardened binary within a sand-box environment such as a virtual machine. The sand-box must confine any (possibly legitimate) side-effects caused by the exploit. For example, in order to reach the exploit point a program may read and write files, update shared memory, etc. We stress that the side-effects may not be specific to the exploit and happen regardless of whether the input is malicious or not. All the side effects should be confined in the virtual machine.

A SVAA is *verified* if the hardened binary generated using the contained VSEF filter catches the contained exploit (as explained in section 2.4), else we say the SVAA is unverifiable.

If a SVAA is verified, it means that the hardened binary generated can detect real exploit attacks on a real vulnerability of the recipient. However, it does not mean that the hardened binary will be the most efficient one to defend against attacks on this vulnerability. In particular, an attacker could try to send a really long VSEF filter containing unnecessary instruction addresses to be instrumented and thus resulting in an extremely inefficient hardened binary as a denial-of-service attack. To defend such attacks, after a SVAA is verified, we go through a *refinement* step. After replaying the exploit against the hardened binary in the sandboxed environment, we can easily see which instruction addresses instrumented in the hardened binary actually have operated on tainted data. If an instruction instrumented in the hardened binary did not operate on tainted data, this means that this instruction did not need to be instrumented in order to detect the attack, and the recipient simply removes instrumentation on this instruction from the hardened binary. By construction, the refined hardened binary will detect the exploit attack and the recipient will then install the refined hardened binary. When the node further disseminates the SVAA, it will remove the redundant instruction addresses from the original VSEF filter.

The whole procedure of verifying a SVAA is shown in Figure 3. As shown in section 5, verifying a SVAA is extremely fast, usually taking less than a second.

Combining SVAA Combining multiple SVAA's is straight-forward: each SVAA contains a list of instruction addresses that can be independently combined. In addition, SVAA's are idempotent: if a site accidentally hardens based on two SVAA's for the same vulnerability, the result is the same as hardening based upon the larger of the two anti-body's. Finally, unlike patches, multiple progressive SVAA's do not need to be received in any particular order.

Security Analysis The SVAA verification procedure is secure, i.e., an attacker will not be able to introduce a false alert and anti-body to affect recipients' normal program execution. Attackers may try and introduce

false alerts into the system in one of four ways (or in combination): (a) introduce alerts that block legitimate traffic (invalid antibodies), (b) introduce valid alerts that are longer than necessary (sub-optimal), (c) introduce invalid alerts in which the anti-body does not detect the vulnerability, or (d) the alert is completely invalid (exploit does not work). We describe how Sting handles each of these cases, thus SVAA's guarantee the desired properties.

A false alerts that contain non-working exploit or invalid SVAA (c and d above) is detected because the hardened binary cannot be verified with th exploit. For (a), we note that self-hardening does not introduce false positives – VSEF instrumentation only returns an error when tainted data is used in an unsafe way. The VSEF will not change how legitimate requests are processed (as described in Section 2.4). Therefore, any SVAA filter that a recipient accepts contains at the least a VSEF that verifies the exploit. Finally, an attacker may attempt to introduce a sub-optimal VSEF (b above), i.e., the VSEF contains more instructions to instrument than necessary to detect the exploit. Our solution to this problem is to introduce the refinement step as described above.

Comparison to Vigilante The high-level idea described in this section is similar to Vigilante [17] which has proposed alert verification as a necessary component for alert dissemination. There are a few differences. Our Self-Verifiable Antibody Alerts includes the actual anti-body, where the disseminated alert in Vigilante only contains the alert, not the actual anti-body. By containing the anti-body, our Self-Verifiable Antibody Alerts verification and self-harden step could be faster than in Vigilante where the receiver has to generate the anti-body.

Moreover, Vigilante alerts modify the original exploit by essentially "pasting" the address of a verification routine into the code in place of the attackers code [17]. Verification involves executing the program given the modified exploit and seeing if the verified routine is called. Vigilante, like Sting, requires that the vulnerable program be executed in a safe environment. Vigilante's method only applies to a limited class of simple programs because pasting in new return values may break the exploit. For example, the protocol may contain a checksum over messages that is invalidated when the verification routine address is pasted in. In this case Vigilante would have to perform additional and probably application-specific actions to create a successful exploit that can be subsequently verified. In other cases the input could be decoded before being used, so simply pasting the verification routine address in the input results in the jump address being overwritten with a completely different and unusable decoding of the verification routine address, and thus the exploit will fail to jump to the verification function and hence the verification procedure will fail. In general, finding the right value to paste in the exploit to have it jump to the verification function can involve understanding and modeling how the input may be manipulated by the program until reaching the vulnerability point – an extremely difficult if not impossible task in some cases. Our verification method is more general as it does not require any modification of the exploit attack. And Vigilante could simply use our verification method to address the aforementioned issue.

