Sweeper: A Lightweight End-to-End System for Defending Against Fast Worms

Joseph Tucek
Shan Lu, Chengdu Huang, Spiros Xanthos
Yuanyuan Zhou

University of Illinois at Urbana Champaign
tucek, shanlu, chuang30, xanthos2, yyzhou@cs.uiuc.edu

James Newsome
David Brumley
Dawn Song

Carnegie Mellon University
jnewsome@ece.cmu.edu, dbrumley, dawnsong@cs.cmu.edu

ABSTRACT

The vulnerabilities that plague computers cause endless grief to users. Slammer compromised millions of hosts in minutes; a hit-list worm would take under a second. Recently proposed techniques respond better than manual approaches, but require expensive instrumentation, which limits deployment. Although spreading "antibodies" (e.g. signatures) ameliorates this limitation, hosts depending on antibodies are defenseless until inoculation; to the fastest hit-list worms this delay is crucial. Additionally, most recently proposed techniques cannot provide recovery to provide continuous service after an attack.

We propose a novel solution called Sweeper that provides both *fast and accurate* post-attack analysis and *efficient* recovery *with low normal execution overhead*. Sweeper innovatively combines several techniques: (1) Sweeper uses *lightweight* monitoring techniques to detect a wide array of suspicious requests, providing a first level of defense. (2) By cleverly leveraging lightweight checkpointing, Sweeper postpones heavyweight monitoring until absolutely necessary — after an attack is detected. Sweeper rolls back and re-executes multiple times to dynamically apply heavyweight analysis techniques via dynamic binary instrumentation. Since only the execution involved in the attack is analyzed, the analysis is efficient, yet thorough. (3) Based on the analysis results, Sweeper automatically generates low-overhead antibodies to prevent future attacks of the same vulnerability. (4) Finally, Sweeper again re-executes to perform fast recovery for continuous service.

We implement Sweeper in a real system. Our experimental results with three real-world servers and four real security vulnerabilities show that Sweeper can detect an attack and generate antibodies in under 60 milliseconds. Our results also show that Sweeper imposes under 1% overhead during normal execution, clearly suitable for widespread production deployment (especially since Sweeper also allows partial deployment). Finally, we analytically show that, for a fast hit-list worm otherwise capable of infecting all vulnerable hosts in under a second, Sweeper contains the extent of infection to under 5%.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EuroSys'07, March 21–23, 2007, Lisboa, Portugal.
Copyright 2007 ACM 978-1-59593-636-3/07/0003 ...$5.00.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection; C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*

General Terms

Security, Performance, Reliability, Design

Keywords

Flash worm, Antibody, VSEF, Dynamic instrumentation

1. INTRODUCTION

1.1 Motivation

Modern society relies on computers; failures of these computers can cost upward of six million dollars per hour [41]. Unfortunately, much software contains security vulnerabilities, with memory overwrite vulnerabilities (e.g., buffer overflows) accounting for over 60% [11]. These vulnerabilities allow self propagating worms, such as Code Red [9], Blaster [8], and SQL Slammer [10], to rapidly do billions of dollars of damage [27]. Advanced worms demonstrate that manual patching is insufficient–by the time an administrator reads an alert about the worm, they are already infected [48].

Since worms attacks are clearly too fast for humans, an automated response is imperative. Consider a hypothetical ideal automatic worm defense with the following behavior. If a worm attempts to infect it, the defense system first detects the attack. It then analyzes the attack attempt to find the underlying vulnerability. Without any human assistance, it devises a shareable "antibody" suitable for stopping all attack attempts of this vulnerability (not just this particular exploit) with no false positives. After the analysis, the machine can recover to continue execution as if the worm had not attacked. Finally, the overheads of running the defense system are low enough to allow deployment on all hosts. This ideal defense system leaves no room for worms; wherever they go, they are detected, picked apart, and have the underlying vulnerability they use sealed off. The only trace of the worm's existence is log messages and new antibodies.

Essentially, what we want is an Internet worm defense system that satisfies three properties:

- *Fast and accurate attack detection/analysis*: The defense system needs to detect and analyze the attack efficiently and accurately to prevent damages and future attacks exploiting the same vulnerability.
- *Low overhead for universal deployment*: The defense system has to have low overhead to enable practical deployment on any production system, especially for performance critical server applications.
- *Efficient recovery*: The defense system should recover from an attack as efficiently as possible to provide non-stop service, especially for high availability applications.

Unfortunately, none of the existing defense systems can deliver the above three desired properties. First, although some existing solutions such as PaX [2], StackGuard [15], LibSafe [49], and ProPolice [19] add reasonably low overhead (22%-0%) so that they can potentially deploy universally, they only detect some types of attacks (as shown by a prior work [54]). Address space randomization [2, 20] detects many memory-related vulnerabilities but provides too limited of information about an exploit to analyze the attack and generate antibodies against future exploits. At best, the program will halt at the vulnerable instruction, at worst the attack will (with low probability) succeed. Similarly, stack canaries tell us that the stack was overwritten, but not how it was overwritten. Tools like LibSafe only detect issues in the specific library functions they target. We can deploy such systems, but we will not learn much from them.

Second, most existing solutions that provide reasonably accurate attack detection and analysis incur too much overhead (up to 30-40X slowdowns [36]) to be practical to deploy universally. Example of these tools include DIRA [45], DACODA [16], Vigilante [14], or our own previous work, TaintCheck [36]. The techniques that can best detect and analyze an attack (e.g., TaintCheck or DACODA) impose the highest overheads. To provide detailed analysis of the exploit, they have to instrument most of the instructions, and record many details about what happens. Due to the high runtime overheads, such tools must instead rely on a limited, sentinel- or canary-like deployment. If an unlucky worm happens to infect such a sentinel host, it *will* be caught, but the bulk hosts are unmonitored, open to attack.

A partial remedy proposed in Vigilante [14] is, once caught at a sentinel machine, to analyze the attack and generate antibodies, i.e. SCAs (Self-Certifying Alerts), to efficiently and automatically to quickly distribute to other hosts against infection. Unfortunately fast hit-list worms can, if unimpeded, infect every vulnerable host in milliseconds [47]; in contrast, the time it takes to generate, distribute, and verify an alert in a Vigilante-like system is too long. In summary, none of these remedies completely address the fundamental limitation of most existing solutions, i.e, failure to provide accurate and fast detection and analysis of Internet attacks without incurring too much overhead during normal execution.

In addition to the above limitation, a parallel shortcoming of existing solutions is recovery: most fail to provide efficient recovery because they have to stop the service and restart after an attack. For example, although TaintCheck will identify the improper use of untrusted data and stop execution, our original implementation of TaintCheck cannot undo the bad effects; any overrun buffers will remain overrun. We merely stop the attack, delegating recovery to restart. Unfortunately, restarting a system or an application usually takes up to several seconds [51]. For servers that buffer significant amount of state in main memory (e.g., data buffer caches), it requires a long period to warm up to full service capacity [5, 52].

In summary, to maximize the level of defense against security attacks, it is highly desirable to develop a solution that can meet all three properties, namely fast and accurate detection/analysis, low overhead for universal deployment, and efficient recovery.

1.2 Contributions

To achieve the above goal, we propose a defensive solution called Sweeper [1] that is able to meet the three desired properties described above by innovatively combining several new techniques.

First, by leveraging a lightweight checkpointing and monitoring support, we postpone heavyweight monitoring until absolutely necessary — after being attacked. In other words, during normal execution, the system takes only lightweight checkpoints to allow re-execution and recovery in case of an attack. It also performs lightweight monitoring to detect a wide range of suspicious requests that exploit both unknown and known vulnerabilities (detecting unknown attacks through address randomization and low-overhead dynamic memory bug detection, and detecting known attacks through antibodies generated during on-line attack analysis). Both checkpointing and monitoring impose very low overhead, making universal deployment practical. In addition, Sweeper also supports partial deployment (discussed in more detail in Section 6).

After an attack is detected, Sweeper can "go back in time" (i.e., rollback) and *dynamically* add heavy-weight instrumentation and analysis during replay to conduct a thorough attack analysis, including such techniques as memory bug detection and dynamic taint analysis. This analysis results in automatically generated antibodies in the form of input signatures and vulnerability-specific execution filters (VSEF) [33] (antibodies are further discussed in Section 3). Doing such allows detailed analysis to be performed only for those recent messages and execution periods that are relevant to the occurred attack—server initialization and long runs of harmless inputs/normal execution need not suffer expensive monitoring and information recording. This use of checkpoint and rollback allows us to have both low overhead and thorough analysis.

Second, we again leverage checkpoint/re-execution, this time to achieve recovery: after an exploit attempt is detected (and any necessary analysis is performed) we roll back, and re-execute the program while dropping the attacker's input. This allows us to use not only input signatures, but VSEFs as well; without recovery, VSEFs only transform a code-execution vulnerability into a denial-of-service vulnerability.