3.3 SVAA Distribution

The design of SVAA enables a recipient to verify the validity and quality of the disseminated alert and anti-body. However, we also need to have a fast and robust dissemination mechanism to guarantee that a legitimate SVAA can reach the Sting participants quickly after it is created. The dissemination system should be robust against distributed denial-of-service attacks, and also should not be used by attackers as a mechanism to disseminate the attack. There has been a great volume of work done in building a fast and robust dissemination/broadcast system such as secure peer-to-peer systems [61, 54, 14, 43, 7, 13] which we can employ for our purpose. In this paper, due to space limitations we consider the problem of building a fast, robust, and secure dissemination system as out of scope and hence we do not describe the details here. As a proof of concept, Vigilante describes a distribution architecture utilizing secure peer-to-peer systems which we can simply use [18].

4 Implementation

Here we describe the implementation of the self-monitoring, self-diagnosis (including SVAA generation), self-hardening, self-recovery, and SVAA verification components.

4.1 Self-monitoring

There are many techniques that could be used as a light-weight detector in Sting. As a proof of concept, we use the stack randomization that is already built into RedHat 9 in our experiments. When a process is started, the stack is placed at a random offset. This causes attacks that inject code into the stack to crash with high probability. Automated diversity mechanisms such as this one detect many attacks and have little or no performance overhead when processing non-attack requests. There are many more sophisticated diversity mechanisms that could easily be used [4, 9, 10, 16, 23, 25, 59], some of which are already included in current Linux distributions.

We build on top of FlashBack [52] to perform checkpointing and logging. FlashBack is a tool to efficiently take checkpoints of a process, roll back the state of a process to a previous checkpoint, and perform deterministic execution replay using a system call log (generated by a modified syscalltrack [56]). We customized FlashBack to fit into the Sting system, and extended its ability to handle multiple checkpoints, and to correctly log and replay a wider range of system calls.

4.2 Self-diagnosis

We implement the self-diagnosis engine on top of the TaintCheck dynamic taint analysis tool [40], which is described in Section 2.3. We perform self-diagnosis by rolling back to a previous checkpoint, and replaying execution from that checkpoint, while using TaintCheck to monitoring the replayed execution.

TaintCheck is currently implemented using Valgrind [37], which performs run-time binary rewriting. Unfortunately, while there is no fundamental reason why it could not be implemented, Valgrind's current implementation is not able to attach to a running process. In our current prototype, we start the monitored software under Valgrind, but only add TaintCheck instrumentation when performing self-diagnosis. This is implemented by invalidating Valgrind's cache of already-translated instruction pages (which TaintCheck initially didn't add instrumentation to), and adding the appropriate instrumentation when Valgrind asks TaintCheck to translate them again. However, it would be straight-forward for us to implement TaintCheck using a tool that is able to attach to running processes, such as PIN [34] or dynInst [2], allowing the monitored process to run natively except when self-diagnosis is being performed.

TaintCheck also logs how tainted data is propagated. When an attack is detected, the self-diagnosis engine performs backtracing analysis from the point that tainted data was misused to identify which request is malicious, and to construct the VSEF filter- the list of instruction addresses that propagated the tainted data from the input point to the point where it was misused. The malicious request and the VSEF filter are included in the SVAA. The backtracing analysis also identifies which calls in the system call log correspond to receiving the attack request, which is used in self-recovery.

4.3 Self-hardening

Part of the output of self-diagnosis is the VSEF filter, *i.e.* the list of instruction addresses that TaintCheck needs to instrument to detect attacks against that vulnerability.

We use the VSEF filter to harden the vulnerable binary in memory, without needing to restart the vulnerable process. We use the same technique as in self-diagnosis to notify TaintCheck to invalidate the

translations for the code pages containing the instruction addresses in the VSEF filter, and then add the appropriate instrumentation. Again, it would be straight-forward for us to use other tools such as PIN [34] or dynInst [2], which would allow us to attach to a natively running process and harden it without restarting it.

4.4 Self-recovery

The self-recovery engine first identifies the most recent checkpoint that was taken before all system calls in the system call log that correspond to the malicious request. These system calls are identified in the self-diagnosis step when dealing with a new attack, or by the hardened binary itself once it has been hardened. The self-recovery engine then rolls back the state of the process to that checkpoint.

As discussed in Section 2.5, the next step is to allow the process to execute while replaying from the system call log, so as to get the process back into a state that is consistent with external state. For system calls that do not modify external state (Type 1), and system calls that modify external state other than the attacker's state (Type 3), we replay the return values of the system calls from the system call log.

We prevent the process from becoming corrupted again by modifying the return values of the system calls that correspond to communication with the attacker (Type 2). In the case where the attack request is a TCP stream, we accurately replay the `accept` from the system call log. However, the first time that a read or write is attempted to the corresponding file descriptor, we simulate that the connection has been closed by the attacker. For example, we return 0 to a `read` system call. We expect that in most cases the process will handle this smoothly, and move on to `accept`ing the next request.

When the attack request is a UDP message, we simply skip replaying the corresponding `recvfrom`. That is, when the process makes the `recvfrom` call during replay, and the next `recvfrom` on that socket in the system call log originally was the attack request, we instead move on to the *next* identical `recvfrom` call in the system call log, and return that data. If there are no such identical calls later in system call log, and there are still Type 3 system calls that need to be replayed, then we abort replay and restart the process. If there are no more Type 3 system calls remaining to be replayed, we can resume normal execution- that is, allow the `recvfrom` to execute normally, and wait for the next incoming request.

When all Type 3 system calls have been replayed, recovery is successful, and we resume normal execution. As we showed in Section 2.5, we can *guarantee* correctness in this case. If the program makes a system call that cannot be replayed (that is, there are no matching system calls after the current position in the system call log) before all Type 3 system calls have been replayed, then we cannot reliably bring the program's state into consistency with external state, and hence cannot guarantee correctness of recovery. We then abort replay, and instead fall back on restarting the process.

4.5 SVAA Verification and Refinement

A Sting participant that receives an SVAA for a vulnerable program that it is running verifies it in the following way. First, it uses the VSEF filter from the SVAA to generate a hardened binary candidate, as described in the self-hardening step above. It should then run the hardened binary candidate in a sandboxed environment such as an isolated virtual machine. This could be implemented using tools such as VMware [28] or Xen [8].

The next step is to execute the hardened binary candidate in the sandboxed environment, and send it the attack request from the SVAA. In the simple case, the original network stream can be sent to the hardened binary without modification. However, for protocols that contain state about the session, the network stream may have to be modified. For example, the server may send the client an identifier (such as a handle to a resource) that the client must use later in its network stream. Using the identifier from the logged network stream instead of the identifier that the server actually returned could cause the attack to fail. Techniques for successfully replaying a network stream are presented in [21].

Component	ATPhttpd	smbd
Self-diagnosis	394 ms	651 ms
Self-hardening	195 ms	410 ms
Self-recovery	7.04 ms	12.3 ms
SVAA verification	589 ms	1061 ms

Figure 4: Time for Sting response to a new attack

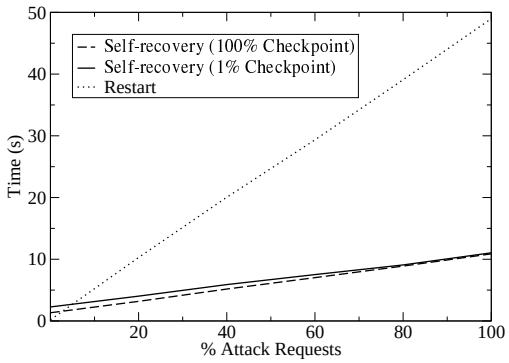


Figure 5: ATPhttpd performance under attack

If the hardened binary candidate does not detect an attack, the SVAA is rejected. In this case, either the exploit does not work, or the VSEF filter does not include the correct set of instruction locations that need to be instrumented to detect the attack.

When the hardened binary candidate does detect an attack, part of the output generated is the set of instruction locations that were actually involved in propagating the tainted data from the point of input to the point where it was misused, and the instruction location where it was misused. This is the set of instruction locations that is instrumented in the final hardened binary. Hence, if an attacker distributes an SVAA with a real attack but a larger set of instruction locations to instrument than is necessary (to trick participants into creating slow hardened binaries), then this step will identify the correct subset of locations that actually must be instrumented.

5 Self-healing System Evaluation

We performed several experiments to verify the functionality of Sting and to measure its performance. Our experiments were performed on a Pentium 4 2.20 GHz with 1 GB of RAM. They were performed inside of a VMware virtual machine (version 5.0) running RedHat 9.