We implement these ideas in a real system. Our functioning prototype is implemented in Linux, building on a modified version of our previous Rx framework [40]. Sweeper

[1]Like a sweeper in soccer, Sweeper is intended to be fast, tough, and add depth to the defense.

uses address space randomization for lightweight detection, backed by post-exploit analysis tools such as dynamic memory bug detection, dynamic taint analysis [36], and backward slicing [53]. We use the PIN [30] dynamic instrumentation tool to add these analysis tools on-demand. We also implement both input based filtering and VSEFs for defense on both hosts performing analysis and those hosts that choose not to.

As this paper focuses on protecting vulnerable applications, our current design and implementation assumes that the operating system is secure. This assumption is not fundamental because (1) our Sweeper should be able to detect most attacks (known or unknown) before they affect the operating system. (2) If necessary, we can also push most Sweeper's system-level operations into the virtual machine hypervisor in a way similar to ReVirt [18].

We test Sweeper using 4 *real* exploits in 3 servers: Apache, Squid, and CVS. The overhead during pre-attack execution (normal execution) is under 1%, making Sweeper clearly suitable for widespread production deployment. Antibodies can be generated in under 60 ms. Finally, we present analytical results showing that even when partially deployed, Sweeper is capable of containing fast hit-list worms. To summarize, Sweeper has the following advantages compared to previous solutions:

1. **It imposes low overhead during normal execution.** During normal execution, only lightweight monitoring and lightweight checkpointing are active. Lightweight monitoring techniques such as randomization [12, 20, 55] or lightweight dynamic bug detection [17, 39, 58] impose reasonable amount of overhead (nearly zero for address space randomization), feasible for production run deployment. In-memory checkpointing, such as our previous Flashback and Rx works [40, 46], also impose only marginal amounts of overhead (e.g., 1-5%). As we demonstrate in our experimental results (Section 5.1), the low overhead makes widespread production run deployment feasible.

2. **It performs comprehensive and thorough attack analysis, and generates effective antibodies.** Low overhead during normal execution is achieved without sacrificing the analysis power. When the lightweight monitoring trips, we can roll back and re-execute with heavyweight analysis. Sweeper then dynamically uses binary instrumentation tools (e.g., PIN [30]) to insert analysis such as dynamic taint analysis [36] or backward slicing [53] after the fact. Therefore, we do not pay for expensive analysis for requests that do not need it, but only for those requests where it matters.

3. **It allows fast recovery.** Simply detecting that an exploit has been attempted is insufficient; we have to restore the server to a safe state. Once an attack is detected, we can use rollback/re-execution to re-execute without the attacker's input. Rollback removes the corruption the attacker may have left, while re-execution allows us to complete servicing concurrent and further valid requests without restarting the program, thus achieving fast recovery.

4. **It provides partial deployment option to hosts that demand even lower overhead.** Although the overheads involved are low, there may be hosts that do not wish to deploy the analysis tools. We do not leave such hosts completely defenseless. As we show in Section 6, Sweeper also provides an effective community defense option that can protect most hosts even in a hit-list worm attack when only a fraction deploy the Sweeper analysis mechanisms.

2. ARCHITECTURE

2.1 Overview

The Sweeper system has four functions: 1) during normal execution, light-weight monitoring for detecting attacks and light-weight checkpoint for potential rollback-and-re-execution for attack analysis; 2) after an attack, analyzing the exploit attempt via multiple iterations of rollback-and-re-execution; 3) generating and deploying an antibody against future exploits; and 4) recovery after an attack is detected and analyzed.

Figure 1 shows the architecture of Sweeper. The above four functions are provided by three modules: runtime, analysis and antibody. Section 3 describes the details of each component; here we discuss their overall function and their interactions.

Runtime module The run time module supports (1) lightweight monitoring and checkpoint during normal execution, (2) re-execution during attack analysis, and (3) recovery after attack is analyzed. During normal execution, the runtime module employs low overhead monitoring techniques such as address randomization and other techniques discussed in more details in Section 3 to detect suspicious request. Moreover, it also uses input signatures and VSEFs generated by the analysis and antibody modules on past attacks to filter out malicious requests and detect exploits of previously known vulnerabilities. In addition to lightweight monitoring, the run time module also takes periodic light-weight, in memory checkpoints similar to our previous work Rx [40] and FlashBack [46] to ensure rollback-and-re-execution for analysis and recovery in case of attacks.

The checkpoints taken by the runtime module, as well as Sweeper's other private state, are isolated from the process we are protecting. The checkpoints themselves are stored inside the operating system as shadow processes; unless an attacker compromises the operating system's own memory space (contrary to our assumptions described in the introduction), the checkpoints cannot be touched. Further, the analysis tools are applied *after* an attack is detected. They take control of the execution path, and disallow any access to their internal state. After they are applied, no instructions are executed without the instrumentation tool first being given the opportunity to monitor it. In this manner, an attacker is prevented from subverting either the analysis tools or the checkpoints.

After an attack is detected, the runtime module is also responsible for providing rollback and re-execution support as guided by the analysis module to perform various attack analysis. To support re-execution from a previous checkpoint, it needs to replay all of or a selected subset of incoming network messages received since that checkpoint based on the type of analysis performed. During re-execution, all side-effects such as outgoing network messages are sandboxed and silently dropped.

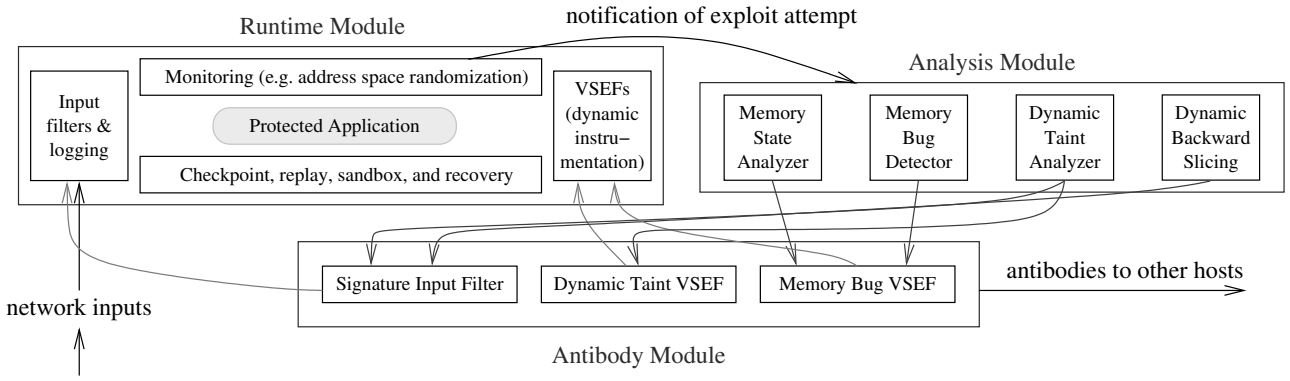

Figure 1: Architecture diagram of Sweeper

Finally, after the attack is analyzed and an antibody is generated, the runtime module rolls back and re-executes again from a selected checkpoint to perform recovery for providing continuous service. The continuing execution will have the new antibody (input signatures and VSEFs) in place to detect future exploits to the same vulnerability. During recovery, the output commit problem and the session consistency are handled in a way similar to our previous work, Rx [40]. We will briefly discuss these issues in Section 4, but for more details refer to our previous Rx work [40].

Analysis module Analysis is performed by the analysis module to generate input filters and VSEFs. The analysis module is activated only when absolutely necessary — after an attack is detected by the light-weight runtime monitors. By using the checkpoint/rollback capabilities of the runtime module, the analysis module can inspect and re-inspect the execution as necessary, going back to a point prior to the attacking requests being read in. Because the execution to be monitored represents only a short amount of time, a few tens of hundreds of milliseconds depending on the checkpoint interval, even expensive analysis tools complete quickly. *Performing heavy-weight analysis only on the periods of execution where it is necessary greatly improves the efficiency of analysis and also enables more thorough and accurate analysis.*

After rollback, the analysis module dynamically attaches various analysis tools that are implemented using dynamic binary instrumentation. There are many possible analysis techniques that could be applied; in our implementation (see Section 3 for details) we perform a static analysis of the memory state, dynamic memory bug detection similar to Valgrind [32] and Purify [22], dynamic taint analysis similar to our previous TaintCheck work [36], and dynamic backward slicing [53]. The overheads of the dynamic techniques range from 20x to *1000x* (for backward slicing). Yet since analysis is only performed when necessary and only on a short execution period that is related to the occurring attack, the total expense is small.