As described in Section 4.2, TaintCheck is currently unable to attach to a running process, because Valgrind does not support it. It would be straightforward for us to implement TaintCheck on other tools that can attach to a running process, such as such as PIN [34] and Dyninst [2]. For now, we use execution under Valgrind with no added instrumentation as the baseline, to simulate being able to attach to a running process.

We use two vulnerable servers in our evaluation. ATPhttpd 0.4 [42] is a web server that is vulnerable to a buffer overflow attack. The Samba daemon (smbd) 2.2.8 implements Windows file and print services, and is also vulnerable to a buffer overflow attack.

5.1 Normal performance overhead

We first measure performance of the ATPhttpd and smbd servers when monitored by Sting. The performance overhead for using address space randomization in our experiments is negligible, so the real performance overhead comes from the checkpointing and system call logging which we explain below.

We first measured the time taken to perform individual checkpoints. We found the latency of the checkpoint operation to be 1.74 ms for smbd and 2.99 ms for ATPhttpd.

We next measure overall performance for each server when performing self-monitoring. We measure ATPhttpd's performance by measuring the time taken to serve 1000 1K static pages. We measure smbd's performance by using it to mount a file system, and then measuring the transactions per second achieved by

the postmark benchmark on that file system (using default settings). For each experiment we measure the cost of performing system call logging and taking a checkpoint once per 100 requests.

In our experiments, self-monitoring added 42.8% overhead for smbd, and 51.5% for ATPhttpd. Thus, we show that the performance overhead of our self-monitoring is orders of magnitude lower than using heavy-weight detectors alone [40], with the additional benefit that the information stored by self-monitoring allows fast and safe recovery. We expect that we can further reduce the performance overhead significantly: Similar experiments show an overhead of checkpointing and system call logging at most 10% [26, 52].

5.2 Performance and Availability Evaluation under attack

We next measure the performance of Sting when protecting the ATPhttpd and smbd servers from attacks.

Response time to a new attack Figure 4 shows the time taken by each Sting component when first receiving the new attack. When a server first receives a new attack, address randomization causes the server to crash. The self-diagnosis engine then rolls the process back to a previous checkpoint and replays the attack while monitoring with TaintCheck, which identifies the attack request and generates the SVAA. This self-diagnosis step takes 394 ms for ATPhttpd, and 651 ms for smbd.

In the next step, the self-hardening module adds instrumentation to the vulnerable binaries to reliably detect other attacks against the same vulnerability in the future. This self-hardening step takes 195 ms for ATPhttpd, and 410 ms for smbd.

Next, the self-recovery module rolls back the process again (to the same checkpoint), modifies the system call log so that the attack does not occur again, and replays the rest of the system calls. This self-recovery takes 7.04 ms for ATPhttpd and 12.3 ms for smbd.

Hence, the time taken to diagnose, recover from, and harden against a new attack is roughly one second.

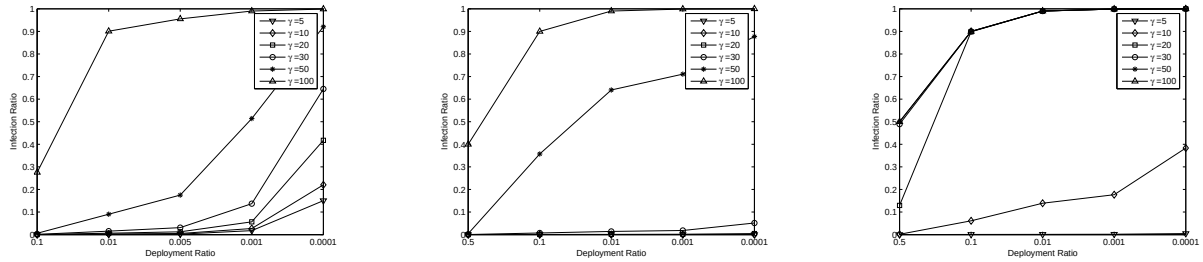
In the collaborative case, we then distribute the SVAA, which contains the attack request and the VSEF filter. The time to generate a hardened binary candidate and verify that it detects the attack is 589 ms for ATPhttpd, and 1061 ms for smbd.

Availability and performance measure under a continuous attack We next evaluate the effectiveness and performance of a Sting-protected ATPhttpd server under a continuous attack, once it has already generated the hardened binary.

We first measure the performance cost of using the hardened binary. We found that the latency to serve a request for a 1K static page was only 6.33% higher than with the vulnerable binary.

Figure 5 shows ATPhttpd's performance when receiving a mix of legitimate requests and attack requests. We send the server 100 requests. Some fraction of these are attack requests, and the rest are requests for a static 1 K file. In the baseline case, the server is hardened against the attack, but does not use self-recovery. Note that a hardened program will detect an attack on the same vulnerability, however, by the time of the detection, part of the memory state may have already been corrupted (although no harm is done yet). Hence, the server must be restarted each time it receives an attack. As described below, some connections fail as a result of the server being restarted. Failed connections are retried until they succeed. In contrast, the Sting hardened binary server performs self-recovery each time it receives an attack. We performed this experiment both with taking a checkpoint before every request, and with taking a checkpointing every 100th request.

Figure 5 shows the results for this experiment. We measure the time it takes for the server to complete 100 requests in the restart case vs. the self-recovery case, as shown in the y-axis. In each experiment, we vary the fraction of the requests that are attack requests, as shown in the x-axis. Overall, as the fraction of attack requests increases, the performance of the restart case significantly worsens compared to the Sting self-recovery case. For example, when the fraction of attack request approaches 1, the restart case has 5 times performance overhead as the Sting self-recovery case, showing that the self-recovery is much more efficient than simple restart. In addition, in our experiments, each server restart results in approximately 20 failed connection attempts, and 1 dropped connection (which the client had established before the server



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

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

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

Figure 6: Effectiveness of Community Defense

crashed). In contrast, the Sting-protected server is able to efficiently perform self-recovery without allowing any legitimate connections to fail.

Note that checkpointing after every request is somewhat more expensive when there are few attacks. However, it reduces the cost of self-recovery, since there are no legitimate requests that need to be replayed after the checkpoint. This suggests that in practice an adaptive approach is most efficient, where checkpoints are taken infrequently when not receiving any attacks, and more frequently when the frequency of attacks rises.

6 Reactive Anti-body Defense and Proactive Protection against Hit-list Worms

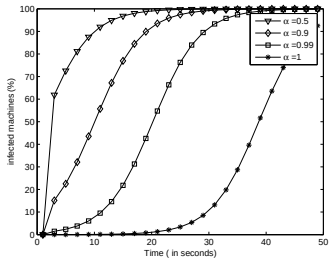
6.1 Reactive Anti-body Defense Evaluation

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. Due to space limitations, we leave the detailed analysis to Appendix A, and only briefly highlight the evaluation result below.

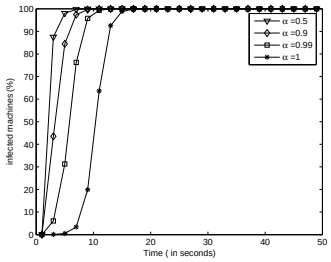
Defense against Slammer worm Figure 6(a) shows the overall infection ratio vs. the producer deployment ratio α for a Slammer worm outbreak (where $\beta = 0.1$ [36]) with different response time r. For example, the figure indicates that given $\alpha = 0.0001$ and $r = 5$ seconds, the overall infection 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 7(c) shows the result of a hit-list worm for $\beta = 1000$ and $\beta = 4000$, and $n = 100,000$[2]. 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 [35, 53] which shows that a hit-list worm can propagate through the entire Internet within a fraction of a second.

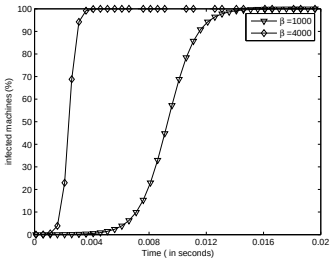
[2]This is basically the same parameters as the Slammer worm, except that instead of a random scanning worm, the worm is a hit-list.



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



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



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

Figure 7: Defense Effectiveness Evaluation

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.

6.2 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 7(a) and 7(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 7(b).

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

6.3 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 6(b) and Figure 6(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 (Section 5) 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 [17]. Thus our system achieves an $r = 2 + 3 = 5$, demonstrating that our system is the first to effectively defend against even hit-list worms.

7 Related

Our work is most related to a nice recent work, Vigilante [17], although the works are largely done independently. There are several important technical differences between the two as we explained earlier in the paper. Unlike Sting, Vigilante does not provide self-recovery, and also does not allow the seamless combination of light-weight detectors and heavy-weight detectors and thus lose the benefits as described in Section 2.2. The two systems generate different anti-bodies and disseminate different alerts as described in Section 2.4 and 3. Finally, we show that reactive anti-body defense, similar to Vigilante, is not fast enough to defend against extremely fast worms such as hit-list worms, and propose a hybrid defense strategy of combining reactive anti-body defense with proactive protection and demonstrate for the first time that it is possible to defend agasint extremely fast worms such as hit-list worms.