Antibody module The antibody module uses the analysis results to derive antibodies of detect future exploits to the same vulnerability. There are two types of antibodies supported by Sweeper: input signature filters, and vulnerability specific execution filters (VSEF) [33]. Given the input responsible for the exploit, an input signature for filtering can be generated [24, 26, 35, 44]. Also, given the instructions involved in the exploit (especially for buffer overflows),

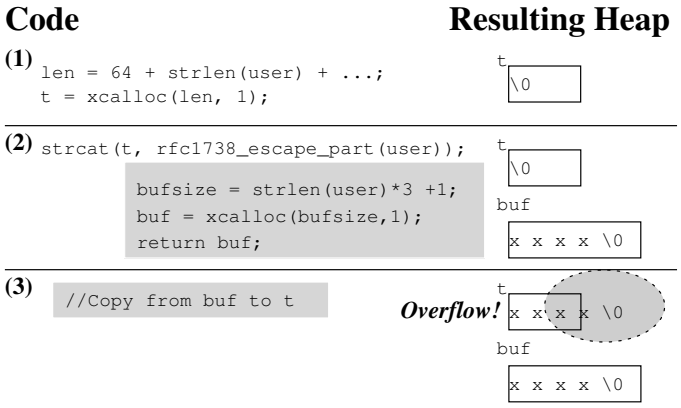

Figure 2: A buffer overflow in Squid (CVE-2002-0068).

we can generate a VSEF. In the case of a memory bug (e.g. stack smashing), the VSEF consists of monitoring the instruction that cause the buffer overflow, or monitoring the return address of the susceptible function. Since these only involve a handful of instructions, these VSEFs are inexpensive. Together, these antibodies are sufficient to prevent future exploit attempts from succeeding. Also, they can be distributed to other hosts. If the other hosts are untrusting, it is sufficient to give them the exploit-containing input; they can then generate their own signatures and VSEFs.

Together, these modules make up the complete Sweeper system. Deploying all of them together is the assumed default case. Ideally, all hosts would use all of the modules; it is possible, and still beneficial, to run only a partial set. This is further discussed in the Section 6.

2.2 Process

To clarify how the system works, we present a concrete walk-though of a real vulnerability. Figure 2 shows an exploitable buffer overflow bug in Squid. In step (1), heap buffer `t` is allocated as `64 + strlen(user)` bytes long. In step (2), the function `rfc1738_escape_part(...)` allocates a buffer `buf` to be `strlen(user) * 3 + 1` bytes long, and then fills it in with an escaped version of the string `user`. In step (3), `buf` is copied into `t` using `strcat(...)`; since `strcat(...)` is not bounds checked, `t` can overflow. The bug is triggered whenever there are many characters that are escaped in the `user` string.

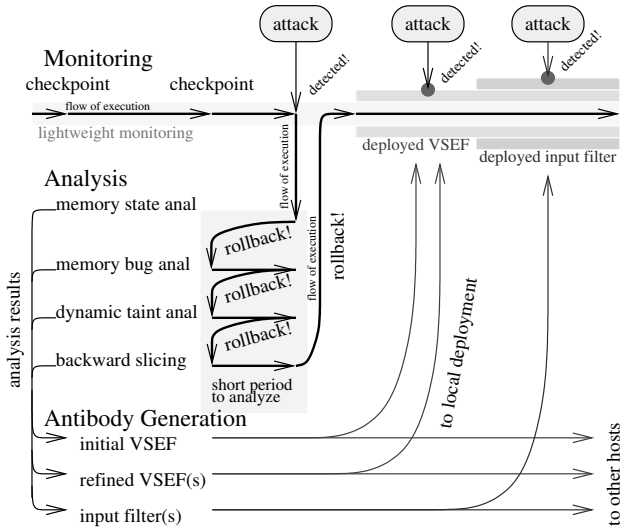

Figure 3: Sweeper defense process.

Figure 3 illustrates the Sweeper defense process. During normal operations, Sweeper takes periodic checkpoints. At an attack, the light-weight sensors and monitors detect that something is amiss—for example, a randomized memory layout has caused a segmentation fault to occur. So, Sweeper begins its attack analysis. The execution is rolled back to the previous checkpoint, and heavier weight analysis techniques are performed. In our current implementation, the first analysis is an examination of the memory state (i.e., analyze the core dump). This is a very fast step, and it generates a good-quality VSEF. In the Squid vulnerability, this tells us that the segmentation fault occurred at instruction 0x4f0f0907 in `strcat`. For this example, this is enough to build an initial VSEF: check for out-of-bounds accesses at that particular instruction. Actually, a small refinement is necessary, since `strcat` is a library function: the return address at that time must also match (0x0804ee82, or `ftpBuildTitleUrl`). Although later analysis steps can be used to detect with more certainty, this VSEF is more effective than the generic sensors and it is available *within only **40ms*** of the first sign of trouble.

Next, memory bug detection is performed. This is more expensive, but it generates improved VSEFs, so we do it second. The analysis includes bounds checking, stack-smashing detection, double free detection, and dangling pointer detection. Monitoring all memory accesses is impractical for normal execution, but since Sweeper can dynamically add instrumentation to a replay from a checkpoint, the overhead is manageable. In our Squid example, the heap is inconsistent, and memory bug detection points out instruction 0x4f0f0907 in `strcat` as the source. This confirms the earlier results, and takes around 37 seconds.

The next step is dynamic taint analysis [36]. This allows us to isolate the input for a signature. Dynamic taint analysis traces the influence of "untrusted" data (e.g. network inputs) through the program, looking for "illegal" uses of tainted data, such as a branch target. Once an illegal use of tainted data is detected, we can trace the taint back to the particular request responsible. The identified request can then be passed on to a signature generator to generate input signatures to filter out further attacks [24, 26, 35, 44]. The identification of the original input responsible for the attack also allows us to do fast recovery: we simply rollback the process and re-execute without the malicious input, and thus bring the process back to a safe state.

The last analysis step is dynamic slicing. The slicing collects the full dependency graph, including data and control flow dependencies, of the instructions executed since the checkpoint. Having the complete set of involved instructions and data allow us to verify the results of the previous results: if they identify an issue that is not in the slice, then they are incorrect. The graph is only for execution on the malicious input, since the checkpoint. *Running full slicing from the very beginning of execution, even in replay, is impractical.* Depending on the program, slicing imposes from 100x to 1000x overhead. Only by dynamically inserting the graph collection from a checkpoint the slicing overhead becomes acceptable and practical for automatic defenses. In the Squid example, within around 107 seconds, Sweeper generates a backward slice that exactly shows the reason of the vulnerability: `t` is allocated too small, and there is no bounds check. Further, none of the other tools report anything outside of the backward slice; if they did, we would suspect that the other tools were incorrect. Backward slicing can then act as a sanity check against the other tools.

In this particular example, everything points to the same instruction, 0x4f0f0907 in `strcat`. The later, more thorough analysis steps serve as a confirmation of the previous steps; here they all fully agree. Consider instead a stack-smashing attack: the crash may occur well after the buffer overflow. Although it is possible to detect from a core dump, and we can create a VSEF (use stack canaries or a separate return-address stack for the effected function), it would be preferable to target the buffer overflow itself. This, however, is not possible until after memory bug analysis is performed. Also, generating a worm signature requires identifying the specific input responsible; again, this is not possible with the simple core analysis. In combining multiple analysis techniques, we get something better than any one; fast but potentially weak results from static analysis can augment slow but thorough results from dynamic analysis.

3. DESIGN AND IMPLEMENTATION

As discussed in Section 2, Sweeper has three components, one for runtime support, one for post-attack exploit analysis, and one for dealing with antibodies. Here we further describe the details of each individual components.

3.1 Runtime Support

During normal execution, Sweeper needs to: 1) monitor against generic attacks, 2) monitor network flows and execution against specific attacks, and 3) take checkpoints sufficient to replay execution for later analysis and recovery. Since these three tasks are being performed continuously, they are performance critical: the higher the overhead imposed, the fewer sites will be willing to sacrifice the performance for protection.

Runtime Monitoring Monitoring against generic attacks can be performed with any lightweight bug detector. In our current prototype implementation, we rely on address space randomization [2, 3, 4, 12, 20, 21, 55], although there are many other mechanisms that could be used [2, 13, 15, 21], including some of our own previous work such as SafeMem [39], LIFT [38]. The advantage of automated diversity

mechanisms like address space randomization, which places the starting point of the stack and heap at a random initial offset and randomizes library entry points, is that they detect many attacks with high probability while imposing minimal performance overhead in processing non-attack requests.

Monitoring for specific attacks has two parts: input monitoring and execution monitoring, based on the antibodies automatically generated by Sweeper's antibody module from past attacks. Monitoring inputs for attack signatures is already widely deployed in network IDS systems. We combine such monitoring with the input logging that is required to support replay; it would be possible to separate the monitoring to a separate machine (e.g. a firewall) if desired. Execution monitoring must occur on the machine in question. We implement execution monitoring by adding dynamic binary instrumentation with PIN [30]. PIN allows the efficient addition of instrumentation to an already running process. However, any instrumentation tool that allows dynamically attaching to a running process would be feasible (e.g. dynInst [1]); we choose PIN due to familiarity and efficiency. Since only a minute portion of the execution needs to be monitored (generally only the instruction that causes a buffer overflow), PIN instrumentation for such monitoring is of negligible overhead; only a handful of extra instructions are inserted, and only in that one location.