Several approaches have been recently proposed to automatically generate input-based filters either from syntactic properties of the input [30, 31, 50, 39] or from program execution [40, 38]. However, as shown in [19], these input-based filters fail to be effective in many cases. Shield is an interesting approach that uses hand-written protocol state machines to provide more accurate host-based filters [58].

Automatically generating patches when source code is available was explored in [47, 48]. Since source code is required, their methods are not applicable to COTS. In addition, the generated patches cannot be verified.

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 [44]. Instead of rolling back execution to a known safe point, Sidiroglou et al have explored aborting the active function when an error is detected [49]. 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 [51]. All of these approaches require source code access, and thus do not apply to COTS applications.

There is a considerable body of research on rollback schemes: see [41] for a more detailed discussion. We choose to use FlashBack [52], 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 [22, 29]. 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.

We use TaintCheck [40] to perform dynamic taint analysis on the binary for self-diagnosis. Others have implemented similar tools [17] which can also be used. Hardware-assisted taint analysis has also been proposed [55, 20]. Unfortunately, such hardware does not yet exist, though we can take advantage of any developments in this area.

Our use of address space randomization is different than analyzed in Shacham et al [46]. Shacham et al show address space randomization can be brute-forced, and conclude that randomization should not be relied upon. Our modeling shows that despite this fact, address space randomization can be used to slow down worms and is thus an important component in defending against extremely fast worms.

Anagnostakis et. al. proposes a nice approach, shadow honeypot, to enable a suspicious request to be examined by a more expensive detector [6]. However, their approach requires source code access and

manual identification of beginning and end of transactions and thus does not work on COTS and blackbox applications. In addition, because they only reverse memory states but 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 (as explained in Section 2.2), whereas our retro-active random sampling addresses this issue.

Rx proposes a nice approach to use environmental changes to defend against failures and uses rollback and perturbation to enable environmental changes [41]. However, their approach does not support detailed self-diagnosis and self-hardening, and simply retries different environmental changes.

Liang and Sekar [33] and Xu et. al. [60] 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. However, their analysis and applicability are limited. Liang and Sekar's approach does not work for sophisticated programs where static binary analysis is difficult and they do not provide detailed diagnosis and their signature generation does not work in many cases (for example, if the inputs are processed or decoded before it is used to cause a buffer overflow). The analysis in Xu et. al.'s approach is also limited, and their signature suffers from similar problems as described in [19]. In addition, these approaches purely rely on the address space randomization protection which can be evaded by many attacks, while our approach enjoys the accuracy of heavy-weight (more accurate) detectors because of the random sampling inspection in our approach. Finally, our approach is much more general as we allow different light-weight and heavy-weight detectors to be plugged into our system.

8 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 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] Dyninst. www.dyninst.org.

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

[4] PaX. http://pax.grsecurity.net/.

[5] The design and architecture of the microsoft cluster service. In *Proceedings of the* $28th$ *Annual International Symposium on Fault-Tolerant Computing*, 1998.

[6] 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.

[7] D. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient overlay networks. In 18^{th} *ACM SOSP*, October 2001.

[8] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, Oct. 2003.

[9] S. Bhatkar, D. 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.

[10] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium*, 2005.

[11] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 1989.

[12] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot - a technique for cheap recovery. In *Proceedings of the* 6^{th} *Symposium on Operating System Design and Implementation*, 1999.

[13] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. Wallach. Secure routing for structured peer-to-peer overlay networks. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2002.

[14] M. Castro, P. Druschel, Y. Hu, and A. Rowstron. Exploiting Network Proximity in Peer-to-peer Networks. Technical Report MSR-TR-2002-82, Microsoft Research, 2002.

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

[16] M. Chew and D. Song. Mitigating buffer overflows by operating system randomization. Technical report, Carnegie Mellon University, 2002.

[17] M. Cost, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In 20^{th} *ACM Symposium on Operating System Principles (SOSP 2005)*, 2005.

[18] M. Costa, J. Crowcroft, M. Castro, and A. Rowstron. Can we contain internet worms? In *HotNets 2004*, 2004.

[19] J. Crandall, Z. Su, S. F. Wu, and F. 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.

[20] J. R. Crandall and F. Chong. Minos: Architectural support for software security through control data integrity. In *International Symposium on Microarchitecture*, December 2004.