Checkpointing Another task performed during normal execution is checkpointing. We modify the Rx [40] checkpoint and rollback system. Checkpoints are taken using a `fork()`-like operation, which copies all process state (e.g. registers and file descriptors) and uses copy-on-write to duplicate modified memory pages. The use of in memory checkpoints is feasible since we keep them for a short time (a few minutes at most) and then discard them. The advantage is much lower overhead than present in systems that write checkpoints to disk.

Similar to Rx [40], a checkpoint is captured using a shadow process. This provides a unique advantage for security purpose because a shadow process has a separate address space from the monitored process and *is entirely invisible at the user level*, even though some of their virtual pages may point to the same physical pages due to copy-on-write. Since we assume that the operating system is secure, an attack which corrupts the monitored process is unlikely to affect any checkpoint state; the first update to any page in the monitored process after a checkpoint will trigger the operating system's copy-on-write engine to copy the old page to a different location.

Rollback is also straightforward: reinstate the stored state back to the process. This is nearly instantaneous as it is almost identical to a context switch. File state can be handled similarly to previous work [29, 46] by keeping a copy of accessed files and file pointers at the beginning of a checkpoint interval. Network state is logged by a separate proxy process; this proxy facilitates replaying messages for re-execution and also implements signature-based input filtering. The re-execution runs faster than the original, due to lower IO costs (that is, there are no network delays or disk cache misses). More details can be found in the Rx paper [40].

Recovery As mentioned, the identification of the original input responsible for the attack also allows us to do fast recovery: we simply rollback the process and re-execute without the malicious input, and thus bring the process back to a safe state. In our system, rollback is accomplished by reverting to a previously saved system checkpoint. We then restart the system and replay legitimate (non-malicious) requests received after the checkpoint. We further discuss issues related to recovery of stateful services in Section 4.

3.2 Exploit Analysis

After the lightweight monitors have triggered, Sweeper performs a more thorough analysis of the attack. We use a variety of static and dynamic analysis tools, including static core dump analysis, memory bug detection, dynamic taint tracking, and dynamic backward slicing.

Core dump analysis By looking at the state of the program at the time when the lightweight monitor detects an attack, we can learn some things about the attack. This tool checks the consistency of the heap data structures, walks the stack to check for consistency, and determines the faulting instruction. This step is very fast (a few milliseconds), and can provide an initial VSEF. The disadvantage is that, given only a static glimpse of the program, we cannot achieve highly precise results. It is possible that an exploit may trigger the monitors and leave memory in a seemingly consistent state. Hence, we must still use more powerful tools later. For straightforward attacks (e.g. a stack buffer overflow) this step is sufficient to create a VSEF targeting the exact buffer overflow. If the attack is a stack-smashing attack, and it is detected at the time of the `ret` instruction, a VSEF to add stack canaries to that function can be generated. Although a more precise VSEF would be desired (target the overflow directly), this initial analysis is available almost immediately. Furthermore, anything detected in this stage, useful for a VSEF or not, is a potential starting point for dynamic backward slicing.

Memory bug detection Memory bug detection is a important step for vulnerability analysis because memory bugs, such as heap overflows or stack smashing, are commonly exploited for security attacks [20]. Detecting the misbehaving memory instruction usually gives an important clue to find the exploited instruction. Furthermore, detecting a memory bug gives a straightforward VSEF: insert the checks necessary to catch that particular bug.

There are many existing powerful memory bug detection tools commonly used by experienced programmers during debugging. They are usually not used in production runs due to the huge overhead (up to 100X slowdowns [58]). Fortunately, in Sweeper, such tools are *dynamically plugged in during replay* after an attack is detected, when overhead is less of a concern and can also be minimized due to the focused monitoring period. Operationally, we *dynamically* attach the memory bug detectors during sandboxed replay. In the short period of replay from the previous checkpoint, memory operations are monitored and many types of memory bugs throughout this period can be caught.

Specifically, Sweeper detects three important types of memory bugs, all of which are serious security vulnerabilities. The first is *stack smashing*. The memory bug detector records the stack return address location at every function entry and monitors this location for writes. Pre-existing stack frames are inferred from the stack frame base pointer register (*ebp*). The second memory misbehavior we detect is

heap overflow. Sweeper uses a modified red-zone technique that is simple and reasonably efficient—use `malloc()`'s own inline data structures. We monitor these areas for invalid access (e.g., not by `malloc()` or `free()`). Buffers allocated prior to the checkpoint are inferred from the memory image at the checkpoint. This technique has the advantage, over many existing techniques, that it can begin mid-execution. For the third type of memory bug—*double free*, all `malloc()` and `free()` calls are monitored to catch any `free()` calls to a previously freed location.

With the above described memory bug detection, Sweeper can generate efficient and accurate vulnerability monitor predicates, and use them to guard the application from future exploits. Specifically, bounds checking inserted at the effected instruction(s), or monitoring for double-frees at that particular free, can catch future exploit attempts. This monitoring is much more efficient than full memory bug detection, since it only involves a few code locations.

Dynamic Taint Analysis As we demonstrated in our previous work [36], dynamic taint analysis is a powerful means of detecting a wide range of exploits, including buffer overrun, format string, and double free attacks, some of which may be missed by the aforementioned memory bug detection. For Sweeper, we have reimplemented TaintCheck using PIN, so that it can be inserted after an exploit is detected.

TaintCheck tracks the flow of "taint" throughout a program: data read from untrusted sources are tainted, and the taint is maintained through data movement and arithmetic operations. Further, TaintCheck verifies that tainted data is not used in a sensitive manner, e.g. as a return address or as a function pointer. If tainted data is used in such a way, we can trace back to the responsible input, identifying the instructions that passed it along the way. For more details, please see [36].

Dynamic Backward Slicing A backward program slice is the set of instructions that effected the execution of a particular instruction [53]. That is, for a specific instruction, the backward slice is the set of dynamic instructions which were necessary for the instruction to execute. Instructions not in the slice are therefore *irrelevant*: if they were skipped, the execution of the selected instruction would not be influenced. This is similar to dynamic taint tracking, however *all* influences, including control flow and pointer indirection, are tracked. Consider the following code:

```
j=read(taint);
if(w==0)
  x=y[i];
else
  x=y[j];
z=x;
```

Suppose `w` were 3. In a backward slice from `z=x`, we would find a dependence on `x=y[j]`, `if(w==0)`, and `j=read(taint)`. We would also find a dependence on whichever instructions assigned to `y[j]` and `w` last. Dynamic taint analysis would not notice the dependence on `j` or `w`, and hence not identify that `z` is tainted.

We implement dynamic backward slicing in a way similar to [57]. We track the last dynamic instruction to write to each register and memory location, as well as the last to modify the control flags. The PC depends on the last conditional or indirect jump. Instructions, in turn, depend on any registers they read, any memory they read, and the PC. We construct a dependency tree from these relations; generating a backward slice from this tree is as simple as walking backward from the selected instruction.

Dynamic backward slicing gives similar (but more thorough) results as dynamic taint analysis, however it is much much more expensive: our implementation imposes *100x to 1000x* overhead. Only because this analysis is performed *only when necessary* is it at all practical. This again shows the benefits of deferring analysis until *after* an attack is detected.

It is also possible to compute a forward slice: the set of all instructions influenced by a starting instruction. A forward slice from the exploit input would reveal all instructions and memory potentially tainted by it. The dependence tree we generate can compute such a slice; currently we do not do so.

3.3 Antibodies

Sweeper's antibodies provide protection against further attacks. They can either be input signatures, or vulnerability specific execution filters (VSEFs).

Input Signatures Input filters are commonly used to eliminate known exploits before they reach vulnerable servers [24, 26, 36, 35, 44]. Based on the input that caused the exploit (derivable from either dynamic taint analysis or backward slicing), many existing techniques can be used to generate filters. Since Sweeper has VSEFs to provide a safety net, we can start by generating signatures as exact matches. This has the benefit of very low false positives, and being impervious to malicious training [34]. Polymorphic signatures are also feasible; see our work [7] for details.

VSEFs Vulnerability specific execution filters, or VSEFs [33], provide a low false-negative approach to detecting attacks. VSEFs in Sweeper function like the heavyweight dynamic analysis tools, except that *they only monitor the instructions necessary to detect the exploit*. Since the number of instructions monitored is much smaller, they are no longer heavy-weight but are light-weight. VSEF-hardened binaries are able to reliably detect various attacks against the same vulnerability, even in the face of poly- and meta-morphism. Since they look for the same behavior as the heavyweight dynamic analysis, they have similar false negative and false positive properties. Sweeper considers VSEFs derived both from memory bug detectors and from dynamic taint analysis.