[21] W. Cui, V. Paxson, and N. Weaver. Network replay. In *Proceedings of NDSS*, 2006.

[22] G. Dunlap, S. King, S. Cinar, M. Basrai, and P. 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.

[23] D. C. DuVarney, R. Sekar, and Y.-J. Lin. Benign software mutations: A novel approach to protect against large-scale network attacks. Center for Cybersecurity White Paper, October 2002.

[24] A. for submission. Semantic-based automatic signature generation against zero-day exploits. Technical report, Anonymized institution for submission, 2005.

[25] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *Proceedings of 6th workshop on Hot Topics in Operating Systems*, 1997.

[26] A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara. The taser intrusion recovery system. In *2005 Symposium on Operating System Principles (SOSP)*, 2005.

[27] H. W. Hethcote. The Mathematics of Infectious Diseases. *SIAM Review*, 42(4):599–653, 2000.

[28] V. Inc. http://www.vmware.com/.

[29] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 2005 Symposium on Operating Systems Principles (SOSP)*, 2005.

[30] H.-A. Kim and B. Karp. Autograph: toward automated, distributed worm signature detection. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

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

[32] R. Lemos. Counting the cost of the slammer worm. `http://news.com.com/2100-1001-982955.html`, 2003.

[33] Z. 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.

[34] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation (PLDI)*, 2005.

[35] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the slammer worm. In *IEEE Security and Privacy*, volume 1, 2003.

[36] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the slammer worm. In *IEEE Security and Privacy*, volume 1, 2003.

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

[38] J. Newsome, D. Brumley, D. Song, J. Chamcham, and X. Kovah. 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.

[39] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2005.

[40] J. Newsome and D. 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.

[41] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies—a safe method to survive software failures. In 20^{th} *ACM Symposium on Operating System Principles (SOSP)*.

[42] Y. Ramin. ATPhttpd. http://www.redshift.com/∼yramin/atp/atphttpd/.

[43] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Topologically-Aware Overlay Construction and Server Selection. In *Proceedings of IEEE INFOCOM*, 2002.

[44] M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and W. B. Jr. Enhancing server availability and security through failure-oblivious computing. In *Operating System Design & Implementation (OSDI)*, 2004.

[45] D. Scott. Assessing the costs of application downtime, 1998.

[46] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, October 2004.

[47] S. Sidiroglou and A. 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.

[48] S. Sidiroglou and A. D. Keromytis. Countering network worms through automatic patch generation. *IEEE Security and Privacy*, 2005.

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

[50] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. Technical report, December 2004.

[51] A. Smirnov and T. 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.

[52] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the 2004 USENIX Technical Conference*, 2004.

[53] S. Staniford, V. Paxson, and N. Weaver. How to 0wn the Internet in your spare time. In *11th USENIX Security Symposium*, 2002.

[54] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM 2001*, San Diego, CA, USA, August 2001.

[55] G. E. Suh, J. Lee, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of ASPLOS*, 2004.

[56] syscalltrack. `http://syscalltrack.sourceforge.net/how.html`.

[57] W. Vogels, D. Dumitriu, A. Agrawal, T. Chia, and K. Guo. Scalability of the microsoft cluster service. In *Proceedings of the* 2^{nd} *USENIX Windows NT Symposium*, 1998.

[58] H. J. Wang, C. Guo, D. Simon, and A. Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *ACM SIGCOMM*, August 2004.

[59] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. Technical report, Center for Reliable and Higher Performance Computing, University of Illinois at Urbana-Champaign, May 2003.

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

[61] B. Zhao, K. Kubiatowicz, and A. Joseph. Tapestry: An Infrastructure for Fault-Resilient Wide-Area Location and Routing. Technical Report UCB//CSD-01-1141, University of California at Berkeley, April 2001.

A Worm Modeling for Defense Evaluation

Notation. Worm propagation can be well described with the classic Susceptible-Infected (SI) epidemic model [27]. Let β be the average contact rate at which a compromised host contacts vulnerable hosts to try to infect them, t be time, N the total number of vulnerable hosts. Let $I(t)$ represent the total number of infected hosts at time t. Let α be the fraction of vulnerable hosts which are Sting producers, and the remaining population are Sting consumers. (Note that for simplicity, we assume that all the vulnerable hosts participate in Sting as producers or consumers; it can be easily extended to the case where some vulnerable hosts do not participate in Sting.) Let P_t be the total number of producers contacted by at least one infection attempt at time t.