Memory-bug derived VSEFs consist of the instruction responsible for the memory bug, and the type of the bug. For a buffer overflow, this is the `store` instruction which overflows the buffer. For a double-free, this is the call to `free()` which is redundant. In both cases, the implementation of the VSEF is to monitor for the type of bug at that location: is the write within bounds, or is the buffer to be freed already free? In the case of stack overflows, this may be relaxed to simply ensure that a return address is not being overwritten, if information about the stack layout is not available. The static memory analysis may generate another sort of memory VSEF: monitor the return address of one particular function. The `call` who's return address is overwritten is recorded in the usual place, and also copied separately. Just prior to the `ret` call that pops the return address, the stored value is compared to the stack's value. This is simpler than using canaries because the structure of the stack can

remain the same. All of these memory bug derived VSEFs only insert a handful of instrumentation instructions, and therefore impose negligible overhead.

Dynamic taint analysis VSEFs consist of a list of instructions that propagated the taint, and the instruction which incorrectly consumed tainted data. Ordinary dynamic taint analysis instrumentation is applied *for those instructions only*. Again, this imposes much less overhead than full analysis. For more details, please refer to our paper on VSEFs [33].

Distribution The generated anti-bodies can be disseminated to other hosts to protect them against further attacks. The concrete manifestation of an antibody to be disseminated is a set of VSEFs and an exploit-triggering input. Together, these allow hosts to protect themselves in multiple ways. Including the exploit-triggering input allows hosts to verify the antibodies: in a sandbox, feed the input to the vulnerable program while performing heavy-weight analysis.

Since receiving and applying VSEFs is a time-critical operation, hosts may want to apply them without verifying them first. By deferring verification, hosts reduce their exposure to infection. A VSEF is a set of instruction addresses that need to have certain monitoring (e.g. buffer overflow monitoring, dynamic taint analysis, etc.). By their nature, then, VSEFs cannot be harmful; incorrect or malicious VSEFs will result in unnecessary bounds checking or taint tracking, but cannot create behaviors that full monitoring would not. At worst they cause a performance degradation. Unneeded VSEFs can be removed when they are verified. Since verification is deferred, we distributed antibodies piecemeal. As each step completes, a host will distribute results *as it generates them*. Similarly, hosts consuming antibodies apply them *as they receive them*, deferring verification until after the exploit input is isolated.

4. ISSUES AND DISCUSSION

4.1 Recovery and Re-Execution

The Rx-based re-execution allows recovery in many practical cases. However, there may be instances where dropping the attacking requests and re-executing is not sufficient to maintain consistency. Consider, for example, an SSL-enabled web server. Session keys depend on random numbers; for connections concurrent to the attack these numbers may turn out different on re-execution. An alternative to Rx is to use a Flashback [46] based checkpointing system. Flashback *logs* all of the system calls made by the process, in order to allow *deterministic* re-execution. For Sweeper, this allows us to either re-execute the application with more consistency or, failing that, to detect the inconsistency and abort. If the execution depends on a system call returning the same result (e.g. a `read()` to a file, or a call to `gettimeofday()`), Flashback *will* replay the same result as previous executions. Therefore, differences in the results of system calls will not perturb the execution. To verify the consistency of results, Sweeper can compare the re-execution's calls to `write()` to the previous results Flashback recorded; if they match, we know that we have been successful. In the case that the lack of the attack has caused a change in program state (e.g., a counter of the number of connections accepted) that changes the output, we can abort the re-execution and resort to restart. It is our practical experience with Rx that this is a rare case, however, for those instances where the execution is sensitive to small changes, this alternative exists.

A further issue is reliance on other, non-checkpointed programs, or the possibility that the operating system itself becomes compromised *prior* to the lightweight monitoring tripping. In both cases we would be unable to apply a correct rollback and re-execution. To prevent this, we could apply the same checkpointing techniques to the whole OS through a virtual machine (e.g. as done in Time Traveling Virtual Machines [25]). This allows rollback of an entire software stack, including the OS, any helper applications, and even disk state. Although we feel that the OS is unlikely to be corrupted prior to the lightweight monitoring registering an attack, we can certainly feel safe that the VM hypervisor will not become corrupted by a network-based attack on one of its guests.

4.2 Sampling to Catch More Attacks

In order to deal with a broader range of attacks, Sweeper can use more expensive monitoring to analyze a fraction of requests. Although many security attacks involve memory corruption attacks that can be noticed by lightweight bug detectors, those which are not can be caught through sampling and analysis with heavy-weight detection mechanisms. Since the instrumentation is dynamic, the decision to more thoroughly analyze a message can be made at runtime. It would even be feasible for hosts to use heavier-weight detection when they are idle, and shift to address space randomization as they become fully loaded.

4.3 Effects of Limited Deployment

Although Sweeper has very low overhead, widespread deployment does not necessarily mean 100%; it is unlikely to reach such high levels. Sweeper does not require universal deployment to function. Hosts may choose to act as consumers of antibodies; the lightweight monitoring will still make them more difficult to exploit. There will be, however, a chance that such hosts will become infected, since multiple infection attempts are likely to be made before an antibody is available. If deployment rates are too low, the worm is too fast, and the antibodies are too slow to be delivered, Sweeper will be unable to contain the worm. Compared to previous systems, however, failure comes in more extreme conditions. Section 6 discusses in much greater detail the performance of Sweeper as a whole under varying conditions.

5. EXPERIMENTAL RESULTS

5.1 Experiment Setup

Implementation We implemented Sweeper in Linux by modifying the Linux kernel 2.4.22 to support lightweight checkpoint and rollback-and-replay. The various monitoring and analysis techniques are implemented using the PIN binary instrumentation tool [30]. All of the tools are integrated together except for taint analysis; it is implemented stand alone and we are in the processes of integration. Hence we provide functionality results but not performance numbers for taint analysis. In lieu of taint analysis performance, we measure the time to isolate the exploit input by sending

Name	Program Description	CVE ID [50]	Bug Type	Security Threat Description
Apache1	Apache-1.3.27 web server	CVE-2003-0542	Stack Smashing	Local exploitable vulnerability enables unauthorized access
Apache2	Apache-1.3.12 web server	CVE-2003-1054	NULL Pointer	Remotely exploitable vulnerability allows disruption of service
CVS	cvs-1.11.4 version control server	CVE-2003-0015	Double Free	Remotely exploitable vulnerability provides unauthorized access and disruption of service
Squid	squid-2.3 proxy cache server	CVE-2002-0068	Heap Buffer Overflow	Remotely exploitable vulnerability provides unauthorized access and disruption of service

Table 1: List of tested exploits

the potentially suspicious requests one at a time. Both provide the exploit input as a result, but we expect taint analysis to be faster, based on our experience with Valgrind-based TaintCheck.

Experiment Environment and Parameters Our experiments are conducted on single-processor machines with a 2.4GHz Pentium 4 processor. By default, Sweeper keeps the 20 most recent checkpoints, and checkpoints every 200ms.

Evaluation Applications We evaluate Sweeper on four *real* vulnerabilities in three server applications, as shown in Table 1. All of the vulnerabilities are recorded by US-CERT/NIST [50].

Experimental Design In our experiments, we evaluate the functionality of Sweeper, as well as the efficiency of exploit and vulnerability analysis. We also show the normal overhead of checkpointing for various checkpoint intervals.

5.2 Functionality Evaluation

Table 2 presents a summary of Sweeper's functionality results for four exploits. The second column summarizes the results: for all four exploits, Sweeper detects the attack, generates a VSEF, and identifies the original input that triggered the fault.

The detailed results columns show what each of the analysis steps determines. The first step, memory state analysis, looks at the stack, heap, and instruction pointer at the time the lightweight monitoring trips. For all four vulnerabilities, this results in a VSEF; for the Apache2 and Squid bugs this VSEF ends up being the final "best" VSEF. The second step, memory bug detection, identifies various memory bugs through dynamic instrumentation. For the Apache1 and CVS exploits this step provides a more specific VSEF. Consider specifically the Apache1 VSEFs. The initial VSEF only protects the return address. For this exploit, this is sufficient. However, the specific buffer overflow *may* also be exploitable by overwriting a stack function pointer[2]; the initial VSEF won't catch this. The improved VSEF identifies more exactly the underlying software flaw the resulted in the vulnerability: "stack buffer overflow". The initial VSEF captures a subset vulnerability: "overwrite return address". However, the initial VSEF will still catch all instances of this *exploit*, and all exploits that use the specific sub-vulnerability; hence it will still stop the worm outbreak.

The third step is input/taint analysis—the purpose is to identify the input responsible so that it can be fed to a signature generator. This is done successfully for all four vulnerabilities. For the Apache1 bug, however, the input is configuration specific. This makes it difficult to share the result with other hosts, but it also makes it difficult to exploit. Finally, dynamic slicing is performed. It serves as a sanity check on the other stages; if a previous stage claims an instruction or data value is involved in the attack and dynamic slicing disagrees, then the previous step is incorrect. In all four cases, however, dynamic slicing is consistent with the other analysis steps.

These results demonstrate that Sweeper is capable of defending against a variety of vulnerabilities: a stack overflow, a null pointer dereference, a double free, and a heap buffer overflow. In all four cases, Sweeper generates a VSEF and identifies the exploit input.

[2]To the best of our knowledge, this particular buffer overflow does *not* have such multiple methods of exploitation.

5.3 Performance Evaluation

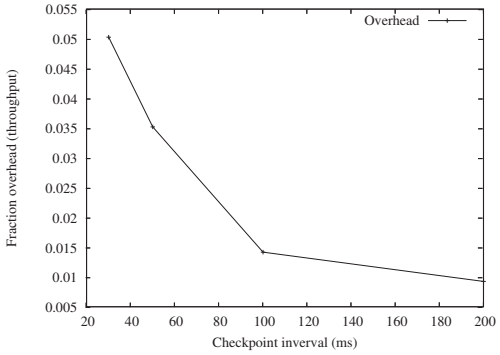

Figure 4: Performance at varying checkpoint intervals for Squid

Checkpointing Since Sweeper is intended for widespread deployment, overhead is an important concern. As demonstrated by Figure 4, the performance overhead of checkpointing and network logging is low; at a 200ms checkpoint interval, Sweeper only degrades performance by *.925%* — throughput drops from 93.45 Mbps to 92.59Mbps. The fastest checkpoint interval, 30 ms, only shows a 5% performance degradation. These results clearly demonstrate that the checkpoint overhead is nominal, and suitable for production run deployment. More detailed discussion of the performance of checkpointing can be found in our previous Rx paper [40].

Vulnerability Monitoring Sweeper's VSEFs only check a small subset of instructions; hence they have good performance properties. *It is not necessary to bounds check the entire program, but only the one vulnerable callsite.* For Squid,

App.	Defense Result Summary	Detailed Processes and Results		
		Step	Technique	Main Results From Each Step
Apache1	Correct buggy instruction and memory location	#1	Memory State Analysis	Crash at 0x805e33f (*try_alias_list*); stack inconsistent VSEF: use a side stack for (*try_alias_list*)
	Correct VSEFs	#2	Memory Bug Detection	Stack smashing by 0x808c3ee (*lmatcher*) VSEF: 0x808c3ee should not overflow stack buffer
		#3	Input/Taint Analysis	*GET.../trigger/crash.html...*
	Configuration-specific input	#4	Slicing	Verifies results
Apache2	NULL pointer dereference correctly identified	#1	Memory State Analysis	Crash at 0x8060029 (*is_ip*); accessing NULL pointer VSEF: check for NULL pointer
	Correct VSEFs	#2	Memory Bug Detection	No memory bug detected, just a NULL pointer dereference
		#3	Input/Taint Analysis	** Referrer: (ftp://|http:///){0}? **
	Finds input	#4	Slicing	Verifies results
CVS	Correct buggy instruction and memory location	#1	Memory State Analysis	Crash at 0x4f0eaaa0 (lib. *free*); heap inconsistent VSEF: Check for double frees
	Correct VSEFs	#2	Memory Bug Detection	Double free by 0x808d7ac (*dirswitch*) VSEF: 0x808d7ac should not double-free
		#3	Input/Taint Analysis	*/CVS request stream/*
	Finds input	#4	Slicing	Verifies results
Squid	Correct buggy instruction and memory location	#1	Memory State Analysis	Crash at 0x4f0f0907 (lib. *strcat*); heap inconsistent VSEF: Heap bounds-check 0x4f0f0907 (in lib. *strcat*) when called by 0x804ee82 (*ftpBuildTitleUrl*)
	Correct VSEFs	#2	Memory Bug Detection	Heap buffer overflow at 0x4f0f0907 (lib. *strcat*) VSEF: Verified above
		#3	Input/Taint Analysis	*ftp://\\...\\@ftp.site*
	Finds input	#4	Slicing	Verifies results

Table 2: Overall Sweeper results

Application	Time to First VSEF	Time to Best VSEF	Initial Analysis Time	Total Analysis Time	Component Diagnosis Time			
					Memory State Analysis	Memory Bug Detection	Input/Taint Analysis	Dynamic Slicing
Apache1	60 ms	14 sec	24 sec	68 sec	0.06 sec	14 sec	9 sec	45 sec
Squid	40 ms	40 ms	38 sec	145 sec	0.04 sec	30 sec	7 sec	108 sec

Table 3: Sweeper failure analysis time. The component diagnosis times are the times for each individual component; the other time values are cumulative from the lightweight monitoring triggering. After the time to first VSEF, we can begin spreading an antibody. Initial time is the time it takes to generate both VSEFs and isolate the exploit's input; total time includes the slicing step.

the VSEF checks for a heap buffer overflow at 0x4f0f0907 (in `strcat`), and then only when `strcat` is called by 0x804ee82 (in `ftpBuild-TitleUrl`). This results in a .93% drop in throughput (91.6 Mbps vs. 92.5Mbps). Much of the overhead comes from monitoring calls to `malloc` and `free` to get the exact ranges of live buffers; if a second heap buffer overflow was identified, the combined overhead would increase less. In the worst case, overhead is linear with the number of vulnerabilities; systems running software with many *unpatched* vulnerabilities that have wild exploits will experience higher overheads. Users who wish to avoid such overhead can do so by applying patches as they become available. Again, the overheads are clearly suitable for production run deployment.

Analysis Times Sweeper can generate VSEFs very quickly: 60 ms for Apache and 40 ms for Squid. As we show in Section 6, fast antibody generation is important for dealing with the fastest of worms; 60 ms is more than fast enough. Table 3 shows the details of our analysis performance. For both measured applications, the time to get the "best" VSEF was under 15 seconds; in Squid's case the initial result was the best. The time to get the VSEFs and to isolate the input responsible is under 40 seconds.

Although the complete analysis results are not available immediately, the intermediate results (i.e., initial VSEFs) are sufficient to use for antibodies because they do not have false positives even though they may have a higher false negatives. Waiting for the full analysis to complete is inadvisable, because the further delay will allow a fast worm to spread. Instead, antibodies should be distributed as soon as they are available (e.g., within 60 ms). The initial VSEF is more than sufficient to stop the particular exploit being used (and will catch poly- and meta-morphic variants); because it is available sooner, it is *best for this worm outbreak*. The improved VSEF can be distributed as a follow-on.

Recovery Once VSEFs are applied, we perform recovery. Figure 5 shows the throughput as a function of time. Approximately 24 seconds in, the throughput drops due to recovery taking place; no requests complete service during this time, and clients perceive increased latency. Shortly thereafter, service resumes as normal. In contrast, a restart of Squid takes over 5 seconds, and clients perceive dropped connections and refused connection attempts.

6. COMMUNITY DEFENSE AGAINST FAST-SPREADING WORMS

As we have shown, Sweeper protects individual hosts even from fast-spreading worm that exploits previously unknown vulnerabilities, *i.e.*, zero-day hit-list worms. The first time that such a worm tries to infect a Sweeper-protected host,

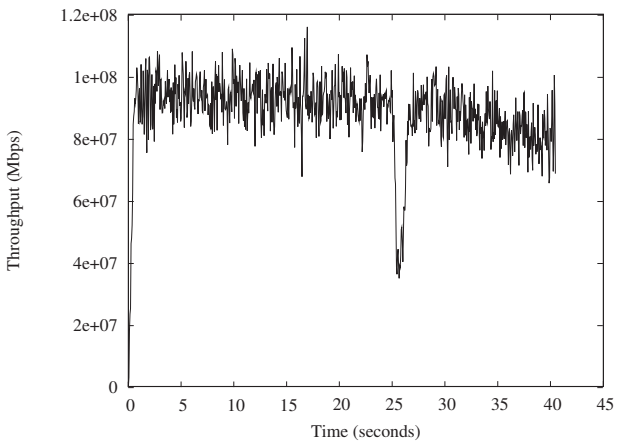

Figure 5: Throughput during a single attack against Squid

the exploit will be detected, analyzed, and one or more antibodies deployed to prevent further attacks against that vulnerability.

We next show how a Sweeper *community* can protect even those who do not deploy Sweeper from new exploit attacks, including fast-spreading worms. In this community, we call those who deploy the complete Sweeper system *Producers*. When a Producer detects a new attack and generates the corresponding antibodies, it shares those antibodies with *Consumers*, thus preventing them from becoming infected. Given the low ($> 5\%$) overhead involved in being a producer, we would expect that the percentage of Producers to be high; here we present Producer deployment ratios far below our expectations.

The challenge is to generate antibodies and distribute them to the Consumers before they are infected. We use worm modeling techniques to show that most Consumers can be protected from even the fastest observed worms. Further, we show that if Consumers deploy light-weight *proactive* defense mechanisms, we can protect most Consumers from even hit-list worms.