A.1 Defense against Slammer Worm

From the SI model, before T_0 where $P(T_0) = 1$ (before any Sting producer is contacted by an infection attempt), we have:

$$\frac{dI(t)}{dt} = \beta I(t)(1 - \alpha - I(t)/N) \tag{1}$$

$$\frac{dP(t)}{dt} = \alpha\beta I(t)(1 - P(t)/(\alpha N)) \tag{2}$$

We can solve the above equation for T_0. Once a Sting producer is contacted with an infection attempt, it takes time r_1 until the producer creates a SVAA using self-diagnosis, and then it takes time r_2 until the SVAA can be disseminated to all other vulnerable hosts. It takes time r_3 for a Sting consumer to verify the SVAA and install the anti-body. Let $r = r_1 + r_2 + r_3$, and we call r the response time of Sting. Thus, after time $T_0 + r$, all the vulnerable hosts have received and installed the anti-body and become immune to the worm outbreak. Thus, the total number of infected

hosts throughout the worm outbreak is $I(T_0 + r)$, and $I(T_0 + r)/N$ is the infection ratio. We plot the infection ratio vs. α for different r in Figure 6(a), using the parameters of Slammer worm (where $\beta = 0.1$ and $N = 100000$ [36]).

A.2 Defense against Hit-list Worm

We showed above that the Sting collaborative community defense is extremely effective against some of the fastest scanning worms such as the Slammer worm attack. However, for even faster worms such as a hit-list worm or flash worm which can reach all the vulnerable hosts within seconds, the Sting reactive anti-body defense may not be fast enough when the fraction of producers is low. To address this problem, we propose a hybrid defense where the Sting consumers would deploy a proactive protection mechanism such as address randomization defense in addition to participate in Sting to receive SVAA. The proactive protection mechanism such as address randomization techniques alone does not provide a complete defense, it simply makes an attack harder to succeed, e.g., the attacker needs to do multiple trials until it guesses the correct program internal state. When combining the proactive protection mechanism with the Sting reactive anti-body defense, the proactive protection mechanism can slow down the worm outbreak until the Sting reactive anti-body defense can create and disseminate the SVAA to all the vulnerable hosts. We illustrate the analysis below.

Let ρ be the probability that each infection attempt could succeed on a vulnerable host deploying a proactive protection mechanism such as address randomization. Before T_0 where $P(T_0) = 1$ (before any Sting producer is contacted by an infection attempt), we have:

$$\frac{dI(t)}{dt} = \beta\rho I(t)(1 - \alpha - I(t)/N) \tag{3}$$

$$\frac{dP(t)}{dt} = \alpha\beta I(t)(1 - P(t)/(\alpha N)) \tag{4}$$

Similar to the analysis in Section A.1, we can solve the above equations for T_0, and then calculate the total number of infected hosts throughout the worm outbreak $I(T_0 + r)$, and $I(T_0 + r)/N$ is the infection ratio. We plot the infection ratio vs. α for different r in Figure 6(b) and 6(c), assuming $\rho = 2^{-12}$, $N = 100000$, and $\beta = 1000$ and 4000 respectively.

B VSEF

When the hardened binary is run, the added instrumentation propagates taint information as the program executes. If the detection point is reached, the added instrumentation checks to see if the sensitive operand (e.g., return address or function pointer) is tainted. When the self-hardened binary is run with benign input, then taint information will not be used in a sensitive way (by definition) at the exploit point and the program will execute as normal.

Since we are not instrumenting all data movement and arithmetic instructions, locations that have been marked as tainted may be overwritten with untainted data by uninstrumented instructions. This could lead to false positives if, for example, 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 prevent this problem by recording the value that a location takes on when we mark it as tainted. When we later check to see if that location is still tainted, we can check to see if it still has the same value. If not, then it has been overwritten by an uninstrumented instruction, and we mark it as no longer tainted.

Performance. Note that the execution overhead of the hardened program is proportional to the number of instructions instrumented. Our experiments show that the number of instructions that need to be instrumented is usually small, and as a result the performance overhead of a hardened binary is usually small, e.g., only a few percentage.

Accuracy. The VSEF-hardened binary has no false positives. 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.

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 {poly,meta}-morphic variants created by tools such as MetaSploit [3] will be detected from a single filter. The reason is such {poly,meta}-morphic variants differ in the payload which would be executed strictly after exploit point. Only an exploit that is polymorphic in the execution path exploited will be missed.

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 [38].

Combining filters. We may want to combine several different VSEF's. 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 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.