6.1 Community Model

Worm propagation can be well described with the classic Susceptible-Infected (SI) epidemic model [23]. 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 that are Producers, and the remaining vulnerable population $(1-\alpha)$ be Consumers. Let $P(t)$ be the total number of producers contacted by at least one infection attempt at time t.

From the SI model, we have:

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

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

We call the time at which at least one Producer has received an infection attempt, and hence can begin generating and distributing antibodies, T_0. By this definition, $P(T_0) = 1$. We can solve the above equation to find T_0.

Once a Producer is contacted with an infection attempt, it takes time γ_1 until the producer creates an antibody using exploit analysis, and then it takes time γ_2 until the antibody can be disseminated to Consumers (and if needed, verified). Let $\gamma = \gamma_1 + \gamma_2$, and we call γ the response time of the Sweeper community. Thus, after time $T_0 + \gamma$, all the vulnerable hosts have received and installed the antibody and become immune to the worm outbreak. Thus, the total number of infected hosts throughout the worm outbreak is $I(T_0 + \gamma)$, and $I(T_0 + \gamma)/N$ is the infection ratio.

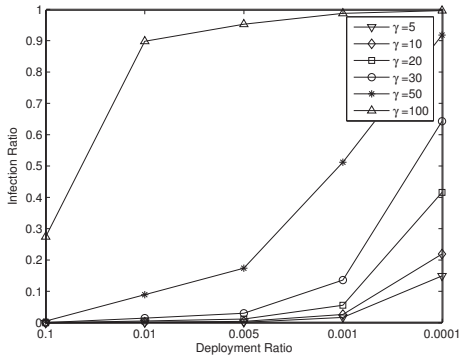

Figure 6: Sweeper defense against Slammer ($\beta = 0.1$)

6.2 Protection Against Slammer

The fastest-spreading worm to date is Slammer. In the Slammer worm outbreak, the contact rate β was 0.1, and the number of vulnerable hosts N was approximately 100000 [10].

Figure 6 shows that a Sweeper community could have prevented the Slammer worm from infecting most vulnerable hosts, for a variety of producer ratios α and response times γ. For example, given a very low deployment ratio $\alpha = 0.0001$, and a reasonable response time $\gamma = 5$ seconds, the overall infection ratio is only 15%. For a slightly higher producer ratio $\alpha = 0.001$, the Sweeper community is even more effective, protecting all but 5% of the vulnerable hosts even for a relatively slow response time of $\gamma = 20$ seconds.

6.3 Protection Against Hit-List Worms

A well designed worm could propagate much more quickly than Slammer. In particular, a *hit-list* worm contains a hit-list of vulnerable machines. Hit-list worms can spread up to orders of magnitude more quickly because they need not scan to find vulnerable hosts [47, 48].

If Slammer had been designed as a hit-list worm, it may have achieved a contact rate of $\beta = 1000$, or even $\beta = 4000$; this is *ten-thousand* to *forty-thousand* times faster than observed. In our model, this would result in 100% of vulnerable hosts becoming infected in mere hundredths of a second. Even if the very first infection attempt was against a Producer (i.e., $T_0 = 0$), this does not provide enough time to produce, distribute, and verify antibodies.

Proactive protection We can protect against even hit-list worms if we combine our reactive strategy of producing and distributing antibodies with a *proactive* strategy to slow down the spread of the worm [6].

For example, for a large class of attacks, address space randomization can provide probabilistic proactive protection. The attack, with high probability, will crash the vulnerable program instead of successfully compromising it. However, because the protection is only probabilistic, repeated or brute-force attacks will succeed; the attacker will eventually "guess" the address space layout and successfully infect the host.

Let ρ be the probability that a particular infection attempt successfully exploits a host with probabilistic protection. We model the spread of a hit-list worm where vulnerable hosts use proactive protection with:

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

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

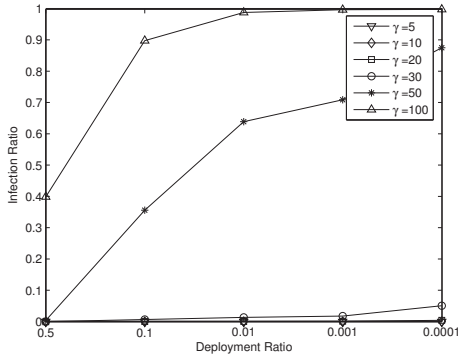

Figure 7: Sweeper with proactive protection against hit-list ($\beta = 1000$). Note that $\gamma = 50$ is much worse than $\gamma = 30$.

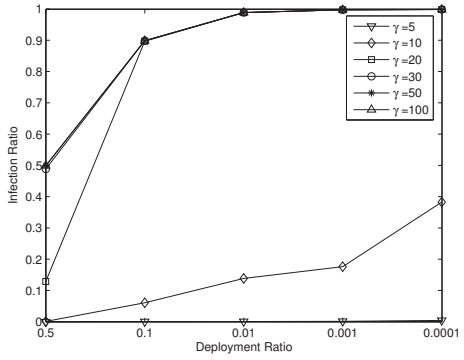

Figure 8: Sweeper with proactive protection against hit-list ($\beta = 4000$). Note that $\gamma = 20$ is much worse than $\gamma = 10$.

We show that Sweeper combined with proactive protection can protect against even hit-list worms with contact rate $\beta = 1000$ in Figure 7, and with contact rate $\beta = 4000$ in Figure 8. Here, we set the probability that an infection attempt succeeds to $\rho = 2^{-12}$, which many address randomizations achieve [42]. We again use $N = 100000$ vulnerable hosts. For example, the figures indicate that given deployment rate $\alpha = 0.0001$ and reaction time $\gamma = 10$ seconds, the overall infection ratio is only 5% for $\beta = 1000$ and 40% for $\beta = 4000$. For $\alpha = 0.0001$ and $\gamma = 5$ seconds, the overall infection ratio is negligible (less than 1%) for both cases. Note the large differences in infection ratio as γ increases: for $\gamma = 50$ in the $\beta = 1000$ case and $\gamma = 20$ in the $\beta = 4000$ case the worm would still infect large fractions of all vulnerable hosts. Hence, even with the proactive protection, we *do* still require an automated defense such as Sweeper.

Our models show that a total end-to-end time (including time for detection, analysis, and antibody dissemination/ deployment) of about 5 seconds will stop a hit-list worm. Note that our experiments (Section 5.1) show that detection and analysis are almost instantaneous, and the total time it takes to create an effective VSEF is well under 2 seconds. Vigilante shows that the initial dissemination of an alert could take less than 3 seconds [14]. Thus our system achieves an $\gamma = 2 + 3 = 5$. By impeding the spread of the worm, our system can effectively defend against effectively defend against even hit-list worms that are thousands of times faster than the fastest observed worm, even for low values of α.

7. RELATED WORK

7.1 Checkpoint and Rollback

While Sweeper leverages a lightweight checkpoint and re-execution support similar to FlashBack or Rx [40, 46], it could use other checkpoint systems like the Time-traveling Virtual Machines [25], or ReVirt [18]. ReVirt also deals in a security setting: specifically, postmortem analysis. However, ReVirt is intended as an offline forensic tool, and does not target on-line systems.

7.2 Bug detection and analysis

Sweeper makes use of various bug detection techniques both to detect the initial exploit attempt and to analyze the exploit attempt after rollback. In general, the more useful the analysis results, the more expensive the tool is to run, and therefore less suitable for use as a lightweight detector.

Sweeper's baseline bug detection method, address space randomization [20], provides an almost free detection mechanism, however, it can be probabilistically bypassed [42]. This is only a minor concern in Sweeper, since for hosts deploying the full system, capturing an attack once is sufficient. Slightly less lightweight monitors like SafeMem [39] may also be used widely. Other monitors, such as StackGuard [15], CCured [13, 31], Purify [22], or Valgrind [32] are options, trading runtime overhead for greater protection. A recent work in dynamic binary instrumentation, LIFT [38], reduces the overhead for information flow tracking to potentially manageable levels (2-4x overhead); this may be deployable for a decent fraction of hosts.

Bug detectors are also used for analysis in Sweeper. The favored techniques are dynamic backward slicing [53, 57] and dynamic taint analysis [36]. Both track the influence of one instruction on another; backward slicing works back from the point of exploit, while taint analysis works forward from inputs. Another technique is memory bug detection, such as found in CCured [13], Purify [22], or Valgrind [32]. In general, such techniques impose higher overheads than address space randomization, but give more precise results. Other possible techniques are similar to those used in DIRA [45],

STEM [43], or DACODA [16]. STEM emulates execution; this allows recovery by rolling back and abandoning changes (effectively cutting off the vulnerable functionality). DIRA logs memory state changes for similar ends. DACODA uses symbolic execution to determine the entire set of inputs capable of triggering an exploit. None of these are feasible for stand-alone use due to high overhead. However, with the ability to apply analysis *after* an exploit attempt is detected, Sweeper could potentially use them.

7.3 Attack Response

A considerable amount of research effort [14, 24, 26, 36, 35, 44] has been devoted to automatically generating attack signatures. Earlybird [44], Honeycomb [26] and Autograph [24], share a common limitation: the signatures generated are single, contiguous strings. Real life attacks can often evade such filters. To tackle such polymorphic worms, techniques like Polygraph [35] generate signatures that consist of multiple disjoint content substrings. However, recent work [34, 37] shows that such polymorphic signature generators can be mislead into generating bad signatures; specifically higher false negative rates. Sweeper does not rely only on input signatures for protection; while signatures provide a guarantee of correctness (since exploit attempts caught by the signatures never touch the application), a safety net of VSEFs allows false negatives to be tolerated. Furthermore, the availability of a highly selective mechanism (i.e. low false positives and low false negatives) allows the input signatures to be "tuned" toward lower false positives at the expense of false negatives.

7.4 Automated Worm Defense

Vigilante [14] is a nice automatic worm defense similar to Sweeper. A subset of nodes monitor their execution with full dynamic taint analysis (or, nodes may sample requests). When an exploit is detected, Vigilante creates a self-verifying signature to distribute to all nodes. There are several important technical differences between Sweeper and Vigilante. First, Sweeper provides a recovery mechanism through rollback and modified re-execution. Second, Vigilante provides no means to combine light-weight and heavy-weight detectors. Therefore, Vigilante either must sample requests or be deployed only on a subset of honey-pot hosts. Hosts that are sampling only have a small chance to analyze an exploit attempt, while honey-pot nodes are vulnerable to being avoided. In combining light- and heavy-weight detectors, Sweeper provides more flexibility, can be more widely deployed, and increases the number of exploit attempts that will be monitored. Third, the two systems generate and distributed different sorts of antibodies. Finally, reactive antibody systems, like Vigilante, can not distribute their antibodies fast enough to deal with a hit-list worm. The additional layer of defense that Sweeper provides with its lightweight monitors provides sufficient robustness to react against extremely fast hit-list worms.

Liang and Sekar [28] and Xu et al. [56] 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 programs where static binary analysis is difficult, and their signature generation does not work in many cases (for example, if the inputs are processed or decoded prior to causing a buffer overflow). The analysis in Xu et.al.'s approach is also limited, and their signatures suffer from similar problems as described in [16]. Additionally, these approaches rely only on address space randomization, which can be bypassed; our approach has the flexibility to allow various light- and heavy-weight detectors to be plugged in, as per an individual host's requirements.

8. CONCLUSIONS

We presented an innovative approach for defending against exploits. By leveraging checkpointing and replay, we allow continuous lightweight monitoring to be combined with heavy-weight analysis. The resulting system has low overhead (1%) during normal execution, which allows more widespread deployment than similar systems. Further, the analysis is used to generate multiple forms of antibodies, which are available starting at 60 ms from the signs of attack.

We demonstrated an implementation of our approach in Sweeper. Against 4 real exploits in 3 different server applications, Sweeper generates effective antibodies quickly (no slower than 60 ms). We also provide analytical results demonstrating how effective Sweeper would be against a fast worm outbreak.

In Sweeper we have realized an architecture that protects applications with lightweight techniques while enabling more sophisticated techniques to perform accurate post-analysis. Sweeper also provides recovery against such attacks without access to source code. Finally, ours implementation is capable of generating sophisticated vulnerability-specific execution filters while maintaining performance at levels feasible for widespread deployment.

9. REFERENCES

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

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

[3] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *12th USENIX Security Symposium*, 2003.

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

[5] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault tolerance under UNIX. *ACM TOCS*, 7(1), Feb 1989.

[6] D. Brumley, L.-H. Liu, P. Poosankam, , and D. Song. Design space and analysis of worm defense strategies. In *Proceedings of the 2006 ACM Symposium on Information, Computer, and Communication Security (ASIACCS 2006)*, March 2006.

[7] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *IEEE Symposium on Security and Privacy*, 2006.

[8] CERT. Blaster `http://www.cert.org/advisories/CA-2003-20.html`.

[9] CERT. CodeRed `http://www.cert.org/advisories/CA-2001-19.html`.

[10] CERT. Slammer `http://www.cert.org/advisories/CA-2003-04.html`.

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

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

[13] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. CCured in the real world. In *PLDI*, 2003.
[14] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *SOSP'05*, 2005.
[15] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *the 7th USENIX Security Symposium*, 1998.
[16] J. R. Crandall, Z. Su, S. F. Wu, and F. T. Chong. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *CCS '05*, 2005.
[17] D. Dhurjati and V. Adve. Backwards-compatible array bounds checking for c with very low overhe ad. In *ICSE*, 2006.
[18] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtualmachine logging and replay. In *OSDI'02*, 2002.
[19] H. Etoh. GCC extension for protecting applications from stack-smashing attacks. http://www.trl.ibm.com/projects/security/ssp/.
[20] S. Forrest, A. Somayaji, and D. H. Ackley. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *HotOS*, 1997.
[21] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *Proceedings of 6th workshop on Hot Topics in Operating Systems*, 1997.
[22] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Usenix Winter Technical Conference*, 1992.
[23] H. W. Hethcote. The mathematics of infectious diseases. *SIAM Rev.*, 42(4):599–653, 2000.
[24] H.-A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *the 13th Usenix Security Symposium*, 2004.
[25] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX*, 2005.
[26] C. Kreibich and J. Crowcroft. Honeycomb: creating intrusion detection signatures using honeypots. *SIGCOMM Comput. Commun. Rev.*, 2004.
[27] R. Lemos. Counting the cost of the slammer worm. `http://news.com.com/2100-1001-982955.html`, 2003.
[28] Z. Liang and R. Sekar. Fast and automated generation of attack signatures: a basis for building self-protecting servers. In *CCS '05*, 2005.
[29] D. E. Lowell and P. M. Chen. Free transactions with Rio Vista. In *SOSP*, 1997.
[30] 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 *PLDI*, 2005.
[31] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *POPL*, 2002.
[32] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *RV*, 2003.
[33] J. Newsome, D. Brumley, and D. Song. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *NDSS*, 2006.
[34] J. Newsome, B. Karp, and D. Song. Paragraph: Thwarting signature learning by training maliciously. In *RAID*, Sept. 2006.
[35] J. Newsome, B. Karp, and D. X. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *IEEE Symposium on Security and Privacy*, 2005.
[36] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.
[37] R. Perdisci, D. Dagon, W. Lee, P. Fogla, and M. Sharif. Misleading worm signature generators using deliberate noise injection. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, May 2006.
[38] F. Qin, H. Chen, Z. Li, Y. Zhou, H. seop Kim, and Y. Wu. LIFT: A low-overhead practical information flow tracking system for detecting general security attacks. In *MICRO*, Dec 2006. To appear.
[39] F. Qin, S. Lu, and Y. Zhou. Safemem: Exploiting ECC-Memory for detecting memory leaks and memo ry corruption during production runs. In *HPCA*, 2005.
[40] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies—A safe method to survive software failures. In *SOSP*, 2005.
[41] D. Scott. Assessing the costs of application downtime, 1998.
[42] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *CCS*, 2004.
[43] S. Sidiroglou, M. Locasto, S. Boyd, and A. Keromytis. Building a reactive immune system for software services. In *USENIX*, 2005.
[44] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *OSDI'04*, 2004.
[45] A. Smirnov and T. cker Chiueh. Dira: Automatic detection, identification and repair of control-hijacking attacks. In *NDSS*, 2005.
[46] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX*, 2004.
[47] S. Staniford, D. Moore, V. Paxson, and N. Weaver. The top speed of flash worms, 2004.
[48] S. Staniford, V. Paxson, and N. Weaver. How to 0wn the internet in your spare time. In *USENIX Security Symposium*, 2002.
[49] T. K. Tsai and N. Singh. Libsafe: Transparent system-wide protection against buffer overflow attacks. In *DSN*, page 541, 2002.
[50] US-CERT. Common vulnerabilities and exposures.
[51] W. Vogels, D. Dumitriu, A. Agrawal, T. Chia, and K. Guo. Scalability of the Microsoft Cluster Service. In *USENIX Windows NT Symposium*, Aug 1998.
[52] W. Vogels, D. Dumitriu, K. Birman, R. Gamache, M. Massa, R. Short, J. Vert, J. Barrera, and J. Gray. The design and architecture of the Microsoft Cluster Service. In *FTCS*, Jun 1998.
[53] M. Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452, 1982.
[54] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *NDSS*, 2003.
[55] 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, May 2003.
[56] J. Xu, P. Ning, C. Kil, Y. Zhai, and C. Bookholt. Automatic diagnosis and response to memory corruption vulnerabilities. In *CCS '05*, 2005.
[57] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *ICSE*, 2003.
[58] P. Zhou, W. Liu, F. Long, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas. AccMon: Automatically detecting memory-related bugs via program counter-based invariants. In *MICRO*, 2004.